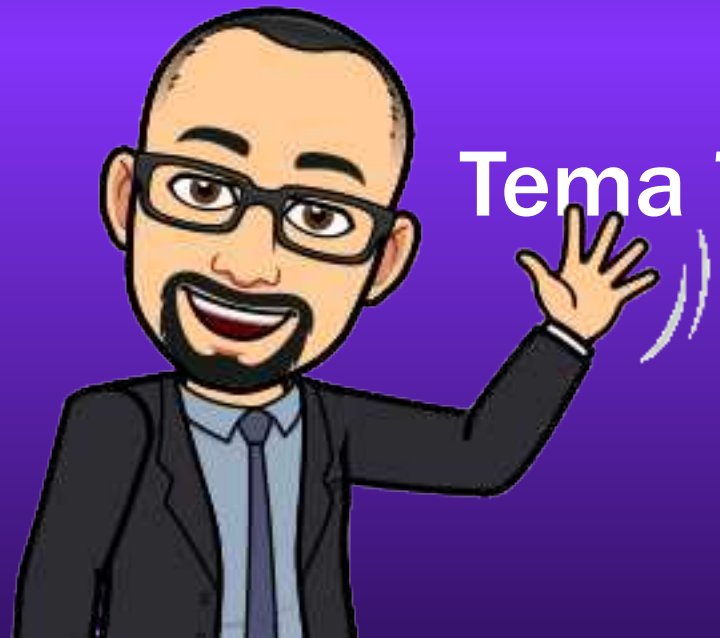
LENGUAJE C



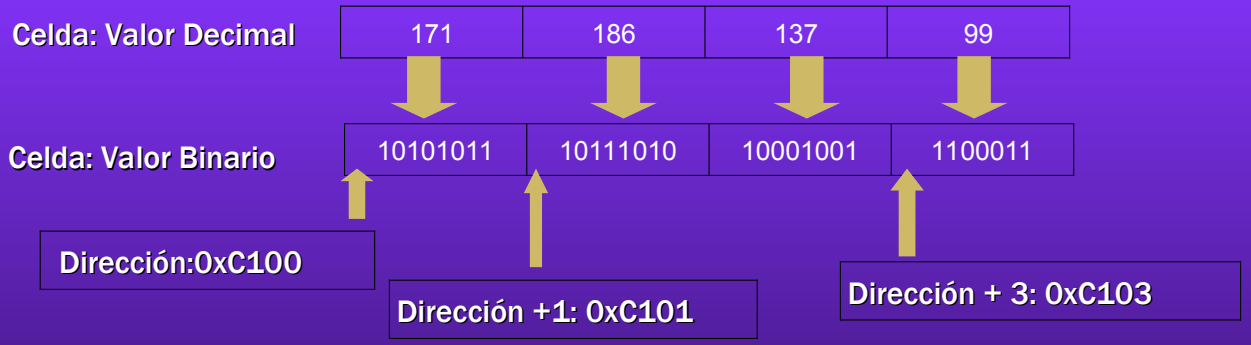
Tema 7 – Memoria Dinámica

LA MEMORIA

- La memoria es un conjunto de celdas contiguas donde se almacenan datos.

- La unidad de memoria más pequeña es el bit.
- El byte es un conjunto de 8 bits. Unidad de memoria.

- El lugar (ubicación) de cada byte es único y es su dirección.
- Si los bytes son consecutivos la dirección se ira incrementando secuencialmente.

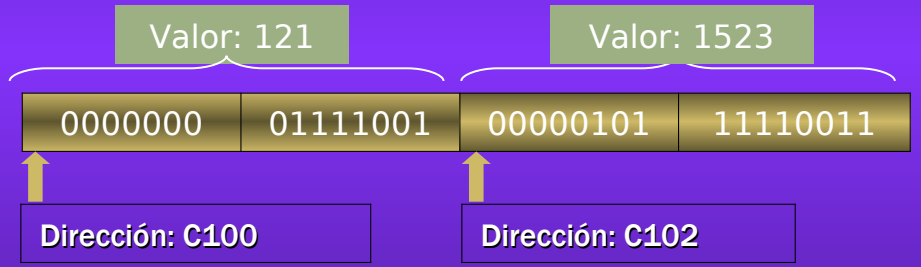


- Cada celda tiene dos valores asociados: Dirección y Contenido.

LAS VARIABLES

- Una variable es una porción de memoria identificada por un nombre.
- El tipo de dato define su representación binaria y longitud.
- Tiene tres valores asociados: Nombre, Contenido y Dirección.

- Declaramos: `int n=1523, m=121;`



- Contenido:
  - n ⇔ 1523
  - m ⇔ 121
- Dirección:
  - &n ⇔ C102
  - &m ⇔ C100

- Es contenido binario se codifica (bits) de acuerdo al tipo de variable.

MEMORIA DINÁMICA

Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.

En muchas ocasiones no podemos saber a priori el espacio que vamos a necesitar para nuestras estructuras de datos. Esto deberá decidirse en tiempo de ejecución y llevarse a cabo mediante funciones de gestión de memoria como malloc, calloc, u otras similares.

- La asignación dinámica de memoria es una técnica de programación muy valiosa.
- Tamaños apropiados de memorias pueden ser asignados desde la memoria libre disponible en el heap del sistema y retornada a este cuando ya no se necesite.
- La asignación y devolución de memoria es llevada a cabo directamente por el programador.
- De esta forma nuestros programas aprovecharán mejor la memoria del hardware en el que se ejecuten, usando sólo lo necesario.
- Permite posponer la decisión del tamaño del bloque de memoria necesario para guardar, por ejemplo un array, hasta el tiempo de ejecución.

FUNCIONES DE MEMORIA DINÁMICA

- `void* malloc(size_t): Reserva memoria dinámica.`

  La función malloc (Memory allocate - asignar memoria) reservar una parte de la memoria y devuelve la dirección del comienzo de esa parte. Esta dirección podemos almacenarla en un puntero y así podemos acceder a la memoria reservada. La función malloc tiene el siguiente formato:

  ```
  puntero = (tipo_de_variable *) malloc( número de bytes a reservar );
  ```

  Si no había suficiente memoria libre malloc devolverá el valor NULL. El puntero por tanto apuntará a NULL. Es muy importante comprobar siempre si se ha podido reservar memoria o no comprobando el valor de puntero:

  `if (puntero)` se cumple si hay memoria suficiente, en caso contrario da falso.

- `free(void*): Libera memoria dinámica.`

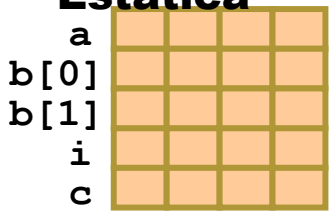
  Cuando ya no necesitemos más el espacio reservado debemos liberarlo, es decir, indicar al CPU que puede destinarlo a otros fines. Si no liberamos el espacio que ya no necesitamos corremos el peligro de agotar la memoria.

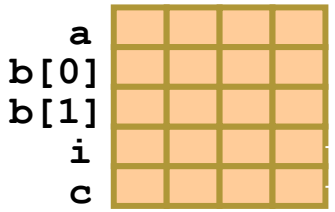
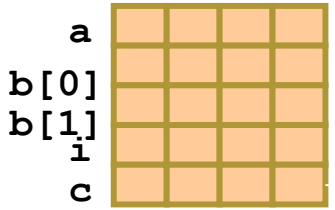
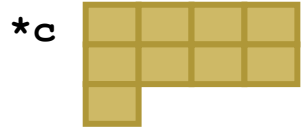
  `free( puntero );` Donde puntero es un puntero que apunta al comienzo del bloque que habíamos reservado. Es muy importante no perder la dirección del comienzo del bloque, pues de otra forma no podremos liberarlo.

- `void* realloc(void*,size_t): Ajusta el espacio de memoria dinámica.`

FUNCIONES DE MEMORIA DINÁMICA

	Estática	Dinámica

`int a, b[2];`
`int *i;`
`char *c;`
		

`i=(int*)malloc(sizeof(int));` `c=(char*)malloc(sizeof(char));`		
`free(i);` `c=(char*)realloc(c,sizeof(char)*9);`		

FUNCIONES DE MEMORIA DINÁMICA

```
int *ptr, i;
ptr = (int *)malloc(10*sizeof(int));
if (ptr==NULL) printf ("Error de Mem.");
for (i=0; i<10; i++)
ptr[i]=1;
```

Utilizando la equivalencia entre la notación de punteros y de arrays, asignamos a todos los elementos del array creado anteriormente un valor constante 1.

```
int nfilas=5, ncols=10
int fila;
int **filaptr;
filaptr = malloc(nfilas * sizeof(int
    *));
if (filaptr == NULL) printf ("Error");
for (fila=0; fila<nfilas; fila++) {
    filaptr[fila] = (int *) malloc(ncols
    * sizeof(int));
    if (filaptr == NULL) printf
    ("Error");
}
```

filaptr es un puntero a puntero a entero. En este caso, el programa crea una matriz de 5 filas por 10 columnas. Esos datos, naturalmente, podrían gestionarse en tiempo de ejecución. La primera llamada a malloc reserva espacio para el array de punteros a los arrays de enteros. Por esta razón, a la función sizeof le pasamos el tipo “puntero a entero”. Con el primer array de punteros creado, pasamos a reservar memoria para los arrays que son apuntados por cada elemento “fila”

ESTRUCTURAS DE DATOS DINÁMICAS

Una de las aplicaciones más interesantes y potentes de la memoria dinámica y de los punteros son las estructuras dinámicas de datos. Las estructuras básicas disponibles en C tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los arrays están compuestos por un determinado número de elementos, número que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Por supuesto, podemos crear arrays dinámicos, pero una vez creados, su tamaño también será fijo, y para hacer que crezcan o disminuyan de tamaño, deberemos reconstruirlos desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas. Pero no sólo eso, también nos permitirán crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

Las estructuras de datos están compuestas de otras pequeñas estructuras a las que llamaremos nodos o elementos, que agrupan los datos con los que trabajará nuestro programa y además uno o más punteros autoreferenciales, es decir, punteros a objetos del mismo tipo. De este modo, cada nodo puede usarse como un ladrillo para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos.

Para acceder a un nodo de la estructura sólo necesitaremos un puntero a un nodo.

ESTRUCTURAS DE DATOS DINÁMICAS

Las estructuras dinámicas son una implementación de TDAs o TADs (Tipos Abstractos de Datos). En estos tipos el interés se centra más en la estructura de los datos que en el tipo concreto de información que almacenan.

Dependiendo del número de punteros y de las relaciones entre nodos, podemos distinguir varios tipos de estructuras dinámicas. Enumeraremos ahora sólo de los tipos básicos:

- Listas enlazadas: cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.
- Pilas: son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.
- Colas: otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.
- Listas circulares: o listas cerradas, el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.
- Listas doblemente enlazadas: cada elemento dispone de dos punteros, uno a punta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.

ESTRUCTURAS DE DATOS DINÁMICAS

- Arboles: cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.
- Arboles binarios: son árboles donde cada nodo sólo puede apuntar a dos nodos.
- Arboles binarios de búsqueda (ABB): son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.
- Arboles AVL: son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
- Arboles B: son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.
- Tablas HASH: son estructuras auxiliares para ordenar listas.
- Grafos: es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.
- Diccionarios.

ARRAYS DINÁMICOS

Si al iniciar un programa no se sabe el número de elementos del que va a constar el array, o no se quiere poner un límite predetermiado, lo que hay que hacer es definir el array dinámicamente. Para hacer esto, primero se define un puntero, que señalará la dirección de memoria del primer elemento del array:

```
tipo_de_elemento *nombre_de_array;
```

y luego se utiliza la función malloc (contenida en stdlib.h) para reservar memoria:

```
nombre_de_array=(tipo_de_elemento *)malloc(tamaño);
```

donde tamaño es el número de elementos del array por el tamaño en bytes de cada elemento. Se suele utilizar la funcion Sizeof(tipo_de_elemento). La función malloc devuelve un puntero void, que indica la posición del primer elemento.

Para arrays bidimensionales, hay que hacerlo dimensión a dimensión; primero se define un puntero de punteros:

```
int **mapa;
```

Luego se reserva memoria para los punteros:

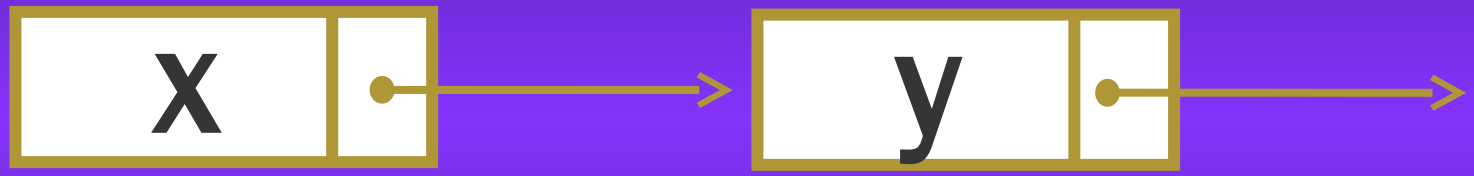
```
mapa=(int **)malloc(sizeof(int *)*Filas);
```

y, por último, para cada puntero se reserva memoria para los elementos:

```
for(i1=0;i1<Filas;i1++)
   mapa[i1]=(int *)malloc(sizeof(int)*Columnas);
```

LISTAS ENLAZADAS

Una lista es una estructura de datos secuencial en donde la posición del siguiente elemento de la estructura la determina el elemento actual. Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.



Su ventaja fundamental es que son flexibles a la hora de reorganizar sus elementos; a cambio se ha de pagar una mayor lentitud a la hora de acceder a cualquier elemento.

En la lista de la figura anterior se puede observar que hay dos elementos de información, x e y. Supongamos que queremos añadir un nuevo nodo, con la información p, al comienzo de la lista. Para hacerlo basta con crear ese nodo, introducir la información p, y hacer un enlace hacia el siguiente nodo, que en este caso contiene la información x.

LISTAS ENLAZADAS

Se puede definir la lista enlazada en C de la siguiente manera:

```
struct lista
{
  int dato;
  struct lista *sig;
};
```

Como se puede observar, en este caso el elemento de información es simplemente un número entero. Además se trata de una definición autorreferencial. Pueden hacerse definiciones más complejas. Ejemplo:

```
struct cl
{
  char nombre[20];
  int edad;
};
```

```
struct lista
{
  struct cl datos;
  int clave;
  struct lista *sig;
};
```

Cuando se crea una lista debe estar vacía. Por tanto para crearla se hace lo siguiente:

```
struct lista *L;
L = NULL;
```

OPERACIONES CON LISTAS ENLAZADAS

- Inserción al comienzo de una lista:

Es necesario utilizar una variable auxiliar, que se utiliza para crear el nuevo nodo mediante la reserva de memoria y asignación de la clave. Posteriormente es necesario reorganizar los enlaces, es decir, el nuevo nodo debe apuntar al que era el primer elemento de la lista y a su vez debe pasar a ser el primer elemento.

- Inserción al final de una lista:

La lista ya debe contar con por lo menos un nodo. Se debe recorrer la lista hasta ubicar el puntero nulo, que es el ultimo de la lista. En esa posición se realiza la inserción del nuevo nodo.

- Recorrido de una lista.

La idea es ir avanzando desde el primer elemento hasta encontrar la lista vacía. Antes de acceder a la estructura lista es fundamental saber si esa estructura existe, es decir, que no está vacía. En el caso de estarlo o de no estar inicializada es posible que el programa falle y sea difícil detectar donde, y en algunos casos puede abortarse inmediatamente la ejecución del programa, lo cual suele ser de gran ayuda para la depuración.

- Eliminar un nodo de la lista:

Se pueden eliminar nodos al inicio, al final o un nodo buscado. En todos los casos utilizamos punteros auxiliares para poder indicar que un nodo deja de pasar a otro. Se debe verificar que no este vacía la lista.

OPERACIONES CON LISTAS ENLAZADAS

- **Inserción en orden:**

Las listas ordenadas son aquellas en las que la posición de cada elemento depende de su contenido. Cuando haya que insertar un nuevo elemento en la lista ordenada hay que hacerlo en el lugar que le corresponda, y esto depende del orden y de la clave escogidos. Este proceso se realiza en tres pasos:

1.- Localizar el lugar correspondiente al elemento a insertar. Se utilizan dos punteros: *anterior* y *actual*, que garanticen la correcta posición de cada enlace.

2.- Reservar memoria para él (puede hacerse como primer paso). Se usa un puntero auxiliar (*nuevo*) para reservar memoria.

3.- Enlazarlo. Esta es la parte más complicada, porque hay que considerar la diferencia de insertar al principio, no importa si la lista está vacía, o insertar en otra posición. Se utilizan los tres punteros antes definidos para actualizar los enlaces.

- **Busqueda de datos:**

se realiza para localizar la posición de un objeto comparando su valor, ya que si se tiene su dirección simplemente haciendo referencia podríamos acceder a su contenido. Necesitaremos el valor a buscar y el inicio de lista. Utilizaremos una variable auxiliar para no perder el inicio de la lista. Recorremos toda la lista y vamos comparando si cada elemento es igual al buscado. Una vez localizado simplemente retornamos la dirección del elemento. Si no se lo encuentra se retornará NULL.

- **Eliminar la lista:**

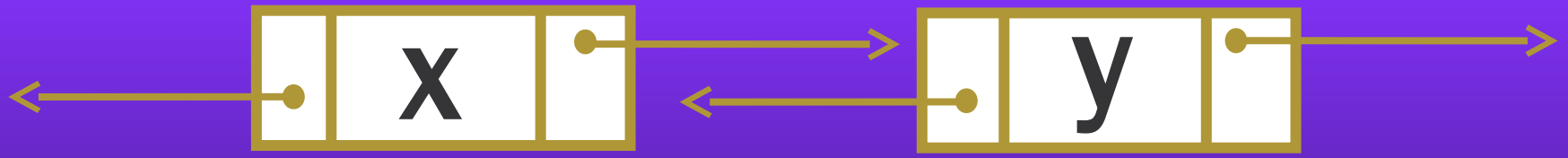
El primer elemento apunta a NULL.

LISTAS DOBLEMENTE ENLAZADAS

Son listas que tienen un enlace con el elemento siguiente y con el anterior.

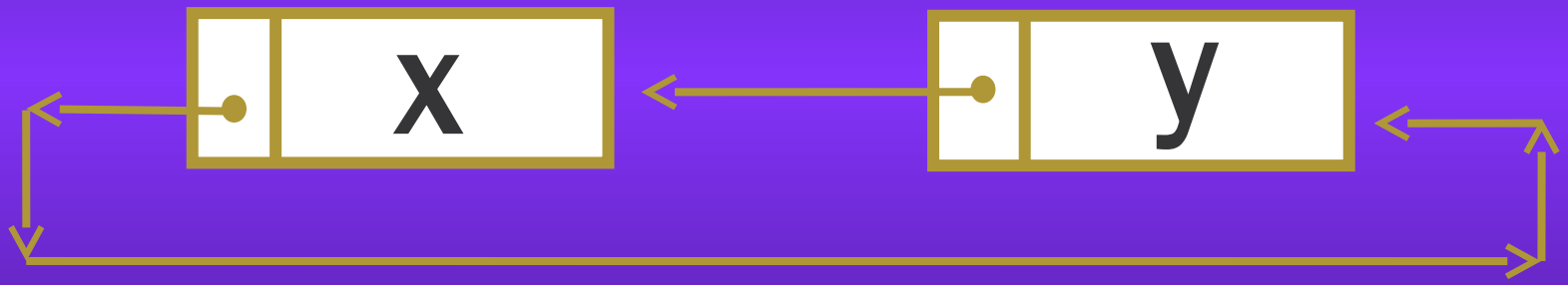
Una ventaja que tienen es que pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar/actualizar y borrar.

La otra ventaja es que las búsquedas son algo más rápidas puesto que no hace falta hacer referencia al elemento anterior. Su inconveniente es que ocupan más memoria por nodo que una lista simple.



LISTAS CIRCULARES

Las listas circulares son aquellas en las que el último elemento tiene un enlace con el primero. Su uso suele estar relacionado con las colas



PILAS

Una pila es una estructura de datos de acceso restrictivo a sus elementos. Se puede entender como una pila de libros que se amontonan de abajo hacia arriba. En principio no hay libros; después ponemos uno, y otro encima de éste, y así sucesivamente. Posteriormente los solemos retirar empezando desde la cima de la pila de libros, es decir, desde el último que pusimos.

En los programas estas estructuras suelen ser fundamentales. La recursividad se simula en una computadora con la ayuda de una pila. Asimismo muchos algoritmos emplean las pilas como estructura de datos fundamental, por ejemplo para mantener una lista de tareas pendientes que se van acumulando.

Las pilas ofrecen dos operaciones fundamentales, que son apilar y desapilar sobre la cima. El uso que se les de a las pilas es independiente de su implementación interna. Es decir, se hace un encapsulamiento. Por eso se considera a la pila como un tipo abstracto de datos.

La inserción y la supresión de datos solo se efectua en el extremo libre de la pila.

Es una estructra de tipo LIFO (Last In First Out), es decir, último en entrar, primero en salir.

COLAS

Una cola es una estructura de datos de acceso restrictivo a sus elementos. Un ejemplo sencillo es la cola del cine o del banco, el primero que llegue será el primero en entrar.

Las colas serán de ayuda fundamental para ciertos recorridos de árboles y grafos.

Las colas ofrecen dos operaciones fundamentales, que son encolar (al final de la cola) y desencolar (del comienzo de la cola). Al igual que con las pilas, la implementación de las colas suele encapsularse, es decir, basta con conocer las operaciones de manipulación de la cola para poder usarla, olvidando su implementación interna.

La inserción se hace por un extremo de la lista (por el final) y la supresión se hace por el otro extremo de la lista (por el principio).

Es una estructura de tipo FIFO (First In First Out), es decir: primero en entrar, primero en salir.

BIBLIOGRAFÍA & LICENCIA

- Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos, entre los que destaco el libro: *C/C++ Curso de programación*, 2da Ed, Javier Ceballos, Alfaomega Ra-Ma.
- Este documento se encuentra bajo Licencia Creative Commons Attribution – NonCommercial - ShareAlike 4.0 International (CC BY-NC-SA 4.0), por la cual se permite su exhibición, distribución, copia y posibilita hacer obras derivadas a partir de la misma, siempre y cuando se cite la autoría del Prof. Matías E. García y sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.
- Autor:

Matías E. García

Prof. & Tec. en Informática Aplicada

www.profmatiasgarcia.com.ar

info@profmatiasgarcia.com.ar