1. Punteros

|  |
| --- |
| Los punteros son una de las poderosas herramientas que ofrece el lenguaje **C** a los programadores, sin embargo, son también una de las más peligrosas, el uso de punteros sin inicializar, etc., es una fuente frecuente de errores en los programas de **C**, y además, suele producir fallas muy difíciles de localizar y depurar. |

* 1. Definición

|  |
| --- |
| Un puntero es una variable que contiene una dirección de memoria, normalmente esa dirección es una posición de memoria de otra variable, por lo cual se suele decir que el puntero “apunta” a la otra variable. |

* 1. Declaración de un puntero

|  |
| --- |
| La sintaxis de la declaración de una variable puntero es: |

|  |
| --- |
| **tipo*\**** nombre***;*** |

|  |
| --- |
| El “**tipo**” base de la declaración sirve para conocer el tipo de datos de la variable apuntada por el puntero.  Esto es fundamental para poder leer el valor que almacena la zona de memoria apuntada por el puntero y para poder realizar ciertas operaciones aritméticas sobre los mismos. |

|  |
| --- |
| Algunos ejemplos de declaración de variables puntero son: |

|  |
| --- |
| **int*\**** a***;* char*\**** p***;* float*\**** f***;*** |

* 1. Operadores

|  |
| --- |
| Existen dos operadores especiales de los punteros:   * El operador ' & ' encargado de retornar la dirección de memoria de una variable. * El operador ' \* ' encargado de retornar el valor de la variable apuntada por un puntero.   Si declaramos:  **int*\**** a***;***  **int** b***;***  Y hacemos:    a ***= &***b***;*** |
| La variable puntero **'a'** contendrá la dirección de memoria de la variable **'b'**.  Veámoslo en otro ejemplo:  **int*\**** a***;***  **int** b***;***  **int** c***;***  Y hacemos***:***    b ***=*** 15***;*** // Asignamos el valor 15 a la variable 'b'    a ***= &***b***;*** // Obtenemos la posicion de memoria de 'b' con el operador '&'    c ***= \****a***;*** // Copiamos el contenido apuntado por el puntero 'a'  Entonces la variable **'c'** contendrá el valor 15.  **'*\**a'** devuelve el contenido de la dirección a la que “apunta” la variable puntero **'a'** que con anterioridad hemos hecho que contenga la “dirección de memoria” de la variable **'b'** usando para ello el operador **'&'** |

* 1. Asignación de punteros

|  |
| --- |
| Es posible asignar el valor de una variable de tipo puntero a otra variable de tipo puntero.  Por ejemplo: |

|  |
| --- |
| **int*\**** a***;***  **int*\**** b***;***  **int** c***;***  a ***= &***c***;*** // Obtenemos la posición de memoria de 'c' con el operador '&'  b ***=*** a***;*** // Asignamos el valor del puntero 'a' al puntero 'b' |

|  |
| --- |
| Entonces **'b'** contiene el valor de **'a'**, por lo tanto, **'b'** también “apunta” a la variable **'c'**. |

* 1. Comparación de punteros

|  |
| --- |
| Sobre las variables de tipo puntero es posible realizar operaciones de comparación.  veamos un ejemplo: |
| **int*\**** punteroA***;***  **int*\**** punteroB***;***  **int** auxiliarC***;***  **int** auxiliarD***;***  punteroA ***= &***auxiliarC***;*** // Obtenemos la posicion de memoria de 'auxiliarC'  punteroB ***= &***auxiliarD***;*** // Obtenemos la posicion de memoria de 'auxiliarD'  **if *(***punteroA ***<*** punteroB***)***  ***{***  printf***(***"El punteroA apunta a una direccion mas baja que punteroB"***);***  ***}***  **else if *(***punteroA ***>*** punteroB***)***  ***{***  printf***(***"El punteroB apunta a una direccion mas baja que punteroA"***);***  ***}*** |

* 1. Aritmética de punteros

|  |
| --- |
| Sobre las variables de tipo puntero es posible utilizar los operadores ***+***, ***-***, ***++*** y ***--*** .  Estos operadores incrementan o decrementan la posición de memoria a la que “apunta” la variable puntero.  El incremento o decremento se realiza de acuerdo al **“tipo base”** de la variable de tipo puntero, de ahí la importancia del tipo del que se declara la variable puntero.  Veamos esto con la siguiente tabla: |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **char\*** a | **int\*** a | **float\*** a |
| a ***=*** a ***-***3 | 0xA07D | 0xB06E | 0xC074 |
| a ***=*** a ***-***2 | 0xA07E | 0xB078 | 0xC078 |
| a***--*** | 0xA07F | 0xB07C | 0xC07C |
| Posición inicial | 0xA080 | 0xB080 | 0xC080 |
| a***++*** | 0xA081 | 0xB084 | 0xC084 |
| a ***=*** a ***+***2 | 0xA082 | 0xB088 | 0xC088 |

|  |
| --- |
| Como podemos observar en la tabla cada operación sobre el puntero se rige por su tipo, por lo tanto, si tenemos: |

|  |
| --- |
| **tipo*\**** a***;***  Y hacemos:  a ***=*** a ***+*** num***;***  La posicion a la que apunta a se incrementa en:    contenido de “a” ***=*** direccion que contiene “a” ***+ (***num ***\** sizeof*(*tipo*))***    Para la resta se decrementa de igual forma en:    contenido de “a” ***=*** direccion que contiene “a” ***- (***num ***\** sizeof*(*tipo*))***  Los operadores ***++*** y ***--*** son equivalentes a realizar num ***=*** 1 , y con ello quedan obviamente explicados. |

* 1. Vectores y punteros

|  |
| --- |
| Existe una estrecha relación entre los punteros y los vectores.  Consideremos el siguiente fragmento de código:  **char** cadena[80];  **char*\**** p***;***  p ***= &***cadena[0];// equivalente a: p = cadena  Este fragmento de código pone en la variable puntero **'p'** la dirección del primer elemento del array **'cadena '**.  Entonces, es posible acceder al valor de la quinta posición del array mediante:  cadena***[***4***]*** y ***\*(***p ***+*** 4***)***  *(recuérdese que los índices de los arrays empiezan en 0).*  Esta estrecha relación entre los arrays y los punteros queda más evidente si se tiene en cuenta que el nombre del array sin el índice es la dirección de comienzo del array, y si además, se tiene en cuenta que un puntero puede indexarse como un array unidimensional, por lo cual, en el ejemplo anterior, podríamos referenciar ese elemento como:  p[4]  Es posible obtener la dirección de un elemento cualquiera del array de la siguiente forma:  **int** arrayInt[80];    **int*\**** p1***;***  **int*\**** p2***;***  p1 ***=*** &arrayInt[4];  p2 ***=*** &arrayInt;  Entonces, el puntero **'p1'** contiene la dirección del quinto elemento del **'arrayInt'** y el puntero **'p2'** contiene la dirección del primer elemento del arrayInt.  Hasta ahora hemos declarado variables puntero aisladas. Es posible, como con cualquier otro tipo de datos, definir un array de variables puntero.  La declaración para un array de punteros **int** de tamaño **10** es:  **int*\**** a[10];  Para asignar una dirección de una variable entera, llamada **'var'**, al tercer elemento del array de punteros, se escribe:  a[2] ***=*** &var;  Y para obtener el valor de **'var'**:  ***\****a[2];  Dado además, que un puntero es también una variable, es posible definir un puntero a un puntero.  Supongamos que tenemos lo siguiente:  **int** a***;***  **int*\**** punteroInt***;***  **int*\*\**** punteroPuntero***;***  punteroInt ***= &***a***;*** // Obtenemos la posicion de memoria  punteroPuntero ***= &***puntero***;*** // Obtenemos la posicion de memoria |

|  |  |  |
| --- | --- | --- |
| POSICION | VARIABLE | CONTENIDO |
| 0xB06E | **int** a***;*** | 2 |
| 0xB078 |  |  |
| 0xB07C | **int*\**** punteroInt***;*** | 0xB06E |
| 0xB080 |  |  |
| 0xB084 | **int*\*\**** punteroPuntero***;*** | 0xB07C |
| 0xB088 |  |  |

|  |
| --- |
| Y entonces,  ¿ De qué formas podemos ahora acceder al valor de la variable **'a'** ?  a *(forma habitual)*  ***\****punteroInt *(a través del puntero)*  ***\*\****punteroPuntero *(a través del puntero a puntero)*  Esto es debido a que **'punteroPuntero'** contiene la dirección de **'punteroInt'**, que contiene la dirección de **'a'**.  Este concepto de puntero a puntero podría extenderse a puntero a puntero a puntero, etc.,  además, existe el concepto de puntero a una función, al cual nos referiremos más adelante. |

1. Punteros II
   1. Tipos de Punteros

|  |
| --- |
| Un puntero es una variable como cualquier otra destinada a guardar una posición de memoria, esta posición de memoria generalmente corresponde a otra variable.  Lo que se pretende decir es que si a un puntero se le carga cualquier valor, éste será interpretado como si fuera una dirección de memoria aunque el valor cargado no corresponda a una dirección de memoria. |

* + 1. Punteros a cadenas de caracteres

|  |
| --- |
| Un puntero a “string” no es más que un puntero a **char**. Por lo tanto la forma de definir un puntero a una cadena de caracteres es:  **char*\**** p***;***  Por ejemplo la siguiente línea declara un puntero a **char** y un vector para guardar la cadena de caracteres.  **char** cadena***[***N\_LONGITUD***];***  **char*\**** puntero***;***  Así se reserva espacio para una cadena de N\_LONGITUD caracteres.  En C, el nombre de las listas monodimensionales equivale a la dirección del primer elemento de las mismas.  Una vez que se dispone de espacio reservado para la cadena, se puede construir un puntero de la cadena empleando las siguientes sentencias:  **char** cadena***[***LONGITUD***];***  **char*\**** puntero***;***  puntero ***=*** cadena*;*// es equivalente a escribir puntero = &cadena[0]  Esto dará lugar a que en **'puntero'** se almacene la dirección de **'cadena'**, es decir, se le asigna a **'puntero'** el valor de la dirección de un carácter, que va a ser el primero de la cadena. |

* + - 1. Recorrido de una cadena mediante aritmética de punteros e índices.

|  |
| --- |
| El mecanismo de la aritmética de punteros en C tiene su expresión más sencilla en el caso de utilizar, precisamente, punteros de caracteres.  Consideremos las siguientes declaraciones:  **tipo** lista***[***NUMERO\_DE\_ELEMENTOS***];***  **tipo*\**** puntero***;***  puntero ***=*** lista***;***  Se debe recordar que en general, cuando se suma 1 a un puntero, el valor numérico de la dirección no se incrementa en 1, sino en:  **sizeof*(***tipo***)***  Entonces, al sumar **'i'** a un puntero se tiene que la dirección obtenida al calcular **'puntero +i'** tiene el valor:  direccion\_actual\_del\_puntero ***+ (***i ***\** sizeof*(***tipo***))***  Dando como resultado un nuevo puntero, que señala una posición de memoria que correspondería a un elemento de una lista situado en **'i'** posiciones más allá del elemento señalado por **'puntero'**.  Se detalla a continuación un ejemplo en el cual aplicando aritmética de punteros recorreremos una cadena de caracteres :  **void** mostrarCadena***(* char*\**** punteroCadena ***)***  ***{***  **while*( \****punteroCadena ***!=*** '\0' ***)***  ***{***  printf***(*** "%c"***, \****punteroCadena ***);***  punteroCadena***++;***  ***}***  ***}***  Como se puede observar, la condicion del bucle indica que mientras que sea el contenido de la posicion apuntada por **'punteroCadena'** distinto a **'\0'** se continuará iterando, en cada iteración se imprime el contenido de la posición apuntada por **'punteroCadena'** y se incrementa el valor del puntero mediante **'punteroCadena++'**. |

* + 1. Punteros a estructura

|  |
| --- |
| Como una estructura es una variable, se la puede apuntar con un puntero. De acuerdo a la definición de puntero, deberá guardar la dirección de memoria de la estructura , si pensamos que una estructura puede estar formada por un grupo de variables distintas, el puntero tendrá la dirección de comienzo del grupo de variables.  Veamos un ejemplo: |

|  |
| --- |
| **typedef sruct**  ***{***  **char** nombre***[***20***];***  **int** nota***;***  ***}***sAlumno***;***  sAlumno auxiliarAlumno***;***  sAlumno***\**** punteroAlumno***;***  punteroAlumno ***= &***auxiliarAlumno***;*** |

|  |
| --- |
| Si se piensa en un vector de estructuras y se almacena en un puntero, del tipo de esa estructura, la posición de memoria que corresponde al comienzo de dicho vector, al momento de recorrerlo, utilizando el puntero, en cada incremento pasará a apuntar a la próxima estructura almacenada en el array. Lo recién enunciado cumple con la aritmética de punteros explicada anteriormente.  Veamos un ejemplo: |

|  |
| --- |
| #define CANTIDAD 5  **typedef struct**  ***{***  **char** nombre***[***20***];***  **int** nota***;***  ***}***sAlumno***;***  **void** main***(* void *)***  ***{***  **int** i***;***    sAlumno arrayAlumnos***[***CANTIDAD***];***  sAlumno***\**** punteroArrayAlumnos***;***    punteroArrayAlumnos ***=*** arrayAlumnos***;***    **for*(***i ***=*** 0***;*** i ***<*** CANTIDAD***;*** i***++)***  ***{***  punteroArrayAlumnos***++;*** //...  ***}***  ***}*** |

* + - 1. Acceso a los campos: Operador flecha ( -> )

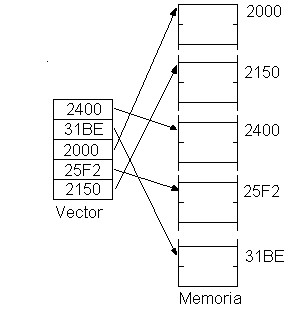
|  |
| --- |
| Cuando se usan los punteros a estructura la forma de acceder a un campo es usando el operador FLECHA ( ***->*** )el cual se forma escribiendo un guion y a continuacion un mayor.  Veamos un ejemplo: |

|  |
| --- |
| **typedef struct**  ***{***  **char** nombre***[***20***];***  **int** nota***;***  ***}***sAlumno***;***  **void** main***(*void*)***  ***{***  sAlumno auxiliarAlumno***;***  sAlumno***\**** punteroAlumno***;***    punteroAlumno ***= &***auxiliarAlumno***;***    strcpy***(*** punteroAlumno***->***nombre***,*** "Ernesto" ***);***  punteroAlumno***->***nota ***=*** 10***;***  ***}*** |

|  |
| --- |
| Como se puede ver al momento de querer acceder al campo “nota” de la estructura “auxiliarAlumno” mediante el puntero “punteroAlumno” , se remplaza el operador punto (auxiliarAlumno.nota) por el operador flecha (punteroAlumno->nota). |

* + - 1. Vector de punteros

|  |
| --- |
| Un vector de punteros es simplemente un vector que dentro de cada uno de sus elementos existe un puntero.  La forma de definir un vector de punteros es:  **tipo*\**** nombre***[***MAX***];***  En el caso que se requiera definir un vector de punteros **int**.  **int*\**** vec ***[***MAX***];***  El vector tiene posiciones de memoria (punteros) de algunas variables, por ejemplo, el elemento vec***[***i***]*** contiene la dirección de memoria de una variable y por lo tanto el contenido de ***\****vec***[***i***]*** será un entero.  Al igual que se inicializaba un puntero, se debe inicializar el vector de punteros para que cada posición del vector apunte a algún lugar determinado.  Veamos el siguiente dibujo que puede ilustrar mejor la situación. |



|  |
| --- |
| A la izquierda del dibujo tenemos el vector “vec” que es un vector de punteros , por lo tanto cada elemento del vector es una dirección de memoria (puntero). Inicialmente como pasa en cualquier variable tiene un valor cualquiera al momento de arrancar el programa, por lo cuál los punteros están apuntando a cualquier posición.  Normalmente se asocia un vector de punteros a otro de estructuras, es decir que el vector de punteros tiene las direcciones de cada uno de los elementos del vector de estructuras. Veamos una utilidad del vector de punteros.  Supongamos que se debe realizar un programa de agenda que cargue los datos de 100 alumnos y luego tenga que ordenar los datos según su nota. En el proceso de ordenamiento lo que se hace es copiar el elemento **'i'** a un **'auxiliar'** , luego copiar el elemento **'j'** al elemento **'i'** y finalmente el **'auxiliar'** al elemento **'j'** , o sea está realizando 3 copias de estructuras lo cual implica un movimiento importante de datos. |

|  |
| --- |
| **for*(***i ***=*** 0***;*** i ***<*** CANTIDAD ***-***1***;*** i***++)***  ***{***  **for*(***j ***=*** i ***+*** 1***;*** i ***<*** CANTIDAD***;*** j***++)***  ***{***  **if*(***arrayAlumnos***[***i***].***nota ***>*** arrayAlumnos***[***j***].***nota***)***  ***{***  auxiliarAlumno ***=*** arrayAlumnos***[***i***];***  arrayAlumnos***[***i***] =*** arrayAlumnos***[***j***];***  arrayAlumnos***[***j***] =*** auxiliarAlumno***;***  ***}***  ***}***  ***}*** |

|  |
| --- |
| Ahora si usted tiene un vector de punteros, el movimiento de datos se realiza en el vector de punteros, con lo cuál la copia se reduce notablemente (la cantidad de bytes a copiar se reduce a 2 bytes que es el tamaño de un puntero) y por otra parte no se modifica el vector de estructuras. Cuando se quiere mostrar los datos ordenados simplemente se recorre el vector de estructuras.  Veamos un ejemplo: |

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define CANTIDAD 3  **typedef struct**  {  **char** nombre[50];  **int** nota;  }sAlumno;  **void** ordenar(sAlumno\* arrayPunterosAlumnos[3], **int** longitudArray);  **int** main()  {  **int** i;  **int** auxiliarNota;  **char** auxiliarNombre[50];  sAlumno arrayAlumnos[CANTIDAD];  sAlumno\* arrayPunterosAlumnos[CANTIDAD];  **for**(i = 0; i < CANTIDAD; i++)  {  // COPIAMOS LA DIRECCION DE CADA ELEMENTO[i]  EN EL ARRAY DE PUNTEROS  arrayPunterosAlumnos[i] = &arrayAlumnos[i];  // CARGAMOS LOS DATOS EN CADA ELEMENTO DEL  ARRAY ESTRUCTURA  printf("Ingrese el nombre: ");  scanf("%s", auxiliarNombre);  strcpy(arrayAlumnos[i].nombre, auxiliarNombre);  //FALTARIA VALIDAR  printf("Ingrese la Nota: ");  scanf("%i", &auxiliarNota);  arrayAlumnos[i].nota = auxiliarNota;  //FALTA VALIDAR  }  printf("-MOSTRAMOS EL ARRAY SIN ORDENAR-\n");  **for**(i=0; i<CANTIDAD; i++)  {  printf("%s", arrayPunterosAlumnos[i]->nombre);  printf("-%i\n", arrayPunterosAlumnos[i]->nota);  }  ordenar(arrayPunterosAlumnos, CANTIDAD);  printf("-MOSTRAMOS EL ARRAY ORDENADO-\n");  **for**(i=0; i<CANTIDAD; i++)  {  printf("%s", arrayPunterosAlumnos[i]->nombre);  printf("-%i\n", arrayPunterosAlumnos[i]->nota);  }  **return** 0;  }  **void** ordenar(sAlumno\* arrayPunterosAlumnos[], **int** longitudArray)  {  sAlumno\* punteroAuxiliarAlumno;  **int** i;  **int** j;  **for**(i = 0; i < longitudArray -1 ; i++)  {  **for**(j = i+1; j < longitudArray ; j++)  {  **if**(arrayPunterosAlumnos[i]->nota > arrayPunterosAlumnos[j]->nota)  {  punteroAuxiliarAlumno = arrayPunterosAlumnos[i];  arrayPunterosAlumnos[i] = arrayPunterosAlumnos[j];  arrayPunterosAlumnos[j] = punteroAuxiliarAlumno;  }  }  }  } |

|  |
| --- |
| Observe que cuando se realiza el ordenamiento se ordena el vector de punteros, con lo cuál el vector de estructuras queda intacto.  Además de no modificarse el vector de estructuras, en el proceso de ordenar al modificar el vector de punteros siempre se mueve la misma cantidad de datos, es decir que no interesa la cantidad de campos que posea la estructura.  Entonces si una estructura contiene 20 campos y otra solo 2, en las asignaciones dentro del algoritmo que ordena solo se copiarán datos del tamaño del puntero. |

*Ejercicios y videos*

[VIDEO PARTE 1](https://youtu.be/sdo7miC8jSQ)

[VIDEO PARTE 2](https://youtu.be/pbjB6l4xMBQ)

[VIDEO PARTE 3](https://youtu.be/xujd4olvCK0)

[RESOLUCION EJERCICIO 1](https://youtu.be/qVW7UfAB7Dk?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[RESOLUCION EJERCICIO 2](https://youtu.be/pQbN5631cqg?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

**Ejercicio 1:**

1) Escribir 4 funciones que reciban un array de números int y su tamaño, e impriman los números por pantalla.

* 1era funcion: Recibe el array como vector y accede a los datos utilizando notación vectorial ([]).
* 2da funcion: Recibe el array como vector y accede a los datos utilizando aritmetica de punteros.
* 3era funcion: Recibe el array como puntero y accede a los datos utilizando notación vectorial ([]).
* 4ta funcion: Recibe el array como puntero y accede a los datos utilizando aritmetica de punteros.

**Ejercicio 2:**

1. Realizar una función para cargar los datos de una persona en una struct, para ello se debe definir una estructura con los campos nombre, edad, dni.
2. La función tendrá el siguiente prototipo:

int loadPerson(struct S\_Person\* pPerson);

1. La función deberá devolver un 1 si los datos ingresados son correctos, y un 0 si no lo son.
2. Hacer otra función que reciba un puntero a la estructura e imprima por pantalla los datos, el prototipo es:

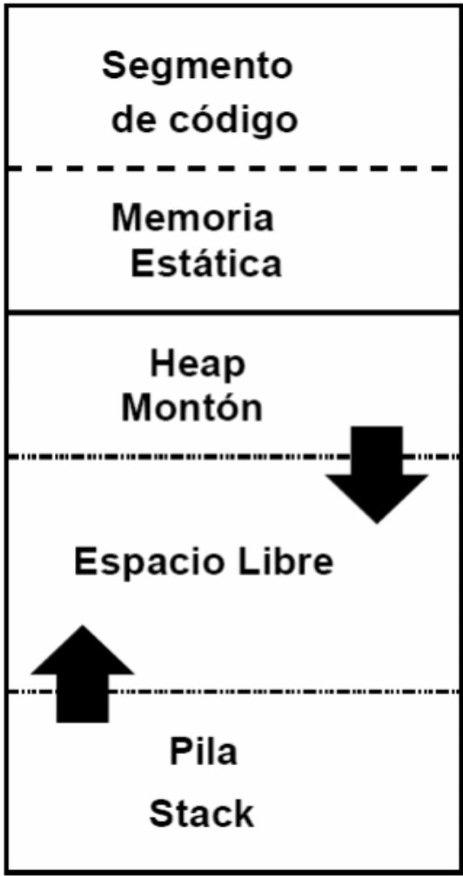
void printPerson(struct S\_Person\* pPerson);

1. Escribir un programa para probar las funciones.
2. Memoria Dinámica
   1. Asignación dinámica de memoria

|  |
| --- |
| Las funciones que realizan un manejo dinámico de la memoria del sistema requieren todas ellas para su correcto funcionamiento la inclusión, mediante la directiva del prepocesador #include del archivo de cabecera <stdlib.h>. |

* + 1. Segmentos de memoria

|  |
| --- |
| Cada vez que se ejecuta cualquier programa, el mismo deberá pasar a memoria.  Los programas en memoria tienen varios segmentos, los cuales sirven para administrar el manejo de la memoria.   * **Segmento de Código:** En este segmento se guardan las instrucciones, en lenguaje máquina, de nuestro programa. * **Segmento de Memoria estática:** En este Segmento se guardan las variables globales del programa. * **Segmento del Stack:** Cada vez que se llama a una función entra en este segmento con toda su información y allí se guardan: * Los llamados a las funciones. * Los parámetros de las funciones llamadas. * Las variables locales. * Otra información necesaria para el funcionamiento del programa. * **Segmento del Heap:** En este segmento se guardan las variables que han sido creadas dinámicamente en tiempo de ejecución. |
|  |



* + 1. Reserva dinámica de memoria

|  |  |
| --- | --- |
| En el lenguaje C, la reserva dinámica de memoria se realiza mediante el uso de funciones, explicaremos el uso de **malloc()** y **calloc()** :  La función **malloc()** tiene la forma:  **void*\**** malloc***(* unsigned int** numBytes ***);***  Siendo **'numBytes'** el número de bytes que se desean reservar.  **malloc()** devuelve un puntero al tipo de datos **'void'** *(sin tipo).*  Dicho puntero puede ser asignado a una variable puntero del tipo que definamos mediante una conversión forzada llamada“casting”.  Veamos un ejemplo:  **int*\**** a***;***    a ***= (*int*\*)*** malloc***(* sizeof*(*int*) );***  Y ahora podríamos realizar la siguiente asignación:  ***\****a ***=*** 3***;***  La función **calloc()** tiene la forma:  **void*\**** calloc***(* unsigned int** numElementos***,* unsigned int** numBytesElemento ***);***  A diferencia de **malloc()** la función **calloc()** inicializa a 0 el contenido de cada elemento de un array.  Podemos apreciar como en la definición de **calloc()** el primer parámetro es el numero de elementos y a continuación el tamaño del elemento.  Veamos un ejemplo:  **int*\**** a***;***    a ***= (*int*\*)*** calloc***(*** 20***,* sizeof*(*int*) );***  Aquí ya tenemos un array de veinte elementos en el cual cada uno de los elementos fue inicializado en cero.  Tanto la función **malloc()** como **calloc()**, devuelven un puntero nulo (**NULL**) si la reserva de memoria no puede realizarse, generalmente por falta de memoria disponible.  Por ello, antes de usar un puntero devuelto por la función **malloc()** o por cualquier otra función de reserva dinámica de memoria es imprescindible, con el fin de evitar posibles fallos en la ejecución del programa, comprobar que dicho puntero no es nulo (**NULL**).  **int*\**** a***;***  a ***= (*int*\*)*** malloc***(* sizeof*(*int*) );***    **if *(***a ***==*** NULL***)***  ***{***  printf***(***"NO QUEDA MEMORIA"***);***  ***}***  Veamos algunos ejemplos de reserva dinámica de memoria:   |  | | --- | | **Ejemplo 1**  **float*\**** a***;***  a ***= (*float*\*)*** malloc***(* sizeof*(*float*) );***  **if *(*** a ***==*** NULL ***)***  ***{***  exit***(***0***);*** /\* Salimos del programa \*/  ***}***  **Ejemplo 2**    **unsigned long int*\**** b***;***  b ***= (*unsigned long int*\*)*** malloc***(* sizeof*(*unsigned long int*) );***  **if*(*** b ***==*** NULL***)***  ***{***  exit***(***0***);*** /\* Salimos del programa \*/  ***}***  Ejemplo **3**    **typedef struct**  ***{***  **unsigned int** a***;***  **float** b***;***  **int *\****c***;***  ***}***sALFA***;***  sALFA***\**** d***;***  **if*( (***d ***= (***sALFA***\*)*** malloc***(* sizeof*(***sALFA***) ) ==*** NULL ***)***  ***{***  exit***(***0***);*** /\* Salimos del programa \*/  ***}*** | |

* + 1. Redimensionamiento dinámico de memoria

|  |
| --- |
| El redimensionamiento dinámico de memoria intenta cambiar el tamaño de un bloque de memoria previamente asignado. El nuevo tamaño puede ser más grande o más pequeño.  Si el bloque se hace más grande, entonces el contenido anterior permanece sin cambios y la memoria es agregada al final del bloque.  Si el bloque se hace más pequeño entonces el contenido sobrante permanece sin cambios.  La función utilizada en el leguaje C para redimensionar memoria es **realloc()**.  La función **realloc()** tiene la forma: |

|  |
| --- |
| **void*\**** realloc***(* void*\**** ptr***,* unsigned int** size ***);*** |

|  |
| --- |
| Si bloque original no puede ser redimensionado, entonces **realloc()** intentará asignar un nuevo bloque de memoria y copiará el contenido anterior.  Por lo tanto, la función devolverá un nuevo puntero, este nuevo valor será el que deberá usarse.  Si no puede ser reasignada nueva memoria la función **realloc()** devuelve **NULL**.  **Ejemplo**:  Imaginemos que tenemos el puntero **'punteroA'** el cual apunta a un espacio de memoria de 2 chars de tamaño que fue reservado con la función **malloc()**a partir de la dirección 0x01. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DIRECCION | ESTADO |  | DIRECCION | ESTADO |
|  | 0x00 | Otros datos |  | 0x00 | Otros datos |
| **punteroA -->** | 0x01 | ASIGNADO |  | 0x01 | LIBRE |
|  | 0x02 | ASIGNADO |  | 0x02 | LIBRE |
|  | 0x03 | Otros datos |  | 0x03 | Otros datos |
| **punteroB -->** | 0x04 | ASIGNADO | **punteroB -->** | 0x04 | ASIGNADO |
|  | 0x05 | ASIGNADO |  | 0x05 | ASIGNADO |
|  | 0x06 | Otros datos |  | 0x06 | Otros datos |
|  | 0x07 | LIBRE | **punteroA -->** | 0x07 | ASIGNADO |
|  | 0x08 | LIBRE |  | 0x08 | ASIGNADO |
|  | 0x09 | LIBRE |  | 0x09 | ASIGNADO |
|  | 0x0A | LIBRE |  | 0x0A | ASIGNADO |

|  |
| --- |
| Si se quiere reasignar la memoria a cuatro **char** en lugar de los dos, se hará: |

|  |
| --- |
| punteroAuxiliar ***= (*char*\*)*** realloc***(*** punteroA***,*** 4 ***\** sizeof*(*char*) );***  **if*(***punteroAuxiliar ***!=*** NULL***)***  ***{***  punteroA ***=*** punteroAuxiliar***;***  ***}***  **else**  ***{***  printf***(***"NO QUEDA MEMORIA"***);***  ***}*** |

|  |
| --- |
| Como se observa en la tabla de la derecha, **'punteroA'** cambió su dirección inicial, ya que de otra manera no se podían conseguir los 4 bytes consecutivos, debido a que el espacio de memoria debajo de **'punteroA'** ya estaba ocupado. |

* + 1. Liberación dinámica de memoria

|  |
| --- |
| La memoria dinámica reservada es eliminada siempre al terminar la ejecución del programa por el propio sistema operativo. Sin embargo, durante la ejecución del programa puede ser interesante, e incluso necesario, proceder a liberar parte de la memoria reservada con anterioridad y que ya ha dejado de ser necesario tener reservada. Esto puede realizarse mediante la función **free().**  La función **free()** tiene la forma:  **void** free***(* void*\**** p ***);***  Donde **'p'** es la variable de tipo puntero cuya zona de memoria asignada de forma dinámica queremos liberar.  Veamos un ejemplo de liberación de memoria:  **int*\**** a***;***  **if*( (***a ***= (*int*\*)*** malloc***(* sizeof*(*int*) )) ==*** NULL ***)***  ***{***  exit***(***0***);*** /\* Salimos del programa \*  ***}***  //...  free***(*** a ***);***  Un aspecto a tener en cuenta es el hecho de que el puntero a liberar no debe apuntar a nulo (**NULL**), pues en tal caso se producirá un fallo en el programa. Es por ello que cobra aún más sentido la necesidad de comprobar al reservar memoria de forma dinámica que la reserva se ha realizado de forma correcta, tal y como se explicó anteriormente. |

* + 1. Ejemplo de asignación, redimensionamiento y liberación de memoria

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  **typedef struct**  ***{***  **char** nombre***[***50***];***  **int** edad***;***  ***}***sPersona***;***  **int** main***()***  ***{***  **int** seguirCargando***;***  **int** i***;***    **int** auxNuevaLogitud***;***  **int** longitudPersonas ***=*** 1***;***    sPersona***\**** pArrayPersona***;***  sPersona***\**** pAuxPersona***;***      // Creamos el array de personas  pArrayPersona ***=*** malloc***(* sizeof*(***sPersona***) );***  **if*(***pArrayPersona ***==*** NULL***)***  ***{***  printf***(***"No hay lugar en memoria\n"***);***  exit***(***0***);***  ***}***  **while*(***1***)***  ***{***  printf***(***"Ingrese nombre: \n"***);***  scanf***(*** "%s"***, (***pArrayPersona ***+***logitudPersonas ***-***1***)->***nombre ***);***  printf***(***"Ingrese edad: \n"***);***  scanf***(*** "%d"***, &( (***pArrayPersona ***+***logitudPersonas***-***1***)->***edad ***) );***  printf***(***"Si desea cargar otra persona ingrese (1): \n"***);***  scanf***(***"%d"***, &***seguirCargando***);***  **if*(***seguirCargando ***==*** 1***)***  ***{***  // Calculamos el nuevo tamaño del array  auxNuevaLongitud ***=* sizeof*(***sPersona***) \**** logitudPersonas***;***  // Redimencionamos la lista  pAuxPersona ***=*** realloc***(*** pArrayPersona***,*** auxNuevaLogitud ***);***  **if*(***pAuxPersona ***==*** NULL***)***  ***{***  printf***(***"No hay lugar en memoria\n"***);***  **break*;***  ***}***  logitudPersonas***++;*** //Incremento el contador de personas  pArrayPersona ***=*** pAuxPersona***;***  ***}***  **else**  ***{***  **break*;***  ***}***  ***}***    **for*(***i ***=*** 0***;*** i ***<*** logitudPersonas***;*** i***++)***  ***{***  printf***(***"Nombre: %s - "***, (***pArrayPersona ***+***i***)->***nombre***);***  printf***(***"Edad: %d \n"***, (***pArrayPersona ***+***i***)->***edad***);***  ***}***    free***(*** pArrayPersona ***);*** // Liberamos la memoria  **return** 0***;***  ***}*** |

*Ejercicios y videos*

[VIDEO PARTE 1](https://youtu.be/cNB1arCvVOQ)

**Ejercicio 1:**

1)  Construir una función llamada **"getDynamicString"** que permita al usuario ingresar un texto y

devuelva un puntero a un espacio de memoria donde esta almacenado el texto ingresado.

Se requiere el uso de memoria dinámica.

1. Archivos
   1. Archivos

|  |
| --- |
| Todos los programas que se ejecutaron hasta el momento requerían de ingreso de datos para realizar determinadas tares y luego mostrar resultados.  Tales programas tienen un gran defecto y es que los datos que se cargan desaparecen cuando el programa termina, por lo tanto, cada vez que se corra el programa hay que cargar los datos nuevamente.  Esto ocurre debido a que los datos ingresados quedan guardados en memoria y al terminar el programa todos los espacios de memoria reservados son devueltos al sistema operativo perdiendo todos los datos que se habían cargado.  Para solucionar el problema debería existir una forma de almacenar los datos en forma permanente, por ejemplo, en un medio como el disco rígido.  Los datos que se ingresan se deberán guardar en un archivo que se almacena en el disco de forma de disponer de la información en cualquier momento.  De tal forma si por ejemplo trabajamos con un programa de agenda, en la cual se cargan los datos de personas, cada vez que se ejecute el programa se ira a leer el archivo que contiene los datos de las personas , en lugar de tener que ingresarlos cada vez . |

* + 1. Archivos de texto

|  |
| --- |
| Un archivo de texto contiene toda su información guardada en binario, pero se interpreta como texto.  Absolutamente todo lo que contiene debe ser interpretado como texto, ya que cuando se escribe el archivo, los datos son enviados como caracteres.  Cuando se dice que la información esta guardada en formato texto se esta haciendo referencia a como es la forma en la que hay que entender el dato, ya que sobre el disco los datos son guardados en forma de secuencia de “UNOS” y “CEROS” es decir según el sistema binario.  Supongamos que se desea guardar en un archivo de texto la siguiente cadena “Ana 12”. Lo que se guarda en el archivo son caracteres es decir la “A” , la “n” , la “a” , “ el espacio” , “el caracter 1” y “el caracter 2” , entonces si analizamos el equivalente ASCII de los caracteres , en el archivo queda: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Caracter | A | n | a | 1 | 2 |
| ASCII | 33 | 110 | 97 | 49 | 50 |
| Binario | 0b00100001 | 0b01101110 | 0b01100001 | 0b00110001 | 0b00110010 |

|  |
| --- |
| Lo que figura en la fila que dice binario es lo que queda en el disco rígido.  Si por ejemplo se quiere guardar en el archivo de texto el número “45678” se va a guardar como caracteres, es decir que en el disco queda: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Caracter | 4 | 5 | 6 | 7 | 8 |
| ASCII | 52 | 53 | 54 | 55 | 56 |
| Binario | 0b00110100 | 0b00110101 | 0b00110110 | 0b00110111 | 0b00111000 |

|  |
| --- |
| Se observa que para guardar un número como texto la cantidad de bytes a usar es igual a la cantidad de dígitos, que en este caso en particular estamos usando 5 bytes.  En el caso de usar un tipo de variable int solo se necesitan 2 bytes pero para ello deberíamos hacerlo en un archivo binario. |

* + 1. Archivos binarios

|  |
| --- |
| En un archivo binario se guardan datos con distinto formato, es decir se pueden guardar caracteres mezclados con enteros y flotantes.  Si bien todos los datos terminan escritos en el disco en sistema binario, la interpretación de los datos guardados cambia.  Si tomamos el último ejemplo dado para archivos de texto, y queremos escribir en el archivo el número 45678, en el programa va a estar definido como un entero y por lo tanto se escribirán 2 bytes al archivo.  El número quedaría de la siguiente forma: |

|  |
| --- |
| 0b1011001001101110 |

|  |
| --- |
| Normalmente el hecho de trabajar con archivos binarios esta asociado con el uso de estructuras, dado que la forma mas simple de trabajar es cargar una estructura para luego escribirla en el archivo y de esa forma guardar los datos.  Si bien se puede trabajar con archivos binarios y no usar estructuras, el hecho de utilizarlas permite tener un programa mejor armado y mas consistente. |

* + 1. Trabajo sobre archivos

|  |
| --- |
| En C, todas las operaciones que se realizan sobre archivos son hechas a travez de funciones.  Básicamente existen 2 categorías de funciones para trabajar con archivos y son las que usan “buffer” y las que acceden directamente al archivo.  Durante el transcurso del apunte se usarán solamente la primera categoría, o sea aquellas que usan “buffer”.  El hecho de utilizar un buffer significa que no se tiene acceso directo al archivo y que cualquier operación que se desee realizar (lectura o escritura) va a ser hecha sobre el buffer.  Cuando el buffer se llena o se vacía se actualizan los datos desde y hacia el archivo.  Algunas de las funciones usadas para trabajar con archivos son: |

|  |
| --- |
| **int** fclose(FILE \*fp);    **int** fcloseall(**void**);    **int** fread(**void** \*memoria,**int** num,**int** cont,FILE \*fp);  **int** fwrite(**void** \*memoria,**int** num,**int** cont,FILE \*fp);  **int** fclose(FILE \*f);  **int** feof(FILE \*f);    **int** ferror(FILE \*fp);    **void** rewind(FILE \*fp);    **int** fseek(FILE \*fp,**long** num,**int** origen);    **long** ftell(FILE \*f);    **int** getc(FILE \*fp);    **int** putc(**int** ch,FILE \*fp);    **char**\* fgets(**char** \*str,**int** n,FILE \*fp);    **int** fputs(**const char** \*str,FILE \*fp);    **int** fscanf(FILE \*fp,**const char** \*formato[ ,dirección,...]);    **int** fprintf(FILE \*fp,**const char** \*formato[ ,argumento,...]); |

|  |
| --- |
| Se debe tener en cuenta que al usar las funciones sobre archivos estamos trabajando con un intermediario que accede al archivo.  Por lo tanto todos los datos que van al archivo (escritura) y que vienen (lectura) se encuentran en memoria, por lo tanto 'fread()' y 'fwrite()' toman o dejan esos datos en el archivo.    Gráficamente podemos verlo de la siguiente forma. |

|  |
| --- |
|  |

* + 1. El tipo de dato FILE

|  |
| --- |
| Para trabajar con archivos en C, las funciones utilizan un puntero a la estructura FILE.  Dicha estructura se encuentra definida en el archivo <stdio.h> y se detalla a continuación. |

|  |
| --- |
| **typedef struct**{  **int** level; /\* fill/empty level of buffer \*/  **unsigned** flags; /\* File status flags \*/  **char** fd; /\* File descriptor \*/  **unsigned char** hold; /\* Ungetc char if no buffer \*/  **int** bsize; /\* Buffer size \*/  **unsigned char** \_FAR \*buffer; /\* Data transfer buffer \*/  **unsigned char** \_FAR \*curp; /\* Current active pointer \*/  **unsigned** istemp; /\* Temporary file indicator \*/  **short** token; /\* Used for validity checking \*/  } FILE; |

|  |
| --- |
| Entonces para el manejo de archivos es indispensable definir un puntero a la estructura FILE, como, por ejemplo:  FILE \*pArch;  Donde 'pArch' es el puntero a la estructura FILE. |

* + 1. Apertura de un archivo

|  |
| --- |
| Cada vez que se necesite trabajar con un archivo, lo primero que se debe hacer es abrirlo.  Si no se realiza esto no se puede leer ni escribir en el mismo.  La función que permite la apertura del archivo es 'fopen()' y su formato es el siguiente: |

|  |
| --- |
| FILE \* fopen (**const char** \*Nombre\_de\_archivo, **const char** \*Modo); |

|  |
| --- |
| Donde:  **Nombre\_de\_archivo**:  Es una cadena de caracteres que representa el archivo , es decir se pone la ruta y el nombre del archivo |

|  |
| --- |
| **Modo**:  Es una cadena de caracteres que determina el modo en el que será abierto el archivo. |

|  |
| --- |
| Los modos en los que se puede abrir un archivo estan detallados en la siguiente tabla. |

|  |  |
| --- | --- |
| **Modo** | **Detalle** |
| r | Abre un archivo de texto para operaciones de lectura. |
| w | Abre un archivo de texto para operaciones de escritura. |
| a | Abre un archivo de texto para añadir datos. |
| rb | Abre un archivo binario para operaciones de lectura. |
| wb | Abre un archivo binario para operaciones de escritura. |
| ab | Abre un archivo binario para añadir datos. |
| r+b | Abre un archivo binario para operaciones de lectura escritura. |
| w+b | Abre un archivo binario para operaciones de lectura escritura. |
| a+b | Abre un archivo binario para operaciones de lectura escritura. |
| r+ | Abre un archivo de texto para operaciones de lectura escritura. |
| w+ | Abre un archivo de texto para operaciones de lectura escritura. |

|  |
| --- |
| **Modo escritura (w):**  Si se abre un archivo para operaciones de escritura ( w , wb , w+b , w+ ) y el archivo no existe se creará, pero si existe todos los datos del archivo serán borrados.  Si el archivo que se desea abrir tiene atributo de solo lectura o el disco esta lleno, etc.  la función *'*fopen()' devuelve error.  Cada vez que se abra un archivo en este modo, el indicador de posición se encuentra al comienzo del archivo.  Si se abre un archivo en el modo **“ w ”** o **“ wb ”** solamente se puede escribir.  Si se intenta leer datos del archivo no va a aparecer ningún error, simplemente lo que aparezca como dato leido no va a reflejar la realidad. |

|  |
| --- |
| **Modo lectura (r):**  Si se abre un archivo para operaciones de lectura ( r , rb , r+b , r+ ) , si el archivo no existe la función'fopen()' devuelve error.  Cada vez que se abra un archivo en este modo, el indicador de posición se encuentra al comienzo del archivo.  Si se abre un archivo en el modo **“ r ”** o **“ rb ”** solamente se pueden realizar lecturas.  No tiene ningún efecto realizar operaciones de escritura sobre al archivo, es decir por mas que se intente escribir sobre el archivo no se va a poder. |

|  |
| --- |
| **Modo append (a):**  Si se abre un archivo para agregar datos ( a , ab , a+b , a+ ) y el archivo no existe se creará, caso contrario el indicador de posición del archivo queda posicionado al final del mismo de forma de poder agregar datos. |

|  |
| --- |
| Cada vez que se agregan datos se hace al final del archivo.  Con el modo append no tienen efecto las operaciones de desplazamiento a traves del archivo.  **Valor retornado:**  Si el archivo es abierto exitosamente, la función devuelve un puntero a la estructura FILE asociada al archivo. En caso de detectarse un error devuelve NULL |

|  |
| --- |
| **A tener en cuenta:**   * Se pueden abrir varios archivos al mismo tiempo siempre y cuando exista por lo menos un puntero a FILE para cada uno. * La cantidad de archivos que se pueden abrir al mismo tiempo depende del sistema operativo. * No se debe modificar el valor del puntero devuelto por 'fopen()'.   Veamos un ejemplo: |

|  |
| --- |
| #include <stdio.h>    **void** main (**void**)  {  FILE \*parch;    **if**( ( parch = fopen("banco.dat","rb") ) == NULL )  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }  fclose(parch);  } |

|  |
| --- |
| En el ejemplo se utiliza un ***“ if ”*** para detectar la correcta apertura del archivo.  Se debe recordar que cuando la función 'fopen()' retorna NULL significa que se ha fallado en abrir el archivo en el modo solicitado.  Para crear un archivo por primera vez se debe usar el modo **“ w ”**, pero primero nos debemos asegurar que el archivo no exista, ya que en ese caso el contenido del archivo se borra.  Para contemplar esta situación se modifica levemente el código del ejemplo. |

|  |
| --- |
| #include <stdio.h>  **void** main (**void**)  {  FILE \*parch;    //Se abre en modo lectura  **if**( ( parch = fopen("banco.dat","rb") ) == NULL)  {    //Si el modo anterior dio error el archivo  **if**( ( parch = fopen("banco.dat","wb") ) == NULL)  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }    //no existe, por lo tanto se crea  fclose(parch);  }  } |

|  |
| --- |
| La idea es abrir un archivo para leer, en el caso de que exista se trabaja normalmente, pero si no existe lo abre el segundo 'fopen ()' .  De esta forma nos evitamos borrar un archivo que existe y tiene datos.  En el caso de querer ingresar el nombre del archivo por teclado, el programa se modifica de la siguiente manera: |

|  |
| --- |
| #include <stdio.h>    **void** main (**void**)  {  FILE \*parch;  **char** nombre[20];    printf("\nIngrese el nombre y ruta del archivo que desea abrir: ");  gets(nombre);      **if**( ( parch = fopen(nombre, "rb") ) == NULL)  {  **if**( ( parch = fopen(nombre, "wb") ) == NULL)  {  printf("\nEl archivo %s no puede ser abierto", nombre);  exit (1);  }  fclose(parch);  }  } |

* + 1. Cierre de un archivo

|  |
| --- |
| Todo archivo que se abre debe ser cerrado antes de terminar el programa. El terminar el programa sin cerrar el o los archivos puede causar pérdida de datos.  La función 'fclose()' es la que se encarga de cerrar un archivo. El formato de la función es el siguiente:  **int** fclose ( FILE\* pArch );  Donde 'pArch' es el puntero a la estructura FILE asociada con el archivo que se desea cerrar.  **Valor retornado:**  Si el archivo es cerrado exitosamente se retorna un (0), en caso contrario se devuelve (-1).  Veamos un ejemplo: |

|  |
| --- |
| **void** main (**void**)  {  FILE \*parch;      //Se abre en modo lectura  **if**( ( parch = fopen("banco.dat","rb") ) == NULL)  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }  //Se cierra el archivo  **if**( ( fclose(parch) ) == -1)  {  printf("\nNo se pudo cerrar el archivo");  }  **else**  {  printf("\nEl archivo se cerro exitosamente");  }  } |

|  |
| --- |
| En el ejemplo se chequea que se haya cerrado correctamente el archivo. En el caso de abrir mas de un archivo, antes de terminar el programa se deben cerrar.  Existe una función llamada fcloseall() que cierra todos los archivos que se encuentran abiertos.  El formato de la función es:  **int** fcloseall ( **void** );  Si la operación es exitosa retorna la cantidad de archivos que se cerraron, en caso contrario devuelve  (-1).    Si modificamos el ejemplo anterior: |

|  |
| --- |
| **void** main (**void**)  {  FILE \*parch1, \*parch2;    //Se abre en modo lectura  **if**( ( parch1 = fopen("banco.dat","rb") ) == NULL)  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }  //Se abre en modo escritura  **if**( ( parch2 = fopen("result.dat","wb") ) == NULL)  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }    fcloseall();  } |

* + 1. Escritura de un archivo

|  |
| --- |
| Una vez que el archivo se encuentra abierto se puede empezar a trabajar para leer o escribir.  La función utilizada para realizar la escritura es fwrite.  Esta función sirve para escribir archivos de texto o binarios.  El prototipo de la función es el siguiente: |

|  |
| --- |
| **int** fwrite ( **void** \* origen, **size\_t** tamaño, **size\_t** cantidad, FILE \*arch); |

|  |
| --- |
| Donde:   * **origen:** Es un puntero al lugar desde donde se obtienen los datos para escribir en el archivo. * **tamaño:** Es el tamaño en bytes del dato que se va a escribir. * **cantidad:** Es la cantidad de datos de longitud tamaño que se van a escribir. * **arch:** Es el puntero a FILE asociado al archivo.   Nota: **size\_t** es un unsigned int definido en stdio.h |

|  |
| --- |
| **Valor retornado:**  Devuelve el número de datos escritos (cantidad).  Si el valor retornado es menor al que se indicó por medio de la variable cantidad, significa que hubo un error en la escritura.  La función fwrite() toma cantidad de datos de longitud tamaño desde la dirección ‘origen’ y los escribe en el archivo asociado al puntero ‘arch’ comenzando desde la posición actual del indicador de posición del archivo.  Una vez que se completó la operación de escritura el indicador de posición es actualizado (queda apuntando al lugar donde se puede escribir el próximo dato).  Ejemplo:  - Si queremos escribir en un archivo de texto. |

|  |
| --- |
| FILE \*parch;  **char** texto[ ] = "Prueba de escritura";  **int** cantidad,  **int** longitud;  //Se abre en modo escritura  **if**( ( parch = fopen("prueba.txt","w") ) == NULL)  {  printf("\nEl archivo no puede ser abierto");  exit (1);  }  longitud = strlen (texto);  //Se escribe al archivo  cantidad = fwrite ( texto, **sizeof** ( **char** ), longi, parch );  **if** ( cant < longitud )  {  printf("\nError al escribir el archivo");  }  **else**  {  printf("\nSe escribieron %d caracteres", cant);  }  fclose(parch); |

|  |
| --- |
| -Si se trata de un archivo binario el programa será el siguiente: |

|  |
| --- |
| **typedef struct**{  **char** nombre[10];  **int** edad;  }a;  **void** main (**void**)  {  FILE \*bin;  a pers;    bin = fopen("bin.dat", "wb");    printf("\nIngrese el nombre: ");  gets( pers.nombre );    printf("Ingrese la edad: ");  scanf("%d", &pers.edad);    fflush(stdin);  fwrite( &pers, **sizeof**(pers), 1, bin );    fclose(bin);  } |

|  |
| --- |
| Antes de escribir el archivo se debe cargar el dato que se desea guardar, en nuestro caso debemos cargar la esctructura.  Una vez que se cargaron todos los campos se llama a ‘fwrite’ y se le pasa la dirección de comienzo de la estructura (&pers), el tamaño en bytes de la estructura (se puede escribir directamente o usar **sizeof**), la cantidad de estructuras que se van a escribir (se cargó solo una por lo tanto se escribe una) y finalmente el puntero que hace referencia al archivo.  En el caso que no se desee escribir la estructura entera se deberán hacer 2 fwrite, uno para la edad y el otro para el nombre. |

* + 1. Lectura de un archivo

|  |
| --- |
| Para realizar la lectura de un archivo se utiliza la función fread que tiene el siguiente prototipo |

|  |
| --- |
| **int** fread ( **void** \* destino, **size\_t** tamaño, **size\_t** cantidad, FILE \*arch ); |

|  |
| --- |
| Donde:   * **destino:** Es un puntero al lugar donde se va a dejar el dato leido con fread. * **tamaño:** Es el tamaño en bytes del dato a leer. * **cantidad:** Es la cantidad de elementos de longitud tamaño que se van a leer. * **arch:** Es el puntero a la estructura FILE asociada al archivo desde el que se va a leer. |

|  |
| --- |
| **Valor retornado:**  Devuelve el número de datos leidos (cantidad). Si el valor retornado es menor al que se indicó por medio de la variable cantidad, significa que hubo un error en la lectura o que se llegó al final de archivo.  La función ‘fread’ lee desde el archivo referenciado por ‘arch’ a partir de la posición actual del indicador de posición, cantidad de elementos de longitud tamaño y deja los elementos leidos en la dirección de memoria indicada por destino.  Una vez que se completó la operación de lectura se actualiza automáticamente el indicador de posición del archivo.  A diferencia de lo que ocurre en la escritura, se debe verificar que se realice la lectura mientras no se haya llegado al final del archivo.  Esta operación se realiza por medio de la función ‘feof’.  Ejemplo de lectura de archivo binario: |

|  |
| --- |
| **typedef struct**{  **char** nombre[10];  **int** edad;  }a;  **void** main (**void**)  {  FILE \*bin;  a pers;    **int** cant;    **if** ( (bin = fopen("bin.dat","rb") ) == NULL)  {  printf("No se pudo abrir el archivo");  exit(1);  }  **while**( !feof(bin) )  {  cant = fread( &pers, **sizeof**(pers), 1, bin);  **if**(cant!=1)  {  **if**( feof(bin) ){  **break**;  }  **else**{  error("No leyo el ultimo registro");  **break**;  }  }  printf("\n%s\t%d", pers.nombre, pers.edad);  }    fclose(bin);  getch();  } |

|  |
| --- |
| Es por esto que cuando se entra al ***“ if ”*** que verifica la cantidad, debemos preguntar si se llego al final del archivo.  La función feof determina si se ha llegado al final de el archivo , el prototipo es:  **int** feof (FILE\* arch );  Donde ‘arch’ es el puntero a la estructura FILE asociada con el archivo.  El valor retornado por la función es (0) si no se llegó al final del archivo y distinto de cero (!= 0) si se llegó al final del archivo. |

* + 1. Búsqueda y modificación

|  |
| --- |
| En la lectura y escritura de archivos el indicador de posición se actualiza automáticamente, pero existen casos, por ejemplo, en las búsquedas y modificaciones sobre archivos, en los cuales se necesita mover el indicador de posición a algún lugar en particular.  Para ello se cuenta con 2 funciones que permiten realizar tal operación, ellas son ‘fseek’ y ‘rewind’.  Por otra parte, se cuenta con una función que permite obtener el lugar donde se encuentra el indicador de posición del archivo, esa función es ‘ftell’.  A continuación se detallan las 3 funciones. |

* + - 1. rewind()

|  |
| --- |
| Esta función permite llevar el indicador de posición al comienzo del archivo.  El prototipo es el siguiente:    **void** rewind (FILE\* arch);  La función rewind ubica el indicador de posición del archivo referenciado por el puntero arch al principio y limpia los indicadores de fin de archivo y error que se encuentran en la estructura FILE.  Si se utiliza fread luego de ejecutar rewind, se podra leer el archivo desde el comienzo como cuando se abre con fopen. |

* + - 1. fseek()

|  |
| --- |
| Esta función permite desplazar el indicador de posición del archivo a la posición que se le indique.  El prototipo es: |

|  |
| --- |
| **int** fseek ( FILE \*arch , **long** desplazamiento , **int** origen); |

|  |
| --- |
| **Donde:**   * **arch:** Puntero a la estructura FILE asociada con el archivo. * **desplazamiento:** Es la cantidad de bytes que se desplazará el indicador de posición a partir de origen. * **origen:** Es una constante que determina el punto de referencia a partir del cuál se realiza el desplazamiento. |

|  |
| --- |
| Los valores que se le pueden dar a origen figuran en la siguiente tabla:  Dichos valores se encuentran definidos en stdio.h. |

|  |  |
| --- | --- |
| SEEK\_SET | A partir del comienzo del archivo. |
| SEEK\_CUR | A partir de la posición actual del archivo. |
| SEEK\_END | A partir de el final del archivo. |

|  |
| --- |
| **Valor retornado:**  Si la operación es exitosa devuelve cero , caso contrario retorna un valor distinto de cero.  La función fseek mueve el indicador de posición del archivo desplazamiento bytes a partir de la posición indicada por origen. |

|  |
| --- |
| **Ejemplos:**  fseek ( ptr , 0L, SEEK\_SET );  Mueve el indicador de posición al comienzo del archivo. El origen es SEEK\_SET que indica el comienzo del archivo y se desplaza 0 bytes , por lo tanto queda al principio. Es aconsejable que cuando se desee llevar el indicador de posición al comienzo del archivo se utilice rewind ya que fseek no limpia el indicador de error ni el de fin de archivo , por lo tanto cuando en determinada situación se use fseek no va a dar los resultados esperados.  fseek ( ptr , 0L , SEEK\_END );  Mueve el indicador de posición al final del archivo. El origen es SEEK\_END que indica el final del archivo y a partir de alli se desplaza 0 bytes, por lo tanto esta al final del archivo. Si se desea en algún momento agregar datos , simplemente se debe usar esta función para enviar el indicador de posición al final del archivo.  fseek ( ptr , 20L , SEEK\_SET );  Mueve el indicador de posición 20 bytes a partir del comienzo del archivo.  fseek ( ptr , (**long**) (-1)\***sizeof** (**struct** x) , SEEK\_CUR );  Mueve el indicador de posición una estructura para atras a partir de la posición actual. Normalmente esta forma se utiliza cuando se estan editando datos del archivo. Al realizar una búsqueda se va leyendo cada uno de los datos del archivo por medio de fread, pero cuando encontramos el dato el indicador de posición del archivo esta en el dato siguiente al que queremos modificar, con lo cual al hacer fwrite para escribirlo se modificará otro dato. Por lo tanto antes de escribir se debe mover el indicador de posición una estructura para atrás. |

* + - 1. ftell()

|  |
| --- |
| La función ftell me permite obtener la posición actual del indicador de posición.  El prototipo es el siguiente: |

|  |
| --- |
| **long** ftell ( FILE \* arch ); |

|  |
| --- |
| Donde arch es el puntero a la estructura FILE asociada al archivo.  **Valor retornado:**  Si la operación es exitosa devuelve la cantidad de bytes que hay desde el commienzo del archivo hasta el lugar en que se encuentra el indicador de posición del archivo, en caso contrario devuelve (-1 **l**) *(-1 como tipo long).*  **Ejemplo**:  -Obtener el tamaño de un archivo en bytes. |

|  |
| --- |
| **void** main (**void**)  {  FILE \*bin;  **long int** cant;  **if** ( ( bin = fopen("bin.dat","rb") ) == NULL)  {  printf("No se pudo abrir el archivo");  exit(1);  }  //Se envía la posición al final del archivo  fseek ( bin, 0L, SEEK\_END );  cant = ftell (bin);  printf("\nEl archivo tiene %ld bytes", cant);  fclose(bin);  getch();  } |

*Ejercicios y videos*

[VIDEO PARTE 1](https://youtu.be/IWPgUEKq57A)

|  |
| --- |
| **Ejercicio 1:**   1. Crear una struct con dos campos de cadena de caracteres, uno llamado 'key' y otro llamado 'value'.   struct S\_Data  {  char\*key;  char\*value; };  Hacer una función que reciba el nombre de un archivo y un array de estas estructuras.  La función leerá un archivo de texto con el siguiente formato:  clave = valor  clave = valor  La función deberá leer cada línea del archivo y separar por el caracter **'='** la clave y el valor, y deberá cargarlo en una struct del array, generando de forma dinámica las variables para almacenar ambas palabras, y luego guardando los punteros en la struct.  Prototipo de la función:  int loadDataFile(char\* fileName, struct S\_Data\* array, int arrayLen);  La función devolverá la cantidad de items que se cargaron en el array, y nunca se superará el tamaño  del array |

1. Creación de una lista dinámica
   1. Generación dinámica de memoria

|  |
| --- |
| La función **malloc()** nos permite obtener un espacio de memoria en tiempo de ejecución.  Cuando el programa se ejecuta y se llama a esta función, el sistema operativo le asigna la porción de memoria solicitada al programa.  Este comportamiento nos permitirá la generación dinámica de variables.  En el siguiente ejemplo definimos un array de una cantidad determinada de variables del tipo **int**.  **int** numeros***[***5***];***  Si necesitamos que el tamaño del array sea definido en tiempo de ejecución, por ejemplo, si le preguntamos al usuario cuántos números va a ingresar, y luego pretendemos guardar los números que ingresa en dicho array esta solución “estática” de definir un array de 5 elementos, no nos serviría, ya que si el usuario decide ingresar 6 o más números, no habría posibilidad de almacenarlos.  El siguiente ejemplo nos permitiría solucionar el problema:  **int** cantidad ***=*** pedirCantidadAlUsuario***();***  **int*\**** numeros***;***  numeros ***= (*int*\*)*** malloc***(* sizeof*(*int*) \**** cantidad ***);***  De esta manera reservamos espacio para una cantidad de variables del tipo **int**, según la variable '**cantidad**', la cual se carga con el número que devuelve una supuesta función que le pide ingresar un número al usuario.  Es importante destacar que no podemos hacer lo siguiente:  **int** numeros***[***cantidad***];***  Ya que las definiciones de arrays estáticos deben realizarse con un valor conocido en tiempo de compilación, y el valor de la variable cantidad se conoce en tiempo de ejecución. |

* 1. Creación de tipos de datos particulares

|  |
| --- |
| La generación de tipos de datos particulares se realiza mediante “estructuras”, en las cuales se pueden definir distintos campos, indicando su tipo y nombre, los cuales en conjunto forman un nuevo tipo de dato que el programador no tenía en el lenguaje puro (**int**, **double**, **char**, etc.)  En el siguiente ejemplo se observa la creación del tipo de dato **'Persona'** el cual está compuesto por un nombre y una edad  **struct** S\_Persona  ***{***  **int** edad***;***  **char** nombre***[***20***];***  ***}***Persona***;***  Cuando se define de forma estática una variable del tipo **'Persona'**, dicha variable ocupa en memoria la cantidad de bytes que ocupa un **int** más **20** **char**.  Para generar de manera dinámica una variable del tipo **'Persona'**, simplemente cambiamos la manera de calcular la cantidad de bytes que le pasamos a la función **malloc()**. |

* 1. Utilización de memoria dinámica para la creación de tipos de datos

|  |
| --- |
| Un vez más, utilizamos memoria dinámica cuando no sabemos la cantidad de variables que vamos a crear de antemano, sino que dicho valor se define en tiempo de ejecución. Imaginemos un programa donde se le pide al usuario que ingrese un nombre y una edad, de forma indefinida hasta que decida salir, si lo planteamos sin la utilización de memoria dinámica:  **do**  ***{***  Persona persona***;***  preguntarNombre***( &***persona***->***nombre ***);***  persona***.***edad ***=*** preguntarEdad***();***  // aquí se debe hacer algo con la variable Persona creada,  // ya que en el  // próximo loop, se pisarían los datos y los anteriores  // se perderían.  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);***  Para solucionar el problema, podemos tener un array estático de variables **'Persona'** definido con anterioridad:  Persona lista***[***20***];***  De esta manera, después de cargar el nombre y la edad, podemos copiar la variable **'persona'** a un ítem del array llamado **'lista'** e incrementar el índice.  lista***[***index***] =*** persona***;***  index***++;***  ¿Qué ocurre cuando el usuario quiere ingresar la persona numero 21?  no existirá lugar para guardar dicha persona, por lo que la solución “estática” al problema, no es una solución después de todo.  La ventaja de la utilización de memoria dinámica nos permite crear la variable **'persona'** en cada iteración del bucle, de modo que a medida que las necesitamos, las iremos creando. Necesitaremos crear de forma dinámica la variable **'persona'** dentro del bucle, y también el array llamado **'lista'**, el cual tendrá un valor inicial, y luego haremos que crezca su tamaño mediante la función **realloc()**. |

* 1. Solución al problema del “array estático” mediante “memoria dinámica”

|  |
| --- |
| Comenzaremos creando el array de forma dinámica:  **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\**** lista ***= (***Persona***\*)*** malloc***(* sizeof*(***Persona***) \**** size ***);***  **do**  ***{***  Persona persona***;***  preguntarNombre***( &***persona***->***nombre ***);***  persona***.***edad ***=*** preguntarEdad***();***  lista***[***index***] =*** persona***;***  index***++;***  **if*(***index ***>=*** size***)***  ***{***  // incrementamos el tamaño del array  size ***+=*** 10***;***  lista ***=*** realloc***(*** lista***,* sizeof*(***Persona***) \**** size ***);***  ***}***  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);*** |

* + 1. Desventajas de este modelo

|  |
| --- |
| Cuando incrementamos el tamaño de la lista, estamos desperdiciando mucho espacio, ya que lo incrementamos en 10 (se podría haber multiplicado por dos el tamaño, o usar otro algoritmo de crecimiento) por lo que generamos 10 posiciones del tamaño del tipo de dato **'Persona'**, el cual en nuestro caso tiene la edad (4 bytes) y el nombre (**20** **bytes**), por lo que desperdiciamos mucho espacio sin saber si se va a utilizar o no.  La solución a este problema, es no tener una lista en donde cada ítem sea una variable del tipo **'Persona'**, sino tener una lista donde cada ítem sea un “puntero” a una variable del tipo **'Persona'**, es decir, un array de punteros. Como cada puntero ocupa 4 bytes (en arquitecturas de 32 bits) o 8 bytes (en arquitecturas de 64 bits), cada ítem ocupará menos espacio que si reservamos el valor para guardar variables **'Persona'**.  Gráfico comparativo lista de variables y lista de punteros |

|  |  |
| --- | --- |
|  |  |
| *Array de punteros* | *Array de tipos Persona* |

* 1. Solución al problema con lista de punteros

|  |
| --- |
| Modificaremos el ejemplo anterior, para crear en “forma dinámica” una lista de “punteros” del tipo **Persona\***: |

|  |
| --- |
| **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\*\**** lista ***= (***Persona***\*\*)*** malloc***(*sizeof*(***Persona***\*) \**** size***);***  **do**  ***{***  Persona persona***;***  preguntarNombre***( &***persona***->***nombre ***);***  persona***.***edad ***=*** preguntarEdad***();***  lista***[***index***] = &***persona***;*** // Ver explicacion sobre esta linea  index***++;***  **if*(***index ***>=*** size***)***  ***{***  // incrementamos el tamaño del array  size ***+=*** 10***;***  lista ***=*** realloc***(***lista***,* sizeof*(***Persona***\*) \**** size***);***  ***}***  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);*** |

|  |
| --- |
| Como se observa en el ejemplo, ahora nuestra lista es de “punteros” a variables del tipo **Persona**. Luego de cargar el nombre y la edad en la variable auxiliar **'persona'**, almacenamos el puntero a esta variable dentro de la lista de punteros:  lista***[***index***] = &***persona***;***  Es importante destacar, que **esta solución no es correcta**, debido a que la variable auxiliar **'persona'** tiene siempre la misma dirección y en las diferentes iteraciones se asignan diferentes valores a sus campos. Si se deja el código como en el ejemplo, la lista se cargaría con punteros hacia la misma posición de memoria, donde se encuentra nuestra variable auxiliar, y los sucesivos datos que son cargados se perderían.  Para solucionar este problema, debemos crear un espacio de memoria para una variable **Persona**, cada vez que el usuario ingresa un nuevo ítem en la lista.  Esto es posible mediante el uso de **malloc()**:  **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\*\**** lista ***= (***Persona***\*\*)*** malloc***(*sizeof*(***Persona***\*) \**** size***);***  **do**  ***{***  Persona***\**** persona ***= (***Persona***\*)*** malloc***(* sizeof*(***Persona***) );***  preguntarNombre***(*** persona***->***nombre ***);***  persona***->***edad ***=*** preguntarEdad***();***  lista***[***index***] =*** persona***;***  index***++;***  **if*(***index ***>=*** size***)***  ***{***  // incrementamos el tamaño del array  size ***+=*** 10***;***  lista ***=*** realloc***(***lista***,* sizeof*(***Persona***\*) \**** size***);***  ***}***  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);***  Ahora la variable auxiliar no es una variable del tipo **Persona**, sino un puntero a dicha variable, la cual se genera reservando memoria en forma dinámica. Luego de cargar los datos en dicha variable, se copia el puntero a la variable, a la lista. En la próxima iteración, se generará otro espacio de memoria para una nueva variable del tipo **Persona**, para guardar los nuevos datos ingresados, por lo que no corremos riesgo de que se pisen con datos anteriores.  Gráfico indicando un array de punteros del tipo **Persona**, donde cada ítem del array apunta a un espacio de memoria del tamaño de una variable del tipo Persona creado con **malloc()**. |

|  |
| --- |
|  |

*Ejercicios y videos*

[SOLUCION EJERCICIO 1](https://youtu.be/Ar8OPMmzYao?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 2](https://youtu.be/1C7nohjt0BU?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 3](https://youtu.be/j6DunBAsBv8?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

**Ejercicio 1:**

1. Ingresar datos de 1 persona e imprimirlos.

Crear la estructura "Persona":

nombre[32];

edad;

Si se ingresa 'salir' como nombre, se debe salir del programa,

de lo contrario se pedirán los datos de nuevo.

1. Modificar el ejercicio anterior para guardar las personas en un array estatico e imprimirlas al salir.

**Ejercicio 2:**

1. Tomar el ejercicio de la clase 20 y cambiar el array estático por uno dinámico usando malloc.

**Ejercicio 3:**

1. Tomar el ejercicio de la clase 21 y en lugar de crear un array de personas, crear uno de punteros a estructuras "Person".
2. Creación de una biblioteca para el manejo de 1 (UNA) lista dinámica

|  |
| --- |
| Trataremos de encapsular el manejo de nuestro tipo de dato **Persona** mediante la utilización de un archivo **'.c'** y otro archivo **'.h'** además del utilizado por la función **main()**. |

* 1. Creación y utilización de la biblioteca

|  |
| --- |
| Crearemos el archivo **Persona.h**, en donde colocaremos por el momento, la definición de la estructura con el tipo de dato **'Persona'**:  Persona.h  **struct** S\_Persona  ***{***  **int** edad***;***  **char** nombre***[***20***];***  ***}*** Persona***;***  En nuestro programa, vemos que es común la generación dinámica de variables del tipo **'Persona'**.  Por lo que escribiremos en el archivo **Persona.c**, una función que se encargue de crear en forma dinámica dicha variable, y devolver un puntero hacia la misma.  Llamaremos a esta función **newPersona()**.  Persona.c  Persona***\**** persona\_newPersona***(*void*)***  ***{***  Persona***\**** persona ***= (***Persona***\*)*** malloc***(* sizeof*(***Persona***) );***  **return** persona***;***  ***}***  Como se observa en la definición, le agregamos al nombre de la función el prefijo “persona\_” haremos esto con todas las funciones que se encuentren dentro de este archivo, de modo de agrupar la funcionalidad de “todo lo que tiene que ver con personas” dentro de este archivo, y para una fácil utilización de las funciones, ya que el programador que utilice nuestra biblioteca, sabrá que todas las funciones comienzan con “persona\_x”  Agregamos el prototipo de esta función en el archivo **Persona.h**:  Persona.h  **struct** S\_Persona  ***{***  **int** edad***;***  **char** nombre***[***20***];***  ***}*** Persona***;***  Persona***\**** persona\_newPersona***(* void *);*** |

|  |
| --- |
| Ahora estamos en condiciones de modificar nuestro programa para utilizar la función:  #include "Persona.h"  **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\*\**** lista ***= (***Persona***\*\*)***malloc***(* sizeof*(***Persona***\*) \**** size ***);***  **do**  ***{***  Persona***\**** persona ***=*** persona\_newPersona***();***    preguntarNombre***(*** persona***->***nombre ***);***  persona***->***edad ***=*** preguntarEdad***();***    lista***[***index***] =*** persona***;***  index***++;***  **if*(***index ***>=*** size***)***  ***{***  // incrementamos el tamaÃ±o del array  size ***+=*** 10***;***  lista ***=*** realloc***(***lista***,* sizeof*(***Persona***\*) \**** size***);***  ***}***  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);*** |

* + 1. Validación de campos

|  |
| --- |
| En muchas ocasiones, no deberíamos dejar que uno o algunos de los campos de nuestra variable **'Persona'**, se cargue con un valor fuera de rango, por ejemplo, la edad debe ser un número mayor a 0.  Pero en nuestro programa, debido a la asignación directa:  persona***->***edad ***=*** preguntarEdad***();***  Si la función **preguntarEdad()** no realiza la validación, el valor cargado en el campo podría ser inválido.  Una manera de resolver el problema sería:  **int** edadAux ***=*** preguntarEdad***();***  **if*(*** edadAux ***>*** 0***)***  ***{***  persona***->***edad ***=*** edadAux***;***  ***}***  **else**  ***{***  printf***(***"La edad no es valida"***);***  ***}***  Trataremos de “migrar” el deber y la tarea de validar, a una función que asigne un valor al campo **'edad'** de la variable **Persona**.  Como esto está muy relacionado con el tipo de dato **Persona**, agregaremos esta función en nuestro archivo **Persona.c**, y la llamaremos **persona\_setEdad()**  Escribiremos la función de modo tal que le pasaremos el puntero a la variable **Persona** a la cual queremos asignarle el campo **'edad'**, y el posible valor a ser cargado (en el caso de ser correcto)  **int** persona\_setEdad***(*** Persona***\**** pPersona***,* int** edad ***)***  ***{***  **if*(***edad ***>*** 0***)***  ***{***  pPersona***->***edad ***=*** edad***;***  **return** 0***;*** // OK  ***}***  **return** 1***;*** // error  ***}***  No olvidemos colocar el prototipo de esta función en **Persona.h** |

|  |
| --- |
| Ahora podemos escribir nuestro programa de la siguiente manera:  #include "Persona.h"  **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\*\**** lista ***= (***Persona***\*\*)***malloc ***(* sizeof*(***Persona***\*) \**** size ***);***  **do**  ***{***  Persona***\**** persona ***=*** persona\_newPersona***();***  preguntarNombre***(*** persona***->***nombre ***);***  **int** edadAux ***=*** preguntarEdad***();***  **if*(*** persona\_setEdad***(***persona***,*** edadAux***) )***  ***{***  printf***(***"La edad no es valida"***);***  ***}***  ***...***  De manera similar, podemos validar el campo del nombre, chequeando que posea más de 3 caracteres:  **int** persona\_setName***(***Persona***\**** pPersona***,* char*\**** pName***)***  ***{***  **if*(*** strlen***(***pName***) >*** 3 ***)***  ***{***  strcpy***(*** pPersona***->***nombre***,*** pName ***);***  **return** 0***;***  ***}***  **return** 1***;***  ***}***  No olvidemos colocar el prototipo de esta función en **Persona.h** |

Implementando la función en nuestro programa

|  |
| --- |
| #include "Persona.h"  **int** size ***=*** 10***;***  **int** index ***=*** 0***;***  Persona***\*\**** lista ***= (***Persona***\*\*)***malloc***(* sizeof*(***Persona***\*) \**** size ***);***    **do**  ***{***  Persona***\**** persona ***=*** persona\_newPersona***();***    **char** nombreAux***[***20***];***  preguntarNombre***(***nombreAux***);***  **if*(*** persona\_setName***(***persona***,*** nombreAux***) )***  ***{***  printf***(***"El nombre no es valido"***);***  ***}***  **int** edadAux ***=*** preguntarEdad***();***  **if*(*** persona\_setEdad***(***persona***,*** edadAux***) )***  ***{***  printf***(***"La edad no es valida"***);***  ***}***  ***...*** |

|  |
| --- |
| Nuestra Biblioteca “Persona” ahora nos permite crear una variable del tipo Persona en forma dinámica, y poder cargar sus campos mediante validaciones de contenido.  Imaginemos que queremos imprimir toda la información de cada variable Persona que tenemos, podemos simplemente escribir:  printf***(*** "Nombre:%s - Edad: %d"***,*** persona***->***nombre***,*** persona***->***edad ***);***  en cualquier parte de nuestro programa, pero como esta funcionalidad también es parte, o tiene relación, del tipo de dato Persona, escribiremos una función en nuestro archivo Persona.c que se encargue de imprimir por pantalla dicha información, la ventaja de esta manera, es que si en algún momento agregamos o quitamos un campo a la estructura, sabemos que debemos modificar la función que imprime, que estará en el mismo archivo, y no deberemos recorrer todo nuestro programa buscando llamadas a la función printf. Llamaremos a la función que imprime los datos de una variable Persona persona\_toString  **void** persona\_toString***(*** Persona***\**** pPersona ***)***  ***{***  printf***(*** "Nombre:%s - Edad:%d"***,*** pPersona***->***nombre***,*** pPersona***->***edad ***);***  ***}***  No olvidemos colocar el prototipo de esta función en Persona.h |

* + 1. Lectura de los campos por medio de funciones

|  |
| --- |
| Es una buena práctica leer los valores de los campos a través de funciones, en vez de hacerlo directamente:  **int** edadAux ***=*** persona***->***edad***;***  **int** edadAux ***=*** persona\_getEdad***(*** persona ***);***  Como se observa en el ejemplo, dejamos de acceder al campo **'edad'** directamente desde el “puntero” a la variable **'persona'**, y escribimos una función, en nuestro archivo **Persona.c** que se encarga de dicha tarea.  **int** persona\_getEdad***(***Persona***\**** pPersona***)***  ***{***  **return** pPersona***->***edad***;***  ***}***  **char*\**** persona\_getNombre***(***Persona***\**** pPersona***)***  ***{***  **return** pPersona***->***nombre***;***  ***}***  No olvidemos colocar el prototipo de estas funciones en **Persona.h** |

* + 1. Agrupación de funcionalidad

|  |
| --- |
| Si observamos nuestro “archivo” **Persona.c** y **Persona.h** encontraremos la definición de “un tipo de dato” y una “cierta cantidad de funciones” que interaccionan con este tipo de dato.  **char*\**** persona\_getNombre***(*** Persona***\**** pPersona ***);***  **int** persona\_getEdad ***(*** Persona***\**** pPersona ***);***  **void** persona\_toString ***(*** Persona***\**** pPersona ***);***  **int** persona\_setName ***(*** Persona***\**** pPersona***,* char*\**** pName ***);***  **int** persona\_setEdad ***(*** Persona***\**** pPersona***,* int** edad ***);***  Persona***\**** persona\_newPersona***(* void *);***  Si observamos detenidamente las funciones, descubriremos un patrón que se repite en todas excepto en la función que genera la variable **Persona**: Todas las funciones reciben como primer argumento, un puntero a una variable del tipo **Persona**.  Esta característica no es menor, ya que todas estas funciones, tienen la particularidad de realizar una tarea interaccionando con esta variable persona pasada como argumento.  Podemos decir que cada función de este tipo, tiene asignada una tarea con respecto a la variable **Persona**, por ejemplo asignar el campo edad de la misma, y que tiene acceso a todos los campos de la variable **persona** para poder resolver la tarea en cuestión, ya que le pasamos el puntero a la variable, como primer argumento.  Esto también implica que el resultado de estas funciones, dependerá exclusivamente de la variable **Persona** pasada como argumento, y no dependerá de ninguna otra variable, por ejemplo, si tenemos una variable **Persona 'p1'** la cual tiene cargada en el campo **'edad'** un **23** y otra variable **'p2'** que tiene cargada en el campo **'edad'** un **33**, cuando llamemos a la función **persona\_getEdad()** nos devolverá un número que solo depende de la variable **Persona** que le pasamos como argumento:  **int** buffer***;***  buffer ***=*** persona\_getEdad***(*** p1 ***);***  printf***(*** "Edad de p1: %d"***,*** buffer ***);***  buffer ***=*** persona\_getEdad***(*** p2 ***);***  printf***(*** "Edad de p2: %d"***,*** buffer ***);*** |

* + 1. Pasando el uso de la lista a la biblioteca

|  |
| --- |
| Para pasar la “lista dinámica” que tenemos creada en el **main()** a la biblioteca, nos basta con cambiar de lugar la definición de las tres variables que utilizamos para manejar la lista:  **int** size***;***  **int** index***;***  Persona***\*\**** lista***;***  y definirlas dentro del archivo de la biblioteca, como se observa, falta la creación del array mediante el uso de **malloc()**, por lo que crearemos una función más en la biblioteca que se encargue de la inicialización del array:  **void** persona\_initLista***(*void*)***  ***{***  size ***=*** 10***;***  index ***=*** 0***;***  lista ***= (***Persona***\*\*)***malloc***(* sizeof*(***Persona***\*) \**** size ***);***  ***}***  También cambiaremos de lugar las líneas de código que teníamos dentro del bucle, las cuales se encargaban de copiar el puntero de **Persona** al array y luego verificar si el array se había quedado sin lugar para hacerlo crecer, por lo que crearemos la función “**persona\_addPersona()**” la cual recibirá el puntero a agregar, y hará las tareas mencionadas.  **void** persona\_addPersona***(***Persona***\**** p***)***  ***{***  lista***[***index***] =*** p***;***  index***++;***  **if*(***index ***>=*** size***)***  ***{***  printf***(***"no hay mas lugar, redefinimos el array\r\n"***);***  size ***=*** size ***+*** 10***;***  lista ***= (***Persona***\*\*)***realloc***(*** lista***,* sizeof*(***Persona***\*) \**** size ***);***  ***}***  ***}***  No debemos olvidar definir el prototipo de estas dos funciones en el archivo **Persona.h**  **void** persona\_initLista***(* void *);***  **void** persona\_addPersona***(*** Persona***\**** p ***);***  Con estas dos funciones, hemos migrado el manejo de la lista dinámica a la biblioteca, por lo que nuestro programa en el **main()** quedará mucho más simple de leer y escribir. |

* + 1. Programa final utilizando la biblioteca

|  |
| --- |
| #include "Persona.h"  persona\_initLista***();***  **do**  ***{***  Persona***\**** persona ***=*** persona\_newPersona***();***  **char** nombreAux***[***20***];***  preguntarNombre***(***nombreAux***);***  **if*(*** persona\_setName***(***persona***,***nombreAux***) )***  ***{***  printf***(***"El nombre no es valido"***);***  ***}***  **int** edadAux ***=*** preguntarEdad***();***  **if*(*** persona\_setEdad***(***persona***,*** edadAux***) )***  ***{***  printf***(***"La edad no es valida"***);***  ***}***    persona\_addPersona***(***persona***);***  ***}*while*(***preguntarSalir***() !=*** 'S'***);*** |

*Ejercicios y videos*

[SOLUCION EJERCICIO 1](https://youtu.be/kKuvJEsxDGo?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 2](https://youtu.be/v-j1xhPH2R4?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 3](https://youtu.be/3m9wXd8FmR4?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

**Ejercicio 1:**

1) Tomar el ejercicio de la clase 22 y crear la biblioteca 'People' que pueda manejar una lista de 'Personas' (agregar a la lista personas)

**Ejercicio 2:**

1) Tomar el ejercicio de la clase 23 y agregar a la biblioteca del ejercicio anterior, la posibilidad de borrar un item.

**Ejercicio 3:**

1) Tomar el ejercicio de la clase 24 y agregar a la biblioteca la posibilidad de crear más de una lista.

1. Creación de una biblioteca para el manejo de mas de una (VARIAS) listas dinámicas

|  |
| --- |
| Modificaremos nuestra biblioteca para poder manejar más de una lista, es decir, la biblioteca tendrá una función que me permitirá crear una lista en particular, y luego usar las funciones existentes hasta el momento, sobre la lista generada, dando siempre la posibilidad de crear más de una lista e interactuar con cualquiera de ellas. |

* 1. Definiendo la estructura que representa a la lista

|  |
| --- |
| Hasta el momento la única estructura que utilizamos representa a un ítem de la lista, es decir una “Persona”, definiremos otra estructura que nos represente la lista. Dentro de esta estructura, definiremos los campos que necesitamos para que la lista funcione, que son las 3 variables que por el momento declaramos globales en nuestra biblioteca:  **int** size***;***  **int** index***;***  Persona***\*\**** lista***;***  Borramos estas tres variables globales, y definimos una “**struct**” que representa a la lista con estas tres variables como campos:  **struct** S\_PeopleList  ***{***  **int** size***;***  **int** index***;***  Persona***\*\**** lista***;***  ***};***  **typedef struct** S\_PeopleList PeopleList***;***  Cada vez que creamos una variable de este tipo, tendremos dentro los campos necesarios para manejar el array en forma dinámica utilizando las funciones que ya escribimos.  Deberemos realizarles unos cambios menores que detallaremos a continuación. |

* 1. Creación de una lista

|  |
| --- |
| Modificaremos la función **persona\_initLista()**, ya que ahora esta función no inicializará las variables globales (porque ya no existen) sino que creará una estructura del tipo **'PeopleList'** e inicilizará sus campos:  PeopleList***\**** persona\_initLista***(* void *)***  ***{***  PeopleList***\**** pl ***= (***PeopleList***\*)***malloc***(* sizeof*(***PeopleList***) );***    pl***->***index ***=*** 0***;***  pl***->***size ***=*** 2***;***  pl***->***lista ***= (***Persona***\*\*)***malloc***(* sizeof*(***Persona***\*) \**** pl***->***size ***);***    **return** pl***;***  ***}***  Como se observa en el código, ahora en la inicialización, lo que hacemos es crear una estructura **'PeopleList'** en forma dinámica e inicializamos sus campos de la misma forma que antes inicializabamos las variables globales. |

* 1. Agregar ítems a una lista en particular

|  |
| --- |
| A continuación, modificaremos la función que agrega una persona a la lista, ya que como no existen más las tres variables globales que conformaban “la lista”, ahora la lista debe ser pasada como argumento, por lo que la función quedaría:  **void** persona\_addPersona***(*** PeopleList***\**** pl***,*** Persona***\**** p ***)***  ***{***  pl***->***lista***[***pl***->***index***] =*** p***;***  pl***->***index***++;***    // si no hay mas lugar, pedimos más memoria  // para hacer un array más grande    **if*(*** pl***->***index ***>=*** pl***->***size ***)***  ***{***  printf***(***"no hay mas lugar, redefinimos el array\r\n"***);***  pl***->***size ***+=*** 10***;***  pl***->***lista ***=***  ***(***Persona***\*\*)***realloc***(*** pl***->***lista***,* sizeof*(***Persona***\*) \**** pl***->***size ***);***  ***}***  ***}***  Como se observa en el código, el primer argumento de la función es la lista **'PeopleList'** a la que vamos a agregar un ítem, y el segundo argumento el ítem a agregar, esto nos deja ver que ahora la función es “multi-lista” y no trabaja con una sola lista global como antes, sino que la lista con la que va a operar, se pasa como argumento.  Dentro de la función seguimos haciendo lo mismo que antes, pero ahora en los lugares donde se usaban algunas de las tres variables globales, usaremos el campo de la **struct PeopleList** correspondiente. |

* 1. Programa final

|  |
| --- |
| La única modificación que haremos sobre el programa, es que ahora la función **persona\_initLista()** nos devolverá un puntero a la estructura que representa a la lista, y dicho puntero deberemos pasarlo como argumento en la función **persona\_addPersona()**.  #include "Persona.h"  PeopleList***\**** pl ***=*** persona\_initLista***();***  **do**  ***{***  Persona***\**** persona ***=*** persona\_newPersona***();***  **char** nombreAux***[***20***];***    preguntarNombre***(*** nombreAux ***);***    **if*(*** persona\_setName***(***persona***,*** nombreAux***) )***  ***{***  printf***(***"El nombre no es valido"***);***  ***}***  **int** edadAux ***=*** preguntarEdad***();***    **if*(*** persona\_setEdad***(***persona***,*** edadAux***) )***  ***{***  printf***(***"La edad no es valida"***);***  ***}***  persona\_addPersona***(*** pl***,*** persona ***);***  ***}***  **while*(***preguntarSalir***() !=*** 'S'***);***  **NOTA:**  Es importante aclarar que todos los espacios de memoria reservados deben ser liberados, en el código este material se omitió para simplificar la explicación. |

*Ejercicios y videos*

[SOLUCION EJERCICIO 1](https://youtu.be/kKuvJEsxDGo?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 2](https://youtu.be/v-j1xhPH2R4?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

[SOLUCION EJERCICIO 3](https://youtu.be/3m9wXd8FmR4?list=PLZU3OfSJlsrfKiKElmKxjCsyQQF_1ApuV)

**Ejercicio 1:**

1) Tomar el ejercicio de la clase 22 y crear la biblioteca 'People' que pueda manejar una lista de 'Personas' (agregar a la lista personas)

**Ejercicio 2:**

1) Tomar el ejercicio de la clase 23 y agregar a la biblioteca del ejercicio anterior, la posibilidad de borrar un item.

**Ejercicio 3:**

1) Tomar el ejercicio de la clase 24 y agregar a la biblioteca la posibilidad de crear más de una lista.

1. Trabajo practico N°4

**Trabajo practico N°4**

***Programación I – Laboratorio I.***

***Tecnicatura Superior en Programación.***

***UTN-FRA***

1. Objetivo

|  |
| --- |
| El objetivo del siguiente trabajo es que el alumno sea capaz de demostrar que puede integrar lo aprendido durante la cursada en un caso real.  Los conocimientos necesarios para la realización del TP son los siguientes:   * Manejo de punteros. * Manejo de arrays de punteros. * Manejo de estructuras. * Manejo de memoria dinámica. |

* 1. Etapas del trabajo

|  |
| --- |
| **Etapa 1:**  Se deberá desarrollar una biblioteca **ArrayList.c** y **ArrayList.h** la cual contendrá el tipo de dato **ArrayList**, tal que cumpla con la especificación del documento, con las funciones mínimas requeridas.  **Etapa 2:**  Realizar una aplicación que dé uso del “ArrayList” (usando todas las funciones) y que permita interactuar con estructuras de datos almacenadas en archivos. |

* 1. Condiciones de Entrega

|  |
| --- |
| El trabajo práctico es de carácter individual y cada una de sus entregas (sin excepción) deben ser enviadas en la fecha establecida por los docentes.  Para reducir el uso de papel, la modalidad de las entregas será en forma totalmente digital utilizando como medio un repositorio de github.com, el cual será informado al docente a través de un mensaje en el campus.  Las devolución de los docentes será realizado por el mismo medio. En la misma, se indicarán las observaciones pertinentes, pudiendo solicitarse la realización de algunos cambios si fuera necesario, además de informar si la entrega está aprobada o no. |

* 1. Condiciones de Aprobación

|  |
| --- |
| Se deberá entregar un proyecto de código ANSI C el cual estará compuesto de un programa que utilice la biblioteca **ArrayList** en su totalidad, el mismo deberá contar como mínimo con las funciones obligatorias, todas ellas con su respectiva documentación, y un programa que utilice de manera integral la biblioteca.  En la fecha del segundo parcial se realizará una defensa oral del trabajo por parte del alumno a efectos de determinar si se encuentra o no en condiciones de rendir examen final. |

* 1. Funciones Obligatorias

|  |
| --- |
| * **al\_newArrayList()** * **al\_add()** * **al\_deleteArrayList()** * **al\_len()** * **al\_get()** * **al\_contains()** * **al\_set()** * **al\_remove()** * **al\_clear()** * **al\_push()** * **al\_indexOf()** * **al\_isEmpty()** * **al\_pop()** |

1. Biblioteca ArrayList

|  |
| --- |
| El “ArrayList” es una estructura que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena es **dinámico**, es decir, que no es necesario declarar su tamaño como pasa con los Arrays.  Los ArrayList nos permiten añadir, eliminar y modificar elementos de forma transparente para el programador.  **struct** ArrayList  ***{***  **void*\*\**** pElements***;***  **int** size***;***  **int** reservedSize***;***    **int *(\****add***)();***  **int *(\****len***)();***  **int *(\****contains***)();***  **int *(\****set***)();***  **int *(\****remove***)();***  **int *(\****clear***)();***  **int *(\****push***)();***  **int *(\****indexOf***)();***  **int *(\****isEmpty***)();***  **void*\* (\****get***)();***  **void*\* (\****pop***)();***  **int *(\****containsAll***)();***  **int *(\****sort***)();***    **struct** ArrayList***\* (\**** clone***)();***  **struct** ArrayList***\* (\****subList***)();***    **int *(\****deleteArrayList***)();***    ***}* typedef** ArrayList***;***  Cada función de la biblioteca cuenta con un [Test unitario](https://es.wikipedia.org/wiki/Prueba_unitaria) asociado mediante el cual se podrá verificar el correcto funcionamiento de la misma. |

* 1. Función al\_newArrayList()

|  |
| --- |
| Crea y retorna un nuevo **ArrayList**.  Es el constructor, ya que en él daremos valores iniciales a las variables y asignaremos las funciones a sus punteros.  ArrayList***\**** al\_newArrayList***(*void*)***  ***{***  //...  ***}***  **Ejemplo uso:**  ArrayList***\**** lista***;***    lista ***=*** al\_newArrayList***();***  **Ejemplo de uso del test*:***  startTesting***(***1***);*** |

|  |  |
| --- | --- |
| arrayList.c |  |
| ArrayList***\**** al\_newArrayList***(*void*)***  ***{***  ArrayList***\**** retorno ***=*** NULL***;***  ArrayList***\** this*;***  **void*\**** pElements***;***  **this *= (***ArrayList ***\*)***malloc ***(* sizeof*(***ArrayList***) );***  // puntero a la nueva variable “lista”    **if*(* this *!=*** NULL ***)***  ***{***    pElements ***=*** malloc***(* sizeof*(*void*\*) \**** AL\_INITIAL\_VALUE ***);***  // puntero a la “lista” contenida  // dentro de la nueva varible (lista) |  |
|  | arrayList.h |
| **if*(***pElements ***!=*** NULL***)***  ***{*** | #ifndef \_\_ARRAYLIST  #define \_\_ARRAYLIST  **struct** ArrayList  ***{*** |
| **this*->***pElements ***=*** pElements***;***  **this*->***reservedSize ***=*** AL\_INITIAL\_VALUE***;***  **this*->***size ***=*** 0***;***  **this*->***add ***=*** al\_add***;***  **this*->***len ***=*** al\_len***;***  **this*->***contains ***=*** al\_contains***;***  **this*->***set ***=*** al\_set***;***  **this*->***remove ***=*** al\_remove***;***  **this*->***clear ***=*** al\_clear***;***  **this*->***get ***=*** al\_get***;***  **this*->***clone ***=*** al\_clone***;***  **this*->***push ***=*** al\_push***;***  **this*->***isEmpty ***=*** al\_isEmpty***;***  **this*->***indexOf ***=*** al\_indexOf***;***  **this*->***pop ***=*** al\_pop***;***  **this*->***subList ***=*** al\_subList***;***  **this*->***containsAll ***=*** al\_containsAll***;***  **this*->***deleteArrayList ***=*** al\_deleteArrayList***;***  **this*->***sort ***=*** al\_sort***;***  retorno ***=* this*;***  ***}***  **else**  ***{***  free***(*this*);***  ***}***  ***}***  **return** retorno***;***  ***}*** | **void*\*\**** pElements***;***  **int** reservedSize***;***  **int size*;***  **int *(\****add***)();***  **int *(\****len***)();***  **int *(\****contains***)();***  **int *(\**set*)();***  **int *(\**remove*)();***  **int *(\**clear*)();***  **void*\* (\****get***)();***  **int *(\****push***)();***  **int *(\****indexOf***)();***  **int *(\****isEmpty***)();***  **void*\* (\****pop***)();***  **int *(\****containsAll***)();***  **int *(\**sort*)();***  **struct** ArrayList***\* (\**** clone***)();***  **struct** ArrayList***\* (\****subList***)();***  **int *(\****deleteArrayList***)();***  ***}*typedef** ArrayList***;***  #endif |

* 1. Función al\_add()

|  |
| --- |
| Agrega un elemento al final de “ArrayList”. Verificando que tanto el puntero **'pList'** como **'pElements'**sean distintos de **NULL**.  Si la verificación falla la función retorna (-1) y si tiene éxito (0).  **int** al\_add***(***ArrayList***\**** pList***,* void*\**** pElement***)***  ***{***  //...  ***}***  **Ejemplo uso:**    Persona auxPersona***;***  r ***=*** lista***->***add***(***lista***, &***auxPersona***);***  **Ejemplo de uso del test*:***  startTesting***(***2***);*** |

* 1. Función al\_len()

|  |
| --- |
| Retorna el tamaño del “ArrayList”. Verificando que el puntero **'pList'** sea distinto de **NULL**.  Si la verificación falla la función retorna (-1) y si tiene éxito retorna la longitud del array.  **int** al\_len***(***ArrayList***\**** pList***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  longitud ***=*** lista***->***len***(***lista***);***  **Ejemplo de uso del test*:***  startTesting***(***3***);*** |

* 1. Función al\_contains()

|  |
| --- |
| Comprueba si existe el elemento que se le pasa como parámetro. Verificando que tanto el puntero **'pList'** como **'pElement'** sean distintos de **NULL**. Si la verificación falla la función retorna (-1) , si encuentra el elemento (1) y si no lo encuentra (0).  **int** al\_contains***(***ArrayList***\**** pList***,* void*\**** pElement***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  **if*(***lista***->***contains***(***lista***, &***auxPersona***))***  ***{***  printf ***(***"SI"***);***  ***}***  **Ejemplo de uso del test*:***  startTesting***(***4***);*** |

* 1. Función al\_set()

|  |
| --- |
| Inserta un elemento en el “ArrayList”, en el índice especificado. Verificando que tanto el puntero **'pList'** como **'pElements'**sean distintos de **NULL** y que index sea positivo e inferior al tamaño del array. Si la verificación falla la función retorna (-1) y si tiene éxito (0).  **int** al\_set***(***ArrayList***\**** pList***,* int** index***,* void*\**** pElement***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  Persona auxPersona***;***  r ***=*** lista***->***set***(***lista***,*** 4***, &***auxPersona***);***  **Ejemplo de uso del test*:***  startTesting***(***5***);*** |

* 1. Función al\_remove()

|  |
| --- |
| Elimina un elemento del “ArrayList”, en el índice especificado. Verificando que el puntero **'pList'** sea distinto de NULL y que index sea positivo e inferior al tamaño del array. Si la verificación falla la función retorna (-1) y si tiene éxito (0).  **int** al\_remove***(***ArrayList***\**** pList***,* int** index***);***  ***{***  //...  ***}***  **Ejemplo uso*:***  r ***=*** lista***->***remove***(***lista***,*** 5***);***  **Ejemplo de uso del test*:***  startTesting***(***6***);*** |

* 1. Función al\_clear()

|  |
| --- |
| Borra todos los elementos de ArrayList. Verificando que el puntero pList sea distinto de NULL. Si la verificación falla la función retorna (-1) y si tiene éxito (0).  **int** al\_clear***(***ArrayList***\**** pList***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  r ***=*** lista***->***clear***(***lista***);***  **Ejemplo de uso del test*:***  startTesting***(***7***);*** |

* 1. Función al\_push()

|  |
| --- |
| Desplaza los elementos e inserta en la posición index. Verificando que tanto el puntero pList como pElement sean distintos de NULL y que index sea positivo e inferior al tamaño del array. Si la verificación falla la función retorna (-1) y si tiene éxito (0).  **int** al\_push***(***ArrayList***\**** pList***,* int** index***,* void*\**** pElement***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  Persona auxPersona***;***  r ***=*** lista***->***set***(***lista***,*** 6***, &***auxPersona***);***  **Ejemplo de uso del test*:***  startTesting***(***8***);*** |

* 1. Función al\_indexOf()

|  |
| --- |
| Retorna el índice de la primera aparición de un elemento (element) en el ArrayList. Verificando que tanto el puntero pList como pElement sean distintos de NULL. Si la verificación falla o no encuentra el elemento la función retorna (-1) y si encuentra el elemento retorna su índice.  **int** al\_indexOf***(***ArrayList***\**** pList***,* void*\**** element***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  r ***=*** lista***->***indexOf***(***lista***,&***auxPersona***))***  **Ejemplo de uso del test*:***  startTesting***(***9***);*** |

* 1. Función al\_isEmpty()

|  |
| --- |
| **Retorna cero si contiene elementos y uno si no los tiene.**  **Verificando que el puntero pList sea distinto de NULL.**  **Si la verificación falla la función retorna (-1),**  **si esta vacío ( 1) y**  **si contiene elementos ( 0).**  **int** al\_isEmpty***(***ArrayList***\**** pList***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  **if*(***lista***->***isEmpty***(***lista***))***  ***{***  printf ***(***"Esta vacío"***);***  ***}***  **Ejemplo de uso del test*:***  startTesting***(***10***);*** |

* 1. Función al\_get()

|  |
| --- |
| Retorna un puntero al elemento que se encuentra en el índice especificado.  Verificando que el puntero pList sea distinto de NULL y que index sea positivo e inferior al tamaño del array.  Si la verificación falla la función retorna (NULL) y si tiene éxito retorna el elemento.  **void*\**** al\_get***(***ArrayList***\**** pList***,* int** index***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  Persona***\**** elemento***;***  elemento ***= (***Persona***\*)***lista***->***get***(***lista***,*** 5***);***  **Ejemplo de uso del test*:***  startTesting***(***11***);*** |

* 1. Función al\_pop()

|  |
| --- |
| Retorna un puntero al elemento que se encuentra en el índice especificado y luego lo elimina de la lista.  Verificando que el puntero pList sea distinto de NULL y que index sea positivo e inferior al tamaño del array.  Si la verificación falla la función retorna (NULL) y  si tiene éxito retorna el elemento.  **void*\**** al\_pop***(***ArrayList***\**** pList***,* int** index***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  Persona***\**** elemento***;***  elemento ***= (***Persona***\*)***lista***->***pop***(***lista***,*** 5***);***  **Ejemplo de uso del test*:***  startTesting***(***12***);*** |

* 1. Función al\_containsAll()

|  |
| --- |
| Comprueba si los elementos pasados son contenidos por el ArrayList.  Verificando que tanto el puntero pList como pList2 sean distintos de NULL.  Si la verificación falla o no encuentra el elemento retorna (-1)  si las listas difieren ( 0) y  si ambas listas son iguales retorna ( 1).  **int** al\_containsAll***(***ArrayList***\**** pList***,***ArrayList***\**** pList2***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  **if*(*** lista***->***containsAll***(***lista\_A***,*** lista\_B***) )***  ***{***  printf ***(***"Contienen los mismos elementos"***);***  ***}***  **Ejemplo de uso del test*:***  startTesting***(***13***);*** |

* 1. Función al\_sort()

|  |
| --- |
| Ordena los elementos del array recibiendo como parámetro la función que sera la encargada de determinar que elemento es mas grande que otro y si se debe ordenar de manera ascendente o descendente.  Verificando que tanto el puntero pList como el puntero a la funcion pFunc sean distintos de NULL. Si la verificación falla (-1) caso contrario retorna ( 1).  **int** al\_sort***(***ArrayList***\**** pList***,* int *(\****pFunc***)(*void*\*,*void*\*),* int** order***)***  ***{***  //...  ***}***  **Ejemplo de la función de comparación:**  **int** comparaPersonas***(*void*\**** pPersonA***,*void*\**** pPersonB***)***  ***{***  **if*(((***Persona***\*)***pPersonA***)->***edad ***> ((***Persona***\*)***pPersonB***)->***edad***)***  ***{***  **return** 1***;***  ***}***  **if*(((***Persona***\*)***pPersonA***)->***edad ***< ((***Persona***\*)***pPersonB***)->***edad***)***  ***{***  **return *-***1***;***  ***}***  **return** 0***;***  ***}***  **Ejemplo uso*:***  r ***=*** lista***->***sort***(***lista***,*** comparePersonas***,***1***);***  **Ejemplo de uso del test*:***  startTesting***(***13***);*** |

* 1. Función al\_clone()

|  |
| --- |
| Retorna un nuevo ArrayList copia del ArrayList original. Verificando que el puntero pList sea distinto de NULL. Si la verificación falla la función retorna (NULL) y si tiene éxito retorna el nuevo array.  ArrayList***\**** al\_clone***(***ArrayList***\**** pList***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  ArrayList***\**** arrayClon***;***  arrayClon ***=*** lista***->***clone***(***lista***);***  **Ejemplo de uso del test*:***  startTesting***(***15***);*** |

* 1. Función al\_sublist()

|  |
| --- |
| Retorna un nuevo ArrayList con el subconjunto de elementos.  Verificando que el puntero pList sea distinto de NULL y que tanto el indice 'from' como 'to' sean positivos e inferiores al tamaño del array.  Si la verificación falla la función retorna (NULL) y si tiene éxito retorna el nuevo array.  ArrayList***\**** al\_subList***(***ArrayList***\**** pList***,* int** from***,* int** to***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  Persona***\**** elemento***;***    elemento ***= (***Persona***\*)***lista***->***pop***(***lista***,***5***);***  **Ejemplo de uso del test*:***  startTesting***(***16***);*** |

* 1. Función al\_deleteArrayList()

|  |
| --- |
| Elimina el “ArrayList” . Verificando que el puntero **'pList'** sea distinto de **NULL**.   * si la verificación falla la función retorna (-1) * si esta vacío (1) * si contiene elementos (0)   **int** al\_deleteArrayList***(***ArrayList***\**** pList***)***  ***{***  //...  ***}***  **Ejemplo uso*:***  r ***=*** lista***->***deleteArrayList***(***lista***);***  **Ejemplo de uso del test*:***  startTesting***(***17***);*** |

1. Como realizar la práctica

|  |
| --- |
| Se aclaro anteriormente que cada función de la biblioteca cuenta con un [Test unitario](https://es.wikipedia.org/wiki/Prueba_unitaria) asociado, mediante el cual se podrá verificar su correcto funcionamiento , para poder utilizarlo deberemos seguir los siguientes pasos. |

* 1. Proyecto

|  |
| --- |
| Descargar el proyecto desde el campus. |

|  |
| --- |
|  |

* 1. Donde descomprimir el proyecto

|  |
| --- |
| Descomprimir el proyecto en un directorio en el disco **'** **d:\'** ya que no se encuentra frizado y ante un imprevisto reinicio de la computadora la información no se vera afectada. |

* 1. Abrir el proyecto en **Code::Blocks**

|  |
| --- |
| El proyecto se encuentra dentro del directorio:  Unix: ***utest/examples/example\_4/src***  *Win:* ***utest\examples\example\_4\src***  Dentro de **'src'** encontraran el proyecto de **Code::Blocks** “**utest.cbp**” |

|  |
| --- |
|  |

* 1. El archivo **“main.c”**

|  |
| --- |
| En el archivo main.c, encontraremos las siguientes lineas de código: |

|  |
| --- |
| **int** main***(*void*)***  ***{***  #ifdef TEST  startTesting(1);  //startTesting(2);  //startTesting(3);  //startTesting(4);  //...  //...  //...  //...  //startTesting(17);  #else run();  #endif return 0;  ***}*** |

|  |
| --- |
| Cada una de ellas corresponde a un caso de test, por ejemplo para ejecutar el test numero dos, comentaremos el test numero uno y descomentaremos el numero dos: |

|  |
| --- |
| **int** main***(*void*)***  ***{***  #ifdef TEST  //startTesting(1);  startTesting(2);  //startTesting(3);  //startTesting(4);  //...  //...  //...  //...  //startTesting(17);  #else run();  #endif return 0;  ***}*** |

* 1. Abrir el archivo **“array.c”**

|  |
| --- |
| En este archivo es donde llevaremos adelante la practica, la cual consiste en una primer instancia en desarrollar el código que satisfaga todo aquello que la documentación exige. Cada uno de los casos de test prueba una función según el siguiente esquema: |

|  |  |
| --- | --- |
| **Caso de Test** | **Función Testeada** |
| startTesting(1) | al\_newArrayList |
| startTesting(2) | al\_add |
| startTesting(3) | al\_deleteArrayList |
| startTesting(4) | al\_len |
| startTesting(5) | al\_get |
| startTesting(6) | al\_contains |
| startTesting(7) | al\_set |
| startTesting(8) | al\_remove |
| startTesting(9) | al\_clear |
| startTesting(10) | al\_clone |
| startTesting(11) | al\_push |
| startTesting(12) | al\_indexOf |
| startTesting(13) | al\_isEmpty |
| startTesting(14) | al\_pop |
| startTesting(15) | al\_subList |
| startTesting(16) | al\_containsAll |
| startTesting(17) | al\_sort |

|  |
| --- |
| **Ejemplo:**  Para trabajar en el desarrollo de la funcion **al\_add()** debemos:  A. Descomentar “startTesting***(***4***)***” en el archivo **main*.*c**  //startTesting(1);  startTesting***(***2***);***  //startTesting(4);  //startTesting(4);  B. Borrar el “**return** 0***;”*** que aparece y escribir dentro de la función el código que satisfaga el requerimiento.  **int** al\_add***(*** ArrayList***\**** pList***,* void*\**** pElement ***)***  ***{***  **int** retorno ***= -***1***;*** //solo a modo ilustrativo  //...  //...  **return** retorno***;*** //solo a modo ilustrativo  ***}*** |

* 1. Realizar la prueba

|  |
| --- |
| La idea de tener implementado el test unitario de la funciones radica en permitir verificar el correcto funcionamiento de las mismas, por lo tanto cuando compilemos y ejecutemos nuestro programa obtendremos un informe pormenorizado de cuales son las cosas que fallan en nuestra implementación. |

|  |
| --- |
|  |

|  |
| --- |
| En este caso podemos ver que nuestra implementación de la función initEmployees no supera ninguno de los tests que se realizan. Analicemos en detalle la primer falla. |

|  |
| --- |
|  |

|  |
| --- |
| Indica cual es la prueba que le realiza a la función. |

|  |
| --- |
|  |

|  |
| --- |
| Indica que el test fallo |

|  |
| --- |
|  |

|  |
| --- |
| Da un detalle de la falla y un indicio de como solucionarla |

|  |
| --- |
|  |