

## **Trabajo practico N°4**

***Programación I – Laboratorio I.  
Tecnicatura Superior en Programación.  
UTN-FRA***

**Autores:** *Lic. Mauricio Dávila*

**Revisores:** *Ing. Ernesto Gigliotti*

*Versión : 5*



Esta obra está bajo una [Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional](http://creativecommons.org/licenses/by-sa/4.0/).

## Índice de contenido

1Objetivo .....	3
1.1Etapas del trabajo .....	3
1.2Condiciones de Entrega.....	3
1.3Condiciones de Aprobación .....	3
1.4Funciones Obligatorias.....	3
2Biblioteca ArrayList.....	5
2.1Función al_newArrayList.....	5
2.2Función al_add.....	6
2.3Función al_len.....	6
2.4Función al_contains.....	6
2.5Función al_set.....	6
2.6Función al_remove.....	7
2.7Función al_clear.....	7
2.8Función al_push.....	7
2.9Función al_indexOf.....	8
2.10Función al_isEmpty.....	8
2.11Función al_get.....	8
2.12Función al_pop.....	9
2.13Función al_containsAll.....	9
2.14Función al_sort.....	10
2.15Función al_clone.....	10
2.16Función al_sublist.....	10
2.17Función al_deleteArrayList.....	11
3Como realizar la práctica.....	12
3.1Proyecto.....	12
3.2Donde descomprimir el proyecto.....	12
3.3Abrir el proyecto en Code::Blocks.....	12
3.4El archivo "main.c".....	13
3.5Abrir el archivo "array.c".....	14
3.6Realizar la prueba.....	15

## 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.

### 1.1 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.

### 1.2 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 devoluciones 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.

### 1.3 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.

### 1.4 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`

## 2 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{
    int size;
    void **pElements;
    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](#) asociado mediante el cual se podrá verificar el correcto funcionamiento de la misma.

### 2.1 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);
```

## 2.2 Función al\_add

Agrega un elemento al final de ArrayList. Verificando que tanto el puntero pList como pElement 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);
```

## 2.3 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);
```

## 2.4 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);
```

## 2.5 Función al\_set

Inserta un elemento en el ArrayList, en el índice especificado. 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_set(ArrayList* pList, int index, void* pElement)
{
    //.....
}
```

Ejemplo uso:

```
Persona auxPersona;
r = lista->set(lista, 4, &auxPersona);
```

Ejemplo de uso del test:

```
startTesting(5);
```

## 2.6 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);
```

## 2.7 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);
```

## 2.8 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);
```

## 2.9 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);
```

## 2.10 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);
```

## 2.11 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);
```

## 2.12 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);
```

## 2.13 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 la función 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);
```



### 2.14 Función al\_sort

Ordena los elementos del array recibiendo como parámetro la función que será la encargada de determinar que elemento es más grande que otro y si se debe ordenar de manera ascendente o descendente. Verificando que tanto el puntero pList como el puntero a la función 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 de uso

```
r = lista->sort(lista, comparaPersonas, 1);
```

Ejemplo de uso del test:

```
startTesting(13);
```

### 2.15 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);
```

### 2.16 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 índice '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

```
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);
```

## 2.17 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) y si contiene elementos (0).

```
int al_deleteArrayList(ArrayList* pList)
{
    //.....
}
```

Ejemplo uso:

```
r = lista->deleteArrayList(lista);
```

Ejemplo de uso del test:

```
startTesting(17);
```

### 3 Como realizar la práctica

Se aclaro anteriormente que cada función de la biblioteca cuenta con un [Test unitario](#) asociado, mediante el cual se podrá verificar su correcto funcionamiento , para poder utilizarlo deberemos seguir los siguientes pasos.

#### 3.1 Proyecto

Descargar el proyecto desde el campus.

##### Trabajos Prácticos

La materia cuenta con cuatro trabajos prácticos, las notas obtenidas en los tres primeros serán promediadas a efectos de determinar si el alumno aprueba o no la parte practica de la materia. El cuarto trabajo practico sera defendido por el alumno en la fecha del segundo parcial de Laboratorio. Los trabajos entregados fuera de fecha tendrán como nota máxima un cuatro, a continuación el cronograma de entregas:

Trabajos Prácticos			
Nº	Enunciado	Esqueleto	Fecha de Entrega
1	<a href="#">[Descargar]</a>	<a href="#">[Descargar]</a>	8/4/2016
2	<a href="#">[Descargar]</a>	<a href="#">[Descargar]</a>	29/4/2016
3	<a href="#">[Descargar]</a>	<a href="#">[Descargar]</a>	20/5/2016
4	<a href="#">[Descargar]</a>	<a href="#">[Descargar]</a>	10/6/2016

#### 3.2 Donde descomprimir el proyecto

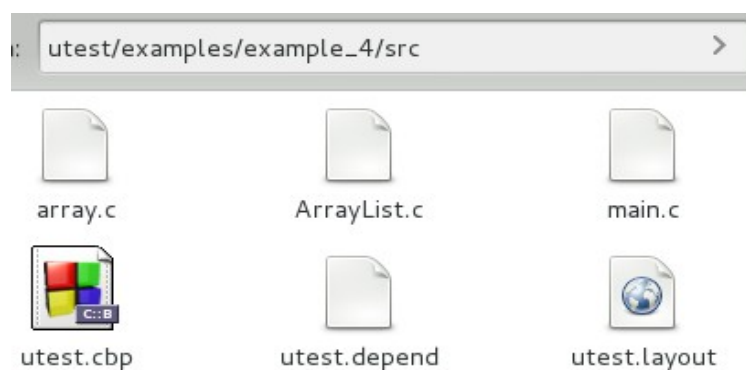
Descomprimir el proyecto en un directorio en el disco 'd' ya que no se encuentra frizado y ante un imprevisto reinicio de la computadora la información no se vera afectada.

#### 3.3 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**".



### 3.4 El archivo "main.c"

En el archivo main.c, encontraremos las siguientes líneas 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;
}
```

### 3.5 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 función **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
}
```

### 3.6 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.

```
***** Start Testing of:    initEmployees() *****
*****
-----
>Case[Verifica el retorno de la funcion al inicializar correctamente el array]
Setup...
assert equals failed '1' != '0'
Error en valor de retorno, si se pudo inicializar el array el valor a retornar e
s (0)
TEST FAILED
FILE:firstTest.c LINE:48
-----
>Case[Analiza si fue inicializado correctamente el array]
Setup...
assert equals failed '0' != '1'
Error en el valor de <.isEmpty> para indicar que una posicion esta vacia es (1)
TEST FAILED
FILE:firstTest.c LINE:57
-----
>Case[Verifica el retorno de la funcion al recibir un puntero NULL]
Setup...
assert equals failed '-1' != '1'
Error en valor de retorno, si se recibe una logitud erronea [length < 1]
el valor a retornar es (-1)
TEST FAILED
FILE:firstTest.c LINE:65
-----
>Case[Verifica el retorno de la funcion al recibir una longitud invalida]
Setup...
assert equals failed '-1' != '1'
Error en valor de retorno, si se recibe un puntero NULL el valor a retornar es (
-1)
TEST FAILED
FILE:firstTest.c LINE:73
***** Unit Tests Statistics:    initEmployees() *****
*****
**| Total Test | Succed Test | Failed Test | Effectiveness |**
**| 4          | 0           | 4          | 0 perc.      |**
*****
```

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.


```
-----
>Case[Verifica el retorno de la funcion al inicializar correctamente el array]
Setup...
assert equals failed '1' != '0'
Error en valor de retorno, si se pudo inicializar el array el valor a retornar e
s (0)
TEST FAILED
FILE:firstTest.c LINE:48
```

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




```
-----
>Case[Verifica el retorno de la funcion al inicializar correctamente el array]
Setup...
assert equals failed '1' != '0'
Error en valor de retorno, si se pudo inicializar el array el valor a retornar e
s (0)
TEST FAILED
FILE:firstTest.c LINE:48
```

Indica que el test fallo



```
>-----
>Case[Verifica el retorno de la funcion al inicializar correctamente el array]
Setup...
assert equals failed '1' != '0'
Error en valor de retorno, si se pudo inicializar el array el valor a retornar e
s (0)
TEST FAILED
FILE:firstTest.c LINE:48
```

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



```
>-----
>Case[Verifica el retorno de la funcion al inicializar correctamente el array]
Setup...
assert equals failed '1' != '0'
Error en valor de retorno, si se pudo inicializar el array el valor a retornar e
s (0)
TEST FAILED
FILE:firstTest.c LINE:48
```