Universidad de Santiago de Chile
Departamento de Ingeniería Informática
Métodos de Programación
Guía de preparatoria para las actividades de laboratorio
Experiencia N°3

1 INTRODUCCIÓN

En este documento se seguirá mostrando más información sobre el lenguaje de programación C, en especial ahora se trabajará con conceptos de variables, tanto locales como globales, y un nuevo "*tipo de variable*", llamado arreglo, en donde se ven aquellos que son llamados arreglos estáticos y los arreglos dinámicos. Estos conceptos serán explicados y desarrollados, para posteriormente, realizar un ejercicio, y finalizar el documento con un conjunto de ejercicios propuestos.

2 DESARROLLO

2.1 VARIABLES

Las variables sirven dentro de la programación sirven para almacenar valores dentro de la ejecución del código, en ellas se encuentran aquellas que son llamadas variables locales y variables globales.

Las variables locales corresponden a aquellas que están definidas dentro de una función, ya sea una función creada por el usuario o el ***main***, mientras que las variables globales se encuentran fuera de éstas. La principal diferencias de ellas es el nivel del ámbito dónde son conocidas las funciones, mientras que las variables globales son conocidas por todo el código incluyendo las funciones, incluyendo el ***main***; las variables locales son solo conocidas por la función en dónde se definen. En la Figura 2.1 se muestra un código en donde se ven dos variables definidas, una de forma local y otra de forma global. Para temas de orden y buenas prácticas de programación, nuestros programas **SIEMPRE** tendrán las variables globales definidas posterior a todos las importaciones que se posean en el código.

Figura 2.1: Variables locales y variables globales.

```c
#include <stdio.h>

int variableGlobal=5;

int main (){
    int variableLocal=3;
    printf("variableLocal = %d\nvariableGlobal = %d\n", variableLocal, variableGlobal);
    return 0;
}
```

2.2 ARREGLOS

Los arreglos en C corresponden a colecciones elementos, los cuales poseen sus valores almacenados en memoria de forma "consecutiva"[1], y que pueden ser referenciados bajo un mismo nombre. En sí son muy similares al tipo de dato list o listas en Python, siendo las principales diferencias las siguientes:

- Debido a que el lenguaje de programación C es fuertemente tipado, es necesario definir el tipo de dato del arreglo, y al hacer eso, el arreglo solo puede almacenar elementos de este tipo de dato, por ejemplo, si se declara un arreglo del tipo int, solo se podrán almacenar enteros, y no flotantes. En cambio en Python esto si era posible hacer.
- Hay casos en que es necesario definir el tamaño o cantidad de elementos que el arreglo va a contener, en cambio en Python esto no era necesario.

Si, es necesario definir el tamaño, dado que los arreglos poseen un tamaño estático, aunque se pueden construir unos dinámicos, estos tipos se ven a continuación.

2.2.1 ARREGLOS ESTÁTICOS

Estos son arreglos que al definirlos, se indica el tamaño o cantidad de elementos que éste posee, y sólo podrá almacenar esta cantidad de elementos. Para acceder a los elementos del arreglo es necesario hacerlo mediante la posición de éstos, que al igual que en Python, van de 0 a n-1, siendo n la cantidad de elementos que posee el arreglo. Hay que tener cuidado eso sí, ya que otra de las diferencias de los lenguajes, es que no existen posiciones negativas dentro de C, al igual que no tendremos métodos parecidos al append, o cómo saber cuántos elementos posee el arreglo. En el código mostrado en la Figura 2.2 se muestra como se define un arreglo y cómo se muestran todos los elementos que este posee.

Figura 2.2: Trabajando con un arreglo estático

```c
#include <stdio.h>
int main(){
    int cantidad;
    printf("Ingrese la cantidad de elementos del arreglo que usted desea: \n");
    scanf("%d",&cantidad);
    int array[cantidad];
    //Se le asignarán valores al arreglo
    for (int i = 0; i < cantidad; ++i){
        printf("Ingrese el valor a almacenar:\n");
        scanf("%d",&array[i]);
    }
    //Se imprime por pantalla los valores del arreglo
    for (int i = 0; i < cantidad; i++){
        printf("El valor en la posicion %d es %d\n", i, array[i]);
    }
    return 0;
}
```

[1] Se habla de forma consecutiva para no complicar para poder simplificar el tema, pero en realidad, cada elemento indica dónde está el siguiente.

2.2.2 ARREGLOS DINÁMICOS

Bueno, como las colecciones indican solamente direcciones de memoria, suponga que en la Tabla 2.1 se muestra como se almacena la instrucción L=[1,2,3], en color azul.

Como se mencionó anteriormente, las colecciones solo indican en qué dirección de memoria se comienzan a almacenar los valores, por esta razón la variable L definida no es nada más y nada menos que una dirección de memoria, que señala el primer elemento, y los posteriores están de forma "consecutiva".

Entonces, al realizar la instrucción L2=L, lo que se está diciendo es que en la variable L2 se almacenará lo que hay en la variable L, por lo que L2 solo posee la dirección de memoria en donde comienza el arreglo. Esto se ve en la Tabla 2.1, con la instrucción de color rojo.

Tabla 2.1

Dirección	Valor
01FF2020	L = 01FF2023
01FF2021	
01FF2022	L2=01FF2023
01FF2023	1
01FF2024	2
01FF2025	3
01FF2026	

Mediante lo explicado anteriormente, si, por ejemplo se quiere agregar un elemento a la lista L, pero al hacerlo también se modificará la lista L2, sin la necesidad de querer hacerlo. De hecho, esto se puede complicar más aún, agregando a la lista la misma lista o ir modificando los elementos de la lista.

Por lo explicado anteriormente, se indica que los arreglos en si solo poseen la dirección de memoria en dónde comienzan los elementos, por lo tanto, los arreglos bidimensionales, es decir las matrices, no son nada más que un arreglo de direcciones de memoria que almacena las direcciones de memoria donde comienzan cada uno de los siguientes arreglos. Por ejemplo, una matriz definida como:

- M = [[1, 2], [3, 4]]

Se encuentra almacenada en memoria de la forma que se representa en la Tabla 2.2.

Tabla 2.2: Representación del almacenamiento en memoria de una matriz.

Dirección de Memoria	Valor Guardado
0022FF00	M = 0022FF02
0022FF01	
0022FF02	0022FF05
0022FF03	0022FF08

0022FF04	
0022FF05	1
0022FF06	2
0022FF07	
0022FF08	3
0022FF09	4

De esta forma, en vez de ver una matriz como se muestra en la Figura 2.3, la representación gráfica es como se ve en la Figura 2.4; e inclusive, se puede hablar de arreglos multidimensionales como el que se muestra en la Figura 2.5.

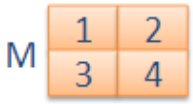

Figura 2.3: Representación errada de una matriz.

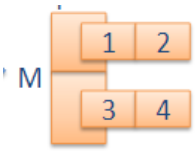

Figura 2.4: Representación correcta de una matriz.

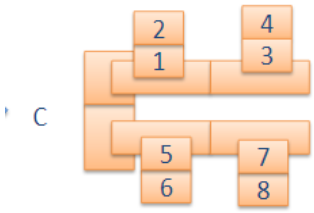

Figura 2.5: Representación de un arreglo multidimensional.

Entonces, dado que ya se sabe que los arreglos no son nada más que direcciones de memoria, se puede definir uno utilizando lo enseñado en la clase pasada, punteros, mediante la instrucción de declaración de uno:

- **int** *arreglo;

Y con esto se podrán trabajar arreglos de tamaño dinámico, pero hay que tener cuidado, ya que es necesario calcular cuánta memoria debemos asignar para almacenar nuestro arreglo, se deberá reservar y adjudicar, para lo cual se utilizan funciones como el ***malloc*** y ***sizeof***.

- ***malloc***: Es una función de la librería ***stdlib.h***, que permite reservar y adjudicar memoria, para lo que necesita como parámetro de entrada es la cantidad de memoria a adjudicar. En caso de ser adjudicada de forma correcta la memoria, la función retorna un puntero a *void*, mientras que en caso contrario, retorna un puntero nulo.
- ***sizeof***: No es una función en sí, sino que es un operador, pero se utiliza de forma similar a estas. Su funcionamiento consiste en que se le ingresa como parámetro de entrada un tipo de dato y el operador retorna un número entero que representa el espacio en memoria utilizado por ese tipo de dato.

Entonces, mediante lo mencionado, la forma de almacenar la memoria para un arreglo de 5 elementos enteros, los pasos que se deben seguir son:

1. Se debe saber cuanta memoria se quiere asignar, es decir, cuanta memoria utilizan los 5 números enteros.
    a. Se calcula cuánto espacio de memoria tiene un entero: ***sizeof(int)***
    b. Se multiplica ese valor por 5, la cantidad de enteros a almacenar: ***sizeof(int)*5***
2. Se reserva y se adjudica esa cantidad de memoria:
    a. ***malloc(sizeof(int)*5)***
3. Se debe castear el puntero generado por la función malloc, para esto se debe anteponer el tipo de dato al cual se quiere transformar ese tipo de dato void. En este caso, un puntero a números enteros.
    a. ***(int*)malloc(sizeof(int)*5)***
4. Se le asigna el resultado de esta operación a una variable, la cual debe ser un puntero a enteros:
    a. ***int** *a = (int*)malloc(sizeof(int)*5)*

Por ejemplo, si se quisiera definir un arreglo bidimensional, es necesario primero almacenar memoria para un puntero a enteros, y no a un entero. Posteriormente se debe reservar memoria para cada uno de los elementos que estén dentro de este arreglo bidimensional. Un código que realiza esta acción se muestra en la Figura 2.6.

Ahora, si se quisiera agregar un elemento más al arreglo o simplemente cambiar el tamaño, solamente hay que hacer un nuevo arreglo, del nuevo tamaño, copiar todos los elementos del arreglo original en el nuevo, y posteriormente agregar los nuevos elementos al arreglo nuevo. Pero se debe tener mucho cuidado, ya que cada vez que se deje de utilizar un arreglo, es necesario liberar memoria, para lo cual se utiliza la función ***free***. Esta función recibe por parámetro de entrada el puntero que apunta a la memoria adjudicada, que en este caso será el arreglo que se está trabajando. Si lo que se adjudicó es memoria para un arreglo multidimensional, **nunca** se debe liberar directamente la memoria de este, sino que se debe hacer desde la los elementos que posean los valores y luego los punteros que estaban apuntando a estos valores. Un ejemplo de uso de matrices y liberar memoria se muestra en la Figura 2.7.

Figura 2.6: Código para la creación de un arreglo bidimensional.

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

int main(){
    int **a = (int **)malloc(sizeof(int *)*3);

    a[0] = (int *)malloc(sizeof(int)*2);
    a[1] = (int *)malloc(sizeof(int)*2);
    a[2] = (int *)malloc(sizeof(int)*2);

    a[2][0] = 23;

    printf("%d\n", a[2][0]);
}
```

Figura 2.7: Utilizando de forma correcta la asignación de memoria.

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

int main(){
    int filas, columnas;
    printf("Ingrese el numero de filas de la matriz:");
    scanf("%d", &filas);
    printf("\nIngrese el numero de columnas de la matriz:");
    scanf("%d", &columnas);
    int **a = (int **)malloc(sizeof(int *)*filas);//Se crea la matriz
    for (int i = 0; i < filas; ++i){//Se asigna memoria para las columas de la matriz
        a[i] = (int *)malloc(sizeof(int)*columnas);
    }
    for (int i = 0; i < filas; ++i){//Se le solicita al usuario rellenar la matriz
        for (int j = 0; j < columnas; ++j){
            printf("\nIngrese un valor a almacenar en (%d, %d):", i, j);
            scanf("%d",&a[i][j]);
        }
    }
    for (int i = 0; i < filas; ++i){//Se muestran los valores al usuario
        for (int j = 0; j < columnas; ++j){
            printf("\nEl valor guardado en (%d, %d) es: %d", i, j, a[i][j]);
        }
    }
    for (int i = 0; i < filas; ++i){//Se libera memoria de las filas
        free(a[i]);
    }
    free(a);//Se libera memoria de la matriz
}
```

3 EJERCICIO RESUELTO

3.1 CONTEXTO

Más que explicar un ejercicio en esta oportunidad, se mostrará como trabajar con arreglos dinámicos y funciones, para esto se crea una función que calculará el número mayor que se encuentre dentro de una matriz.

3.2 EXPLICACIÓN DE LA SOLUCIÓN

Para obtener el número mayor, se utiliza la idea de que se tomará el primer elemento de la matriz como posible mayor, posteriormente, revisando uno a uno los elementos, se verá si se encuentra o no un número mayor, en caso de ser así se cambia por el valor almacenado anteriormente. Al recorrer toda la lista, se debe retornar el valor que se mantiene dentro de la variable.

3.3 SOLUCIÓN FINAL

En la solución final, la función mayor, recibe como parámetro de entrada un arreglo de direcciones de memoria de arreglos de enteros, es decir, una matriz de enteros, además, como no es posible obtener la cantidad de elementos de la matriz, estos parámetros también son pasados como entrada. La matriz en sí se pasa a la función por referencia, mientras que los valores son pasados por valor.

En la Figura 3.1 se muestra la solución al problema.

Figura 3.1: Solución al problema.

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

int mayor(int ** arreglo, int filas, int columnas){
    int mayor = arreglo[0][0];
    for (int i = 0; i < filas; ++i){
        for (int j = 0; j < columnas; ++j){
            if(mayor<arreglo[i][j])
                mayor=arreglo[i][j];
        }
    }return mayor;
}
int main(){
    int filas, columnas;
    printf("Ingrese el numero de filas de la matriz:");
    scanf("%d", &filas);
    printf("\nIngrese el numero de columnas de la matriz:");
    scanf("%d", &columnas);
    int **a = (int **)malloc(sizeof(int *)*filas);//Se crea la matriz
    for (int i = 0; i < filas; ++i){//Se asigna memoria para las columas de la matriz
        a[i] = (int *)malloc(sizeof(int)*columnas);
    }
    for (int i = 0; i < filas; ++i){//Se le solicita al usuario rellenar la matriz
        for (int j = 0; j < columnas; ++j){
            printf("\nIngrese un valor a almacenar en (%d, %d):", i, j);
            scanf("%d",&a[i][j]);
        }
    }
    int m = mayor(a, filas, columnas);
    printf("El numero mayor es: %d\n", m);
    int aux;
    printf("\nIngrese un numero para finalizar.\n");
    scanf("%d",&aux);
    for (int i = 0; i < filas; ++i){//Se libera memoria de las filas
        free(a[i]);
    }
    free(a);//Se libera memoria de la matriz
}
```

4 EJERCICIOS PROPUESTOS

4.1 EJERCICIO N°1:

Cree un programa que posea un arreglo de N elementos, los cuales son entregados por el usuario. Identifique cual es el elemento de menor y mayor valor dentro del arreglo. Para esto utilice arreglos estáticos y dinámicos.

4.2 EJERCICIO N°2:

Cree un programa que defina dos arreglos, uno de tamaño N y otro de tamaño M, cree una función que sume todos los elementos de ambos arreglos. Para esto utilice arreglos estáticos y dinámicos.

4.3 EJERCICIO N°3:

Cree un programa que cree un arreglo de 1 elemento, y solicite al usuario agregar un valor. Posteriormente, el programa debe consultar al usuario si desea agregar un nuevo elemento o no, en caso de querer hacerlo, el arreglo debe aumentar en uno su tamaño y se debe solicitar el nuevo valor, después se debe volver a consultar al usuario si desea ingresar un elemento nuevo. Si el usuario indica que no desea ingresar más elementos, el programa debe mostrar todos los elementos que se poseen almacenados en el arreglo.