

Laboratorio # 4

Introducción a C

Sebastián Rojas Gutiérrez

Laboratorio de IE-0117 Programación Bajo Plataformas Abiertas
Profesor: Ing. Marco Villalta Fallas, MSc.
27 de abril del 2023

Resumen

En este laboratorio del curso IE-0117 programación Bajo Plataformas Abiertas se llevó a cabo la realización de distintos programas utilizando el lenguaje de programación C, para cumplir ciertas funciones, como verificar si un número era primo, llevar a cabo una suma de variables aleatoria, imprimir una pirámide de números, contar el tipo de caracteres y determinar si una matriz era regular y calcular su determinante. Dentro de este laboratorio también se llevó a cabo la utilización de una combinación tanto de Bash como de C para saber si los programas de C podían ser compilados y ejecutados.

1. Nota teórica

Para la elaboración de este laboratorio, es necesario entender y tener claros los diferentes comandos básicos del lenguaje de programación C, para esto es útil consultar The GNU C Reference Manual [2]. La razón por la que es necesaria estudiar estos comandos y conceptos antes de realizar este laboratorio es debido a que, si no se tiene un correcto entendimiento sobre la utilización de estos, no se podrá llevar a cabo la realización de este laboratorio de una manera efectiva. Por otro lado, es necesario investigar y conocer para qué sirven comandos como `srnad` y `rand`, ya que comandos como estos son necesarios para la elaboración del laboratorio. Para conocer más sobre estos comandos y otros necesarios para la realización de este laboratorio es necesario consultar la página C documentation [1] de DevDocs. Para este laboratorio, también es necesario consultar la guía de uso de git EIE [3], para entender de mejor manera como utilizar git. Por último, es importante repasar las presentaciones realizadas por MSc. Marco Villalta Fallas [5] y [6] para refrescar lo visto en clase sobre el lenguaje de programación C.

2. Desarrollo

En esta sección se lleva a cabo la explicación de los distintos programas realizados en este laboratorio. Todo lo que se realizó en esta parte se puede encontrar en el repositorio <https://github.com/seb-byte/lab04> de github. En este caso se usó github debido a que al tratar de subir el repositorio a giteie daba el error que se puede ver en la Figura 1.

Figura 1

```
seblinux@debianL:~/labo4$ touch README.md
git init

git add README.md
git commit -m "first commit"
git remote add origin git@git.eie.ucr.ac.cr:srojasg/SRGlabo4.git
git push -u origin master
Reinitialized existing Git repository in /home/seblinux/labo4/.git/
On branch master
nothing to commit, working tree clean
error: remote origin already exists.
ssh: connect to host git.eie.ucr.ac.cr port 22: Connection timed out
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

1. Números primos

En esta sección del desarrollo se escribió un programa en c para detectar si el número introducido por un usuario era primo o no. Para esto se escribió una función que se encarga de determinar si un número es primo, esto lo hace, analizando primero los casos de 0, 1 y 4 debido a sus condiciones especiales. En caso de que el número introducido no fuera ninguno de estos, se procedía a analizar el restante de la división del número digitado entre todos los números, empezando desde dos hasta llegar a la mitad del número introducido. De este modo se encuentra si el número es divisible por cualquier otro número que no sea 1 o sí mismo, determinando así si el número introducido es primo o no. El código utilizado puede ser visto en la Figura 2, mientras que el output de una de las pruebas realizadas puede ser visto en la Figura 3.

Figura 2

```
GNU nano 5.4                                primos.c
#include <stdio.h>

#include <stdio.h>

int detcondición(int num_usuario);

int main(void) {
    int num_usuario;
    printf("Digite un número: \n");
    scanf("%d", &num_usuario);
    if (detcondición(num_usuario)) {
        printf("El número introducido es primo \n");
    } else {
        printf("el número introducido no es primo \n");
    }
    return 0;
}

int detcondición(int num_usuario) {
    if (num_usuario == 0 || num_usuario == 1) return 0;
    if (num_usuario == 4) return 0;
    for (int x = 2; x < num_usuario / 2; x++) {
        if (num_usuario % x == 0) return 0;
    }
    return 1;
}
```

Figura 3

```
Terminal - seblinux@debianL: ~/labo4/src/primos
File Edit View Terminal Tabs Help
seblinux@debianL:~/labo4/src/primos$ pico primos.c
seblinux@debianL:~/labo4/src/primos$ gcc primos.c -o primos
seblinux@debianL:~/labo4/src/primos$ ./primos
Digite un número:
17
El número introducido es primo
seblinux@debianL:~/labo4/src/primos$ ./primos
Digite un número:
12
el número introducido no es primo
seblinux@debianL:~/labo4/src/primos$
```

2.Sumas de variables aleatorias

Aquí se buscaba hacer un programa que pudiera sumar 1000 valores aleatorios entre un límite inferior y otro superior que fueran escogidos por el usuario, al igual que la semilla que podía ser escogida por el usuario o pseudoaleatoria. Para esto, era necesario pedirle tres valores al usuario utilizando la función scanf. Empleando srand y rand se podían generar los 1000 números aleatorios. La función srand tomaba el valor de la semilla, que en caso de ser pseudoaleatoria se emplearía el tiempo para que siempre fuese una diferente. Por otro lado, la función rand hacía uso de los límites escogidos para generar así los números, como es posible analizar en la Figura 4. Una de las pruebas que se realizaron para ver si el programa funcionaba de la manera correcta se puede ver en la Figura 5.

Figura 4

```
GNU nano 5.4 suma.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int lim_inferior, lim_superior, semilla, suma = 0;

    printf("Digite el límite inferior: ");
    scanf("%d", &lim_inferior);

    printf("Digite el límite superior: ");
    scanf("%d", &lim_superior);

    printf("Digite la semilla o digite 0 para generarla pseudoaleatoriamente: ");
    scanf("%d", &semilla);

    if (semilla == 0) {
        semilla = time(NULL);
        printf("La semilla generada es: %d\n", semilla);
    }

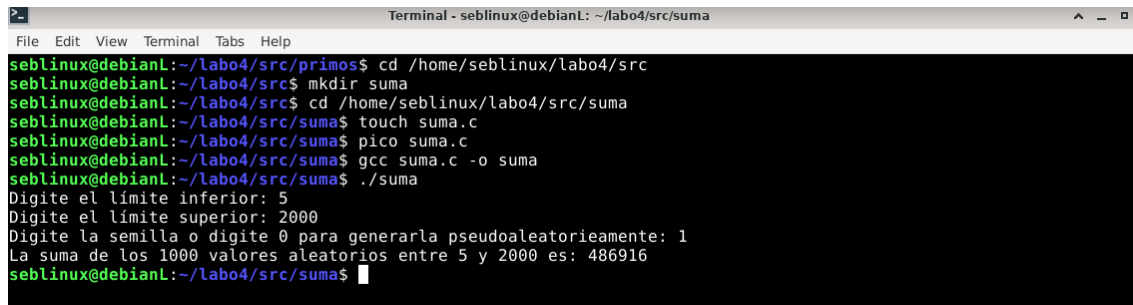
    srand(semilla);

    for (int i = 0; i < 1000; i++) {
        int num_random = rand() % (lim_superior - lim_inferior + 1) + lim_inferior;
        if (num_random % 2 == 0) {
            suma += num_random;
        }
    }

    printf("La suma de los 1000 valores aleatorios entre %d y %d es: %d\n", lim_inferior, lim_superior, suma);

    return 0;
}
```

Figura 5

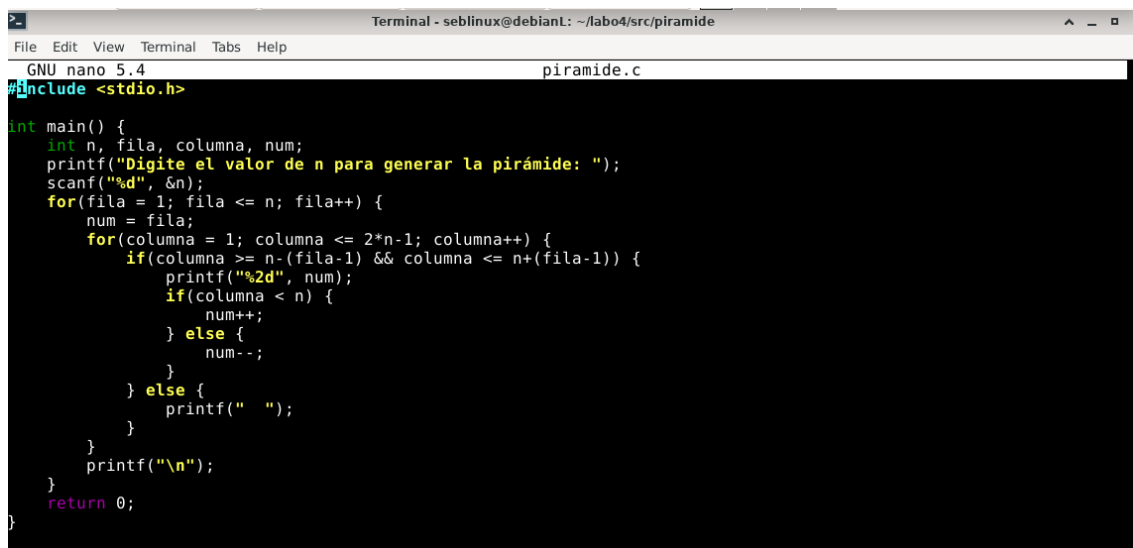


```
Terminal - seblinux@debianL: ~/labo4/src/suma
File Edit View Terminal Tabs Help
seblinux@debianL:~/labo4/src/primos$ cd /home/seblinux/labo4/src
seblinux@debianL:~/labo4/src$ mkdir suma
seblinux@debianL:~/labo4/src$ cd /home/seblinux/labo4/src/suma
seblinux@debianL:~/labo4/src/suma$ touch suma.c
seblinux@debianL:~/labo4/src/suma$ pico suma.c
seblinux@debianL:~/labo4/src/suma$ gcc suma.c -o suma
seblinux@debianL:~/labo4/src/suma$ ./suma
Digite el límite inferior: 5
Digite el límite superior: 2000
Digite la semilla o digite 0 para generarla pseudoaleatoriamente: 1
La suma de los 1000 valores aleatorios entre 5 y 2000 es: 486916
seblinux@debianL:~/labo4/src/suma$
```

3. Impresión de números

En esta parte, se buscaba imprimir de forma iterativa una pirámide de n números. Para esto, primero se necesitaba solicitarle al usuario el valor de n que este quería para generar así la pirámide de números. Para generar la pirámide se utilizaron dos for anidados para poder generar las filas como se requería para lograr imprimir la pirámide, también fue necesario el uso de la función `printf` al igual que el `/n` para cambiar de línea. El código resultante puede ser visto en la Figura 6, mientras que la prueba realizada para ver si cumplía con lo solicitado puede ser vista en la Figura 7.

Figura 6



```
Terminal - seblinux@debianL: ~/labo4/src/piramide
File Edit View Terminal Tabs Help
GNU nano 5.4 piramide.c
#include <stdio.h>

int main() {
    int n, fila, columna, num;
    printf("Digite el valor de n para generar la pirámide: ");
    scanf("%d", &n);
    for(fila = 1; fila <= n; fila++) {
        num = fila;
        for(columna = 1; columna <= 2*n-1; columna++) {
            if(columna >= n-(fila-1) && columna <= n+(fila-1)) {
                printf("%2d", num);
                if(columna < n) {
                    num++;
                } else {
                    num--;
                }
            } else {
                printf(" ");
            }
        }
        printf("\n");
    }
    return 0;
}
```

Figura 7

```
seblinux@debianL:~/labo4/src/piramide$ touch piramide.c
seblinux@debianL:~/labo4/src/piramide$ pico piramide.c
seblinux@debianL:~/labo4/src/piramide$ gcc piramide.c -o piramide
seblinux@debianL:~/labo4/src/piramide$ ./piramide
Digite el valor de n para generar la pirámide: 5
  1
 2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
seblinux@debianL:~/labo4/src/piramide$
```

4. Contar tipo de caracteres

En este caso, se escribió un programa que pudiera contar la cantidad de vocales, consonantes y dígitos. Para esto, se hizo una función que se fijaba si lo introducido por el usuario era una letra, para esto se fijaba carácter por carácter si iba de la a la z o de A a las Z. En el caso de que si se cumpliera esto se revisaba si cumplía con que el carácter era aeiou o AEIOU, si era alguno de esto se le sumaba uno al número de vocales y de lo contrario, le sumaba uno al número de consonantes. En el caso de detectar que no fuera una letra se contaba como dígito y se le sumaba uno a los dígitos. El código final se puede ver en la Figura 8 y una de las pruebas realizadas se puede ver en la Figura 9.

Figura 8

```
GNU nano 5.4                                cadena.c
#include <stdio.h>
#include <string.h>

int main()
{
    char str[20];
    int vocales = 0, consonantes = 0, dígitos = 0;

    printf("Ingrese una cadena de texto menor a 20 caracteres: ");
    scanf("%[^\n]", str);

    for(int i=0; i<strlen(str); i++){
        if(str[i] >= 'a' && str[i] <= 'z' || str[i] >= 'A' && str[i] <= 'Z'){
            if(strchr("aeiouAEIOU", str[i])){
                vocales++;
            } else {
                consonantes++;
            }
        } else if(str[i] >= '0' && str[i] <= '9'){
            dígitos++;
        }
    }

    printf("La cantidad de vocales es: %d\n", vocales);
    printf("La cantidad de consonantes es: %d\n", consonantes);
    printf("La cantidad de dígitos es: %d\n", dígitos);

    return 0;
}
```

Figura 9

```
seblinux@debianL:~/labo4/src/cadena$ pico cadena.c
seblinux@debianL:~/labo4/src/cadena$ gcc cadena.c -o cadena
seblinux@debianL:~/labo4/src/cadena$ ./cadena
Ingrese una cadena de texto menor a 20 caracteres: Hola me llamo 47
La cantidad de vocales es: 5
La cantidad de consonantes es: 6
La cantidad de dígitos es: 2
seblinux@debianL:~/labo4/src/cadena$
```

5. Matriz regular y determinante

En este caso, se le pedía al usuario que introdujera una matriz 3x3 la cual se generaba utilizando dos fors anidados. Para saber si la matriz era regular bastaba con calcular el determinante de esta debido a que si el determinante de una matriz es 0 es debido a que esta es singular, de lo contrario la matriz es regular. Para esto se empleó la fórmula para calcular el determinante de una matriz 3x3 y en el caso de que el determinante fuera distinto a cero era posible asegurar que la matriz era regular, por lo que se imprimía esto y el valor de su determinante. Mientras que si el determinante era cero, se imprimía que la matriz no era regular debido a lo explicado anteriormente. El código usado se puede ver en la Figura 10 y una de las pruebas realizadas se puede ver en la Figura 11.

Figura 10

```
GNU nano 5.4                                matriz-regular.c
#include <stdio.h>

int main() {
    int matriz[3][3];
    int i, j;
    int det;

    printf("Ingrese una matriz 3x3: \n");

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &matriz[i][j]);
        }
    }

    det = matriz[0][0] * (matriz[1][1] * matriz[2][2] - matriz[1][2] * matriz[2][1]) -
          matriz[0][1] * (matriz[1][0] * matriz[2][2] - matriz[1][2] * matriz[2][0]) +
          matriz[0][2] * (matriz[1][0] * matriz[2][1] - matriz[1][1] * matriz[2][0]);

    if (det != 0) {
        printf("La matriz es regular.\n");
        printf("El determinante de la matriz es: %d.\n", det);
    } else {
        printf("La matriz no es regular, su determinante es: %d.\n", det);
    }

    return 0;
}
```

Figura 11

```
seblinux@debianL:~/labo4/src/matriz-regular$ pico matriz-regular.c
seblinux@debianL:~/labo4/src/matriz-regular$ gcc matriz-regular.c -o matriz-regular
seblinux@debianL:~/labo4/src/matriz-regular$ ./matriz-regular
Ingrese una matriz 3x3:
3 1 1
1 3 1
1 1 3
La matriz es regular.
El determinante de la matriz es: 20.
```

6. C y Bash

Para realizar un script de bash que compile, ejecute y verifique la existencia de estos programas, y al finalizar la ejecución de los programas debe borrar los binarios correspondientes. Se utilizó el código que se puede ver en la Figura 12. Lo primero era compilar los programas, después de esto se analizaba si los binarios se habían creado, en caso de que su hubieran creado, se ejecutaban los programas como se ve en la Figura 13 y por último se usaba el comando rm para borrar los binarios de cada programa. Por último, se empleó el comando tree para ver la estructura, la cual quedó similar a lo solicitado, el único cambio siendo el orden, como se ve en la Figura 14.

Figura 12

```
GNU nano 5.4 script.sh
#!/bin/bash

# Compilar los programas
gcc primos/primos.c -o programa1
gcc suma/suma.c -o programa2
gcc piramide/piramide.c -o programa3
gcc cadena/cadena.c -o programa4
gcc matriz-regular/matriz-regular.c -o programa5

# Se fija si se crearon los binarios
if [ -f programa1 ] && [ -f programa2 ] && [ -f programa3 ] && [ -f programa4 ] && [ -f programa5 ]; then
    # Ejecutar los 5 programas
    ./programa1
    ./programa2
    ./programa3
    ./programa4
    ./programa5

    #rm borra los binarios
    rm programa1 programa2 programa3 programa4 programa5
    echo "Todos los programas fueron compilados, ejecutados y se borraron sus binarios correctamente"
else
    echo "Error: Uno o mas binarios no fueron creados con exito"
fi
```

Figura 13

```
seblinux@debianL:~/labo4/src$ bash script.sh
Digite un número:
17
El número introducido es primo
Digite el límite inferior: 5
Digite el límite superior: 20
Digite la semilla o digite 0 para generarla pseudoaleatoriamente: 1
La suma de los 1000 valores aleatorios entre 5 y 20 es: 6328
Digite el valor de n para generar la pirámide: 5
  1
 2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
Ingrese una cadena de texto menor a 20 caracteres: hola
La cantidad de vocales es: 2
La cantidad de consonantes es: 2
La cantidad de dígitos es: 0
Ingrese una matriz 3x3:
3 1 1
1 3 1
1 1 3
La matriz es regular.
El determinante de la matriz es: 20.
Todos los programas fueron compilados, ejecutados y se borraron sus binarios correctamente
seblinux@debianL:~/labo4/src$
```

Figura 14

```
seblinux@debianL:~/labo4$ tree
.
├── informe
│   └── informe.pdf
└── src
    ├── cadena
    │   └── cadena.c
    ├── matriz-regular
    │   └── matriz-regular.c
    ├── piramide
    │   └── piramide.c
    ├── primos
    │   └── primos.c
    ├── script.sh
    ├── suma
    │   └── suma.c
    └──
```

7 directories, 7 files

```
seblinux@debianL:~/labo4$
```


7. Diagramas de flujo

Los diagramas de flujo pueden encontrarse al final de este documento.

3. Conclusiones y recomendaciones

A modo de conclusión, es necesario investigar sobre el uso correcto de los distintos comandos de C y en general sobre debido a que al principio aprender un nuevo lenguaje de programación puede ser difícil, por lo que es necesario tener paciencia a la hora de hacer este laboratorio. También, utilizar C es una herramienta que es de mucha utilidad cuando se trata de llevar a cabo ciertas funciones, por lo que este laboratorio es de suma importancia para ejercitar el conocimiento que se tiene sobre C y en caso de no tener mucho, ampliarlo. A modo de recomendación, es útil a haber atendido a las distintas clases que se han tenido sobre este tema y repasar las presentaciones antes de realizar este laboratorio, esto para refrescar lo aprendido y tener claros ciertos conceptos que son claves para la elaboración de este laboratorio.

4. Bibliografía

Referencias

- [1] Devdocs. (s.f.). *C documentation*. <https://devdocs.io/c/>
- [2] GNU. (s.f.). *The GNU C Reference Manual*. <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>
- [3] Jiménez, I. (2021). *Manual de configuración y uso de Git EIE*. Escuela de Ingeniería Eléctrica, UCR.
- [4] Pinzón, J. (2018). *Informe de laboratorio*.
- [5] Villalta, M. (2023). *C: Características*. [Diapositivas de PowerPoint]. Escuela de Ingeniería Eléctrica, UCR.
- [6] Villalta, M. (2023). *C: Introducción*. [Diapositivas de PowerPoint]. Escuela de Ingeniería Eléctrica, UCR.
- [7] VMware. (s.f). *Máquina virtual*. <https://www.vmware.com>