Universidad de Santiago de Chile
Departamento de Ingeniería Informática
Métodos de Programación
Guía de preparatoria para las actividades de laboratorio
Experiencia N°2

# 1 INTRODUCCIÓN

En este documento se desarrollan nuevos conceptos, complementando lo ya explicado en la clase anterior, en específico se tratan temas como funciones y estructuras de control.

En el documento se desarrolla temáticamente iniciando con la solicitud de información al usuario por la entrada estándar, para posteriormente dar paso a la explicación de funciones, explicando en esta sección conceptos como lo son el paso por valor y paso por referencia. Posteriormente se continúa con las distintas estructuras de control, especial con las bifurcaciones y los ciclos que se pueden implementar con el lenguaje C, aunque no todos, ya que algunos se verán más adelante en la materia. A continuación, se trabaja con un ejercicio dentro del documento, para formalizar los temas mencionados tanto en esta clase como en la anterior, dejando para el final una serie de ejercicios propuestos de complemento del estudio individual.

# 2 DESARROLLO

## 2.1 SOLICITANDO INFORMACIÓN AL USUARIO

En la clase anterior, se abordó el cómo enviar información al usuario por la salida estándar, mostrando solo un mensaje inicialmente (“Hola mundo”) para que al final también se pueda mostrar el valor almacenado dentro de una variable, utilizando banderas para esto.

Ahora ha llegado el momento para que nuestro programa pueda solicitar información al usuario, para lo cual se utiliza la función ***scanf*** de la librería de entradas y salidas estándar ***stdio.h***, esta función es del estilo:

- **scanf** ( *Arg1*, *Arg2*)

Donde *Arg1* corresponde a la bandera del tipo de dato que se desea obtener, obedeciendo en estos momentos a las mismas banderas utilizadas en el caso del ***printf***, por ejemplo, para almacenar un entero, la bandera puede ser tanto ***%d*** como ***%i***.

El *Arg2* corresponde a la dirección de memoria en donde debe ser guardado el valor ingresado por el usuario, por lo que un programa que solicite un número al usuario y muestre este número por pantalla sería como se muestra en la Figura 2.1.

Figura 2.1: Programa que solicita un número al usuario y lo muestra posteriormente por pantalla.

```c
#include <stdio.h>

int main(){
    int variable;
    printf("Ingrese un numero: \n");
    scanf("%d",&variable);
    printf("El numero ingresado es: %d\n", variable);
    return 0;
}
```

Si, para la función scanf es necesario indicar la dirección de memoria donde se almacena el valor ingresado por el usuario, y para trabajar con los conceptos de direcciones de memoria, se tratan a continuación temas de referenciación, desreferenciación y punteros.

### 2.1.1 REFERENCIACIÓN

El operador de referenciación u operador de dirección en C es representado por el símbolo *ampersand* (**&**) y la funcionalidad de este operador es obtener la dirección de memoria en dónde se encuentra almacenada la variable. Por ejemplo, en código mostrado en la Figura 2.2, declara una variable, llamada a la cual es del tipo entero con un valor 0, posteriormente la instrucción printf envía un mensaje por pantalla mostrando el valor (**%d**, dado que es un número entero) que posee almacenado la variable y la ubicación de esta en memoria (**%p**, dado que se está mostrando un puntero a memoria).

Figura 2.2: Mostrando el valor y dirección de memoria de una variable.

```c
#include <stdio.h>

int main()
{
    int a=0;
    printf("La variable a con valor = %d esta almacenda en %p\n", a, &a);
    return 0;
}

```

Una de las salidas que genera este código es la mostrada en la Figura 2.3. Se habla de una de las salidas, dado que la dirección de memoria mostrada puede ir variando por temas internos como externos.

Figura 2.3: Resultado de ejecución del programa de la Figura 2.2.

```
La variable a con valor = 0 esta almacenda en 0061FF2C
```

Cabe mencionar que la dirección que se muestra está escrita en formato hexadecimal.

### 2.1.2 PUNTEROS

Los punteros son un tipo de dato en C que permiten indicar un lugar en la memoria que almacenará cierto tipo de dato, en sí, los punteros apuntan al espacio físico del dato o variable. Los punteros pueden apuntar a direcciones de memoria de cualquier tipo, incluidas dentro de estas las variables, arreglos, funciones y estructuras.

La forma de declaración de un puntero es la siguiente:

- <Tipo de dato> ***<Nombre del puntero>;**
- <Tipo de dato> ***<Nombre del puntero> = NULL;**

Al igualar la segunda opción muestra que el puntero está apuntando a nada (**NULL**), es decir, no posee una dirección de memoria asignada.

Para mostrar el funcionamiento de la declaración de punteros, se tiene en la Figura 2.4, donde se declara un puntero y una variable, se muestra el valor por defecto que posee el puntero, para posteriormente almacenar la dirección de memoria de la variable, mostrando la nueva dirección almacenada en el puntero y la dirección de la variable. El resultado de la ejecución se puede ver en la Figura 2.5.

Figura 2.4: Código de ejemplo de funcionamiento de punteros.

```c
#include <stdio.h>

int main()
{
    int *puntero;
    int variable = 23;
    printf("Puntero pose el valor: %p\n", puntero);
    printf("La variable esta en la direccion de memoria: %p\n", &variable);
    puntero = &variable;
    printf("Puntero pose el valor: %p\n", puntero);
    printf("La variable esta en la direccion de memoria: %p\n", &variable);
    return 0;
}
```

Figura 2.5: Resultado de la ejecución del código de la Figura 2.4

```
Puntero pose el valor: 00400080
La variable esta en la direccion de memoria: 0061FF28
Puntero pose el valor: 0061FF28
La variable esta en la direccion de memoria: 0061FF28
```

Al igual que el código anterior, se debe tener en cuenta que las direcciones de memoria varían, esto es debido a que el Sistema Operativo es quien maneja esto.

### 2.1.3 DESREFERENCIACIÓN

El operador de desreferenciación u operador de indirección en C es representado por el símbolo *asterisco* (*), siendo un operador unitario, que su funcionalidad es obtener el valor almacenado en la dirección de memoria que se está accediendo. Por ejemplo, el código mostrado en la Figura 2.6, es muy similar al código de la Figura 2.4, con la diferencia de que aparte de mostrar la dirección de memoria muestra el valor almacenado en ella. El resultado de la ejecución del código se muestra en la Figura 2.7.

Figura 2.6: Código de ejemplo de desreferenciación.

```c
#include <stdio.h>

int main()
{
    int *puntero;
    int variable = 23;
    printf("Puntero pose el valor: %p\n", puntero);
    printf("El valor almacenado en el puntero es: %d\n", *puntero);
    printf("La variable esta en la direccion de memoria: %p\n", &variable);
    printf("La variable posee almacenado el valor: %d\n", variable);
    printf("\n\n\n");
    puntero = &variable;
    printf("Puntero pose el valor: %p\n", puntero);
    printf("El valor almacenado en el puntero es: %d\n", *puntero);
    printf("La variable esta en la direccion de memoria: %p\n", &variable);
    printf("La variable posee almacenado el valor: %d\n", variable);
    return 0;
}

```

Figura 2.7: Resultado de la ejecución del código de la Figura 2.6.

```
Puntero pose el valor: 00400080
El valor almacenado en el puntero es: 17744
La variable esta en la direccion de memoria: 0061FF28
La variable posee almacenado el valor: 23



Puntero pose el valor: 0061FF28
El valor almacenado en el puntero es: 23
La variable esta en la direccion de memoria: 0061FF28
La variable posee almacenado el valor: 23
```

Como se ve en la ejecución del código, inicialmente el puntero está apuntando a la dirección de memoria *00400080*, la cual fue asignada por el sistema[1], mientras que la variable está almacenada en la dirección de memoria *0061FF28*. En la instrucción de la línea 12 del código fuente, se igualan los valores de las direcciones de memoria, por lo que posteriormente, en la segunda parte donde se imprimen los valores, tanto el puntero como la variable está almacenada en el mismo lugar.

Además, se puede apreciar que, a pesar de que en ningún momento se le asignó un valor a lo que está almacenado en la dirección de memoria del puntero, al imprimir el valor por pantalla este muestra un *17774*, a esto se le llama que se posea basura almacenada, que es un tema que se

[1] Si se hubiera realizado la creación y asignación del puntero a NULL no se hubiera podido imprimir, ya que aparecería un error llamado *Violación de Segmento*.
(https://es.wikipedia.org/wiki/Violaci%C3%B3n_de_acceso)

abordará más adelante en la materia. Posteriormente, al igualar las direcciones de memoria, al consultar por los valores, se muestra el mismo valor *23* tanto en la variable como lo que almacena el puntero.

## 2.2 FUNCIONES

### 2.2.1 DECLARACIÓN DE FUNCIONES

Las funciones son una herramienta muy útil dentro de la programación, ya que permite encapsular cierta parte de la solución del problema, e incluso para procesos largos y repetitivos. Dentro del curso anterior, ya se han construido funciones, por ejemplo, en la Figura 2.9 se muestra el código fuente en Python 2.7 de un programa que solicita dos números al usuario y los suma mediante el uso de una función llamada suma.

Figura 2.9: Programa en Python que solicita dos números y los suma.

```
#Entrada: Dos numeros
#Salida: Un numero
def suma(a, b):
    resultado = a + b
    return resultado

valor1 = input("Ingrese el primer numero: ")

valor2 = input("Ingrese el segundo numero: ")

resultadoSuma = suma (valor1, valor2)

print "La suma de", valor1,"y",valor2,"es",resultadoSuma
```

La realización del mismo programa en C, sería algo como se muestra en la Figura 2.10, donde, mediante lo ya visto en este documento se solicitan dos números al usuario, los cuales son almacenados en las variables numero1 y numero2.

Figura 2.10: Programa que suma dos números en C.

```
1  #include <stdio.h>
2  //Entrada: Dos numeros
3  //Salida: La suma de los dos numeros
4  int suma(int a, int b){
5      int resultado;
6      resultado = a + b;
7      return resultado;
8  }
9
10 int main()
11 {
12     int numero1, numero2, respuesta;
13     printf("Ingrese el primer numero para sumar:\n");
14     scanf("%d",&numero1);
15     printf("Ingrese el segundo numero para sumar:\n");
16     scanf("%d",&numero2);
17     respuesta = suma(numero1, numero2);
18     printf("El resultado de la suma de %d con %d es %d\n", numero1, numero2, respuesta);
19     return 0;
20 }
```

Entre las líneas 4 a la 8 se encuentra definida la función que realiza la suma de los dos números, la cual está estructurada de la siguiente forma:

- Encabezado: *int suma (int a, int b)*
  - **int**: Este es el encabezado de la función, el cual en primer lugar posee el tipo de dato de retorno de la función, y todos los caminos que se puedan tomar en la ejecución de la función, deben llegar a este mismo tipo de dato[2]. En caso de que se desee construir una función que retorne nada, es necesario colocar la palabra reservada ***void***.
  - **suma**: Es el nombre de la función. Recordar que siempre los nombres de las funciones deben ser representativos.
  - int a, int b: Son los parámetros formales de la función, con ellos se realizan las acciones dentro de la función en sí. Las funciones pueden admitir de 0 a n parámetros formales, pero cada uno debe tener el tipo de dato que será y el nombre que recibirá.
- Cuerpo de la función
  - El cuerpo de la función se identifica porque se encuentra entre llaves "**{**", "**}**".
  - Está compuesta por un conjunto de instrucciones, donde cada una debe terminar en punto y coma "**;**".
  - Las instrucciones de esta función son las de las líneas 5, 6 y 7.
- Retorno: *return resultado*
  - Es el valor que retornará la función al ser llamada, en este caso, el valor que posea la variable resultado en ese momento.
  - Hace que la función termine su ejecución.

[2] Recordar que es distinto el tipo de dato a el valor, por ejemplo 5 es un valor, del tipo de dato entero (**int**). Por lo que se debería colocar en los encabezados palabras como *int, float, char* entre otros.

- En caso de que se retorne nada o ***void***, igualmente es necesario colocar la instrucción ***return***, pero no se debe colocar nada a su lado.

Un ejemplo de la ejecución del programa es la mostrada en la Figura 2.11.

Figura 2.11: Ejecución del programa de la Figura 2.10

```
Ingrese el primer numero para sumar:
11
Ingrese el segundo numero para sumar:
12
El resultado de la suma de 11 con 12 es 23
```

### 2.2.2 PASO POR VALOR

El paso por valor es una de las formas en que se realizan las funciones dentro del lenguaje C, el cual, consiste en crear una copia exacta de los parámetros actuales, para que la función utilice esa copia como parámetros formales, y se desarrolle con normalidad. En la Figura 2.12 se muestra un programa que posee una función que utiliza el paso por valor.

Figura 2.12: Programa con paso por valor.

```c
#include <stdio.h>

int funcion(int a){
    a = a + 1;
    return a;
}

int main()
{
    int numero, respuesta;
    printf("Ingrese un numero:\n");
    scanf("%d",&numero);
    printf("El valor actual, antes de llamar a la funcion, de numero es: %d\n", numero);
    respuesta = funcion(numero);
    printf("El resultado de la funcion es: %d.\n", respuesta);
    printf("El valor actual, despues de llamar a la funcion, de numero es: %d\n", numero);
    return 0;
}
```

Entonces, analizando el código paso a paso, se tiene lo siguiente:

1. Se definen dos variables en la línea 10, llamadas *numero* y *respuesta*. Suponga que se almacenan en memoria interna, con dirección *dir1* y *dir2* cada una respectivamente.
2. Se imprime por pantalla un mensaje al usuario, solicitando información.
3. El programa espera, mediante el scanf, que el usuario ingrese un número (Si ingresa otra cosa ocurrirá un error), y ese valor es almacenado en la dirección de memoria perteneciente a la variable *numero*, entonces suponiendo que el usuario ingresa un 23, en *dir1* se almacenará un 23.
4. Se imprime por pantalla el valor de *numero*, el cual es 23 actualmente.

5. Se llama a la función con el valor de *numero*, por lo que se ejecuta la línea 3, creando una variable *a*, la cual estará almacenada en la dirección de memoria *dir3*, ya que es otra variable y poseerá el valor 23.
6. Al ejecutarse la línea 4, se aumenta en uno el valor almacenado en la dirección de memoria *dir3*, por lo que el 23 es borrado y ahora se almacena 24.
7. Se ejecuta la línea 5, retornando el valor que hay en a, en este caso, lo que hay en *dir3*, es decir 24.
8. Se ejecuta la línea 14, por lo que el valor de *respuesta*, que estaba en *dir2*, será reemplazado por lo que hay en *dir3*, lo que significa que *dir2* ahora posee el valor 24.
9. Se ejecuta la línea 15, donde se muestra por pantalla lo que hay en la variable *respuesta*, es decir, lo que está almacenado en *dir2*, lo que es un 24.
10. Se ejecuta la línea 16, mostrando lo que hay almacenado en la variable *numero*, es decir en *dir1*, que es un 23.

Este proceso se puede ver en la Figura 2.13.

Figura 2.13: Ejecución del programa de la Figura 2.12 con entrada = 23.

```
Ingrese un numero:
23
El valor actual, antes de llamar a la funcion, de numero es: 23
El resultado de la funcion es: 24.
El valor actual, despues de llamar a la funcion, de numero es: 23
```

### 2.2.3 PASO POR REFERENCIA

El paso por referencia consiste en que, en vez de enviar una copia del parámetro actual, se utiliza el parámetro por sí mismo. Para lograr hacer esto, es necesario trabajar con las direcciones de memoria (recordar los operadores ***** y **&**). En la Figura 2.14 se muestra un código que realiza el mismo proceso del código mostrado en la sección 2.2.2, pero esta vez, realizando el proceso mediante paso por referencia.

Los principales cambios que hay en este programa son que:

- La función define como parámetro formal un puntero a entero (debe recibir la dirección de memoria de la variable, no la variable en sí).
- Las operaciones de la función se realizan sobre los valores que hay en el puntero (***a**).
- La función es llamada con la dirección de memoria de la variable *numero* (***&numero***).

Entonces, si se realiza el mismo procedimiento anterior, y se analiza el código paso a paso, se tiene lo siguiente:

11. Se definen dos variables en la línea 10, llamadas *numero* y *respuesta*. Suponga que se almacenan en memoria interna, con dirección *dir1* y *dir2* cada una respectivamente.
12. Se imprime por pantalla un mensaje al usuario, solicitando información.
13. El programa espera, mediante el scanf, que el usuario ingrese un número (Si ingresa otra cosa ocurrirá un error), y ese valor es almacenado en la dirección de memoria perteneciente

a la variable *numero*, entonces suponiendo que el usuario ingresa un 23, en *dir1* se almacenará un 23.

Figura 2.14: Programa con paso por referencia.

```
1  #include <stdio.h>
2
3  int funcion(int *a){
4      *a = *a + 1;
5      return *a;
6  }
7
8  int main()
9  {
10     int numero, respuesta;
11     printf("Ingrese un numero:\n");
12     scanf("%d",&numero);
13     printf("El valor actual, antes de llamar a la funcion, de numero es: %d\n", numero);
14     respuesta = funcion(&numero);
15     printf("El resultado de la funcion es: %d.\n", respuesta);
16     printf("El valor actual, despues de llamar a la funcion, de numero es: %d\n", numero);
17     return 0;
18 }
```

14. Se imprime por pantalla el valor de *numero*, el cual es 23 actualmente.
15. Se llama a la función con el valor de la dirección de memoria donde se almacena *numero*, por lo que se ejecuta la línea 3, creando una variable *a*, la que es un puntero a entero, es decir, una dirección de memoria que almacena un entero, la cual estará almacenada en la dirección de memoria *dir3*, ya que es otra variable. Y poseerá el valor de la dirección de memoria de *numero*, la cual es *dir1*.
16. Al ejecutarse la línea 4, se accede al valor almacenado en *dir1*, el cual es 23, se aumenta en 1 y se almacena como valor en *dir1*, por lo que ahora, *dir1* almacena el valor 24.
17. Se ejecuta la línea 5, retornando el valor que hay en la dirección de memoria *a*, en este caso, lo que hay en *dir1*, es decir 24.
18. Se ejecuta la línea 14, por lo que el valor de *respuesta*, que estaba en *dir2*, será reemplazado por lo que hay en *dir3*, lo que significa que *dir2* ahora posee el valor 24.
19. Se ejecuta la línea 15, donde se muestra por pantalla lo que hay en la variable *respuesta*, es decir, lo que está almacenado en *dir2*, lo que es un 24.
20. Se ejecuta la línea 16, mostrando lo que hay almacenado en la variable *numero*, es decir en *dir1*, que es un 24.

Este proceso se puede ver en la Figura 2.14.

Figura 2.14: Ejecución del programa de la Figura 2.13 con entrada = 23.

```
Ingrese un numero:
23
El valor actual, antes de llamar a la funcion, de numero es: 23
El resultado de la funcion es: 24.
El valor actual, despues de llamar a la funcion, de numero es: 24
```

## 2.3 ESTRUCTURAS DE CONTROL

Dado que los programas al ejecutarse, se realiza de forma lineal, desde arriba a abajo, según las instrucciones del código fuente, y hay veces que se necesita tomar alguna decisión o repetir ciertos procesos, nos son útiles las distintas estructuras de control que nos ofrece el lenguaje, que para esta vez, serán divididas en dos, las bifurcaciones, útiles para cuando el programa deba realizar una u otra cosa, y los ciclos, necesarios en caso de repetir cierto conjunto de instrucciones.

### 2.3.1 BIFURCACIONES

Las bifurcaciones son un tema que ya se debe haber visto con anterioridad en el curso previo de Fundamentos de Computación y Programación, pero esta vez se dan a conocer, en primera instancia, su sintaxis en C y posteriormente algunas otras nuevas formas de hacer que nuestro programa tome decisiones.

Dentro de las bifurcaciones más utilizadas dentro de los lenguajes de programación, son las llamadas bifurcaciones *Sí-Sino* o *If-Else*, las cuales ayudan a realizar cierto conjunto de instrucciones acorde al cumplimiento de una cierta condición, gráficamente se puede representar como se muestra en la Figura 2.15.

Dentro del lenguaje C, esta instrucción posee una sintaxis similar a la que se poseía en Python, pero con detalles como se muestra en la Figura 2.16, donde se utiliza la palabra reservada ***if***, para posteriormente, entre paréntesis se coloca la condición, la que en caso de ser verdadera, se ejecutan las instrucciones que están posteriores a ésta, y están encerradas entre llaves "**{**" y "**}**". En caso contrario se ejecutan las instrucciones que están en el ***else***, que también están encerradas entre llaves "**{**" y "**}**".

Al igual que en Python, en este lenguaje también es posible realizar anidaciones de ***if***, es decir, colocar ***if*** dentro de otro ***if*** o ***else***. En caso de que la primera instrucción a realizar dentro de un ***else***, sea un ***if***, este se puede simplificar. Tanto el caso anidado como el caso simplificado se puede ver en la Figura 2.16.

Figura 2.15: Representación gráfica de las instrucciones *if-else*

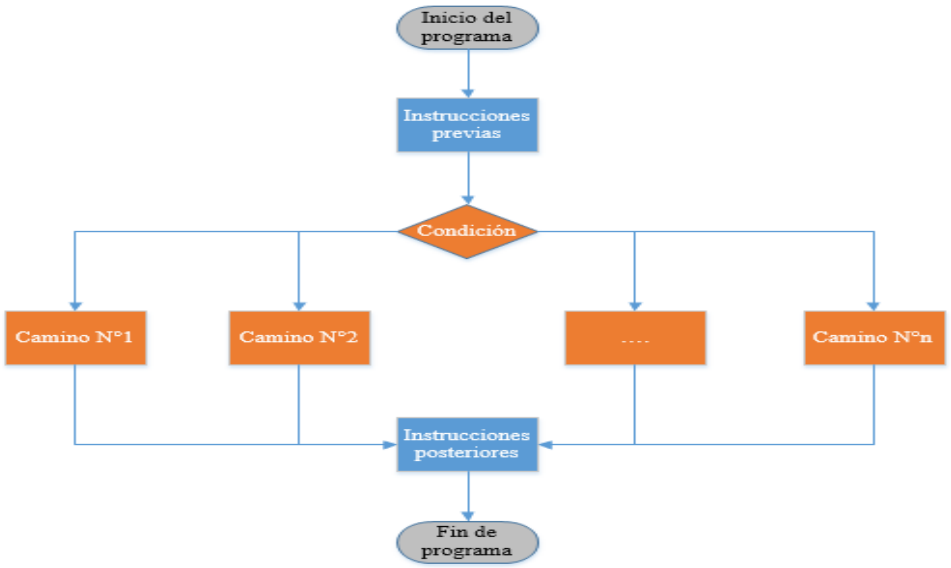



Figura 2.16: Programa utilizando *if-else*

```c
#include <stdio.h>
int main(){
    int a=5;
    if(a<0){
        printf("El número es negativo\n");
    }else{
        printf("El número es positivo\n");
    }
}
```

Figura 2.17: Comparación entre *if* anidado e *if* simplificado.

IfAnidado.c

```c
#include <stdio.h>

int main()
{
    int numero;
    printf("Ingrese un numero:\n");
    scanf("%d",&numero);
    if(numero > 0){
        printf("El numero ingresado es mayor que cero.\n");
    }else{
        if (numero < 0){
            printf("El numero ingresado es menor que cero.\n");
        }else{
            printf("El numero es un 0.\n");
        }
    }
    return 0;
}
```

IfSimplificado.c

```c
#include <stdio.h>

int main()
{
    int numero;
    printf("Ingrese un numero:\n");
    scanf("%d",&numero);
    if(numero > 0){
        printf("El numero ingresado es mayor que cero.\n");
    }else if (numero < 0){
            printf("El numero ingresado es menor que cero.\n");
    }else{
        printf("El numero es un 0.\n");
    }
    return 0;
}
```

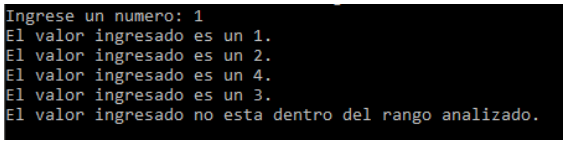
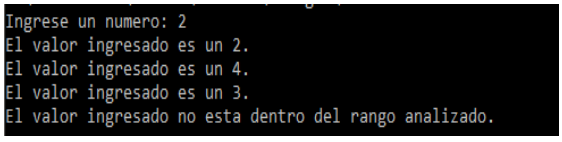
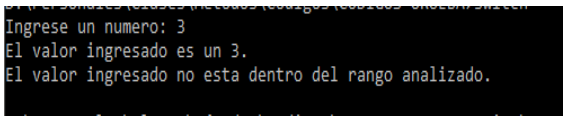
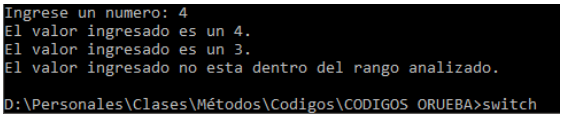
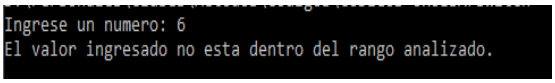
La estructura de un switch se puede ver en la Figura 2.18, mientras que los resultados de la ejecución con distintos valores se pueden observar en la Tabla 2.1.

Figura 2.18: Código de un *switch-case*.

```c
#include <stdio.h>

int main(){
    int valor;
    printf("Ingrese un numero: \n");
    scanf("%d",&valor);
    switch(valor){
        case 1:
            printf("El valor ingresado es un 1.\n");
        case 2:
            printf("El valor ingresado es un 2.\n");
        case 4:
            printf("El valor ingresado es un 4.\n");
        case 3:
            printf("El valor ingresado es un 3.\n");
        default:
            printf("El valor ingresado no esta dentro del rango analizado.\n");
    }
}
```

Tabla 2.1: Distintas ejecuciones del programa de la Figura 2.18

| | |
|---|---|
|  |  |
| Caso en que el usuario ingrese un 1 | Caso en que el usuario ingrese un 2 |
|  |  |
| Caso en que el usuario ingrese un 3 | Caso en que el usuario ingrese un 4 |
|  | |
| Caso en que el usuario ingrese cualquier número distinto a 1, 2 , 3, 4. | |

La sentencia ***switch*** permite que el programa realice una ejecución desde cierto punto del código en adelante, como se puede ver en la Tabla 2.1. Por ejemplo, en el caso de que se ingrese un 1, se realiza la ejecución desde que dentro del ***switch***, el caso posea este valor, que en este ejemplo es el primer caso, razón por la cual se muestran los 5 mensajes por pantalla. Posteriormente

en el caso 2, cuando la variable posee un valor 2, se muestran 4 mensajes y así sucesivamente. Es importante hacer notar que los casos no deben tener un orden acorde al valor buscado, ni tampoco deben manejar todos los valores que pueda tener la variable, sino que el orden es acorde a que queramos realizar y que deseemos manejar.

Además de los casos, que se manejan con la instrucción case, existe la instrucción ***default***, u opción por defecto, corresponde al caso en que se desee que una instrucción se realice siempre o en caso de que no haya sido válido ninguno de los casos anteriores.

Los ***switch*** son muy comunes verlos con la instrucción ***break***, que hace que se quiebre la ejecución del ***switch*** en cierto momento, por lo cual, normalmente se colocan como la última sentencia en cada uno de los casos, como se muestra en la Figura 2.19[3].

Figura 2.19: Código de un *switch-case* utilizando *break*.

```c
#include <stdio.h>

int main(){
    int valor;
    printf("Ingrese un numero: ");
    scanf("%d",&valor);
    switch(valor){
        case 1:
            printf("El valor ingresado es un 1.\n");
            break;
        case 2:
            printf("El valor ingresado es un 2.\n");
            break;
        case 4:
            printf("El valor ingresado es un 4.\n");
            break;
        case 3:
            printf("El valor ingresado es un 3.\n");
            break;
        default:
            printf("El valor ingresado no esta dentro del rango analizado.\n");
            break;
    }
}
```

¿La última forma de realizar bifurcaciones con que se trabajará en el curso es mediante el uso del llamado operador ternario (? **:**), el cual posee la siguiente configuración:

- <variable> = <condición>**?**<instrucción si es verdadera> **:** <instrucción en caso falso>;

Un ejemplo del uso de este operador es mediante el programa de la Figura 2.20, el cual recibe un número por parte del usuario e indica si ese número es par o impar.

[3] Escriba el programa en C, compile y ejecute para ver qué sucede.

Figura 2.20: Programa de uso del operador ternario.

```c
#include <stdio.h>

int main(){
    int variable, entrada;
    printf("Ingrese un numero: \n");
    scanf("%d",&entrada);
    variable=entrada%2==0?printf("El numero es par\n"):printf("El numero es impar\n");
    return 0;
}
```

### 2.3.2 CICLOS

Los ciclos permiten que un conjunto de instrucciones se repitan mientras se cumpla una cierta condición, siendo el primer tipo de forma de hacer ciclos, es utilizando la instrucción ***while***, la cual puede ver en el ejemplo de la Figura 2.21, la cual es una función que muestra por pantalla los números desde el 0 al n, siendo n un número ingresado por el usuario.

Figura 2.21: Programa con ciclo ***while***.

```c
#include <stdio.h>

int main()
{
    int i=0, n;1
    printf("Ingrese un numero:\n");
    scanf("%d",&n);
    while(i<=n){
        printf("El numero es: %d\n", i);
        i++;
    }
    return 0;
}
```

Otro tipo de ciclos existentes en los lenguajes de programación es el ciclo para o ***for***, el cual posee la siguiente estructura:

- ***for***(<inicialización>;<Condición>;<Incremento>); donde:
    - **<Inicialización>:** Es el valor inicial de una variable, puede ser la variable se inicializa acá o en algún punto anterior. Este elemento se puede omitir.
    - **<Condición>:** Es la condición mientras la cual el ciclo ***for*** se repetirá. Este elemento se puede omitir.
    - **<incremento>:** Es el cambio o instrucción que se realizará al final de realizar todas las instrucciones que estén dentro del ***for***. Este elemento se puede omitir.

Un ejemplo de cómo se escribe un ciclo ***for*** que realice lo mismo mostrado en el ciclo while, se puede ver en el código de la Figura 2.22.

Figura 2.22: Código de un ciclo ***for***.

```c
#include <stdio.h>

int main(){
    int n;
    printf("Ingrese un numero:\n");
    scanf("%d",&n);
    for(int i=0;i<=n;i++){
        printf("El numero es: %d\n", i);
    }
    return 0;
}
```

Existe otro tipo de ciclo ***for***, el llamado ***foreach***, que sirve para recorrer elementos iterables, y es el que se vió en el curso de Fundamentos de Programación como ***for in***. Este tipo de ciclo se verá más adelante en este curso.

## 2.4 Ejercicio resuelto

## 2.5 Contexto

Suponga que se debe solicitar un número al usuario, y mediante el uso de una función que reciba los parámetros por valor y otra, que realice lo mismo, pero reciba los parámetros por referencia, indique si el número ingresado por el usuario es o no primo.

## 2.6 Explicación de la solución

Para solucionar el problema, se realizan dos funciones, tal como se indica. Cómo un número n es primo solo si es divisible por los números 1 y n, si existe un divisor entre estos dos números, directamente el número no será primo. Las funciones retornarán 0 si es falso y 1 en caso verdadero.

## 2.7 Solución final

La solución se muestra en el código de la Figura 2.23.

Figura 2.23: Solución del ejercicio propuesto.

```c
#include <stdio.h>

int esPrimmoValor(int numero){
    for(int i=2;i<numero;i++){
        if(numero%i == 0)
            return 0;
    }
    return 1;
}

int esPrimmoReferencia(int *numero){
    for(int i=2;i<*numero;i++){
        if(*numero%i == 0)
            return 0;
    }
    return 1;
}

int main(){
    int n;
    printf("Ingrese un numero\n");
    scanf("%d",&n);
    int resValor = esPrimmoValor(n);
    int resReferencia = esPrimmoReferencia(&n);
    if(resValor==1)
        printf("Es primo segun la funcion por valor.\n");
    else
        printf("No es primo segun la funcion por valor.\n");
    if(resReferencia==1)
        printf("Es primo segun la funcion por referencia.\n");
    else
        printf("No es primo segun la funcion por referencia.\n");
    return 0;
}
```


# 3 EJERCICIOS PROPUESTOS

Realice todos estos ejercicios implementando funciones con paso por valor y paso por referencia.

## 3.1 EJERCICIO N°1:

Utilizando solo la operación **+**, implemente la operación potencia.

## 3.2 EJERCICIO N°2:

Utilizando solo la operación -, implemente la operación módulo.

## 3.3 EJERCICIO N°3:

Cree un programa en donde el usuario ingrese dos números a y b, y una opción, Si la opción es 1, el programa debe realizar la operación de a elevado a b, utilizando la función por valor del ejercicio 1, mientras que, si es 2, deberá entregar el módulo de a en b, utilizando la función por referencia del ejercicio 2. En caso de ingresar la opción 3, el programa deberá finalizar, mientras que sea cualquier otro número el programa deberá solicitar otra vez la opción al usuario, indicando que se ha ingresado una opción incorrecta. El programa debe finalizar solo cuando se ingresa la opción 3.