UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

PROGRAMACIÓN

Dr. Vladimir Villarreal

David, 03 de junio del 2017

TABLA DE CONTENIDO:

CAPITULO I. INTRODUCCIÓN A LA PROGRAMACIÓN

Objetivos:

- Conocer las estructuras básicas de los algoritmos para la resolución de problemas.
- Utilizar las estructuras básicas para la escritura de un algoritmo en la solución de un problema.

¿De qué trata esta sesión de aprendizaje?

En esta sesión de aprendizaje se presentan los conceptos básicos de la programación, una clasificación de los tipos de programación. Se explica además las instrucciones básicas que definen un algoritmo, operaciones básicas, ya la clasificación de las estructuras de control.

I. INTRODUCCION A LA PROGRAMACIÓN

1.1. Introducción a la programación

La principal razón para que las personas aprendan lenguajes y técnicas de programación es utilizar la computadora como una herramienta para resolver problemas.

La resolución de un problema exige al menos los siguientes pasos:

- Definición o análisis del problema.
- Diseño del algoritmo.
- Transformación del algoritmo en un programa.
- Ejecución y validación del programa.

Todo sistema de proceso de información hace referencia a tres elementos definidos en la figura 1:

Figura 1.1 Modelo de un sistema de proceso de información.

En este capítulo, explicaremos los elementos esenciales para aprender a programar, asociando las estructuras de programación con el uso de conceptos básicos.

1.2. Conceptos de lenguaje de programación

En este punto vamos a definir los conceptos básicos de los lenguajes de programación, así como sus tipos y clases.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, algoritmo proviene de Mohammed al–Khowarizmí, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el

enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra algorismus derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV antes de Cristo), que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al–Khowarizmí el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth –inventor de Pascal, Modula-2 y Oberon– tituló uno de sus más famosos libros, Algoritmos + Estructuras de datos = Programas, significándonos que sólo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

Los pasos para la resolución de un problema son:

- Diseño del algoritmo que describe la secuencia ordenada de pasos –sin ambigüedades– que conducen a la solución de un problema dado. (Análisis del programa y desarrollo del algoritmo.)
- Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (Fase de codificación.)
- Ejecución y validación del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan solo un medio

para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será el diseño de algoritmos.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, la solución de un problema se puede expresar mediante un algoritmo.

La definición de un algoritmo debe describir tres partes: Entrada, proceso y salida. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: Ingredientes y utensilios empleados

Proceso: Elaboración de la receta en la cocina.

Salida: Terminación del plato (por ejemplo, arroz con pollo).

Un ejemplo de un algoritmo se puede ver en el siguiente enunciado:

> *Un comprador ejecuta un pedido a una empresa. La empresa examina en su* base de datos el expediente del comprador, si el comprador es sujeto de crédito entonces la empresa acepta el pedido; en caso contrario, rechazará el pedido.

Los pasos del algoritmo son:

1. Inicio.
2. Leer el pedido.
3. Examinar el expediente del comprador.
4. Si el cliente es sujeto de crédito, aceptar pedido; en caso contrario, rechazar pedido
5. Fin

1.2.1 Tipos

Existen dos **tipos de lenguajes** claramente diferenciados; los **lenguajes** de bajo nivel y los de alto nivel. El ordenador sólo entiende un **lenguaje** conocido como código binario o código máquina, consistente en ceros y unos. Es decir, sólo utiliza 0 y 1 para codificar cualquier acción.

Lenguajes de bajo nivel

Son lenguajes totalmente dependientes de la máquina, es decir que el programa que se realiza con este tipo de lenguajes no se puede migrar o utilizar en otras máquinas. Al estar prácticamente diseñados a medida del hardware, aprovechan al máximo las características del mismo.

Dentro de este grupo se encuentran:

- *El lenguaje maquina*: este lenguaje ordena a la máquina las operaciones fundamentales para su funcionamiento. Consiste en la combinación de 0's y 1's para formar las ordenes entendibles por el hardware de la máquina.
- Este lenguaje es mucho más rápido que los lenguajes de alto nivel. La desventaja es que son bastantes difíciles de manejar y usar, además de tener códigos fuente enormes donde encontrar un fallo es casi imposible.
- *El lenguaje ensamblador* es un derivado del lenguaje máquina y está formado por abreviaturas de letras y números llamadas mnemotécnicos. Con la aparición de este lenguaje se crearon los programas traductores para poder pasar los programas escritos en lenguaje ensamblador a lenguaje máquina. Como ventaja con respecto al código máquina es que los códigos fuentes eran más cortos y los programas creados ocupaban menos memoria. Las desventajas de este lenguaje siguen siendo prácticamente las mismas que las del lenguaje ensamblador, añadiendo la dificultad de tener que aprender un nuevo lenguaje difícil de probar y mantener.

Lenguaje de alto nivel:

Son aquellos que se encuentran más cercanos al lenguaje natural que al lenguaje máquina. Están dirigidos a solucionar problemas mediante el uso de estructuras de datos. Se tratan de lenguajes independientes de la arquitectura del ordenador. Por lo que, en principio, un programa escrito en un lenguaje de alto nivel, lo puedes migrar de una máquina a otra sin ningún tipo de problema.

Estos lenguajes permiten al programador olvidarse por completo del funcionamiento interno de la maquina/s para la que están diseñando el programa. Tan solo necesitan un traductor que entiendan el código fuente como las características de la máquina. Suelen usar tipos de datos para la programación y hay lenguajes de propósito general (cualquier tipo de aplicación) y de propósito específico (como FORTRAN para trabajos científicos).

Existen muchos lenguajes de programación de alto nivel con sus diferentes versiones. Por esta razón es difícil su tipificación, pero una clasificación muy extendida desde el punto de vista de trabajar de los programas y la filosofía de su creación es la siguiente:

- **Lenguajes de programación imperativos:** entre ellos tenemos el Cobol, Pascal, C y Ada.
- **Lenguajes de programación declarativos:** el Lisp y el Prolog.
- **Lenguajes de programación orientados a objetos:** el Smalltalk y el C++.
- **Lenguajes de programación orientados al problema:** son aquellos lenguajes específicos para gestión.
- **Lenguajes de programación naturales:** son los nuevos lenguajes que pretender aproximar el diseño y la construcción de programas al lenguaje de las personas.

Otra clasificación de los lenguajes de programación de alto nivel, es teniendo en cuenta el desarrollo de las computadoras según sus diferentes generaciones:

- **Lenguajes de programación de primera generación:** el lenguaje máquina y el ensamblador.
- **Lenguajes de programación de segunda generación**: los primeros lenguajes de programación de alto nivel imperativo (FROTRAN, COBOL).
- **Lenguajes de programación de tercera generación:** son lenguajes de programación de alto nivel imperativo, pero mucho más utilizados y vigentes en la actualidad (ALGOL 8, PL/I, PASCAL, MODULA).
- **Lenguajes de programación de cuarta generación:** usados en aplicaciones de gestión y manejo de bases de dados (NATURAL, SQL).

- **Lenguajes de programación de quinta generación:** creados para la inteligencia artificial y para el procesamiento de lenguajes naturales (LISP, PROLOG).

1.2.2 Clases

A través de los años se han ido creando y modificando diferentes clases de lenguajes de programación, facilitando ya no solo la labor de los programadores, sino que algunos lenguajes gracias a su sencillez han permitido que personas no especializadas puedan crear aplicaciones afines a sus necesidades.

Entre estas clases de lenguaje se pueden mencionar los siguientes:

- **MATLAB:** es un lenguaje de alto nivel y una herramienta de software del tipo matemático (En sí al lenguaje se le designa comúnmente como lenguaje M o MATLAB). Es un lenguaje (y software), muy utilizado en el desarrollo industrial, de investigación y académico para la creación de códigos y programas especializados, principalmente en finanzas y programas científicos, por su carácter matemático, su disponibilidad para crear diversos gráficos, sumasiones y cálculos matemáticos complejos (por ejemplo, para la meteorología o en aplicaciones especiales para finanzas).
- **Pascal:** se trata de un lenguaje de alto nivel cuyo origen se remonta a los años 1968-1969, cuando se inicia este proyecto como una opción de enseñanza de programación para los alumnos, gracias a que es un lenguaje con una sintaxis sencilla, aunque muy estructurado, siendo muy utilizado aún para la enseñanza de la programación en sectores informáticos.
- **Python:** se trata tanto del lenguaje como de un programa para la creación de aplicaciones web y programas varios, es decir, hablamos de un lenguaje de propósito general, que se caracteriza por poseer cierta simplicidad y versatilidad en cuanto a su uso, así como por ser un lenguaje de programación interpretado, entendiéndose que su código no se debe compilar para su ejecución, lo que genera rapidez al trabajar, además es fácil de utilizar ya que cuenta con gran cantidad de librerías con las que se pueden realizar fácilmente las funciones de programación. Es de propósito general no está supeditado a un tipo específico de programación como seria por ejemplo la programación web, por lo que suele utilizársele para ello precisamente por su sencillez y rapidez para laborar con él.
- **Visual Basic:** el visual Basic es uno de los lenguajes informáticos más conocidos desde su creación a principios de los años 90s, esto porque fue desarrollado para la empresa

Microsoft que es la empresa de sistemas operativos y otros programas más difundida en el mundo, así como por estar enfocado a la creación de contenidos informáticos gráficos, por lo que este lenguaje se ha extendido con facilidad para la creación de aplicaciones compatibles con dicha tecnología de Microsoft.

- **Action Script:** este es un lenguaje de programación especializado en la plataforma Adobe Fash, lo que permite la creación de aplicaciones y comandos en dicha plataforma de una manera más eficaz.
- **ADA:** el lenguaje ADA cuenta con un alto nivel en la programación de software, por lo que suele ser empleado en la creación de programas con un alto nivel de confiabilidad, siendo usado por ejemplo para el desarrollo de softwares militares e industriales de alta precisión y de un costo elevado.
- **ASP:** este es un sub-lenguaje de programación enfocado a la creación de aplicaciones para servidores creado por Microsoft.
- **BASIC:** es uno de los primeros lenguajes de programación que se enfocaban a un público no especializado en informática ni en las variadas ecuaciones y cálculos necesarios para el manejo de las primeras computadoras. Fue diseñado en 1964 originalmente como un medio para facilitar la programación de computadoras para los estudiantes universitarios, pero posteriormente se fueron creando versiones mejoradas del lenguaje BASIC, para distintos campos de la programación siendo usado ampliamente durante las décadas de los 70s y 80s.
- **C:** se trata de un lenguaje de programación cuyo desarrollo se dio entre los años 1969 y1972, y está enfocado en la creación de sistemas operativos de computadoras. Fue el sustituto de los antiguos lenguajes de programación de las primeras computadoras (como COBOL u otros). De este lenguaje en particular se derivan el lenguaje C# y C++, que si bien cuentan con mejoras apreciables continúan teniendo como base los principios del lenguaje C.
- **C#:** es una versión consecutiva del lenguaje C, cuenta con la estructura del anterior, pero en distintas variaciones. Está enfocado a la formación de sistemas operativos.
- **C++:** es la versión más reciente derivada del lenguaje C, este posee varios conceptos, mecanismos y adaptaciones que mejoran la decodificación y creación de comandos, facilitándonos la creación de aplicaciones. Por medio de este se pueden realizar compilaciones de programas construidos con el lenguaje C pero no a la inversa.

- **Cobol:** es uno de los sistemas lingüísticos informáticos más antiguos ideados para el manejo de computadores. Se creó en 1960 como lenguaje compatible para los distintos sistemas informáticos existentes en aquel entonces.
- **Ensamblador o assembler: e**ste lenguaje de programación se utiliza para la programación de circuitos integrados, microprocesadores, micro controladores y diversos circuitos integrados en computadoras y otros aparatos. Se trata de un lenguaje de bajo nivel cuya estructura se acerca mucho al lenguaje utilizado por las máquinas, es decir, al código binario.
- **Fortran:** es un lenguaje de programación de alto nivel, esto es, se encuentra adaptado y se usa para la creación de programas y aplicaciones propios de la computación de carácter científico, por estar adaptado al cálculo numérico. Se le utiliza en la creación de aplicaciones de uso científico y técnico, como en ingeniería, astronomía, las matemáticas financieras y crear aplicaciones para realizar cálculos muy complicados. Se han ido creando versiones mejoradas como: FORTRAN IV, FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008.
- **J# o también llamado o J-sharp: s**e trata de un lenguaje "transicional" o intermedio del lenguaje de programación Java.
- **Java Script: e**ste lenguaje es un sub-lenguaje o "dialecto" derivado del ECMAScript, que se encuentra enfocado a objetos, su sintaxis y estructura posen similitudes con el lenguaje C, C#, y C++ sin embargo presenta características propias.
- **JAVA:** Java es tanto un lenguaje de programación como una plataforma o tecnología informática. Como lenguaje de programación se le utiliza en la creación de software tanto en equipos de cómputo como en otros dispositivos digitales como celulares y demás, así también es usado para crear programas den distintas arquitecturas como la PC y la Mac, y en diversos sistemas operativos como Windows, Solaris, Unix, Linux y OS.
- **LISP:** en este caso hablamos de la familia de dialectos de programación conocidos genéricamente como LISP, se trata de un lenguaje sencillos o cuya estructura es relativamente simple. Es uno de los lenguajes de programación más antiguos ya que se inició con el proyecto de este, en 1958 derivándose dicho proyecto en un lenguaje cuyo código fuente se compone de "listas" amanera de estructura de datos.

- **Oracle PL/SQL: e**ste se focaliza en crear aplicaciones para la web especializadas en el manejo de bases de datos relacionales. Es un lenguaje y una tecnología perteneciente a la empresa Oracle.

Otros de los lenguajes de programación son: Logo, Perl, PHP, Prolog, y SHELL's de UNIX.

1.3 Instrucciones básicas de algoritmos

El proceso de diseño del algoritmo o posteriormente de codificación del programa consiste en definir las acciones o instrucciones que resolverán al problema. Las acciones o instrucciones se deben escribir y posteriormente almacenar en memoria en el mismo orden en que han de ejecutarse, es decir, en secuencia. Un programa puede ser lineal o no lineal. Un programa es lineal si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones.
Un programa es no lineal cuando se interrumpe la secuencia mediante instrucciones de bifurcación.

1.3.1Operaciones básicas

Todo programa que se haga pasará por secciones o pasos, como lo son entrada de datos, procesos, operaciones de cálculo y la salida de datos o información. A continuación, se muestran estos procesos más detallados:

1.3.1.1 Acceso a los datos

El acceso a los datos consiste en recibir desde un dispositivo de entrada (p.ej. el teclado) un valor o dato. Este dato va a ser almacenado en la variable que aparece a continuación de la instrucción. Esta operación se representa así:

Figura 1.2: Representación de acceso a los datos

1.3.1.2 Operaciones de Cálculo

Consiste, en el paso de valores o resultados a una zona de la memoria. Dicha zona será reconocida con el nombre de la variable que recibe el valor. Se pueden clasificar de la siguiente forma:

- Simples: Consiste en pasar un valor constante a una variable (a ← 15)
- Contador: Consiste en usar un verificador del número de veces que se realiza un proceso

(a ← a + 1)

- Acumulador: Consiste en usar un sumador en un proceso (a ← a + b)
- De trabajo: Donde puede recibir el resultado de una operación matemática que involucre muchas variables (a ← c + b*2/4). En general el formato a utilizar es el siguiente:

< Variable > ← <valor o expresión >

El símbolo ← debe leerse asigne

1.3.1.3 Salida de información

Consiste en mandar por un dispositivo de salida (p.ej. monitor o impresora) un resultado o mensaje. Esta instrucción presenta en pantalla el mensaje escrito entre comillas o el contenido de la variable. Este proceso se representa, así como sigue:

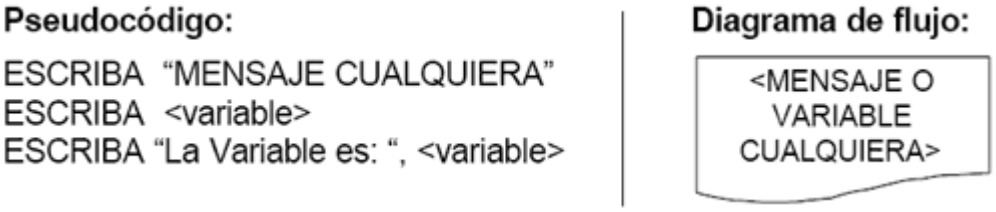

Figura 1.3: Representación de salida de información

1.3.2 Estructura de control

Existen tareas complejas que no pueden ser resueltas empleando un esquema sencillo, en ocasiones es necesario repetir una misma acción un número determinado de veces o evaluar una expresión y realizar acciones diferentes en base al resultado de dicha evaluación.

Para resolver estas situaciones existen las denominadas estructuras de control que poseen las siguientes características:

• Una estructura de control tiene un único punto de entrada y un único punto de salida.

• Una estructura de control se compone de sentencias o de otras estructuras de control.

Tales características permiten desarrollar de forma muy flexible todo tipo de algoritmos aun cuando sólo existen tres tipos fundamentales de estructuras de control:

• Secuencial.

• Alternativa.

• Repetitiva

1.3.2.1 Secuencial

Una estructura de control secuencial, en realidad, no es más que escribir un paso del algoritmo detrás de otro, el que primero se haya escrito será el que primero se ejecute. Dicho de otra forma, la estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia, y las tareas se suceden de tal modo que la salida de una es la entrada de otra y así sucesivamente hasta el fin del proceso.

El flujo del programa coincide con el orden físico en el que se han ido poniendo las instrucciones. Dentro de este tipo podemos encontrar operaciones de inicio/fin, inicialización de variables, operaciones de asignación, cálculo, etc.

Su representación y el diagrama de flujo se muestra en la Figura.

```
...
Instrucción 1
Instrucción 2
Instrucción 3
...
```

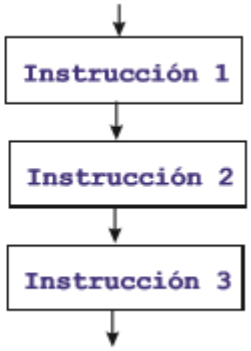

Figura 1.4: Representación de la estructura de control secuencial

1.3.2.2 Alternativa

Las estructuras alternativas controlan si una sentencia o bloque de sentencias se ejecutan, en función del cumplimiento o no de una condición o expresión lógica.

C++ tiene dos estructuras de control alterntivas, **if** y **switch**.

Existen 3 tipos de estructuras alternativas: alternativa simple, alternativa doble y alternativa múltiple.

- **Alternativa Simple:** Se realiza una acción o conjunto de acciones si se cumple una determinada condición.
- **Alternativa doble:** Si una condición se cumple se realizan unas acciones, si no se cumple la condición, se realizan otras.
- **Alternativa múltiple:** Evalúa una expresión que pueda tomar n valores (enteros, caracteres y lógicos, pero nunca reales) y ejecuta una acción o grupo de acciones diferente en función del valor tomado por la expresión selectora.

Ejemplo:

```
Según mes
caso 1,3,5,7,8,10,12:
escribir '31'
caso 4,6,9,11:
escribir '30'
caso 2:
escribir '28'
otro caso:
escribir 'Mes incorrecto'
fin según
```

```
select case (mes)
case (1,3,5,7,8,10,12)
print
*, '31'
case (4,6,9,11)
print
*, '30'
case (2)
print
*, '28'
case
default
print
*, 'Mes incorrecto'
end select
```

1.3.2.3 Repetitiva

Las estructuras de control repetitivas son aquellas que permiten ejecutar un conjunto de instrucciones varias veces, de acuerdo con el valor que genere la expresión relacional y/o lógica. Esto significa que una instrucción repetitiva permite saltar a una instrucción anterior para volver a ejecutarla.

A las estas estructuras se les conoce también como ciclos o bucles, por su funcionamiento. Existen 3 estructuras repetitivas:

1. **While**
2. **Do-while**
3. **For**

Las tres instrucciones tienen el mismo fin, y difieren únicamente en su sintaxis, siendo posible sustituir una solución en la que se utiliza "while", por una en la que se utiliza "do-while" o "for".

Las estructuras de control repetitivas utilizan dos tipos de variables: Contadores y Acumuladores.

Contadores:

Un contador es una variable de tipo entero, que incrementa o decrementa su valor de forma **constante** y requiere ser inicializada generalmente en 0 o 1, aunque en realidad depende del problema que se está resolviendo. Como su nombre lo indica se utilizan en la mayoría de las veces para contar el número de veces que se ejecuta una acción, o para contar el número de veces que se cumple una condición (expresión relacional/lógica).

Por ejemplo, si se desea sumar los números del 1 al 5, se necesitará una variable que genere esos números, es decir que empiece en 1 y llegue hasta el 5.

Acumuladores:

Un acumulador es una variable numérica, que incrementa o decrementa su valor de forma **no constante** y requiere ser inicializada. Como su nombre lo indica se utilizan para acumular valores en una sola variable, ya sea de suma o producto. Por lo tanto, existen dos modos de inicialización:

- Para Suma: Inicializar en 0
- Para Producto: Inicializar en 1

Esto con el objetivo de no alterar los valores de las respectivas operaciones.

Ejemplo: Si se desea conocer el acumulado de los pagos realizados a un grupo de empleados, se necesitará una variable que vaya sumando los sueldos de cada empleado, se requiere una variable que permita calcular el acumulado.

CAPÍTULO II. INTRODUCCIÓN AL LENGUAJE C

Objetivos:

- Conocer las estructuras básicas de los algoritmos para la resolución de problemas.
- Utilizar las estructuras básicas para la escritura de un algoritmo en la solución de un problema.
- Presentar y desarrollar problemas prácticos usando las distintas variables y ciclos propios del lenguaje C.

¿De qué trata esta sesión de aprendizaje?

En esta sesión de aprendizaje se presentan los conceptos básicos de la programación, una clasificación de los tipos de programación. Se explica además las instrucciones básicas que definen un algoritmo, operaciones básicas, ya la clasificación de las estructuras de control.

CAPÍTULO II. INTRODUCCIÓN AL LENGUAJE C

2.1 Introducción al lenguaje C

El lenguaje C fue diseñado en los años sesenta por Dennis Ritchie, de los laboratorios Bell. El propósito era convertirse en el lenguaje del sistema operativo UNIX.

Apareció a partir de dos lenguajes de programación de sistemas anteriores, el BCPL, y B. En 1978 Kernighan y Ritchie publican el manual de uso del lenguaje C en el libro "The C Programming Language", versión que es llamada hoy en día "K&R C".

A mediados de los ochenta había en el mercado numerosos compiladores C, lo cual hizo que muchos fabricantes introdujeran nuevas mejoras en el lenguaje. Todas estas mejoras fueron recogidas por un comité de estandarización ANSI, formando así las especificaciones del lenguaje C estándar, conocido hoy en día como "ANSI C".

El lenguaje C tuvo su éxito enorme al combinar las ventajas de los lenguajes compilados y los ensambladores. Entre estas ventajas se cuentan las siguientes:

- El lenguaje C es un lenguaje de propósito general. Puede ser utilizado para la programación de una gran variedad de aplicaciones, desde bases de datos a juegos, pasando por cualquier programa científico, administrativo o de comunicaciones.
- Es un lenguaje fácilmente transportable. Su diseño original, al utilizar funciones de librería para realizar tareas que dependan de la arquitectura del ordenador, permite compilar un programa en casi cualquier máquina con mínimas modificaciones.
- Es un lenguaje de nivel medio. Reúne las características tanto del lenguaje de alto nivel como el de bajo nivel, permitiendo un dominio completo de la máquina con el uso del ensamblador, además permite utilizar conceptos de programación estructurada como los usan los lenguajes de alto nivel.
- El lenguaje C es modular. Los programas pueden escribirse en módulos, almacenando cada módulo en un fichero aparte y compilándolo separadamente. Luego, cada una de estas unidades compiladas puede combinarse en un programa entero.
- Es un lenguaje conciso. Lo compacto del código escrito en C ahorra espacio en disco, tiempo de compilación y tiempo que el programador invierte en introducir el programa en el ordenador.

- Es un lenguaje muy adecuado para aplicaciones que necesiten acceder a los recursos del sistema. Esto lo hacen muy atractivo en aplicaciones de ingeniería, ya que permite controlar con relativa facilidad dispositivos externos, como por ejemplo tarjetas controladoras.

2.2 Reglas generales de C

Las buenas prácticas o normas de programación son un conjunto de reglas, de carácter opcional u obligatorias. Se implementan con el fin último de mejorar la calidad del software desarrollable. Podremos obtener gracias a ellas beneficios como:

- Facilitar la etapa de desarrollo para el programador o programadores
- La legibilidad y mantenibilidad del código generable mejorarán en gran medida
- Cierta clase de errores comunes podrán ser evitados de inicio

Estructura del programa

- El código deberá presentarse indentado dentro de cada estructura de control, como pueden ser los bloques **if**, **for**, **while**, etc., y también para el contenido de funciones, registros, constructores, etc. De este modo logramos resaltar de forma visual una estructura lógica del código, lo cual simplifica enormemente su lectura.
- Siempre utilice llaves en todas las estructuras de control, aún incluso si sólo tienen 1 instrucción en su interior. Con esto se evitarán posibles errores si, posteriormente, deseamos añadir más instrucciones bajo la estructura de control afectada. La única excepción aceptable a esta regla es cuando ponemos la instrucción a ejecutar en la misma línea que la instrucción de control.
- Deberemos usar de forma consistente un estilo de apertura y cierre de llaves.
- No se escribirá código justo después de una llave de apertura.
- Sólo se insertará una instrucción por línea.
- Deberá emplearse un espacio después de las comas.
- Usaremos paréntesis para especificar el orden de prioridades en operaciones aritméticas complejas, evitando que el programador deba recordar todas las reglas de precedencia de operadores. La única excepción aceptable es para los índices de los arreglos.

- Separaremos los operadores binarios con espacios a ambos lados, ya que de este modo se resalta el operador y se facilita la lectura del programa.
- Deberemos elegir desde el principio entre utilizar o no espacios antes de cada llamada a función y uso de una estructura de control.
- Deberemos evitar especialmente el uso de instrucciones que no se ejecuten nunca o que su resultado no sea válido.
- Toda instrucción **switch** deberá disponer de un caso **default**, siendo éste el último caso.

Variables y usos

- Como norma general, nunca debemos usar variables globales.
- Las variables usadas como índices en iteraciones serán la i, j, k… típicamente darán comienzo en la i.
- Su nombrado deberá dar una idea del uso que se le dará a la variable.
- Es recomendable usar un formato **camelCase** para el caso de variables de nombre compuesto. De esta forma, si tenemos una variable que almacenará el resultado de una suma, podría llamarse con formato camelCase como **sumaDigitos** por ejemplo.
- Los valores constantes deberán declararse al comienzo del programa usando **#define** y su nombre deberá escribirse en mayúsculas.
- Toda variable debería iniciarse con algún valor en su declaración.
- Para el caso de variables tipo puntero, siempre deberán inicializarse a **NULL**.

Sobre los comentarios

- Todo fichero de código del programa debe comenzar con un comentario que describa su propósito, fecha de creación y modificación, motivos de las modificaciones, autor o autores, etc.
- Cada bloque de código comenzará con un bloque de comentarios propio en el que al menos se detalle su propósito, los parámetros de entrada que use, posibles valores de retorno, códigos de error o retorno que genere, etc. La parte de los parámetros de entrada y salida estarán bien separados.
- Es muy importante actualizar los comentarios de cada bloque de código que modifiquemos, siendo una cuestión prioritaria.

- Incluiremos un espacio en blanco después de la apertura del bloque de comentarios y antes del cierre.
- En caso de referirnos a una variable dentro del bloque de comentarios, usaremos comillas simples. Por ejemplo: /* La variable 'sumaItems' representa el sumatorio total de 'Items' en el array */
- Cada bloque de comentario se refiere al código que le precede inmediatamente y por ello compartirá su mismo nivel de indentación.
- Deberemos separar los párrafos dentro de un bloque de comentarios con una línea en blanco.

2.3 Elementos básicos de C

Básicamente el lenguaje C está compuesto por los siguientes elementos

• Constantes
• Identificadores
• Palabras reservadas
• Comentarios
• Operadores

Para representar estos elementos se utilizan los caracteres habituales (letras, números, signos de puntuación, subrayados, ...) aunque no todos los elementos pueden usar todos estos caracteres.
Una característica importante del Lenguaje C es que en todos los elementos anteriormente enumerados distingue letras mayúsculas y minúsculas. Así, int es una palabra reservada del lenguaje que sirve para declarar variables enteras, mientras que Int podría ser el nombre de una variable.

2.3.1 Identificadores y palabras reservadas

Antes de proceder a explicar los identificadores en C, es necesario resaltar que C es un lenguaje sensible al contexto, a diferencia por ejemplo de Pascal, por lo cual, C diferencia entre mayúsculas y minúsculas, y, por tanto, diferencia entre una palabra escrita total o parcialmente en mayúsculas y otra escrita completamente en minúsculas.

En el lenguaje C, un identificador es cualquier palabra no reservada que comience por una letra o por un subrayado, pudiendo contener en su interior letras, números y subrayados. La longitud máxima de un identificador depende del compilador que se esté usando, pero, generalmente, suelen ser de 32 caracteres, ignorándose todos aquellos caracteres que compongan el identificador y sobrepasen la longitud máxima. Recuérdese, además, que, al ser C sensible al contexto, un identificador escrito como **esto_es_un_ident** y otra vez como **Esto_Es_Un_Ident** será interpretado como dos identificadores completamente distintos.

En C, como en cualquier otro lenguaje, existen una serie de palabras clave (keywords) que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas palabras sirven para indicar al computador que realice una tarea muy determinada (desde evaluar una comparación, hasta definir el tipo de una variable) y tienen un especial significado para el compilador. El C es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación, se presenta la lista de las palabras clave de C, es importante evitarlas como identificadores):

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

Tabla 2.1: palabras reservadas del lenguaje C.

2.4 Tipos de datos básicos

Un tipo de dato se define como un conjunto de valores que puede tener unas variables, junto con ciertas operaciones que se pueden realizar con ellas.

Tipo de dato	Descripción.
char	Carácter o entero pequeño (byte)
int	Entero
float	Punto flotante
double	Punto flotante (mayor rango que *float*)
void	Sin tipo (uso especial)

Tabla 2.2: tipos de datos en C.

2.4.1 Constantes

Las constantes, como su nombre lo indica, son valores que se mantiene invariables durante la ejecución del programa.

En C, las constantes se refieren a los valores fijos que el programa no puede alterar. Algunos ejemplos de constantes de C son:

Tipo de dato	Constantes de ejemplo		
char	'a'	'9'	'Q'
int	1	-34	21000
long int	-34	67856L	456
short int	10	-12	1500
unsigned int	45600U	345	3
float	12.45	4.34e-3	-2.8e9
double	-34.657	-2.2e-7	1.0e100

Tabla 3.3: Ejemplos de constantes

Su formato es el siguiente:

const tipo_de_dato nombre= valor;

donde **const,** es una palabra reservada, para indicarle al compilador que se está declarando una constante**.**

Ejemplo:

const int dia=7;

const float pi=3.14159;

const char caracter= 'm';

const char fecha[]="25 de diciembre";

Caso Especial Constantes Simbólicas

Las constantes simbólicas, se declaran mediante la directiva #define, como se explicó anteriormente. Funcionan de la siguiente manera, cuando C, encuentra el símbolo que representa a la constante, lo sustituye por su respectivo valor.

Ejemplo:

#define N 150

#define PI 3.1416

#define P 50

2.4.2 Variables

Una Variable, como su nombre lo indica, es capaz de almacenar diferentes valores durante la ejecución del programa, su valor varía. Es un lugar en la memoria el cual, posee un nombre (identificador), y un valor asociado.

La declaración de variables en **C,** se hace en minúsculas.

Formato:

Tipo_de_dato nombre_de_la_variable;

Ejemplos:

*Declare una variable de tipo entero y otra de tipo real, una con el nombre de "x" y otra con el identificador "y":

```
int x;
float y;
```

*Declare una variable de tipo entero llamada moon, e inicialícela con un valor de 20

```
int x = 20;
```

*Declare una variable de tipo real, llamada Pi, e inicialícela con un valor de 3.1415

```
float pi=3.1415;
```

*Declare una variable de tipo carácter y asígnele el valor de "M"

```
char car = 'M';
```

*Declare una variable llamada nombre, que contenga su nombre:

```
char nombre[7]="Manuel";
```

Explicación:

C, no tiene el tipo de dato llamado string, o mejor conocido como cadenas de texto, pero nosotros podemos hacer uso de ellas, por medio de un *arreglo,* (de lo cual hablaremos con más detalle, posteriormente); pero para declarar este tipo de datos colocamos el tipo de datos, es decir la palabra reservada *char* luego el nombre, e inmediatamente abrimos, entre corchetes, va el número de letras, que contendrá dicha variable. Es muy importante que al momento de declarar el tamaño, sea un número mayor, al verdadero número de letras; por ejemplo, la palabra "Manuel", solo tiene 6 letras, pero debemos declararlo para 7 letras ¿Por qué?

Veámoslo gráficamente, en la memoria, C crea un variable llamada nombre y esta posee la palabra Manuel, así:

M	a	n	u	e	l	\0
0	1	2	3	4	5	6

Tabla 2.4: Ejemplo de declaración de variable en memoria.

En realidad, hay 7 espacios, pero la cuanta llega hasta 6, porque c, toma la primera posición como la posición cero, y para indicar el final de la cadena lo hace con un espacio en blanco.

2.4.3 Declaraciones

En C, toda variable, antes de poder ser usada, debe ser declarada, especificando con ello el tipo de dato que almacenara. Toda variable en C se declara de la forma:

<tipo de dato> <nombre de variable> [, nombre de variable];

En C, las variables pueden ser declaradas en cuatro lugares del módulo del programa:

• Fuera de todas las funciones del programa, son las llamadas variables globales, accesibles desde cualquier parte del programa.

• Dentro de una función, son las llamadas variables locales, accesibles tan solo por la función en las que se declaran.

• Como parámetros a la función, accesibles de igual forma que si se declararan dentro de la función.

• Dentro de un bloque de código del programa, accesible tan solo dentro del bloque donde se declara. Esta forma de declaración puede interpretarse como una variable local del bloque donde se declara.

Para una mejor comprensión, veamos un pequeño programa de C con variables declaradas de las cuatro formas posibles:

```
#include <stdio.h>
int sum; /* Variable global, accesible desde cualquier parte */
/* del programa*/
void suma(int x) /* Variable local declarada como parámetro, */
/* accesible solo por la función suma*/
{
sum=sum+x;
```

```
return;
}
void intercambio(int *a,int *b)
{
if (*a>*b)
{
int inter; /* Variable local, accesible solo dentro del */
/* bloque donde se declara*/
inter=*a;
*a=*b;
*b=inter;
}
return;
}
int main(void) /*Función principal del programa*/
{
int contador,a=9,b=0; /*Variables locales, accesibles solo */
/* por main*/
sum=0;
intercambio(&a,&b);
for(contador=a;contador<=b;contador++) suma(contador);
printf("%d\n",suma);
return(0);
}
```

2.5 Entrada y salida de datos

Se refiere a las operaciones que se producen en el teclado y en la pantalla de la computadora. En C no hay palabras claves para realizar las acciones de Entrada/Salida, estas se hacen mediante el uso de las funciones de la biblioteca estándar (stadio.h). Para utilizar las funciones de E / S debemos incluir en el programa el archivo de cabecera, ejemplo: stdio.h, mediante la declaratoria:

#include <stdio.h>

2.5.1 Scanf()

La función scanf() es, en muchos sentidos, la inversa de printf(). Puede leer desde el dispositivo de entrada estándar (normalmente el teclado) datos de cualquier tipo de los manejados por el compilador, convirtiéndolos al formato interno apropiado. Funciona de manera análoga a printf(), por lo que su sintaxis es:

scanf(cadena_de_formato, datos);

El prototipo de scanf() se encuentra en el archivo de cabecera stdio.h (de "std" =standard e "io" = input/output, es decir, entrada/salida)La cadena_de_formato tiene la misma composición que la de printf(). Los datos son las variables donde se desea almacenar el dato o datos leídos desde el teclado. ¡Cuidado! Con los tipos simples, es necesario utilizar el operador & delante del nombre de la variable, porque esa variable se pasa por referencia a scanf() para que ésta pueda modificarla.

Por ejemplo:

```
int a, b;
float x;
scanf("%d", &a);
scanf("%d%f", &b, &x);
```

La primera llamada a scanf() sirve para leer un número entero desde teclado y almacenarlo en la variable a. La segunda llamada lee dos números: el primero, entero, que se almacena en b; y, el segundo, real, que se almacena en x.

Ejemplo: Se trata de un algoritmo que lee dos números enteros, A y B. Si A es mayor que B los resta, y en otro caso los suma.

```
/* Programa suma y resta */
#include <stdio.h>
int main()
{
int a, b;
printf ("Introduzca dos números enteros\n");
scanf("%d%d", &a, &b);
if (a < b);
printf("La suma de %d y %d es: %d", a, b, a+b);
else
printf("La resta de %d menos %d es: %d", a, b, a–b);
return 0;
}
```

2.5.2 Get(), getch(), getche

Lee un carácter del teclado, espera un retorno, es decir un enter y el eco aparece o no, es decir, la tecla presionada. Estas instrucciones se encuentran en la biblioteca conio.h

Getch() No aparece el eco

Getche() Aparece el eco

Ejemplo 1:

```
#include <stdio.h>
#include <conio.h>
main()
{char car;
clrscr(); /*Se encarga de borrar la pantalla por eso se
llama clear screen*/
car=getchar();
```

```
putchar(car+1);
getch();
return 0;
}
```

Ejemplo 2:

```
#include <stdio.h>
#include <conio.h>
main()
{char x; /*Declaramos x como caracter*/
printf("Para Finalizar Presione cualquier Tecla:");
x= getchar();/*Captura y muestra el caracter presionado*/
getch();/*Espera a que se presione cualquier otra tecla para finalizar*/
return 0;
}
```

2.5.3 Printf()

La función printf() se usa para escribir cualquier tipo de dato a la pantalla. Su formato es:

int printf(const char *formato[,argumento,...]);

La cadena apuntada por formato consta de dos tipos de elementos. El primer tipo está constituido por los caracteres que se mostraran en pantalla. El segundo tipo contiene las ordenes de formato que describen la forma en que se muestran los argumentos. Las ordenes de formato están precedidas por el signo % y le sigue el código de formato.

Especificador	Descripción
%c	Carácter.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Punto flotante en notación científica (e minúscula).
%E	Punto flotante en notación científica (E mayúscula).
%f	Punto flotante.
%g	Usar el más corto de %e y %f.
%G	Usar el más corto de %E y %f.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimales sin signo (letras minúsculas).
%X	Hexadecimales sin signo (letras mayúsculas).
%p	Mostrar un puntero.
%n	El argumento asociado es un puntero a un entero, el cual recibirá el número de caracteres escritos.
%%	Imprimir el signo %.

Tabla 2.5: Especificadores de formato de la función printf().

La función printf() devuelve el número de caracteres escritos. En caso de error devuelve el valor EOF.

Ejemplo:

```
#include <stdio.h>
int main(void)
{
int a,b;
printf("\nIntroduce el valor de a: ");
scanf("%d",&a);
printf("\nIntroduce el valor de b: ");
scanf("%d",&b);
if (b!=0)
printf("\nEl valor de %d dividido %d es: %f\n",
a,b,(float)a/b);
else
printf("\nError, b vale 0\n");
return 0;
}
```

2.5.4 Put(), Putchar()

Putchar(): Imprime un carácter en la pantalla, en la posición actual del cursor. Su uso en sencillo y generalmente está implementada como una macro en la cabecera de la biblioteca estándar.

Ejemplo:

```
#include <stdio.h>

main()
{
        putchar('H');
        putchar('o');
        putchar('l');
        putchar('a');

        putchar(32);

        putchar('m');
        putchar('u');
        putchar('n');
        putchar('d');
        putchar('o');

        putchar('\n');
}
```

El resultado es:

Hola mundo

En el código anterior putchar(32); muestra el espacio entre ambas palabras (32 es el código ASCII del carácter espacio ' ') y putchar('\n'); imprime un salto de línea tras el texto.

2.5.5 Aplicación práctica de conceptos (laboratorios)

1. Escribir en lenguaje C un programa que:

1º) Pida por teclado el nombre (dato cadena) de una persona.

2º) Muestre por pantalla el mensaje: "Hola <nombre>, buenos días.".

Solución:

```
/* Programa: Saludo */
#include <conio.h>
#include <stdio.h>

int main()
{
   char nombre[20];

   printf( "Introduzca su nombre: " );
   scanf( "%s", nombre );
   printf( "Hola %s, buenos d%cas.", nombre, 161 );

   getch(); /* Pausa */

   return 0;
}
```

2. Programa que imprime en pantalla: HOLA COMO ESTAS.

```
#include <stdio.h>
#include <conio.h>
int main()
{
printf(“HOLA\n”);
printf(“COMO ESTAS\n”);
getch();
return 0;
```

```
}
```

3. Programa que lee 2 números, los suma, imprime el resultado de la suma y lo multiplica por 2.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int num1, num2, res1, res2;
printf("Anote el primer numero\n");
scanf("%i",&num1);
printf("Anote el segundo numero\n");
scanf("%i",&num2);
res1=num1+num2;
res2=res1*2;
printf("El resultado de la suma es: %i \n y el de la multiplicación es: %i",res1, res2);
getch();
return 0;
}
```

4. Escribir en lenguaje C un programa que:

1º) Pida por teclado una hora en horas, minutos y segundos (datos enteros).

2º) Calcule cuántos segundos han pasado desde las 0:0:0 horas.

3º) Muestre por pantalla el resultado (dato entero).

Nota: Se asume que la hora introducida es correcta.

```
#include <conio.h>
#include <stdio.h>

int main()
{
   int horas, minutos, segundos, total;
```

```
   printf( "\n   Introduzca horas: " );
   scanf( "%d", &horas );
   printf( "\n   Introduzca minutos: " );
   scanf( "%d", &minutos );
   printf( "\n   Introduzca segundos: " );
   scanf( "%d", &segundos );

   total = horas * 3600 + minutos * 60 + segundos;

   printf( "\n   Desde las 0:0:0 horas han pasado %d segundos.", total );

   getch(); /* Pausa */

   return 0;
}
```

5. Escribir en lenguaje C un programa que:

1º) Pida por teclado tres números (datos enteros) y sean almacenados en tres variables, llamadas v1, v2 y v3.

2º) Intercambie los valores de las variables de la siguiente manera:

- **El contenido de v1 pasa a v2.**
- **El contenido de v2 pasa a v3.**
- **El contenido de v3 pasa a v1.**

3º) Muestre por pantalla los valores contendidos en las variables.

```
#include <conio.h>
#include <stdio.h>
int main()
{
   int auxiliar, v1, v2, v3;
```

```
    printf( "\n   Introduzca el valor de v1: " );
    scanf( "%d", &v1);
    printf( "   Introduzca el valor de v2: " );
    scanf( "%d", &v2);
    printf( "   Introduzca el valor de v3: " );
    scanf( "%d", &v3);

    printf( "\n   Intercambiando los valores..." );

    auxiliar = v3;
    v3 = v2;
    v2 = v1;
    v1 = auxiliar;

    printf( "\n\n   Ahora, el valor de v1 es: %d", v1 );
    printf( "\n   Ahora, el valor de v2 es: %d", v2 );
    printf( "\n   Ahora, el valor de v3 es: %d", v3 );

    getch(); /* Pausa */

    return 0;
}
```

2.6 Operaciones y expresiones

Una expresión es una fórmula matemática cuya evaluación especifica un valor. Los elementos que constituyen una expresión son: constantes, variables y operadores.

2.6.1 Operadores aritméticos

Los operadores aritméticos existentes en C son, ordenados de mayor a menor precedencia:

Operador		Operador		Operador	
++	Incremento	--	Decremento		
-	Menos unario				
*	Multiplicación.	/	División	%	Módulo
+	Suma	-	Resta		

Tabla 2.6: operadores aritméticos de C

Los operadores ++, -- y % solo pueden usarse con datos de tipo int o char. El operador incremento (++), incrementa en una unidad el valor de la variable sobre la que se aplica, el operador decremento (--), decrementa en una unidad el valor de la variable, y el operador módulo (%), calcula el resto de una división de dos variables de tipo entero o carácter.

Un aspecto que conviene explicar es el hecho de que los operadores incremento y decremento pueden preceder o posceder a su operando, lo cual permite escribir, si x es una variable de tipo int, las expresiones ++x o x++. Usado de forma aislada no presenta ninguna diferencia, sin embargo, cuando se usa en una expresión existe una diferencia en el orden de ejecución del mismo. Cuando el operador incremento (o decremento) precede al operando, C primero realiza el incremento (o decremento), y después usa el valor del operando, realizándose la operación al contrario si el operador poscede al operando.

2.6.2 Operadores monarios

C incluye una clase de operadores que actúan sobre un solo operando para producir un nuevo valor. Estos operadores se denominan *operadores monarios.* Los operadores monarios suelen preceder a su único operando, aunque algunos operadores monarios se escriben detrás de su operando.

Es probable que el operador monario de uso más frecuente sea el *menos monario,* que consiste en un signo menos delante de una constante numérica, una variable o una expresión. (Algunos lenguajes de programación permiten que se incluya el signo menos como parte de una constante numérica. Sin embargo, en C todas las constantes numéricas son positivas. Por tanto, un número negativo es en realidad una expresión, que consiste en el operador monario menos, seguido de una constante numérica positiva.)

Adviértase que la operación menos monaria es distinta del operador aritmético que representa la resta (-). El operador resta requiere dos operandos**.**

Ejemplo 1:

Se ilustra varios ejemplos del uso de la operación menos monaria.

-743	0X7FFF	-0.2	-5E-8
-raiz1	- (X + Y)	-3 * (X + Y)	

Tabla 2.7: Ejemplo de operadores monarios

En cada caso, el signo menos es seguido por un operando numérico que puede ser una constante entera, una constante en coma flotante, una variable numérica o una expresión aritmética.

Otros dos operadores monarios de uso frecuente son el *operador incremento,* + +, y el *operador decremento,* - -. El operador incremento hace que su operando se incremente en uno, mientras que el operador decremento hace que su operando se decremente en uno. El operando utilizado con cada uno de estos operadores debe ser una variable simple.

Ejemplo 2:

Supongamos que i es una variable entera que tiene asignado el valor 5. La expresión ++i, que es equivalente a escribir i = i + 1, hace que el valor de i sea 6. Análogamente la expresión --i, que es equivalente a i = i - 1, hace que el valor (partiendo del original) de i pase a ser 4.

Los operadores incremento y decremento se pueden utilizar, cada uno de ellos, de dos formas distintas, dependiendo de si el operador se escribe delante o detrás del operando. Si el *operador precede* al operando (por ejemplo ++i), el valor del operando se modificará *antes* de que se utilice con otro propósito. Sin embargo, si el operador *sigue* al operando (por ejemplo i++), entonces el valor del operando se modificará *después* de ser utilizado.

2.6.3 Operadores racionales y lógicos

Los operadores relacionales se utilizan principalmente para elaborar condiciones en las sentencias condicionales e iterativas.

La tabla muestra los distintos operadores de relación en C. Al relacionar (comparar) dos expresiones mediante uno de estos operadores se obtiene un resultado lógico, es decir: “CIERTO” o “FALSO”.

Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=
Igual que	==
Distinto que	!=

Conjunción ó Y lógico	&&
Disyunción u O lógico	||
Negación ó NO lógico	!

Tabla 2.8: operadores relacionales (izquierda), operadores lógicos (derecha).

Por ejemplo, la expresión 4 > 8 da como resultado el valor falso, la expresión num == num da como resultado cierto, la expresión 8 <= 4 da como resultado falso, etc.

C representa un resultado Falso como el valor numérico entero cero, y un resultado "CIERTO" como cualquier valor entero diferente de cero.

Los operadores lógicos se utilizan principalmente en conjunción con los relacionales para elaborar condiciones complejas en las sentencias condicionales e iterativas

2.6.4 Operadores de asignación

El operador de asignación permite asignar valores a las variables. Su símbolo es un signo igual =. Este operador asigna a la variable que está a la izquierda del operador el valor que está a la derecha. Un ejemplo de expresiones válidas con el operador de asignación son: x = 1; z = 1.35;.

El lenguaje C, a diferencia de otros lenguajes tales como Pascal, no diferencia la asignación de cualquier otro operador del lenguaje. Para C, la asignación es un operador, el llamado operador asignación (=), el cual posee la prioridad más baja de todos los operadores. Es por ello que en C podemos escribir expresiones del tipo:

```
if ((c=a*b)<0) /* if es la comprobación condicional de C, que */ /* se vera con posterioridad */
```

Esta expresión asigna a la variable c el valor de a*b y devuelve su valor para compararlo con el valor constante 0. Los paréntesis son necesarios pues el operador asignación tiene la prioridad más baja de todos los operadores.

2.6.5 Aplicación práctica de conceptos (Laboratorios).

1. Programa que lee continuamente un carácter, lo copia y después lo pega en pantalla, el programa termina cuando el carácter sea igual a S.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int opción
clrscr();
opcion = ' ';
printf("INTRODUSCA UN CARACTER O S PARA SALIR\n");
while (opcion!='S')
{
opcion=getc(stdin);
putchar(opcion);
}
printf("\nBYE");
getch();
}
```

2. Escribir en lenguaje C un programa que:

1º) Pida por teclado dos números (datos enteros) y sean almacenados en dos variables, llamadas v1 y v2.

2º) Intercambie los valores de las variables.

3º) Muestre por pantalla los valores contendidos en las variables.

```
 #include <conio.h>
#include <stdio.h>

int main()
```

```
{
   int auxiliar, v1, v2;

   printf( "\n   Introduzca el valor de v1: " );
   scanf( "%d", &v1 );
   printf( "\n   Introduzca el valor de v2: " );
   scanf( "%d", &v2 );

   printf( "\n   Intercambiando los valores..." );

   /* Para hacer el intercambio utilizamos una variable auxiliar */

   auxiliar = v1;
   v1 = v2;
   v2 = auxiliar;

   printf( "\n\n   Ahora, el valor de v1 es: %d", v1 );
   printf( "\n\n   Ahora, el valor de v2 es: %d", v2 );

   getch(); /* Pausa */

   return 0;
}
```

3. Escribir en lenguaje C un programa que:

1º) Pida por teclado un número (dato entero).

2º) Muestre por pantalla el número anterior y el número posterior (datos enteros).

```
#include <conio.h>
#include <stdio.h>
int main()
{
```

```
   int numero;
   printf( "\n   Introduzca un n%cmero entero: ", 163 );
   scanf( "%d", &numero );
    printf( "\n   El n%cmero anterior es: %d", 163, numero - 1 );
   printf( "\n\n   El n%cmero posterior es: %d", 163, numero + 1 );
   getch(); /* Pausa */
   return 0;
}
```

4. Escribir en lenguaje C un programa que:

1º) Pida por teclado un año (dato entero).

2º) Muestre por pantalla:

- **"ES BISIESTO", en el caso de que el año sea bisiesto.**
- **"NO ES BISIESTO", en el caso de que el año no sea bisiesto.**

<u>Nota:</u> Son bisiestos todos los años múltiplos de 4, excepto aquellos que son múltiplos de 100 pero no de 400. Por ejemplo, años múltiplos de 4 son:

4, 8, 20, 100, 200, 400, 1000, 2000, 2100, 2800...

De ellos, años múltiplos de 100 pero no de 400 son:

100, 200, 1000, 2100...

Así que, de los años enumerados, bisiestos son:

4, 8, 20, 400, 2000, 2800...

```
#include <conio.h>
#include <stdio.h>
int main()
{
   int anio;
   printf( "\n   Introduzca un a%co: ", 164 );
   scanf( "%d", &anio );
   if ( anio % 4 == 0 && anio % 100 != 0 || anio % 400 == 0 )
      printf( "\n   ES BISIESTO" );
```

```
   else
      printf( "\n   NO ES BISIESTO" );
   getch(); /* Pausa */
   return 0;
}
```

2.7 Sentencias de Control

El concepto de sentencia en C es igual que el de otros muchos lenguajes. Por sentencia se entiende en C cualquier instrucción simple o bien, cualquier conjunto de instrucciones simples que se encuentren encerradas entre los caracteres { y }, que marcan respectivamente el comienzo y el final de una sentencia.

2.7.1 Alternativas (lf)

Antes de empezar a explicar las sentencias de control del lenguaje C, conviene explicar los conceptos de verdadero/falso y de sentencia que posee el lenguaje C.

El lenguaje C posee un concepto muy amplio de lo que es verdadero. Para C, cualquier valor que sea distinto de cero es verdadero, siendo por tanto falso solo si el valor cero. Es por ello que una expresión del tipo if(x) será verdad siempre que el valor de la variable x sea distinto de cero, sea cual sea el tipo de la variable x.

La forma general de la sentencia if es:

```
if (condición
   sentencia;
else
   sentencia;
```

Siendo el else opcional. Si la condición es verdadera se ejecuta la sentencia asociada al if, en caso de que sea falsa la condición se ejecuta la sentencia asociada al else (si existe el else). Veamos algunos ejemplos de sentencias if:

```
int a,b;

if (a>b)
{
   b--;
   a=a+5;
}
else
{
   a++;
   b=b-5;
}
if (b-a!=7)
   b=5;
```

2.7.1.2 Compuestas

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de sentencias compuestas. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves {...}. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas.

2.7.1.3 Anidadas

Las sentencias de control if pueden ir anidadas. Un if anidado es una sentencia if que es el objeto de otro if o else. Esta anidación de if/else puede presentar la problemática de decidir que else va asociado a cada if. Considérese el siguiente ejemplo:

```
if (x)
   if (y) printf("1");

else printf("2");
```

¿A que if se refiere el else?. C soluciona este problema asociando cada else al if más cercano posible y que todavía no tiene ningún else asociado. Es por ello que en este caso el if asociado al else es el if(y). Si queremos que el else este asociado al if(x), deberíamos escribirlo de la siguiente forma:

```
if (x)
{
   if (y)
      printf("1");
}
else
   printf("2");
```

2.7.2 Selección múltiple

Esta sentencia permite realizar una ramificación múltiple, ejecutando una entre varias partes del programa según se cumpla una entre n condiciones.

La forma general es la siguiente:

if (expresion_1)

sentencia_1

else if (expresion_2)

sentencia_2

else if (expresion_3)

sentencia_3

else if (...)

...

[else

sentencia_n]

Explicación: Se evalúa expresion_1. Si el resultado es true, se ejecuta sentencia_1. Si el resultado es false, se salta sentencia_1y se evalúa expresion_2. Si el resultado es true se ejecuta sentencia_2, mientras que si es false se evalúa expresion_3 y así sucesivamente. Si ninguna de las expresiones o condiciones es true se ejecuta expresion_n que es la opción por defecto (puede ser la sentencia vacía, y en ese caso puede eliminarse junto con la palabra else). Todas las sentencias pueden ser simples o compuestas.

(switch)

La forma general de la sentencia switch es:

```
{
    case constante1:
        instrucciones;
```

```
            break;
        case constante 2:
            instrucciones;
            break;
        . . .
        default:
            instrucciones;
    }
```

En una instrucción switch, expresión debe ser una expresión con un valor entero, y constante1, constante2, ..., deben ser constantes enteras, constantes de tipo carácter o una expresión constante de valor entero. Expresión también puede ser de tipo char, ya que los caracteres individuales tienen valores enteros

Dentro de un case puede aparecer una sola instrucción o un bloque de instrucciones.

La instrucción switch evalúa la expresión entre paréntesis y compara su valor con las constantes de cada case. Se ejecutarán las instrucciones de aquel case cuya constante coincida con el valor de la expresión, y continúa hasta el final del bloque o hasta una instrucción que transfiera el control fuera del bloque del switch (una instrucción break, o return). Si no existe una constante igual al valor de la expresión, entonces se ejecutan las sentencias que están a continuación de default si existe (no es obligatorio que exista, y no tiene porqué ponerse siempre al final).

Ejemplo de uso de la instrucción switch.

Programa que lee dos números y una operación y realiza la operación entre esos números.

```
#include <iostream>
using namespace std;
int main(void)
{
   int A,B, Resultado;
   char operador;
   cout << "Introduzca un numero:";
   cin >> A;
   cout << "Introduzca otro numero:";
```

```
cin >> B;
cout <<"Introduzca un operador (+,-,*,/):";
cin >> operador;
Resultado = 0;
switch (operador)
{
    case '-' : Resultado = A - B;
               break;
    case '+' : Resultado = A + B;
               break;
    case '*' : Resultado = A * B;
               break;
    case '/' : Resultado = A / B; //suponemos B!=0
               break;
    default : cout << "Operador no valido"<< endl;
}
cout << "El resultado es: ";
cout << Resultado << endl;
system("pause");
```

Otro ejemplo de uso de switch.

Programa que determina si un carácter leído es o no una vocal. En ese caso como la sentencia a ejecutar por todas las etiquetas case es la misma, esta sentencia se pone una única vez al final:

```
#include <iostream>
using namespace std;
int main(void)
{
   char car;
   cout << "Introduzca un caracter: ";
   cin >> car;
   switch (car)
```

```
    {
        case 'a':
        case 'A':
        case 'e':
        case 'E':
        case 'i':
        case 'I':
        case 'o':
        case 'O':
        case 'u':
        case 'U': cout << car << " es una vocal" << endl;
                  break;
        default : cout << car << " no es una vocal" << endl;
    }
    system("pause");
}
```

2.7.3 Sentencia Do

Es un bucle que, por lo menos, se ejecuta una vez. Do significa literalmente "hacer", y while significa "mientras"

Su forma es esta:

```
do {
        /* CODIGO */
} while (/* Condición de ejecución del bucle */)
```

2.7.4 While

El bucle WHILE consiste en un bucle en el que el código se repite mientras se cumpla alguna condición booleana (es decir, una expresión que dé como resultado verdadero o falso). Hay variaciones, como el REPEAT...UNTIL, que se diferencia en el momento de comprobar si se hace verdadera o no la condición.

2.7.5 For

La sintaxis del bucle for es:

```
for(inicialización,condición,incremento) sentencia;
```

En primer lugar, conviene destacar el hecho de la gran flexibilidad del bucle for de C. En C, el bucle for puede no contener inicialización, condición o incremento, o incluso pueden no existir dos e incluso las tres expresiones del bucle. El bucle for se ejecuta siempre que la condición sea verdadera, es por ello que puede llegar a no ejecutarse.

Veamos algunos ejemplos de bucles for:

```
int i,suma=0;
for(i=1;i<=100;i++)
   suma=suma+i;

int i,j;


for(i=0,j=100;j>i;i++,j--)
{
   printf("%d\n",j-i);
   printf("%d\n",i-j);
}

float a=3e10;
for(;a>2;a=sqrt(a)) /* sqrt() calcula la raíz cuadrada */
   printf("%f",a);

char d;
for(;getc(stdin)!='\x1B';); /* Bucle que espera hasta que se */
                            /* pulsa la tecla Esc */

char d;
for(;;)
{
   d=getc(stdin);
   printf("%c",d);
   if (d=='\x1B')
      break;
}
```

2.7.6 Break

La sentencia break provoca la salida del bucle en el cual se encuentra y la ejecución de la sentencia que se encuentra a continuación del bucle.

Las sentencias de control break permite modificar y controlar la ejecución de los bucles anteriormente descritos. Ejemplo:

```
int x;
for(x=0;x<10;x++)
{
   for(;;)
      if (getc(stdin)=='\x1B')
         break;
   printf("Salí del bucle infinito, el valor de x es: %d\n",x);
}
int x;
for(x=1;x<=100;x++) /* Esta rutina imprime en pantalla los */
{                   /*números pares */

  if (x%2)
     continue;
  printf("%d\n",x);
}
```

2.7.7 Aplicación práctica de conceptos (Laboratorios).

1. Programa que imprime el valor actual de una variable en cada ciclo. (FOR, WHILE, DO WHILE)

//Con la sentencia FOR:

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

void main()
{
int i;

for(i=0; i<=20; i++)
printf("Soy la variable i, mi valor en esta iteracion es: %d\n",i);

system("pause");

}
```

//Con la sentencia WHILE:

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

```
void main()
{
int i=0;

while(i<=20)
{
printf("Soy la variable i, mi valor en esta iteracion es: %d\n",i);
i++;
}

system("pause");
}
```

//Con la sentencia DO-WHILE:

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

void main()
{
int i=0;

do
{
printf("Soy la variable i, mi valor en esta iteracion es: %d\n",i);
i++;
}
while(i<=20);

system("pause");
}
```

2. Programa que suma los 10 primeros números enteros. (FOR, WHILE, DO WHILE)

//Con la sentencia FOR:

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

void main()
{
int num;
int suma=0;

for(num=0; num<=10; num++)
{
```

```
suma = suma + num;
}

printf("Suma = %d\n",suma);

system("pause");
}
```

//Con la sentencia WHILE:

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

void main()
{
int num=0;
int suma=0;

while(num<=10)
{
suma = suma + num;
num++;
}

printf("Suma = %d\n",suma);

system("pause");
}
```

//Con la sentencia DO-WHILE:

```
void main()
{
int num=0;
int suma=0;

do
{
suma = suma + num;
num++;
} while(num<=10);

printf("Suma = %d\n",suma);
```

```
system(“pause”);
}
```

3. Programa que muestra de forma descendente los números del 0 al 10. (FOR, WHILE, DO WHILE)

//Con la sentencia FOR:

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

void main()
{
int numero;

for(numero=10; numero >= 0; numero–)
printf(“%d\n”,numero);

system(“pause”);
}
```

//Con la sentencia WHILE:

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

void main()
{
int numero=10;

while(numero >= 0)
{
printf(“%d\n”,numero);
numero–;
}

system(“pause”);
}
```

//Con la sentencia DO-WHILE:

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

```
void main()
{
int numero=10;

do
{
printf(“%d\n”,numero);
numero–;
}
while(numero >= 0);

system(“pause”);
}
```

4. Se quiere escribir un programa que:

1º) Pida por teclado un número (dato entero).

2º) Pregunte al usuario si desea introducir otro o no.

3º) Repita los pasos 1º y 2º, mientras que, el usuario no responda 'n' de (no).

4º) Muestre por pantalla la suma de los números introducidos por el usuario.

En pantalla:

Introduzca un numero entero: 7

¿Desea introducir otro (s/n)?: s

Introduzca un numero entero: 16

¿Desea introducir otro (s/n)?: s

Introduzca un numero entero: -3

¿Desea introducir otro (s/n)?: n

La suma de los números introducidos es: 20

```
#include <stdio.h>
int main()
{
  char seguir;
```

```
  int acumulador, numero;


  /* En acumulador se va a guardar la suma de
     los números introducidos por el usuario. */
  acumulador = 0;
  do
  {
     printf( "\n   Introduzca un numero entero: " );
     scanf( "%d", &numero);
     acumulador += numero;
     printf( "\n   Desea introducir otro numero (s/n)?: " );
     fflush( stdin );
     scanf( "%c", &seguir);
  } while ( seguir != 'n' );
  /* Mientras que el usuario desee introducir
     más números, el bucle iterará. */
  printf( "\n   La suma de los numeros introducidos es: %d",
          acumulador );
  return 0;
}
```

5. Que pida un número del 1 al 5 y diga si es primo o no.

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

int main(void)
{
          int i;
          printf("Introduzca número del 1 al 5:");
   scanf("%d",&i);

   if (i!=4) {
     printf("Es primo.");
   }
   else
```

```
   {
     printf("No es primo.");
   }

   system("PAUSE");
   return 0;
}
```

6. Que pida un número y diga si es par o impar.

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

int main(void)
{
        int i;
        printf("Introduzca número:");
   scanf("%d",&i);

   if (i%2==0) {
     printf("Es par.");
   }
   else
   {
     printf("Es impar.");
   }

   system("PAUSE");
   return 0;
}
```

7. Que pida un número del 1 al 7 y diga el día de la semana correspondiente.

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

int main(void)
{
        int i;
        printf("Introduzca número del 1 al 7:");
   scanf("%d",&i);

   switch(i){
          case 1:
              printf ("Lunes\n");
              break;
          case 2:
```

```
            printf ("Martes\n");
            break;
        case 3:
            printf ("Miércoles\n");
            break;
        case 4:
            printf ("Jueves\n");
            break;
        case 5:
            printf ("Viernes\n");
            break;
        case 6:
            printf ("Sábado\n");
            break;
        case 7:
            printf ("Domingo\n");
            break;
        default:
            printf ("Opción no válida\n");
            break;
   }

   system("PAUSE");
   return 0;
}
```

8. Que pida un número del 1 al 12 y diga el nombre del mes correspondiente.

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

int main(void)
{
        int i;
        printf("Introduzca número del 1 al 12:");
   scanf("%d",&i);

   switch(i){
        case 1:
            printf ("Enero\n");
            break;
        case 2:
            printf ("Febrero\n");
            break;
        case 3:
            printf ("Marzo\n");
            break;
```

```
        case 4:
            printf ("Abril\n");
            break;
        case 5:
            printf ("Mayo\n");
            break;
        case 6:
            printf ("Junio\n");
            break;
        case 7:
            printf ("Julio\n");
            break;
        case 8:
            printf ("Agosto\n");
            break;
        case 9:
            printf ("Septiembre\n");
            break;
        case 10:
            printf ("Octubre\n");
            break;
        case 11:
            printf ("Noviembre\n");
            break;
        case 12:
            printf ("Diciembre\n");
            break;
        default:
            printf ("Opción no válida\n");
            break;
    }

    system("PAUSE");
    return 0;
}
```

9. Programa que lee 5 números e imprime la suma de ellos.

```
#include <stdio.h>

#include <conio.h>

int main()

{

int i, num, r=0;
```

```
for (i=1; i<=5; i++)
{
printf("Anote un numero\n");
scanf("%i",&num);
r=r+num;
}
printf("La suma es: %i",r);
getch();
return 0;
}
```

10. Programa que suma números indefinidamente hasta que el usuario lo desee.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int a, b, opcion=1;
while(opcion==1)
{
printf("Anote un numero\n");
scanf("%i",&a);
b=a*2;
printf("El resultado es: %i\n",b);
printf("Desea continuar 1. SI 2. NO\n");
scanf("%i",&opcion);
}
printf("Presione una tecla para salir\n");
getch();
}
```

11. Programa que lee dos números y dice cuál es el mayor y cuál es el menor.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int num1, num2;
printf("Anote un numero\n");
scanf("%i",&num1);
printf("Anote otro numero\n");
scanf("%i",&num2);
if (num1>num2)
printf("El numero %i es mayor que %i\n",num1, num2);
else
printf("El numero %i es menor que %i\n",num1, num2);
getch();
return 0;
}
```

12. Programa que muestra en pantalla una serie de opciones a elegir e imprime el costo del producto elegido.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int p;
clrscr();
printf("SELECCIONE UN PRODUCTO\n\n");
printf("1. REFRESCO\n");
printf("2. PAPAS FRITAS\n");
printf("3. HAMBURGUESA\n");
printf("4. JUGO\n");
```

```
scanf("%i",&p);
if (p = = 1)
printf("EL COSTO ES: $5.00");
else if(p = = 2)
printf("EL COSTO ES: $10.00");
else if(p = = 3)
printf("EL COSTO ES: $20.00");
else
printf("EL COSTO ES: $8.00");
getch();
return 0;
}
```

13. Programa que pide un número del 1 al 7 y te dice que día de la semana es.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int opcion;
clrscr();
printf("INTRODUSCA UN NUMERO ENTER0 DEL 1 AL 7\n");
scanf("%i",&opcion);
switch(opcion)
{
case 1:
printf("EL DIA %i ES DOMINGO",opcion);
break;
case 2:
printf("EL DIA %i ES LUNES",opcion);
break;
case 3:
```

```
printf("EL DIA %i ES MARTES", opcion);
break;
case 4:
printf("EL DIA %i ES MIERCOLES",opcion);
break;
case 5:
printf("EL DIA %i ES JUEVES", opcion);
break;
case 6:
printf("EL DIA %i ES VIERNES", opcion);
break;
case 7:
printf("EL DIA %i ES SABADO", opcion);
break;
default:
printf("EL NUMERO NO PERTENECE AL RANGO");
break;
}
getch();
return 0;
}
```

14. Escribir en lenguaje C un programa que muestre por pantalla todos los números múltiplos de 4 que hay entre el 64 y el 44, ambos inclusive.

Nota: utilizar un bucle for.

```
#include <conio.h>
#include <stdio.h>
int main()
{
   int numero;
```

```
   printf( "\n   " );
   for ( numero = 64 ; numero >= 44 ; numero += -4 )
   {
      printf( "%d ", numero );
   }
   getch(); /* Pausa */
   return 0;
}
```

15. **Escribir en lenguaje C un programa que muestre por pantalla la sucesión de números: 1 -1 2 -2 3 -3 4 -4 5 -5**

<u>Nota:</u> Utilizar un bucle for.

```
#include <conio.h>
#include <stdio.h>
int main()
{
   int numero;
   printf( "\n   " );
   for ( numero = 1 ; numero <= 5 ; numero++ )
   {
      printf( "%d %d ", numero, -numero );
   }
   getch(); /* Pausa */
   return 0;
}
```

III. INTRODUCCIÓN A FUNCIONES

Objetivos

- Definir los conceptos básicos relacionados con las funciones en C
- Identificar la forma en la que se declara o define las funciones en el lenguaje de programación C
- Presentar y desarrollar problemas prácticos utilizando funciones.

¿De qué trata esta sesión de aprendizaje?

En esta sesión de aprendizaje se hace referencia a las funciones como un método efectivo para descomponer problemas de programación se conoce su forma de definición y procesamiento básico, y métodos de acceso, además se presentan problemas y sus respectivas soluciones. Reconociendo mediante ejemplos su funcionalidad de implementación a problemas presentados.

III. INTRODUCCIÓN A FUNCIONES

3.1 Introducción a funciones

Una de las formas más adecuadas de resolver un problema de programación consiste en descomponerlo en subproblemas. A cada uno de ellos se le asocia una función que lo resuelve, de tal modo que la solución del problema se obtiene por medio de llamadas a funciones. A su vez, cada función puede descomponerse en subfunciones que realicen tareas más elementales, intentando conseguir que cada función realice una y sólo una tarea.

3.1.1 Definición de una función

En Lenguaje C una función se define de la siguiente forma:

```
tipo NombreFunción (parámetros formales)
{
    ...
    cuerpo de la función
    ...
}
```

El tipo es el tipo de dato que devuelve la función por medio de la sentencia return. Cuando no se especifica un tipo, se asume que el tipo devuelto es int. El NombreFunción es un identificador válido en C. Es el nombre mediante el cual se invocará a la función desde main() o desde otras funciones. Los parámetros formales son las variables locales mediante las cuales la función recibe datos cuando se le invoca. Deben ir encerrados entre paréntesis. Incluso si no hay parámetros formales los paréntesis deben aparecer. La siguiente función devuelve el mayor de dos números enteros:

```
int mayor (int x, int y)
{
    int max;

    if (x > y) max = x;
    else max = y;

    return max;
}
```

3.1.2 Acceso a una función

Al manejar funciones en C debemos tener en cuenta que a una función sólo se puede acceder por medio de una llamada. Nunca se puede saltar de una función a otra mediante una sentencia goto. Tampoco está permitido declarar una función dentro de otra.

3.1.3 Paso de parámetros

El único método para que una función reciba valores es por medio de los parámetros formales. Los parámetros formales son variables locales que se crean al comenzar la función y se destruyen al salir de ella. Al hacer una llamada a la función los parámetros formales deben coincidir en tipo y en número con las variables utilizadas en la llamada. Si no coinciden, puede no detectarse el error. Esto se evita usando los llamados prototipos de funciones.

Los parámetros de una función pueden ser:

- **valores** (llamadas por valor)
- **direcciones** (llamadas por referencia)

Llamadas por valor

En las llamadas por valor se hace una copia del valor del argumento en el parámetro formal. La función opera internamente con estos últimos. Como las variables locales a una función (y los parámetros formales lo son) se crean al entrar a la función y se destruyen al salir de ella, cualquier cambio realizado por la función en los parámetros formales no tiene ningún efecto sobre los argumentos. Por ejemplo:

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambio (&a, &b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambio (int *x, int *y)
{
    int aux;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

La salida de este programa es:

Valor de a: 3 - Valor de b: 4

es decir, NO intercambia los valores de las variables a y b. Cuando se hace la llamada

intercambia (a, b);

se copia el valor de a en x, y el de b en y. La función trabaja con x e y, que se destruyen al finalizar, sin que se produzca ningún proceso de copia a la inversa, es decir, de x e y hacia a y b.

Llamadas por referencia

En este tipo de llamadas los parámetros contienen direcciones de variables. Dentro de la función la dirección se utiliza para acceder al argumento real. En las llamadas por referencia cualquier cambio en la función tiene efecto sobre la variable cuya dirección se pasó en el parámetro. Esto es así porque se trabaja con la propia dirección de memoria, que es única, y no hay un proceso de creación/destrucción de esa dirección.

Aunque en C todas las llamadas a funciones se hacen por valor, pueden simularse llamadas por referencia utilizando los operadores & (dirección) y * (en la dirección). Mediante & podemos pasar direcciones de variables en lugar de valores, y trabajar internamente en la función con los contenidos, mediante el operador *.

El siguiente programa muestra cómo se puede conseguir el intercambio de contenido en dos variables, mediante una función.

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambio (&a, &b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambio (int *x, int *y)
{
    int aux;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

Este programa SÍ intercambia los valores de a y b, y muestra el mensaje

Valor de a: 4 - Valor de b: 3

Además, a diferencia del de la página 83, trabaja con direcciones. Como veremos en el siguiente capítulo, en la declaración

void intercambio (int *x, int *y);

los parámetros x e y son unas variables especiales, denominadas punteros, que almacenan direcciones de memoria. En este caso, almacenan direcciones de enteros.

3.2 Aplicación práctica de conceptos (Laboratorios).

1. Crear una función que calcule cual es el menor de dos números enteros.
El resultado será otro número entero. */

```
 #include <stdio.h>
int minus(int num1, int num2)
 {
   int min;

   min = num1 < num2? num1: num2;
   return min;
 }

 int main()
 {
    int numero1, numero2, menor;
    while (numero1, numero2)
    {
```

```
    printf(" Escriba un numero\n");
    scanf("%d",&numero1);
    printf(" Escriba otro numero\n");
    scanf("%d",&numero2);
    menor=minus(numero1, numero2);
    printf("   El menor es %d\n\n", menor);

    }
    return 0;
}
```

2. Crear una función que calcule el cubo de un número real (float). El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5. */

```
#include <stdio.h>
float cubica(float numero)
{
  float cubica;
  cubica = numero * numero * numero;
  return cubica;
}

int main()
{
  float numero;
  float cubo;
  int i=0;

  while(numero >= -1000000)
  {
    if(i==0)
```

```
      puts("\nEscriba un numero real");
    else
      puts("\nEscriba otro numero real");
    scanf("%f", &numero);
    printf("%f elevado al cubo es %f\n",
      numero,cubo=cubica(numero));
    i ++;
  }
 return 0;
}
```

3. Crear una función llamada "signo", que reciba un número real, y devuelva un número entero con el valor: -1 si el número es negativo, 1 si es positivo o 0 si es cero. */

```
#include <stdio.h>
 int signo (float num)
{
  int sig;

  if (num > 0)
    sig = 1;
  if (num == 0)
    sig = 0;
  if (num < 0)
    sig = -1;
      return sig;
  }
int main()
{
  float numero;
  while (numero > -1000000)
```

```
  {
    printf("\nEscriba un numero real: ");
    scanf("%f", &numero);
    printf("%d", signo(numero));
  }
  return 0;
  }
```

4. Crear una función que devuelva la primera letra de una cadena de texto. Probar esta función para

calcular la primera letra de la frase "Hola" */

```
#include <stdio.h>
#include <string.h>
char primeraLetra (char* cadena)
{
  return cadena[0];
}
int main()
{
  char palabra[10];
    printf("Escriba una palabra\n");
  scanf("%s", palabra);
  printf("La primera letra es %c.", primeraLetra(palabra));
    return 0;
}
```

5. Crear una función "escribirTablaMultiplicar", que reciba como parámetro un número entero, y escriba

la tabla de multiplicar de ese número (por ejemplo, para el 3 deberá llegar desde 3x0=0 hasta 3x10=30). */

```
  #include <stdio.h>
  void escribirTablaMultiplicar (int numero)
{
  int tabla;
        for (tabla=1; tabla<=10; tabla++)
     printf("%d x %d = %d\n", numero, tabla, numero * tabla);
        return;
}
int main()
{
  int num, tabla;
      printf("Escriba un numero: ");
  scanf("%d", &num);
  escribirTablaMultiplicar(num);
      return 0;
}
```

6. Programa que lee por teclado la fecha actual y la fecha de nacimiento de una persona y calcula su edad.

El programa utiliza tres funciones:

fecha_valida: comprueba si la fecha leída es correcta.

bisiesto: comprueba si un año es bisiesto. La llama la función fecha_valida

calcular_edad: recibe las dos fechas y devuelve la edad

```
#include <iostream>
using namespace std;
int bisiesto(int);
int fecha_valida(int , int, int);
int calcular_edad(int, int, int, int, int, int);
int main()
```

```
{
  int diaa, mesa, anioa, dian, mesn, anion, edad;
  do
  {
     cout << "Introduce fecha actual: " << endl;
     cout << "dia : "; cin >> diaa;
     cout << "mes : "; cin >> mesa;
     cout << "a" << (char)164 << "o: "; cin >> anioa;
  }while(!(fecha_valida(diaa, mesa, anioa)));
  do
  {
     cout << endl << "Introduce fecha de nacimiento: " << endl;
     cout << "dia : "; cin >> dian;
     cout << "mes : "; cin >> mesn;
     cout << "a" << (char)164 << "o: ";  ; cin >> anion;
  }while(!(fecha_valida(dian, mesn, anion)));
  edad = calcular_edad(diaa, mesa, anioa, dian, mesn, anion);
  cout << endl << "Edad : " << edad << endl << endl;
  system("pause");
}

int calcular_edad(int da, int ma, int aa, int dn, int mn, int an)
{
   int edad = aa - an;
   if(ma < mn)
     edad--;
   else if(ma == mn and da < dn)
         edad--;
   return edad;

}
```

```
int bisiesto(int a)  // definición de la función bisiesto
{
   if(a%4==0 and a%100 !=0 or a%400==0)
      return 1;
   else
      return 0;
}

int fecha_valida(int d, int m, int a) //definición de fecha_valida
{
   if(d < 1 or d > 31 or m < 1 or m > 12 or a < 1)
   {
      return 0;
   }
   switch(m)
   {
      case 4:
      case 6:
      case 9:
      case 11: if(d > 30)
               {
                 return 0;
               }
               break;
      case 2: if(bisiesto(a))
              {
                 if(d > 29)
                 {
                   return 0;
                 }
```

```
            }
            else if(d > 28)
                {
                  return 0;
                }
            break;
    }
    return 1;
}
```

IV. ARREGLOS

Objetivos

- Definir los conceptos básicos relacionados con los arreglos en C.
- Identificar la forma en la que se declara o define un arreglo.
- Realizar problemas prácticos con arreglos.

¿De qué trata esta sesión de aprendizaje?

En esta sesión de aprendizaje se hace referencia a la estructura llamada arreglo en el lenguaje de programación C, se conoce su forma de definición y procesamiento básico, además se presentan problemas y sus respectivas soluciones. Reconociendo mediante ejemplos su funcionalidad de implementación a problemas presentados.

IV. ARREGLOS

4.1 Introducción a arreglos

Los arreglos son estructuras de datos consistentes en un conjunto de datos del mismo tipo. Los arreglos tienen un tamaño que es la cantidad de objetos del mismo tipo que pueden almacenar. Los arreglos son entidades estáticas debido a que se declaran de un cierto tamaño y conservan este todo a lo largo de la ejecución del programa en el cual fue declarado.

4.2 Definición de un arreglo

Un arreglo es un conjunto de datos o una estructura de datos homogéneos que se encuentran ubicados en forma consecutiva en la memoria RAM (sirve para almacenar datos en forma temporal).

Un arreglo puede definirse como un grupo o una colección finita, homogénea y ordenada de elementos. Los arreglos pueden ser de los siguientes tipos:

- De una dimensión.
- De dos dimensiones.
- De tres o más dimensiones

Ejemplo de declaración:

int arreglo1[30]

declara que arreglo1 es un arreglo que puede contener 30 enteros.

#define TAMAÑO 100
int arreglo2[TAMAÑO]

declara que arreglo2 es un arreglo que puede contener TAMANIO enteros.

La ventaja de esta última declaración es que todos los programas que manipulen arreglo2 utilizar como tamaño TAMAÑO y si quiero cambiar el tamaño del arreglo alcanza con cambiar la definición de TAMAÑO.
Observar que en la declaración se especifica: tipo de los elementos, número de elementos y nombre del arreglo.
Un arreglo consta de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Para acceder a un elemento en particular se utiliza un índice.
En C, todos los arreglos usan cero como índice para el primer elemento y si el tamaño es n, el índice del último elemento es n-1.
Ejemplo de programa que utiliza un array:

```
main ()
{
int arreglo2[TAMAÑO];
```

```
/* cargo array con valor igual al índice más 10*/
for (int i=0;i
<
TAMAÑO;i++)
arreglo2[i]=i+10;
/* imprimo contenido del arreglo */
for (int i=0;i
<
TAMAÑO;i++)
printf("Elemento %d del array es %d
\
n",i+1,arreglo2[i]);
}
```

Del programa anterior podemos extraer que:

1. para cargar un elemento de un array coloco el nombre de array seguido de un índice entre paréntesis rectos y asocio el valor que desee.
2. para acceder al valor de un elemento del array coloco el nombre de array seguido de un índice entre paréntesis rectos.
3. los índices con los que accedo al array varían entre 0 y la cantidad de elementos menos 1.

Los nombres de arrays siguen la misma convención que los nombres de va-riable.

Ejemplos de declaraciones:

```
int notas[8] /* almacena ocho notas */
char nombre[21] /* almacena nombres de largo menor o igual a 20 */
int multiplos[n] /* donde n tiene un valor, declara un arreglo de tamaño n*/
```

Los índices pueden ser cualquier expresión entera. Si un programa utiliza una expresión como sub índice esta se evalúa para determinar el índice. Por ejemplo, si a=5 y b=10, el enunciado arreglo2[a+b] +=2, suma 2 al elemento del arreglo número 15. Puedo utilizar un elemento del arreglo en las mismas expresiones que variables del tipo correspondiente. En el caso de arreglo2,

puedo utilizar cualquiera de sus elementos en expresiones donde pueda utilizar una variable entera, por ejemplo printf(" %d", arreglo2[0]+arreglo2[15]+arreglo2[30]);

Los arreglos pueden ser declarados para que contengan distintos tipos de datos. Por ejemplo, un arreglo del tipo char puede ser utilizado para almacenar una cadena de caracteres.

4.3 Procesamiento de un arreglo

Los procesamientos con arreglos pueden darse de las siguientes formas:

- Lectura: este proceso consiste en leer un dato de un arreglo y asignar un valor a cada uno de sus componentes
- Escritura: Consiste en asignarle un valor a cada elemento del arreglo.
- Asignación: No es posible asignar directamente un valor a todo el arreglo
- Actualización: Dentro de esta operación se encuentran las operaciones de eliminar, insertar y modificar datos. Para realizar este tipo de operaciones se debe tomar en cuenta si el arreglo está o no ordenado.
- Ordenación.
- Búsqueda.
- Insertar.
- Borrar.
- Modificar.

Ordenaciones en Arreglos

La importancia de mantener nuestros arreglos ordenados radica en que es mucho más rápido tener acceso a un dato en un arreglo ordenado que en uno desordenado.

Existen muchos algoritmos para la ordenación de elementos en arreglos, algunos de ellos son:

Selección directa

Este método consiste en seleccionar el elemento más pequeño de nuestra lista para colocarlo al inicio y así excluirlo de la lista. Para ahorrar espacio, siempre que vayamos a colocar un elemento en su posición correcta lo intercambiaremos por aquel que la esté ocupando en ese momento.

Ordenación por burbuja

Es el método de ordenación más utilizado por su fácil comprensión y programación, pero es importante señalar que es el más ineficiente de todos los métodos. Este método consiste en llevar los elementos menores a la izquierda del arreglo o los mayores a la derecha de este. La idea básica

del algoritmo es comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados.

Ordenación por mezcla

Este algoritmo consiste en partir el arreglo por la mitad, ordenar la mitad izquierda, ordenar la mitad derecha y mezclar las dos mitades ordenadas en un array ordenado. Este último paso consiste en ir comparando pares sucesivos de elementos (uno de cada mitad) y poniendo el valor más pequeño en el siguiente hueco.

Algoritmos de búsqueda que existen

- **Búsquedas en Arreglos:** Una búsqueda es el proceso mediante el cual podemos localizar un elemento con un valor específico dentro de un conjunto de datos. Terminamos con éxito la búsqueda cuando el elemento es encontrado.
- **Búsqueda secuencial:** A este método también se le conoce como búsqueda lineal y consiste en empezar al inicio del conjunto de elementos, e ir a través de ellos hasta encontrar el elemento indicado o hasta llegar al final de arreglo. Este es el método de búsqueda más lento, pero si nuestro arreglo se encuentra completamente desordenado es el único que nos podrá ayudar a encontrar el dato que buscamos.
- **Búsqueda binaria:** Las condiciones que debe cumplir el arreglo para poder usar búsqueda binaria son que el arreglo este ordenado y que se conozca el numero de elementos. Este método consiste en lo siguiente: comparar el elemento buscado con el elemento situado en la mitad del arreglo, si tenemos suerte y los dos valores coinciden, en ese momento la búsqueda termina. Pero como existe un alto porcentaje de que esto no ocurra, repetiremos los pasos anteriores en la mitad inferior del arreglo si el elemento que buscamos resulto menor que el de la mitad del arreglo, o en la mitad superior si el elemento buscado fue mayor. La búsqueda termina cuando encontramos el elemento o cuando el tamaño del arreglo a examinar sea cero.
- **Búsqueda por hash:** La idea principal de este método consiste en aplicar una función que traduce el valor del elemento buscado en un rango de direcciones relativas. Una desventaja importante de este método es que puede ocasionar colisiones.

4.4 Aplicación práctica de conceptos (Laboratorios)

1. Escribir un programa (es decir una función *main*) que lea el contenido de un archivo de texto y lo vuelque por pantalla, indicando posteriormente la longitud de la línea más larga. El programa debe recibir el nombre del archivo como argumento, no teniendo que introducirse éste por pantalla. Para ello la función *main* debe tener el prototipo estándar int main(int argc, char *argv[]).

```
#include <stdio.h>
#include <string.h>
#define ERR -1
#define OK 1
#define MAX_DIM 80      /* Maxima longitud de la linea */
#define mayor(a,b)((a) > (b) ? (a):(b))

char *fgets(char *linea, int max, FILE * fp);
int mayor_linea(void);
int leer_archivo(char *archivo[]);
void ayuda(void);

FILE * fp;
int max = MAX_DIM;

int main(int argc, char *argv[])
{
 /* Debe introducir <nombre del ejecutable> <nombre del archivo> */
 if (argc != 2)
  {
   ayuda();
   return ERR;
  }

 if (leer_archivo(&argv[1]) == ERR)
  return ERR;

 printf("\nLa linea mas larga tiene %d caracteres.\n", mayor_linea());

 fclose(fp);
 return OK;
}

/*****************************************************************************
*   Funcion: int leer_archivo(char *archivo[])
*
*   IN: Nombre del archivo
*   OUT: OK si se ha leido correctamente, ERR si ha habido algun fallo
```

```
*   MAKE: Lee un archivo introducido por el usuario.
*
*****************************************************************************/
int leer_archivo(char *archivo[])
{

 if ((fp = fopen(archivo[0], "r")) == NULL)
  {
   printf("Error al intentar abrir el archivo\n");
   return ERR;
  }

 return OK;
}

/*****************************************************************************
*   Funcion: int mayor_linea(void)
*
*   IN:
*   OUT:  Tamagno de la linea de mayor longitud del archivo.
*   MAKE: Calcula el tamagno de la linea mas larga.
*
*****************************************************************************/
int mayor_linea(void)
{

 char cad[MAX_DIM];
 int mayor_cad = 0;      /* Inicializamos el valor de la cadena */
 while (!feof(fp))
  {
   fgets(cad, MAX_DIM, fp);
   mayor_cad = mayor(mayor_cad, (int) strlen(cad));
  }

 return mayor_cad;
}

/*****************************************************************************
*   Funcion: void ayuda(void)
*
*   IN:
*   OUT: Mensaje de ayuda para el usuario.
*   MAKE: Muestra ayuda al usuario para que introduzca los datos correctamente.
*
*****************************************************************************/
void ayuda(void)
{
```

```
  printf("\nDebe introducir: <nombre del ejecutable> <nombre del archivo>\n");
}
```

2. Realice un programa en C: Que rellene un array con los 100 primeros números enteros y los muestre en pantalla en orden ascendente.

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

int main(void)
{
        int x,tabla[100];

        for (x=1;x<=100;x++)
        {
      tabla[x]=x;
   }

        for (x=1;x<=100;x++)
        {
      printf("%d\n",tabla[x]);
   }

   system("PAUSE");
   return 0;
}
```

3. Realice un programa que rellene un array con los números primos comprendidos entre 1 y 100 y los muestre en pantalla en orden ascendente.

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

int main(void)
{
        int x,cont,z,i,tabla[100];

        i=0;
        for (x=1;x<=100;x++)
        {
      cont=0;
      for (z=1;z<=x;z++)
      {
         if (x%z==0)
         {
           cont++;
```

```
        }
      }

      if (cont==2 || z==1 || z==0)
      {
       tabla[i]=x;
       i++;
      }

   }

        for (x=0;x<i;x++)
        {
      printf("%d\n",tabla[x]);
   }

   system("PAUSE");
   return 0;
}
```

4. Realice un programa que lea 10 números por teclado, los almacene en un array y muestre la suma, resta, multiplicación y división de todos.

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

int main(void)
{
        int x,tabla[10];
        int sum,res,mul,div;

        for (x=0;x<10;x++)
        {
      printf("Introduzca número\n");
      scanf("%d",&tabla[x]);
   }

        sum=tabla[0];
   res=tabla[0];
   mul=tabla[0];
   div=tabla[0];

        for (x=1;x<10;x++)
        {
      sum=sum+tabla[x];
      res=res-tabla[x];
```

```
      mul=mul*tabla[x];
      div=div/tabla[x];
   }

         printf("Suma: %d\n",sum);
         printf("Resta: %d\n",res);
         printf("Multiplicación: %d\n",mul);
         printf("División: %d\n",div);

   system("PAUSE");
   return 0;
}
```

5. Realice un programa que lea 5 números por teclado, los copie a otro array multiplicados por 2 y muestre el segundo array.

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

int main()
{
 int aux, numeros1[5],numeros2[5];
 int i,j;

 for (i=0;i<5;i++){
                printf("Escriba un número");
                scanf("%d",&numeros1[i]);
    }

 for(i=0;i<5;i++)
 {
  numeros2[i]=numeros1[i]*2;
 }

 for (i=0;i<5;i++){
    printf("%d\n",numeros2[i]);
 }

 system("PAUSE");
 return 0;
}
```

6. Realice un Programa C++ que lea por teclado la nota de los alumnos de una clase y calcula la nota media del grupo. También muestra los alumnos con notas superiores a la media.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
   float notas[20];   //array de NUMALUM elementos tipo float
   int i = 0;
   float suma = 0, media;
   // Entrada de datos. Se asigna a cada elemento del array
   // la nota introducida por teclado
   for (i=0; i<20; i++)
   {
      cout << "Alumno " << i+1 << " Nota final: ";
      cin >> notas[i];
   }
   // Sumar todas las notas
   for (i=0; i<20; i++)
       suma = suma + notas[i];
   // Calcular la media
   media = suma / 20;
```

```
   // Mostrar la media
   cout << fixed << setprecision(2);
   cout << endl<< endl << "Nota media del curso: " << media << endl;
      // Mostrar los valores superiores a la media
   cout << "Listado de notas superiores a la media" << endl;
   cout << "-------------------------------------" << endl;
   for (i=0; i<20; i++)
      if (notas[i] > media)
      {
         cout << "Alumno numero " << setw(3) << i+1;
         cout << " Nota final: " << notas[i] << endl;
      }
   cout << endl;
   system("pause");
}
```

7. Realice un programa que gestione las notas de una clase de 20 alumnos de los cuales sabemos el nombre y la nota. El programa debe ser capaz de:

a. **Buscar un alumno.**
b. **Modificar su nota.**
c. **Realizar la media de todas las notas.**
d. **Realizar la media de las notas menores de 5.**
e. **Mostrar el alumno que mejores notas ha sacado.**
f. **Mostrar el alumno que peores notas ha sacado.**

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

```
#include <stdlib.h>

struct alumno {
   char nombre[50];
   float nota;
};

int main(int argc, char *argv[])
{
   struct alumno alum,alumnos[5];

   int x,opcion=1;
   float sum=0,cont=0,mejor,peor;


   for (x=0;x<5;x++)
   {
      printf("Introduzca nombre alumno:");
      gets(alumnos[x].nombre);
      gets(alumnos[x].nombre);
      printf("Introduzca nota:");
      scanf("%f",&alumnos[x].nota);
   }

   while ((opcion==1 || opcion==2 ||
         opcion==3 || opcion==4 ||
         opcion==5 || opcion==6) && (opcion!=7))
   {

      printf("1- Buscar un alumno\n");
      printf("2- Modificar nota\n");
      printf("3- Media de todas las notas\n");
      printf("4- Media de todas las notas inferiores a 5\n");
      printf("5- Alumno con mejores notas\n");
      printf("6- Alumno con peores notas\n");
      printf("7- Salir\n");
      printf("Introduzca una opción: ");
      scanf("%d",&opcion);

      if (opcion==1)
      {
        printf("Introduzca un nombre: ");
        gets(alum.nombre);
        gets(alum.nombre);

        for(x = 0; x < 5;x++)
        {
```

```
      if (strcmp(alumnos[x].nombre,alum.nombre)==0)
      {
       printf("\nNombre: %s\n",alumnos[x].nombre);
       printf("Nota: %f\n",alumnos[x].nota);
      }
    }
    printf("\n\n");
  }
  else if (opcion==2)
  {
    printf("Introduzca un nombre: ");
    gets(alum.nombre);
    gets(alum.nombre);

    for(x = 0; x < 5;x++)
    {
      if (strcmp(alumnos[x].nombre,alum.nombre)==0)
      {
       printf("Introduzca una nota: ");
       scanf("%f",&alumnos[x].nota);
       printf("\nNota modificada.");
      }
    }
    printf("\n\n");
  }
  else if (opcion==3)
  {
   sum=0;
   for(x = 0; x < 5;x++)
   {
     sum=sum+alumnos[x].nota;
   }
   printf("\nLa media de las notas es de: %f \n",(sum/5));
  }
  else if (opcion==4)
  {
   sum=0;
   cont=0;
   for(x = 0; x < 5;x++)
   {
     if (alumnos[x].nota<5)
     {
      sum=sum+alumnos[x].nota;
      cont++;
     }
   }
   printf("\nLa media de las notas inferiores a 5 es: %f \n",sum/cont);
```

```
      }
      else if (opcion==5)
      {
       mejor=0;
       for(x = 0; x < 5;x++)
       {
         if (alumnos[x].nota>mejor)
         {
          mejor=alumnos[x].nota;
          alum.nota=alumnos[x].nota;
          strcpy(alum.nombre,alumnos[x].nombre);
         }
       }
       printf("\nEl alumno con mejores notas es: %s \n",alum.nombre);
      }
      else if (opcion==6)
      {
       peor=10;
       for(x = 0; x < 5;x++)
       {
         if (alumnos[x].nota<peor)
         {
          peor=alumnos[x].nota;
          alum.nota=alumnos[x].nota;
          strcpy(alum.nombre,alumnos[x].nombre);
         }
       }
       printf("\nEl alumno con peores notas es: %s \n",alum.nombre);
      }
   }

      system("PAUSE");
   return 0;
}
```

BIBLIOGRAFÍA

1. DIDACT S.L. 2005. Manual de Programación Lenguaje C++. Colección Temarios Generales. Primera Edición. Editorial MAD S.L. España. 208 pág.

2. Marco A. Peña Basurto, José M Cela. 2000. Introducción a la Programación en C. Primera Edición. Ediciones UPC. Barcelona.

3. Diego R. Llanos Ferrari. 2010. Fundamentos de Informática y Programación en C. Primera Edición. Para Info, S.A. España.

4. José Antonio Gómez Ruiz. Fundamentos de Informática. ETSI Industrial 1 Universidad de Málaga.

5. García Carrillo, Rosalba. Técnicas de Programación. Editorial Mc Graw-Hill. 2003.

6. Gonzalo Ferreira Cortés. Informática Paso a Paso. Editorial Alfaomega 2011.

7. Aguilar, Luis Joyanes y Martínez, Ignacio Romero. "Programación en C –Metodología de la Programación", Mc Graw Hill, España, 2001.

8. Antonakos, James y otros. "Programación estructurada en C". Pearson -Prentice Hall, 2004.

9. Byron, Gottfried, "Programación en C". Editorial McGraw-Hill. Segunda Edición. 1997.

10. Dale, Nell. "Programación y resolución de problemas con C++". Editorial McGraw Hill, 2007.