



Elementos de Programación

UNIDAD 1. INTRODUCCION A LA PROGRAMACION

INDICE

| | |
|---|---|
| <i>INTRODUCCIÓN</i> | 2 |
| 1. <i>ESQUEMA BÁSICO DE UNA COMPUTADORA - MODELO DE VON NEUMANN</i> | 2 |
| 2. <i>EVOLUCIÓN DE LAS COMPUTADORAS.....</i> | 4 |
| 3. <i>CARACTERÍSTICAS GENERALES.....</i> | 4 |
| 4. <i>¿QUÉ ES UNA COMPUTADORA Y QUE PROBLEMAS PUEDE RESOLVER?.....</i> | 5 |
| 5. <i>PROGRAMACIÓN.....</i> | 6 |
| 5.1 <i>¿QUÉ ES PROGRAMAR?</i> | 6 |
| 5.2 <i>LENGUAJES DE PROGRAMACIÓN.....</i> | 6 |
| 5.3 <i>COMPILADORES E INTÉRPRETES</i> | 7 |
| 6. <i>ETAPAS PARA SOLUCIÓN DE PROBLEMAS CON LA COMPUTADORA</i> | 8 |
| 7. <i>CONSTRUCCIÓN DE ALGORITMOS</i> | 8 |
| 8. <i>PROGRAMACIÓN ESTRUCTURADA</i> | 9 |

UNIDAD 1- INTRODUCCION A LA PROGRAMACION

OBJETIVOS: *Informar al Alumno del mundo Informático, la computadora, sus características y evolución. Exponer el ámbito de la programación con distintos tipos de programas y lenguajes. Trasmitir los pasos para encarar un problema y como diseñar una posible solución al mismo.*

Introducción

Antes de comenzar se introducirá de forma sencilla algunos conceptos:

- **Computadora:** es un equipo electrónico capaz de realizar muchas operaciones muy rápidamente, con la particularidad de que no está diseñada para hacer una sola tarea, sino que es capaz de realizar tareas diferentes dándole las instrucciones de cómo hacerlo mediante un programa.
- **Programar:** es dar instrucciones a una computadora, decirle que es lo que queremos que haga.
- **Informática:** Las computadoras manejan datos y esos datos al procesarlos se transforman en información. El manejo de dicha información mediante un programa de computadora da origen a la informática haciendo que los datos e información puedan ser procesados en forma automática.
- **Digitalizar:** Hacer que los datos sean adaptados para ser procesados por un sistema informático.
- **Hardware:** Componentes físicos de la computadora.
- **Software:** Parte no tangible que se ejecuta en los equipos informáticos. Son los programas y/o aplicaciones.

Usted se encuentra en la era de la información. Cada vez más y más procesos cotidianos se encuentran digitalizados y los datos son manipulados por sistemas computacionales. Por ejemplo, al ir a un supermercado los productos son leídos con un lector de códigos de barra y procesados mediante un sistema que recupera su precio y va calculando el gasto realizado. También al subir a un colectivo y pasar la tarjeta SUBE, hay detrás un proceso informático que descuenta de nuestro saldo el viaje realizado. Estos son solo dos pequeños ejemplos, pero si piensa en cuantas cosas de la vida cotidiana dependen de un sistema informático se va a dar cuenta de que cada vez son más y más.

Las computadoras están presentes en la vida cotidiana y se presentan de muchas formas y tamaños: computadoras de escritorio, notebooks, tablets, teléfonos celulares, entre otros. Es decir, está rodeado de máquinas capaces de ejecutar distintos programas que permiten realizar tareas de una forma fácil y rápida.

1. Esquema Básico de una Computadora - Modelo de Von Neumann

John Von Neumann fue un matemático húngaro nacido en 1903, que diseñó una arquitectura general que define como deber estar organizada una máquina de propósito general (computadora).

Esta arquitectura fue tomada en cuenta a la hora de diseñar las primeras computadoras y sigue siendo la base fundamental de las computadoras actuales.

El diseño realizado por Von Neumann plantea la arquitectura en tres bloques principales:

- **CPU:** sigla que viene del inglés, Central Processing Unit (Unidad Central de Procesamiento), es la encargada de interpretar y ejecutar las instrucciones de los programas que se quieren ejecutar.
- **Memoria Principal:** Es un espacio donde se almacenan tanto los programas que se quieren ejecutar como los datos que dichos programas manipulan.
- **Unidad de Entradas y Salidas:** Es el bloque encargado de la comunicación con el mundo exterior, y, por lo tanto, con los usuarios. Por ejemplo, para mostrar información por una pantalla o recibir los datos ingresados por un mouse o teclado.

La figura 1 muestra los bloques principales de la arquitectura de Von Neumann.

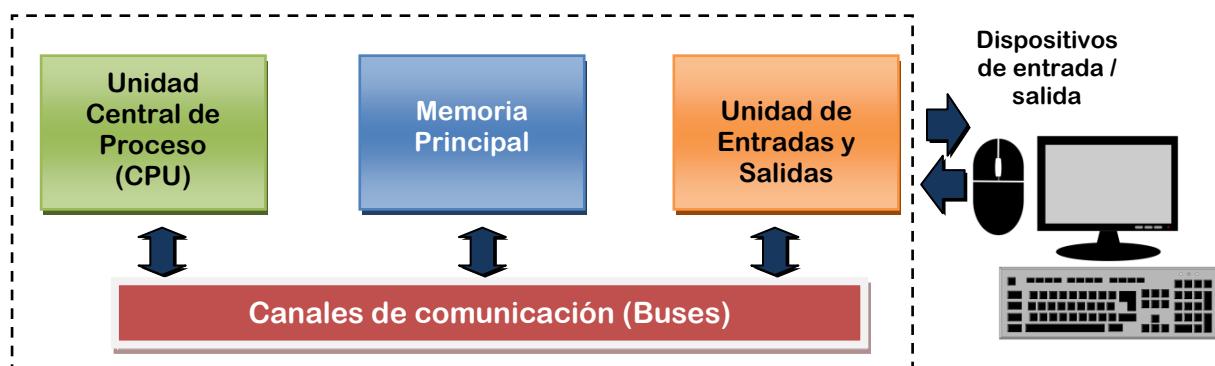


Figura 1: Modelo de Von Neumann

Este es un esquema general, ya que dentro de cada bloque hay varios componentes que trabajan en conjunto, para lograr que la computadora pueda ejecutar instrucciones y comunicarse con el usuario mediante dispositivos de entrada y salida como mouse, teclado, monitor, impresora, etc.

La encargada de ejecutar las instrucciones es la CPU, donde internamente incluye lo que se conoce como la unidad aritmético-lógica, que es capaz de realizar operaciones matemáticas y lógicas para poder procesar datos. Esas instrucciones se dan a la computadora mediante un programa que es almacenado en la memoria principal, y de allí se lleva a la CPU cada una de las instrucciones, una a una para ejecutarlas. En este esquema la memoria principal también guarda los datos con los que el programa trabaja y los resultados obtenidos. Esta memoria se divide en distintos "casilleros" donde cada uno puede guardar una cierta cantidad de información. Para acceder a cada casillero se utiliza una "dirección" diferente, por lo tanto, siempre que se recupera un dato o instrucción de la memoria se dice que accede a una dirección de memoria particular.

El bloque de entradas y salidas es un intermediario, que permite realizar la conexión con los dispositivos externos para permitir ingresar datos, o mostrar información. Este bloque también adecúa velocidades ya que los dispositivos externos son mucho más lentos que los componentes internos de la computadora.

Los bloques se comunican entre sí mediante distintos canales de comunicación llamados buses, donde por ellos pueden viajar direcciones de memoria, datos y señales de control.

2. Evolución de las computadoras

Las computadoras de hoy en día se presentan en una amplia variedad de formas, tamaños y precios. Las grandes computadoras de propósito general se utilizan en muchos negocios, universidades, hospitales y agencias gubernamentales para desarrollar sofisticados cálculos científicos y financieros.

Se conoce a estas grandes computadoras o grandes sistemas como mainframes, que necesitan de equipamiento y mobiliario especial, en cuanto a espacio, electricidad, humedad, etc. También existen otras computadoras más chicas en tamaño, como las computadoras personales que con la incorporación de las redes, permiten lograr recursos importantes y a un menor costo.

A comienzo de los cincuenta se disponía de grandes sistemas, aunque muy poca gente sabía cómo utilizarlos, lo hacían generalmente científicos, ingenieros y analistas financieros.

Durante los sesenta, fue cada vez más frecuente que aprendieran a programar grandes sistemas los estudiantes en las universidades, esto produjo la aparición de jóvenes profesionales provocando la desaparición del temor y recelo existentes hasta el momento, con la utilización de los grandes equipos.

A finales de los sesenta y comienzos de los setenta, se desarrollaron minicomputadoras, más pequeñas y menos caras, lo que permitió el acceso a las mismas de comerciantes más pequeños y en los hogares.

Hasta llegar a nuestra época, donde podríamos decir que es una herramienta imprescindible para cualquier profesional y comerciante para que pueda realizar sus actividades con eficiencia.

La evolución de las computadoras es el resultado de la innovación de los componentes electrónicos, donde con cada avance tecnológico nació una nueva generación de computadoras que mejoraban tanto en tamaño (haciéndose más pequeñas) como en velocidad (haciéndose más rápidas). Según la tecnología se pueden identificar 4 generaciones de computadoras:

- 1ra generación válvula de vacío
- 2da generación Transistor
- 3ra generación Circuito Integrado
- 4ta generación Microprocesador

Hoy en día si bien no hay un cambio radical en la tecnología, se avanza en la incorporación de varios núcleos de procesamiento en un mismo microprocesador, lo que permite ejecutar distintas instrucciones al mismo tiempo.

3. Características generales

Todas las computadoras digitales, independientemente de su tamaño, son básicamente dispositivos electrónicos que pueden transmitir, almacenar y manipular información (datos). Una computadora puede procesar distintos tipos de datos. Esto incluye datos numéricos, alfanuméricos (nombres, direcciones, etc.), datos gráficos (mapas, dibujos, fotografías, etc.), y sonido (música, lectura de textos, etc.). Desde el punto de vista del programador recién iniciado, los dos tipos de datos más familiares son los números y los caracteres, involucrando frecuentemente las aplicaciones de negocios el tratamiento de estos tipos de datos.

Para que la computadora procese un lote particular de datos es necesario darle un conjunto apropiado de instrucciones llamado programa. Estas instrucciones se introducen en la computadora y se almacenan en una parte de la memoria de la máquina.

Un programa almacenado se puede ejecutar en cualquier momento. La ejecución de un programa supone lo siguiente:

1. Un conjunto de datos, los datos de entrada, se introduce en la computadora (desde un teclado o un archivo), y se almacena en una porción de memoria de ésta.
2. Los datos de entrada se “procesarán” para producir ciertos resultados deseados, que son los datos de salida.
3. Los datos de salida, y probablemente algunos de los datos de entrada, se imprimirán en papel o se presentarán en un monitor.

Este procedimiento de tres pasos se puede repetir tantas veces como se desee. En cualquier caso, se debe tener presente que estos pasos, especialmente el 2 y el 3, pueden ser largos y complicados.

Ejemplo. Una computadora ha sido programada para calcular el área de un círculo utilizando la fórmula:

Área = $\pi \times r^2$, dando el valor numérico del radio como dato de entrada, es necesario dar los siguientes pasos para efectuar el cálculo:

1. Leer el valor numérico del radio del círculo.
2. Calcular el valor del área utilizando la fórmula anterior. Este valor se almacenará entre los datos de entrada en la memoria de la computadora.
3. Imprimir (mostrar en pantalla) los valores del radio y el área correspondiente.
4. Parar.

Cada uno de estos pasos requiere de una instrucción o más en un programa.

Lo anteriormente dicho ilustra dos características importantes de una computadora digital: memoria y capacidad de ser programada. Otras características importantes son su velocidad y fiabilidad.

4. ¿Qué es una computadora y que problemas puede resolver?

Para decidir cuáles y cómo se pueden resolver los problemas recordemos cómo funciona una computadora digital, comencemos por dar una definición:

“Una computadora es un sistema electrónico dedicado al **procesamiento de datos** (información), formado por varias unidades, cuyo funcionamiento viene dictado por el **programa** que se halla almacenado en su memoria principal”

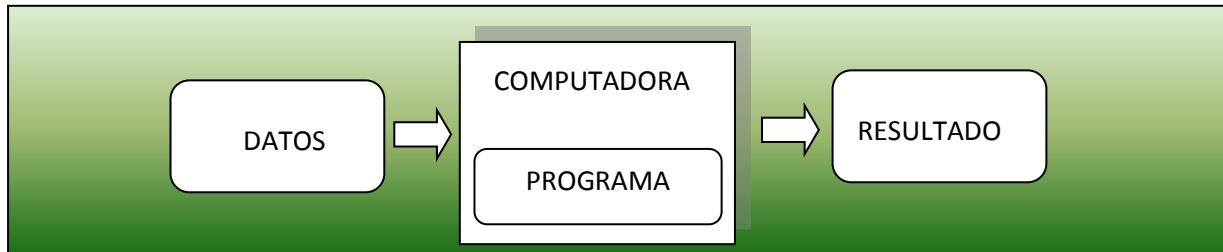


Figura 2: Resolución de problemas mediante una computadora

La computadora es un mecanismo complejo que solo opera sobre datos, resolviendo operaciones muy sencillas (sumas y comparaciones) a una elevadísima velocidad y con gran precisión.

Como la tarea es la resolución de problemas utilizando una computadora, se debe expresar una secuencia de operaciones lógicas, en un lenguaje suficientemente básico y preciso, en

instrucciones que la computadora sepa resolver, éstas serán las instrucciones que componen a cada lenguaje de programación.

Básicamente, la computadora puede realizar los siguientes tipos de operaciones:

- Ingresar datos. Pueden ser varios y de diferente tipo – numéricos y no numéricos.
- Realizar cualquier tipo de operación algebraica y/o lógica, efectuando comparaciones y generando ciclos definidos o condicionados.
- Almacenar información en forma transitoria (memoria principal), o permanente (memorias auxiliares)
- Exhibir datos y/o resultados en diferentes dispositivos como ser en una pantalla, una impresora, etc.

5. Programación

5.1 ¿Qué es programar?

Programar es la actividad de comunicar a la computadora lo que se tiene que hacer. Cualquiera que desee que la computadora haga algunas cosas, deberá acoplarse a este tipo de comunicación persona-computadora, utilizando un lenguaje concreto. Una manera de apreciar lo que es un programa, es considerarlo como si fuera un plan detallado para transformar información, aceptando datos y produciendo una nueva información. A las reglas de transformación se las denomina algoritmos.

El problema que se establece al escribir un programa es básicamente el de la comunicación entre un ser humano y una computadora. Probablemente el ser humano sabe cómo encarar el problema, o al menos sabe cómo resolver algunos problemas afines, pero si el problema se va a resolver con la ayuda de una computadora, los humanos deben transmitirle su propósito.

Los lenguajes humanos no son apropiados para la comunicación con las computadoras, principalmente porque acarrean una notable serie de ambigüedades. De persona a persona la comunicación puede tolerar esta ambigüedad debido a acuerdos tácitos, y a que la comunicación no verbal está también presente. Pero en la comunicación con una máquina todo se lleva a cabo de forma literal, de modo que se necesita ser muy preciso y específico en describir la labor que se tenga que realizar. De ahí que la gente haya elaborado lenguajes especializados, denominados lenguajes de programación, para la comunicación entre la persona y la máquina.

5.2 Lenguajes de programación

Se necesita un sistema para decirle a la computadora lo que se quiere hacer. Los lenguajes de programación permiten realizar dicha tarea. Se trata de un lenguaje especial que no incluye ambigüedades. Estos lenguajes deben tener descripciones precisas de las tareas que se deben realizar.

Las computadoras actuales trabajan en sistema binario, es decir, con dos niveles de corriente que se representa con 0 y 1. Para un programador sería muy difícil escribir instrucciones de un programa como una secuencia de ceros y unos, por lo tanto, surgen los lenguajes de programación.

Cada CPU, cuyo componente físico en una computadora personal es el microprocesador, puede interpretar y ejecutar distintas instrucciones. Hay conjuntos de instrucciones estandarizadas que fueron evolucionando a lo largo del tiempo, por lo tanto, los microprocesadores en general soportan la ejecución de varios sets de instrucciones diferentes según su arquitectura.

El lenguaje de programación más cercano a la computadora es el Assembler, donde cada instrucción del lenguaje de programación se corresponde directamente con una instrucción que el microprocesador es capaz de ejecutar. Este lenguaje está muy ligado al hardware ya que solo funcionará con procesadores que entiendan dichas instrucciones. A este tipo de lenguajes se los conoce como lenguajes de bajo nivel por su cercanía al hardware.

Programar en Assembler no es una tarea fácil, ya que tareas sencillas muchas veces requieren de varias instrucciones para lograr realizarlas, como, por ejemplo, una multiplicación se hará sumando sucesivamente un número, y, además, el programa realizado va a funcionar en un hardware particular. Por este motivo, surgen lenguajes de programación más avanzados, donde una sola instrucción del lenguaje se traduce luego en varias instrucciones interpretables por el procesador haciendo la tarea del programador más sencilla. Además, estos lenguajes avanzados se independizan del hardware ya que el mismo programa puede ser traducido para ser ejecutado en distintos procesadores con distintos sets de instrucciones. Por eso se los llama lenguajes de Alto Nivel.

Todo lenguaje de programación está compuesto por:

- **Léxico:** que símbolos del lenguaje puedo utilizar para escribir mi programa. Letras, números y algunos símbolos conforman el léxico de la mayoría de los lenguajes.
- **Sintaxis:** son las reglas que indican de qué forma se combinan los símbolos para lograr escribir instrucciones válidas. La sintaxis también define palabras que están reservadas por el lenguaje y son necesarias para la escritura de las instrucciones. Por ejemplo, la palabra `if`, es reconocida en muchos lenguajes como el comienzo de una condición lógica, por lo tanto, esa palabra no puede utilizarse para otro fin.
- **Semántica:** cada instrucción válida del lenguaje de programación tiene un propósito particular. Indica algo que se quiere que la computadora haga. Ese sentido de cada instrucción es la semántica. La semántica, por lo tanto, es el significado de las instrucciones de un programa.

Un programa de computadora es básicamente una colección secuencial de “sentencias” siguiendo cada una de ellas las reglas sintácticas del lenguaje. Cada sentencia contiene información para la computadora. Usted puede considerar estas sentencias declarativas como si fueran absorbidas por la computadora una a una en un orden secuencial; entonces la computadora “ejecuta” cada sentencia, hace lo que la sentencia le pide que haga.

| | |
|-------------|-------------------------|
| Sentencia 1 | (primera en ejecutarse) |
| Sentencia 2 | (segunda en ejecutarse) |
| --- | ----- |
| Sentencia n | (enésima en ejecutarse) |

5.3 Compiladores e intérpretes

El programa en el lenguaje en que se escribe las instrucciones se llama programa fuente. Este programa fuente es un archivo de texto que la computadora no puede comprender directamente, y, por lo tanto, necesita que sea traducido al lenguaje máquina.

Hay dos mecanismos diferentes para realizar esa traducción, mediante un compilador o mediante un intérprete. El compilador toma el programa fuente, analiza si está bien escrito según la sintaxis del lenguaje de programación elegido y genera un programa en un lenguaje que la computadora puede interpretar llamado programa objeto. Ejemplos de lenguajes compilados son C, C++, Java, Fortran. En cambio, un intérprete va tomando una a una las instrucciones del programa fuente, las traduce y las ejecuta, es decir, no analiza todo el programa, sino que va traduciendo y ejecutando las instrucciones a medida que lee el programa fuente. El problema de los lenguajes interpretados es que, al no analizar el programa completo, puede ser que se comience a ejecutar un programa

hasta que se llegue a una instrucción que no pueda interpretarse por lo que se terminaría abruptamente la ejecución del programa. Un ejemplo de un lenguaje interpretado es javascript, utilizado para ejecutar instrucciones en una página web y agregarle funcionalidad.

6. *Etapas para solución de problemas con la computadora*

La solución de problemas mediante la computadora consta de ocho etapas, articuladas de tal forma que cada una depende de las anteriores, lo cual indica que se trata de un proceso complementario y, por lo tanto, cada paso exige el mismo cuidado en su elaboración. Las ocho etapas son:

1. Comprensión, definición y delimitación del problema a solucionar.

Para resolver el problema es fundamental conocerlo y delimitarlo por completo, determinar con qué datos se cuenta, la información a obtener. Es importante tener claridad sobre el problema a resolver para no dar soluciones incorrectas.

2. Estrategia - definir los algoritmos.

Disponer de los algoritmos necesarios para resolver los problemas. Identificar en forma manual como resolvería sin la computadora el problema propuesto, realizando todos los pasos necesarios en una hoja entendiendo bien el proceso.

3. Diagrama de flujo.

Es la primera etapa para realizar luego de tener en claro el problema y su manera de resolver. En esta etapa, es donde se determinan los pasos e instrucciones que deben llevarse a cabo y el orden lógico de su ejecución, para una eficiente solución al problema representado en forma gráfica con los símbolos de diagramas propuestos. Se puede decir que es aquí donde radica toda la dificultad para solucionar un problema a realizar.

4. Prueba de escritorio.

Consiste en hacer un seguimiento manual, a través, de un lote de prueba que permita verificar los pasos que realiza el diagrama desarrollado, esto permitirá reconocer a través del resultado si funciona correctamente o no.

5. Codificación.

Es la escritura de las instrucciones, determinadas en la etapa de la diagramación, en un lenguaje de programación determinado.

6. Compilación o interpretación del programa.

En esta etapa, la computadora chequea si todas las instrucciones están escritas correctamente en el código utilizado, respetando, simbología y sintaxis.

7. Ejecución del Programa.

Luego de la etapa anterior, si el compilador no dio errores, el programa objeto es ejecutado, para llegar a los resultados esperados.

8. Evaluación de los resultados.

Obtenidos los resultados, se los evalúa para verificar si son correctos. En caso contrario, se revisa las etapas anteriores para detectar la falla o error.

7. *Construcción de Algoritmos*

De las etapas mencionadas, ésta es la que traerá las mayores dificultades por lo cual será la principal del trabajo a realizar. Se debe destacar que el concepto de "algoritmo" es similar al de "estrategia", pero con la diferencia que en lugar de hablar de "procesos o tareas ", se hablará de "instrucciones", lo cual involucra un mayor grado de detalle, donde estarán presentes operaciones minuciosas y precisas.

El algoritmo se lo puede describir en forma gráfica mediante un diagrama, y luego, codificarlo en un lenguaje de programación particular generando un programa.

La utilización de diagramas para pensar y diseñar los algoritmos antes de codificarlos tiene las siguientes ventajas:

1. permite una sencilla y rápida visualización total del problema y sus distintas alternativas.
2. es un verdadero medio de comunicación entre quien explica y quien aprende.
3. es un medio claro y conciso de documentación de los problemas.
4. facilita los posibles cambios y visualiza distintas alternativas posibles.

Existen varios métodos para encarar la construcción del algoritmo. A estas diferentes formas de encarar la construcción de una solución a un problema mediante la computadora, se las conoce como paradigmas de programación. Entre los principales paradigmas se puede mencionar: programación estructurada, orientada a objetos, funcional, lógica, etc. Un buen punto de partida para comenzar a comprender y dominar el diseño de algoritmos es utilizar el enfoque de la programación estructurada, ya que nos dará la base necesaria para poder luego utilizar otros paradigmas diferentes.

8. Programación estructurada

Este método tiene como soporte teórico al teorema fundamental de la programación estructurada, (Jacerina y Bohn) que afirma que todo algoritmo puede ser construido en forma correcta utilizando "únicamente" tres estructuras básicas:

- **SECUENCIAL:** las instrucciones del programa se ejecutan una a continuación de la otra, en forma ordenada, y siguiendo la secuencia que el programa indica.
- **SELECCIÓN:** permite tomar decisiones en el programa, abriendo distintos caminos haciendo que se ejecuten determinadas instrucciones solo si se cumple alguna condición particular. De esta forma el programa puede tener distintas variantes de ejecución y adaptarse a los datos que se están procesando.
- **ITERACION:** esta estructura permite repetir un conjunto de instrucciones, ya sea una cantidad de veces fija o dependiendo de alguna condición. Al utilizar instrucciones de iteración los programas quedan más cortos, y se pueden modificar más fácilmente que si se escribieran varias veces las mismas instrucciones.

También la programación estructurada indica que todo programa o módulo de un programa debe tener un comienzo y un único punto de finalización. Esto ayuda a la comprensión y diseño de los algoritmos haciendo que el proceso se ejecute en forma secuencial hasta llegar al final de mismo, existiendo un único punto donde el programa finaliza. Más adelante se verá que hay lenguajes que soportan que existan múltiples puntos de finalización de un programa. En este tipo de lenguajes para seguir la programación estructurada se recomendará unificar las salidas a un único punto para simplificar la lógica.

Una vez comprendidas las estructuras básicas de la programación estructuradas se verán distintas herramientas y estructuras de datos que permiten realizar programas mas complejos de forma más reducida y óptima.



Elementos de Programación

UNIDAD 2. ESTRUCTURA SECUENCIAL

INDICE

| | | |
|--------------|---|----------|
| 1. | ESTRUCTURA SECUENCIAL | 2 |
| 2. | ALMACENAMIENTO DE LOS DATOS | 2 |
| 2.1 | VARIABLE:..... | 2 |
| 2.2 | TIPOS DE DATOS: | 2 |
| 2.3 | IDENTIFICADORES: | 3 |
| 2.4 | CONSTANTE:..... | 3 |
| 3. | REPRESENTACIÓN GRÁFICA DE UN ALGORITMO - DIAGRAMA..... | 4 |
| 3.1 | INGRESO DE DATOS | 4 |
| 3.2 | OPERACIONES..... | 5 |
| 3.2.1 | Operación de Asignación | 5 |
| 3.2.2 | Operaciones matemáticas | 6 |
| 3.2.3 | Salida de Resultados / Mensajes | 7 |
| 3.2.4 | Símbolos de indicación..... | 8 |
| 4. | ALGUNOS EJEMPLOS | 8 |
| | EJEMPLO 1 | 8 |
| | EJEMPLO 2 | 9 |
| | | 9 |
| | EJEMPLO 3 | 9 |

UNIDAD 2 - Estructura Secuencial

OBJETIVOS: Construir algoritmos utilizando la programación estructurada. Resolver sencillos ejemplos con “estructura secuencial” representándolos gráficamente mediante diagrama de flujo.

1. Estructura Secuencial

La primera estructura básica de la programación estructurada es la secuencial. Consiste en una sucesión de instrucciones que se ejecutan en el mismo orden en que fueron escritas, una a continuación de la otra.

Un algoritmo secuencial se expresa como una sucesión de instrucciones que se ejecutan TODAS, una a continuación de la otra, teniendo un único punto de inicio y un único punto de fin. Esas instrucciones pueden ser de distintos tipos:

- Operaciones aritméticas
- Operaciones de asignación
- Ingreso de datos
- Salida / Mostrar información

2. Almacenamiento de los datos

Antes de poder escribir las instrucciones que forman parte de los algoritmos, debe saber cómo se guardan los datos internamente en la computadora, ya que las instrucciones serán operaciones que se realizarán sobre dichos datos.

Cada fragmento de información almacenado en la memoria de la computadora es codificado como una combinación de ceros y unos. Estos ceros y unos se llaman bits (dígitos binarios). Un dispositivo electrónico representa cada BIT, el cual estará de alguna forma: “apagado” (cero) o “encendido” (uno).

A un grupo de 8 bit, se lo denomina byte. Normalmente, un carácter (una letra, un dígito, un carácter especial) ocupara un byte de memoria.

2.1 Variable:

Una variable es un espacio de memoria reservado para almacenar un valor que corresponde a un tipo de dato soportado por el lenguaje de programación. Una variable, es representada y usada a través de una etiqueta (un nombre) que le asigna un programador, o que ya viene predefinida.

Por ejemplo, se define una variable con el nombre NUM, en la cual se almacena el número 8, NUM es la etiqueta que le asigna el programador y ocho es su contenido. Lo que se carga en NUM puede ser cualquier número, 1, 2, 3, 0, 12, 33, etc., (variable). Una variable, por lo general, como su nombre lo indica, puede variar su valor durante la ejecución del programa.

2.2 Tipos de Datos:

Los tipos de datos indican que es lo que se puede almacenar en una variable. Por ejemplo, un número entero, un carácter, un número con decimales, etc.

Cada tipo de dato ocupa en la memoria de la computadora una cantidad distinta de bytes. Por lo tanto, para optimizar un programa, por ejemplo, si se necesita trabajar con números pequeños, se puede utilizar una variable con un tipo de dato que permita almacenar números pero hasta cierta cantidad haciendo que esa variable ocupe menos espacio en memoria.

Dependiendo del lenguaje de programación usado, también es posible cambiar el tipo de dato que almacena una variable a lo largo del programa, pero en el lenguaje C, que es el que se verá en esta materia, el tipo de dato no puede modificarse, por lo que debe definirse la variable correctamente según lo que se necesite almacenar en ella.

Los tipos de dato que se utilizarán para los algoritmos son los siguientes:

- **Caracter:** para almacenar un único carácter, una letra, un símbolo, un dígito numérico.
- **Entero:** para almacenar un número entero (sin decimales).
- **Real:** para almacenar un número con decimales.

2.3 Identificadores:

Los nombres de variables o identificadores deben ser nemotécnicos, es decir, que con solo leer el nombre de la variable se pueda entender o determinar con facilidad lo que ella significa o contiene. En el ejemplo anterior la variable con el nombre NUM almacena un número por lo que se ve a simple vista, pero si a esta variable se le hubiese dado el nombre X o DS, estos nombres pueden significar muchas cosas o, tal vez, no significar absolutamente nada haciendo que lectura del programa sea más compleja. Los identificadores tienen ciertas reglas de escritura que van a depender del lenguaje de programación que se esté utilizando. Como regla general, un identificador tiene que cumplir con las siguientes reglas:

- No puede comenzar con números.
- No puede tener espacios.
- No puede tener acentos ni letra ñ.
- No puede tener símbolos (excepto el guión bajo que es el único permitido).

A continuación se muestra la tabla 1 con ejemplos de identificadores:

Tabla 1: ejemplo de identificadores

| Identificador | ¿es correcto? |
|---------------|---------------|
| Num | CORRECTO |
| Numero_1 | CORRECTO |
| _Número1 | CORRECTO |
| Num1 | CORRECTO |
| Número | INCORRECTO |
| 1Número | INCORRECTO |
| Num 1 | INCORRECTO |
| año | INCORRECTO |

Algunos lenguajes son sensibles a los cambios entre mayúsculas y minúsculas (case-sensitive), y otros no. En aquellos que son sensibles, por ejemplo num1 y Num1, hacen referencia a dos identificadores diferentes por lo que debe respetarse siempre el mismo nombre cada vez que se lo utiliza.

2.4 Constante:

Constante son todos aquellos valores que no cambian en el transcurso de un algoritmo y son introducidos en el momento de utilizarse.

Por ejemplo, el número 10 es una constante del tipo entera que puede asignarse a una variable de dicho tipo. En cambio, si se quiere representar una constante del tipo carácter se expresa entre comillas simples, por ejemplo, 'A' , '1', 'a' , '-' son constantes de tipo carácter.

3. Representación gráfica de un algoritmo - Diagrama

Para comenzar a diseñar los algoritmos, y ayudar a encontrar una solución al problema planteado, se va a utilizar una representación gráfica de las instrucciones que se le darán a la computadora, de esta forma se podrá visualizar más fácilmente el algoritmo y una vez terminado y probado se puede codificar en el lenguaje de programación elegido.

Todo diagrama tendrá un símbolo de inicio, una serie de operaciones y un símbolo que indica el fin del mismo. Recuerde que inicialmente realizará algoritmos en forma secuencial, por lo tanto, los símbolos que utilizará son los que están en la Figura 1.

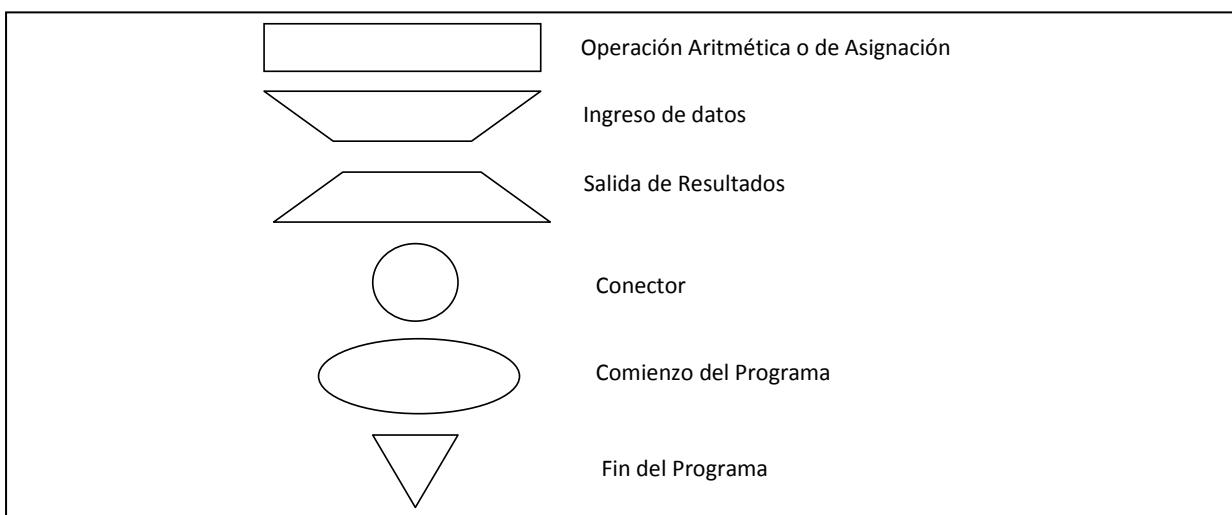


Figura 1. Símbolos para la construcción de diagramas que representan algoritmos secuenciales.

3.1 Ingreso de datos

Para resolver un problema, se necesitan datos, esos datos serán procesados para alcanzar el resultado esperado.

Los algoritmos, en general, se diseñan para que puedan funcionar con distintos datos según la necesidad del usuario que esté utilizando el programa. Por ejemplo, poca utilidad tendrá un programa que calcule una suma de números pre-establecidos: si se suma 5 + 2, siempre el resultado será 7, por lo tanto, hacer un programa con esos números fijos (constantes) no tiene mucha utilidad. En cambio si esos dos números a sumar pueden ingresarse y variar cada vez que se ejecute el programa, se podrá utilizar para sumar 5 + 2, 9 + 4, 25458 + 24577, o para cualquier par de números que se deseé.

Analice el problema a resolver: Se necesita realizar un programa que permita sumar dos números cualquiera. Una forma de darle la información a la computadora es ingresar ese número por teclado, ¿pero una vez ingresado donde queda ese número? Bien, se necesita entonces un lugar para guardar cada uno de los números que se ingresan en el programa, y como vimos anteriormente, los datos se guardan en la memoria de la computadora en espacios reservados que se llaman variables.

El programa va a necesitar entonces dos variables para los datos de entrada, una variable para guardar el primer número a sumar, y la otra para guardar el segundo número a sumar:

- En la variable que se llama Num1 se va a guardar el primer número
- En la variable que se llama Num2 se va a guardar el segundo número

En el algoritmo cuando se quieran ingresar uno o más datos se utilizará el símbolo de ingreso, y dentro del símbolo se pondrá el nombre de la o las variables donde se guardará la información a ingresar. Si se ingresa más de un dato, las variables se pondrán separadas por coma. La figura 2 muestra la presentación gráfica para permitir el ingreso de datos desde teclado y que se almacenen los datos ingresados en las variables Num1 y Num2.

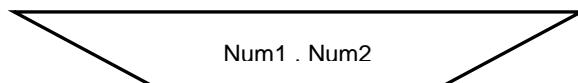


Figura 2. Simbología para el ingreso de dos datos que se almacenarán en las variables Num1 y Num2

Num1 y Num2 son dos variables distintas. Num1 va a contener el primer dato que el usuario ingrese por teclado, y Num2 el segundo. Antes de ingresar los datos en dichas variables no se sabe qué hay, por lo tanto, no se pueden utilizar si antes no se leyó un dato o se les asignó un valor (proceso que se verá a continuación).

3.2 Operaciones

Sobre los datos ingresados se realizan operaciones para obtener el resultado deseado. Se pueden realizar operaciones matemáticas y de asignación.

3.2.1 Operación de Asignación

La asignación consiste en guardar un dato en una variable. El dato puede ser:

- Otra variable
- Una constante
- El resultado de una operación matemática

La asignación se realiza de derecha a izquierda, es decir, que a la izquierda se escribe el identificador de la variable en la cual desea guardar el dato, luego se escribe el símbolo = (igual) que indica la operación de asignación, y a la derecha se coloca la constante, variable u operación matemática. Gráficamente la operación de asignación se representa con un rectángulo. En la Figura 3 pueden verse distintos ejemplos de asignación.

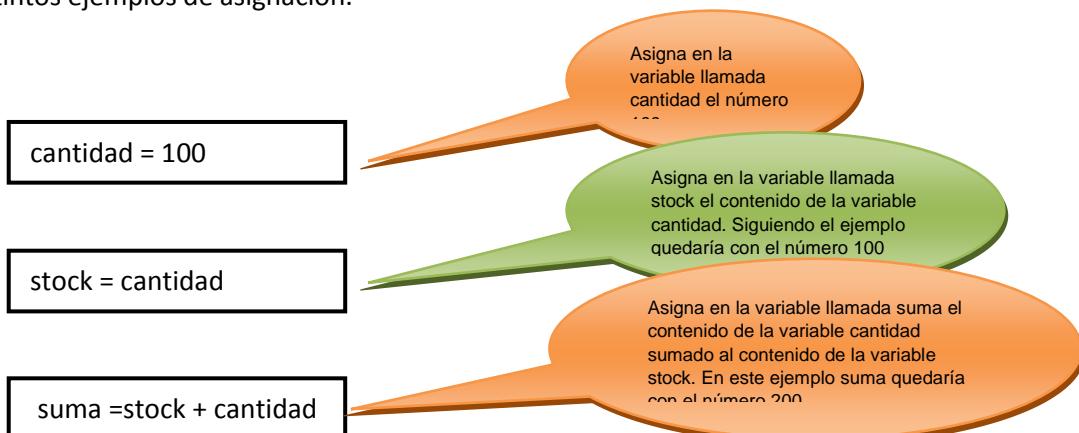


Figura 3. Ejemplos de asignación

3.2.2 Operaciones matemáticas

Sobre los datos se pueden realizar distintas operaciones. En la tabla 2 se detallan las operaciones posibles y el símbolo que se utiliza para expresarlas. A estos símbolos se los denomina operadores matemáticos.

Tabla 2. Operadores Matemáticos

| Operación | Operador matemático |
|-----------------------------|---------------------|
| Suma | + |
| Resta | - |
| Multiplicación | * |
| División | / |
| Resto de la división entera | % |

Las operaciones se pueden realizar tanto sobre variables (tomando el contenido que tienen almacenado), como sobre constantes. En la figura 4 pueden verse distintos ejemplos de operaciones que se realizan una a continuación de la otra.

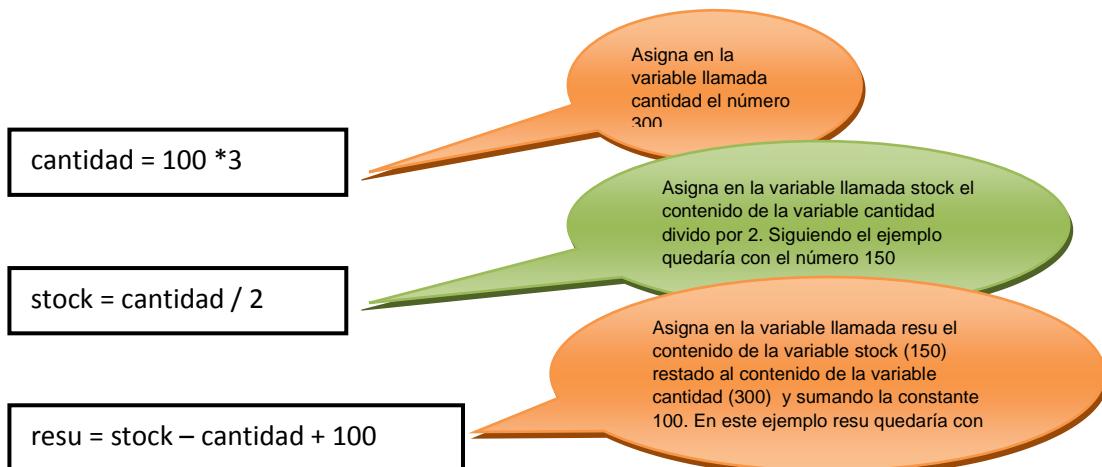


Figura 4. Ejemplos de operaciones matemáticas

Una mención especial requiere la operación de división ya que se comporta diferente según sea el tipo de dato de las variables o constantes sobre las que se está operando. Por ejemplo, al dividir el numero 15 con el numero 2 ($15 / 2$) como ambas son constantes de tipo entero la operación dará por resultado la de división entera. Es decir, que el resultado de $15 / 2$ será 7 y no 7.5 como sería lo esperado. Esta característica hay que tenerla en cuenta al realizar las operaciones ya que puede llevar a resultados erróneos pero brinda la posibilidad de realizar muchas otras tareas de forma sencilla, como por ejemplo, al descomponer un número. Siguiendo el ejemplo anterior si se quiere obtener 7.5 como resultado al menos una de las constantes debe ser del tipo real, es decir, un número con decimales. Como separador de decimales se usa el punto, por lo tanto, el resultado de $15. / 2$ será 7.5. El mismo resultado puede obtenerse al realizar la operación $15 / 2$.

Los operadores tienen precedencia al igual que en matemática, por ejemplo, el operador suma (+) separa términos y se ejecuta luego de realizar las operaciones de mayor precedencia como la multiplicación o la división. Muchas veces es necesario cambiar dicha precedencia y, por lo tanto, debemos agrupar las operaciones. Para ello se utilizan paréntesis. En una expresión pueden utilizarse todos los niveles de paréntesis que sean necesarios (NO corchetes). Por ejemplo, en la siguiente instrucción:

```
suma = 3 + 2 * 5
```

La variable `suma` quedará con el valor 13 ya que primero realiza la multiplicación y luego la suma. En cambio, si la misma instrucción se escribiera de la siguiente manera:

```
suma = (3 + 2) * 5
```

La variable `suma` quedará con el valor 25 ya que se utilizaron los paréntesis para resolver la suma primero que la multiplicación.

3.2.3 Salida de Resultados / Mensajes

Una vez realizado los cálculos sobre los datos el resultado debe mostrarse al usuario. Para ello, se utiliza la simbología que se muestra en la figura 5, donde pueden verse distintas formas de mostrar un resultado, donde se puede combinar datos con mensajes literales. Por defecto, la salida se realizará por pantalla, es decir, que se especifica lo que el usuario va a visualizar en la pantalla. Los mensajes se escriben entre comillas, haciendo que todo lo que se ponga entre las comillas se vea tal cual en la pantalla. Ahora si se desea mostrar el resultado de una operación que está almacenado en una variable, el identificador de la variable NO se pone entre comillas, de esa forma lo que se verá en pantalla es el contenido que tiene la variable en ese momento.

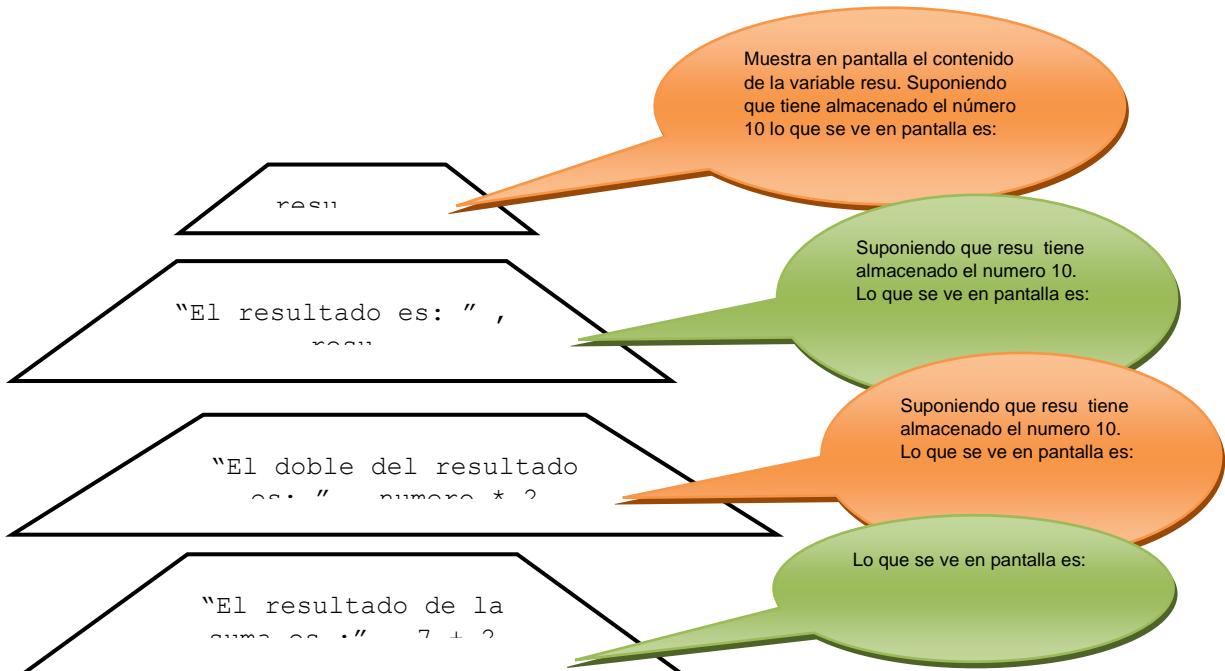


Figura 5. Ejemplos de salida por pantalla

3.2.4 Símbolos de indicación

Además de la entrada, salida y proceso hay tres símbolos adicionales que se utilizan para dar una mayor legibilidad al diagrama. Todo diagrama comenzará con un óvalo que indica el comienzo del algoritmo, dentro de este óvalo opcionalmente se puede escribir un nombre que represente el algoritmo que se está desarrollando. Al finalizar y para cerrar el diagrama se utilizará un triángulo invertido donde opcionalmente dentro se puede poner la letra F que indica fin. Por último, si se está desarrollando un diagrama largo que no entra en una hoja y se desea continuar en la hoja siguiente se utiliza un círculo como conector de la siguiente forma: al final de la hoja en lugar de la instrucción siguiente se pone el círculo con un número dentro, por ejemplo el número 1, ya que es el primer conector del algoritmo, luego en la hoja siguiente se repite en la parte superior el círculo con el mismo número para indicar que allí continua el diagrama. La Figura 6 muestra un ejemplo de la utilización de los símbolos de indicación.

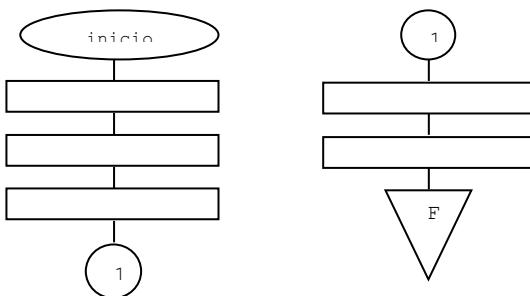
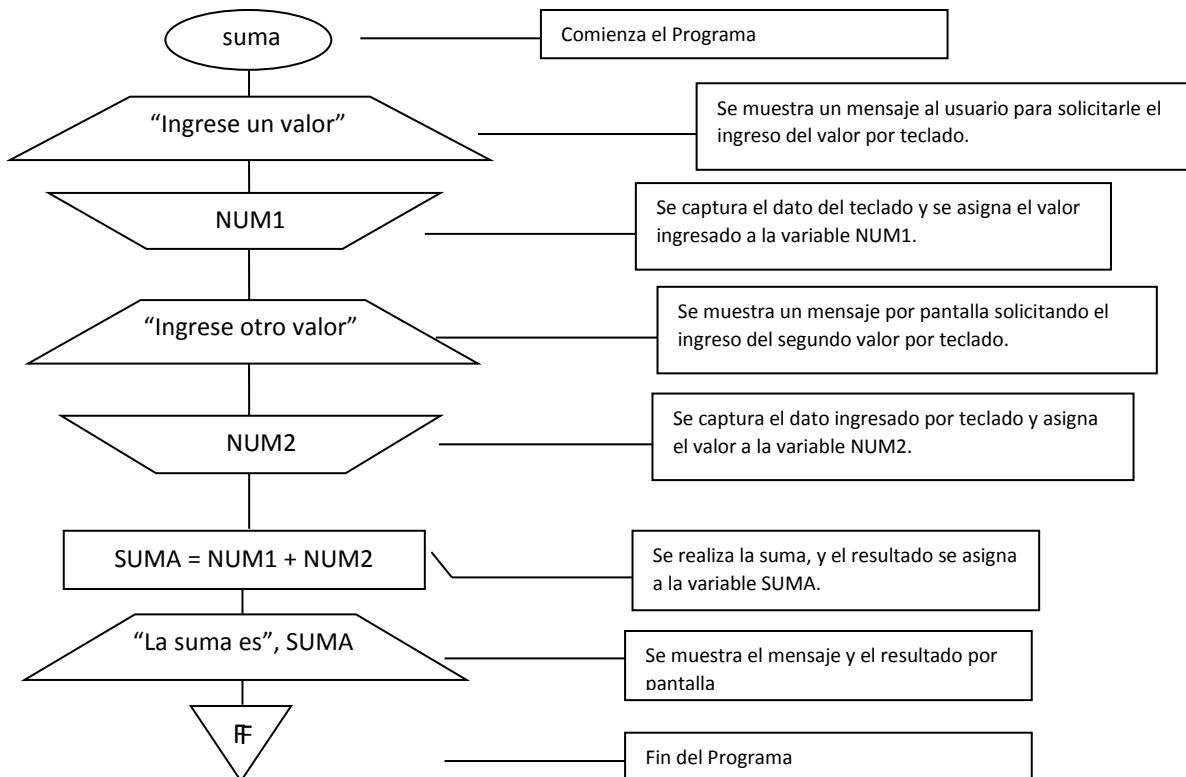


Figura 6. Ejemplos de utilización de símbolos de indicación

4. Algunos ejemplos

Ejemplo 1

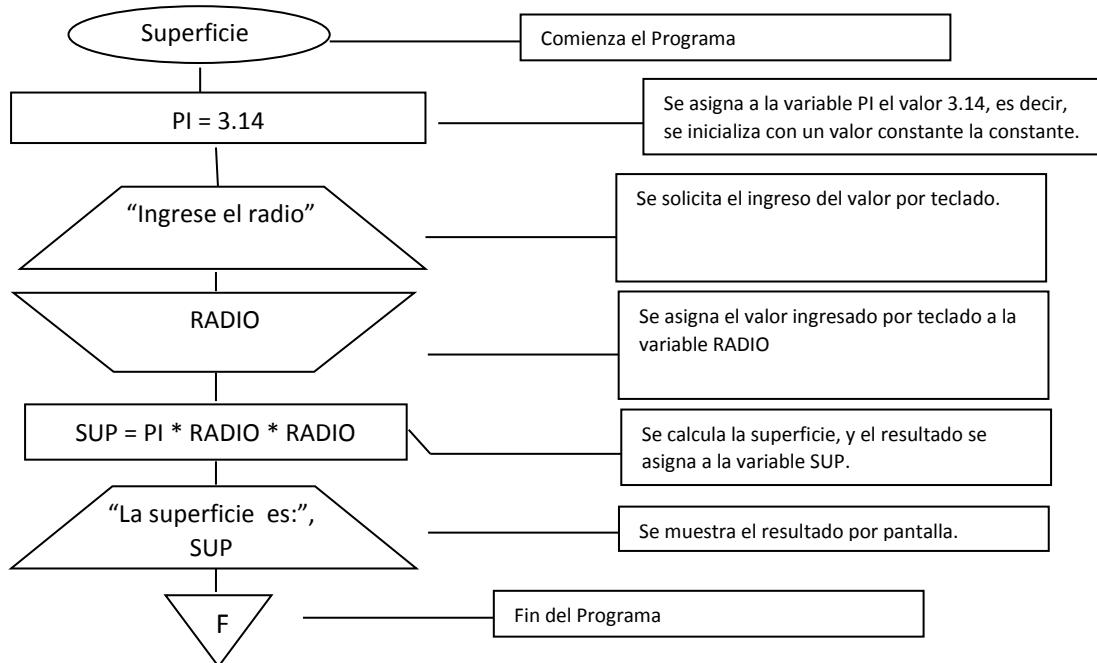
Realizar un programa que permita ingresar dos números enteros por teclado, realice la suma, y muestre por pantalla el resultado.



En este ejemplo se incorporaron al diagrama todos los mensajes que se muestran al usuario para indicar lo que debe hacer, en ejercicios siguientes, más complejos, los mensajes al usuario pueden ser omitidos para facilitar el diagrama, pero luego sí, al codificar en el lenguaje de programación, es muy importante siempre poner mensajes aclaratorios indicándole al usuario lo que debe hacer y cómo se comporta el programa.

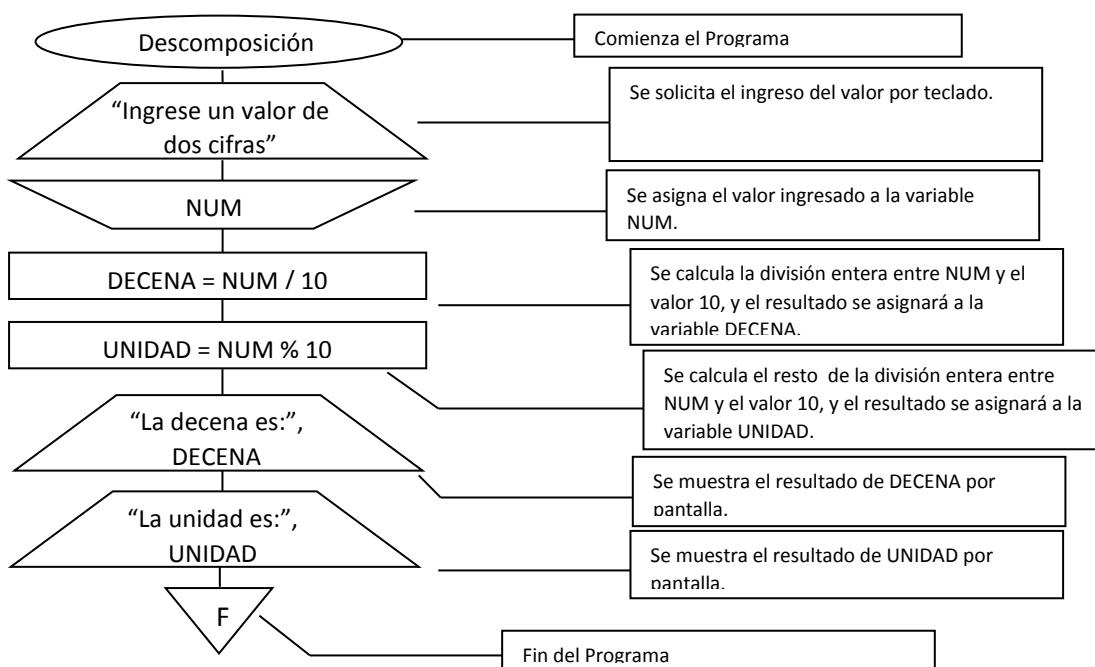
Ejemplo 2

Realizar un programa que permita ingresar el radio de un círculo, calcule la superficie, y muestre por pantalla el resultado.



Ejemplo 3

Realizar un diagrama que permita ingresar un número de dos cifras, y muestre por pantalla las unidades y las decenas.





Elementos de Programación

UNIDAD 3. EL LENGUAJE DE PROGRAMACION C

INDICE

| | | |
|----------------|--|-----------|
| 1 | UN POCO DE HISTORIA | 2 |
| 2 | DESDE EL CÓDIGO FUENTE HASTA EL EJECUTABLE | 2 |
| 3 | ESTRUCTURA GENERAL DE UN PROGRAMA EN C | 4 |
| 3.1 | BLOQUE DECLARATIVO GENERAL | 4 |
| 3.1.1 | DIRECTIVAS AL PREPROCESADOR..... | 4 |
| 3.1.1.1 | INCLUDE | 5 |
| 3.1.1.2 | DEFINE | 6 |
| 3.1.2 | DEFINICIÓN DE VARIABLES GLOBALES Y ESTRUCTURAS..... | 6 |
| 3.1.3 | PROTOTIPOS DE FUNCIONES PROPIAS | 6 |
| 3.2 | BLOQUE PRINCIPAL | 7 |
| 3.3 | DESARROLLO DE FUNCIONES PROPIAS | 7 |
| 4 | ELEMENTOS DEL LENGUAJE..... | 7 |
| 4.1 | ALFABETO..... | 8 |
| 4.2 | IDENTIFICADORES | 8 |
| 4.3 | CONSTANTES | 8 |
| 4.4 | VARIABLES | 8 |
| 4.5 | TIPOS DE DATO | 9 |
| 4.5.1. | NÚMEROS ENTEROS | 9 |
| 4.5.2. | NÚMEROS REALES..... | 10 |
| 4.5.3. | CARACTERES | 10 |
| 4.6 | SENTENCIAS | 10 |
| 4.7 | REGLAS DE PUNTUACIÓN | 10 |
| 5 | DECLARACIÓN DE VARIABLES | 11 |
| 6 | FUNCIONES DE ENTRADA / SALIDA | 12 |
| 6.1 | SALIDA - FUNCIÓN PRINTF ():..... | 12 |
| 6.1.1 | CADENA DE CONTROL..... | 12 |
| 6.1.1.1 | LISTA DE ARGUMENTOS:..... | 14 |
| 6.2 | ENTRADA - FUNCIÓN SCANF () | 14 |
| 7 | SENTENCIAS DE ASIGNACIÓN Y ARITMÉTICAS | 16 |
| 7.1 | COMBINACIÓN DE OPERADORES / OPERADORES ESPECIALES | 16 |
| 7.2 | EVALUACIÓN DE LAS SENTENCIAS DE ASIGNACIÓN | 17 |
| 7.3 | PAUSA PARA MOSTRAR LOS RESULTADOS | 17 |
| 8 | PROGRAMA DE EJEMPLO | 17 |

UNIDAD 3 - *El lenguaje de programación C*

OBJETIVOS: Conocer el lenguaje de programación “C”. Confeccionar programas en C con sentencias de entrada, de salida, de asignación, y utilizar “funciones de biblioteca”. Utilizar distintos tipos de datos simples.

1 *Un poco de Historia.*

El lenguaje C fue desarrollado por Dennis Ritchie en 1972, cuando trabajaba en los laboratorios Bell, junto a Ken Thompson en el diseño del sistema operativo UNIX. Es la evolución de dos lenguajes anteriores, el primero fue el BCPL (Basic Compiled Programming Language – Lenguaje Básico de programación compilado), luego como una simplificación del lenguaje BCPL adaptado a microcomputadoras (que son las computadoras actuales) crearon el lenguaje B. Por último, el lenguaje C fue la evolución del B y de allí provienen su nombre.

A raíz de la creciente popularidad de las microcomputadoras, comenzaron a surgir numerosas implementaciones de C que diferían en parte de la definición de lenguaje original, creando pequeñas incompatibilidades y disminuyendo la portabilidad del lenguaje. La portabilidad es la capacidad de ejecutar un mismo programa en distintas computadoras (con distinto hardware) y/o distintos entornos (por ejemplo, distintos sistemas operativos). Esto hizo necesaria la búsqueda de un C estándar que fue definido por el Instituto Nacional Americano de Estándares (ANSI) en el año 1983. Por lo tanto, cuando se habla de la versión estandarizada del lenguaje se hace referencia al ANSI C.

En 1977 Kernighan y Ritchie publicaron una descripción definitiva del lenguaje “El manual de referencia del lenguaje C”, que es una fuente de consulta fundamental para todo programador en este lenguaje.

Ha mediados de los 80, Bjarne Stroustrup, también en los laboratorios Bell, crea el lenguaje C++ que soporta el paradigma de programación orientada a objetos.

2 *Desde el código fuente hasta el ejecutable*

Como se mencionó en la unidad 1, un programa es una secuencia de instrucciones que se escriben en un lenguaje de programación que el programador entiende, pero no la computadora, por lo tanto, necesita de un proceso de traducción. El lenguaje C, es un lenguaje compilado, por lo tanto, todo el código será analizado y traducido por completo hasta obtener un programa ejecutable. Para llegar desde el código fuente hasta el programa que se puede ejecutar en una computadora hay una serie de pasos que se deben llevar a cabo. Esos pasos pueden verse en la figura 1.



PASOS PARA GENERAR EL EJECUTABLE

Figura 1: pasos para generar un programa ejecutable partiendo del código fuente

Inicialmente se escribe el programa en el lenguaje C generando un programa fuente. El programa fuente no es más que un archivo de texto escrito respetando las sentencias del lenguaje.

Dentro del código fuente, el programador puede utilizar partes de código ya desarrolladas, de forma que las cosas repetitivas no tengan que escribirlas en cada programa. Por ejemplo, para leer datos del teclado o para mostrar mensaje por pantalla no hay instrucciones del lenguaje que puedan hacerlo en forma directa, pero, sin embargo, esa funcionalidad es parte de lo que se llaman bibliotecas de funciones. Una función es una secuencia de instrucciones que realiza una tarea específica. Las funciones se identifican por un nombre, mediante el cual desde el programa principal se “invoca” a dicha función. Invocar una función significa que el programa que se viene ejecutando en forma secuencial hace un salto, y comienza a ejecutar las instrucciones que contiene la función, cuando llega al final de la función vuelve a la línea del programa que la invoco. A las funciones se les puede pasar datos y pueden retornar un único valor. Por ejemplo, a una función que permite leer un valor de teclado se le indica en qué variable va a guardar ese dato leído. A la función para mostrar un mensaje en pantalla se le envía el mensaje a mostrar, de esta forma las funciones se pueden reutilizar y adaptar a distintos programas con los datos que reciben.

Al escribir un programa si se quieren utilizar funciones ya desarrolladas se le debe avisar al compilador donde está la función que queremos utilizar, es decir, si se necesita utilizar, por ejemplo, las funciones para leer datos de teclado se le debe indicar al compilador en qué biblioteca se encuentra dicha función para que la incorpore a nuestro código fuente y de ese modo podamos utilizarla. Para ello, en el código fuente se agrega una línea de código por cada una de las bibliotecas que se necesiten incluir. Las bibliotecas provistas por el entorno de programación están optimizadas de forma que cuando se incluyen el código fuente solo se incluye lo que se conoce como cabecera de las funciones. La cabecera de una función es una línea que indica el nombre de la función, qué datos se le puede enviar, y si retorna o no un valor. El preprocesador es el encargado de agregar esas cabeceras de las funciones de las bibliotecas que el programador indica en el código fuente. Además, el preprocesador modifica el código fuente escrito originalmente reemplazando las constantes definidas por su valor correspondiente. Es decir, que el pre-procesador es el que genera el archivo fuente final a compilar.

El compilador es el encargado de tomar el programa fuente, chequear que cumpla con la gramática del lenguaje C y si no hay errores generar el programa objeto, que es el programa en un formato que

la computadora puede comprender. Si hay errores se informan al usuario indicando el número de línea donde se encuentra.

El programa objeto si bien ya está en un formato entendible por la computadora, aún no puede ejecutarse porque está incompleto, falta el código objeto de las funciones de biblioteca que se utilizan. Es el linker el que toma el código de dichas funciones y lo junta con el código de nuestro **programa creando finalmente el programa ejecutable completo. Las funciones de biblioteca** están optimizadas de forma que el linker solo incorpora el código de las funciones que realmente se utilizan en el programa y NO el código de todas las funciones de la biblioteca a la cual pertenece.

3 Estructura general de un programa en C

Todo programa en C está estructurado en funciones. El lenguaje C tiene una función principal, que es por la cual comenzará la ejecución del programa. Dicha función se llama main, no pudiendo faltar, ya que a ésta le transfiere el control el sistema operativo al ejecutarse un programa.

Un programa en C se estructura según la imagen de la figura 2.

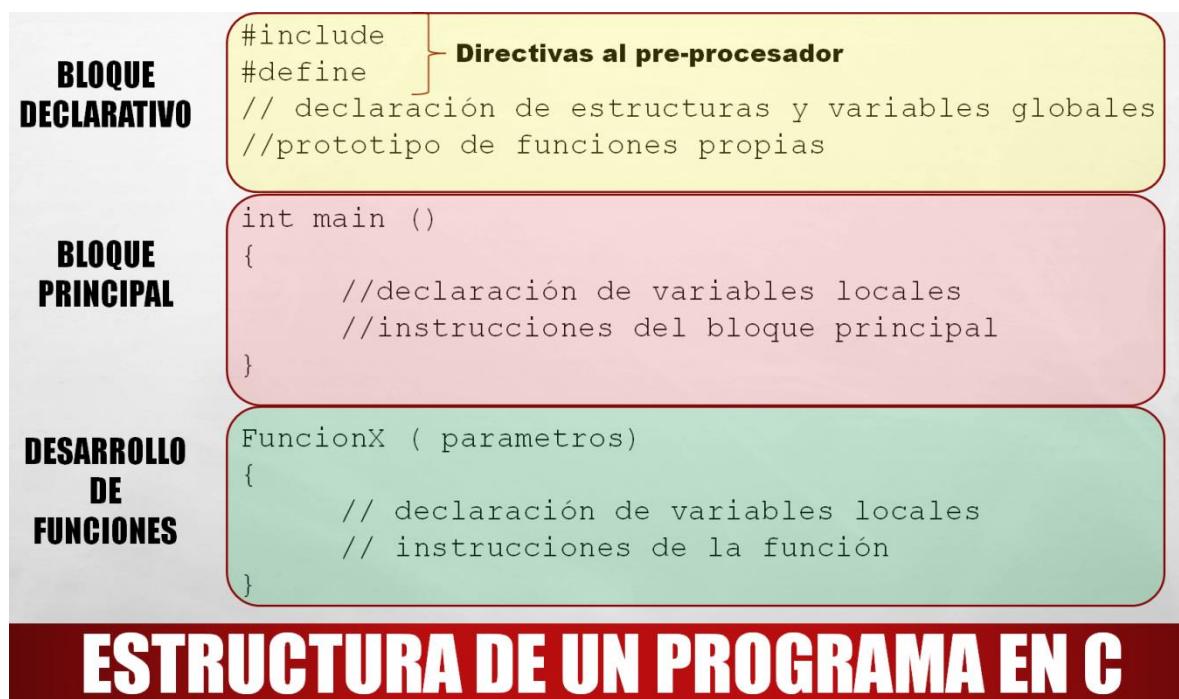


Figura 2: Estructura de un programa en C

Se analiza a continuación el contenido de los bloques enunciados anteriormente.

3.1 Bloque declarativo general

En este bloque se incorporan las directivas al preprocesador, la definición de variables globales, estructuras y prototipos de funciones propias.

3.1.1 Directivas al preprocesador

Todos los comandos al preprocesador comienzan con el símbolo #.

Si bien existen otros comandos se utilizarán solo dos: include y define.

3.1.1.1 Include

Agrega al archivo fuente el contenido de otro archivo. Se utiliza para incorporar el código de las bibliotecas al código fuente.

Para incluir bibliotecas estándar el nombre de la biblioteca se escribe entre los símbolos menor y mayor. Estas bibliotecas las busca dentro de la carpeta llamada include dentro de la estructura de directorios del compilador. Recuerde que las bibliotecas estándar solo tienen los prototipos de la función también llamados cabeceras, por lo tanto, estos archivos tienen la extensión .h que viene de header, cabecera en inglés.

La forma general es:

```
#include <nombrebiblioteca.h>
```

Existen muchas bibliotecas que se pueden utilizar algún de ellas son:

- stdio.h para operaciones de entrada y salida.
- string.h para operaciones con cadenas de caracteres.
- math.h declara funciones matemáticas.
- stdlib.h declara funciones para conversiones numéricas, asignar memoria y otras funciones.
- time.h funciones para manipulación de fecha y hora.
- conio.h funciones de pantalla.
- graph.h incluye funciones para gráficos.
- dos.h incluye las funciones del sistema operativo DOS.
- ctype.h funciones para operar con caracteres.

También puede incluirse cualquier otro archivo que no sea una biblioteca estándar, lo que nos dará la posibilidad de crear nuestras propias bibliotecas.

Para incluir cualquier otro archivo en lugar de escribir su nombre entre menor y mayor, se utilizan comillas. Dentro de las comillas se puede escribir una ruta completa o directamente el nombre del archivo. Si solo se pone el nombre del archivo lo buscará en la misma carpeta donde se encuentra el código fuente.

Por ejemplo, #include "misFunciones.c". Incluye todo el código de archivo misFunciones.c en el programa fuente. Dicho archivo DEBE estar en la misma carpeta que archivo fuente sobre el que estamos trabajando sino no lo encontrará.

En cambio, #include "c:\\funciones\\misFunciones.c" buscará el archivo a incluir en el disco c dentro de la carpeta funciones. Observe que se utilizan dos barras en lugar de una, esto es porque la barra es un carácter llamado de escape utilizado para realizar funciones especiales dentro del lenguaje, esto se verá a continuación.

Siempre es recomendable para mayor portabilidad utilizar la primera opción y poner los archivos a incluir en la misma carpeta que el código original.

3.1.1.2 Define

Permite definir un texto a reemplazar dentro del programa. Se utiliza para definir constantes y macros.

Forma general es:

```
#define nombre-simbólico    texto de reemplazo
```

A partir de esta instrucción cualquier aparición del nombre-simbólico, dentro del programa, será reemplazada por el texto de reemplazo. Permite definir constantes globales ya que la etiqueta será buscada y reemplazada en todo el programa, siempre teniendo en cuenta que se trata de un reemplazo de texto, es decir, actúa como el comando buscar y reemplazar de cualquier editor de texto. El programa a compilar será el resultante luego de realizar los reemplazos.

Ejemplos:

```
#define MAXIMO 100  
  
#define PI 3.14159  
  
#define TITULO "RESUMEN"
```

También puede definirse lo que se conoce como macros haciendo que la etiqueta sea reemplazada por varias instrucciones incluso pudiendo enviarle datos para hacerlo más genérico. Las macros no se utilizarán en esta materia.

3.1.2 Definición de variables globales y estructuras

Al escribir un programa, para guardar los datos con los que trabajará, necesita variables. Dentro de los lenguajes de programación, las variables tienen lo que se conoce como “ámbito” de vigencia de la variable. El ámbito indica desde que parte de nuestro programa podemos acceder a esa variable. En general, cada bloque o función del programa define sus propias variables, lo que se conoce como variables locales. Si se necesita una variable que pueda ser accedida desde cualquier parte del programa se deberá declarar en forma global, antes del bloque principal. No es recomendable la utilización de variables globales ya que fácilmente puede perderse el control de donde fue modificada, y además, hace mucho más difícil que el código realizado pueda reutilizarse en otros programas.

Más adelante se verá el tema de estructuras de datos, en general una estructura deberá ser accedida desde todo nuestro programa por lo que su declaración será en el ámbito global.

3.1.3 Prototipos de funciones propias

Además de poder utilizar funciones de bibliotecas ya desarrolladas, más adelante se construirán funciones propias. En el bloque declarativo se incluyen lo que se conoce como prototipo de funciones, que es la especificación del nombre de la función, que datos recibe y que retorna. El código con las instrucciones de dichas funciones se pondrá debajo del bloque principal.

El prototipo de funciones se pone para que el programa quede más ordenado y poder encontrar rápidamente el bloque principal de nuestro programa. También es posible no poner los prototipos y escribir la función completa antes del programa principal, pero esto haría más difícil ubicarse dentro del código fuente. Si dentro de una función se llama a otra función, debe ponerse primero el prototipo de la que es llamada, y luego, el prototipo de la función que la llamó.

3.2 Bloque Principal

Es el inicio del programa. Como el lenguaje C está estructurado en funciones, esta es la función principal que siempre debe estar presente y es lo primero que se ejecuta. El formato general es el siguiente:

```
int main ()  
{  
  
    return 0;  
}
```

Donde main es el nombre de la función (main significa principal en inglés). La palabra int significa que esta función retorna un número entero. Todo programa debe retornar al sistema operativo un número entero que indica si la ejecución fue exitosa o no, retornar un 0 significará que no hubo errores y cualquier otro valor servirá para indicar error. Los paréntesis vacíos indican que la función no recibe datos, en materias siguientes se trabajara con datos enviados al programa al iniciar el mismo. También para indicar que no recibe datos en lugar de los paréntesis vacíos puede ponerse dentro la palabra void, que es el tipo de dato vacío, indicando que no recibe nada.

Las llaves forman el bloque de la función principal, dentro de las llaves se escribirán todas las instrucciones del programa, comenzando siempre por la declaración de variables locales que se verán en la sección siguiente.

Antes de cerrar la llave del bloque del programa principal se pone la palabra return y en este caso se retorna un 0 indicando al sistema operativo que el programa finalizó correctamente.

3.3 Desarrollo de Funciones Propias

Si se utilizan funciones propias y en el bloque declarativo se incluyó el prototipo de las mismas, debajo de la función principal se escribe el código completo de dichas funciones. Cada una de ellas tendrá su nombre, datos que recibe, valor de retorno y su bloque definido entre llaves. Nuevamente dentro de cada función al comienzo del bloque se definen las variables que se utilizan localmente en la misma, y luego, las instrucciones para realizar la funcionalidad deseada.

4 Elementos del lenguaje

Todo lenguaje de programación está compuesto por una serie de elementos que se detallan a continuación y son los elementos básicos que componen el C:

1. Alfabeto
2. Identificadores
3. Constantes
4. Variables
5. Tipos de datos
 - números enteros o de punto fijo
 - números reales ó de punto flotante
 - caracteres – cadenas de caracteres – string
 - modificadores de tipo
6. Sentencias
7. Reglas de puntuación

4.1 Alfabeto

Como en cualquier lenguaje este consta de:

- Letras mayúsculas y minúsculas (son distintas MAYUSCULAS y minúsculas)
- Dígitos del 0 al 9
- Caracteres especiales: () {} ; : " ' , . & * % #
- Palabras reservadas: main, for, while, include, If, switch, auto, etc. (estas palabras se llaman reservadas ya que no pueden ser utilizadas como identificadores)
- Operadores (aritméticos, de relación, lógicos, de incremento, de asignación, etc.). Ej.: + *, >, <, &&, ++, =

4.2 Identificadores

Se llama así, a los nombres que se utilizan para nombrar a las constantes, variables de cualquier tipo, y a las funciones.

Puede ser una sucesión de letras, dígitos y algunos caracteres especiales, comenzando siempre con letra. Se admite como letra al carácter _ (guion bajo). No hay límites en cuanto a cantidad máxima, como mínimo una letra.

Ejemplos:

a A suma x33 G lote superficie AutoLuz k_k23

4.3 Constantes

Son aquellos valores que no se alteran durante la ejecución del programa. Hay dos tipos de constantes, según su contenido, numéricas (pueden escribirse en sistema decimal, octal y hexadecimal) y NO numéricas. Por ejemplo:

- Numéricas: 56, 5, 8.90E12, -1.25, 3.1415
- No numéricas: '\$', "mts", "Grados", '5', "34-35"

Las constantes pueden utilizarse como parte de operaciones, por ejemplo, si a la variable suma se le quiere sumar diez entonces se puede escribir:

```
suma = suma +10;
```

En este caso 10 es una constante numérica entera.

También pueden definirse constantes con un identificador de dos formas:

- En el bloque declarativo usando el define, visto anteriormente.
- Anteponiendo al tipo, en la declaración de variables, la palabra const.

4.4 Variables

Se define una variable, como la representación simbólica de un lugar de la memoria, donde se guarda un valor, que es el valor de la variable. Su nombre es un identificador, cuyo valor se puede alterar durante la ejecución de un programa.

A cada variable se le debe asignar un tipo de datos e inclusive, se le puede asignar un valor inicial, como se verá luego.

Debemos tener clara la diferencia entre letras minúsculas y mayúsculas, ya que se asumen como caracteres distintos en el lenguaje C.

4.5 Tipos de dato

Una de las características más importantes del lenguaje C es su capacidad para operar con una gran diversidad de tipos de datos, admitiendo la creación por parte del usuario de nuevos tipos.

Los algoritmos operan sobre datos, los cuales pueden ser de distinto tipo y se caracterizan por:

- Un rango de valores posibles
- Un conjunto de operaciones realizables sobre dicho tipo de datos
- Su representación interna.

Significa que al asumir un tipo de dato se indica la clase de valores que pueden tomar las variables, y las operaciones que se pueden hacer sobre ellos.

Por ahora se analizarán los tipos de datos “simples”, que son aquellos donde una variable ocupa “un lugar de memoria” y contiene un único valor.

Los tipos de datos simples más comunes se muestran en la tabla 1.

Tabla 1: Tipos de dato simples

| Tipo | Nombre | Espacio que ocupa | Rango |
|---------------------|-----------|-------------------|--|
| Entero corto | short int | 2 bytes | -32768 a 32767 |
| Entero | int | 4 bytes | -2.147.483.648 a 2.147.483.647 |
| Carácter | char | 1 byte | -128 a 127 |
| Real | float | 4 bytes | 1.7 10 ⁻³⁸ a 1.7 10 ³⁸ (7 díg.) |
| Real doble | double | 8 bytes | 1.7 10 ⁻³⁰⁸ a 1.7 10 ³⁰⁸ (16 díg.) |

Si no se necesitan utilizar números negativos, al tipo de dato sin decimales, puede anteponerse el modificador unsigned de forma que el rango ya no se divide entre positivos y negativos sino que toma solo números positivos quedando los rangos según la tabla 2.

Tabla 2: Modificador unsigned

| Tipo | Nombre | Espacio que ocupa | Rango |
|-------------------------------|--------------------|-------------------|-------------------|
| Entero corto sin signo | unsigned short int | 2 bytes | 0 a 65535 |
| Entero sin signo | unsigned int | 4 bytes | 0 a 4.294.967.295 |
| Carácter sin signo | unsigned char | 1 byte | 0 a 255 |

El lenguaje también dispone de tipos de datos ESTRUCTURADOS, que como su nombre lo indica, cada variable puede tener una adecuada estructura conteniendo una serie de valores de igual ó distinto tipo.

Ejemplos: cadenas, arrays, estructuras, etc.

4.5.1. Números enteros

Son aquellos que solo contienen dígitos, precedidos o no por el signo. No tienen exponente (E) ni punto decimal.

Ejemplos:

- En decimal: 0 32767 -12345 231 89
- En octal: Se antepone al nro un 0 (cero). Ej. 071 en octal es igual al 57 en decimal
- En hexadecimal: Se antepone al nro. 0X. Ej.: 0x100 en hexadecimal es igual al 256 en decimal

4.5.2. Números reales

Pueden escribirse de forma diferente, incluyendo dígitos, “un punto decimal”, y opcionalmente, pueden incluir un signo y/o un exponente.

Ejemplos:

12.3 -9.0009 3.0E12 4E-15 -2.5E-12 0.235

4.5.3. Caracteres

Las variables del tipo char pueden guardar un solo símbolo, un carácter. Pero en realidad internamente guarda un número que corresponde al código ASCII (sigla en inglés de American Standard Code for Information Interchange - Código Americano Estándar para el Intercambio de Información). Por ejemplo, el código ASCII de la letra A es el 65 el de la B el 66, y así sucesivamente. El código de las letras minúsculas comienza en el 97 para la letra a.

Por lo tanto, una variable del tipo char se puede utilizar para guardar un carácter o números pequeños, dependerá luego de cómo mostremos dicha información.

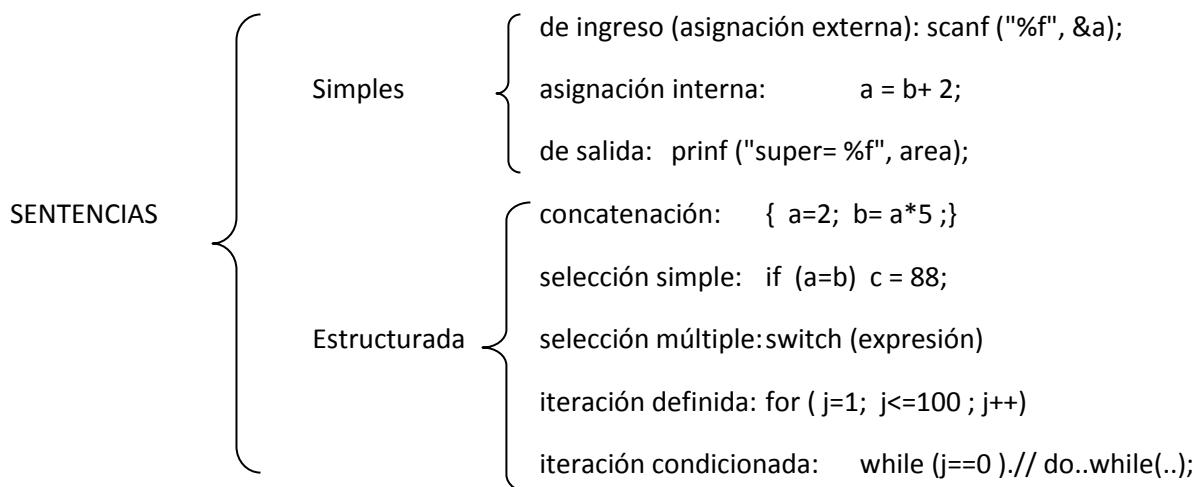
Las constantes del tipo char se escriben entre apostrofes (o comillas simples)

Ejemplos: 's' 'v' '*' '@' '9'

Es importante destacar que las letras tienen números correlativos lo que servirá más adelante para realizar validación de los datos ingresados.

4.6 Sentencias

Una sentencia en C puede ser una sola instrucción ó un grupo de instrucciones encerradas entre llaves. Hay dos tipos de sentencias, las simples y las estructuradas. Más adelante se verá en detalle cada una de ellas.



4.7 Reglas de puntuación

A medida que se avance en el desarrollo de los temas se verá puntualmente la forma correcta de escribir cada una de las sentencias y bloques del lenguaje. Como regla general debe tener en cuenta que:

- Las instrucciones finalizan con; (punto y coma)
- Los bloques se delimitan entre {} (llaves)
- Debe respetarse las minúsculas y mayúsculas ya que el lenguaje es sensible, teniendo en cuenta que la mayoría de palabras reservadas se escriben todas en minúscula.
- Si dentro del código fuente quieren poner comentarios se utiliza:
 - // para los comentarios de una sola línea.
 - /* comentario */, para comentarios de varias líneas.
 - Los comentarios se utilizan para clarificar el código fuente agregando notas que expliquen lo que se está haciendo. Los comentarios no agregan peso al programa final ya que son eliminados por el pre-procesador. Solo quedan en el código fuente.

5 Declaración de variables

Todas las variables que se utilicen en el programa (main) deben ser declaradas al inicio del mismo o de cada función que las use, asignando valores iniciales, si se requieren. Esta declaración tiene por objeto asociar a cada variable un tipo de dato y efectuar una reserva de memoria.

Forma general es:

tipo de dato lista de variables;

En realidad, antes del tipo de dato en el lenguaje C se especifica donde se va a guardar dicha variable. Pero en esta materia se usa solo el almacenamiento por defecto y por lo tanto no se escribe. Es equivalente de escribir la palabra `auto` delante del tipo de dato para indicar el tipo de almacenamiento automático. Es por eso que `auto` es una palabra reservada del lenguaje.

Ejemplos:

- Para definir una única variable del tipo entera llamada n1:

```
int n1;
```

- Para definir tres variables enteras:

```
int num1, num2, suma;
```

- Para definir dos variables reales, el segundo inicializado en un valor:

```
float a, b=7.2;
```

- Para definir dos variables char, el primero inicializado en un valor:

```
char letra='s', símbolo;
```

A la declaración de tipo se le puede anteponer la palabra `const`, en cuyo caso no serán variables sino constantes y luego se asigna el correspondiente valor.

Ejemplo:

```
const float pi = 3.14159;
```

Esto significa que pi ocupa un lugar en memoria, pero su valor al ser constante se asigna por única vez y ya no puede ser modificado en todo el programa.

AMBITO DE INCUMBENCIA: Las variables según el lugar en el cual se las declare pueden ser:

- **VARIABLES GLOBALES.** Son las definidas en el bloque declarativo general y su campo de validez es todo el programa, incluyendo todas las funciones.
- **VARIABLES LOCALES.** Son las definidas dentro de cada función, incluso en el main (), y su campo de validez se limita solo a la función donde fueron definidas. Cabe destacar que es posible crear variables locales con el mismo identificador en distintas funciones, teniendo siempre en cuenta que a pesar de tener el mismo nombre son variables distintas ya que cada función define sus variables locales en áreas separadas de memoria. También dentro de una función es posible definir un bloque interno (encerrado entre llaves) pudiendo definir variables locales a ese bloque, pero pudiendo acceder a las variables del bloque de la función que lo contiene.

6 Funciones de entrada / salida

El lenguaje C no posee sentencias con capacidad de generar la entrada y/o salida de datos, por lo cual se recurre a funciones de biblioteca. Estas funciones permiten la transferencia de información entre los dispositivos de entrada y/o salida con la computadora. Se puede invocar a estas funciones desde cualquier punto del programa. Las que más se utilizarán son scanf () y printf (), que son sentencias con formato, ambas están incluidas en la biblioteca stdio.h.

6.1 SALIDA - función printf ():

Permite la transferencia de información entre la computadora y un dispositivo de salida como ser la pantalla o la impresora.

Forma general es:

```
printf ( "cadena de control", lista de argumentos );
```

6.1.1 Cadena de Control

La cadena de control: se compone de una lista de:

1. Especificaciones de formatos
2. Secuencias de escape
3. Leyendas, títulos ó mensajes aclaratorios

Con estos elementos entre comillas puedo diseñar la salida que deseo exhibir.

1) Especificaciones de formato.

Se debe indicar una especificación de formato para cada variable a informar, el cual se compone del símbolo % seguido del carácter de conversión. Indica al compilador que tipo de variable va a ser exhibida. Los códigos de formatos más utilizados son:

- % d para un número entero con signo
- % f para un número real
- % c para un carácter
- % s para una cadena de caracteres (un texto)
- % e para números reales con exponente
- % hd para entero corto (short int)
- % x para un entero en hexadecimal – imprime en hexadecimal.

Se puede formatear la exhibición, o sea, especificar “la longitud de campo deseada” para cada valor a exhibir (un valor entero luego del %). Esta longitud puede precederse con un signo -, lo cual significa alinear por izquierda, con + o sin poner nada alinea por derecha. Si tengo números reales, especifico el total de posiciones, un punto y la cantidad de cifras decimales.

Ejemplos

%4d: muestra un número entero en cuatro lugares, si el número tiene menos de cuatro deja espacios en blanco a la izquierda, si tiene más lo muestra igual completo.

% 8.3 f: muestra un número real que en total ocupa 8 espacios con 3 decimales

.%0f (sin decimales): muestra un número real sin decimales.

Los datos tipo char – como enteros

Una variable definida “tipo char” también puede ser impresa con un formato entero. Dependiendo del formato que se utilice para imprimirla, ya sea %c ó %d, se obtienen distintos resultados.

La siguiente línea de código muestra la constante de la letra a con su valor en decimal (código ASCII) y en formato carácter

```
printf ("El carácter %c tiene el valor %d ", 'a', 'a');
```

Muestra en pantalla: El carácter a tiene el valor 97

2) Secuencias de escape.

Luego de mostrarse un valor, el cursor queda después del último carácter exhibido, en la misma línea. Si quiero que vuelva al principio de la próxima línea debo colocar antes de la comilla de cierre el "carácter de escape" \n. Puede colocarse en cualquier lugar de la cadena de control. Existen otras secuencias de escape, están formadas por una \ y un carácter que puede ser una letra o una combinación de dígitos. Se usan para tabular, cambio de línea y representar caracteres no imprimibles.

| SECUENCIAS | ACCION |
|------------|--------------------------------|
| \n | nueva línea |
| \t | tabulación horizontal |
| \" | imprime comillas |
| \' | imprime un apóstrofe |
| \b | retrocede un espacio el cursor |
| \r | retorno de carro |

3) Leyendas, títulos ó mensajes aclaratorios.

Todo texto que se ponga entre las comillas de la cadena de control que no sea un formato o una secuencia de escape se verá tal cual, en pantalla, por lo que pueden escribirse mensajes para el usuario. Hay que aclarar que los caracteres especiales del español como la letra eñe y acentos no se visualizan correctamente.

6.1.1 Lista de argumentos:

Es una lista de expresiones, constantes y variables, cuyos valores se van a imprimir, separadas por comas y en concordancia con los formatos especificados en la cadena de control.

Ejemplo 1:

```
int codi=7 ; float temp= -31.42;  
printf ("\n El codigo es %4d y la temperatura es = %7.2f \n", codi, temp);
```

Comienza cambiando el cursor de línea y luego imprime:

El código es 7 y la temperatura es = -31.42

Luego el cursor baja una línea.

Ejemplo 2

```
printf ("31+21 es igual a: %d cms", 21+31);
```

Imprime: 31+21 es igual a: 52 cms

Ejemplo 3:

```
printf("chau %c seis = %d", '@', 6 );
```

Imprime: chau @ seis = 6

6.2 ENTRADA - función scanf ()

Se utiliza para el ingreso de información a la computadora, a través, de las variables indicadas. Es la encargada de leer los datos que se ingresan desde el teclado. Al encontrar esta función, la computadora queda a la espera del ingreso de uno ó más valores desde el teclado, luego de ingresar el último dato, debo pulsar “enter”. Si hay más de un dato a digitar, se pueden separar con espacio, una tabulación o con enter, pero siempre al final se debe presionar enter.

Forma general:

```
scanf ("formatos", lista de argumentos);
```

Donde el primer parámetro se compone de una lista de especificaciones de formatos entre comillas. Se debe indicar una especificación de formato para cada variable que se quiera leer. Son los mismos que se utilizan para el printf(...).

Es importante no dejar espacios ni poner otros caracteres entre los formatos para las variables para evitar problemas al leerlos.

La lista de argumentos está formada por la dirección de memoria de las variables que se desean leer. Para obtener la dirección de memoria de una variable se utiliza el operador &. Este operador indica el lugar de memoria donde será enviado el valor ingresado por teclado y registrado por la función, mediante el nombre del identificador. Se debe respetar el orden de los formatos según las direcciones de memoria que se especifiquen en el argumento, el primer formato corresponderá con la primera dirección de memoria, el segundo con la segunda y así sucesivamente.

Ejemplo1: Leer un entero:

```
int a;  
  
scanf ("%d", &a);
```

Ejemplo 2: Leer dos enteros:

```
int n1, n2;  
  
scanf ("%d%d", &n1, &n2);
```

Ejemplo 3: Leer un real y un entero:

```
int num;  
  
float f;  
  
scanf ("%f%d", &f, &num);
```

Todos los datos al ser ingresados por teclado se almacenan temporalmente en un buffer de teclado, que es una posición de memoria intermedia que guarda todas las teclas presionadas. Luego, la función scanf toma los datos de ese buffer y los pasa a la dirección de memoria indicada según el formato establecido. Este procedimiento funciona perfectamente para variables numéricas pero el scanf tiene un problema al leer caracteres. El formato %c lee un solo carácter, por lo tanto, lo va a buscar al buffer de teclado, pero el buffer muchas veces no está vacío ya que si antes se leyó otro valor el código de la tecla enter que se presiona para confirmar los datos queda en el buffer ya que no es parte del dato ingresado. Este código de la tecla enter es tomado automáticamente por el scanf, y, por lo tanto, no permite ingresar el dato. Entonces para poder leer un carácter de teclado si antes se leyó otro dato hay que limpiar el buffer de teclado. Para ello hay dos formas:

- Usando la función fflush(stdin): esta función limpia el buffer de la entrada estándar, es decir, del teclado. El problema es que esta no es una función estándar por lo que no funciona en todos los compiladores.
- Quitar el carácter que sobra del buffer utilizando getchar (): getchar es una función que recupera un carácter del buffer, por lo tanto, se puede poner luego de leer un dato para "limpiar" ese carácter del enter que quedó en el buffer.

Ejemplo de un programa completo que lee primero un dato entero y luego una letra, limpiando el buffer en el medio:

```
#include <stdio.h>  
int main ()  
{  
    int num;  
    char letra;  
    printf ("Ingrese un numero: ");  
    scanf ("%d", &num);  
    getchar();  
    printf("Ingrese una letra: ");  
    scanf ("%c", &letra);  
    printf("Se ha ingresado el numero: %d y la letra: %c", num, letra);  
    return 0;  
}
```

7 Sentencias de asignación y aritméticas

Las sentencias secuenciales se escriben directamente tal como se explicó en la unidad 2, tanto para operaciones matemáticas como para asignaciones. La única diferencia con el diagrama es que en el código C cada instrucción al final se termina con el punto y coma.

7.1 Combinación de operadores / operadores especiales

El C lenguaje admite la simplificación de una variable en las sentencias aritméticas cuando se combinan los operadores aritméticos y el igual.

Lo común es:

variable= variable operador expresión;

Se puede reemplazar por:

variable operador= expresión;

Algunas combinaciones posibles son:

* = mult. + asignación

/ = división + asignación

% = resto + asignación

+ = suma + asignación

- = resta + asignación

Ejemplos:

x = x + 3; se puede escribir: x += 3 (suma y asignación)

x = x * 3; se puede escribir : x *= 3 (multiplica y asigna)

El lenguaje C dispone además de dos operadores especiales el ++ y el --.

El operador ++ permite incrementar en 1 el valor de una variable, y el operador—decrementar en 1 el valor de una variable.

Por ejemplo, a= a +1; puede escribirse como a++; pero también, puede escribirse como ++a; o incluso usando la combinación de operadores como a +=1;

El operador ++ o -- puede usarse tanto antes como después de la variable. Si se la aplica solo a la variable tiene el mismo efecto pero en una asignación el efecto es diferente. Por ejemplo, el siguiente programa:

```
#include <stdio.h>
int main()
{
    int n1=5,n2;
    n2=n1++;
    printf("n1:%d  n2:%d", n1,n2);
}
```

Mostrará: n1: 6 n2: 5

Ya que el operador `++` se toma como un post-incremento y se aplica luego de asignar el valor de `n1` a `n2`. En cambio, si esa línea se reemplaza por `n2=++n1;`

Mostrará: `n1: 6 n2: 6`

Ya que el operador `++` se toma como un pre-incremento y se aplica antes de asignar el valor a `n2`.

Esto mismo aplica para el operador `--`

7.2 Evaluación de las sentencias de asignación

Se realiza en tres pasos:

- Se evalúa la expresión, o sea, se la resuelve obteniendo un resultado.
- Si es posible, adecua el tipo de dato del resultado de la expresión al tipo de dato de la variable de la izquierda. Si no es posible puede o no indicar error y asignar cualquier valor (entero=real: asigna parte entera).
- Asigna el resultado de la expresión a la variable de la izquierda.

PRECAUCIONES:

No colocar dos operadores consecutivos, salvo los permitidos, usar paréntesis si es necesario o existen dudas sobre la precedencia de los operadores.

La división por cero es una operación prohibida, cancela el programa.

7.3 Pausa para mostrar los resultados

Los programas que se realizarán son programas de consola. Si se trabaja con el sistema operativo Windows al ejecutar desde la interfaz gráfica un programa de consola (haciendo doble click al ejecutable) el mismo se ejecuta, pero al finalizar automáticamente se cierra y no permite ver los resultados. Por lo tanto, es necesario antes de finalizar el programa poner una pausa para permitir al usuario visualizar los resultados. Para ello podemos utilizar:

- La función `getch()` disponible en la biblioteca `conio.h`, esta función es similar al `getchar`, pero no muestra el carácter presionado en pantalla. El problema es que `getch()` no es una función estándar, por lo tanto, no funciona con todos los compiladores.
- Hacer una llamada al sistema operativo e invocar la pausa. En Windows la instrucción sería: `system("pause");` y para poder usar la función `system` debe incluirse la biblioteca `stdlib.h`.
- Poner un mensaje propio personalizado y esperar a que se presione una tecla cualquiera (usando `getchar` o `scanf`) o una tecla particular validando el ingreso de datos utilizando sentencias repetitivas que se verán en unidades siguientes.

8 Programa de ejemplo

Siempre existen diversas formas para realizar un programa. Puede cambiar desde el nombre que se les den a las variables hasta las instrucciones que se utilicen, por lo tanto, es posible que nuestros programas nunca queden exactamente iguales al de un compañero o al del profesor, pero aún así siguen estando correctos. Vea un sencillo ejemplo y distintas formas de resolverlo:

Programa a realizar: Ingresar dos números enteros, calcular y mostrar la suma:

Primera forma: con dos scanf y una variable para el resultado

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n1,n2, suma;
    printf("Ingrese el primer numero a sumar:");
    scanf("%d", &n1);
    printf("Ingrese el segundo numero a sumar:");
    scanf("%d", &n2);
    suma = n1+n2;
    printf ("El resultado de la suma es: %d\n", suma);
    system("pause");
    return 0;
}
```

Segunda forma: con un solo scanf y una variable para el resultado

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n1,n2, suma;
    printf("Ingrese dos numeros a sumar:");
    scanf("%d%d", &n1, &n2);
    suma = n1+n2;
    printf ("El resultado de la suma es: %d\n", suma);
    system("pause");
    return 0;
}
```

Tercera forma: con un solo scanf y sin variable de resultado:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n1,n2;
    printf("Ingrese dos numeros a sumar:");
    scanf("%d%d", &n1, &n2);
    printf ("El resultado de la suma es: %d\n", n1+n2);
    system("pause");
    return 0;
}
```



Elementos de Programación

UNIDAD 4. ESTRUCTURA DE SELECCIÓN SIMPLE / MULTIPLE

INDICE

| | | |
|----------|--|-----------|
| 1 | ESTRUCTURA DE SELECCIÓN SIMPLE (SENTENCIA IF) | 2 |
| 2 | CODIFICACIÓN EN C | 7 |
| 3 | OPERADOR CONDICIONAL (?:) [IF REDUCIDO]..... | 8 |
| 4 | CASO PARTICULAR DE LA SENTENCIA IF | 9 |
| 5 | OPERADORES LÓGICOS | 11 |
| 6 | ESTRUCTURA DE SELECCIÓN MÚLTIPLE (SWITCH)..... | 15 |

UNIDAD 4 - Estructura selección simple / Múltiple

OBJETIVOS: Conocer en detalle y confeccionar programas utilizando las estructuras de selección simple y múltiple. Elaborar juegos de prueba que abarquen las distintas variantes de los programas. Conocer y utilizar los operadores lógicos y de comparación.

1 Estructura de Selección Simple (Sentencia if)

Esta estructura permite realizar la selección entre dos posibles cursos de acción, en base a la verdad o falsedad de una expresión que se escribe en forma de "expresión lógica".

Para escribir una expresión lógica es necesario utilizar "operadores de comparación". Estos operadores permiten evaluar una condición entre dos partes, ya sean dos variables, o una variable y una constante. Los operadores de comparación son los que se muestran en la tabla 1.

Tabla 1: Operadores de comparación

| Operador | Símbolo |
|---------------|---------|
| Mayor | > |
| Menor | < |
| Mayor o igual | >= |
| Menor o igual | <= |
| Igual | == |
| Distinto | != |

La representación gráfica de la estructura de selección simple puede verse en la Figura 1.

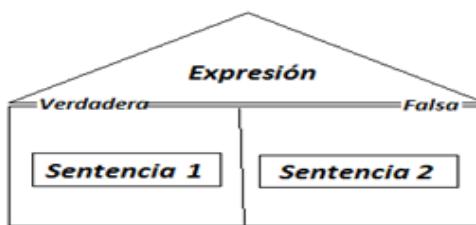


Figura 1: Representación gráfica de la estructura de selección simple

Por ejemplo, dada una variable A numérica, se puede escribir como expresión $A > 6$ en cuyo caso será verdadera o falsa según el resultado de la evaluación del contenido ingresado en A. Si es verdadera, se ejecuta la "Sentencia 1", y si es falsa se ejecuta la "Sentencia 2". Dentro del verdadero y/o falso puede haber una o más sentencias.

El lenguaje C admite también que se utilice una expresión aritmética, tomando el resultado de la evaluación como "falso" si es igual a cero y verdadero por distinto de cero.

Ejemplo: si se evalúa $5-6+3$ esta expresión devuelve un valor distinto de cero, por lo cual, continua la secuencia por verdadero. En cambio, al evaluar $4+3-7$ esta expresión devuelve como resultado un valor igual a cero, por lo que la secuencia continua por el lado falso.

Se puede utilizar, si es que lo desea, solo la parte verdadera, o sea que las acciones por 'falso' son opcionales. Las expresiones que se utilizarán al principio serán simples, es decir, una relación entre dos valores y luego se ampliarán utilizando los operadores lógicos.

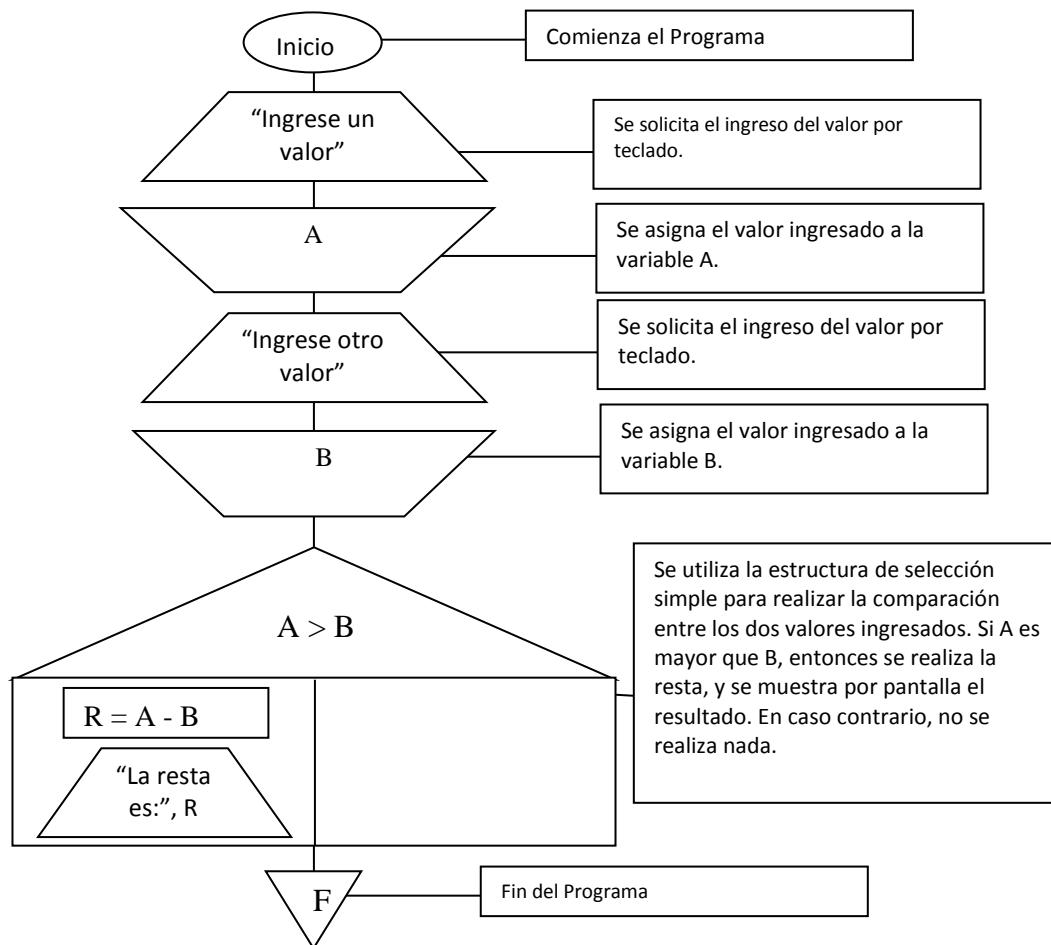
Estas estructuras se pueden enlazar en forma 'descendente', es decir, que dentro del verdadero y/o falso puede haber otras estructuras condicionales. Esto se verá en otros ejercicios más adelante. Esta estructura tiene un solo punto de entrada y uno solo de salida cualquiera fuese la alternativa ejecutada.

Ejemplo 1: Ingresar dos valores reales en dos variables, que se llamaran A y B. Si A es mayor a B calcular e informar la diferencia entre A menos B.

ESTRATEGIA

1. Ingresar los dos valores en A y B
2. Si A es mayor que B calcular la diferencia
3. Informar la diferencia calculada si se cumplió la condición del punto 2.

DIAGRAMA:

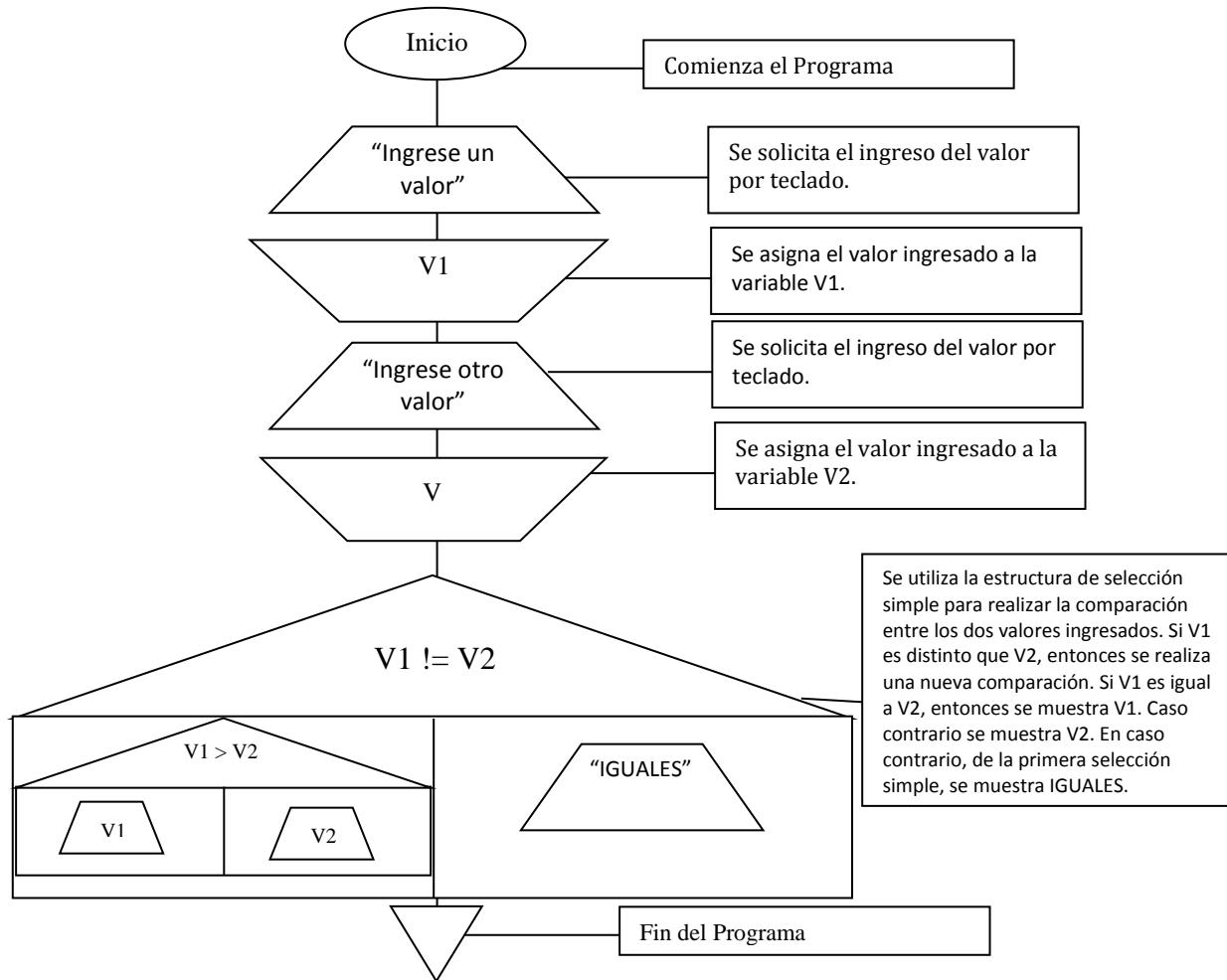


Ejemplo 2: Confeccionar un programa que ingrese dos valores numéricos y determine e informe al mayor de ellos si son distintos, o un mensaje que diga IGUALES en caso de serlo.

ESTRATEGIA

- 1] Ingresar los dos valores
- 2] Si no son iguales, determino el mayor
- 3] Informo el mayor o el mensaje IGUALES

DIAGRAMA :

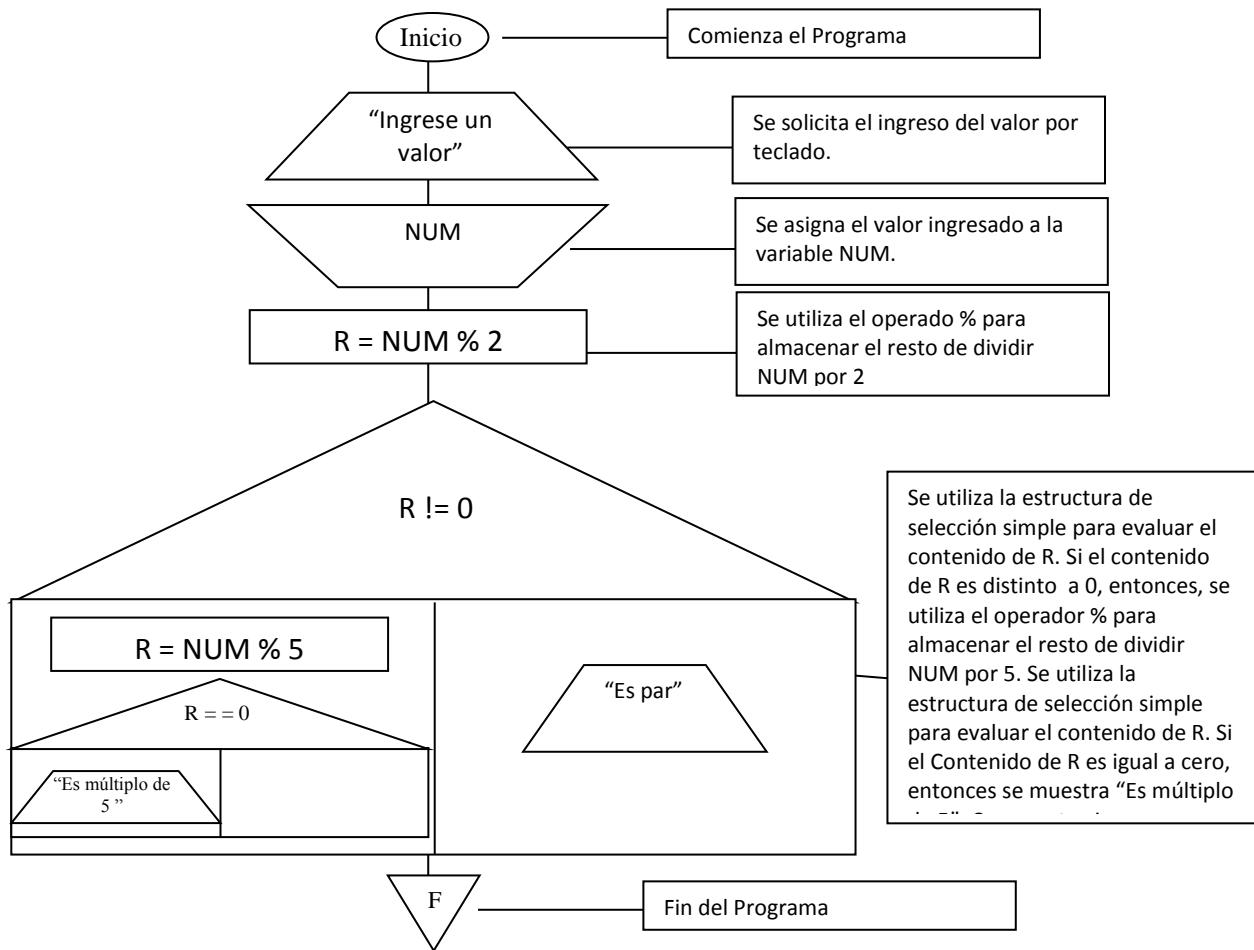


Ejemplo 3: Confeccionar un programa que pueda recibir un valor entero y nos informe si es un valor PAR, en caso de no serlo determinar e informar si dicho valor es múltiplo de 5.

ESTRATEGIA

1. Ingresar valor
2. Realizar operación para saber si es par, usando el operador % que almacena el resto.
3. Si R es igual a 0 informa "PAR"
4. De lo contrario, realiza operación para saber si es múltiplo de 5. Utilizando %
5. Si R es igual a 0 informa "MULT 5"

DIAGRAMA:



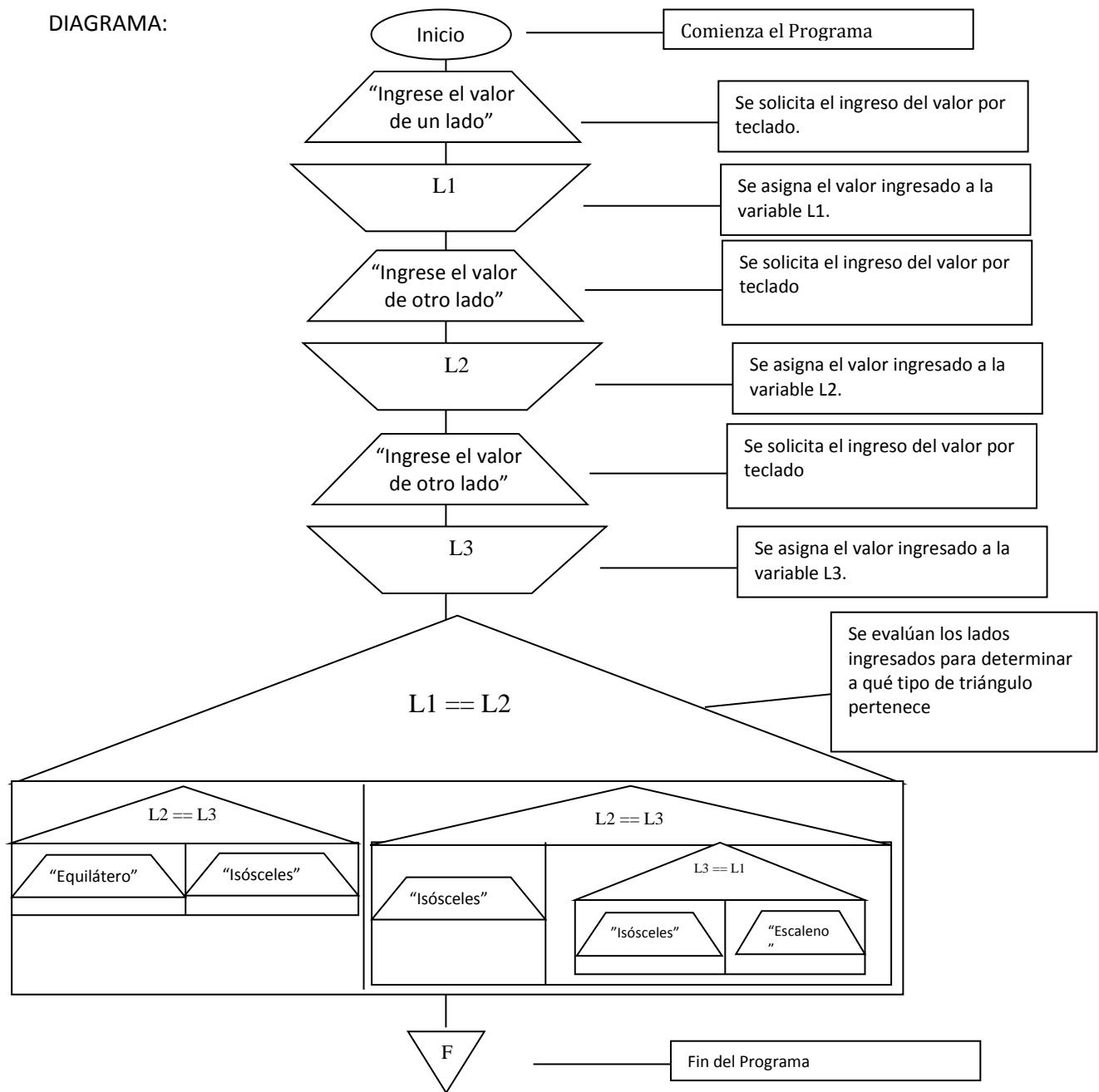
VARIANTE: ¿Cómo se debería modificar el diagrama si se quiere detectar los pares que a su vez son múltiplos de 5?

Ejemplo 4: Confeccionar un programa que ingresando los valores de los tres lados de un triángulo determine e informe si el triángulo es EQUILATERO, ISOSCELES ó ESCALENO.

ESTRATEGIA

1. Ingresar los valores de los 3 lados (L1, L2 y L3)
2. Determino si es equilátero (tres lados iguales), si no lo es, determino si es escaleno (tres lados distintos) y si o lo es, será isósceles (dos lados iguales y uno no).

DIAGRAMA:



La única forma que hay de verificar si un programa que se realiza funciona adecuadamente para las diversas alternativas que se pueden presentar, es confeccionando un JUEGO o LOTE de PRUEBA. Este consiste en una serie de datos con los cuales se ejecuta el programa efectuando un seguimiento de las sentencias tal cual lo efectuaría la computadora y se analizan los resultados obtenidos. Es importante que al diseñar un lote de prueba se tenga en cuenta todos los casos posibles y asegurarse que se ejecuten todos los caminos posibles del programa. La tabla 2 muestra un posible lote de prueba para el ejemplo anterior.

Tabla 2: Posible lote de prueba para determinar el tipo de triángulo

| L1 | L2 | L3 | Resultado esperado |
|----|----|----|--------------------|
| 5 | 5 | 5 | Equilátero |
| 5 | 5 | 7 | Isósceles |
| 5 | 7 | 7 | Isósceles |
| 5 | 7 | 5 | Isósceles |
| 5 | 6 | 7 | Escaleno |

2 Codificación en C

La estructura de selección simple se codifica de la siguiente forma:

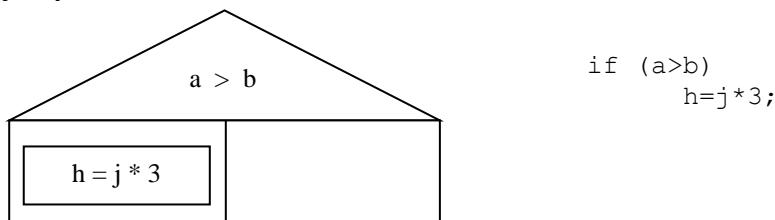
```
if ( Expresión Lógica )
{
    //instrucciones que se ejecutan si se cumple la condición es
    VERDADERA
}
else
{
    //instrucciones que se ejecutan si la condición es FALSA
}
```

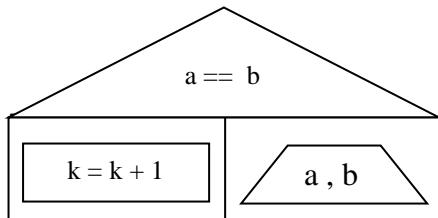
Puede verse que tanto para la parte verdadera como para la falsa habrá un bloque que contiene las instrucciones entre llaves. La utilización del bloque entre llaves solo es necesario si dentro del verdadero o del falso hay más de una instrucción. Si solo hay una instrucción las llaves son opcionales.

Si desea realizar una acción "solo" si se cumple la condición, se utiliza solo la parte verdadera. El uso del else es opcional, por lo tanto, si no hay instrucciones por la parte falsa directamente no se escribe el else.

Es importante ser ordenados al escribir el código fuente para poder interpretarlo rápidamente al leerlo, para ello se utilizará lo que se conoce como identación. La identación es escribir el texto dejando espacio delante, se recomienda dejar ese espacio presionando la tecla tab (tabulación). En estas estructuras servirá para indicar cuáles son las instrucciones que están en la parte verdadera y cuales, en la parte falsa, rápidamente en un golpe de vista. En los siguientes ejemplos se mostrará el código correctamente identado.

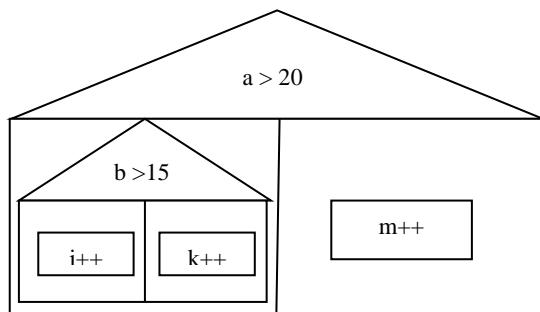
Ejemplo 1:



Ejemplo 2:


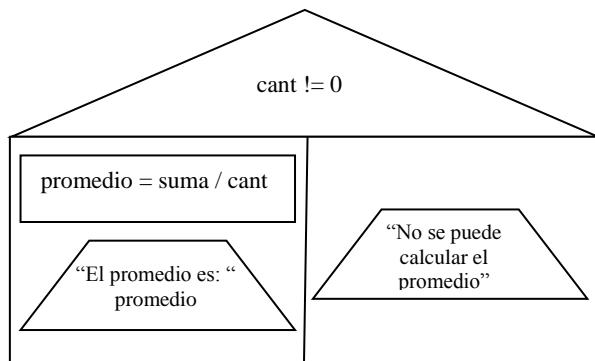
```

if ( a == b)
    k = k + 1;
else
    printf(" %f %f ",a,b);
  
```

Ejemplo 3:


```

if (a>20)
    if (b>15)
        j++;
    else
        k++;
else
    m++;
  
```

Ejemplo 4:


```

if (cant!=0)
{
    promedio = suma/cant;
    printf ("el promedio es: %f", promedio);
}
else
    printf ("no se puede calcular el promedio");
  
```

3 Operador Condicional (?:) [if reducido]

Forma general:

```
expresion? Instruccion1: instrucion2;
```

Si la expresión es verdadera se ejecuta la instrucción1, caso contrario se ejecuta la instrucción2.

Por ejemplo, el siguiente programa muestra si un número ingresado es par o impar utilizando el operador condicional.

```

#include <stdio.h>
int main()
{
    int n;
    printf("Ingrese un numero:");
    scanf("%d", &n);
    (n%2==0)?printf("par"):printf("impar");
    return 0;
}
  
```

El mismo programa utilizando la sentencia if quedaría:

```
#include <stdio.h>
int main()
{
    int n;
    printf("Ingrese un numero:");
    scanf("%d", &n);
    if (n%2==0)
        printf("par");
    else
        printf("impar");
    return 0;
}
```

Solo sirve para el caso de que se tenga una única instrucción en el verdadero y una única instrucción en el falso. Se utiliza generalmente para asignar uno de dos valores a una variable. Por ejemplo, realizar un programa que ingrese dos números y calcula la resta si el primero es mayor o igual al segundo y sino calcule la suma.

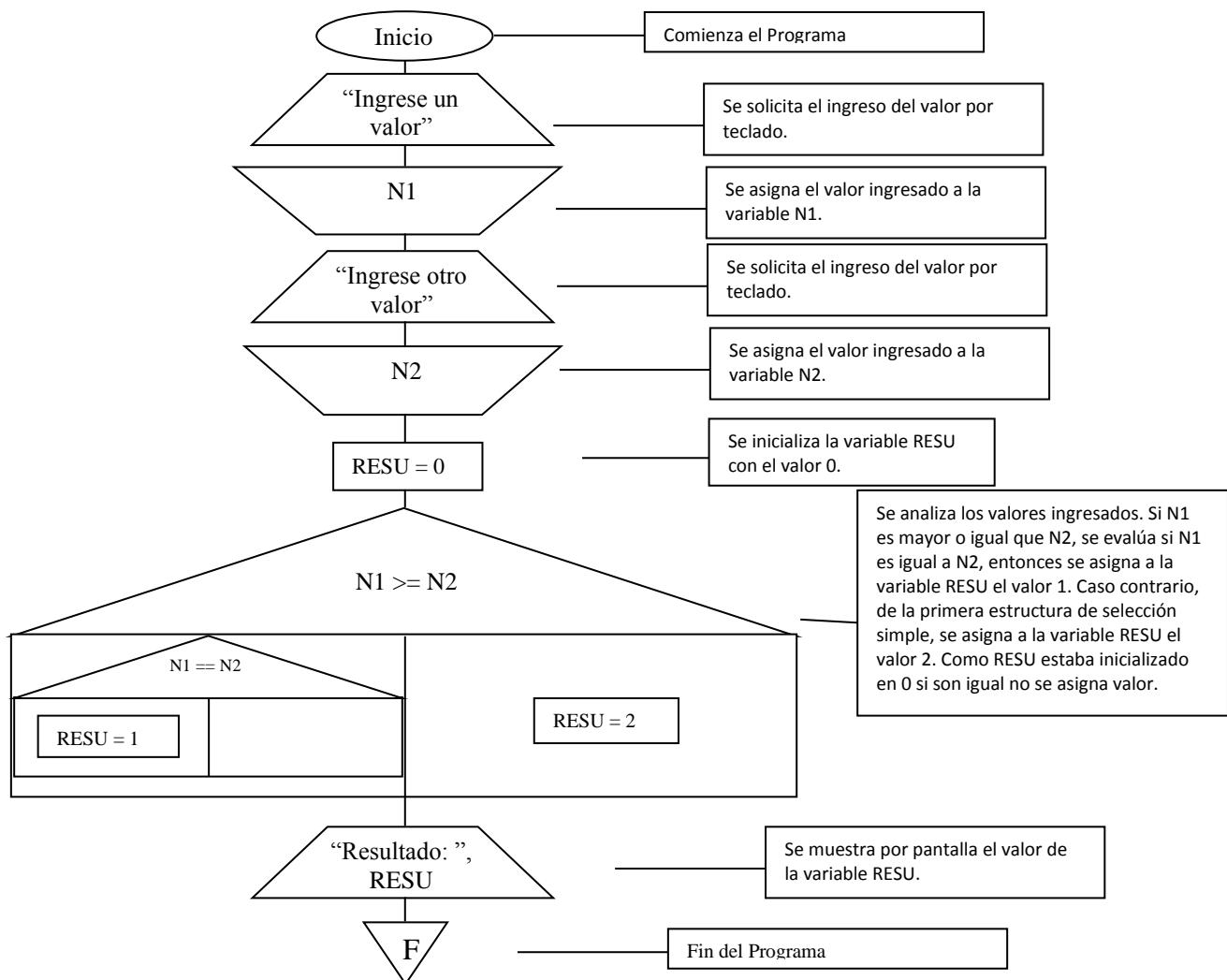
```
#include <stdio.h>
int main()
{
    int n1,n2, resultado;
    printf("Ingrese dos números:");
    scanf("%d%d", &n1, &n2);
    resultado = (n1>=n2)?n1-n2:n1+n2;
    printf ("El resultado es: %d", resultado);
    return 0;
}
```

4 Caso Particular de la sentencia if

Este caso se presenta cuando una estructura "completa" tiene en su rama "verdadera" a una estructura de selección "incompleta". La solución para este caso llega con un par de llaves que encierren a la estructura de selección incompleta, transformándola en una estructura completa.

Recuerde también que cada estructura se caracteriza por tener un "único" punto de entrada y un "único" punto de salida, lo que equivale a hablar de una "sola" sentencia. La causa es debido a que por regla general "cada else siempre es apareado con el inmediato if anterior no cerrado"

Ejemplo: realizar un programa que permita ingresar dos números y de cómo resultado 0 si el primero es mayor que el segundo, 1 si son iguales, 2 si el segundo es mayor que el primero.



Recuerde que siempre para resolver un ejercicio hay muchas soluciones posibles, seguramente esta no es la mejor forma de plantearlo, pero sirve para exemplificar el caso particular que se quiere mostrar ya que dentro del verdadero del primer if hay un segundo if incompleto (sin else), pero el segundo if si tiene su parte falsa, por lo tanto, si se codifica de la siguiente forma:

```

#include <stdio.h>
int main()
{
    int n1,n2, resu=0;
    printf("Ingrese dos numeros:");
    scanf("%d%d", &n1, &n2);
    if (n1>=n2)
        if (n2==n1)
            resu=1;
        else
            resu=2;
    printf ("Resultado: %d", resu);
    return 0;
}
  
```

Al probarlo si se ingresa, por ejemplo, el valor 10 para el primero número y 2 para el segundo, se debería obtener como resultado un 0 ya que el primer número es mayor al segundo, sin embargo, muestra un 2 ya que el código está mal hecho.

Es incorrecto ya que el else será tomado como parte del if inmediatamente anterior. Para solucionarlo se debe armar un bloque para completar la secuencia if incompleta de la siguiente forma:

```
if (n1>=n2)
{
    if (n1==n2)
        resu =1;
}
else
    resu =2;
```

Otra forma de resolverlo sería incluyendo un else vacío de la siguiente forma:

```
if (n1>=n2)
    if (n1==n2)
        resu =1;
    else;
else
    resu =2;
```

5 Operadores Lógicos

Los operadores lógicos permiten asociar varias expresiones lógicas ó aritméticas formando solo una, lo cual simplificará la cantidad estructuras de selección que se deben utilizar.

Los operadores lógicos son 3:

- AND (y)
- OR (ó)
- NOT (negación)

En el lenguaje C, se utilizan los símbolos de la tabla 3 para expresar los operadores:

Tabla 3: Operadores lógicos en el lenguaje C

| Operador | Símbolo |
|----------|---------|
| AND | && |
| OR | |
| NOT | ! |

Estos operadores trabajan sobre valores lógicos, por lo tanto, pueden aplicarse sobre condiciones o sobre valores numéricos de una variable recordando que el 0 es falso y cualquier valor distinto de cero es verdadero.

El operador AND (&&) evalúa las dos condiciones y retorna verdadero solo en el caso de que ambas condiciones sean verdaderas, es decir, se deben cumplir ambas al mismo tiempo.

El operador OR (||) evalúa las dos condiciones y retorna verdadero cuando al menos una de las dos es verdadera, es decir, se debe cumplir al menos una de las condiciones para que sea verdadera la expresión.

El operador NOT (!) trabaja sobre una sola condición negándola. Es decir, que si es verdadera la hace falsa y si es falsa la hace verdadera.

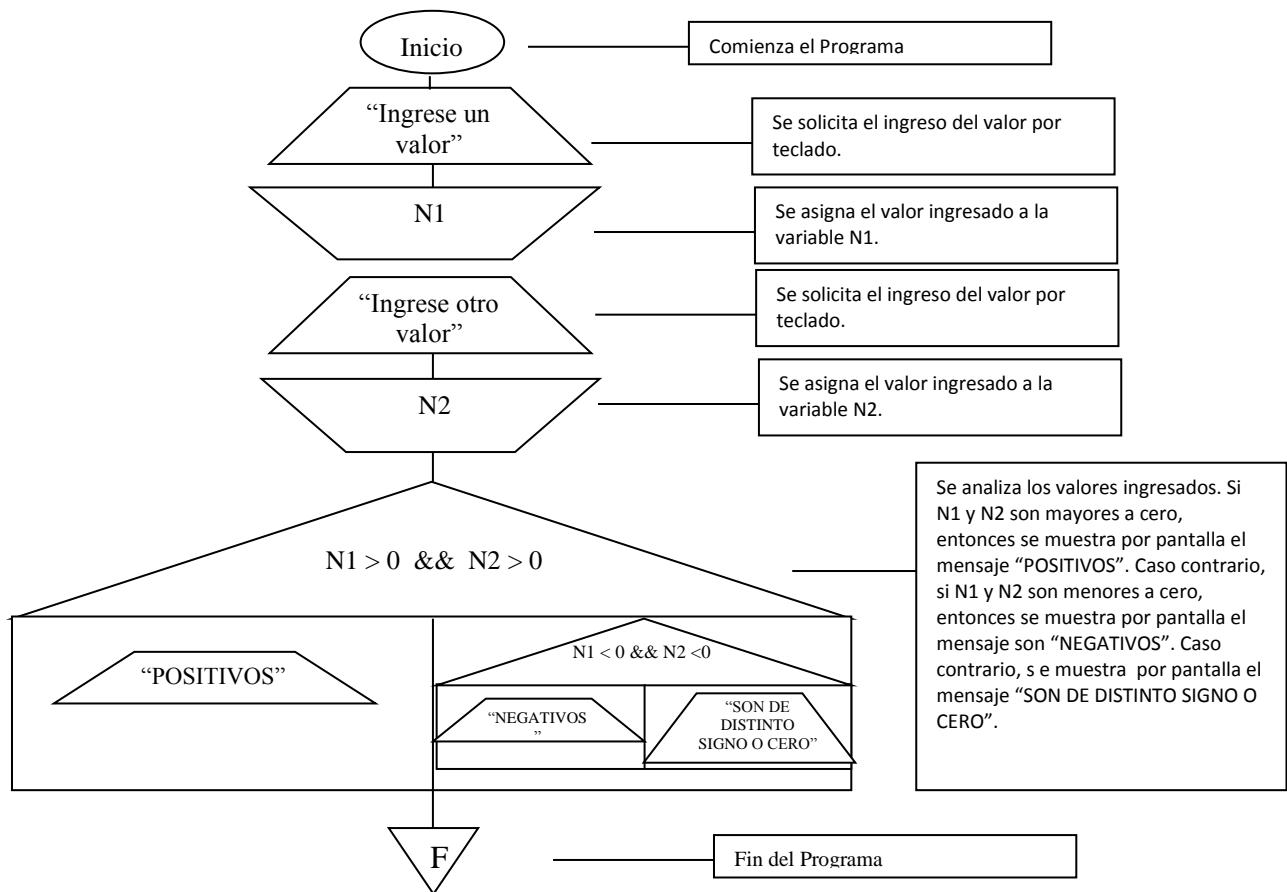
La tabla 4 muestra la tabla de verdad de los operadores lógicos, siendo A y B condiciones lógicas.

Tabla 4: Tabla de verdad de los operadores lógicos

| A | B | A && B | A B | !A |
|---|---|--------|--------|----|
| F | F | F | F | V |
| F | V | F | V | V |
| V | F | F | V | F |
| V | V | V | V | F |

Las precedencias de los operadores son: NOT (!) - AND (&&) - OR (||)

Ejemplo 1: Ingresar dos números. Indicar si ambos son positivos, ambos son negativos, o son distinto signo o cero.

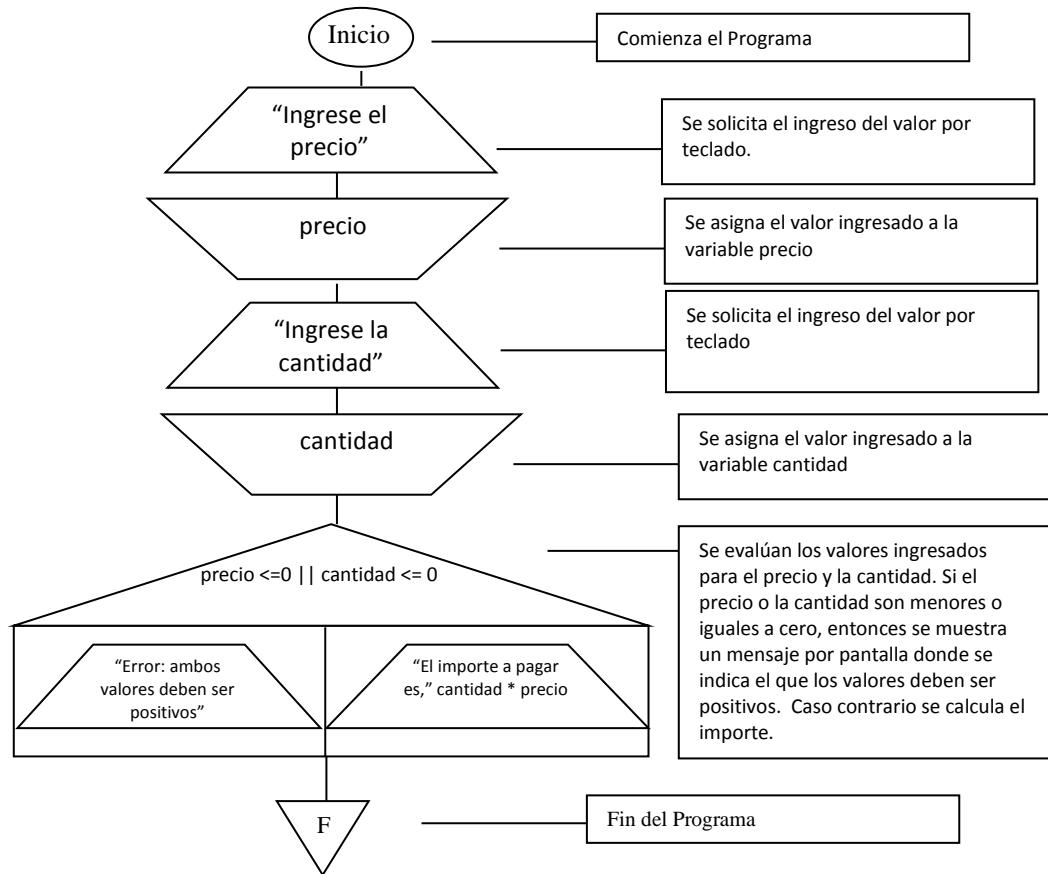


Esta es solo una de las formas de resolver este ejercicio ya que otra persona podría comenzar preguntando por ejemplo si ambos son negativos, o si son cero o de distinto signo. Recordemos que siempre hay varias formas para resolver un mismo ejercicio.

ACLARACIÓN IMPORTANTE: El lenguaje C evalúa las condiciones en orden, si detecta que ya se cumple o no, no sigue evaluando las otras condiciones. Otros lenguajes evalúan todas las condiciones por más que ya se sepa el resultado.

Por lo tanto, en este ejemplo si en N1 se ingresa un número negativo o cero y al ser la primera condición un AND, la misma ya es falsa y no necesita evaluar el valor de N2.

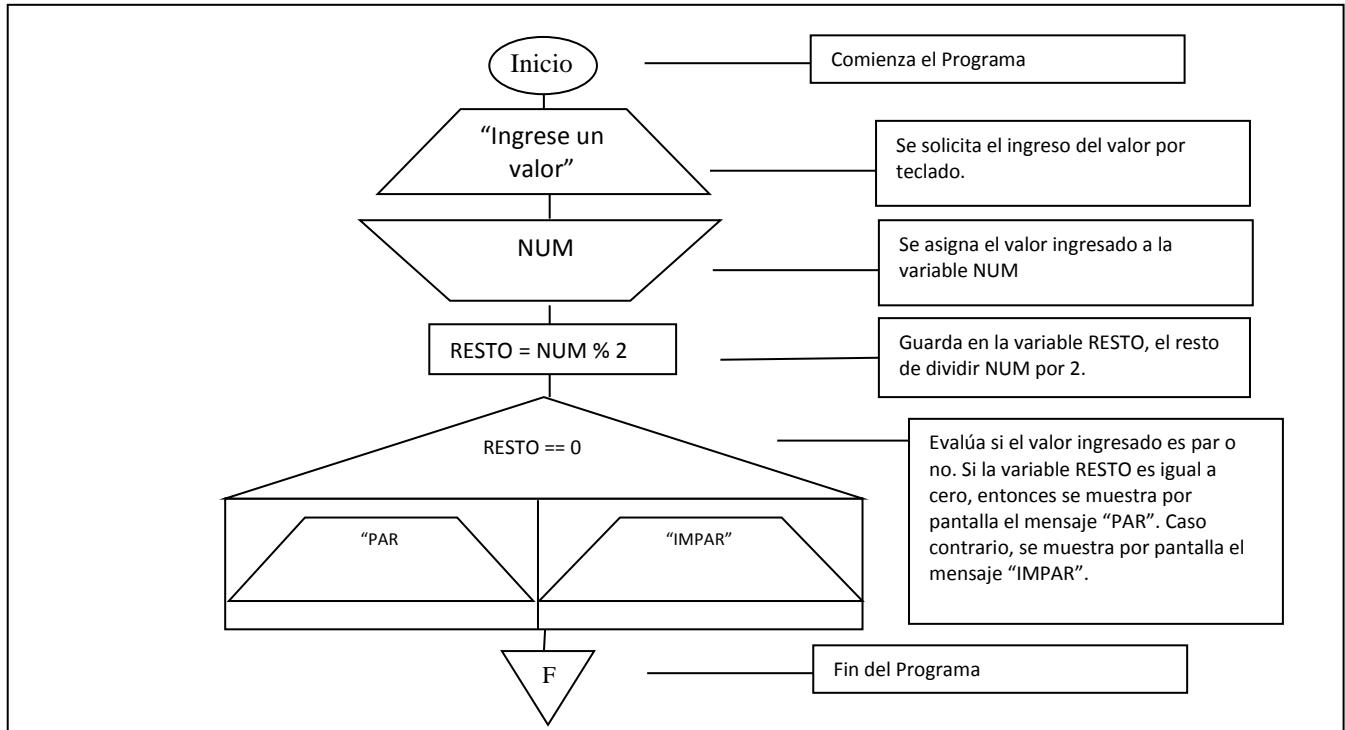
Ejemplo 2: Ingresar precio y cantidad de productos a vender. Mostrar el importe a abonar. Si alguno de los dos es menor o igual a cero informar un error.



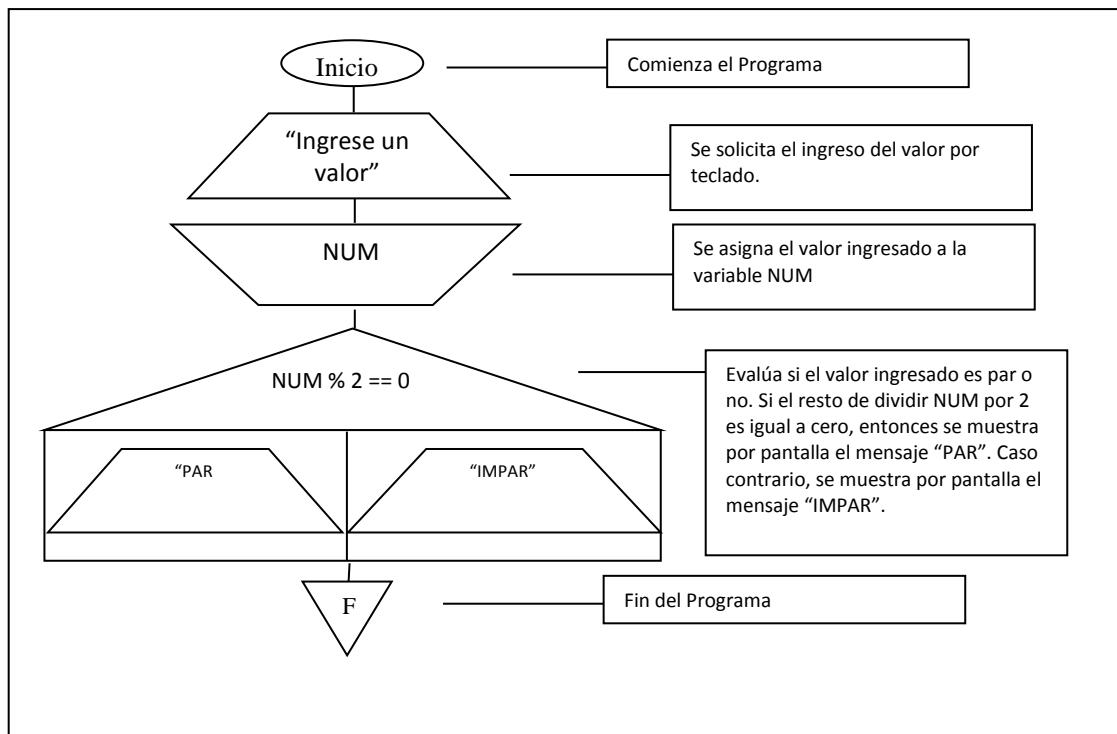
Nuevamente hay que tener en cuenta cómo evalúa las condiciones el lenguaje C. En la primera condición si precio es menor o igual a cero al ser un OR ya la condición es verdadera, y por lo tanto, no evalúa el valor de cantidad.

Ejemplo 3: Ingresar un número e indicar si es par o impar.

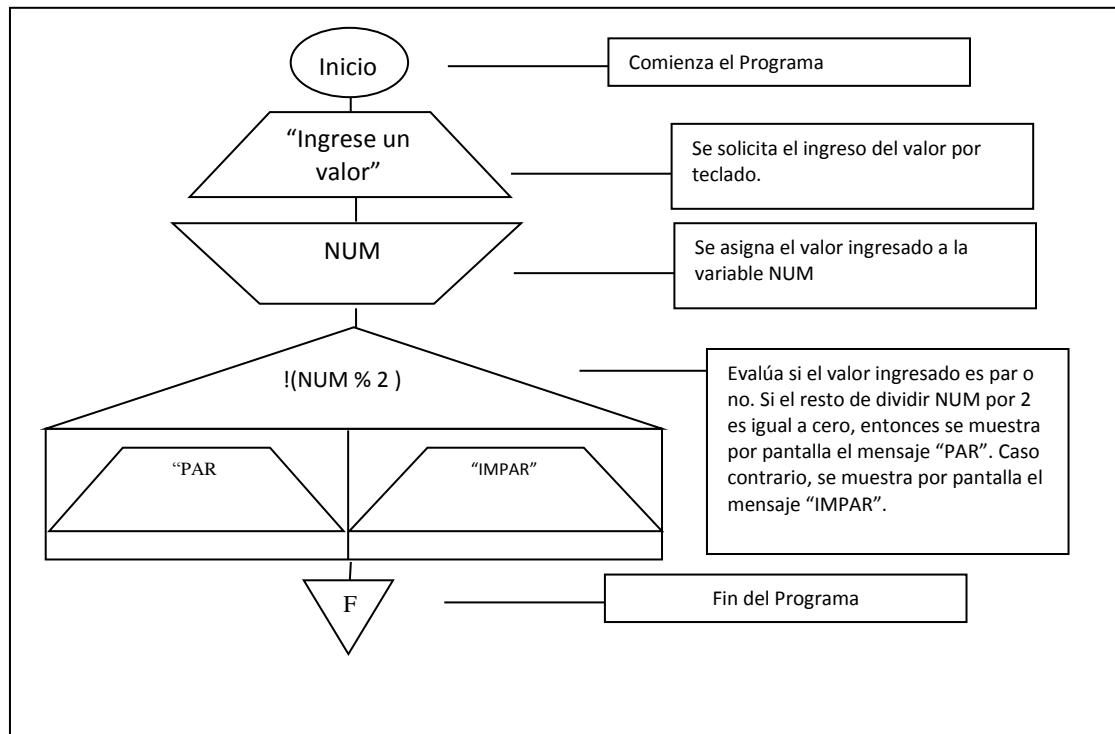
Una forma de resolver este ejercicio es la siguiente:



También se puede quitar la variable resto y hacer la pregunta directamente en la condición de la siguiente manera:



Si se toma en cuenta que en el lenguaje C el 0 es falso y cualquier valor distinto de cero es verdadero se puede directamente evaluar el resultado de la operación sin utilizar un comparador. Si la operación NUM % 2 da como resultado 0 será entonces falsa, si se quiere hacer verdadera cuando sea cero se debe negar quedando de la siguiente forma:

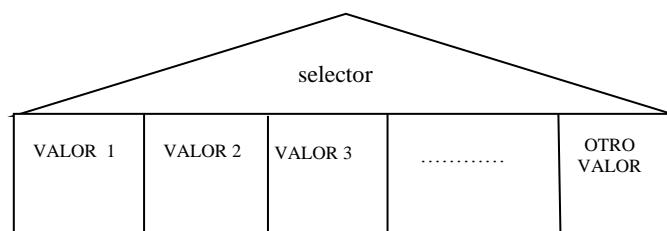


6 Estructura de Selección Múltiple (switch)

Piense como sería un programa que ingresando el número de un mes tiene que informar el nombre. Utilizando la sentencia if se tienen que anidar las condiciones preguntando si el mes es 1 mostrar enero, sino preguntar si el mes es dos mostrar febrero, sino preguntar si el mes es 3, y así sucesivamente, hasta abarcar todas las condiciones.

Para casos de este tipo existe una “estructura condicional múltiple”, que generaliza a la sentencia <if –else>. Esta estructura es útil, por ejemplo, cuando hay un problema en el cual se deben realizar acciones diferentes en base al valor de un código que puede adoptar diversos valores, lo cual obliga a utilizar diversas sentencias if encadenadas.

Su diagrama y codificación son los siguientes:



```

switch (selector)
{
    case valor1:
        //instrucciones a ejecutar si selector toma el valor1
        break;
    case valor2:
        //instrucciones a ejecutar si selector toma el valor1
        break;
    case valor3:
        //instrucciones a ejecutar si selector toma el valor1
        break;
}

```

...

```
default: //instrucciones a ejecutar si selector no toma ninguno de los valores
anteriores
}
```

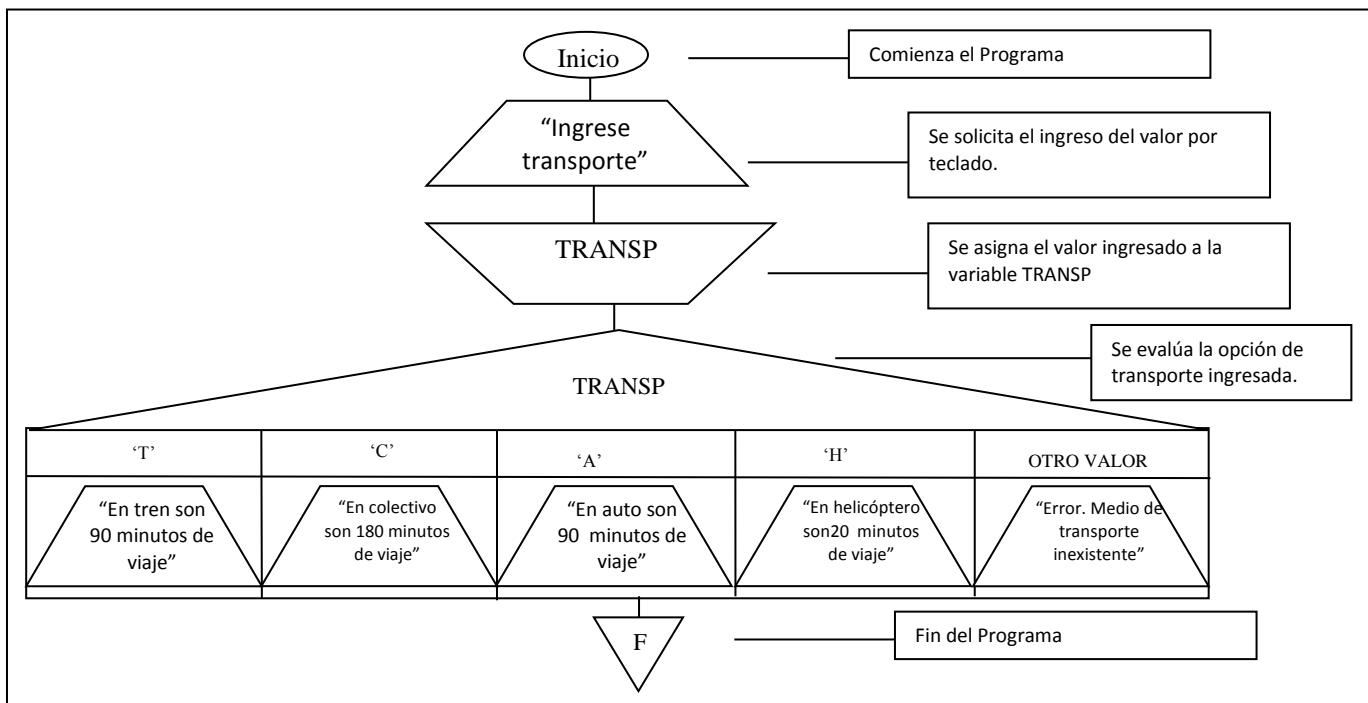
Consideraciones:

- Se evalúa el selector y si su valor coincide con alguno de los casos establecidos se pasa directamente a ejecutar el código de dicho caso
- El default es el valor por defecto, es opcional y se ejecuta si selector no toma ninguna de los valores anteriores.
- Selector puede ser una variable o una expresión, pero debe ser del tipo **entera o carácter**.
- Los valores deben ser constantes (numéricas o caracteres). **NO pueden ser rangos**. Si se necesita evaluar rangos se debe utilizar la sentencia if.
- Cada caso termina con la instrucción break que hace que la ejecución se corte y salga de la estructura del switch. Si no se coloca el break en un caso seguirá ejecutando el código de los casos que están debajo hasta encontrar un break o llegar a la llave de fin. Es por esto que el default, o si el default no está, el último caso no necesita de la instrucción break. Esta particularidad permite que distintos casos compartan el mismo código.

Los casos no tienen necesidad de estar ordenados, pero cada valor solo puede aparecer una vez.

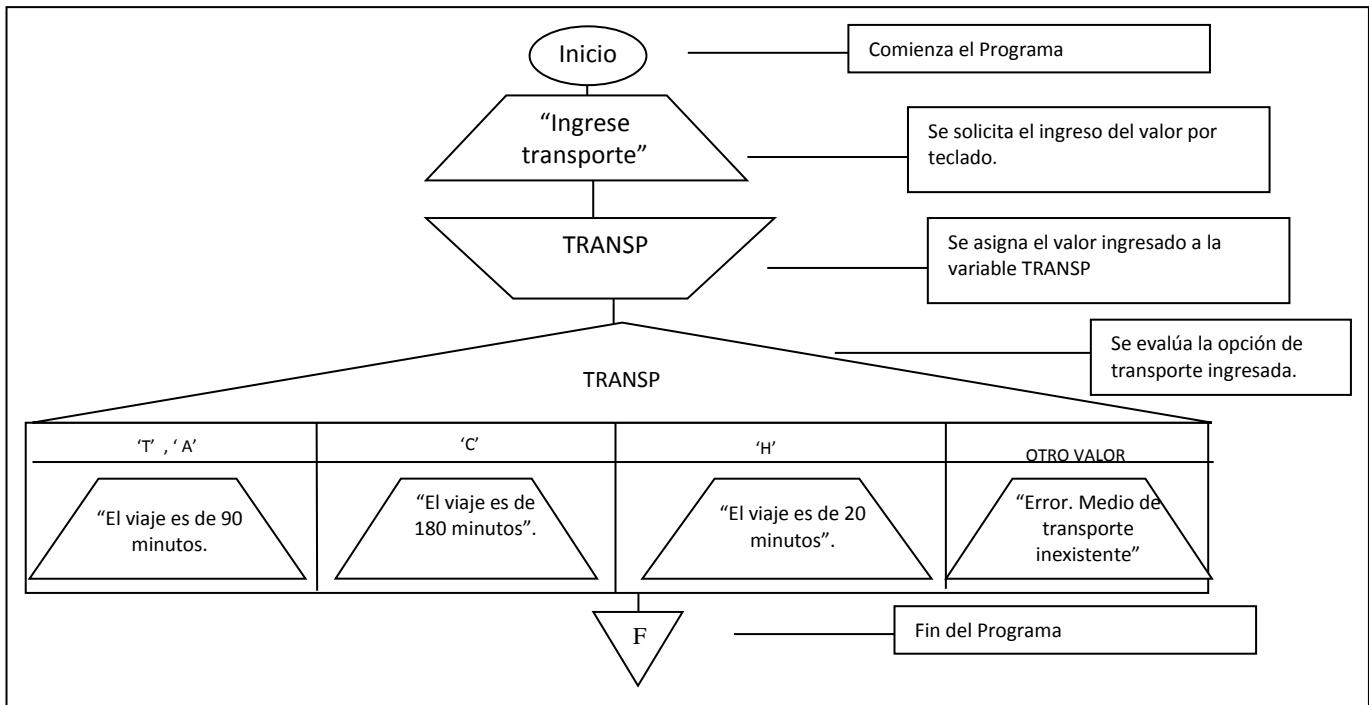
Ejemplo 1: Confeccionar un programa que solicite el medio de transporte para llegar desde La Plata hasta la UNLaM. Mostrando en pantalla el tiempo estimado de viaje para cada medio de transporte. El medio de transporte se ingresa con una letra siendo:

- T: tren, tiempo estimado de viaje 90 min.
- C: colectivo, tiempo estimado de viaje 180min
- A: auto, tiempo estimado de viaje, 90 min.
- H: helicóptero, tiempo estimado de viaje, 20 min.



Nótese que los tiempos en tren y auto son los mismos.

Entonces el algoritmo anterior se puede mejorar de la siguiente manera.



El diagrama quedará de la siguiente manera en código:

```

#include <stdio.h>
int main()
{
    char transp;
    printf("Ingrese transporte:");
    scanf("%c",&transp);

    switch(transp)
    {
        case 'T':
        case 'A': printf("El viaje es de 90 minutos.");
                    break;
        case 'C': printf("El viaje es de 180 minutos.");
                    break;
        case 'H': printf("El viaje es de 20 minutos.");
                    break;
        default: printf("Error. Medio de transporte
inexistente.");
    }

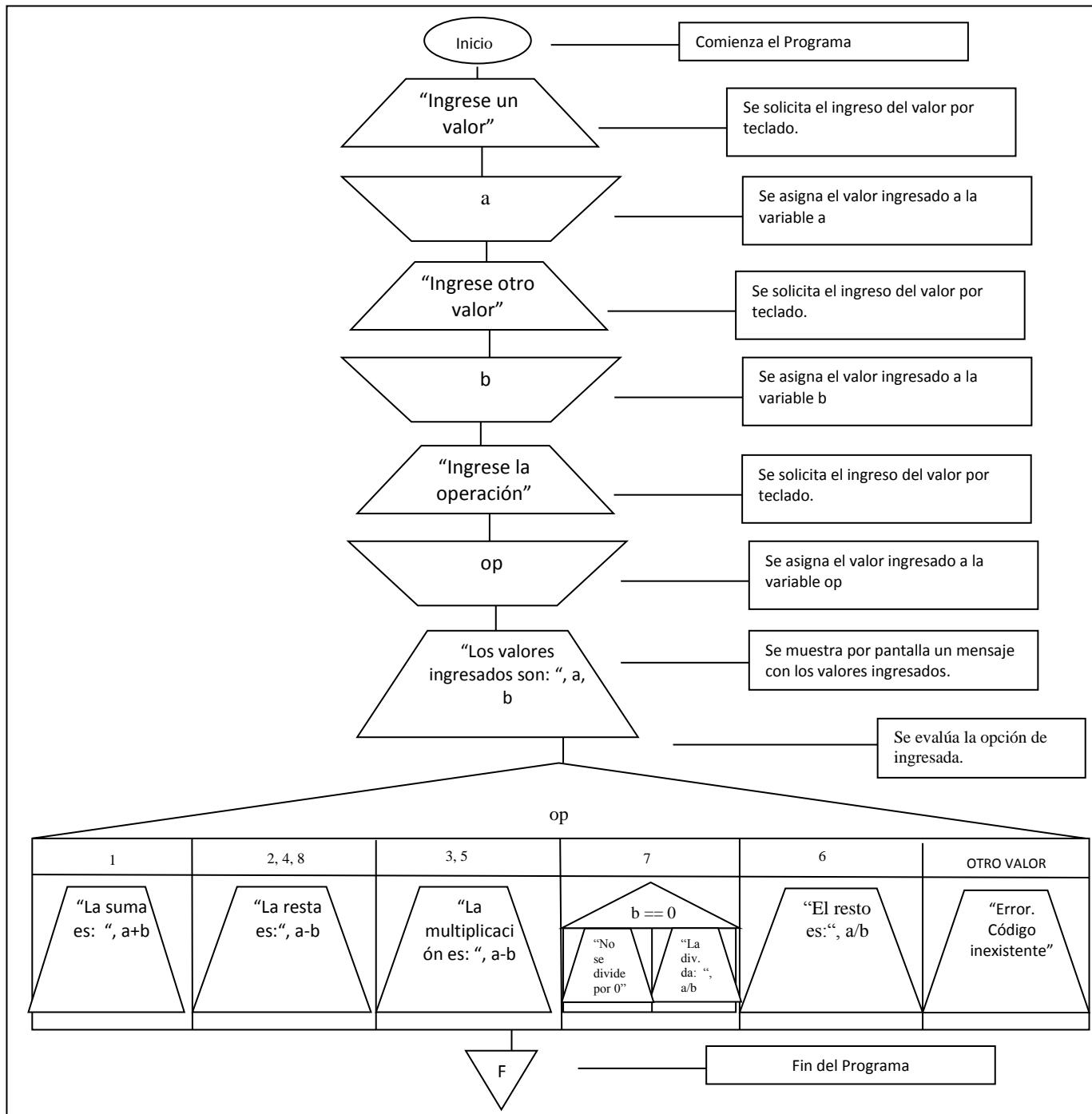
    return 0;
}
    
```

Ejemplo 2: Confeccionar un programa que permita ingresar un par de valores enteros a y b, que corresponden a las temperaturas registradas en un día y un código que indica la operación que se debe realizar con dichos valores.

El código de operación puede variar entre 1 y 8. Según el valor del código se pide calcular e informar:

| | |
|---------------------|-----------------|
| Valor código | Calcular |
| 1 | $a + b$ |
| 2, 4 ó 8 | $a - b$ |
| 3 ó 5 | $a * b$ |
| 7 | a / b |
| 6 | $a \% b$ |
| < 1 ó > 8 | rechazar |

Por cada código ingresado imprimir los valores leídos y el resultado obtenido, con leyendas.



El diagrama quedará de la siguiente manera en código:

```
#include <stdio.h>
int main()
{
    int a,b,op;

    printf("Ingrese un valor:");
    scanf("%d",&a);

    printf("Ingrese otro valor:");
    scanf("%d",&b);

    printf("Ingrese la opción:");
    scanf("%d",&op);

    printf("Los valores ingresados son:%d %d",a,b);

    switch(op)
    {
        case 1: printf("La suma es: %d",a+b);
                  break;
        case 2:
        case 4:
        case 8: printf("La resta es: %d",a-b);
                  break;
        case 3:
        case 5: printf("La multiplicación es: %d",a*b);
                  break;
        case 7: if(b==0)
                  printf("No se puede dividir por cero");
                  else
                  printf("La división es: %d",a/b);
                  break;
        case 6: printf("El resto de la división es: %d",a%b);
                  break;
        default: printf("Error. Código inexistente.");
    }

    return 0;
}
```



Elementos de Programación

UNIDAD 5. ITERACION

INDICE

| | | |
|----|--|----|
| 1. | ITERACIÓN O REPETICIÓN | 2 |
| 2. | ESTRUCTURAS DE ITERACIÓN DEFINIDA | 2 |
| 3. | CONTADORES Y ACUMULADORES | 5 |
| 4. | LENGUAJE C: ESTRUCTURAS DE ITERACIÓN DEFINIDA | 6 |
| 5. | MÁXIMOS Y MÍNIMOS..... | 8 |
| 6. | ESTRUCTURAS DE ITERACIÓN DEFINIDA ANIDADAS | 10 |
| 7. | ESTRUCTURAS DE ITERACIÓN CONDICIONADA | 11 |



UNIDAD 5 - Iteración

Objetivos: Aplicar en la solución de los problemas las estructuras iterativas, en sus variantes. Realizar programas con algoritmos de búsqueda de máximos y mínimos. Comprender los conceptos de variables acumuladoras y contadoras. Utilizar las estructuras de iteración para validar los datos de entrada.

1. Iteración o repetición

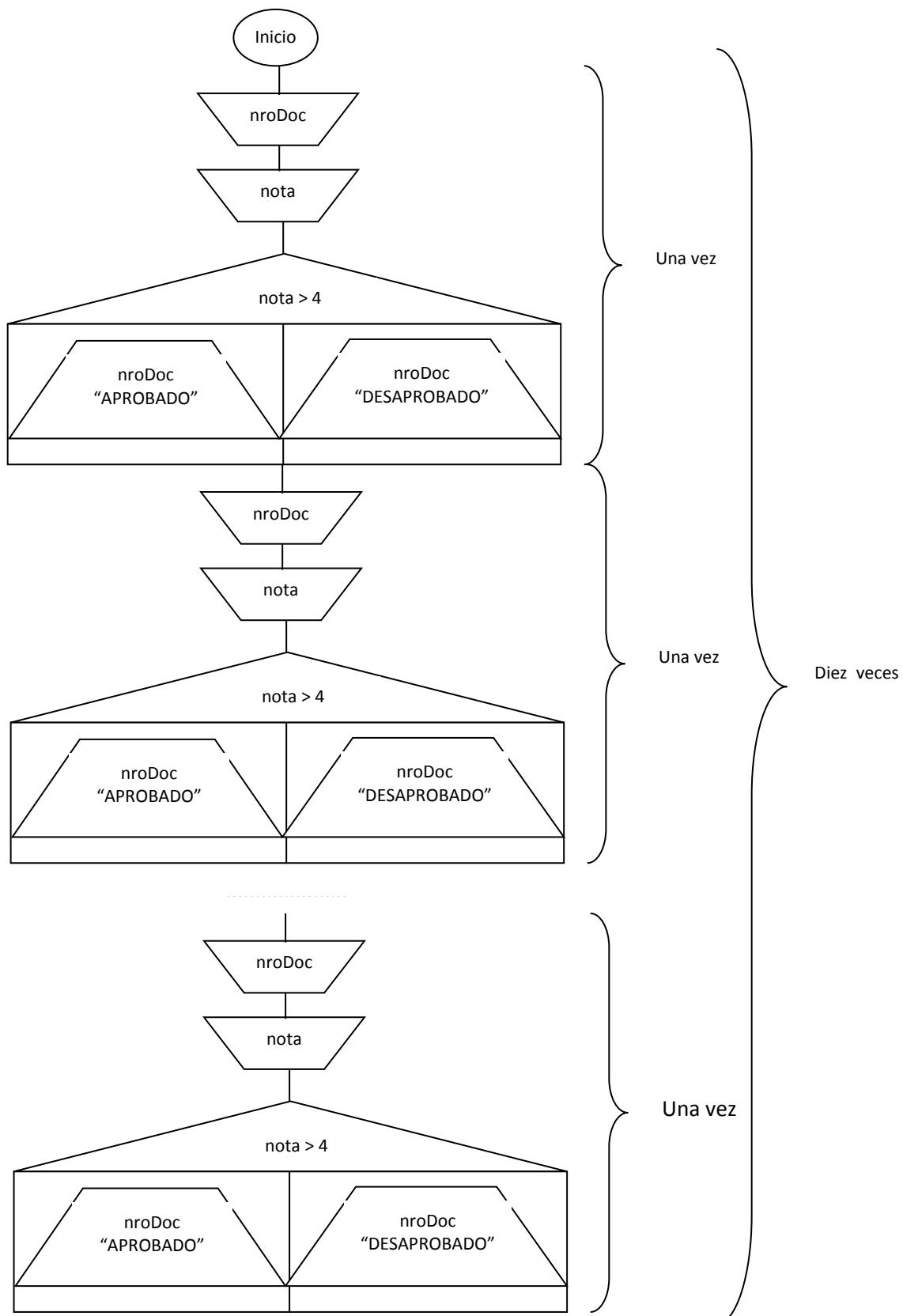
En los programas vistos hasta ahora cada instrucción se ejecutaba una sola vez, en el orden que aparece en el programa. Pero a veces es necesario repetir la ejecución de un grupo de instrucciones, para lo cual se van a utilizar las estructuras de repetición o iteración.

Existen dos tipos de ciclos de repetición:

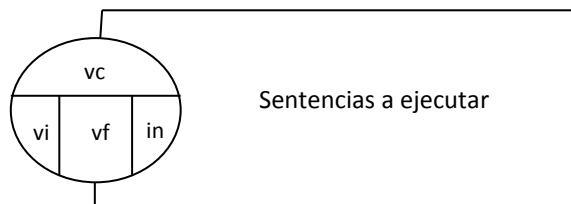
- ITERACIÓN DEFINIDA: Cuando se conoce de antemano la cantidad “exacta” de veces que se debe repetir ese proceso o grupo de sentencias.
- ITERACIÓN CONDICIONADA: Cuando NO se conoce la cantidad de iteraciones a efectuar, es decir, que la repetición depende del cumplimiento de cierta condición.

2. Estructuras de iteración definida

Se desarrolla el siguiente ejemplo: Ingresar número de documento y nota de los 10 alumnos de un curso, informar por cada uno el número de documento y la leyenda “APROBADO” o “DESAPROBADO” si la nota es mayor a 4 o no.



Analizando el diagrama se observa que un grupo de instrucciones se repiten 10 veces. Para eliminar las repeticiones se debe escribir las instrucciones repetidas una sola vez y efectuar un mecanismo

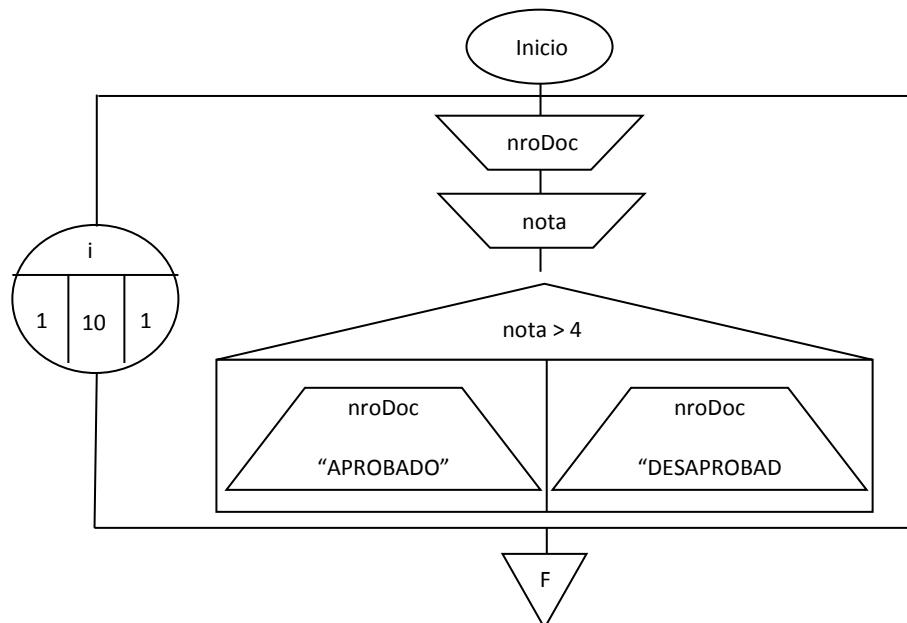


para realizar la repetición 10 veces. Para ello, se debe agregar en el diagrama el ciclo de repetición definida:

Donde:

- **vc**: es la variable de control, ya que con ella se controla el ciclo.
- **vi**: es el valor inicial que toma la variable de control.
- **vf**: es el valor final que debe tomar la variable.
- **in**: incremento que se le aplica a la variable cada vez que se ejecuta el ciclo.

Funcionamiento: Se asigna a la variable de control el valor inicial, se compara con el valor final, si es menor o igual que éste, se ejecutan las sentencias dentro del ciclo, luego incrementa a la variable con el valor del incremento, si sigue siendo menor o igual al valor final realiza otra ejecución y así hasta que llegue a superar el valor final.



Se aplica esta estructura al problema anterior:

El ciclo de iteración definida no siempre necesariamente debe iniciar en 1 ni ser creciente, por ejemplo, para repetir 10 veces un proceso también se puede:

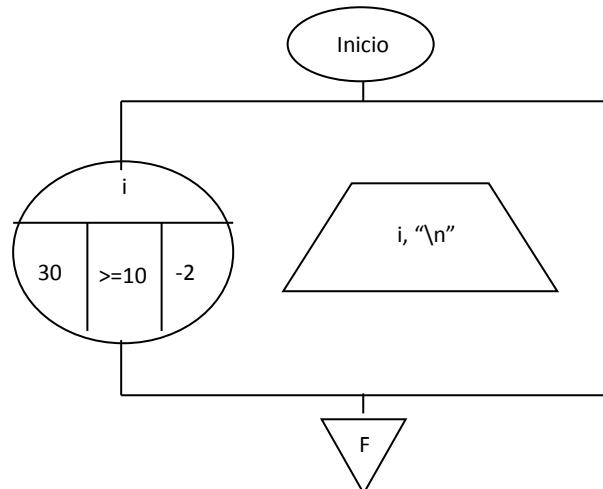
- Poner como valor inicial el 10
- Poner como valor final la condición > 0
- Poner -1 como el valor de incremento, generando de esta forma un decremento

Observe entonces que en realidad el valor final no es un número sino una condición, que se denomina condición de permanencia dentro del ciclo. Si esa condición se cumple entonces el ciclo

se repite. Si no se escribe el operador de comparación se toma por defecto el `<=` (menor o igual) en cualquier otro caso se debe escribir el operador de comparación.

Tampoco el incremento o decremento debe ser siempre 1 o -1 sino que puede ser cualquier otro valor que modifique la variable de control.

Ejemplo: Realizar un programa para mostrar los números pares entre 30 y 10 en forma descendente.



3. Contadores y acumuladores

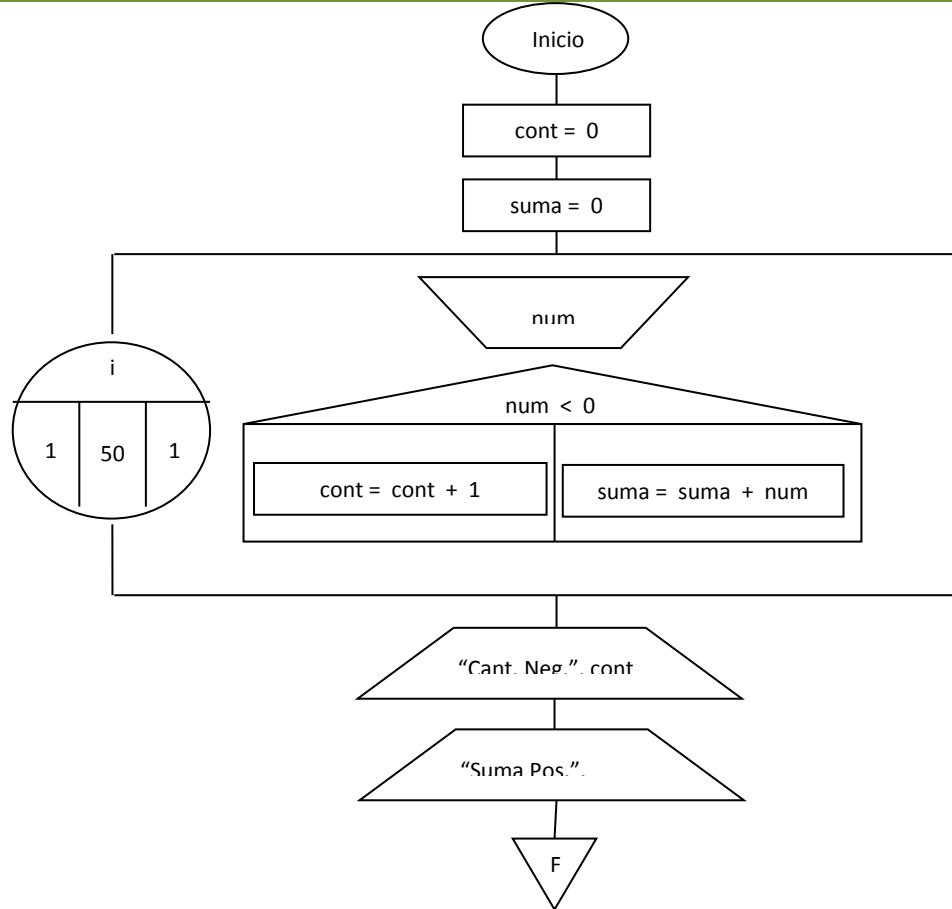
Un contador es una variable en la memoria que se incrementará en una unidad cada vez que se ejecute el proceso. El contador se utiliza para llevar la cuenta de determinadas acciones que se pueden solicitar durante la resolución de un problema. Hay que realizar la inicialización del contador o contadores. La inicialización consiste en poner el valor inicial de la variable que representa al contador (generalmente con el valor 0).

Un acumulador es una variable que suma sobre sí misma un conjunto de valores, para esta manera tener la suma de todos ellos en una sola variable. De igual forma, se pueden efectuar decrementos en un acumulador.

La diferencia entre un contador y un acumulador es que mientras el primero va aumentando de uno en uno, el acumulador va aumentando o decrementando en una cantidad variable.

Ejemplo: Ingresar 50 números, informar la cantidad de números negativos y la sumatoria de los positivos.

Se deben ingresar los números en un ciclo de repetición definido. Por cada número se tiene que consultar si es menor que cero o no. Si es menor que cero se cuenta utilizando un contador (incrementando el contador en uno), si no se suma en un acumulador (sumarle al acumulador ese número).



4. Lenguaje c: estructuras de iteración definida

El ciclo de repetición definida se codifica en lenguaje C con el siguiente formato:

```

for (expresión de inicialización; expresión de verificación; expresión de incremento)
{
    sentencial;
    sentencia2;
    .....
}

```

Las llaves son opcionales de haber una sola sentencia.

Ejemplo: Se escribe en lenguaje C el Ejemplo anterior que ingresa 50 números, cuenta y acumula:

```

#include <stdio.h>
int main( )
{
    int cont, suma, i, num;
    cont = 0;
    suma = 0;
    for ( i = 1; i <= 50; i++)
    {
        printf("\n Ingrese un numero: ");
        scanf("%d", &num);
        if (num < 0)
            cont = cont + 1;
        else
            suma = suma + num;
    }
}

```



```
printf("La cantidad de numeros negativos es %d \n", cont);
printf("La sumatoria de los numeros positivos es %d \n",
suma);
    return 0;
}
```

Para incrementar el contador, en lugar de `cont = cont + 1` se podría usar el operador de autoincremento `cont++`. En el uso del acumulador, se podría usar el operador de acumulación `suma += num`.

OBSERVACIONES IMPORTANTES:

- La prueba de la condición de final se efectúa siempre al inicio del bucle.
- El valor de la variable que actúa como índice, en el lenguaje C, puede ser modificado dentro del ciclo, pero se recomienda NO hacerlo ya que se modifica el comportamiento de la estructura de repetición definida.
- Finalizado el for, el valor del índice queda con el valor de la última ejecución realizada más el valor del incremento, o sea el valor con el cual no pudo ejecutar otra pasada.
- La expresión del incremento puede ser del tipo `in = in + x`, puede contener variables que no figuran como parámetros del for.
- Tener la precaución de "no" colocar un ; (punto y coma) luego de los paréntesis del for, lo cual hace ignorar el bucle, generando un ciclo vacío.
- Pueden omitirse una o todas las expresiones del for, pero NO los ";"
 - a) Si se omite la primera expresión, NO SE ASIGNA VALOR INICIAL dentro del ciclo, pero puede asignarse antes:

Ejemplo:

```
j = 1;
suma = 0;
for ( ; j <=10 ; j++ )
    suma += j; // Suma los números del 1 al 10.
```

- b) Si se omite la segunda, se genera un ciclo INDEFINIDO, porque al no estar la expresión 2, se establece que la prueba siempre será verdadera y por lo tanto nunca saldrá del ciclo.

Ejemplo: `for (j=1; ; j++)`

- c) Si se omite la tercera, se puede realizar el incremento dentro del ciclo for. NO SE RECOMIENDA realizar esta operación ya que es poco clara. Siempre el mejor utilizar todas las expresiones del for.

Ejemplo `j = 1;`
`suma = 0;`
`for (; j <=10 ;)`
`{`
 `suma += j; // Suma los números del 1 al 10.`
 `j++;`
`}`

5. Máximos y mínimos

Es un **problema** común querer encontrar el mayor o el menor valor de una determinada serie de valores. Si se quiere encontrar el mayor valor, se debe pensar igual a un trabajo manual con una pila de hojas donde cada una tiene un valor numérico. Tomo el primero y lo retengo en la mano izq. (**variable mayor**), tomo al segundo con la mano derecha (**variable dato**) y lo comparo con el de la izquierda, si es más grande descarto el de la mano izquierda y paso el de la derecha a la izquierda; se procede igual con los restantes valores, quedando finalmente el mayor en la mano izquierda, o sea, en la variable mayor.

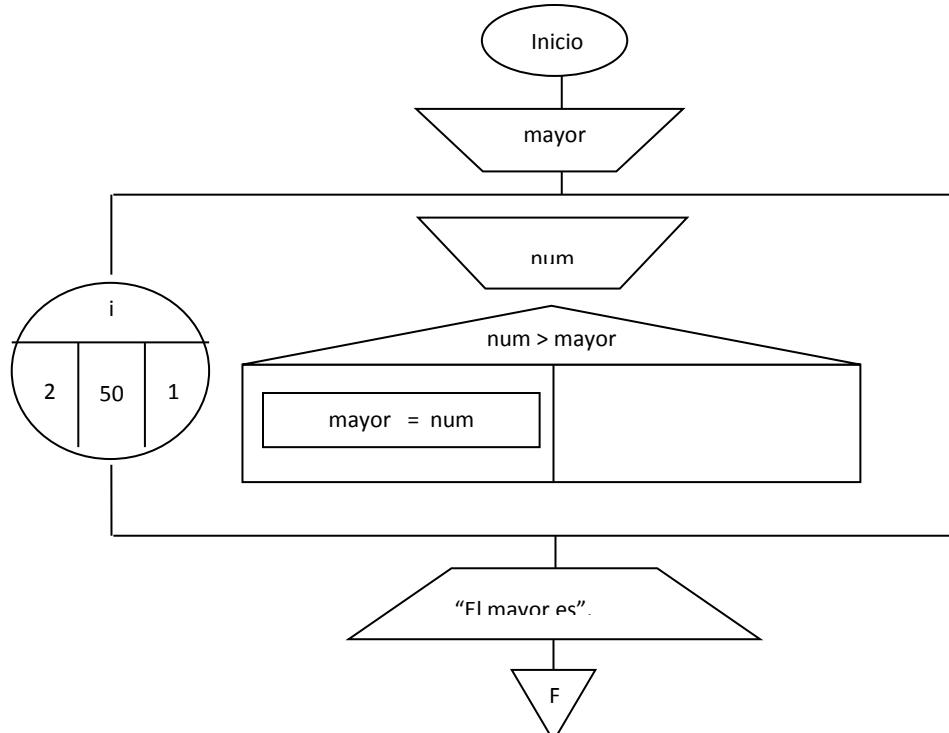
Es decir, que para calcular un máximo de un conjunto de valores se toma el primer valor como referencia, siendo este momentáneamente el más grande. Luego al ingresar otro número se compara con el que se tiene de referencia. Si el nuevo número es mayor al de referencia entonces se reemplaza por el recién ingresado, de esta forma siempre quedará como valor de referencia el más grande y se comparará con cada uno del número ingresados. Si se desea buscar el valor mínimo el procedimiento es exactamente igual, pero se compara si el número recién ingresado es menor al de referencia en cuyo caso se lo reemplaza.

Ejemplo: Ingresar 50 valores numéricos y determinar e informar el mayor.

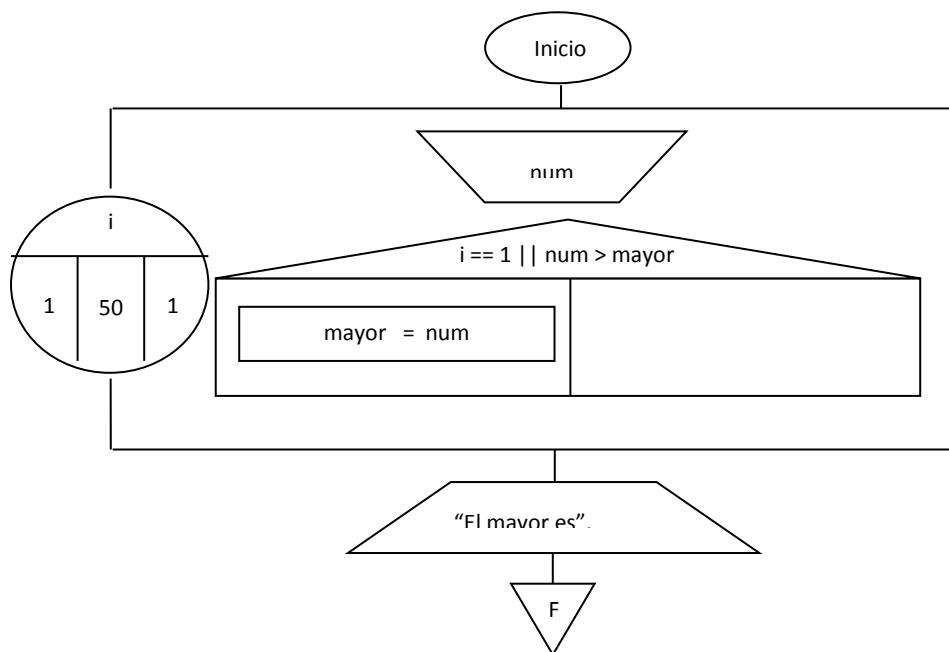
Se puede realizar de 3 formas:

- Leer el primero afuera del ciclo
- Una pregunta para separar el primero del resto.
- Usar una señal para separar el primero del resto.

- a) Leer el primero afuera de ciclo y lo almacenamos en la variable mayor.

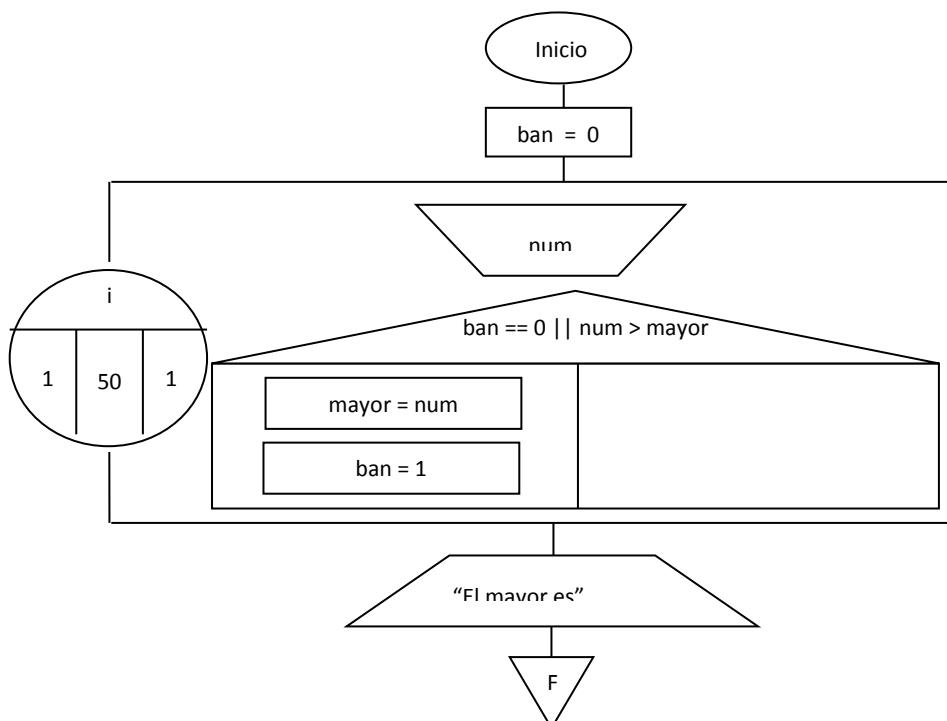


- b) Una pregunta para separar el primero del resto, para darle valor inicial a la variable mayor.



En este ejemplo se realiza un solo if con dos condiciones, se pregunta si es la primera vez que se ingresa al ciclo, condición que se da cuando i vale 1. Luego esa condición será siempre falsa y por lo tanto se evalúa si el número recién ingresado es más grande que el guardado en la variable mayor. Sin embargo, la primera vez que se ingresa la variable mayor aún no tiene valor asignado, pero como el lenguaje C no necesita evaluar todas las condiciones, la primera vez que se ingresa al ciclo la primera condición es verdadera y al ser un or con una condición que sea verdadera es suficiente y por lo tanto no se evalúa la segunda condición.

- c) Una pregunta para separar el primero del resto, para darle valor inicial a la variable mayor, utilizando una señal.



Una bandera o señal cumple con la función de tener el conocimiento de que un evento haya sucedido o no, es una variable a la cual se le asigna solo dos valores, uno se le asigna al inicio del proceso y el otro, al suceder el evento buscado.

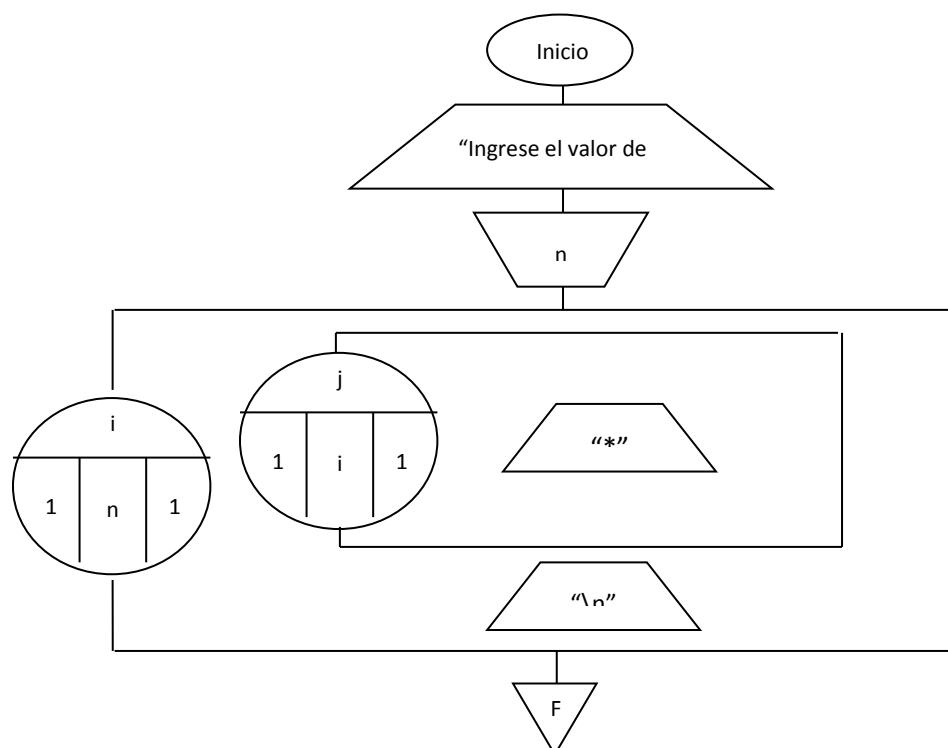
6. Estructuras de iteración definida anidadas

Las estructuras de control iterativas pueden anidarse, es decir, incluir a un ciclo completo dentro de otro. Las estructuras interior y exterior no necesitan ser del mismo tipo, pero es "esencial" que la estructura interior esté "totalmente" incluida en la exterior, no pueden "solaparse". Estamos analizando "for anidados", destacando que pueden ser 2 o más y que las variables utilizadas como control en los ciclos deben ser distintas.

Ejemplo 5: Hacer un algoritmo que imprima un triángulo rectángulo de * con base y altura de n cantidad de *. El valor de n debe ser ingresado por teclado.

Por ejemplo, si n es 4, debe imprimir:

```
*  
**  
***  
****
```



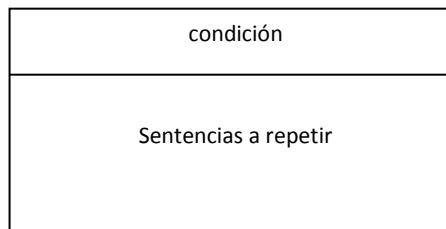
En código C:

```
#include <stdio.h>

int main()
{
    int n, i, j;
    printf("Ingrese el valor de n \n");
    scanf("%d",&n);
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=i; j++)
        {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

7. Estructuras de iteración condicionada

El ciclo definido no da respuesta a los casos en que desconocemos la cantidad de datos a procesar. Existen otras estructuras iterativas que permiten una mayor flexibilidad en la resolución de los problemas, sin tener que conocer previamente la cantidad de datos. Esta estructura requiere que pueda establecerse previamente una condición para poder finalizar, que, en general, es sencilla de definir.



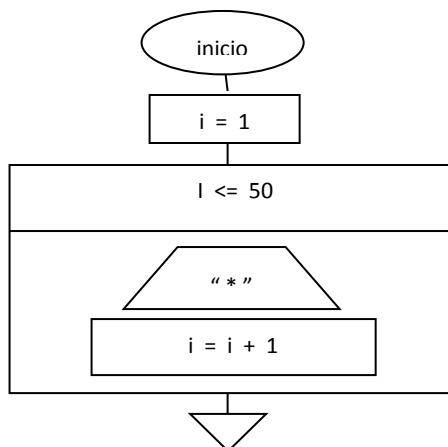
En esta estructura, primero se evalúa la condición, si es verdadera se ejecutan las sentencias que están dentro del ciclo, si es falso se pasa a la sentencia que continua, o sea se abandona el ciclo. Se debe tener precaución en no quedar atrapado dentro del ciclo, para lo cual dentro de las sentencias a repetir se debe incluir alguna que altere la condición de iteración.

Su codificación en lenguaje C es:

```
while (condición)
{
    Sentencia 1;
    Sentencia 2;
}
```

Las llaves son opcionales de haber una sola sentencia.

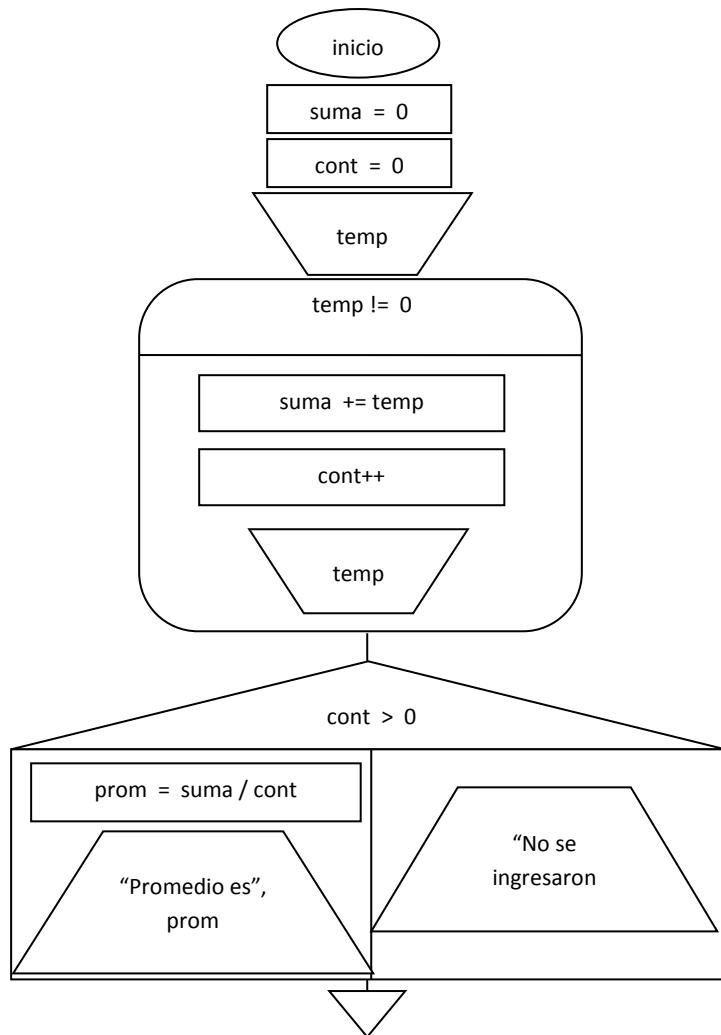
Ejemplo: Confeccionar un programa para imprimir una línea con 50 asteriscos.



```
#include <stdio.h>
int main()
{
    int i = 1;
    while(i <= 50)
    {
        printf("*");
        i++;
    }
    return 0;
}
```

Podríamos resolverlo con una estructura de iteración definida, pero también podemos utilizar una estructura de iteración condicionada como se muestra en el ejemplo.

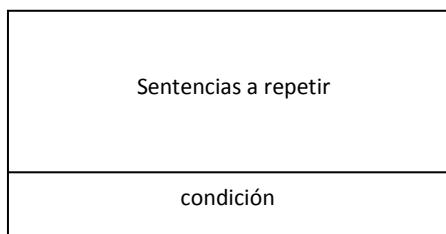
Ejemplo: Confeccionar un programa para ingresar diversos valores de temperatura hasta que aparezca uno igual a cero. Calcular e informar el promedio de los valores ingresados, sin considerar el cero.



```

#include <stdio.h>
int main()
{
    int cont;
    float temp, suma, prom;
    suma = 0;
    cont = 0;
    printf("Ingrese una temperatura (0 fin): ");
    scanf("%f",&temp);
    while(temp != 0)
    {
        suma += temp;
        cont++;
        printf("Ingrese una temperatura (0 fin):");
        scanf("%f",&temp);
    }
    if(cont > 0)
    {
        prom = suma/cont;
        printf("El promedio de las temperaturas es: %.2f",prom);
    }
    else
        printf("No se ingresaron temperaturas (la primera fue cero)");
    return 0;
}
  
```

Existe otra estructura de iteración condicionada, similar en gran medida al “while” denominada do while.

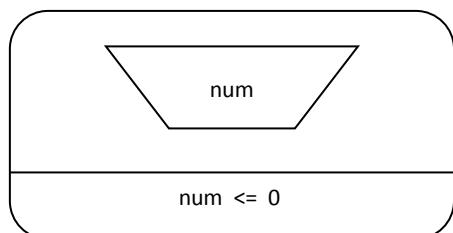


```
do
{
    sentencial;
    sentencia2;
    .....
} while (condición);
```

Se van a ejecutar las sentencias SIEMPRE UNA VEZ y todas las otras veces que sean necesarias mientras la condición sea “verdadera”.

Es útil cuando es necesario ejecutar una serie de sentencias por lo menos una vez. Es un ciclo (1 – n), a diferencia con el ciclo while que es (0 - n). Una aplicación puede ser para controlar el ingreso de los datos.

Ejemplo: Solicitar el ingreso de un valor que debe ser positivo, reiterar la solicitud mientras no se ingrese un valor positivo.



```
do
{
    printf("Ingrese un entero positivo para
continuar: ");
    scanf("%d", &num);
} while(num <= 0);
```

Comparación entre while y do/while

- En ambas estructuras la repetición se produce cuando la condición es verdadera.
- En el while la condición se evalúa “antes” de ejecutar el bucle. Ciclo 0 – N por lo tanto las variables de la condición deben tener valor asignado antes del ciclo.
- En el do/while la condición se evalúa “después” de ejecutar una vez el cuerpo del bucle. Ciclo 1 – N. Por lo tanto, las variables de la condición pueden asignarse y modificarse directamente dentro del ciclo.
- Las variables que figuren en la condición deben modificar su valor dentro del ciclo sino una vez dentro nunca se saldrá del mismo.



Elementos de Programación

UNIDAD 6. FUNCIONES

INDICE

| | | |
|-----------|---|----------|
| 1 | PROGRAMACIÓN MODULAR | 2 |
| 2 | FUNCIONES | 2 |
| 3 | PARÁMETROS Y VALOR DE RETORNO | 3 |
| 4 | DECLARACIÓN DE VARIABLES - GLOBALES Y LOCALES..... | 6 |
| 5. | CODIFICACIÓN DE LAS FUNCIONES | 6 |
| 5.1 | DECLARACIÓN DE UNA FUNCIÓN – PROTOTIPO..... | 7 |
| 5.2 | LLAMADA A UNA FUNCIÓN. | 8 |
| 5.3 | DESARROLLO DE UNA FUNCIÓN | 8 |
| 6. | METODOLOGÍAS PARA LA PROGRAMACIÓN MODULAR..... | 9 |

UNIDAD 6. Funciones

OBJETIVOS: Realizar programas partiendo de pequeñas partes conocidas, denominadas "subprogramas", y que en C se conocen como "Funciones". Crear nuevas funciones. Comprender y aplicar los mecanismos para el intercambio de información entre las funciones.

1 Programación modular

Para simplificar la resolución de los problemas, basado en la programación estructurada, es posible la división del problema PRINCIPAL en pequeños problemas o subproblemas de fácil solución y mantenimiento. Cada subproblema se transformará en una FUNCION.

La solución de los problemas se efectúa desarrollando un Algoritmo, el cual estará compuesto por un tronco o "Algoritmo principal", acompañado de una serie de sub-algoritmos, que unidos adecuadamente resuelven el problema.

El algoritmo principal genera el programa principal, que en C es una función llamada "main". Los sub-algoritmos serán funciones con nombres propios, definidos por el programador.

En general un programa, va a constar de un "tronco" ó programa principal y una serie de subprogramas vinculados con éste. Cuando se ejecuta el programa que contiene funciones, se transfiere en cada llamada, el control al subprograma, se ejecutan sus instrucciones y retorna el control al programa llamador, es como si estuviese intercalado dentro del programa principal.

Las funciones constituyen una herramienta esencial en la programación modular. Por un lado, permiten desarrollar sólo una vez procedimientos (rutinas o subroutines) que son de utilización frecuente o que se utilizan en diferentes lugares de un mismo programa; por otro lado, permiten utilizar mecanismos de abstracción que facilitan la construcción de un programa concentrándose en los aspectos más relevantes y despreocupándose de detalles que pueden ser encarados en otra etapa del diseño.

Una función es, básicamente, un subprograma; por lo tanto, tiene una estructura similar a la de cualquier programa:

Entradas → Proceso → Salida

Sólo que interactúa (se comunica) con otra función o directamente con el programa principal; dicha comunicación se establece a través de argumentos (parámetros) que conforman los datos (entradas) y resultados (salidas). Como cualquier programa, utilizan variables de trabajo denominadas variables locales proveyendo mecanismos de protección que hacen a éstas inaccesibles desde cualquier otro ámbito fuera de la misma función.

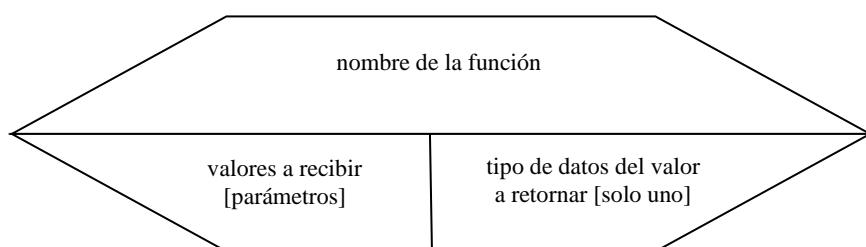
2 Funciones

Una función es una colección de declaraciones y sentencias que realizan una tarea específica, tiene cuatro componentes.

1. El nombre
2. La información que recibe o toma a través de los parámetros para realizar la tarea. (opcional)
3. Las sentencias que realizan la tarea
4. El valor que devuelve cuando termina su tarea (opcional).

Cada función se diagrama por separado, como si fuese otro programa. En la codificación serán incluidas a continuación de la función principal llamada main () .

Para la primera línea de la “definición de la función” se utiliza un gráfico especial, que es el siguiente



Para las llamadas a las funciones se utiliza el gráfico que corresponde a una instrucción donde se invocará a la función por su nombre y se le enviarán los datos para utilizarla si es que los requiere.

Tanto los parámetros como el valor de retorno son opcionales, por lo tanto, podemos tener las siguientes combinaciones:

1. Funciones que NO reciben parámetros y NO retornan ningún valor: por ejemplo, una función que muestra un mensaje fijo, como un encabezado o título del programa.
2. Funciones que reciben parámetros y NO retornan ningún valor: por ejemplo, alguna función que realice algún cálculo y muestre el resultado dentro de la función. También se utiliza este tipo de funciones para los siguientes temas como arrays y archivos.
3. Funciones que NO reciben parámetros y retornan un valor: por ejemplo, para solicitar el ingreso de un dato, validarla y retornar el número ingresado al programa principal.
4. Funciones que reciben parámetros y retornan un valor: por ejemplo, para funciones que realicen cálculos, pero no lo muestren, sino que retorne el resultado para ser utilizado en el programa principal.

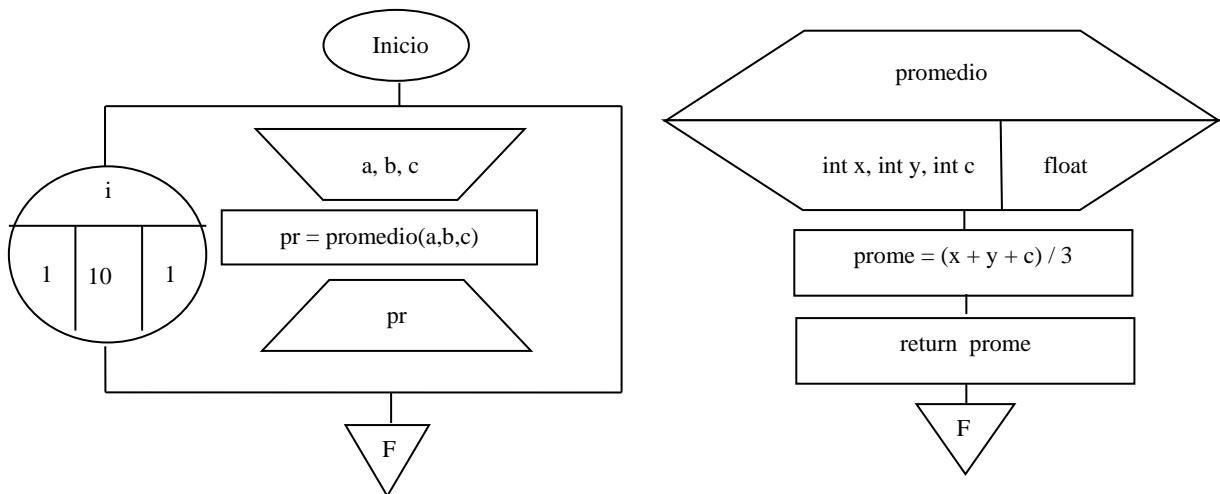
VENTAJAS DEL USO DE FUNCIONES

- Facilita la programación porque divide el problema en subproblemas.
- Simplifica la puesta a punto, ya que puedo corregir o cambiar una función sin tocar el resto del programa. Ídem los futuros cambios.
- Permite trabajar simultáneamente a varios programadores.
- Permite la reutilización de estas funciones sin volver a programarlas, solo se utilizan.
- Ahorra memoria, reduce el tamaño ocupado por el programa cuando uso la misma función varias veces.

3 Parámetros y valor de retorno

A una función opcionalmente se le pueden enviar datos, esos datos son recibidos en la función para ser utilizados dentro de ella. Al especificar la función se detallan esos datos que recibe con su tipo de dato y un identificador que será el nombre de la variable LOCAL a la función donde se guardará el dato recibido. Estos datos que la función recibe se denominan parámetros formales. Veamos el siguiente ejemplo:

Ejemplo: Confeccionar un programa que solicite el ingreso de 10 ternas de valores reales y para cada una de las ternas calcule e informe el promedio de sus valores. Para el cálculo del promedio confeccionar y utilizar una función.

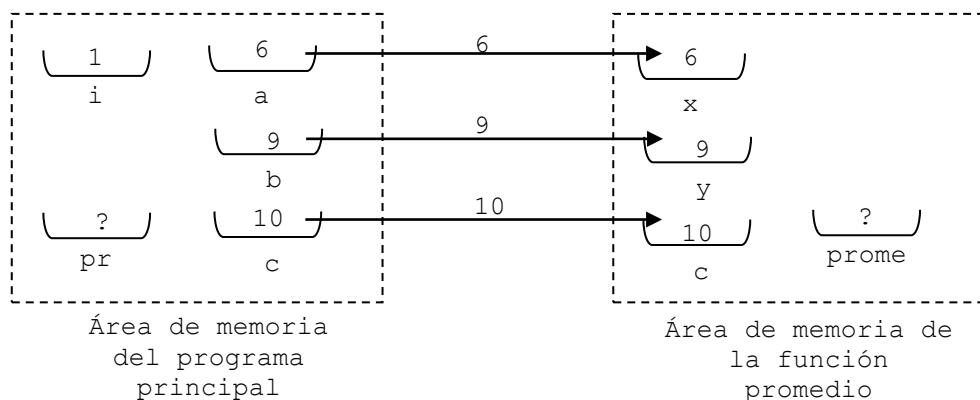


Como se mencionó anteriormente se realizan dos diagramas separados, un diagrama para el programa principal y otro diagrama para el desarrollo de la función. En este ejemplo se ingresan 3 números reales en el programa principal que se almacenan en las variables a, b y c. Luego, se invoca la función promedio enviándole como parámetro los tres valores recién leídos. Los parámetros trabajan por posición, es decir, que el primer valor que se indique en la llamada a la función va a ser guardado en una variable llamada x dentro de la memoria de la función promedio, el segundo la variable y, y el tercero en la variable c.

Las funciones trabajan con direcciones de memoria totalmente separadas a las del programa principal, al invocar a una función lo que se envía es un valor, es decir, el contenido de una variable y no la variable. Este valor es almacenado en un área distinta de memoria propia de la función. Cada vez que se invoca a una función se reserva un área de memoria separada distinta a la que se utilizó en otras llamadas. Además, al ser un área distinta de memoria es posible utilizar el mismo identificador que en el programa principal, pero teniendo en cuenta que aún así es una variable diferente.

Suponga que el usuario ingresa los valores 6, 9 y 10 en el programa principal en las variables a, b y c respectivamente. Al invocar la función lo que se envían son esos números que se COPIAN en los parámetros de la función promedio, por cada parámetro se crea automáticamente una variable local a la función por lo que no deben volver a definirse. En la figura 1 puede ver un esquema de las áreas de memoria del programa principal y de la función promedio.

Figura 1: Esquema de memoria y envío de parámetros del programa principal a una función



Cada casillero representa un espacio de memoria con un identificador que es el nombre de la variable definida. El programa principal tiene 5 variables: k, utilizada como índice del ciclo for, pr para guardar el resultado del promedio y los tres valores ingresados por el usuario a, b y c. La primera vez que se ejecute el ciclo for k toma el valor 1 y a, b y c tomarán los valores ingresados por el usuario mediante el teclado. La variable pr, no tiene valor asignado aún por lo que no se sabe qué valor tiene, tiene un valor aleatorio que estaba ya en la memoria, pero no se sabe cuál es. Como no se inicializa la variable se dice que trae “basura” de memoria porque es un valor anterior que no tiene ningún sentido para el programa.

La función promedio tiene 4 variables ya que se crea automáticamente una variable para almacenar cada uno de los parámetros formales y además se declara dentro de la función una variable local llamada prome. Puede notar que la función promedio tiene una variable c al igual que el programa principal, pero son variables distintas porque están en áreas de memoria separadas. Dentro de la función solo se puede acceder a sus variables locales, por lo tanto, no puede acceder a las variables definidas en el main o en otras funciones. Cada función trabaja con sus propios datos lo que facilita que pueda reutilizarse.

Al invocar a la función desde el programa principal los valores de las variables a, b y c viajan hacia la función, se copian. Esos valores se denominan argumentos de la función y son recibidos y almacenados en las variables definidas automáticamente por los parámetros formales. Desde el programa principal como argumentos de una función se pueden enviar:

- Variables (se enviará el contenido de la misma)
- Constantes
- Operaciones (se resuelve la operación y se envía el resultado)
- Funciones que retornen un valor (al invocar a una función este si retorna un valor, puede ser utilizado como parámetro de otra función y lo que se envía es el valor retornado por dicha función)

Dentro de la función se calcula el promedio y el resultado se guarda en la variable local de la función prome. Luego se utiliza un comando que permite retornar un valor al programa principal, el comando return. Este comando retorna un valor cuyo tipo de dato debe coincidir con el especificado como retorno en la función. En este caso la función debe retornar un valor float, por lo tanto, la variable local prome debe ser también float para poder ser retornada. El esquema de la figura 2 muestra el estado de la memoria al retornar el valor.

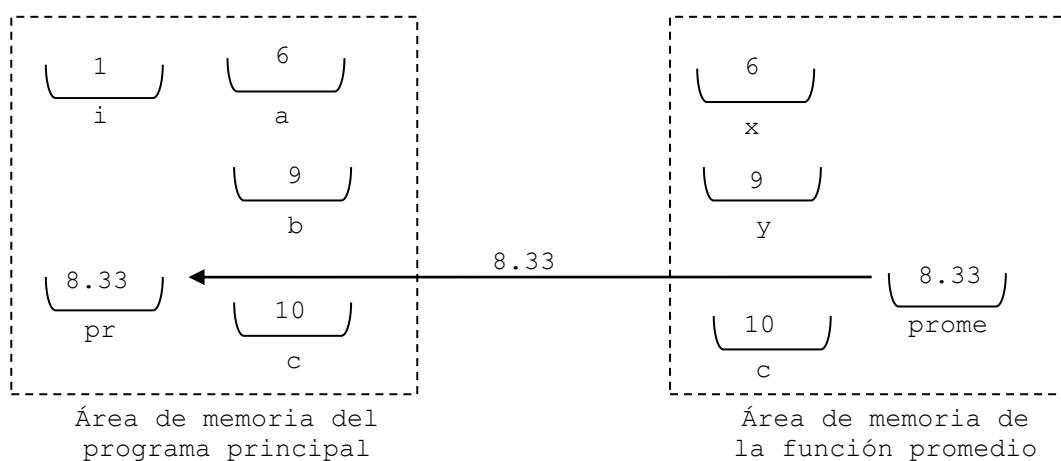


Figura 2: Esquema de memoria y envío del valor de retorno desde la función al programa principal

El valor de la variable prome retorna al programa principal como resultado de la función. Dicho resultado es almacenado en la variable pr. Si no se le asigna el resultado de la función a la variable con la instrucción `pr = promedio (a, b, c)` el valor de retorno se pierde, por lo tanto, si se desea guardar el resultado de una función dicho resultado debe asignarse a una variable local desde donde se la invoca.

Luego de retornar el valor la función finaliza, es decir, que siempre al llegar a una instrucción return se sale de la función por lo que es recomendable usar solo una instrucción de este tipo al final de esta para evitar confusiones. Al salir toda la memoria asignada a la función se pierde, ya no es accesible, por lo tanto, todo lo declarado y calculado dentro de la función se pierde.

4 Declaración de variables - globales y locales

Existen 2 lugares en el programa donde se pueden definir variables:

- Antes del bloque principal, éstas serán variables GLOBALES. Su ámbito de incumbencia es todo el programa, incluso todas las funciones que éste contenga.

Esta definición debe evitarse debido a que puede complicar el programa ya que, si se altera por error una variable global dentro de una función, dicho cambio se reflejará en todo el programa. Además, complica el uso de funciones realizadas por otras personas, ya que debería analizar los nombres y uso de todas las variables.

- Dentro del main () (es una función más) ó dentro de cualquier función, estas variables se llaman LOCALES, porque su ámbito de incumbencia queda reducido solo a la función donde se la define.

Las variables LOCALES ocupan memoria solo cuando se utiliza la función, por lo cual no es posible conservar datos en una variable local, entre 2 ejecuciones sucesivas de la misma función. El uso de las variables LOCALES, aparte de ahorrar memoria permite escribir FUNCIONES AUTOSUFICIENTES, que pueden usarse en otros programas sin más que copiarlas, evitando errores en el uso de las variables. Como ejemplo puede ver el caso de las Funciones de Biblioteca.

Algunas consideraciones:

- Pueden utilizarse variables con igual nombre en distintas funciones: esto puede realizarse, ya que la función busca primero a la variable a usar dentro de las variables definidas como locales, si no la encuentra pasa a buscarla dentro de las variables globales definidas en el programa llamador. Esta es la causa por la cual puedo usar el mismo nombre para una variable local de una función que para una global, o una definida en el main ().
- En general, si se usan los mismos nombres se debe tener cuidado en no confundirse.
- La llamada a una función se puede realizar, en general, desde cualquier sentencia.
- La utilización más eficiente de las funciones es haciendo uso de los PARAMETROS, que son realmente quienes establecen la comunicación entre el programa principal (main) y las funciones, usando variables locales.

5. Codificación de las funciones

La estructura de una función es similar a la de la función principal main (), salvo que comienza su cabecera con "su nombre" en lugar de la palabra "main".

En "tres" lugares del programa se debe mencionar a cada función que se utiliza:

- Declaración del prototipo (solo la primera línea, con los parámetros y sus tipos de datos, ó solo los tipos de datos), al comienzo del programa, luego de las directivas al preprocesador y antes de la línea main ().
- Uso de la función (llamada a la misma) dentro del programa, en el lugar adecuado. Puede usarse en el programa principal o dentro de otra función.
- Desarrollo de la función completa, con su cabecera y sentencias que la componen. Este desarrollo se realiza debajo de la función main ().



```
#directivas al preprocesador (include y define)
declaración de variables "globales";
(a) prototipo →{tipo/void} nombre-función ([parámetros(solo el tipo) ] ) ;
    void main()
    {
        -   declaración de variables locales del main;
        -   sentencias;
(b) uso →      [var =] nombre-función ([argumentos]);
    }
(c) desarrollo→{tipo /void} nombre-función ([parámetros (tipo y nombre)])
    {
        -   declaración variables locales de la función;
        -   sentencias;
        [return expresión;]
    }
```

A continuación, se muestra la codificación del ejemplo presentado anteriormente para calcular el promedio de 10 ternas de valores ingresadas por teclado mediante una función.

```
#include <stdio.h>
float promedio (float,float ,float ); //prototipo de la función
int main()
{
    float a, b, c, pr;
    int k;
    for (k=1;k<=10;k++)
    {
        printf("\nIngrese una terna de valores : ");
        scanf("%f%f%f", &a, &b, &c);
        pr = promedio (a,b,c); //llamada a la función
        printf ("\n Promedio = %6.2f", pr);
    }
    return 0;
}

float promedio(float x, float y, float c) //desarrollo de la función
{
    float prome;
    prome = (x+y+c)/3;
    return (prome);
}
```

5.1 Declaración de una función – Prototipo

Esta declaración, que comprende solo la cabecera de la función y que se hace previa al comienzo del main (), le permite al compilador conocer el nombre de esta, conocer los parámetros y sus tipos de datos y luego poder chequear a éstos con los de la llamada y también el tipo del resultado. No se define el cuerpo de la función.

Una función no puede ser llamada si no se escribió su prototipo, salvo que la función sea declarada completa antes del main (), en dicho caso no es necesario el prototipo.

Si la función no recibe parámetros se pueden dejar los paréntesis vacíos o escribir la palabra void, que es el tipo de dato vacío, es decir, indica que no recibe nada. En el valor de retorno siempre hay que especificar un tipo de dato, si la función no retorna nada se DEBE especificar la palabra void. Si se olvida escribir un tipo de dato por defecto el compilador asume que la función retornará un int (entero). Es recomendable escribir siempre el tipo de dato de retorno para mayor claridad. Si se asigna un identificador a los parámetros en el prototipo los mismos serán ignorados.

```
[void ó tipo] nombre ([parámetros formales con tipos ó solo el tipo de datos]);
```

Ejemplos:

```
int suma (int, int, int);
void CalcularYMostrarPromedio(float, int);
char IngresarVocal();
int IngresaNumeroPar(void);
void MostraTitulo();
```

Orden de los prototipos: Si hay varias funciones y una función llama a otra debe ponerse primero el prototipo de la función más interna. Es decir, que: si la función "a" llama a la "b", debe definirse previamente la "b". Pero si a su vez la función "b" llama a una función "c" entonces el primer prototipo que se debe escribir es el de la función "c", luego el de la "b", y, por último, el de la "a". El desarrollo de las funciones puede hacerse en cualquier orden si los prototipos fueron escritos en la parte superior correctamente.

5.2 Llamada a una función.

Es similar al llamado a una función de biblioteca. La llamada significa la ejecución de la función.

Tiene la forma general:

```
[variable = ]   nombre   ( [ argumentos] ) ;
```

Por ejemplo, en esta llamada a la función potencia:

```
raíz = potencia (3.5 , a);
```

La variable raíz representa el lugar donde se guardará el resultado traído de la función; potencia es el nombre asignado a la función y los valores 3.5 y a, son los argumentos y son los valores que se envían a la función para ser asignados a sus parámetros formales. Ambas listas de parámetros deben coincidir en cantidad y tipo.

Ejemplos:

```
sum = suma (n1, n2, n3);
sum = suma (n1, 10, n3);
CalcularYMostrarPromedio(suma, 5);
CalcularYMostrarPromedio(suma, cant);
CalcularYMostrarPromedio(100, 5);
vocal = IngresarVocal();
num= IngresaNúmeroPar();
MostraTitulo();
```

5.3 Desarrollo de una función

```
[void o tipo] nombre ([parámetros formales con tipos de datos])
{
    [ declaración de variables locales]
    sentencias;
    [ return expresión; ]
}
```

Tipo: si no se indica la palabra void significa que la función retorna UN VALOR, que puede ser de cualquier tipo, menos array (esto se verá en unidades posteriores). Por omisión es int

Nombre: es un identificador del nombre de la función

Parámetros formales: Los parámetros formales son las variables que reciben los valores pasados en la llamada a la función. Si no se reciben valores se deja vacía o se coloca la palabra void. Por cada parámetro formal se debe especificar el tipo de dato y el nombre.

Declaraciones: si se declaran variables, éstas serán LOCALES de la función.

Sentencias: Estas son las instrucciones que forman el cuerpo de la función

Valor retorna por una función: Cada función puede retornar un solo valor cuyo tipo se indica en la cabecera de la función. Este valor para retornar se debe indicar en la sentencia return puede ser una constante una variable o una expresión.

Ejemplos:

```
return 1;      //retorna siempre el valor 1
return a+b;    //retorna la suma del contenido de las variables a y b
return suma;   //retorna el contenido de la variable suma
```

6. Metodologías para la programación modular

La programación modular permite separar el problema en subproblemas menores. Es decir, que parte de la lógica se puede delegar para que sea resuelta por una función lo que permite encarar el diseño de un algoritmo de dos formas distintas:

Botton-Up (ascendente)

Esta metodología se basa en comenzar a diseñar el algoritmo desarrollando primero las funciones o viendo que funciones se tienen para luego con esas funciones armar el diseño del programa principal. Es decir, que parte de lo particular para luego generar una solución al problema.

Top-Down (descendente)

Se realiza un diseño general del sistema centrándose en la lógica del programa principal, delegando algunas tareas en funciones que serán desarrolladas luego. Es decir, que primero se hace un diseño general y luego se va a lo particular especificando las partes de la solución que falten.



Elementos de Programación

UNIDAD 7. ARRAYS

INDICE

| | | |
|------------|--|-----------|
| 1. | <i>INTRODUCCIÓN</i> | 2 |
| 2. | <i>VECTORES</i> | 2 |
| 3. | <i>INICIALIZACIÓN DE VECTORES</i> | 6 |
| 4. | <i>ARRAYS COMO PARÁMETROS DE FUNCIONES</i> | 7 |
| 5. | <i>BÚSQUEDA EN VECTORES</i> | 11 |
| 6. | <i>CARGA DE UN VECTOR SIN ADMITIR VALORES REPETIDOS</i> | 13 |
| 7. | <i>MÁXIMOS Y MÍNIMOS MÚLTIPLES</i> | 13 |
| 8. | <i>VECTORES PARALELOS</i> | 16 |
| 9. | <i>ORDENAMIENTO DE VECTORES</i> | 17 |
| 10. | <i>ORDENAMIENTO DE VECTORES PARALELOS</i> | 22 |
| 11. | <i>MATRICES (ARRAYS BIDIMENSIONALES)</i> | 23 |
| 11.1 | DECLARACIÓN | 23 |
| 11.2 | INICIALIZACIÓN | 23 |
| 12. | <i>MATRICES COMO PARÁMETROS DE FUNCIONES</i> | 24 |
| 13. | <i>MANIPULACIÓN DE MATRICES</i> | 24 |
| 13.1 | RECORRIDO POR FILAS | 25 |
| 13.2 | RECORRIDO POR COLUMNAS | 26 |
| 13.3 | ACCESO DIRECTO | 27 |
| 13.4 | SUMA POR FILAS | 28 |
| 13.5 | SUMA POR COLUMNAS | 29 |

UNIDAD 7 - Arrays

OBJETIVOS: Utilizar arrays en los programas, tanto vectores y matrices. Generar y mostrar tablas y listados. Realizar búsquedas y ordenar los datos.

1. Introducción

Un array es un conjunto finito de elementos del mismo tipo de dato, almacenados en posiciones consecutivas de memoria. Es una forma de declarar muchas variables del mismo tipo que pueden ser referenciadas por un nombre común y subíndices haciendo que el acceso a cada una de las variables pueda generalizarse mediante estructuras repetitivas.

El tipo de dato de los elementos se denomina tipo base del array. Los arrays en el lenguaje C pueden ser de una o más dimensiones. Dentro de esta materia veremos los arrays de una dimensión llamados vectores y los de dos dimensiones llamados matrices.

El tamaño de los arrays debe ser definido al escribir el programa, es decir, que se debe conocer o estimar la cantidad máxima de elementos que va a necesitar almacenar para así dar un tamaño adecuado a los arrays.

En el lenguaje C los arrays se definen de forma similar a una variable poniendo el tipo y el identificador, pero se agrega luego del mismo, entre corchetes, la cantidad de elementos que contendrá el array en cada una de sus dimensiones. Por ejemplo:

```
int a [10]; //define un array de una dimensión (vector) que puede almacenar 10 variables enteras
```

```
int b [5][10]; // define un array de dos dimensiones (matriz) que puede almacenar en total 50 elementos.
```

Las variables del tipo array en el lenguaje C en realidad guardan una dirección de memoria. Esa dirección de memoria corresponde a la dirección del primer elemento del conjunto de datos definido. Luego, mediante los subíndices se calcula la dirección del elemento puntual al que se quiere acceder. En el lenguaje C, las variables que guardan direcciones de memorias se denominan punteros, porque al contener una dirección de memoria pueden “apuntar” a otra variable, es decir guardar la dirección, la referencia de donde se encuentra otra variable. El tema de punteros está fuera del alcance de esta materia. Solo debe saber entonces que el identificador del vector guarda la dirección de memoria de inicio de este ya que es un puntero denominado puntero estático, debido a que esa dirección no se puede cambiar una vez definida. Esto dará un comportamiento particular cuando se envíe un array como parámetro de función como se verá más adelante.

2. Vectores

Los vectores son arrays de una dimensión en el cual cada elemento se identifica con el nombre del conjunto y un subíndice que determina su ubicación relativa en el mismo.

Para definir un vector se utiliza la siguiente notación:

```
tipo_de_dato_base nombre_del_vector [cantidad de elementos];
```

Donde **cantidad de elementos** es una constante que define la capacidad del vector, es decir, la cantidad máxima de elementos que puede almacenar.

En la declaración:

```
int ve [100]
```

ve es el nombre de un vector de 100 elementos cuyo tipo base es int (o sea, un vector de 100 enteros).

El acceso a cada elemento del vector se formaliza utilizando el nombre genérico (nombre_del_vector) seguido del subíndice encerrado entre corchetes. Por ejemplo, ve[i] hace referencia al elemento i del vector. Luego, la información almacenada puede recuperarse tanto en forma secuencial (asignando valores al subíndice desde 0 hasta la capacidad -1) como en forma directa con un valor arbitrario del subíndice que esté dentro de dicho rango.

El subíndice define el desplazamiento del elemento con respecto al inicio del vector; puede ser una constante, una variable, una expresión o, inclusive, una función en la medida que resulte ser un entero comprendido entre 0 y la capacidad del vector menos 1, dado que representa el desplazamiento del elemento con respecto al inicio del vector.

Ej.: VECTOR [5], VECTOR[M], VECTOR [M – K + 1], siempre que 5, M y M – K + 1, respectivamente, sean un entero entre 0 (cero) y cantidad de elementos - 1.

Podemos representar gráficamente un vector int v [12] de la siguiente manera:

| | | | | | | | | | | | | |
|---|----|----|---|-----|---|----|---|-----|----|---|----|----|
| v | 23 | 12 | 5 | -57 | 0 | 45 | 1 | -96 | 32 | 7 | 10 | 30 |
|---|----|----|---|-----|---|----|---|-----|----|---|----|----|

v [0]: 23; v [4]: 0; v [7]: -96; El último elemento es v [11]: 30

Habitualmente, no se conoce exactamente cuántos elementos han de procesarse en un determinado programa y, además, en distintas ejecuciones de dicho programa, pueden procesarse diferentes volúmenes. Por ejemplo, si se define un vector para almacenar datos sobre los alumnos de un curso es evidente que distintos cursos tienen distinta cantidad de alumnos; en esos casos, se define un vector con capacidad suficiente para almacenar la información del curso más numeroso. Este dato suele no ser preciso por lo que generalmente se sobredimensiona el array (conviene no exagerar en el sobredimensionamiento pues esto significa mantener memoria ociosa). Así, si en una universidad se manejan cursos de 50 ó 60 alumnos, podría definirse el array de 100 elementos. Aún así, es necesario controlar, cuando se agregan elementos, que no se supere esa capacidad: Si quisieramos agregar el elemento 101, éste ocuparía memoria que no está reservada para el conjunto con lo cual pueden destruirse otras variables del programa.

El lenguaje C no chequea los límites de los arrays. Por lo tanto, es responsabilidad del programador cuidar que el programa se mantenga siempre dentro de los límites definidos.

Debe mantenerse siempre una variable donde guardar la cantidad de elementos que realmente tiene almacenado el vector.

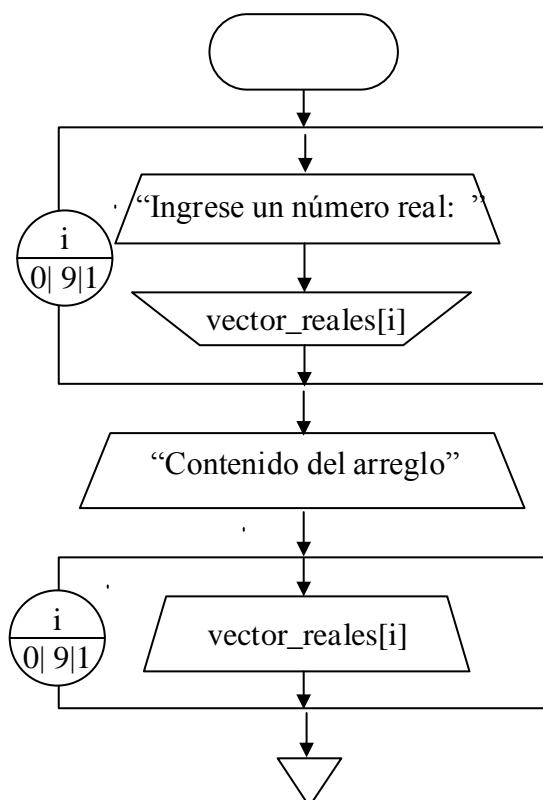
```
int v [10]; int cant_elem;
```

| | | | | | | | | | | | |
|----|----|---|---|----|----|-----|---|---|---|---|--|
| v: | 23 | 1 | 0 | 38 | 15 | 227 | 4 | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |

cant_elem 7

Note que en las posiciones 7, 8 y 9, si bien en el gráfico no se especificó contenido alguno por ser éste irrelevante, el mismo existe: siempre hay algo en cualquier dirección de memoria y, obviamente, por ser un vector de enteros, lo que hay se interpretará como un entero. Esto significa que si se recorre el array desde 0 hasta `cant_elem -1`, siempre se encontraran valores enteros, aunque no se hayan colocado en el transcurso del programa, esos datos no los debemos tener en cuenta. Decimos de forma genérica que en esas posiciones hay “basura” ya que son datos no controlados.

Ejemplo 1: Generar un vector de 10 números reales leyendo dichos valores del teclado y luego mostrarlo:



Codificación en C

```
#include <stdio.h>
int main()
{
    float vector_reales[10];
    int i;
    // Lee los 10 números reales y los ubica secuencialmente en el array
    for(i=0; i<=9; i++)
    {
        printf("\nIngrese un numero real: ");
        scanf("%f", &vector_reales[i]);
    }
    // Muestra el contenido del array
    printf("\n Contenido del array\n");
    for(i=0; i<=9; i++)
    {
        printf("%6.2f\t", vector_reales[i]);
    }
}
```

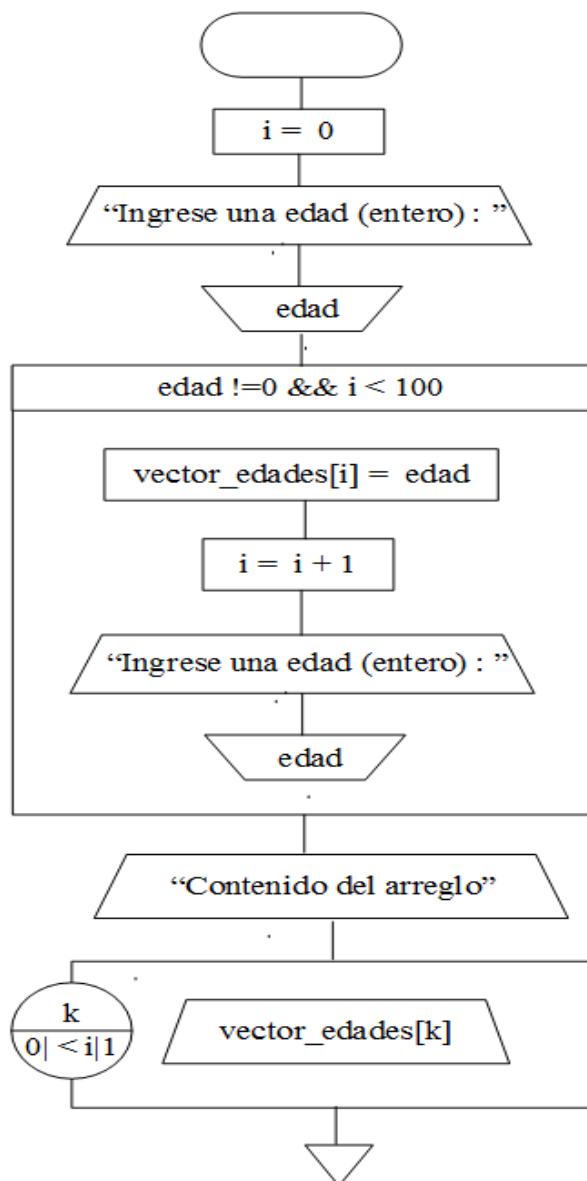
La salida de este programa es:

```
Ingrese un numero real: 23
Ingrese un numero real: 31
Ingrese un numero real: 3.25
Ingrese un numero real: .98
Ingrese un numero real: 5.5
Ingrese un numero real: 678
Ingrese un numero real: 500.60
Ingrese un numero real: 12
Ingrese un numero real: 0
Ingrese un numero real: 1
```

```
Contenido del array
23.0 31.00 3.25 0.98 5.50 678.00 500.60 12.00 0.00 1.00
```

Ejemplo 2: Generar un vector con edades de los alumnos de un curso que se ingresan por teclado. Se estima que los cursos no tienen más de 100 alumnos. La carga finaliza con una edad igual a 0.

En este caso, no se conoce cuántos datos se van a leer; se asume que, como máximo, hay 100. Se define entonces un vector de 100 enteros, pero debe verificarse en la carga de la información que no se supere dicho valor.



Nota importante: si bien puede ingresarse un elemento directamente en el array, en este caso no debe procederse, así pues, si se ingresaran más de 100 elementos, el elemento 101 se colocaría en la posición 100 del vector, que es memoria no reservada para el mismo. Luego, se ingresa el dato en una variable auxiliar y, una vez confirmada la disponibilidad de espacio, se lo pone en el array.

Codificación en C

```
#include <stdio.h>

int main()
{
    int vector_edades[100];
    int edad, i, k;
    i=0;
    printf("\nIngrese una edad (entero): ");
    scanf("%d", &edad);
    while (edad != 0 && i< 100)
    {
        vector_edades[i]=edad;
        i++;
        printf("\nIngrese una edad (entero): ");
        scanf("%d", &edad);
    }

    /* queda el array armado con, a lo sumo, 100 edades.
       La cantidad real quedo en i */

    printf("\n Contenido del array\n");
    for(k=0; k<i; k++)
    {
        printf("%5d", vector_edades[k]);
    }
    return 0;
}
```

3. Inicialización de Vectores

Al momento de declarar una variable del tipo vector es posible asignarle valores, para ello se pueden detallar cada uno de sus elementos. Por ejemplo, la instrucción:

```
int vec [5] = {2,52,100,58,18};
```

Define un vector de 5 posiciones donde en cada lugar ya tiene un número asignado, quedando el vector de la siguiente manera

| | | | | | |
|-----|---|----|-----|----|----|
| Vec | 2 | 52 | 100 | 58 | 18 |
|-----|---|----|-----|----|----|

El mismo resultado se puede obtener con la siguiente instrucción:

```
int vec [] = {2,52,100,58,18};
```

Dejar vacío el tamaño del vector al declarar memoria SOLO es válido si se detallan todos sus elementos ya que de forma automática le asignará el tamaño según la cantidad de elementos especificados en este caso será un vector de 5 posiciones.

Un caso muy habitual es utilizar cada posición de un vector como un contador o un acumulador, en dicho caso y para no poner muchos ceros como datos iniciales se puede escribir:

```
int vCont[10] = {0};
```

Esta instrucción crea un vector de 10 posiciones y pone todas las posiciones del mismo en 0. De esta forma rápidamente se puede poner en 0 un vector de cualquier tamaño.

Pero supongamos ahora que necesitamos que cada posición del vector se va a utilizar para calcular una productoria (es decir el contenido se lo multiplica por otro valor y se lo vuelve a guardar). En dicho caso será necesario que todas las posiciones del vector comiencen en 1 y no en 0 ya que si lo inicializamos en 0 al multiplicarlo por sí mismo siempre diera 0. Si escribimos esta instrucción:

```
int vProd[10] = {1};
```

Obtendríamos el siguiente resultado:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|

Es decir que pone el 1 en la primera posición, pero el resto como no está especificadas la completa con 0. Entonces para inicializar un vector de 10 posiciones todo con 1 tendríamos que escribir:

```
int vProd[10] = {1,1,1,1,1,1,1,1,1,1};
```

Esto puede traer confusiones y más cuando son vectores aún más grandes. Lo recomendable en estos casos es recorrer el vector posición a posición y asignar el número 1 en cada una de ellas. El siguiente programa declara un vector de 10 elementos y lo inicializa todo con el valor 1.

```
int main()
{
    int vProd[10], i;
    for (i=0; i<10; i++)
        v[i] = 1;
}
```

Es importante recordar que la inicialización directa del vector (usando las llaves) SOLO puede utilizarse al definir la variable, por lo tanto, si en el medio del programa un vector tiene que ser puesto por ejemplo nuevamente en 0, se debe utilizar sí o sí el método de recorrerlo y asignarle a cada posición el dato deseado.

Si se define un vector de mayor cantidad de los datos que se inicializan, el resto de los datos se completan con 0. Este ejemplo:

```
int vec[10] = {1,2,1,8};
```

genera el siguiente vector:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|

4. Arrays como parámetros de funciones

Como se mencionó con anterioridad al definir una variable del tipo array (ya sea un vector o una matriz), esa variable guarda la dirección de memoria de inicio del conjunto de datos. Entonces cuando se envíe un array como parámetro de una función, lo que se envía y se copia en una variable local a la función es esa misma dirección, es decir, que dentro de la función se sigue referenciando a la misma dirección de memoria que está reservada para el conjunto de datos en el bloque desde el cual se invoca a la función. Esta particularidad hace que los cambios que se hagan sobre el array dentro de la

función se van a ver reflejados desde donde se llamó a la función ya que trabaja directamente sobre la memoria de este.

Para definir un vector como parámetro formal de una función se indica el tipo base, el nombre y luego un par de corchetes que es lo que identifica al parámetro como un array. Ejemplos:

```
int ve[], char vc[], float vf[]
```

No es necesario especificar el tamaño del vector ya que solo se copia la dirección de memoria de inicio y NO los datos del vector

Por ejemplo, si en el programa principal se define un vector de 5 elementos y se desea invocar una función para que solicite los datos por teclados y los guarde en el vector, a la función se puede enviar el vector y la cantidad de datos a completar. Una posible cabecera para dicha función sería: void CargarVector (int v [], int N). Donde V es la dirección de inicio del vector donde se van a guardar los datos y N es la cantidad de elementos a solicitar. Nótese que la función no retorna ningún valor (void) ya que los datos cargados en el vector dentro de la función se verán reflejados en el programa principal porque se trabaja directamente sobre la memoria de este. La Figura 1 muestra el esquema de memoria durante el pasaje de parámetros (las direcciones de memoria son ficticias ya que su largo depende de la arquitectura del procesador de la computadora donde se ejecute el programa. El dato 5 no sale de una variable de la memoria principal ya que se envía como una constante en la llamada a la función. Además, en la función se declara una variable local i que servirá de subíndice para recorrer el vector.

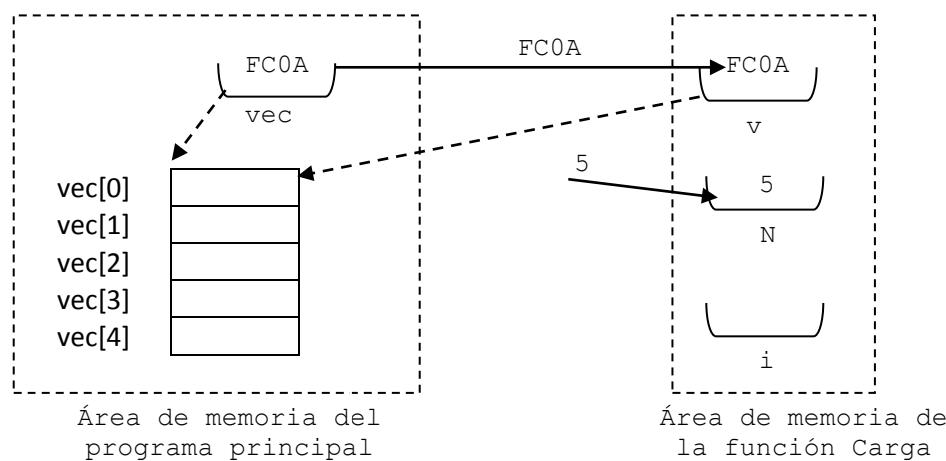
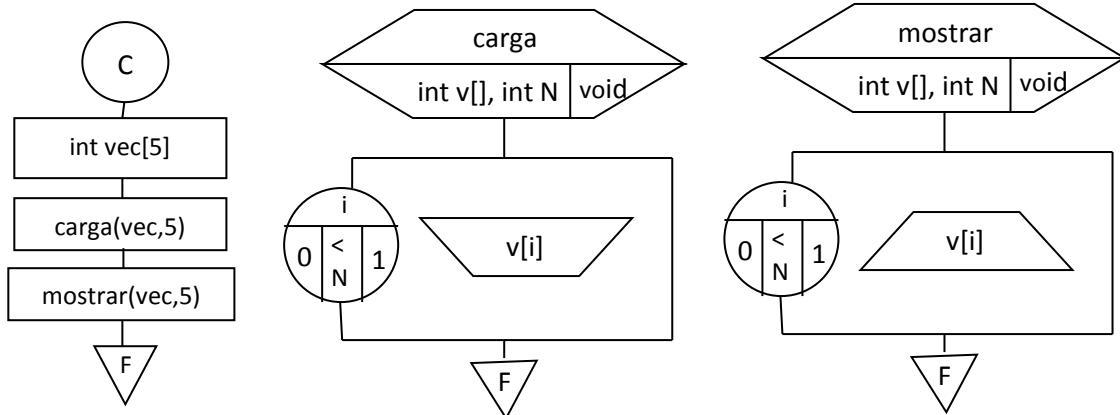


Figura 1: esquema de memoria al enviar un vector como parámetro a una función

Al enviar solo la dirección de memoria del vector, la misma función carga puede reutilizarse para cargar vectores de distinto tamaño siempre que sean del mismo tipo de dato. Esta función servirá igualmente para cargar un vector de 5, uno de 10 o uno de 1000 elementos, por ejemplo, simplemente se debe declarar una variable con la capacidad suficiente en el programa principal y enviar en N la cantidad de elementos a cargar.

A continuación, se muestra un programa que, utilizando dos funciones, carga un vector de 5 elementos y los muestra por pantalla (si bien en el diagrama no es necesario declarar las variables a utilizar, es recomendable hacerlo en el caso de los arrays para conocer la cantidad de datos disponibles).



Codificación en C:

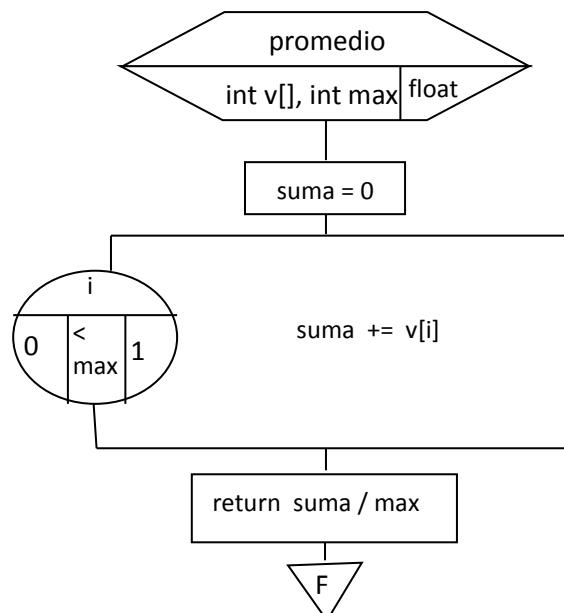
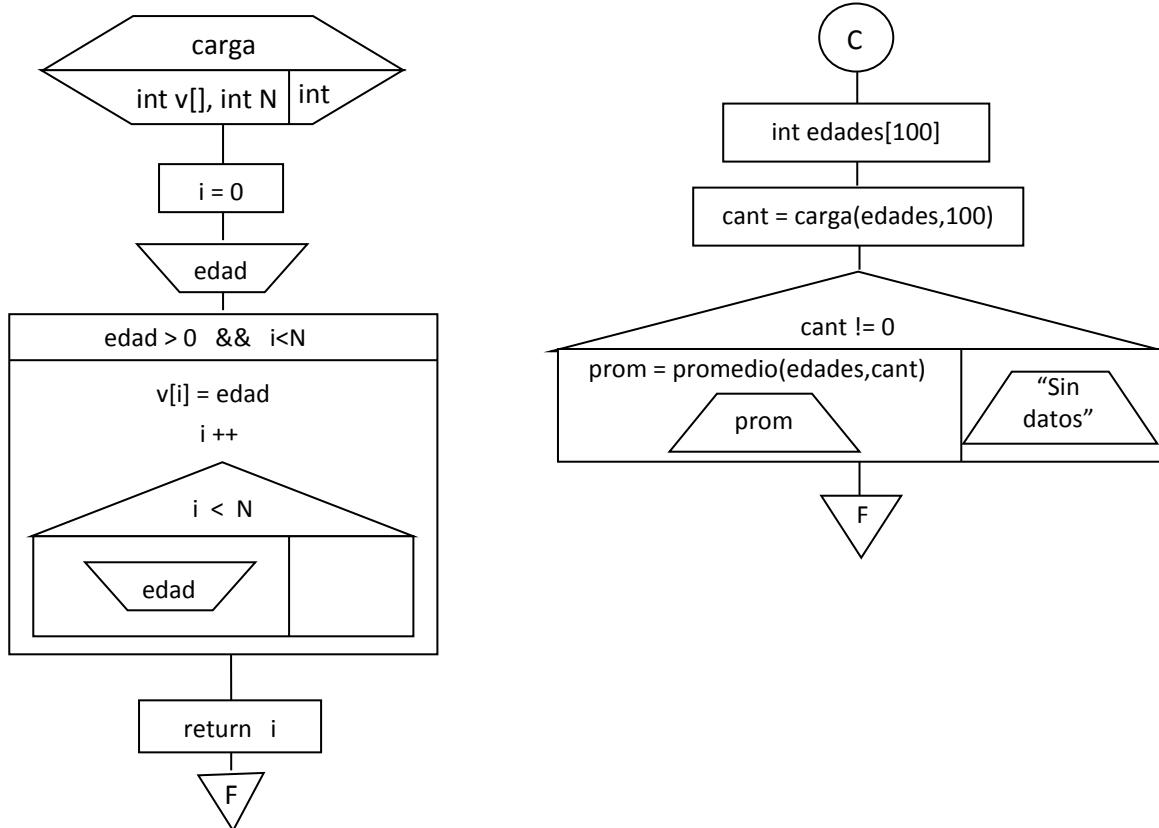
```
#include <stdio.h>
void carga(int[], int);
void mostrar(int[], int);
int main ()
{
    int vec[5];
    carga(vec, 5);
    mostrar(vec, 5);
    return 0;
}

void carga(int v[], int N)
{
    int i;
    for (i=0;i<N;i++)
    {
        printf("Ingrese un numero: ");
        scanf("%d",&v[i]);
    }
}

void mostrar(int v[], int N)
{
    int i;
    for (i=0;i<N;i++)
        printf("%d\n",v[i]);
}
```

Muchas veces no se sabe exactamente la cantidad de datos a ingresar por lo que se dimensiona el vector con un tamaño suficiente y se debe realizar una función de cargar que complete los datos sin pasarse del tamaño del vector, pero esta función debe además retornar la cantidad de elementos realmente ingresados.

A modo de ejemplo realizaremos el siguiente programa: ingresar las edades de los alumnos de un curso. Calcular luego el promedio de edades. No se sabe la cantidad exacta de alumnos, pero sí que no son más de 100. El ingreso de datos finaliza con una edad menor o igual a 0.



Como puede verse la función de carga retorna la cantidad de datos ingresada y luego este dato se pasa a la función que calcula el promedio para que no tome posiciones del vector que no tengan datos válidos. El promedio solo debe calcularse si se ingresaron datos, caso contrario se produciría una división por cero, generando un error de ejecución en el programa.

Dentro del ciclo repetitivo de la función de carga se agrega una condición para no solicitar una edad sino queda espacio en el vector para guardarla.

Código C:

```
#include <stdio.h>
int carga(int[], int);
float promedio (int[], int);
int main()
{
    int edades[100], cant;
    float prom;
    cant=carga(edades,100);
    if (cant!=0)
    {
        prom = promedio(edades, cant);
        printf("El promedio de edades del curso es %.2f", prom);
    }
    else
        printf("Sin Datos");
    return 0;
}

int carga(int ve[], int N)
{
    int i=0,edad;
    printf ("Ingrese la edad del alumno:");
    scanf("%d",&edad);
    while (edad>0 && i<N)
    {
        ve[i]=edad;
        i++;
        if (i<N) //evita ingresar un dato cuando no hay espacio para almacenarlo
        {
            printf ("Ingrese la edad del alumno:");
            scanf("%d",&edad);
        }
    }
    return i;
}

float promedio (int v[], int max)
{
    int suma=0,i;
    for (i=0;i<max;i++)
        suma+=v[i];
    return (float)suma/max;
}
```

5. Búsqueda en vectores

Frecuentemente es necesario buscar un determinado elemento en un vector, ya sea para determinar si el mismo está en el vector o para recuperar la posición en donde se encuentra.

Los datos del problema son:

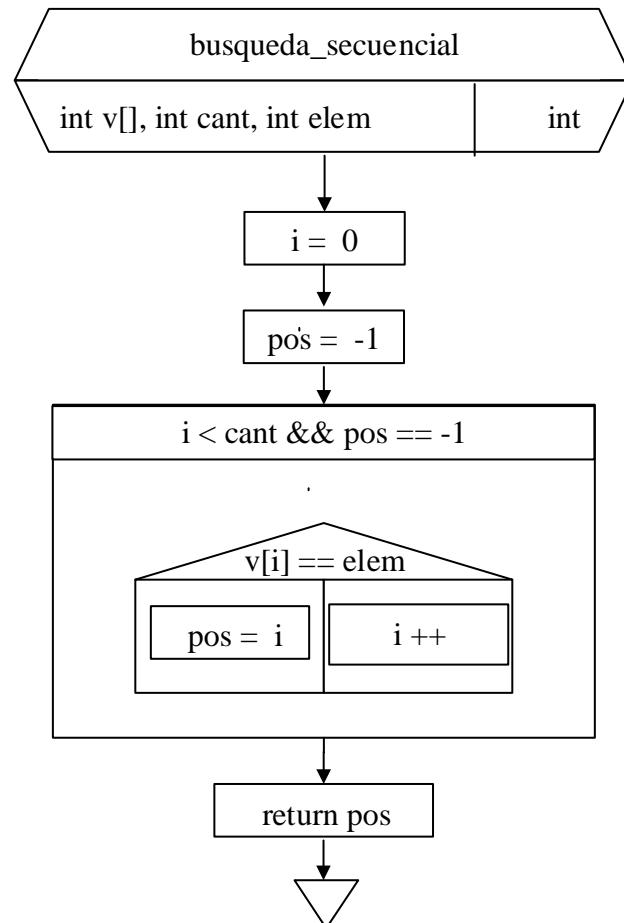
- El vector.
- La cantidad de elementos que tiene.
- El elemento para buscar.

Las salidas son:

- Un indicador de la existencia del elemento en el array
- La posición donde se encuentra.

Como la función solo puede retornar un valor, ambas salidas se unifican en una que represente, si existe, la posición donde se encuentra ($0 \leq \text{posición} \leq \text{cantidad de elementos}$) y si no existe retornará un -1 (valor inválido como posición).

Si bien existen distintos métodos de búsqueda, veremos el más sencillo de ellos que consiste en ir recorriendo uno a uno los elementos del vector hasta que encontremos el buscado. Este método se denomina **Búsqueda Secuencial**.

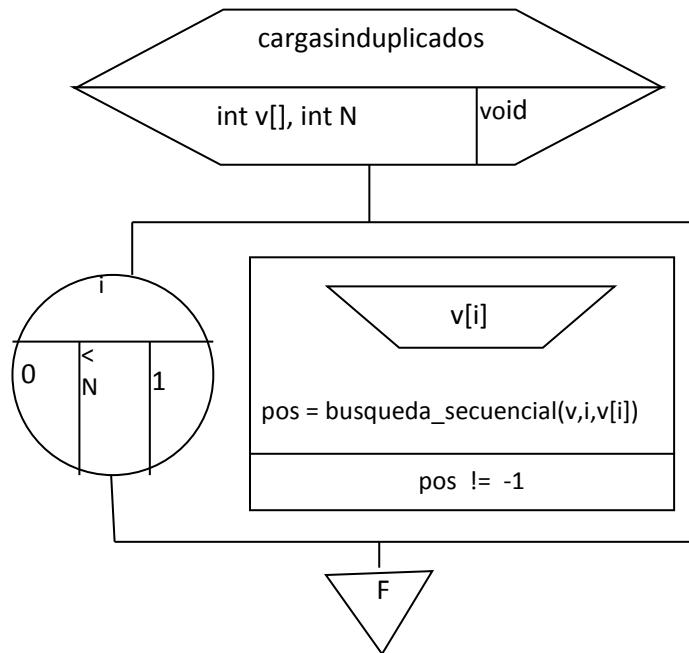


Codificación en C:

```
int busqueda_secuencial (int v[], int cant, int elem)
{
    int i, pos;
    i = 0;
    pos = -1;
    while(i < cant && pos == -1)
    {
        if(v[i] == elem)
            pos = i;
        else
            i++;
    }
    return pos;
}
```

6. Carga de un vector sin admitir valores repetidos

Utilizando la función de búsqueda es posible modificar la carga de un vector asegurándonos de que todos los valores ingresados sean diferentes. Para ello por cada vez que se ingrese un dato se buscará si el mismo ya está en el vector y si está se solicitará un dato diferente.



Puede verse que cada vez que se ingresa un dato, se busca si ya estaba en el vector y si estaba se vuelve a solicitar otro dato que se guarda en la misma posición del vector. No se avanza de posición del vector hasta que no se cargue un dato diferente al resto. La función de búsqueda se llama enviando la variable *i* como cantidad de datos del vector que indica cuantos se cargaron previamente. La primera vez se envía 0 como tamaño del vector por lo tanto dentro de la función de búsqueda ni siquiera ingresa al ciclo y retorna directamente -1, lo que es correcto ya que al no existir datos previos no puede haber duplicados. La segunda vez ya comparará si el segundo dato ingresado coincide con el primero, y así sucesivamente cada vez se hará la búsqueda con más posiciones del vector que ya fueron cargadas previamente.

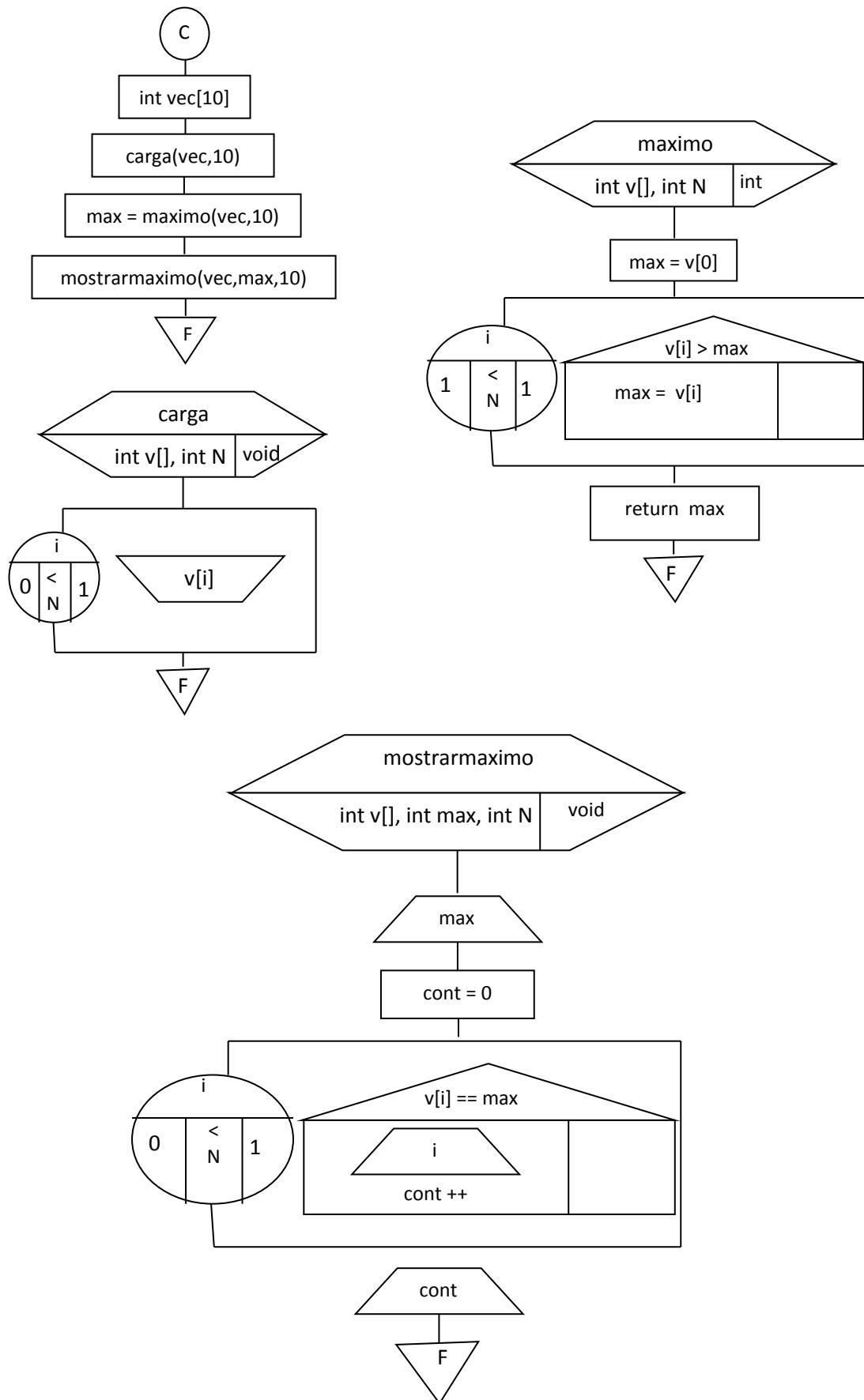
7. Máximos y Mínimos múltiples

Dado un conjunto de datos, muchas veces es necesario recuperar el valor más grande o el más chico del lote, para ello se realizará un algoritmo similar al ya visto en la unidad de estructuras de repetición donde el primer elemento del conjunto es tomado como referencia y luego comparado con el resto. La diferencia es que ahora los datos ya están todos almacenados en un vector y por lo tanto podremos no solo determinar el mayor valor del conjunto sino también determinar cuántas veces se repite y en qué posiciones del vector se encuentra.

El procedimiento consta de dos pasos:

1. Determinar el máximo o el mínimo valor del conjunto de datos
2. Volver a recorrer el conjunto de datos para ver cuántas veces se repite y donde se encuentra

El siguiente programa permite ingresar por teclado un vector de 10 elementos utilizando la función Carga ya realizada en el ejercicio anterior en la sección 4 de este documento, luego determina el valor máximo mediante una función y por último informa mediante otra función, la cantidad de repeticiones y las posiciones del máximo.



Código C:

```
#include <stdio.h>
void carga(int[], int);
int maximo(int[], int);
void mostrarMaximo(int[], int, int);
int main()
{
    int vec[10], max;
    carga(vec,10);
    max = maximo(vec,10);
    mostrarMaximo(vec,max,10);
    return 0;
}

void carga(int v[], int N)
{
    int i;
    for (i=0;i<N;i++)
    {
        printf("Ingrese un numero: ");
        scanf("%d",&v[i]);
    }
}

int maximo(int v[], int N)
{
    int max = v[0], i;
    for (i=1;i<N;i++) //se comienza en 1 ya que el primero se tomó como referencia
    {
        if (v[i]>max)
            max = v[i];
    }
    return max;
}

void mostrarMaximo(int v[], int max, int N)
{
    int cont=0,i;
    printf("El valor maximo es: %d y se encuentra en las siguientes
posiciones del vector:\n",max);
    for (i=0;i<N;i++)
    {
        if (v[i]==max)
        {
            printf("%d\n", i);
            cont++;
        }
    }
    printf ("El valor maximo se repite %d veces", cont);
}
```

Si en lugar del máximo se desea calcular el mínimo el procedimiento es similar. El primero se toma como referencia y luego se compara si alguno de los siguientes es menor al guardado como referencia.

8. Vectores Paralelos

Muchas veces para resolver un problema es necesario guardar más de un dato de una misma entidad y por lo tanto con un solo vector no será suficiente. Por ejemplo, si se desea ingresar la lista de precios de un negocio donde por cada producto se tenga el código y su precio, será necesario definir dos vectores uno que guarde el código del producto y un segundo vector que guarde los precios. Estos vectores estarán relacionados, ya que en la posición 0 del vector de códigos se guardará el código del producto y en la posición 0 del vector de precios se guardará el precio de ese mismo producto. Es decir que los vectores están relacionados guardando en los elementos con igual subíndice información de una misma entidad. A estos vectores se los denomina vectores paralelos o apareados.

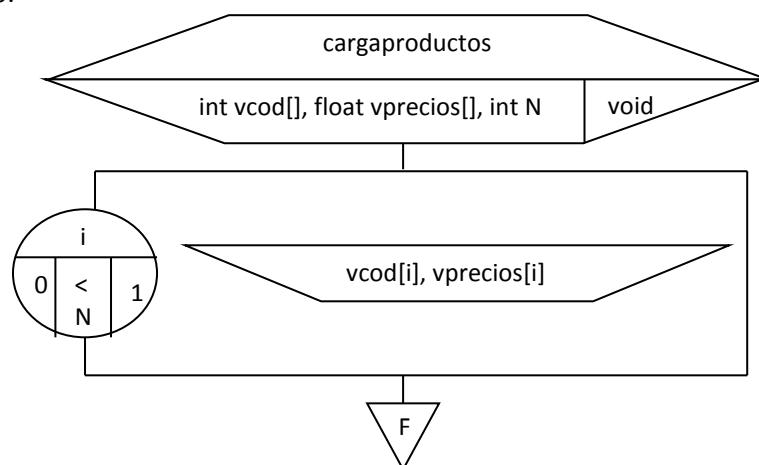
La Figura 2 muestra un ejemplo donde en vectores paralelos se cargó la información de los productos.

| int vcod[4] | float vprecios[4] |
|-------------|-------------------|
| 1500 | 10.55 |
| 5878 | 17.35 |
| 2565 | 5.40 |
| 1566 | 20.30 |

Figura 2: vectores en paralelo con código y precio de productos

El producto con código 1500 tiene un precio de 10.55 mientras que el producto con código 5878 tiene un precio de 17.35 y así sucesivamente siempre la información está relacionada en ambos vectores.

Al momento de realizar una función de carga para este tipo de problema se debe solicitar la información completa de la entidad y guardarla en los vectores, es decir que se ingresa primero el código y precio del primer producto luego código y precio del segundo y NO primero todos los códigos y luego todos los precios ya que de esa forma sería confuso para el usuario. A continuación, se muestra una función de carga de la información de los productos donde se solicita el código y el precio de cada uno.



```

void cargaProductos(int vcod[], float vprecios[], int N)
{
    int i;
    for (i=0;i<N;i++)
    {
        printf("Ingrese el código de producto: ");
        scanf("%d",&vcod[i]);
        printf("Ingrese el precio del producto %d: ", vcod[i]);
        scanf("%f",&vprecios[i]);
    }
}
  
```

9. Ordenamiento de vectores

Los métodos de ordenamiento son numerosos. Podemos considerar dos grandes grupos:

- Directos: Burbujeo, selección e inserción -
- Indirectos (avanzados): Shell, ordenación por mezcla.

En listas pequeñas, los métodos directos se comportan en forma eficiente, mientras que, en vectores con gran cantidad de elementos, se debe recurrir a métodos avanzados.

Veremos el método de ordenamiento por **burbujeo o intercambio**, primero en su versión más sencilla y luego la perfeccionaremos.

Supongamos que tenemos que ordenar en forma ascendente (de menor a mayor) un vector "a", mediante una función, conociendo sus elementos enteros y su dimensión. Una posible solución, sería comparar a [0] con a [1], intercambiándolos si están desordenados, lo mismo con a [1] y a [2], y así sucesivamente hasta llegar al último elemento.

Si nuestro vector está formado por los siguientes 5 elementos:

| | | | | |
|---|----|---|----|---|
| 7 | 26 | 8 | 15 | 9 |
| 0 | 1 | 2 | 3 | 4 |

Al comparar a[0] con a[1] no se producen cambios: $7 < 26$

| | | | | |
|---|----|---|----|---|
| 7 | 26 | 8 | 15 | 9 |
|---|----|---|----|---|

Al comparar a[1] con a[2] : intercambio el 26 con el 8

| | | | | |
|---|---|----|----|---|
| 7 | 8 | 26 | 15 | 9 |
|---|---|----|----|---|

Al comparar a[2] con a[3] : intercambio el 26 con el 15

| | | | | |
|---|---|----|----|---|
| 7 | 8 | 15 | 26 | 9 |
|---|---|----|----|---|

Al comparar a[3] con a[4] intercambio el 26 con el 9, resultando:

| | | | | |
|---|---|----|---|----|
| 7 | 8 | 15 | 9 | 26 |
|---|---|----|---|----|

De esta forma, al finalizar la primera pasada, en la última posición del array, ha quedado el elemento mayor. El resto de los elementos mayores, tienden a moverse, “burbujean” hacia la derecha. Se trata ahora de comenzar nuevamente con las comparaciones de los elementos (segunda pasada) sin necesidad de tratar el último, ubicado en a[4].

Al comparar a[0] con a[1] y a[1] con a[2] no se producen intercambios.

| | | | | |
|---|---|----|---|----|
| 7 | 8 | 15 | 9 | 26 |
|---|---|----|---|----|

Al comparar a[2] con a[3] intercambio el 15 con el 9, resultando:

| | | | | |
|---|---|---|----|----|
| 7 | 8 | 9 | 15 | 26 |
|---|---|---|----|----|

Al finalizar la segunda pasada, con una comparación menos, ya obtuvimos en la anteúltima posición del array el mayor entre los restantes (el valor 15).

Así sucesivamente, en cada ciclo la cantidad de comparaciones disminuye en 1.

De esta forma en la tercera pasada se comparará al elemento $a[0]$ con $a[1]$ y al $a[1]$ con $a[2]$. En nuestro ejemplo no se producirán cambios:

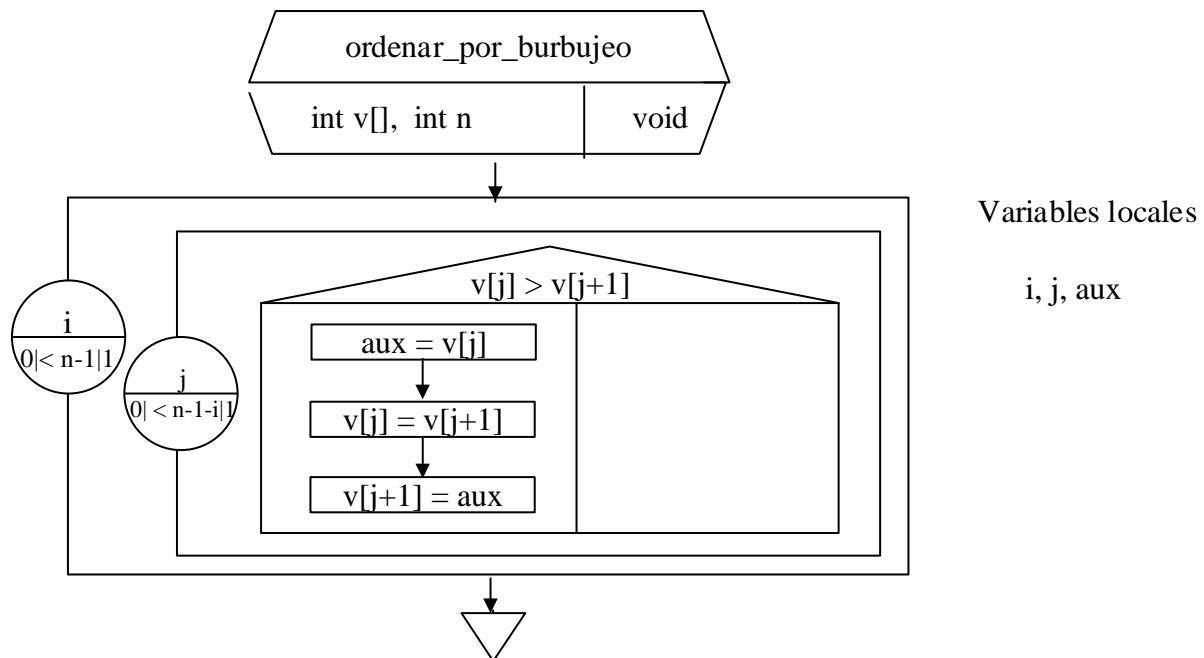
| | | | | |
|---|---|---|----|----|
| 7 | 8 | 9 | 15 | 26 |
|---|---|---|----|----|

En la cuarta y última pasada, dado que nuestro vector “a” contiene 5 elementos, tampoco se efectuarán cambios al comparar $a[0]$ con $a[1]$, asegurándonos que nuestro vector, ya resultó ordenado:

| | | | | |
|---|---|---|----|----|
| 7 | 8 | 9 | 15 | 26 |
|---|---|---|----|----|

En resumen, para ordenar un vector de n elementos, son necesarias realizar como máximas $n-1$ pasadas, y en cada oportunidad una comparación menos.

El algoritmo siguiente resuelve según el método descripto



Codificación en C

```
void ordenar_por_burbujeo (int v[], int n)
{
    int i, j, aux;
    for (i=0; i<n-1; i++)
        for(j=0; j<n-1-i; j++)
            if(v[j] > v[j+1])
            {
                aux=v[j];
                v[j] = v[j+1];
                v[j+1]= aux;
            }
}
```

Volviendo a nuestro vector original:

| | | | | |
|---|----|---|----|---|
| 7 | 26 | 8 | 15 | 9 |
|---|----|---|----|---|

recordamos que después de la primer pasada, el vector resultó:

| | | | | |
|---|---|----|---|----|
| 7 | 8 | 15 | 9 | 26 |
|---|---|----|---|----|

y después de la segunda pasada el vector quedó ordenado:

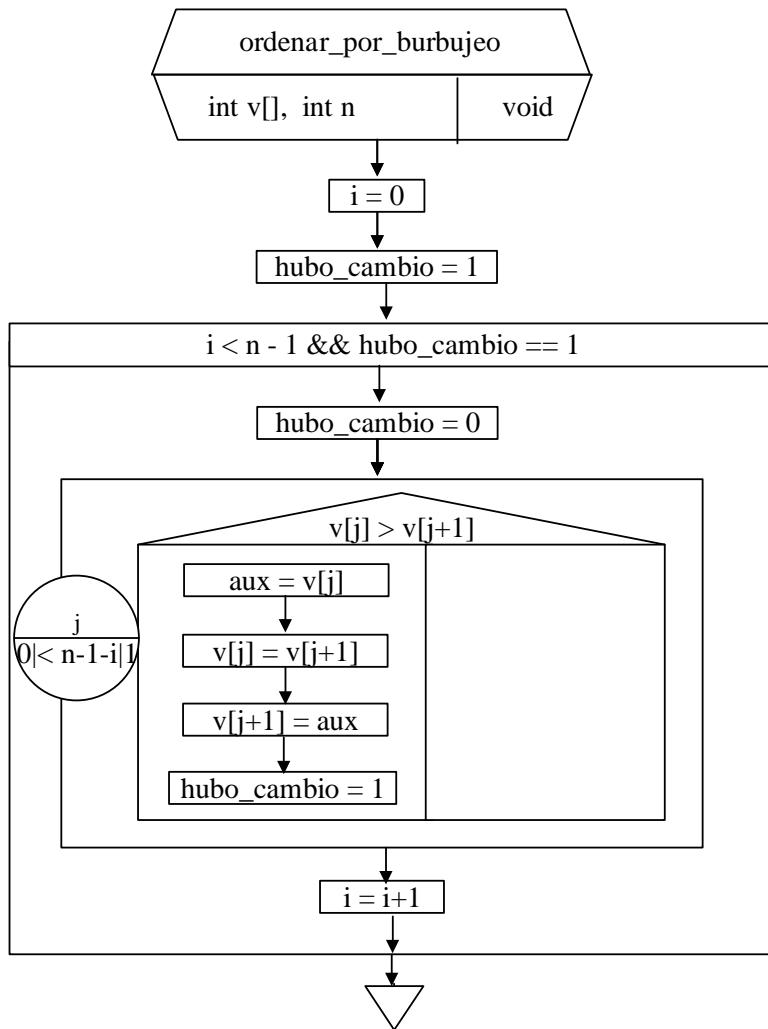
| | | | | |
|---|---|---|----|----|
| 7 | 8 | 9 | 15 | 26 |
|---|---|---|----|----|

Por lo tanto, ¿es necesario continuar con el proceso, si anteriormente detectamos que el vector ya está ordenado?

Método de burbuja mejorado I

La optimización con respecto a nuestra primera versión consiste en interrumpir el proceso, si detectamos que no se producen cambios después de una pasada.

El mismo se puede detectar cuando no se producen intercambios luego de una pasada. Usaremos una variable `hubo_cambio` que se inicializa en falso (0) antes de cada pasada y se activa en verdadero (1) cuando se produce un cambio. Inicialmente ese variable se pone en 1 para que ingrese al ciclo.



Codificación en C

```

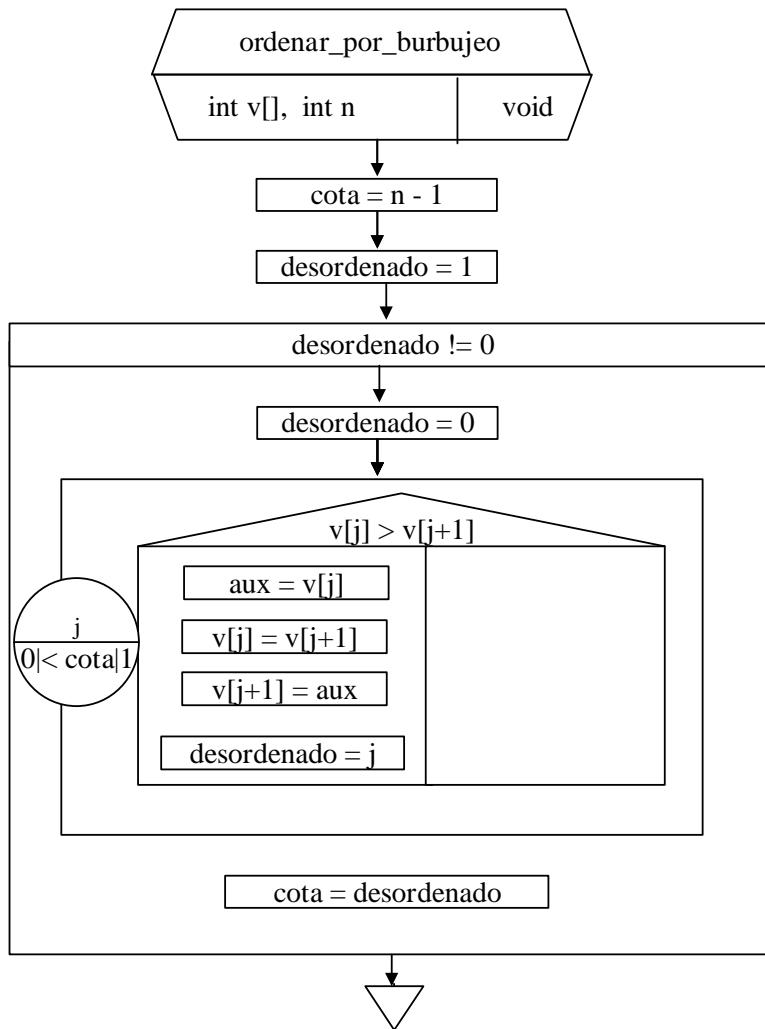
void ordenar_por_burbujeo (int v[], int n)
{
    int i, j, aux, hubo_cambio;
    i=0;
    hubo_cambio=1; // verdadero, para que entre en la primera iteración
    while(i < n-1 && hubo_cambio==1)
    {
        hubo_cambio=0;
        for(j = 0; j < n-i-1; j++)
            if(v[j] > v[j+1])
            {
                aux=v[j];
                v[j] = v[j+1];
                v[j+1]= aux;
                hubo_cambio=1;
            }
        i++;
    }
}

```

Método de burbuja mejorado II

Dado que también tenemos la posibilidad de conocer la posición donde se realizó el último intercambio, reciclaremos hasta esa posición optimizando el método anterior.

Es decir, si después de una pasada se intercambiaron valores, se repite el proceso, pero sólo hasta una posición anterior a la del último cambio porque el resto ya está ordenado.



```

void ordenar_por_burbujeo(int v[], int n)
{
    int j, cota, aux, desordenado;
    cota = n-1;
    desordenado =1;
    while (desordenado !=0)
    {
        desordenado=0;
        for (j=0;j<cota;j++)
            if (v[j]>v[j+1])
            {
                aux = v[j];
                v[j]=v[j+1];
                v[j+1] = aux;
                desordenado = j;
            }
        cota = desordenado;
    }
}

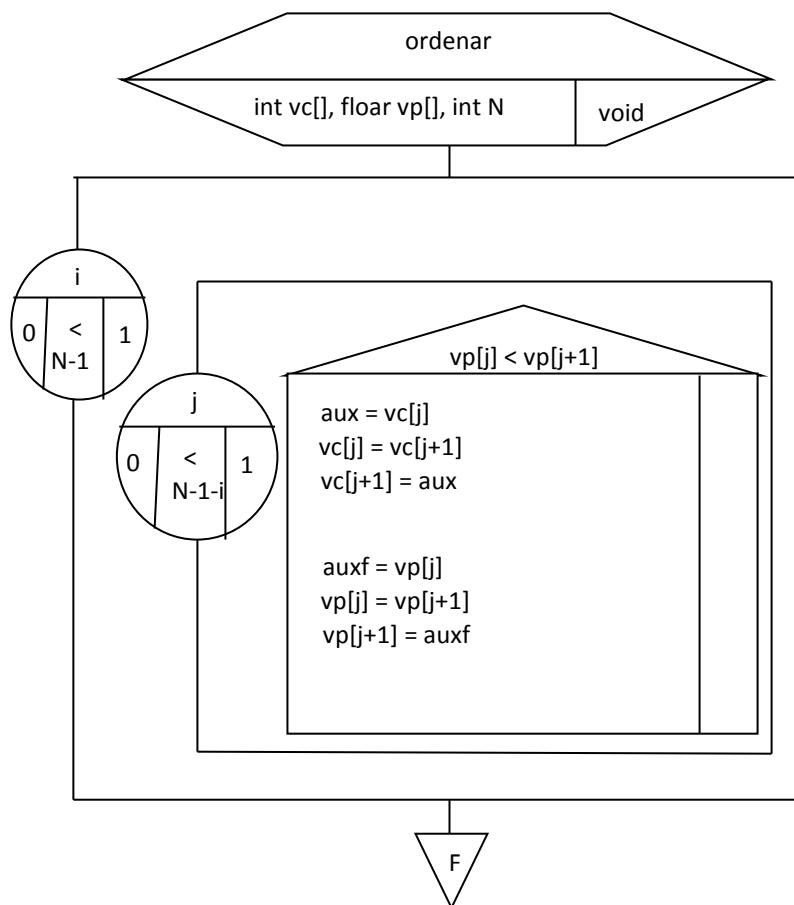
```

Este algoritmo se repite mientras el vector esté desordenado, para ello se utiliza una variable llamada desordenado que se inicializa en 1 indicando que el vector está desordenado (recordemos que en el lenguaje C cualquier valor distinto de 0 es verdadero y 0 es falso por lo tanto la condición del while

podría escribirse directamente poniendo solo el nombre de la variable, consultado de esa forma por su valor lógico). Una vez que ingresa al while se debe corroborar si efectivamente el vector está ordenado o no, para ello se asume ordenado y por eso se pone en 0 la variable desordenado. Si dentro de las comparaciones se realiza algún intercambio significa que no estaba ordenado y se guarda la posición del último intercambio que luego será asignada a cota para que la siguiente pasada sea más eficiente y no recorra posiciones ya ordenadas.

10. Ordenamiento de vectores paralelos

Cuando se tienen vectores paralelos al ordenar un vector se debe también realizar el intercambio en el o los vectores que guardan información relacionada. Volviendo al ejemplo anterior donde se tenía un vector con los códigos de producto y otros con los precios, si se quiere mostrar un listado ordenado por precio de mayor a menor, si los intercambios solo se realizan sobre el vector de precios quedarían mezclados los códigos de producto con un precio que no les corresponde. Al ordenar vectores en paralelo el algoritmo se aplica sobre el vector que se quiere ordenar, pero al momento de realizar el intercambio se lo hace también, sobre los vectores relacionados.



En este caso se necesitaron dos variables auxiliares diferentes ya que los vectores son de distinto tipo, si ambos fueran del mismo tiempo entonces se puede reutilizar la misma variable como auxiliar para realizar el intercambio.

11. Matrices (arrays bidimensionales)

Recordemos la definición de arrays: Un array es un conjunto finito de elementos del mismo tipo de datos, almacenados en posiciones contiguas de memoria. Bien, un array bidimensional es un array en el cual cada elemento es a su vez otro array. Un array llamado B, el cual consiste en M elementos, cada uno de los cuales es un array de N elementos de un cierto tipo T se puede representar como una tabla de M filas (o renglones) y N columnas, como se muestra en la figura 3:

| | 0 | 1 | . | . | j | . | N-1 |
|-----|---|---|---|---|---|---|-----|
| 0 | | | | | | | |
| 1 | | | | | | | |
| . | | | | | | | |
| i | | | | | X | | |
| . | | | | | | | |
| M-1 | | | | | | | |

Figura 3: Esquema de representación de una matriz

Para identificar a cada elemento es necesario utilizar dos subíndices: uno para identificar la fila y otro para la columna. El primer subíndice hace referencia a la fila y el segundo a la columna. Así, el elemento marcado con X en la figura se identifica como B [i] [j]

A este tipo de estructura se lo llama **matriz**

11.1 Declaración

Formato:

```
tipo_dato nombre [cantidad de filas] [cantidad de columnas];
```

Ejemplo:

```
float tabla[20][10];
```

De esta forma, se reservan 200 posiciones en memoria, (producto de 20 filas por 10 columnas), donde se podrán almacenar valores float.

El primer índice varía entre 0 y 19, y el segundo entre 0 y 9.

Se pueden definir los tamaños de los índices de las matrices, utilizando constantes, pero NO variables. Siempre el tamaño debe estar establecido desde la codificación del programa.

11.2 Inicialización

También es factible inicializar una matriz.

Ejemplo 1:

```
int matriz [2][3]={1,2,3,4,5,6};
```

Los valores son asignados por filas. Primero se asignan los elementos de la primera fila, luego los elementos de la segunda fila, y así sucesivamente. En el ejemplo la matriz quedaría con los siguientes valores:

```
matriz[0][0]=1    matriz[0][1]=2    matriz[0][2]=3
```

```
matriz[1][0]=4      matriz[1][1]=5      matriz[1][2]=6
```

Ejemplo 2:

```
int matriz [] [3]={1,2,3,4,5,6};
```

En este ejemplo no se especifican la cantidad de filas. Las mismas se calculan en forma automática según la cantidad de elementos con las que inicializa la matriz. El parámetro de la cantidad de columnas NUNCA puede dejarse vacío.

Ejemplo 3:

```
int matriz [2] [3] ={  
    {1,2,3},  
    {4,5,6}  
};
```

Otra nomenclatura posible para inicializar una matriz. Esta forma es visualmente más adecuada que las anteriores ya que referencia al formato de una tabla y se puede visualizar rápidamente la ubicación de cada uno de los valores.

Muchas veces será necesario inicializar la matriz en 0 si es que se desea utilizarla para contar o acumular. Para esos casos y solo al momento de declarar la variable es posible inicializarla en 0 de la siguiente forma:

```
int matriz [2] [3] ={{0}};
```

Esta nomenclatura solo es válida para el 0 y al momento de declarar la variable.

12. Matrices como parámetros de funciones

Como se explicó anteriormente la variable del tipo array (ya sea un vector o una matriz) contienen la dirección de memoria del primer dato, por lo tanto, al enviar como parámetro a una función un array, los cambios que se hagan dentro de la función se verán reflejados desde donde se la llamó ya que se trabaja con la misma dirección de memoria.

Al enviar un vector se podía dejar sin completar el tamaño en la especificación de los parámetros de la función ya que al ser una dirección de memoria y el lenguaje no valida límites ese dato es irrelevante. En cambio, al enviar una matriz al tener más de una dimensión SIEMPRE se debe indicar la cantidad de columnas, pudiendo dejar vacío solo la cantidad de filas. Esto se debe a que el compilador necesita saber cómo están organizados los datos para hacer los cálculos de a qué dirección de memoria dirigirse cuando se escriben los subíndices de la matriz. Indicándole la cantidad de columnas y debido a que los datos se guardan en forma consecutiva, puede calcular cuantas direcciones de memoria debe desplazarse hasta llegar a la fila indicada.

13. Manipulación de Matrices

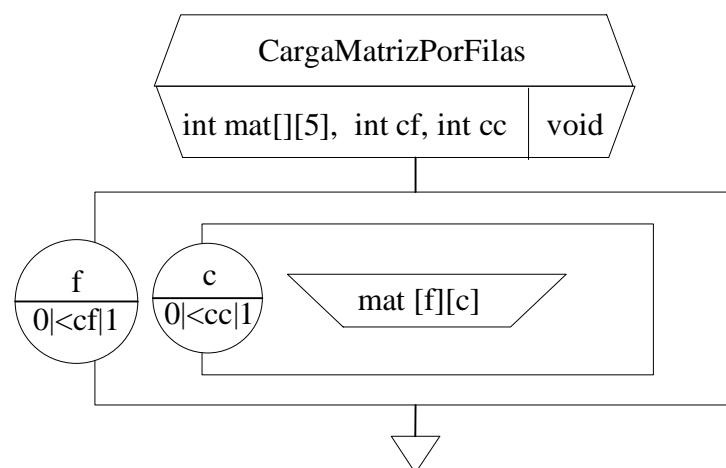
El orden más natural de procesar los vectores es el orden secuencial: del primero al último elemento. En el caso de los arrays bidimensionales, existen diferentes órdenes para su recorrido. Los más usuales son: recorrido por filas y recorrido por columnas. En el primero, se recorre la primera fila (desde la primera columna hasta la última), luego la segunda fila, etc.; En el segundo, se recorre la primera columna (desde la primera fila hasta la última), luego la segunda columna, etc.

Por supuesto, también se puede acceder directamente a un elemento (característica de todos los arrays).

13.1 Recorrido por filas

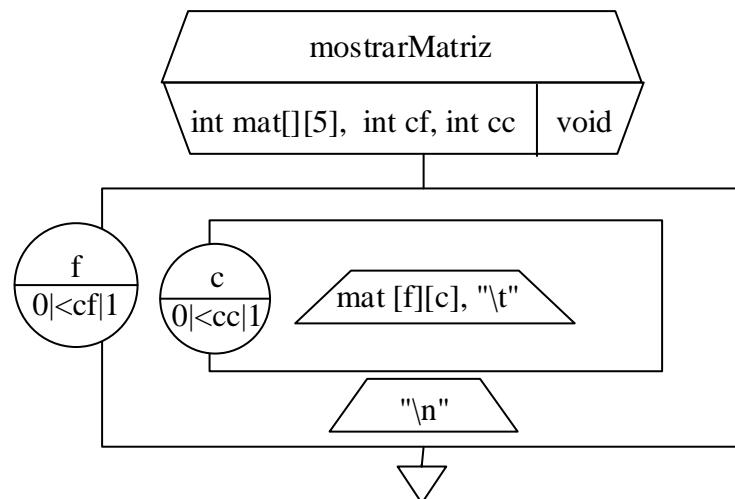
Se utiliza un subíndice para acceder a cada fila. Con el valor fijo de cada fila, se recorre con el otro subíndice todas las columnas de dicha fila.

El siguiente diagrama muestra una función para el ingreso por teclado de los valores de una matriz cargando los datos por fila. Es decir, se cargan primero todos los datos de la fila 0, luego los de la fila 1 y así hasta finalizar. El dato de la cantidad de columnas podría omitirse ya que la función está armada para cargar una matriz entera de 5 columnas (sin importar la cantidad de filas). Debido a que la cantidad de columnas es un dato requerido al especificar una matriz como parámetro, las funciones ya no se pueden hacer tan genéricas como sí es posible hacerlas para los vectores. Es decir que esta función servirá para ingresar datos en una matriz entera, sin importar la cantidad de filas, pero que tenga 5 columnas.



```
void CargaMatrizPorFilas(int mat[][][5], int cf, int cc)
{
    int f, c;
    for (f=0; f<cf; f++)
    {
        for (c=0; c<cc; c++)
        {
            printf("Ingrese un numero para fila %d columna %d: ", f, c);
            scanf("%d", &mat[f][c]);
        }
    }
}
```

El recorrido por filas se utiliza también para visualizar una matriz en forma de tabla en pantalla, para ello se imprime toda la primera fila, un salto de página, toda la segunda fila y así sucesivamente. La siguiente función permite mostrar una matriz de enteros en forma de tabla, recibe la matriz, la cantidad de filas y la cantidad de columnas.



Para lograr la visualización en formato tabla, cada elemento de la fila se separa con una tabulación y al finalizar de mostrar todos los datos de la fila se agrega un salto de línea para que los datos de la siguiente fila se muestren debajo encolumnados con los anteriores.

En la codificación para mostrar datos encolumnados no es recomendable usar tabulaciones ya que si la matriz tiene muchas columnas no entraría en pantalla. Es mejor encolumnar los datos utilizando ancho de campo en el formato del printf dependiendo del largo de los datos que se almacenan. Por ejemplo, si la matriz que se quiere mostrar tiene números enteros de 2 cifras como máximo en el printf usaremos el formato `%3d` para indicarle que cada valor lo muestre en 3 lugares, de forma que quede un espacio delante y luego el dato. El código de la función es el siguiente:

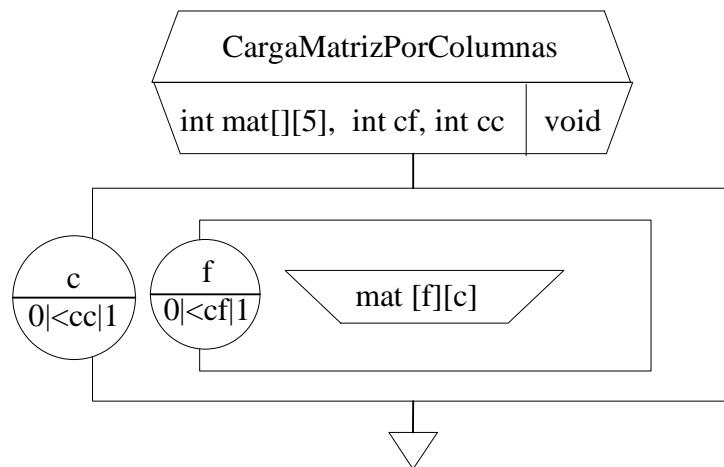
```

void mostrarMatriz(int mat[][][5], int cf, int cc)
{
    int f,c;
    for (f=0;f<cf;f++)
    {
        for (c=0;c<cc;c++)
            printf("%3d",mat[f][c]);
        printf ("\n");
    }
}

```

13.2 Recorrido por columnas

Se invierte el orden de los ciclos for, el exterior va a ser el que recorre las columnas y el interior el que recorre las filas. Se utiliza un subíndice para acceder a cada columna. Con el valor fijo de cada columna, se recorre con el otro subíndice todas las filas de dicha columna.



```

void CargaMatrizPorColumnas(int mat[][5], int cf, int cc)
{
    int f,c;
    for (c=0;c<cc;c++)
    {
        for (f=0;f<cf;f++)
        {
            printf("Ingrese un numero para columna %d fila %d: ", c,f);
            scanf("%d",&mat[f][c]);
        }
    }
}

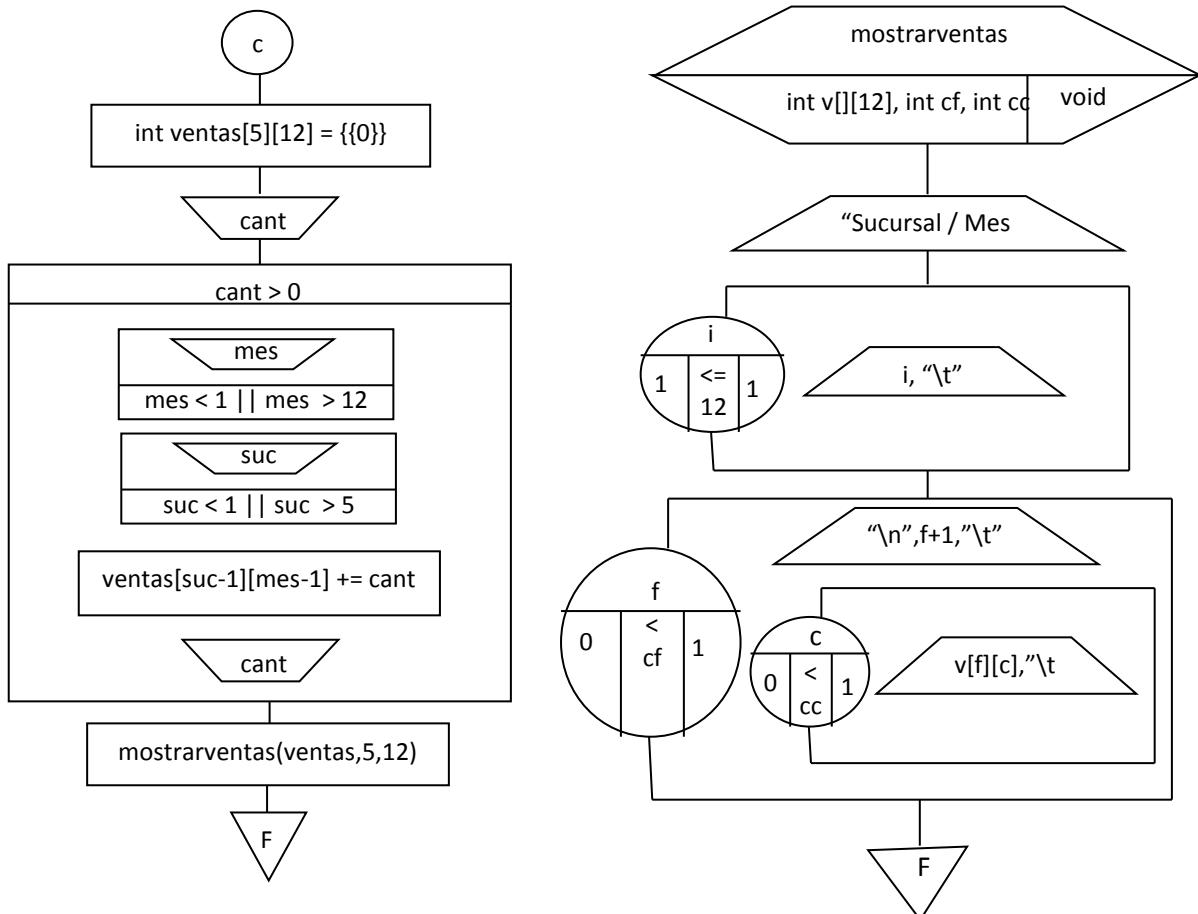
```

13.3 Acceso directo

Al igual que en los vectores es posible acceder directamente a cualquier posición de una matriz indicando subíndices válidos para las filas y columnas sin necesidad de cargar la matriz entera. En estos casos si se le solicita al usuario el número de fila y de columna donde desea cargar el dato, el número ingresado DEBE validarse para asegurarse de que corresponde con una fila o columna válida. A modo de ejemplo se realiza el siguiente programa: Se desea registrar la cantidad de ventas realizadas en el año en una empresa. Para ello se ingresan los datos de las facturas indicando por cada una:

- el mes (1 a 12)
- número de sucursal (entero de 1 a 5)
- cantidad de ventas (entero)

La carga de las facturas finaliza con una cantidad menor o igual a 0. Al finalizar mostrar los datos en forma de tabla.



```
#include <stdio.h>
void MostrarVentas(int [][]12, int, int);
int main()
{
    int ventas[5][12]={0}, cant, mes, suc;
    printf("Ingrese la cantidad vendida: ");
    scanf("%d",&cant);
    while (cant>0)
    {
        do
        {
            printf("Ingrese el mes:");
            scanf("%d",&mes);
        }while (mes<1 || mes>12);

        do
        {
            printf("Ingrese la sucursal:");
            scanf("%d",&suc);
        }while (suc<1 || suc>5);
        ventas[suc-1][mes-1]+=cant;
        printf("Ingrese la cantidad vendida: ");
        scanf("%d",&cant);
    }
    MostrarVentas(ventas,5,12);
    return 0;
}

void MostrarVentas(int v[][]12, int cf, int cc)
{
    int i,f,c;
    printf ("Sucursal/Mes");

    for (i=1;i<=12;i++)
        printf ("%5d", i);

    for (f=0;f<cf;f++)
    {
        printf ("\n%12d", f+1);
        for (c=0;c<cc;c++)
            printf ("%5d", v[f][c]);
    }
}
```

13.4 Suma por filas

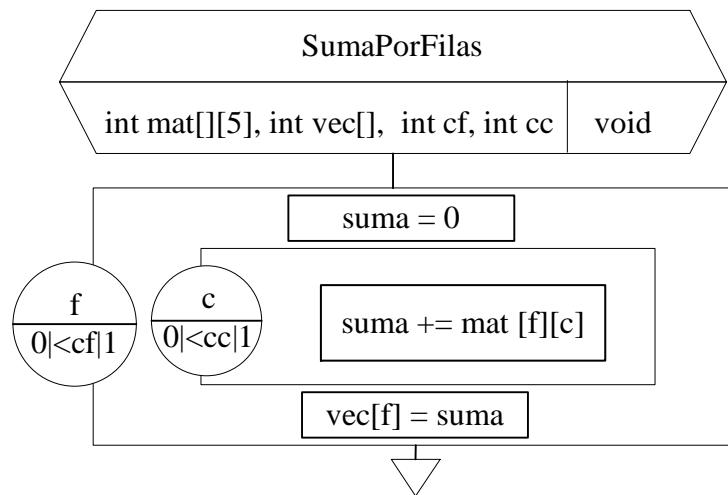
Muchas veces es necesario realizar una suma de todos los valores cada una de las filas de una matriz. Para ello se recorre la matriz por filas y suman todos los valores de esta. En el ejemplo anterior sumar por fila nos permitirá conocer la cantidad total de unidades vendidas por cada sucursal. El resultado serán tantas sumas como filas tenga nuestra matriz, por lo tanto, para guardar esos valores se genera un vector en paralelo a las filas de la matriz. La figura 4 muestra un ejemplo de un vector que guarda los resultados de la suma de cada una de las filas de una matriz de enteros.

| | | | | |
|---|---|---|---|----|
| 3 | 2 | 1 | → | 6 |
| 9 | 1 | 5 | → | 15 |
| 7 | 2 | 4 | → | 13 |
| 7 | 7 | 8 | → | 22 |

Figura 4: vector resultante de la suma por filas de los valores de una matriz

A continuación, se muestra el diagrama y código de la función para sumar las filas de una matriz entera de 5 columnas. En este caso se utiliza una variable suma para ir acumulando, pero esta variable

puede obviarse y sumar directamente en el vector resultado, previamente poniendo en 0 dicha posición o iniciar el vector previamente todo en 0.



```

void SumaPorFilas(int mat[][5], int vec[], int cf, int cc)
{
    int f,c,suma;
    for (f=0;f<cf;f++)
    {
        suma=0;
        for (c=0;c<cc;c++)
            suma += mat [f][c];
        vec[f]=suma;
    }
}

```

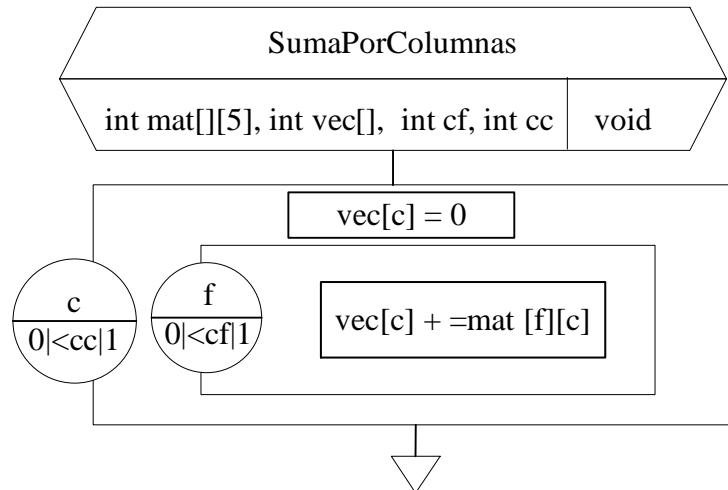
13.5 Suma por columnas

Del mismo modo es posible realizar la suma de todos los valores de cada una de las columnas de la matriz. Para ello se recorre la matriz por columna sumando todos sus valores. Aplicándolo al ejemplo anterior obtendríamos un vector con la información total de unidades vendidas en cada uno de los meses del año ya que se estaría sumando la cantidad vendida en cada sucursal para cada mes. La figura 5 muestra un ejemplo de un vector que guarda los resultados de la suma de cada una de las columnas de una matriz de enteros.

| | | |
|----|----|----|
| 3 | 2 | 1 |
| 9 | 1 | 5 |
| 7 | 2 | 4 |
| 7 | 7 | 8 |
| ↓ | ↓ | ↓ |
| 26 | 12 | 18 |

Figura 5: vector resultante de la suma por columnas de los valores de una matriz

A continuación, se muestra el diagrama y código de una función que permite sumar por columna una matriz de enteros. En este ejemplo directamente se va acumulando sobre el vector sin usar una variable intermedia para calcular la suma.



```
void SumaPorColumnas(int mat[][5], int vec[], int cf, int cc)
{
    int f,c,suma;
    for (c=0;c<cc;c++)
    {
        vec[c]=0;
        for (f=0;f<cf;f++)
            vec[c] += mat[f][c];
    }
}
```



Elementos de Programación

UNIDAD 8. CADENAS DE CARACTERES (STRINGS)

INDICE

| | | |
|------------|--|-----------|
| 1. | INTRODUCCIÓN | 2 |
| 2. | LECTURA POR TECLADO DE STRINGS | 2 |
| 2.1 | SCANF | 2 |
| 2.2 | GETS | 3 |
| 2.3 | FGETS..... | 3 |
| 3. | MOSTRAR STRINGS POR PANTALLA | 5 |
| 4. | INICIALIZACIÓN DE STRINGS..... | 5 |
| 5. | BIBLIOTECA PARA EL MANEJO DE TEXTO (STRING.H) | 5 |
| 6. | VECTOR DE STRINGS..... | 8 |
| 7. | FUNCIONES SOBRE VECTORES DE STRING | 9 |
| 7.1 | CARGA | 9 |
| 7.2 | MOSTRAR..... | 9 |
| 7.3 | BÚSQUEDA SECUENCIAL | 10 |
| 7.4 | ORDEN | 10 |
| 8. | EJEMPLO DE APLICACIÓN | 11 |

UNIDAD 8 - Cadenas de caracteres (strings)

OBJETIVOS: Comprender la forma del manejo de texto en el lenguaje C. Utilizar funciones para trabajar con texto. Incorporar el manejo de texto a los programas existentes.

1. Introducción

El lenguaje C, a diferencia de otros lenguajes, hace un manejo muy particular de las variables para almacenar texto. Un texto no es más que una sucesión de letras, por lo tanto, se necesita almacenar varios caracteres uno a continuación del otro. Una variable del tipo char permite almacenar un único carácter, por lo tanto, para almacenar un texto se debe definir un vector de caracteres con tantas posiciones como letras pueda tener el texto que se quiere almacenar. Entonces, en un principio se podría decir que una variable del tipo string o cadena de caracteres, no es más que un vector de char. Sin embargo, esta afirmación no es del todo cierta ya que hay una característica que las diferencia. Toda cadena de caracteres define su terminación con un carácter de control '\0'. Este carácter especial indica el fin de la cadena.

Por ejemplo, si se quiere ingresar un nombre, no se sabe exactamente el largo del mismo por lo tanto se define un vector con una cantidad suficiente de acuerdo al nombre más largo que se quiera guardar, por ejemplo:

```
char nombre [20];
```

Declara un vector de char de 20 posiciones. Si en dicho vector se almacena un nombre corto, por ejemplo "JUAN", dicho nombre solo ocupará 4 de los 20 caracteres, y por lo tanto, luego de la letra N se debe poner el carácter de fin de cadena indicando hasta donde llega dicho nombre.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| J | U | A | N | \0 | | | | | | | | | | | | | | | |
|---|---|---|---|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Entonces la diferencia principal entre un vector de caracteres y un string es la presencia de ese carácter especial de fin de cadena.

2. Lectura por teclado de strings

Ahora bien, como se indicó, un string es un vector de caracteres, pero al momento de solicitar el ingreso por teclado del dato no se le va a pedir al usuario que ingrese una a una las letras en cada posición del vector. Para la lectura de string el lenguaje dispone de distintas alternativas:

2.1 scanf

Tradicionalmente todos los ingresos desde teclado hasta ahora se realizaron utilizando esta función. Donde se pone como primer parámetro el formato de los datos a ingresar y como segundo parámetro la dirección de memoria donde almacenar el dato. El formato para los strings es %s. Veamos el siguiente ejemplo:

```
#include <stdio.h>
int main()
{
    char nombre[20];
    printf ("Ingrese un nombre: ");
    scanf ("%s", nombre);
    printf ("El nombre ingresado es: %s", nombre);
    return 0;
}
```

Note que en el scanf no fue necesario poner el & para obtener la dirección de memoria ya que al ser nombre un vector contiene la dirección de inicio del mismo. Por lo tanto, el & NO debe ponerse ya contiene la dirección de memoria que se necesita para indicarle donde guardar el dato.

Si se ejecuta y se prueba este programa e ingresa por teclado JUAN y luego presiona enter, automáticamente se completa el vector guardando la J en la posición 0, la U en la posición 1, y así sucesivamente, hasta la última letra y automáticamente en la posición siguiente a la letra N el scanf agrega el carácter de fin de cadena \0 ya que se le indica que está leyendo un string con el formato %s.

Hasta aquí no hay problemas y todo funciona normalmente. Pero el scanf con formato de string tiene un problema. Si prueba nuevamente el ejemplo y en el nombre ingresa JUAN CARLOS, en pantalla mostrará:

```
El nombre ingresado es: JUAN
```

Internamente en el vector solo guardará JUAN y a continuación el carácter de fin de cadena. Esto se debe a que el espacio corta el ingreso del scanf almacenando solo JUAN y haciendo que se pierdan el resto de los datos. Recordemos que cuando con scanf se leía más de una variable una forma de separarlas era con espacio y por ello no funciona correctamente en la lectura de string.

2.2 gets

En la biblioteca stdio.h hay una función llamada gets que permite solucionar el problema del scanf al leer un string. Reemplazando dicha función en el programa anterior quedaría:

```
#include <stdio.h>
int main()
{
    char nombre[20];
    printf ("Ingrese un nombre: ");
    gets(nombre);
    printf ("El nombre ingresado es: %s", nombre);
    return 0;
}
```

De esta forma se elimina el problema del scanf y si se ingresa por teclado JUAN CARLOS en el vector los datos quedarán almacenados de la siguiente forma:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|--|---|---|---|---|---|---|----|--|--|--|--|--|--|--|--|
| J | U | A | N | | C | A | R | L | O | S | \0 | | | | | | | | |
|---|---|---|---|--|---|---|---|---|---|---|----|--|--|--|--|--|--|--|--|

Y en pantalla se mostrará:

```
El nombre ingresado es: JUAN CARLOS
```

2.3 fgets

Tanto el scanf como gets NO CHEQUEAN LIMITES, por lo tanto, si se define el vector para guardar 20 caracteres y se ingresa un nombre más largo la variable se guarda en las posiciones siguientes que NO tienen reservadas haciendo que probablemente se pierdan datos de otras variables, y por lo tanto, haciendo que el programa funcione mal. Algunos compiladores darán una advertencia diciendo que es peligroso utilizar el gets debido a que se puede escribir memoria no reservada.

Para solucionar esto se puede acudir a otra función que se encuentra también en la biblioteca stdio.h. Esta función es fgets y tiene 3 parámetros, el primero es el vector donde se va a guardar el dato, el segundo la cantidad de caracteres que se pueden almacenar en dicho vector y el tercero es de donde se leen los datos que en este caso siempre será desde teclado. La siguiente línea:

```
fgets(nombre,20, stdin);
```

Indica que se va a leer desde teclado (indicado por stdin), como máximo 20 caracteres y se va a guardar a partir de la dirección de memoria indicada por nombre.

Reemplazando en el programa anterior:

```
#include <stdio.h>
int main()
{
    char nombre[20];
    printf ("Ingrese un nombre: ");
    fgets(nombre,20,stdin);
    printf ("El nombre ingresado es: %s", nombre);
    return 0;
}
```

Al probar este programa si se ingresa un texto de más de 20 caracteres, guardará en el vector los 19 primeros y en la última posición pondrá el carácter de fin de cadena \0, el resto de caracteres no los guarda, y por lo tanto, no sobrescribe memoria no reservada.

Al parecer esta función soluciona todo los problemas pero si se prueba nuevamente ingresando por teclado JUAN CARLOS y luego la tecla enter, se verá que el vector almacena lo siguiente:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|--|---|---|---|---|---|---|----|----|--|--|--|--|--|--|--|
| J | U | A | N | | C | A | R | L | O | S | \n | \0 | | | | | | | |
|---|---|---|---|--|---|---|---|---|---|---|----|----|--|--|--|--|--|--|--|

Es decir, que la función fgets almacena también el enter ingresado. Por lo tanto, cuando se muestra la cadena luego de JUAN CARLOS dejará un salto de línea. Dependerá del uso que quiera darle al string si ese salto de línea molesta o no. Una posible solución luego de leer el string con fgets, sería buscar si quedó almacenado el \n y eliminarlo poniendo en su lugar el \0. En el siguiente ejemplo ingreso del string se realiza mediante una función LeerTexto que lee mediante fgets y luego busca y elimina el salto de línea almacenado:

```
#include <stdio.h>
void LeerTexto (char[], int);
int main()
{
    char nombre[20];
    printf ("Ingrese un nombre: ");
    LeerTexto(nombre,20);
    printf ("El nombre ingresado es: %s", nombre);
    return 0;
}

void LeerTexto (char texto[], int largo)
{
    int i;
    fgets(texto, largo, stdin);
    i=0;
    while (texto[i]!='\0')
    {
        if (texto[i]=='\n')
            texto[i]='\0';
        else
            i++;
    }
}
```

3. Mostrar strings por pantalla

En los ejemplos anteriores se vio que es posible mostrar los string mediante un printf con formato %s, esta es la forma más difundida y es la que más utilizará ya que permite darle formato, mostrar otras variables, etc.

Otra forma para mostrar un string es mediante la función puts disponible en la biblioteca stdio.h. Esta función recibe como parámetro el string y lo muestra por pantalla y además automáticamente luego de mostrar el texto hace un salto de línea.

Es decir, que la siguiente línea de código (definiendo nombre como un string):

```
puts (nombre);
```

es equivalente a:

```
printf ("%s\n", nombre);
```

4. Inicialización de Strings

Al declarar la memoria del string es posible asignar un valor, para ello se le asigna una constante del tipo string que se escribe entre comillas. El carácter de fin de cadena se agrega automáticamente luego de la última letra. A continuación, observe algunos ejemplos de inicialización y como queda el vector resultante:

```
char nombre[20] = "ANA MARIA";
```

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|--|---|---|---|---|---|----|--|--|--|--|--|--|--|--|--|--|
| A | N | A | | M | A | R | I | A | \0 | | | | | | | | | | |
|---|---|---|--|---|---|---|---|---|----|--|--|--|--|--|--|--|--|--|--|

```
char nombre[] = "ANA MARIA";
```

| | | | | | | | | | |
|---|---|---|--|---|---|---|---|---|----|
| A | N | A | | M | A | R | I | A | \0 |
|---|---|---|--|---|---|---|---|---|----|

Si no se especifica el tamaño automáticamente reserva la memoria mínima suficiente para almacenar el texto y el carácter de fin de cadena.

Cuidado!! Si se le pone un tamaño al vector y se le asignan más caracteres de los definidos NO guardará el \0, y por lo tanto, ya no puede tratarse como un string ya que la función para mostrar no encontrará el fin de cadena.

```
char nombre[3] = "MARCELO";
```

| | | |
|---|---|---|
| M | A | R |
|---|---|---|

5. Biblioteca para el manejo de texto (string.h)

Debido a que los string son vectores con un carácter especial que indica el final de la cadena hay ciertas cosas como copiar uno a otro, comparar, etc que NO se pueden hacer directamente ya que no son un tipo de dato en sí mismos. Identificando el fin de cadena es posible armar distintas funciones para hacer esas tareas. Por ejemplo, para saber la cantidad de caracteres de un string se puede recorrer posición a posición el vector hasta encontrar el fin de cadena contando cuantas posiciones nos desplazamos. Para copiar un string en otro es similar a la copia de vectores donde

se copia posición a posición en este caso hasta encontrar el fin de cadena. Todas estas funciones las podemos desarrollar, pero dentro del lenguaje C ya existe una biblioteca que maneja distintas funciones relacionadas con cadenas de caracteres. La biblioteca string.h

Dentro de string.h hay muchas funciones, veremos algunas de ellas:

| | |
|---------|--|
| strlen | Determina la longitud de una cadena |
| strcpy | Copia una cadena a otra |
| strcat | Concatena dos cadenas dejando el resultado en la primera |
| strcmp | Compara dos cadenas |
| strncpy | Compara dos cadenas ignorando si son mayúsculas o minúsculas |

Función strlen:

Determina la longitud de una cadena, sin contabilizar el carácter nulo de terminación de la misma.

Sintaxis: strlen (cadena)

La función retorna un entero indicando la cantidad de caracteres.

Ejemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char cadena [21];
    printf ("Ingrese una cadena de no más de 20 caracteres \n");
    gets (cadena);
    printf ("La cadena ingresada contiene: %d caracteres", strlen(cadena) );
    return 0;
}
```

Función strcpy:

Copia desde una cadena de origen hacia una cadena de destino.

Sintaxis: strcpy (cadena destino, cadena origen)

Ejemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char original [15], copia [15];
    printf ("Ingrese una cadena que sera luego copiada \n");
    gets (original);
    strcpy (copia, original);
    printf ("La cadena original es: %s y la copia es: %s", original, copia );
    return 0;
}
```

También strcpy se puede usar para asignar una constante a un string cuando no se hace al momento de inicializar la variable.

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char texto [25];
    strcpy(texto, "MENSAJE");
```

```
    puts(texto);
    return 0;
}
```

Función strcat:

Concatena (añade) una cadena detrás de otras quedando el resultado en la cadena que se encuentra en primer orden.

Sintaxis: strcat (cadena receptora, cadena a añadir)

Ejemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char receptor [40] = "Se agrego lo siguiente", dador [] =", me agregue";
    printf ("Las cadenas por separado son: \n\t %s\n\t %s", receptor, dador);
    strcat (receptor, dador);
    printf ("\nLas cadenas unificadas son: \n\t %s ", receptor );
    return 0;
}
```

Es importante asegurarse de que la cadena receptora tenga el espacio suficiente para guardar la cadena a añadir caso contrario sobrescribe memoria no asignada

Función strcmp:

Esta función compara dos cadenas y devuelve el resultado de la comparación.

Sintaxis: strcmp (cadena 1, cadena 2)

El valor que devuelve que será el resultado de la comparación es el siguiente:

| | |
|---|-----------------------------|
| Si las cadenas son iguales | devolverá un cero (0) |
| Si la cadena 1 es mayor que la cadena 2 | devolverá un valor positivo |
| Si la cadena 1 es menor que la cadena 2 | devolverá un valor negativo |

Ejemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char cadena1[30], cadena2[30];
    printf ("Ingrese la primer cadena:");
    gets(cadena1);
    printf ("Ingrese la segunda cadena:");
    gets(cadena2);
    if (strcmp (cadena1, cadena2) == 0)
        printf ("\nAmbas cadenas son iguales ");
    else
        if (strcmp (cadena1, cadena2) > 0 )
            printf ("\nLa cadena1 es mayor que la cadena2");
        else
            printf ("\nLa cadena2 es mayor que la cadena1");
    return 0;
}
```

La comparación NO es por largo sino alfabética, comparando el peso en valor ASCII de cada una de las letras, posición a posición, hasta encontrar alguna diferencia. Hay que recordar que el ASCII de

las minúsculas es mayor que el de las mayúsculas, por lo tanto, una palabra en minúsculas será mayor que una en mayúscula. A continuación, se muestran algunos ejemplos:

| Cadena1 | Cadena2 | Mayor / iguales |
|------------|---------|-----------------|
| HOLA | HOLA | IGUALES |
| Hola | hola | Cadena2 |
| hola mundo | hola | Cadena1 |
| ana | anana | Cadena2 |
| teXto | texto | Cadena2 |

Función strcmpl:

Es exactamente igual a la función strcmp pero ignora si son mayúsculas o minúsculas es decir que el string “ANA” es igual al string “ana” y también es igual al string “anA” o cualquiera de sus variantes.

6. Vector de strings

Un string es en realidad un vector de caracteres, ahora si se quiere guardar varios string en un vector entonces tiene que definir varios vectores de caracteres y para ello se utiliza una matriz. Por lo tanto, un vector de strings es una matriz de caracteres donde en cada fila se guardará un string finalizado con \0. Si bien se define como una matriz se manejará tanto para la lectura como para mostrar como un vector ya que no se va a recorrer letra a letra, sino que se va a leer y mostrar las palabras completas.

Para definir un vector de string entonces se define una matriz donde:

La cantidad de filas es la cantidad de string que se quiere guardar y la cantidad de columnas indica la cantidad máxima de caracteres de cada uno de los strings (contando el \0).

El siguiente esquema representa un vector de string, donde se pueden almacenar 5 nombres de hasta 10 caracteres cada uno (se define de 11 para el \0).

char nombres [5][11];

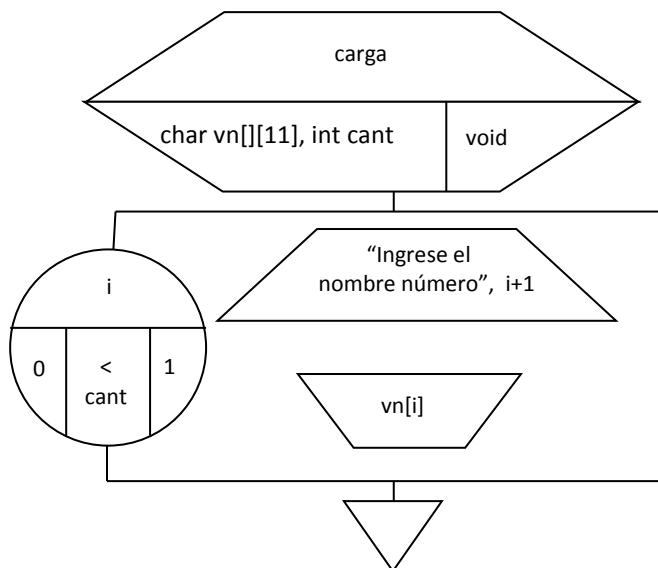
| | | | | | | | | | | |
|---|---|---|----|----|----|---|----|---|---|----|
| A | N | A | \0 | | | | | | | |
| L | U | I | S | \0 | | | | | | |
| J | U | A | N | | P | A | B | L | O | \0 |
| L | U | C | I | A | N | O | \0 | | | |
| M | A | R | I | A | \0 | | | | | |

Los nombres ingresados en cada fila pueden ser de cualquier largo siempre y cuando no pasen la cantidad máxima de caracteres reservada.

7. Funciones sobre vectores de string

Como se indicó anteriormente si bien un vector de string es una matriz se maneja como un vector. A continuación, se desarrollan algunas de las funciones más usadas. Tomando cadenas de 10 caracteres máximo.

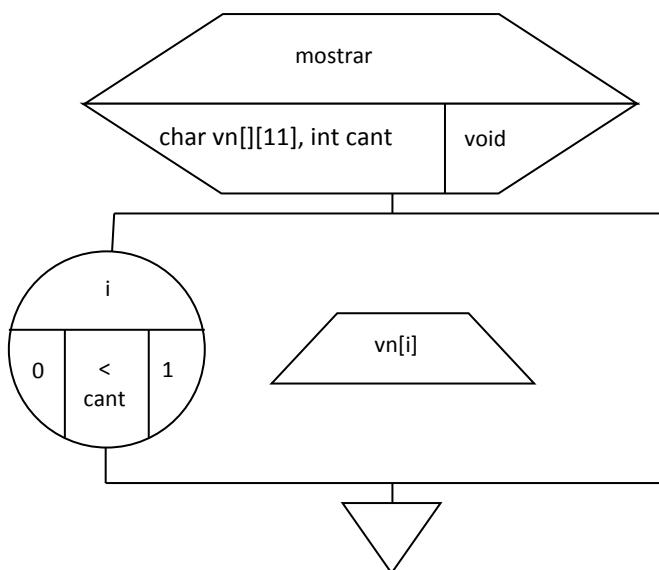
7.1 Carga



```

void carga(char vn[][][11], int cant)
{
    int i;
    for (i=0;i<cant;i++)
    {
        printf ("Ingrese el nombre numero %d: ", i+1);
        gets(vn[i]);
    }
}
  
```

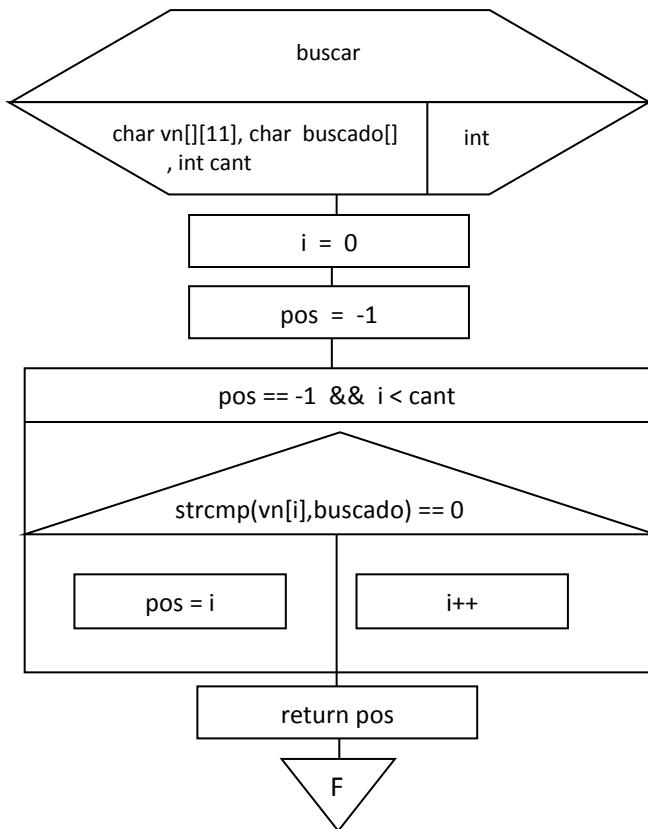
7.2 Mostrar



```

void mostrar(char vn[][][11], int cant)
{
    int i;
    for (i=0;i<cant;i++)
        puts(vn[i]);
}
  
```

7.3 Búsqueda Secuencial



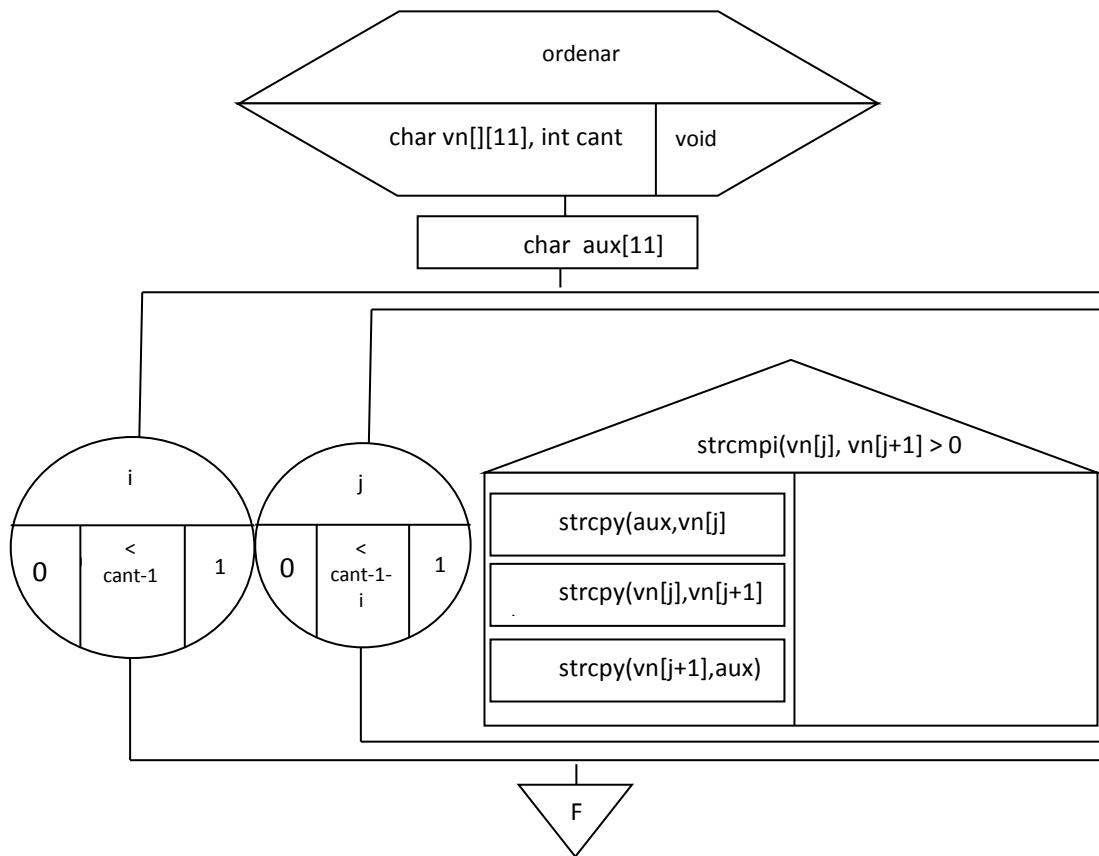
```

int buscar(char vn[][11], char buscado[], int cant)
{
    int i=0, pos=-1;
    while (pos== -1 && i<cant)
    {
        if (strcmp(vn[i], buscado)==0)
            pos =i;
        else
            i++;
    }
    return pos;
}
  
```

7.4 Orden

```

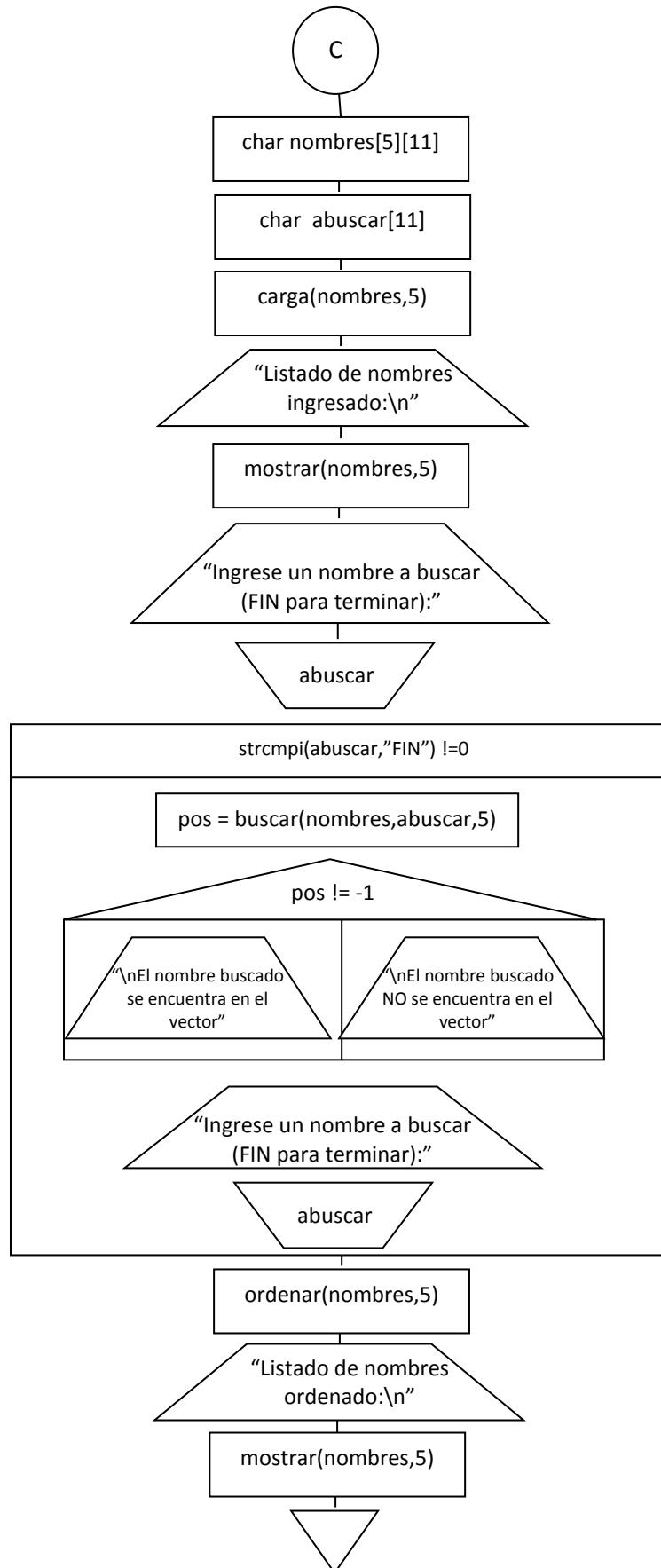
void ordenar (char vn[][11],int cant)
{
    int i,j;
    char aux[11];
    for (i=0;i<cant-1;i++)
    {
        for (j=0;j<cant-1-i;j++)
        {
            if (strcmpi(vn[j],vn[j+1]) > 0)
            {
                strcpy(aux, vn[j]);
                strcpy (vn[j],vn[j+1]);
                strcpy (vn[j+1], aux);
            }
        }
    }
}
  
```



8. Ejemplo de aplicación

Utilizando las funciones definidas anteriormente se desea realizar el siguiente programa, cargar 5 nombres de hasta 10 caracteres cada uno en un vector de string. Luego ingresar nombres por teclado e indicar si se encuentran en el vector, la búsqueda finalizará con un nombre igual a “FIN”. Luego mostrar el listado de nombres ordenado alfabéticamente de menor a mayor.

No se desarrollan las funciones ya que son las mismas del punto 7.



```
#include <stdio.h>
#include <string.h>
void carga(char[][11],int);
void mostrar(char[][11],int);
int buscar(char[][11],char [], int);
void ordenar (char[][11],int);
int main ()
{
    int pos;
    char nombres[5][11], abuscar[11];
    carga(nombres,5);
    printf ("\nListado de nombres ingresado:\n");
    mostrar(nombres,5);
    printf ("\nIngrese un nombre a buscar (FIN para terminar): ");
    gets(abuscar);
    while (strcmpi(abuscar, "FIN")!=0)
    {
        pos = buscar(nombres,abuscar,5);
        if (pos!=-1)
            printf ("\nEl nombre buscado se encuentra en el vector");
        else
            printf ("\nEl nombre buscado NO se encuentra en el vector");

        printf ("\nIngrese un nombre a buscar (FIN para terminar): ");
        gets(abuscar);
    }
    ordenar(nombres,5);
    printf ("\n\nListado de nombres ordenado:\n");
    mostrar(nombres,5);
    return 0;
}
```



Elementos de Programación

UNIDAD 9. ESTRUCTURA DE DATOS

INDICE

| | |
|---|----|
| 1. COMO CREAR UNA ESTRUCTURA DE DATOS EN EL LENGUAJE C. | 2 |
| 2. COMO ACCEDER A LOS MIEMBROS (CAMPOS) DE UNA ESTRUCTURA DE DATOS. | 4 |
| 3. COMO INICIALIZAR LOS MIEMBROS (CAMPOS) DE UNA VARIABLE TIPO ESTRUCTURA. | 5 |
| 4. COMO UTILIZAR ARREGLOS DE ESTRUCTURAS DE DATOS (VECTORES Y MATRICES). | 6 |
| 5. COMO ANIDAR UNA DE ESTRUCTURA DE DATOS. | 8 |
| 6. COMO UTILIZAR UNA ESTRUCTURA DE DATOS EN FUNCIONES. | 9 |
| 7. COPIA DE ESTRUCTURAS | 11 |
| 8. ESTRUCTURAS CON VECTORES | 12 |
| 9. OTRA FORMA DE DECLARAR LAS ESTRUCTURAS | 15 |
| 10. ORDENAR UN VECTOR DE ESTRUCTURAS | 16 |

UNIDAD 9 - ESTRUCTURA DE DATOS

OBJETIVOS: Poder definir nuevos tipos de datos compuestos, para guardar información de una misma entidad en una única variable. Reemplazar el uso de vectores paralelos con vectores de estructuras.

1. Como crear una estructura de datos en el lenguaje C.

Cuando se inicia la construcción (codificación) de programas en los llamados programas de alto nivel se comienza con algoritmos cuyos datos están alojados en lo que se conoce como variables de “tipo simple”: enteros, reales, caracteres, etc. (en Lenguaje C, int, float, char, etc.). Pero en ciertos casos se torna complicado el manejo de las variables en forma separada debiendo generar muchas variables para guardar la información de una entidad. Además, cuando se trabajan con varias entidades se deben generar varios vectores paralelos para guardar la información. Por ejemplo, si se tiene un producto que dispone de un código numérico, una descripción de texto y un precio, ya son 3 variables diferentes que guardan la información de la misma entidad, y se necesitan más variables si se quiere por ejemplo guarda el stock y otros datos de dicho producto. El lenguaje C permite crear una estructura de datos donde se puede guardar toda la información de determinada entidad en una única variable, definiendo un nuevo tipo de dato.

El tipo de dato que se pasará a desarrollar se lo suele llamar “estructura de datos” o “registro” (en el lenguaje C “struct”) y es el agrupamiento de un conjunto de datos simples y muchas veces involucra el anidamiento de estructuras de datos “struct” como se verá más adelante.

La estructura de datos o registro se lo puede ver mejor a través de un ejemplo: Suponiendo que se quiere procesar los datos de una persona, y se establece tener la siguiente información:

| | |
|----------------|--|
| DNI | Como la variable que contiene el Documento Nacional de Identidad (tipo entero, en Lenguaje C es int) |
| ApellidoNombre | Como la variable que contiene el Apellido y el Nombre (suponiendo una cadena de 40 caracteres incluyendo 1 carácter más por el fin de cadena (\0)). |
| Sexo | Como la variable que contiene el Sexo de la persona (tipo carácter, en Lenguaje C es char) donde F corresponde a Femenino y M corresponde a Masculino. |
| DiaNacimiento | Como la variable que contiene el Día de Nacimiento (tipo entero, en Lenguaje C es int) |
| MesNacimiento | Como la variable que contiene el Mes de Nacimiento (tipo entero, en Lenguaje C es int) |
| AnioNacimiento | Como la variable que contiene el Año de Nacimiento (tipo entero, en Lenguaje C es int) |

Los datos de una persona podrían ser muchos más, pero por una cuestión de simplificación y didáctica se utilizarán los enunciados en los párrafos anteriores para una estructura de datos.

En el ejemplo, se pueden observar que son variables simples (variables: int, float, char). Si se codifican los siguientes datos en el Lenguaje C, sería de la siguiente manera (Ejemplo 1):

Ejemplo 1:

```
#include <stdio.h>
int main()
{
    //declaración de variables locales al programa principal
    int DNI;
    char ApellNombre[41], Sexo;
    int DiaNacimiento, MesNacimiento, AnioNacimiento;
    //posteriormente irán las propias sentencias del programa principal
    return 0;
}
```

En el “Ejemplo 1” el acceso a las distintas variables es en forma separada, pero las mismas no darán el concepto de unidad de datos que representan el registro de una persona.

En el gráfico se puede observar cómo se agrupan los mismos datos del “Ejemplo 1” de forma tal que formen una estructura de datos o registro.

| Estructura PERSONA | | | | | |
|--------------------|-------------------|------|------------|-----------|-----------|
| DNI | Apellido y Nombre | Sexo | Dia Nacim. | Mes Nacim | Año Nacim |

En el Lenguaje C la forma para poder lograr el agrupamiento descrito arriba se lo conoce con el nombre de la palabra reservada “struct” y se codifica de la siguiente manera:

Ejemplo 2:

```
#include <stdio.h>

struct PERSONA
{
    int DNI;
    char ApellNombre[41];
    char Sexo;
    int DiaNacimiento;
    int MesNacimiento;
    int AnioNacimiento;
} ;
int main()
{
    ...
}
```

Este es el nombre que le asigna al agrupamiento de datos que llamará PERSONA. Y se lo denomina “etiqueta de estructura”

Importante: “No olvidar” en la llave de cierre de la estructura la colocación del “;” (punto y coma).

Cada uno de los datos dentro de la estructura se lo conoce como “campos” o “miembros” los mismos están englobados/encerrados entre llaves las cuales son obligatorias.

Cabe aclarar que la estructura de datos o registro se declarará dentro del área de declaraciones de variables globales por lo que tendrá una aplicación amplia en el programa principal y en todas las funciones del programa. También puede declararse dentro de funciones, pero esa estructura solo será válida dentro de dicha función ya que su declaración sería local.

Nota: Es importante destacar que la declaración de una estructura de datos o registro no es una variable que ocupa un lugar en memoria, pero es en sí misma “un molde o un sello” que permitirá la declaración de variables que contengan esa estructura de datos o registro. Para declarar una o varias variables de esta estructura de datos (“struct PERSONA”) se deberá proceder como se verá en el “Ejemplo 3”, dentro del main o programa principal.

Ejemplo 3:

```
struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    int DiaNacimiento;
    int MesNacimiento;
    int AnioNacimiento;
} ;
int main()
{
    // declaración de variables locales al programa principal
    {
        struct PERSONA empleado;
        ↓
        Aquí se declara la variable empleado
        que ocupa lugar en memoria y que
        obedece a la estructura PERSONA.
    }
    //posteriormente irán las propias sentencias del programa principal
    return 0;
}
```

Nota: No se pueden repetir los nombres de los miembros (campos) dentro de la declaración de una misma estructura de datos.

2. Como acceder a los miembros (campos) de una estructura de datos.

Para poder tener acceso a un determinado miembro (campo) de una estructura de datos se procede de la siguiente forma:

nombre de la variable tipo estructura. (punto) nombre del miembro

Ejemplo 4:

/* Es un programa que permite ingresar datos por teclado y guardarlos en una variable tipo estructura llamada PERSONA y luego mostrarlos por pantalla. Se podrá observar la utilización el operador miembro de estructura “.” (punto) */

```
#include <stdio.h>
#include <conio.h>
struct PERSONA // Aquí se declara la estructura de datos
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    int DiaNacimiento;
    int MesNacimiento;
    int AnioNacimiento;
} ;
```

```
int main()
{
    /* Aquí se declara la variable "empleado" de tipo de la estructura de datos
     "struct PERSONA"*/
    struct PERSONA empleado;

    /* Aquí se van a ingresar por teclado el contenido de los diferentes campos que
     contiene la variable "empleado" */
    printf("Ingrese Numero de DNI");
    scanf("%d", &empleado.DNI);
    printf("Ingrese Numero de Apellido y nombre");
    fflush(stdin);
    gets(empleado.ApellNombre);
    printf("Ingrese el Sexo");
    fflush(stdin);
    scanf("%c", &empleado.Sexo);
    printf("Ingrese Numero del Dia de Nacimiento");
    scanf("%d", &empleado.DiaNacimiento);
    printf("Ingrese Numero del Mes de Nacimiento");
    scanf("%d", &empleado.MesNacimiento);
    printf("Ingrese Numero del Año de Nacimiento");
    scanf("%d", &empleado.AnioNacimiento);
    printf("El empleado cuyo DNI es : %d , se llama %-40s y nacio el dia
    %2d/%2d/%4d", empleado.DNI , empleado.ApellNombre, empleado.DiaNacimiento,
    empleado.MesNacimiento , empleado.AnioNacimiento);
    return 0;
}
```

3. Como inicializar los miembros (campos) de una variable tipo estructura.

La inicialización de variables simples con un determinado valor se procede de la siguiente manera:

```
int acum = 0;
char caracter = 'F';
char cadena[21] = "Hola que Tal? ";
```

Para inicializar los miembros (campos) de una variable tipo estructuras, se procede de la siguiente manera:

```
struct PERSONA empleado = { 37234546, "Martinez, Julia", 'F',
18, 11, 1982};
```

Nota: Se debe respetar y conservar el orden y la totalidad de los miembros (campos) encerrados entre llaves {} (corchetes), y también sus correspondientes características, en el ejemplo utilizado serán:

```
(long int , cadena de caracteres ("      "), carácter (' '),
int, int , int);
```

Nota: Se debe respetar las capacidades máximas de cada uno tipo de dato de los miembros (campos) de una variable de estructura de dato.

```
#include <stdio.h>
#include <conio.h>
struct PERSONA // Aquí se declara la estructura de datos
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    int DiaNacimiento;
    int MesNacimiento;
    int AnioNacimiento;
};
```

```
int main()
{
    // Aquí se declara la variable "empleado" y se le asignan valores a
    // cada uno de sus miembros (campos)
    struct PERSONA empleado = { 37234546, "Martinez, Julia", 'F', 18, 11,
        1982};

    printf("El empleado cuyo DNI es : %d, se llama %-40s y nacio el dia
        %2d/%2d/%4d", empleado.DNI,
        empleado.Apellido,
        empleado.DiaNacimiento,
        empleado.MesNacimiento,
        empleado.AñoNacimiento);
    return 0;
}
```

4. Como utilizar Arreglos de estructuras de datos (vectores y matrices).

Se pueden generar los arreglos (vectores) y/o matrices con los elementos del tipo estructura de datos (**struct**).

La definición de estructuras hace que la información de una entidad esté unificada en una única variable, y como la definición de la estructura forma un nuevo tipo de dato también es posible definir vectores de estructuras eliminando la necesidad de utilizar vectores paralelos para guardar información relacionada, ahora cada posición del vector podrá contener todos los datos que sean necesarios.

En el “Ejemplo 5” se plantea un arreglo (vector) de 10 posiciones que contiene datos tipo de estructura de datos (struct).

Ejemplo 5:

```
/* Es un programa que permite ingresar datos en el vector de 10 elementos del tipo de dato
PERSONA */

#include <stdio.h>
#include <conio.h>
struct PERSONA
{
    long int DNI;
    char Apellido[41];
    char Sexo;
    int DiaNacimiento;
    int MesNacimiento;
    int AnioNacimiento;
};

int main()
{
    struct PERSONA vectEmpleado[10]; // Aquí se declara la variable vector de 10
    elementos tipo struct PERSONA
    int i;

    // Aquí se ingresan los datos por teclado y se guardan en el vector
    for (i= 0 ; i < 10 ; i++)
    {
        printf("Ingrese Numero de DNI");
        scanf("%d", &vectEmpleado[i].DNI);
        printf("Ingrese Numero de Apellido y nombre");
        fflush(stdin);
```

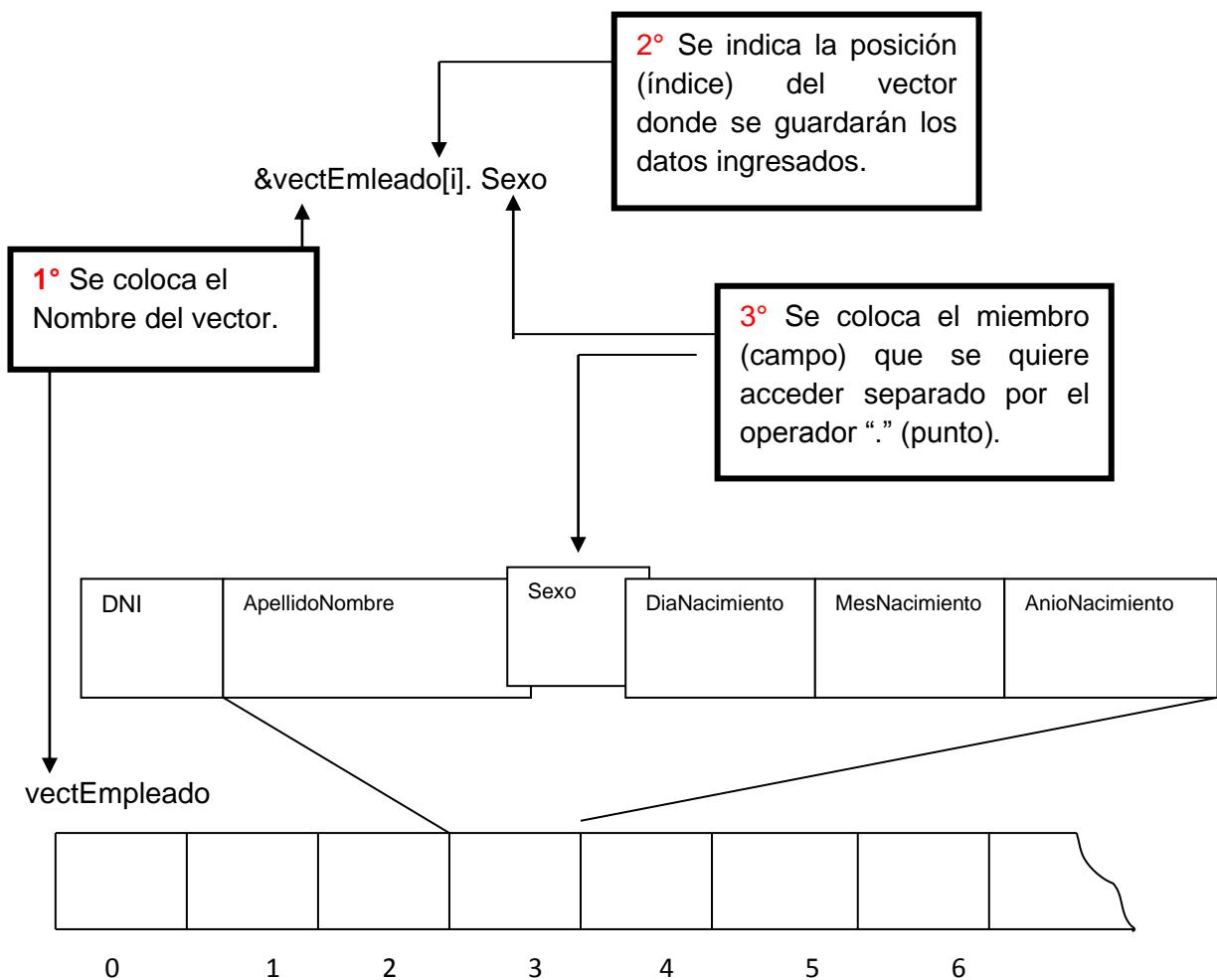
```

gets(vectEmpleado[i].ApellidoNombre);
printf("Ingrese Sexo");
fflush(stdin);
scanf("%c", &vectEmpleado[i].Sexo);
printf("Ingrese Numero de Dia Mes Año de Nacimiento");
printf("\nIngrese Numero del Dia de Nacimiento");
scanf("%d", & vectEmpleado[i].DiaNacimiento);
printf("Ingrese Numero del Mes de Nacimiento");
scanf("%d", & vectEmpleado[i].MesNacimiento);
printf("Ingrese Numero del Año de Nacimiento");
scanf("%d", & vectEmpleado[i].AnioNacimiento);
}

// Aquí se muestra el contenido de los elementos del vector.
for (i= 0 ; i < 10 ; i++)
{
    printf("El empleado cuyo DNI es : %d , se llama %-40s y nacio el dia
%2d/%2d/%4d\n" , vectEmpleado[i].DNI , vectEmpleado[i].ApellidoNombre ,
vectEmpleado[i].DiaNacimiento , vectEmpleado[i].MesNacimiento ,
vectEmpleado[i].AnioNacimiento);
}
return 0;
}

```

A continuación, se podrá observar en forma gráfica la secuencia de anidamiento de los datos ingresados por teclado en un arreglo (vector) de tipo estructura de datos.



5. Como Anidar una de estructura de datos.

Las estructuras de datos permiten su anidamiento, con lo cual dentro de una estructura de datos puede haber una o varias estructuras de datos dentro de ella.

El “Ejemplo 5” podría modificarse para crear una estructura de datos que contenga la fecha de nacimiento, o sea generar una nueva estructura de datos que contenga el Dia, Mes y Año. Este proceso se puede visualizar en el “Ejemplo 6”.

Para visualizar y entender esta nueva ampliación se tomará en cuenta los miembros que conforman la fecha de nacimiento es decir el día, mes y año, y con estos datos se dará lugar a la formación de la típica estructura FECHA, y será una reutilización permanente porque se le podrá dar diferentes usos, por ejemplo: fecha de nacimiento, fecha de casamiento, fecha de ingreso, etc., todas ellas serán del mismo tipo de estructura.

Ejemplo 6:

/* Es un programa que permite ingresar por teclado los datos en la estructura PERSONA y a su vez mostrarlos por pantalla. Aquí se podrá ver cómo se utiliza el operador miembro de estructura “.(punto) */

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

// Aquí se declara la estructura de datos con el nombre FECHA
struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};

// Aquí se declara la estructura de datos con el nombre PERSONA
struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    struct FECHA Nacimiento;
    /* Aquí se crea una variable con los campos de la estructura de datos con el nombre FECHA. Cabe aclarar que la estructura de datos FECHA tiene que estar declarada previamente, de lo contrario daría un error de compilación.*/
};

int main()
{
    struct PERSONA empleado;
    printf("Ingrese Numero de DNI");
    scanf("%d", &empleado.DNI);
    printf("Ingrese Numero de Apellido y nombre");
    fflush(stdin);
    gets(empleado.ApellNombre);
    printf("Ingrese Sexo");
    fflush(stdin);
    scanf("%c", &empleado.Sexo);
    printf("Ingrese Numero del Dia de Nacimiento");
    scanf("%d", &empleado.Nacimiento.Dia);
    printf("Ingrese Numero del Mes de Nacimiento");
    scanf("%d", &empleado.Nacimiento.Mes);
    printf("Ingrese Numero del Año de Nacimiento");
    scanf("%d", &empleado.Nacimiento.Anio);
    printf("El empleado cuyo DNI es : %d , se llama %-40s y nacio el dia %2d/%2d/%4d", empleado.DNI , empleado.ApellNombre, empleado.Nacimiento.Dia , empleado.Nacimiento.Mes , empleado.Nacimiento.Anio);
    return 0;
}
```

En el “Ejemplo 6” se realizó la codificación de un ejercicio donde se accede a un miembro (campo) por medio de estructuras de datos anidadas, utilizando para ello tantos operadores miembros (“.”, punto) como niveles de anidamiento haya.

Aquí se puede observar en forma gráfica como quedaría el anidamiento de estructuras:

Estructura PERSONA

| DNI | Apellido y Nombre | Sexo | Nacimiento | | |
|------------------|-------------------|------|------------|-----|-----|
| | | | Dia | Mes | Año |
| Estructura FECHA | | | | | |

6. Como utilizar una estructura de datos en funciones.

Al formar un nuevo tipo de dato una estructura de datos puede utilizarse como parámetros. En el Ejemplo 7, se recibe como parámetro una estructura de datos que contiene una fecha y la función validará si la misma corresponde a una fecha correcta devolviendo: un entero “1” si es incorrecta o un “0” si es correcta.

Ejemplo7:

1- Llamada en el programa principal (función main):

```
esFechaCorrecta (fecha);
```

2- Prototipo de la función:

```
int esFechaCorrecta(struct FECHA );
```

3- Declaración de la función:

```
/* Aquí se coloca el encabezado de la función en donde se encuentra como parámetro de entrada una variable tipo estructura FECHA. */
```

```
int esFechaCorrecta (struct FECHA fecha)
{
    int retorno, bisiesto, cantidaddiasmes;
    retorno = 0;
    if(fecha.Anio%4==0 && fecha.Anio %100!=0 || fecha.Anio %400==0)
        bisiesto=1;
    else
        bisiesto =0;
    if(fecha.Mes==4|| fecha.Mes ==6|| fecha.Mes ==9|| fecha.Mes ==11)
        cantidaddiasmes =30;
    else
    {
        if(fecha.Mes ==2)
            cantidaddiasmes =28+ bisiesto;
        else
            cantidaddiasmes =31;
    }
    if(fecha.Anio    >=1900&&    fecha.Mes    >=1&&    fecha.Mes    <=12&&    fecha.Dia>=1&&
    fecha.Dia<= cantidaddiasmes )
        retorno =1;
    else
        retorno =0;
    return retorno;
}
```

En el “Ejemplo 8” se puede observar la construcción de una función que retorna una variable tipo estructura FECHA, que contendrá en cada uno de sus miembros (campos) los datos correspondientes a una fecha.

Ejemplo 8:

1- Llamada en el programa principal (función main):

```
struct FECHA f;  
f = IngresoDeFecha ( );
```

2- Prototipo de la función:

```
struct FECHA IngresoDeFecha ( );
```

3- Declaración de la función:

```
struct FECHA IngresoDeFecha ( )  
{  
    // Aquí se declara una variable local tipo estructura fecha  
    struct FECHA fecha;  
    printf("Ingrese Numero del Dia de una fecha");  
    scanf("%d", &fecha.Dia);  
    printf("Ingrese Numero del Mes de una fecha");  
    scanf("%d", &fecha.Mes);  
    printf("Ingrese Numero del Año de una fecha");  
    scanf("%d", &fecha.Anio);  
    return fecha // Aquí se retorna la variable tipo estructura FECHA  
}
```

En el “Ejemplo 9” se puede observar un programa en cual se utilizarán las funciones descriptas en el “Ejemplo 6” y “Ejemplo 7”.

Ejemplo 9:

Es un programa que permite ver el uso de estructuras en funciones.

```
#include <stdio.h>  
#include <conio.h>  
struct FECHA  
{  
    int Dia;  
    int Mes;  
    int Anio;  
};  
struct FECHA IngresoDeFecha(); // Prototipo de la función  
int esFechaCorrecta(struct FECHA ); // Prototipo de la función  
int main()  
{  
    struct FECHA fecha_main; // Declaración de una variable tipo estructura FECHA.  
    int retorno;  
    fecha_main = IngresoDeFecha(); // llamada a la función  
    retorno = esFechaCorrecta(fecha_main);  
    if( retorno == 1)  
        printf("La fecha es correcta");  
    else  
        printf("Es una fecha Incorrecta");  
    return 0;  
}  
  
struct FECHA IngresoDeFecha()  
{  
    struct FECHA fecha;  
    printf("Ingrese Numero del Dia de una fecha:");
```

```
scanf("%d", &fecha.Dia);
printf("Ingrese Numero del Mes de una fecha:");
scanf("%d", &fecha.Mes);
printf("Ingrese Numero del Anio de una fecha:");
scanf("%d", &fecha.Anio);
return fecha;// Aquí se retorna la variable tipo estructura FECHA
}

int esFechaCorrecta (struct FECHA fecha)
{
    int retorno, bisiesto, cantidaddiasmes;
    retorno = 0;
    if(fecha.Anio%4==0 && fecha.Anio %100!=0 || fecha.Anio %400==0)
        bisiesto=1;
    else
        bisiesto =0;
    if(fecha.Mes==4|| fecha.Mes ==6|| fecha.Mes ==9|| fecha.Mes ==11)
        cantidaddiasmes =30;
    else
    {
        if(fecha.Mes ==2)
            cantidaddiasmes =28+ bisiesto;
        else
            cantidaddiasmes =31;
    }
    if(fecha.Anio    >=1900&&   fecha.Mes    >=1&&   fecha.Mes    <=12&&   fecha.Dia>=1&&
    fecha.Dia<= cantidaddiasmes )
        retorno =1;
    else
        retorno =0;
    return retorno;
}
```

7. Copia de estructuras

Las estructuras forman un nuevo tipo de dato y por lo tanto permiten su manejo como una unidad. Si se tienen dos variables de la misma estructura es posible asignar directamente una a otra con el símbolo de asignación (=).

Automáticamente todos los campos de la estructura a la derecha del signo igual se copian a la segunda estructura sin importar de qué tipo sean dichos campos o si hay estructuras anidadas, vectores, etc. todos los campos se copian.

Ejemplo 10:

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

struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};

struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    struct FECHA Nacimiento;
};
```

```
int main()
{
    struct PERSONA p, copia;

    //asignación de datos a la variable p
    p.DNI = 23223122;
    strcpy(p.Apellido, "JUAN PEREZ");
    p.Sexo = 'M';
    p.Nacimiento.Dia = 10;
    p.Nacimiento.Mes = 5;
    p.Nacimiento.Año = 1982;

    copia = p; //copia de la variable estructura

    printf("Los datos copiados son:\n Nombre: %s \n Sexo: %c \n DNI: %d \n Fecha de
nacimiento: %d/%d/%d" , copia.Apellido, copia.Sexo, copia.DNI,
copia.Nacimiento.Dia, copia.Nacimiento.Mes, copia.Nacimiento.Año);

    getch();
    return 0;
}
```

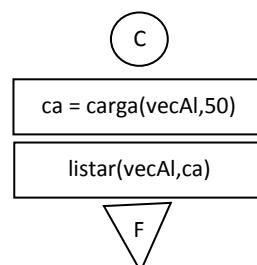
8. Estructuras con vectores

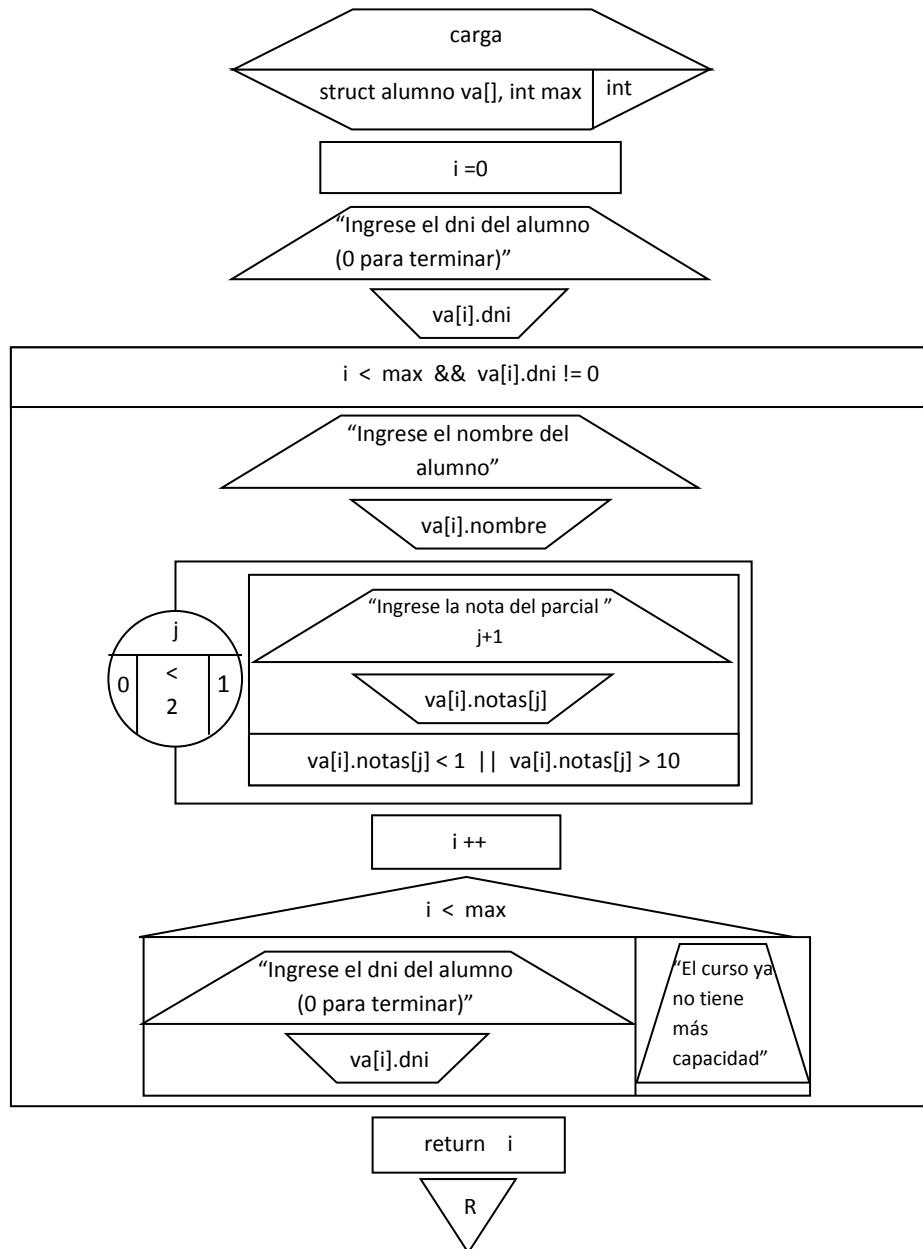
Una estructura puede contener cualquier otro tipo de dato dentro, incluyendo otras estructuras (como se vio en ejemplos anteriores) y también puede tener arrays (vectores y matrices). Por lo tanto, al momento de desarrollar el programa se debe prestar atención en donde colocar los subíndices ya que no debe confundirse un vector de estructuras con una estructura con un vector como campo.

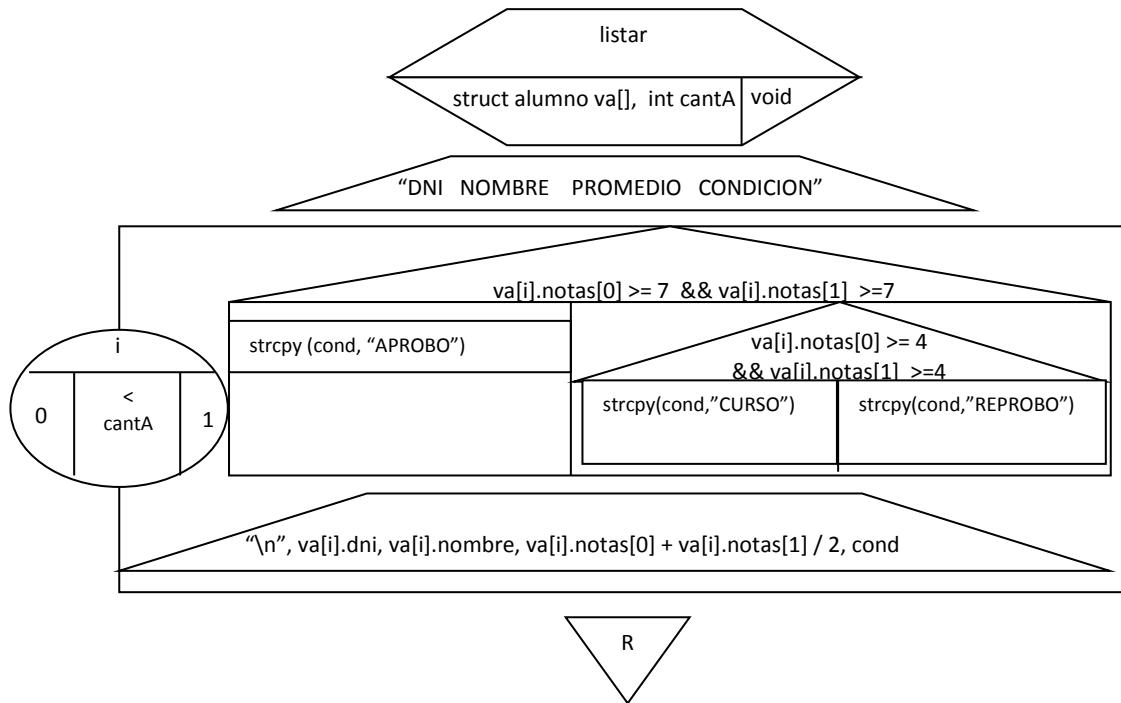
El siguiente ejemplo define la estructura alumno donde uno de sus campos es un vector con las notas de los dos parciales.

```
struct alumno
{
    char nombre [31];
    int dni;
    int notas[2];
};
```

Ejemplo 11: Ingresar los nombres, dni y notas de los alumnos de un curso. Se sabe que no son más de 50 personas (se finaliza con un dni igual a 0). Al finalizar el ingreso mostrar un listado con el promedio y la condición final de cada uno.







```

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

struct alumno
{
    char nombre [31];
    int dni;
    int notas[2];
};

int carga(struct alumno[], int);
void listar(struct alumno[], int);

int main()
{
    struct alumno vecAl[50];
    int ca;
    ca = carga(vecAl,50);
    listar(vecAl,ca);
    getch();
    return 0;
}

int carga(struct alumno va[], int max)
{
    int i=0,j;
    printf("Ingrese el dni del alumno (0 para terminar): ");
    scanf("%d", &va[i].dni);
    while (i<max && va[i].dni!=0)
    {
        printf("Ingrese el nombre del alumno:");
        getchar();
        gets(va[i].nombre);
        for (j=0;j<2;j++)
        {
            do
            {
                printf("Ingrese la nota del parcial %d: ", j+1);
                scanf("%d", &va[i].notas[j]);
            }while(va[i].notas[j]<1 || va[i].notas[j]>10);
        }
    }
}

```

```
i++;
if (i<max)
{
    printf("Ingrese el dni del alumno (0 para terminar): ");
    scanf("%d", &va[i].dni);
}
else
{
    printf("El curso ya no tiene mas capacidad.\n");
}
}
return i;
}

void listar(struct alumno va[], int cantA)
{
    int i;
    char cond[9];
    printf ("%10s%31s%9s%10s", "DNI", "NOMBRE", "PROMEDIO" , "CONDICION");
    for (i=0;i<cantA;i++)
    {
        if (va[i].notas[0]>=7 && va[i].notas[1]>=7)
            strcpy(cond, "APROBO");
        else
            if (va[i].notas[0]>=4 && va[i].notas[1]>=4)
                strcpy(cond, "CURSO");
            else
                strcpy(cond, "REPROBO");

        printf      ("\n%10d%31s%9.2f%10s",           va[i].dni,va[i].nombre,      (va[i].notas[0]
+va[i].notas[1])/2.,cond);
    }
}
```

9. Otra forma de declarar las estructuras

La forma que se mostró hasta ahora para declarar una estructura es utilizando la palabra reservada struct seguido del nombre de la estructura y entre llaves sus campos. Por ejemplo, la estructura fecha que fue utilizada en ejemplos anteriores se define de la siguiente forma:

```
struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};
```

Al definir una estructura de este tipo vimos que al referirnos al nuevo tipo de dato en todos lados debemos poner struct FECHA, es decir siempre hay que poner la palabra reservada struct seguido del nombre de la estructura creada.

Pero existe una forma alternativa para declarar una estructura que elimina la necesidad de escribir siempre la palabra reservada struct para definir una variable del nuevo tipo de dato. Para ello se utiliza otra palabra reservada typedef declarando la estructura de la siguiente forma:

```
typedef struct
{
    int Dia;
    int Mes;
    int Anio;
}FECHA;
```

Esta forma de declarar la estructura hace que no se deba utilizar la palabra reservada struct al declarar una variable ya que FECHA por sí misma ya es un nuevo tipo de dato. La declaración de una variable entonces se hace de la siguiente forma:

```
FECHA fe;
```

Es decir, solo se pone el nombre del nuevo tipo de dato y luego el identificador de la variable a definir. De igual manera al escribir los parámetros de funciones no se debe poner la palabra struct. Esta forma alternativa reduce un poco el código fuente. Pueden utilizarse indistintamente cualquiera de las dos formas según preferencia del programador.

10. Ordenar un vector de estructuras

Para ordenar un vector de estructuras se puede utilizar cualquier método de ordenamiento. Anteriormente se vio el método por burbujeo en el cual se requería de realizar intercambios para ir ordenando los datos. En el caso de los vectores paralelos, se debían realizar varios intercambios para que la información no quede mezclada. La ventaja de los vectores de estructuras es que al definir un nuevo tipo de dato la variable auxiliar para el intercambio será directamente del tipo estructura permitiendo en una sola asignación copiar todos sus campos. El ejemplo 12 muestra el desarrollo de un ejercicio donde se ordena un vector de estructuras.

Ejemplo 12:

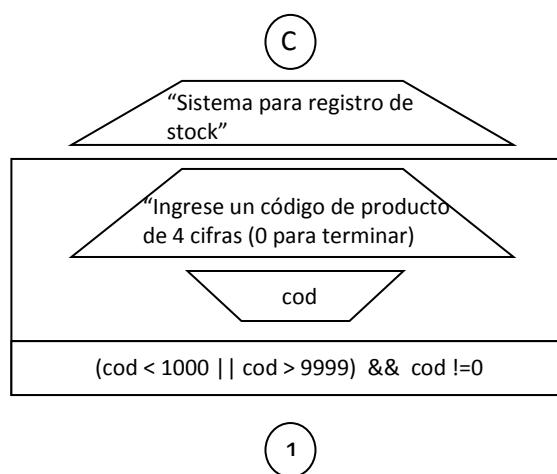
Realizar un sistema que permita hacer el inventario de productos en una fábrica. Para ello por cada producto se debe ingresar

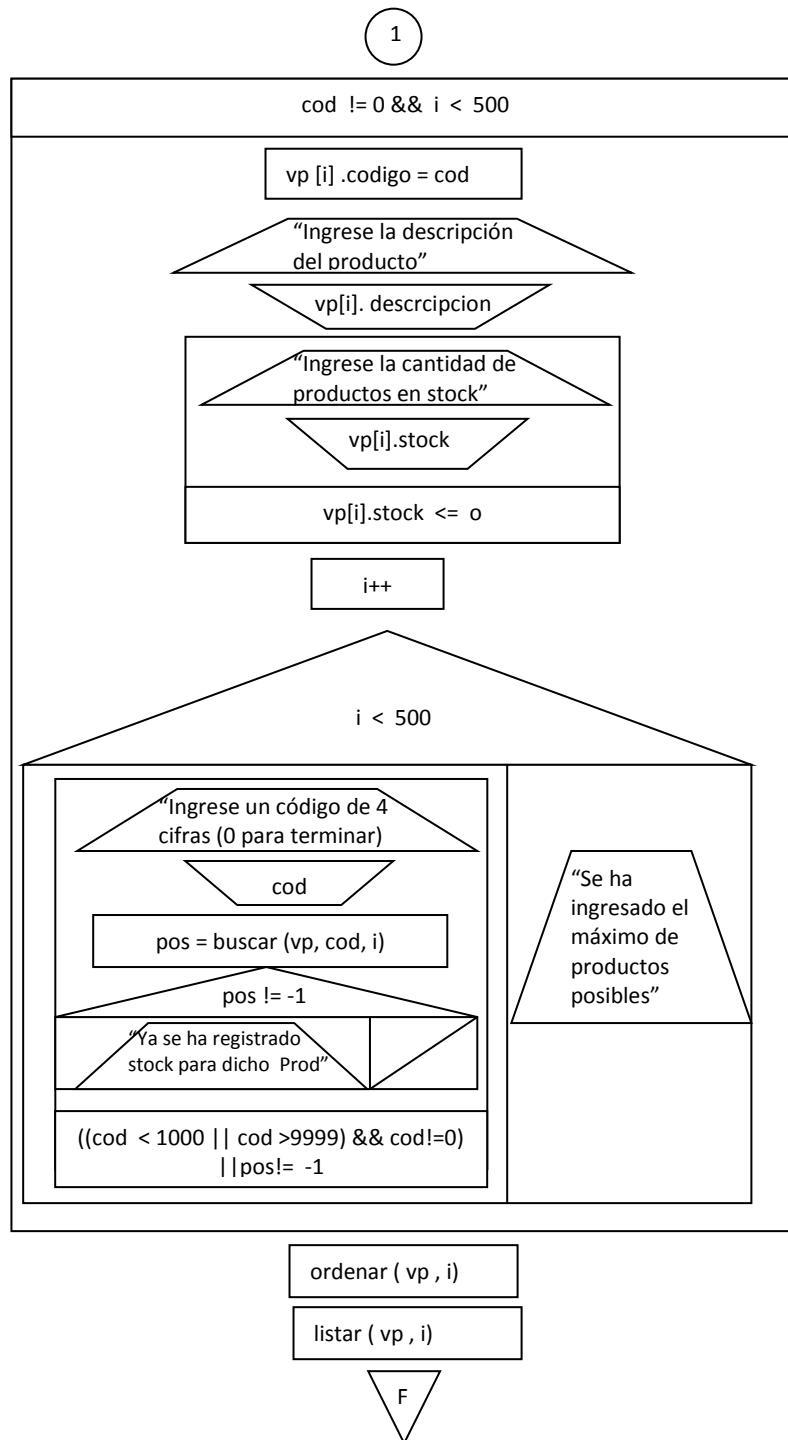
- Código (entero de 4 cifras)
- Descripción (texto de 20 caracteres máximo)
- Cantidad de productos en stock (entero mayor a 0)

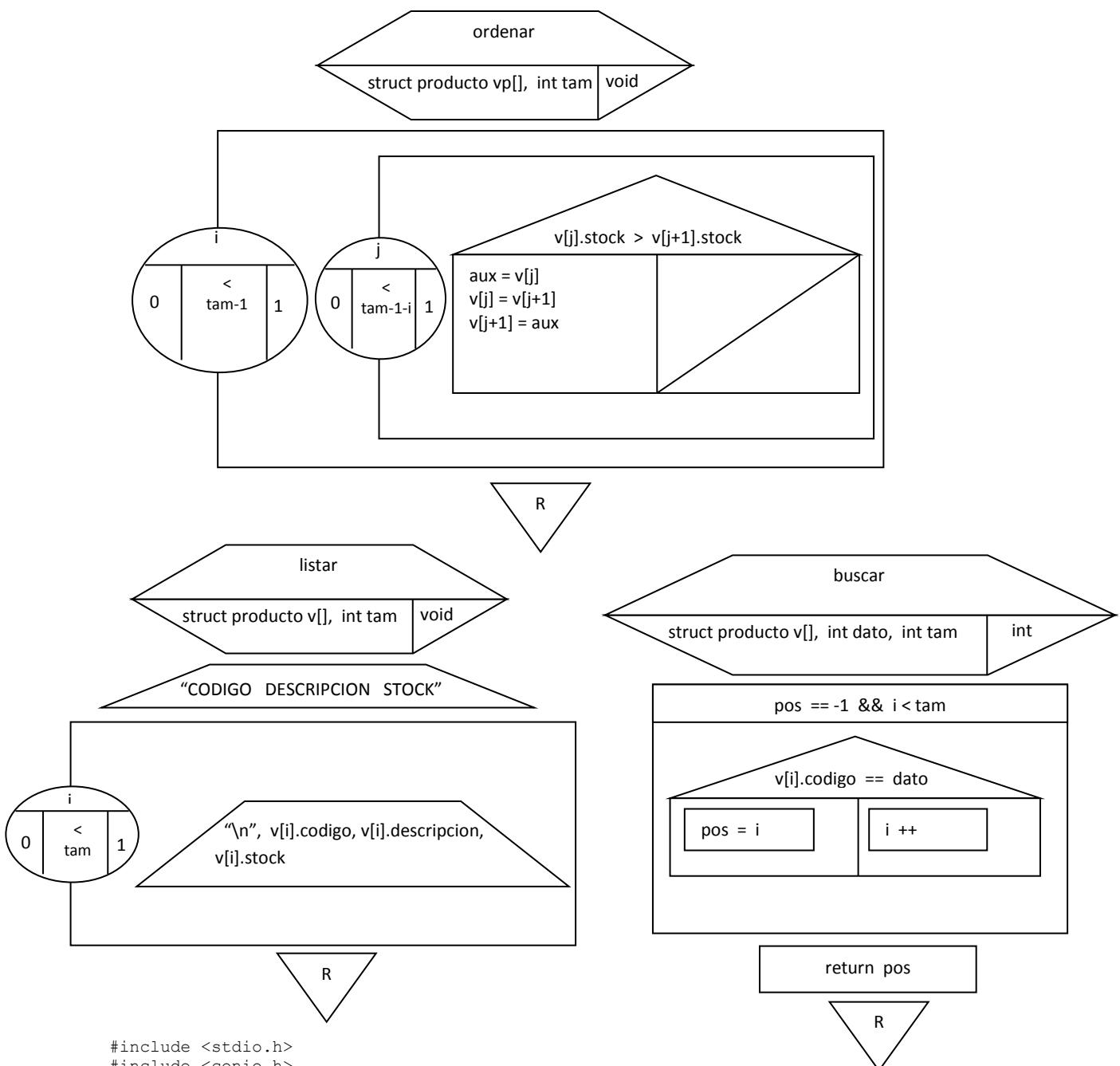
No se sabe la cantidad total de productos, pero sí se sabe que no hay más de 500.

La carga de productos finaliza con un código de producto igual a 0.

Al finalizar mostrar en forma ordenada de mayor a menor por cantidad en stock los productos.







```

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

struct producto
{
    int codigo;
    char descripcion[21];
    int stock;
};

int buscar(struct producto[], int, int);
void ordenar (struct producto[], int);
void listar (struct producto[], int);

int main()
{
    struct producto vp[500];
    int cod, i=0, pos;
    printf ("SISTEMA PARA REGISTRO DE STOCK\n\n");

    do
    {
        printf ("Ingrese un codigo de producto de 4 cifras (0 para terminar):");
    }

```



```
scanf("%d", &cod);
}while ((cod<1000 || cod > 9999) && cod!=0);

while (cod!=0 && i<500)
{
    vp[i].codigo = cod;

    getchar(); //limpia el buffer de teclado
    printf("Ingrese la descripcion del producto:");
    gets(vp[i].descripcion);

    do
    {
        printf("Ingrese la cantidad de productos en stock:");
        scanf("%d", &vp[i].stock);
    }
    while(vp[i].stock<=0);

    i++;
    if (i<500)
    {

        do
        {
            printf ("Ingrese un codigo de producto de 4 cifras (0 para terminar):");
            scanf("%d", &cod);
            pos = buscar(vp, cod, i);
            if (pos!=-1)
                printf("Ya se ha registrado stock para dicho producto.\n");
        }while (((cod<1000 || cod > 9999) && cod!=0 ) || pos!=-1);

        }
        else
            printf ("Se ha ingresado el maximo de productos posible");
    }

ordenar(vp, i);
listar(vp, i);
getch();
return 0;
}

int buscar(struct producto v[], int dato, int tam)
{
    int i=0, pos=-1;
    while (pos== -1 && i<tam)
    {
        if (v[i].codigo == dato)
            pos =i;
        else
            i++;
    }
    return pos;
}

void ordenar (struct producto v[], int tam)
{
    int i,j;
    struct producto aux;

    for (i=0;i<tam-1;i++)
        for (j=0;j<tam-1-i;j++)
            if (v[j].stock > v[j+1].stock)
            {
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] =aux;
            }
    }

void listar (struct producto v[], int tam)
{
    int i;
    printf ("\n%6s%21s%6s", "CODIGO", "DESCRIPCION", "STOCK");
    for (i=0;i<tam;i++)
        printf ("\n%6d%21s%6d", v[i].codigo, v[i].descripcion, v[i].stock);
}
```



Elementos de Programación

UNIDAD 10. MANEJO DE ARCHIVOS

INDICE

| | | |
|----|--|----|
| 1. | QUE ES UN ARCHIVO BINARIO. | 2 |
| 2. | VARIABLE PUNTERO A UN ARCHIVO | 3 |
| 3. | COMO ABRIR UN ARCHIVO. APERTURA DE UN ARCHIVO (FUNCIÓN FOPEN()) | 3 |
| 4. | COMO CERRAR UN ARCHIVO. CIERRE DE UN ARCHIVO (FUNCIÓN FCLOSE()) | 6 |
| 5. | COMO LEER UN REGISTRO DE UN ARCHIVO. FUNCIÓN DE ENTRADA (FUNCIÓN FREAD()) | 6 |
| 6. | COMO ESCRIBIR UN REGISTRO DE UN ARCHIVO. FUNCIÓN DE SALIDA (FUNCIÓN FWRITE()) | 7 |
| 7. | COMO LEER UN ARCHIVO SIN CONOCER LA CANTIDAD DE REGISTROS. FUNCIÓN FEOF(). | 9 |
| 8. | EJEMPLOS DEL USO DE ARCHIVOS CON ESTRUCTURAS DE DATOS | 9 |
| 9. | EXPORTAR DATOS PARA OTROS PROGRAMAS..... | 15 |

UNIDAD 10 - MANEJO DE ARCHIVOS

OBJETIVOS: Realizar programas que puedan almacenar datos y resultados en una memoria no volátil a fin de respaldar la información generada permitiendo construir programas que no pierdan los datos cargados al finalizar su ejecución.

1. Que es un archivo binario.

Los datos con los que se han trabajado hasta el momento han residido en la memoria principal. Sin embargo, las grandes cantidades de datos se almacenan normalmente en un dispositivo de memoria secundaria. Estas colecciones de datos se conocen como archivos (en inglés **FILE**).

Un **archivo** es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas **estructura de datos** (registros) que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajos denominadas **miembros** (campos).

Hay dos tipos de archivos, archivos de texto y archivos binarios. Un **archivo de texto** es una secuencia de caracteres organizadas en líneas terminadas por un carácter de nueva línea. En estos archivos se pueden almacenar canciones, fuentes de programas, base de datos simples, etc. Los **archivos de texto** se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho.

Un **archivo binario** es una secuencia de bytes que tienen una correspondencia uno a uno con un dispositivo externo. Así que no tendrá lugar ninguna traducción de caracteres. Además, el número de bytes escritos (leídos) será el mismo que los encontrados en el dispositivo externo. Ejemplos de estos archivos son Fotografías, imágenes, texto con formatos, archivos ejecutables (aplicaciones), etc.

Ejemplo de una Estructura de Datos:

| Estructura PERSONA | | | | | |
|--------------------|-------------------|------|------------|-----------|-----------|
| DNI | Apellido y Nombre | Sexo | | | |
| | | | Dia Nacim. | Mes Nacim | Año Nacim |

Ejemplo de un Archivo “datos.dat” que utiliza la Estructura de Datos antes descripta:

| | | | |
|--------------------|--------------------|--------------------|------|
| Estructura PERSONA | Estructura PERSONA | Estructura PERSONA | NULL |
|--------------------|--------------------|--------------------|------|

Ejemplo de un Archivo “datos.dat” que utiliza los miembros de la Estructura de Datos antes descripta:

| | | |
|---|---|------|
| DNI Apellido y Nombre Sexo Dia Nacim. Mes Nacim Año Nacim | DNI Apellido y Nombre Sexo Dia Nacim. Mes Nacim Año Nacim | NULL |
|---|---|------|

En el **Lenguaje C**, un archivo es un concepto lógico que puede aplicarse a muchas cosas desde archivos de disco hasta terminales o una impresora. Se asocia una sentencia con un archivo específico realizando una operación de apertura. Una vez que el archivo está abierto, la información puede ser intercambiada entre este y el programa.

Se puede conseguir la entrada y la salida de datos a un archivo a través del uso de la biblioteca de funciones; el **Lenguaje C** no tiene palabras claves que realicen las operaciones de E/S. La siguiente tabla da un breve resumen de las funciones que se pueden utilizar. Se debe incluir la biblioteca **<stdio.h>**. Observe que la mayoría de las funciones comienzan con la letra “F”.

| | |
|------------------|--|
| fopen () | Abre un archivo. |
| fclose () | Cierra un archivo. |
| fread () | Lee un registro del archivo. |
| fwrite () | Guarda un registro en el archivo. |
| feof() | Devuelve un número distinto de 0 si se llega al final del archivo. |

2. Variable puntera a un archivo

El puntero a un archivo es el hilo común que unifica el sistema de E/S con buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia identifica un archivo específico y utiliza la sentencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Un puntero a un archivo es una variable de tipo puntero al tipo de dato **FILE** que se define en **<stdio.h>**. Un programa necesita utilizar punteros a archivos para leer o escribir en los mismos. Para declarar una variable de este tipo se utiliza la siguiente sentencia:

FILE * identificador;

Por ejemplo, si se desea declarar un puntero llamado pf, se escribe:

FILE * pf;

Si es necesario declarar más de un puntero a archivo a cada uno se le debe anteponer el asterisco. El siguiente ejemplo define dos punteros pf y pf2.

FILE *pf, *pf2;

3. Como Abrir un Archivo. Apertura de un archivo (función fopen ())

Antes de utilizar un Archivo en un programa, ya sea para escribir datos en él y/o para leer datos de él, el Archivo debe ser abierto, con lo que se establece un canal de comunicación entre el programa y el Archivo. Y la función para abrir un Archivo se denomina **fopen ()** y su sintaxis es:

<pf> = fopen("<nombre_archivo>", "<modo_apertura>");

En diagrama de lógica:

<pf> = fopen("<nombre_archivo>", "<modo_apertura>");

Como puede verse, esta función devuelve un puntero a Archivo, **<pf>**, que será el puntero que quede asociado al Archivo abierto hasta que sea cerrado. Todas las operaciones sobre el Archivo se

realizarán a través de ese puntero <pf>. Para indicar el Archivo que se desea abrir se usa la cadena de caracteres <**nombre_archivo**>, la cual puede incluir, además del nombre del Archivo, la ruta completa donde está ubicado, es decir la unidad de almacenamiento y la carpeta. No olvidar que la barra invertida (\) de la ruta debe escribirse dos veces (\\), porque si se escribe sólo una vez es interpretada como secuencia de escape. Sino se coloca la ruta el archivo se busca en la misma carpeta donde se encuentra el ejecutable. Para que el programa pueda ser ejecutado en distintas computadoras sin errores se recomienda no poner una ruta fija.

Con la cadena <**modo_apertura**> se indicará si se va a usar el archivo en modo texto o en modo binario, además servirá para especificar si se va a leer del archivo, se va a escribir en él o ambas cosas. En esta materia utilizaremos archivos binarios y sobre estos archivos se pueden emplear los siguientes modos de apertura:

| Modo de apertura | |
|-------------------------|---|
| Modo | Significado |
| rb | Abre un archivo para lectura (read) |
| wb | Crea un archivo para escritura (write) |
| ab | Abre un archivo para añadir (add) |
| r+b | Abre un archivo para lectura/escritura |
| w+b | Crea un archivo para lectura/escritura |
| a+b | Abre o crea un archivo para lectura/escritura |

Como puede verse en la tabla, el <**modo_apertura**> indicará qué debe hacer el sistema con el archivo, dependiendo de si existe o no. Por ejemplo, abrir para lectura un archivo que no exista producirá un error. Además, el <**modo_apertura**> también establece donde apuntará el puntero al abrir el fichero, al principio o al final de este. Por ejemplo, si deseamos añadir datos al archivo, el puntero deberá colocarse al final, mientras que, si queremos leer todos los datos, deberá colocarse al principio. Por otro lado, un archivo abierto solo para lectura no permitirá realizar escrituras en él, a no ser que se cierre y se vuelve a abrir para escritura. En la siguiente tabla se concretan todos estos casos.

| Modo | Acción | Archivo ya existe | Archivo no existe | Lectura | Escritura | Posición Puntero |
|-------------|---------------|--------------------------|--------------------------|----------------|------------------|-------------------------|
| rb | Lectura | Correcto | * Error * | Correcto | * Error * | Principio |
| wb | Escritura | Borra contenido | Se crea | * Error * | Correcto | Principio |
| ab | Añadir | Correcto | Se crea | * Error * | Correcto | Final |
| r+b | Lect/Esc | Correcto | * Error * | Correcto | Correcto | Principio |
| w+b | Lect/Esc | Borra contenido | Se crea | Correcto | Correcto | Principio |
| a+b | Lect/Añadir | Correcto | Se crea | Correcto | Correcto | Principio |

En la tabla se observa que la apertura de un Archivo con el modo “wb” puede ser peligrosa, ya que si el Archivo existe se perderán todos sus datos. Aunque hay situaciones en las que es esa acción precisamente la que se quiere realizar, destruir todos los datos previos del Archivo. Por otra parte, debe tenerse en cuenta que si se utiliza el modo de apertura “ab” o “a+b”, los datos nuevos que se escriban en el Archivo nunca sobrescriben otros datos, sino que siempre se añaden al final, aunque el puntero esté en otra posición. El puntero apuntará al final después de añadir los nuevos datos. Además, al abrir un Archivo con el modo de apertura “ab”, el puntero inicialmente se coloca al principio, aunque después de añadir algún dato el puntero automáticamente pasa al final.

Nota: En esta materia se utilizan solo los tres primeros modos de apertura (rb, wb y ab) el resto de los modos se verán en materias siguientes y requieren del uso de la función fseek para posicionar el puntero en el lugar deseado del archivo para leer o escribir.

Si la función fopen () tiene éxito, es decir abre el Archivo sin ningún problema, devolverá el puntero al Archivo. Por el contrario, si se ha producido algún error al intentar la apertura devuelve el valor NULL. Por tanto, después de ejecutar la función fopen (), el programa deberá comprobar siempre si el puntero devuelto vale NULL.

Ejemplos:

```

FILE *fp;
//Especificando unidad y nombre de Archivo
if((fp = fopen("C:\\cursos.dat", "r+b")) == NULL)
{
    printf ("ERROR");
    getch();
    exit(1);
}

```

```
//Especificando unidad, carpeta y nombre de Archivo
if ((fp = fopen ("C:\\BC\\cursos.dat", "a+b")) == NULL)
{
    printf ("ERROR");
    getch();
    exit(1);
}
```

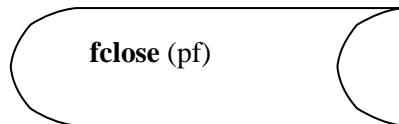
4. Como Cerrar un Archivo. Cierre de un archivo (función fclose ())

Cuando un programa deja de necesitar un Archivo, éste debe cerrarse, cortándose por tanto el canal de comunicación entre el programa y el Archivo. Para cerrar un Archivo se utiliza la función **fclose ()**.

La función **fclose ()** cierra un Archivo que fue abierto mediante una llamada a **fopen ()**. Escribe toda la información que todavía se encuentre en el buffer en el disco y realiza un cierre formal del archivo a nivel del sistema operativo. Un error en el cierre de un Archivo puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y posibles errores intermitentes en el programa, cuya sintaxis es:

```
fclose (pf);
```

En Diagrama de Lógica:



Donde “pf” es el puntero al archivo devuelto por la llamada a **fopen ()**. Generalmente, esta función solo falla cuando un disco se ha retirado antes de tiempo o cuando no queda espacio libre en el mismo.

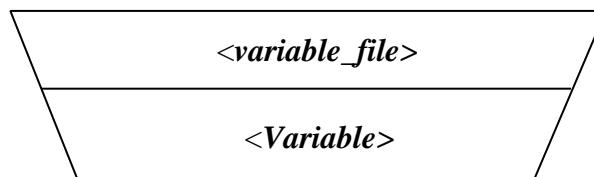
5. Como leer un registro de un archivo. Función de Entrada (función fread ())

La función **fread ()** trabaja con registros de longitud constante. Es capaz de leer desde un Archivo, uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. Se debe asegurar de que haya espacio suficiente para contener la información leída. La sintaxis de esta función es:

En lenguaje C:

```
fread( <Dir_Variable> , < nº_bytes>, <nº_cuenta>, <variable_file>);
```

En diagrama de lógica:



La función **fread ()** significa que se leerá del Archivo que apunta la **<variable_file>** el número de bytes **<nº_bytes>** tantas veces como indique **<nº_cuenta>**, dejando dichos bytes grabados en

memoria a partir de la dirección <Dir_Variable>. Esta dirección será la de una variable con el tamaño suficiente para guardar los datos leídos. El número de bytes que se leen será por tanto el resultado del producto: <nº_bytes> * <nº_cuenta>.

Ejemplos:

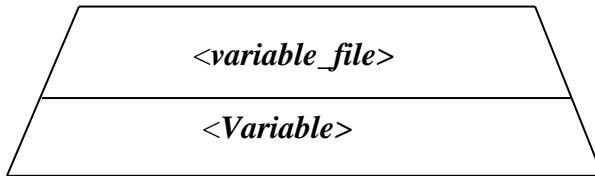
```
int num;  
fread(&num, sizeof(int), 1, fp);  
//Se leen 4*1=4 bytes del archivo, que caben en num.  
float Notas[5];  
fread(Notas, sizeof(float), 5, fp);  
//Se leen 4*5=20 bytes del fichero, que caben en Notas
```

En el Ejemplo N°1, en el segundo caso se leerán 20 bytes, ya que se leerán 5 veces el número de bytes indicado por sizeof(float), o sea $4*5 = 20$. Por tanto, se leen 5 números float desde el archivo y se guardan en el array Notas, cuyo tamaño es de 20 bytes.

6. Como escribir un registro de un archivo. Función de Salida (función **fwrite()**)

La función **fwrite()** permite escribir datos en un archivo como una sucesión de bytes. La sintaxis de esta función es:

```
fwrite( <Dir_Variable> , < nº_bytes>, <nº_cuenta>, <variable_file>);
```



A partir de la dirección de memoria <Dir_Variable> se leerán tantos bytes como se indique en <nº_bytes> tantas veces como se especifique en <nº_cuenta> y se escribirán en el Archivo que apunta la <variable_file>. El número de bytes escritos será por tanto el resultado del producto: <nº_bytes> * <nº_cuenta>.

Ejemplos:

```
int num=1067;  
fwrite(&num, sizeof(int), 1, fp);  
//Graba 4 bytes en el archivo, desde la dirección num, por tanto el  
valor 1067 queda escrito en el archivo.  
float Notas[5]={5.5, 7.25, 8.5, 4.75, 9.5};  
fwrite(Notas, sizeof(float), 5, fp);  
//Se escriben 4*5=20 bytes en el archivo, desde la dirección Notas,  
quedando las 5 notas grabadas en el archivo.
```

La función **fwrite()** devuelve el número de veces que ha escrito el <nº_bytes>, que no siempre coincide con <cuenta>, ya que puede producirse algún error. Por tanto, si el valor que devuelve **fwrite()** no coincide con <cuenta> es que ha ocurrido un error.

Ejemplo 1. Escribir un número real en un Archivo. Leerlo en otra variable real.

```
FILE *fp;  
float var;  
float num = 12.23;  
if ((fp = fopen("prueba.dat", "wb")) == NULL )
```

```
{  
    printf("No se puede abrir.\n");  
    getch(); exit(1);  
}  
fwrite( &num, sizeof(float), 1, fp);  
fclose(fp);  
if ((fp = fopen("prueba.dat", "rb")) == NULL )  
{  
    printf("No se puede abrir.\n");  
    getch(); exit(1);  
}  
fread (&var, sizeof(float), 1, fp );  
fclose(fp);
```

Uno de los usos más comunes de **fread()** y **fwrite()** es el manejo de Archivo en forma de conjunto de registros, donde cada registro está dividido en campos. Para ello en el programa debe definirse un tipo de estructura de datos con el mismo formato que el registro del Archivo, con el objeto de leer y escribir registros completos con las funciones **fread()** y **fwrite()** respectivamente, en lugar de leer y escribir campos concretos.

Ejemplo 2. Teniendo un vector de estructuras ya cargado con datos, deberá grabarse en un Archivo y después dejarlo en otro vector de estructuras.

```
struct datos  
{  
    char nombre[20];  
    char apellido[40];  
};  
  
int main ()  
{  
    FILE *fp;  
    int i;  
    struct datos lista1[30], lista2[30];  
    //Suponemos que lista1 ya contiene datos.  
  
    if ( (fp = fopen("fich.dat", "wb")) == NULL )  
    {  
        printf("No se puede abrir.\n");  
        getch(); exit(1);  
    }  
    //Escribe 30 registros desde el array lista1.  
    for ( i = 0; i < 30; i++)  
        fwrite(&lista1[i],sizeof(struct datos),1,fp)  
    fclose(fp);  
    if ( (fp = fopen("fich.dat", "rb")) == NULL )  
    {  
        printf("No se puede abrir.\n");  
        getch(); exit(1);  
    }  
  
    //Lee 30 registros, dejándolos en el array lista2.  
    for ( i = 0; i < 30; i++)  
        fread(&lista2[i], sizeof(struct datos),1,fp)  
    fclose(fp);  
    return 0;  
}
```

7. Como leer un archivo sin conocer la cantidad de registros. Función feof().

Cuando se abre un archivo para lectura binaria, se puede leer un valor entero igual al de la marca de fin de archivo (EOF). Esto podría hacer que la rutina de lectura indicase una condición de fin de archivo aún cuando el fin físico del mismo no se haya alcanzado. Para resolver este problema, **lenguaje C** incluye la función **feof()**, que determina cuando se ha alcanzado el fin del archivo leyendo datos binarios. La función tiene el siguiente prototipo

```
int feof(FILE *);
```

Su prototipo se encuentra en <stdio.h>. Devuelve un número distinto de 0 si se ha alcanzado el final del archivo, si aún hay datos devuelve un 0.

Cuando no se conoce la cantidad de registros que posee un Archivo, se puede utilizar la función **feof()**, la cual permitirá leer la información hasta el fin de Archivo (ver **Ejemplo 3**).

Esta función debe utilizarse **luego** de hacer una lectura sobre el archivo con fread.

Ejemplo 3:

```
struct datos
{
    char nombre[20];
    char apellido[40];
};

int main ( )
{
    FILE *fp;
    int i;
    struct datos lista[30];

    // Se supone que el archivo no va a contener más de 30 registros.

    if ( (fp = fopen("fich.dat", "rb")) == NULL )
    {
        printf("No se puede abrir.\n");
        getch(); exit(1);
    }
    //Lee los registros del archivo hasta el final del mismo.
    i = 0;
    fread(&lista[i], sizeof(struct datos), 1, fp)
    while ( !feof (fp) )
    {
        i++;
        fread(&lista[i], sizeof(struct datos), 1, fp)
    }
    fclose(fp);
    return 0;
}
```

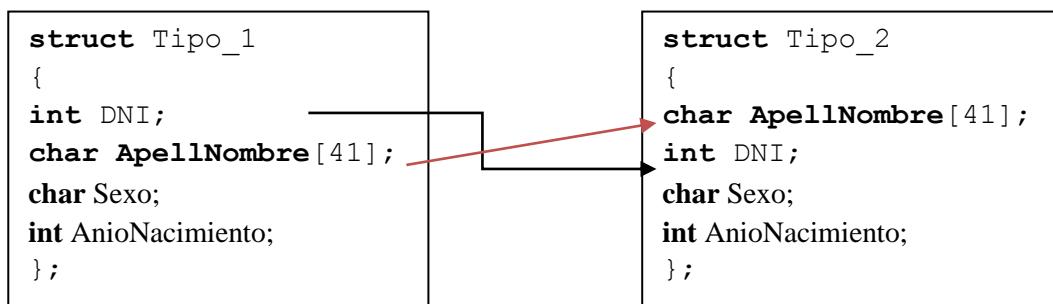
8. Ejemplos del uso de archivos con estructuras de datos

El uso de estructuras de datos en el manejo de “archivos binarios”, resulta imprescindible pues la esencia de estos archivos binarios es la grabación y recuperación de registros es decir estructuras de datos (**struct**).

Tanto para la recuperación (lectura) o grabación (escritura) de un registro del archivo el mismo se realiza a través de una variable tipo estructura (**struct**) que obedezca al diseño del registro del archivo.

Es importante tener en cuenta que para poder recuperar (lectura) o grabar (escritura) la información que se encuentra en un archivo se debe conocer exactamente como está formada la estructura, porque será la única manera que se pueda trabajar con archivos. En el ejemplo que se desarrolla a continuación se corresponden a dos estructuras distintas, aunque ambas posean los mismos datos.

La estructura “**struct Tipo_1**” y la estructura “**struct Tipo_2**” contienen los mismos datos y la misma cantidad de bytes (50 bytes), pero son al mismo momento diferentes porque la información dentro de ambas se encuentra en órdenes diferentes. Para que dos estructuras sean exactamente iguales deben coincidir totalmente y no es el caso anterior por lo cual son distintas e incompatibles en su tratamiento.



Ejemplo 4:

Es un programa que permite grabar los datos de 10 alumnos en un archivo llamado “AlumnosUNLaM.dat”.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};

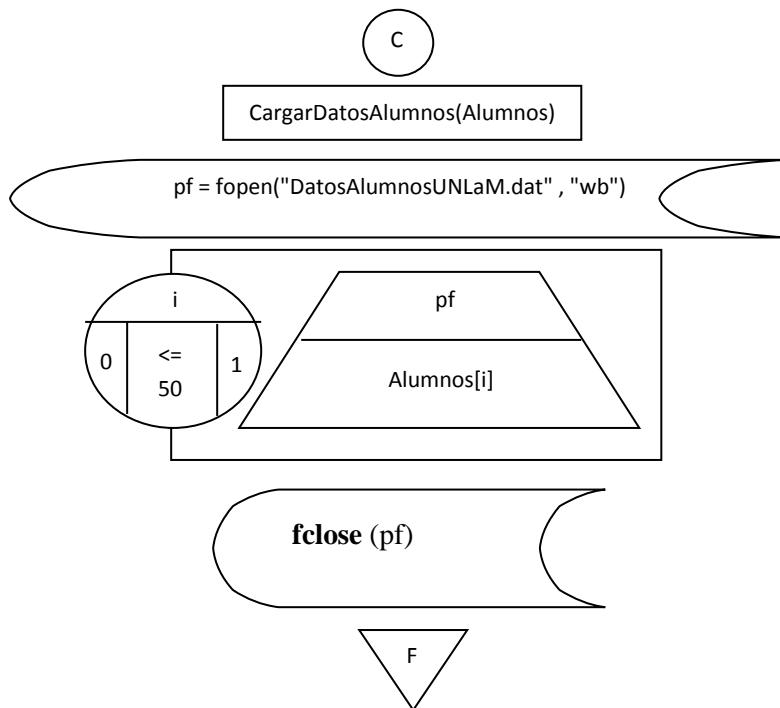
struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    struct FECHA Nacimiento;
};

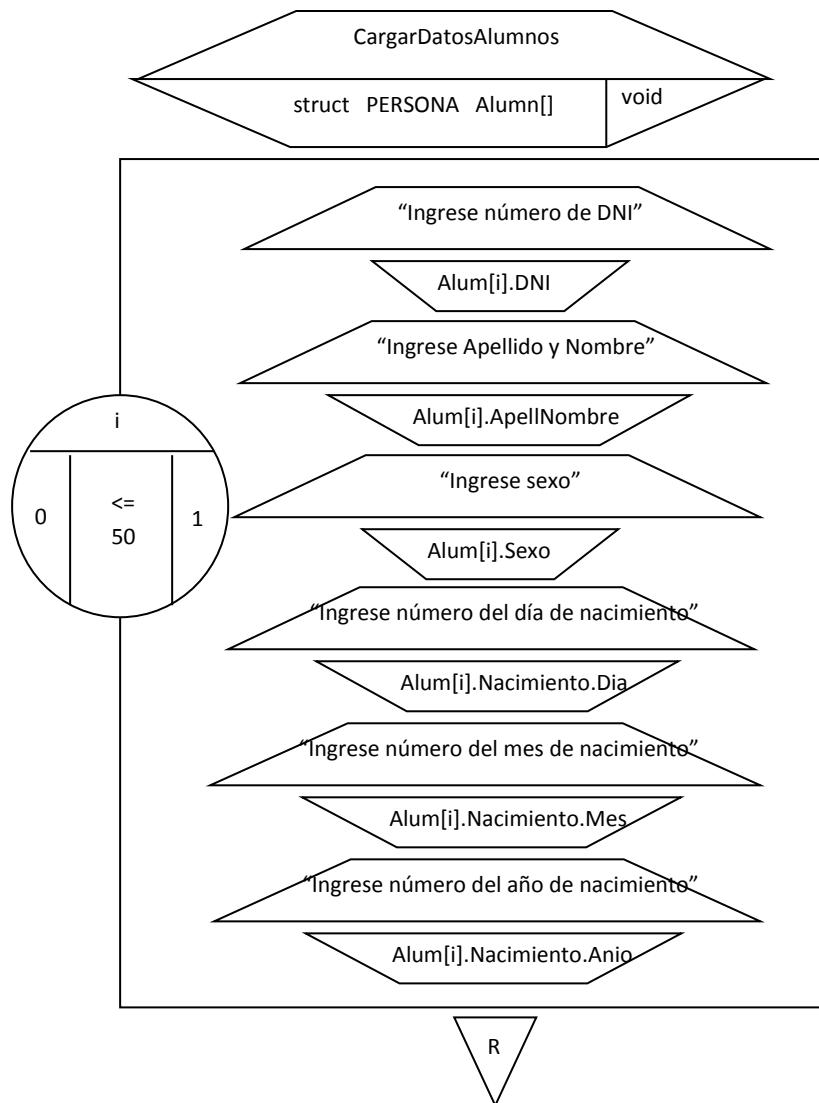
int main()
{
    struct PERSONA alumno;
    int i;
    FILE *pf;
    pf = fopen("AlumnosUNLaM.dat" , "wb");
    if (pf==NULL)
    {
        printf("No se pudo crear el archivo.");
        getch();
        exit (1);
    }
    for(i=0 ; i <= 10; i++)
    {
        printf("Ingrese Numero de DNI");
        scanf("%d", &alumno.DNI);
```

```
printf("Ingrese Numero de Apellido y nombre");
fflush(stdin);
gets(alumno.ApellNombre);
printf("Ingrese Sexo");
fflush(stdin);
scanf("%c", &alumno.Sexo);
printf("Ingrese Numero del Dia de Nacimiento");
scanf("%d", &alumno.Nacimiento.Dia);
printf("Ingrese Numero del Mes de Nacimiento");
scanf("%d", &alumno.Nacimiento.Mes);
printf("Ingrese Numero del Año de Nacimiento");
scanf("%d", &alumno.Nacimiento.Anio);
fwrite(&alumno, sizeof(struct PERSONA), 1, pf);
/*Aquí la función que permite grabar un registro en un archivo binario. Se indica el nombre de la variable estructura "alumno" (registro) que se va a grabar pf que es el puntero del archivo a grabar. La función sizeof determina el largo del registro por eso esta invocada la estructura PERSONA (58 bytes). */
}
return 0;
}
```

Ejemplo 5:

Es un programa que permite grabar los datos que se encuentra en un arreglo (vector) de estructura de datos (los cuales fueron ingresados por teclado dentro de una función llamada “CargaDatosAlumnos”), en un archivo llamado “DatosAlumnosUNLaM.dat”





```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};
struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    struct FECHA Nacimiento;
} ;
void CargaDatosAlumnos (struct PERSONA []); //Prototipo de la función
int main()
{
    struct PERSONA Alumnos[50];
    int i;
    FILE *pf;

    CargaDatosAlumnos( Alumnos );// Llamada de la función
    pf = fopen("DatosAlumnosUNLaM.dat" , "wb");
    if (pf==NULL)
    {
        printf("No se pudo crear el archivo.");
        getch();
    }
}

```

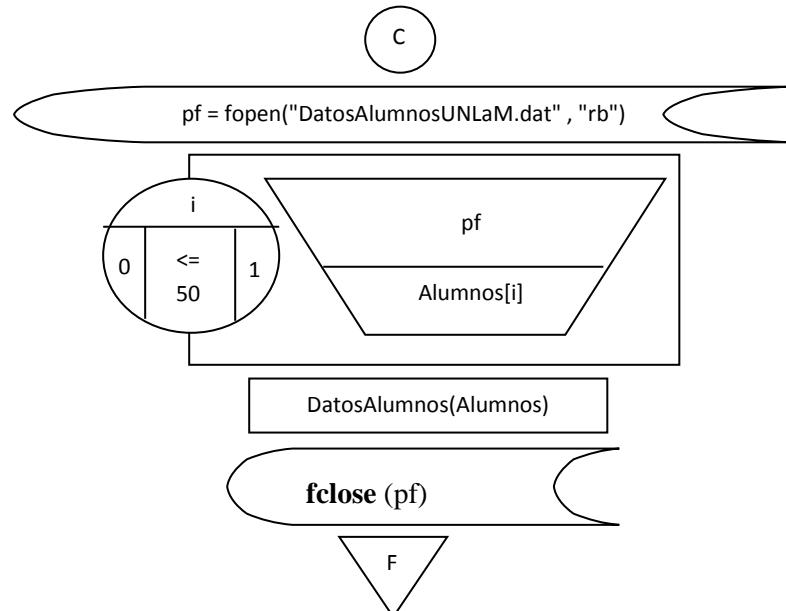
```

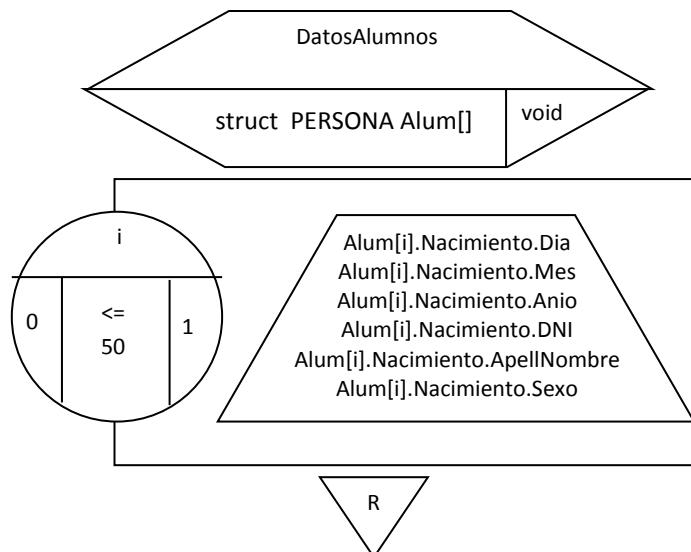
        exit (1);
    }
    for(i=0 ; i <= 50; i++)
    {
        fwrite(&Alumnos[i], sizeof(struct PERSONA), 1, pf);
        //Aquí la función que permite grabar un registro a la vez en un archivo
        binario.
    }
    fclose(pf);
    return 0;
}
void CargaDatosAlumnos (struct PERSONA Alum[]) //Declaración de la función
{
    int i;
    for(i=0 ;i <= 50; i++)
    {
        printf("Ingrese Numero de DNI");
        scanf("%d", & Alum[i].DNI);
        printf("Ingrese Apellido y nombre");
        fflush(stdin);
        gets(Alum[i].Apellido);
        printf("Ingrese Sexo");
        fflush(stdin);
        scanf("%c", & Alum[i].Sexo);
        printf("Ingrese Numero del Dia de Nacimiento");
        scanf("%d", & Alum[i].Nacimiento.Dia);
        printf("Ingrese Numero del Mes de Nacimiento");
        scanf("%d", & Alum[i].Nacimiento.Mes);
        printf("Ingrese Numero del Año de Nacimiento");
        scanf("%d", & Alum[i].Nacimiento.Anio);
    }
}

```

Ejemplo 6:

Es un programa que recupera (lee) los datos que se encuentran en un archivo llamado "DatosAlumnosUNLaM.dat", y luego guardarlos en un arreglo (vector) de estructura de datos, los cuales se muestran por pantalla por medio de una función llamada "DatosAlumnos". Para realizar este ejemplo se debe compilar siempre y cuando exista el archivo llamado "DatosAlumnosUNLaM.dat" y que contenga 50 registros.





```

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

struct FECHA
{
    int Dia;
    int Mes;
    int Anio;
};

struct PERSONA
{
    long int DNI;
    char ApellNombre[41];
    char Sexo;
    struct FECHA Nacimiento;
};

void DatosAlumnos (struct PERSONA []); //Prototipo de la función
int main()
{
    struct PERSONA Alumnos[50];
    int i;
    FILE *pf;
    pf = fopen("DatosAlumnosUNLaM.dat", "rb");
    if (pf==NULL)
    {
        printf("No se pudo abrir el archivo.");
        getch();
        exit (1);
    }
    for(i=0 ; i <= 50; i++)
    {
        fread(&Alumnos[i], sizeof(struct PERSONA), 1, pf);
        //Aquí la función que permite recuperar (leer) un registro a la vez en un
        //archivo binario.
    }
    DatosAlumnos ( Alumnos ); // Llamada de la función
    fclose(pf);
    return 0;
}
    
```

```
void DatosAlumnos (struct PERSONA Alum[])//Declaración de la función. Permite recibir como parámetro un arreglo (vector) de estructura de datos y luego mostrarlos por pantalla.  
{  
    int i;  
    for(i=0 ; i <= 50; i++)  
    {  
        printf("\nFecha de Nacimiento: %d-%d-%d Nro DNI: %d nombre: %s sexo %c",  
            Alum[i].Nacimiento.Dia, Alum[i].Nacimiento.Mes, Alum[i].Nacimiento.Año,  
            Alum[i].DNI, Alum[i].Apellido, Alum[i].Sexo);  
    }  
}
```

En el Anexo de esta unidad encontrará ejercicios resueltos con su correspondiente diagrama y código en lenguaje C abarcando las distintas alternativas que se pueden dar al tratar con archivos binarios.

9. Exportar datos para otros programas

Los archivos binarios son dependientes del tamaño del tipo de dato del lenguaje y de la arquitectura del procesador con el cual fueron creados. Por ejemplo, en los entornos Windows o Linux actuales una variable int ocupa 4 bytes mientras que en sistemas operativos anteriores como D.O.S. un entero ocupaba 2 bytes en memoria. Adicionalmente no se puede leer un archivo binario sino se conoce exactamente como es el formato del registro de datos con el cual fue generado. Es por ello que cuando se desea intercambiar datos entre distintos sistemas en general se utilizan archivos de texto plano ya que pueden ser interpretados por cualquier lenguaje y bajo cualquier arquitectura sin problemas.

El uso de archivos de texto para exportar datos se puede realizar de dos formas distintas:

- Utilizar algún formato estandarizado que permita grabar los datos de una forma estructurada para que puedan ser interpretados. Algunos ejemplos son los formatos XML o JSON este último creado para representar objetos en lenguajes de programación no estructurados.
- Grabar los registros en el archivo de texto con algún separador de campo, por ejemplo, grabando cada campo de la estructura separado por coma o por punto y coma y haciendo que cada registro se grabe en una línea distinta del archivo de texto.

De las dos estrategias utilizaremos la segunda por su sencillez y además permitirá rápidamente poder consultar los datos exportados mediante programas de hojas de cálculo como por ejemplo Excel.

El primer paso es crear un archivo de texto. Para ello se debe modificar el modo de apertura del archivo cambiando la letra b en los modos indicados previamente por la letra t que representa un archivo de texto. Para exportar los datos utilizaremos dos modos de apertura de archivos de texto:

- wt: cuando se quiera crear un archivo de texto únicamente para escritura tomando en consideración de que si ya existía los datos se van a sobrescribir
- at: cuando se quieran exportar datos en un archivo de texto, agregando registros si ya existe o creando un nuevo archivo si el mismo no existía.

Para escribir los datos en el archivo de texto utilizaremos una nueva función **fprintf**. Esta función es similar al printf para mostrar los datos por pantalla con formato solo que en lugar de por pantalla se le indica el archivo en el cual se guardarán los datos. El formato de la función fprintf es el siguiente:

```
fprintf(puntero FILE, texto con formato, lista de variables separadas por coma);
```

A continuación, se muestra un programa que permite ingresar datos de alumnos por teclado y generar un archivo de texto en formato csv (del inglés comma separated values, valores separados por coma) con los datos ingresados. Como separador de campo en lugar de coma se recomienda utilizar punto y coma ya que de esta forma programas como el Excel al abrirlo automáticamente reconocen los distintos campos y ya nos muestran los datos en distintas columnas.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
    char nombre[40];
    int dni;
    float promedio;
}snombres;
int main()
{
    snombres reg;
    FILE * fp;
    //se usa el modo de apertura wt que crea o reemplaza el archivo de texto
    fp=fopen ("alumnos.csv", "wt");
    if (fp==NULL)
    {
        printf("Error al abrir el archivo.");
        exit(1);
    }
    /*Graba una fila de encabezados para identificar los campos, el \n al final hace que se grabe una
    línea de texto completa y luego baje a la fila siguiente */
    fprintf(fp, "Nombre;DNI;Promedio\n");

    printf ("ingrese dni 0 para terminar:");
    scanf("%d", &reg.dni);
    while (reg.dni>0)
    {
        printf ("ingrese el nombre:");
        getchar();
        gets(reg.nombre);
        printf ("Ingrese el promedio:");
        scanf("%f", &reg.promedio);
        //El formato es idéntico al printf solo se agrega el ; como separador entre dato y dato
        fprintf(fp, "%s;%d;%f\n", reg.nombre, reg.dni, reg.promedio);

        printf ("ingrese dni 0 para terminar:");
        scanf("%d", &reg.dni);
    }
    fclose(fp);
    return 0;
}
```



Elementos de Programación

UNIDAD 11. CORTE DE CONTROL

INDICE

| | |
|---|----|
| 1. ¿QUÉ ES EL CORTE DE CONTROL?..... | 2 |
| 2. REGLAS GENERALES PARA APlicAR CORTE DE CONTROL | 2 |
| 3. CÓMO APlicAR CORTE DE CONTROL EN DIFERENTES CASOS..... | 3 |
| 3.1 UN NIVEL DE CORTE CON INGRESO DESDE TECLADO..... | 3 |
| 3.2 DOS NIVELES DE CORTE CON INGRESO DESDE TECLADO..... | 8 |
| 3.3 UN NIVEL DE CORTE DESDE UN ARCHIVO | 13 |
| 3.4 UN NIVEL DE CORTE DESDE UN ARCHIVO GENERANDO ARCHIVO RESUMEN | 15 |
| 3.5 UN NIVEL DE CORTE DESDE UN ARCHIVO GENERANDO ARCHIVOS DINÁMICAMENTE | 18 |

UNIDAD 11 - CORTE DE CONTROL

OBJETIVOS: Procesar lotes de datos ordenados a fin de obtener reportes o estadísticas.
Afianzar el manejo de estructuras repetitivas, contadores y acumuladores.

1. ¿Qué es el corte de control?

El corte de control es un proceso en el cual partiendo de registros ordenados por el valor de uno o más campos se los procesa para obtener información agrupada en lotes y sub-lotes al detectar un cambio en la/las variables por la cual la información se encuentra organizada.

En otras palabras, se procesa un conjunto ordenado de registros en subconjuntos determinados por el cambio de valor de una o más variables.

El corte de control se aplica para obtener información detallada agrupada por uno o más criterios.

Dicha información puede mostrarse en forma de reporte a medida que se detecta el cambio en la variable por la cual la información se encuentra ordenada o puede grabarse un archivo resumen con dicha información.

La condición necesaria para aplicar corte de control es que la información esté ordenada u organizada por el campo sobre el cual queremos agrupar la información, es decir que todos los registros de un mismo tipo deben estar juntos.

El corte de control puede aplicarse en varios niveles siempre y cuando la información venga ordenada por todos los campos sobre los cuales se desea agrupar. Por ejemplo, si se desea obtener información de los ingresos a un establecimiento agrupada por día y en cada día se desea obtener un detalle por hora, los datos deben estar primero ordenados por día y dentro de cada día ordenados por hora. En este caso podría aplicarse dos cortes uno por día y otro por hora a fin de obtener información agrupada en cada uno de ellos. Pero no siempre que la información esté ordenada significa que se deba aplicar corte de control. En el caso anterior si solo se desea obtener información detallada por día, entonces se hará un corte de un solo nivel ya que agrupar también por hora no aportaría información adicional.

Para aplicar corte de control si la información se ingresa por teclado debe ingresarse en forma ordenada, es por ello por lo que lo más común es procesar información ya ordenada que venga en un vector o en un archivo.

2. Reglas Generales para aplicar corte de control

El corte de control se realiza anidando distintas estructuras repetitivas con las siguientes reglas:

- Se realiza un ciclo repetitivo con la condición de fin de datos
- Dentro del ciclo de fin de datos se anida un ciclo por cada nivel de corte que se desea realizar (el ciclo de cada nivel va anidado dentro del ciclo del nivel anterior)
- Las condiciones de los ciclos se van propagando, es decir que el ciclo interno de corte va a incluir la condición de fin del ciclo que lo contiene y su propia condición. Si hay otros niveles de corte va a incluir las condiciones anteriores y su propia condición de fin.
- La información se lee antes de ingresar al ciclo de la condición de fin y se vuelve a leer antes de finalizar el ciclo más interno.

- Las variables que cuentan o acumulan información agrupada deben inicializar antes de comenzar el ciclo del corte y mostrarse al salir de dicho ciclo haciendo referencia a que dicha información corresponde al dato leído anteriormente y no al último, ya que el ciclo termina por que el lote finalizó y por lo tanto la lectura actual va a ser diferente.

3. Cómo aplicar Corte de Control en diferentes casos.

3.1 Un nivel de corte con ingreso desde teclado

Supongamos que en el Departamento de Alumnos de la UNLaM se quiere saber cuántos alumnos se inscribieron en cada Departamento como así también la cantidad total de alumnos inscriptos en la Universidad, considerando que los datos están ordenados por el campo clave “Número de Departamento Inscripto”, para lo cual se han analizado los siguientes datos:

| Estructura PERSONA | | | | |
|--------------------|-------------------|----------------------|---------------|-------------------|
| DNI | Apellido y Nombre | Nro. Dpto. Inscripto | Nro. Comisión | Código de Materia |

Ejemplo de la Lista de Datos a analizar:

| nDNI | ApyN | nDpto | nComi | cMat | |
|----------|--------------------|-------|-------|------|----|
| 46051476 | Martínez, Lucia | 1 | 2 | 1046 | |
| 45765812 | Franco, Luciano | 1 | 2 | 1230 | |
| 46256892 | Ramírez, María | 1 | 3 | 1501 | |
| 6785933 | Luna, Ignacio | 1 | 3 | 1530 | |
| 45368947 | López, Roxana | 1 | 5 | 1501 | |
| 46789256 | Mangano, Luis | 1 | 6 | 1501 | 6 |
| 44589632 | Lu, Lian | 2 | 1 | 1680 | |
| 45635484 | Marengo, Nicolás | 2 | 1 | 1695 | |
| 46489212 | Pellejero, Diana | 2 | 2 | 1680 | |
| 45632584 | Herrera, Mario | 2 | 4 | 1623 | |
| 46869572 | Amaya, Luciara | 2 | 4 | 1625 | |
| 46759154 | Pedrazza, Antonio | 2 | 6 | 1680 | 6 |
| 45389456 | Ramírez, Mario | 4 | 1 | 1742 | |
| 46786428 | Murua, Analía | 4 | 1 | 1748 | |
| 46348922 | Moreno, Claudio | 4 | 1 | 1748 | |
| 46555789 | Velloso, Andrea | 4 | 5 | 1723 | 4 |
| 46893768 | Luciani, Marilu | 5 | 5 | 1944 | |
| 45785354 | Perrota, Mario | 5 | 5 | 1945 | |
| 46223566 | Landaburo, Lucila | 5 | 6 | 1923 | |
| 45883698 | Randazzo, Emiliano | 5 | 6 | 1924 | 4 |
| 99999999 | | | | | 20 |

De acuerdo al requerimiento formulado el resultado que se obtendrá del anterior juego de datos será:

| Departamento | Cantidad de Alumnos |
|--------------|---------------------|
| 1 | 6 |
| 2 | 6 |
| 4 | 4 |
| 5 | 4 |

Total de alumnos en la Universidad: 20

Estrategia para resolver el problema:

Se desea mostrar información agrupada por departamento por lo tanto los datos se deben ingresar ordenados por número de departamento. Por cada ingreso se debe recordar cuál fue el número de departamento que se ingresó anteriormente y compararlo con el recién ingresado. Si el nuevo dato es distinto del anterior entonces se detecta que se finalizó de ingresar la información del lote anterior. Por cada lote se debe contar cuantos inscriptos hay entonces se debe utilizar un contador que debe inicializarse cada vez que se comienza un nuevo lote de información y mostrarse al detectar que el mismo finaliza.

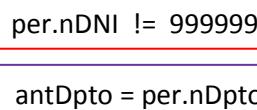
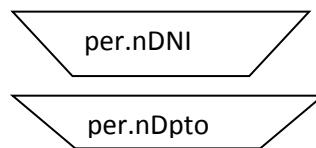
Diagrama de Lógica general para el análisis:

Variables de la función main ()

```
struct PERSONA per;
int antDpto;
```

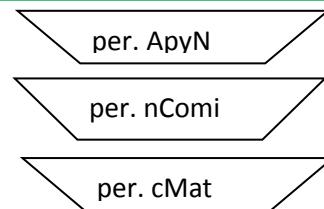
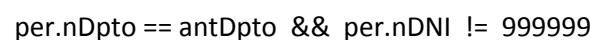


“Aquí se inicializan los contadores/acumuladores/señales generales”



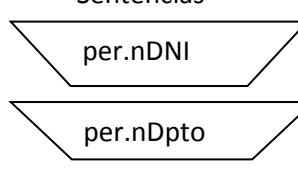
“Aquí se inicializan los contadores/acumuladores/señales del corte x nDpto”

“Sentencias”



“Aquí se incrementan o cambian de valor los contadores/acumuladores/señales de corte x nDpto”

“Sentencias”



“Sentencias de avance de lectura de datos”

“Aquí se muestran valor los contadores/acumuladores/señales de corte x nDpto”

“Sentencias”

“Aquí se muestran valor los generales”

“Sentencias”



Análisis de Diagrama de Lógica:

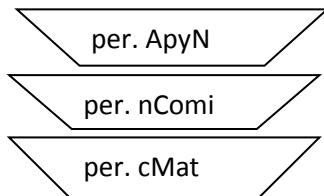
- Hay 2 **while** el primero o sea el externo siempre será la “Condición de Fin” y el segundo o sea el interno será el “Corte de Control”.
- Las variables que deben estar siempre antes del primer/externo **while** son las que están dentro de las “condiciones de cada uno de los **while**”.
- Siempre antes del **while** que será el “Corte de Control” deberá estar la asignación a una variable auxiliar que le permitirá guardar el contenido del “campo clave”, y de esta manera controlar cuando se realiza al cambio del contenido de la variable “campo clave”. En este ejemplo será:

```
antDpto = per.nDpto
```

- En cada **while** se arrastra la condición del **while** anterior y se le agrega con un operador lógico **AND** la condición del nuevo Corte de Control. En este ejemplo será:

```
per.nDpto == antDpto && per.nDNI != 999999
```

- Dentro del **while** mas interno deben estar el resto de las sentencias que corresponden a todas las variables que forman parte de los datos, en este ejemplo serán:



- El incremento de la lectura se realiza dentro del **while** más interno. Allí es donde se avanza con la lectura de datos, que en este ejemplo serán:

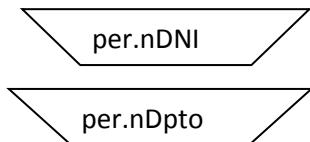
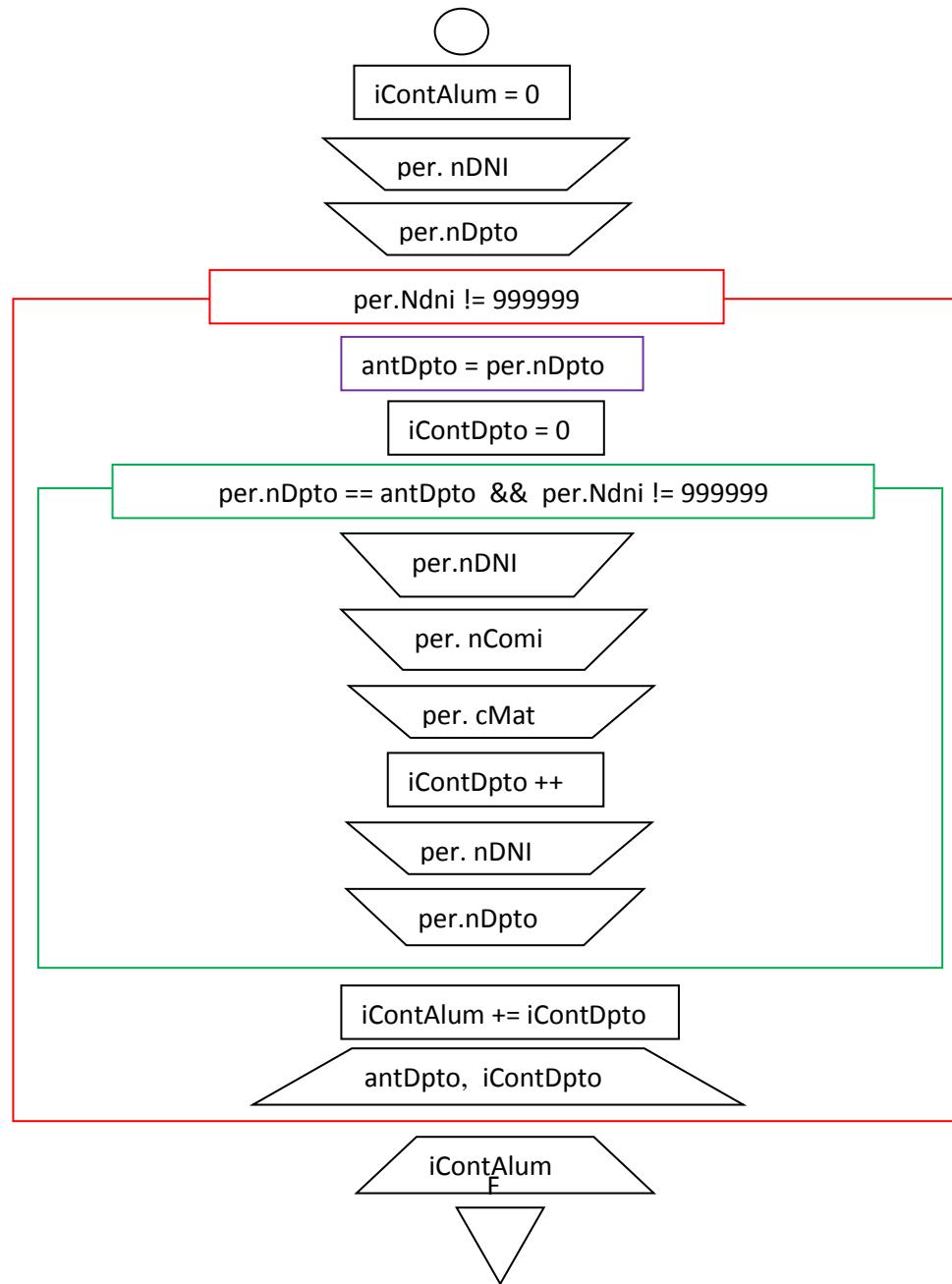


Diagrama completo:

Codificación en Lenguaje C:

```

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

struct PERSONA
{
    int nDNI;
    char ApyN [26];
    int nDpto;
    int nComi;
    int cMat;
};

int main ( )
{
    struct PERSONA per;
    
```

```
int antDpto, iContAlum, iContDpto;  
  
iContAlum = 0;  
  
printf("\nIngrese Numero de DNI (999999 para terminar): ");  
scanf("%d", &per.nDNI);  
  
printf("\nIngrese Numero de Departamento: ");  
scanf("%d", &per.nDpto);  
  
while (per.nDNI != 999999)  
{  
    antDpto = per.nDpto;  
    iContDpto = 0;  
  
    while (per.nDpto == antDpto && per.nDNI != 999999)  
    {  
        getchar(); //limpia el buffer  
        printf("\nIngrese Apellido y Nombre: ");  
        gets(per.ApyN);  
  
        printf("\nIngrese Numero de Comision: ");  
        scanf("%d", &per.nComi);  
  
        printf("\nIngrese Codigo de Materia: ");  
        scanf("%d", &per.cMat);  
  
        iContDpto++;  
  
        printf("\nIngrese Numero de DNI (999999 para terminar): ");  
        scanf("%d", &per.nDNI);  
  
        printf("\nIngrese Numero de Departamento: ");  
        scanf("%d", &per.nDpto);  
    }  
  
    iContAlum += iContDpto;  
    printf("\nLa Cantidad de Alumnos de Nro Departamento %d es %d", antDpto,  
iContDpto);  
}  
  
printf("\nLa Cantidad de Alumnos que tiene la Universidad %d", iContAlum);  
return 0;  
}
```

3.2 Dos Niveles de corte con ingreso desde teclado

Supongamos que en el Departamento de Alumnos de la UNLaM se quiere saber cuántos alumnos se inscribieron en cada Comisión y en cada Departamento como así también la cantidad total de alumnos inscriptos en la Universidad, considerando que los datos están ordenados por el campo clave “Número de Departamento Inscripto” y además por “Número de Comisión”, para lo cual se han analizado los siguientes datos:

Ejemplo de la Estructura de Datos:

| Estructura PERSONA | | | | |
|--------------------|-------------------|--------------------|---------------|-------------------|
| DNI | Apellido y Nombre | Nro Dpto Inscripto | Nro. Comisión | Código de Materia |
| | | | | |

Ejemplo de la Lista de Datos a analizar:

| nDNI | ApyN | nDpto | nComi | cMat | | |
|----------|--------------------|-------|-------|------|---|----|
| 46051476 | Martínez, Lucia | 1 | 2 | 1046 | | |
| 45765812 | Franco, Luciano | 1 | 2 | 1230 | 2 | |
| 46256892 | Ramírez, María | 1 | 3 | 1501 | 2 | |
| 46785933 | Luna, Ignacio | 1 | 3 | 1530 | | |
| 45368947 | López, Roxana | 1 | 5 | 1501 | | |
| 46789256 | Mangano, Luis | 1 | 6 | 1501 | | |
| 44589632 | Lu, Lian | 2 | 1 | 1680 | | |
| 45635484 | Marengo, Nicolás | 2 | 1 | 1695 | | |
| 46489212 | Pellejero, Diana | 2 | 2 | 1680 | | |
| 45632584 | Herrera, Mario | 2 | 4 | 1623 | | |
| 46869572 | Amaya, Luciara | 2 | 4 | 1625 | | |
| 46759154 | Pedrazza, Antonio | 2 | 6 | 1680 | 6 | |
| 45389456 | Ramírez, Mario | 4 | 1 | 1742 | | |
| 46786428 | Murua, Analía | 4 | 1 | 1748 | | |
| 46348922 | Moreno, Claudio | 4 | 1 | 1748 | 4 | |
| 46555789 | Velloso, Andrea | 4 | 5 | 1723 | | |
| 46893768 | Luciani, Marilu | 5 | 5 | 1944 | | |
| 45785354 | Perrota, Mario | 5 | 5 | 1945 | | |
| 46223566 | Landaburo, Lucila | 5 | 6 | 1923 | | |
| 45883698 | Randazzo, Emiliano | 5 | 6 | 1924 | 4 | |
| 99999999 | | | | | | 20 |

De acuerdo al requerimiento formulado el resultado que se obtendrá del anterior juego de datos será:

| Departamento | Comisión | Cantidad de Alumnos |
|--------------|----------|---------------------|
| 1 | 2 | 2 |
| 1 | 3 | 2 |
| 1 | 5 | 1 |
| 1 | 6 | 1 |
| 1 | | 6 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |
| 2 | 4 | 2 |
| 2 | 6 | 1 |
| 2 | | 6 |

| | | |
|---|---|---|
| 4 | 1 | 3 |
| 4 | 5 | 1 |
| 4 | | 4 |
| 5 | 5 | 2 |
| 5 | 6 | 2 |
| 5 | | 4 |

Total de alumnos en la Universidad: 20

Estrategia para resolver el problema:

Ahora lo que se pide es la información agrupada por departamento y dentro de cada departamento agrupados por comisión, por lo tanto, serán necesarios dos niveles de corte. Los datos se deben ingresar ordenados por departamento y comisión para poder ser procesados correctamente. Vamos a necesitar 3 ciclos, uno con la condición de fin, uno para el corte por departamento y otro para el corte por comisión.

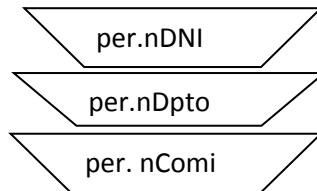
Diagrama de Lógica para el análisis:

Variables de la función main():

```
struct PERSONA per;
int antDpto, antComi;
```



“Aquí se inicializan los contadores/acumuladores/señales generales”



`per.nDNI != 999999`

`antDpto = per.nDpto`

“Aquí se inicializan los contadores/acumuladores/señales del corte x nDpto”

`per.nDpto == antDpto && per.nDNI != 999999`

`antComi = per.nComi`

Aquí se inicializan los contadores/acumuladores/señales del corte x nComi”

`per.nComi == antComi && per.nDpto == antDpto && per.nDNI != 999999`

`per. ApyN`

`per. cMat`

“Aquí se incrementan o cambian de valor los contadores/acumuladores/señales de corte x nComi”

“Sentencias”

`per.nDNI`

`per.nDpto`

`per. nComi`

“Sentencias de avance
de lectura de datos”

“Aquí se incrementan o cambian de valor los contadores/acumuladores/señales de corte x nDpto”

“Aquí se muestran valor los contadores/acumuladores/señales de corte x nComi”

“Sentencias”

“Aquí se muestran valor los contadores/acumuladores/señales de corte x nDpto”

“Sentencias”

“Aquí se muestran valor los generales”

“Sentencias”



Análisis de Diagrama de Lógica:

- Hay 3 **while** el primero o sea el externo siempre será la “Condición de Fin”, el segundo y el tercero serán los “Cortes de Control”.
- Las variables que deben estar siempre antes del primer/externo **while** son las que están dentro de las “condiciones de cada uno de los **while**”.
- **Siempre** antes de cada **while** que será el “Corte de Control” deberá estar la asignación a una variable auxiliar que le permitirá guardar el contenido del “campo clave”, y de esta manera controlar cuando se realiza al cambio del contenido de la variable “campo clave”.

antDpto = per.nDpto

antComi = per.nComi

En este ejemplo será:

- En cada **while** se arrastra la condición del **while** anterior y se le agrega con un operador lógico **AND** la condición del nuevo Corte de Control. En este ejemplo será:

per.nDpto == antDpto && per.nDNI != 999999

per.nComi == antComi && per.nDpto == antDpto && per.nDNI != 999999

- Dentro del **while** mas interno deben estar como las primeras sentencias el resto de todas las variables que forman parte de los datos, en este ejemplo serán:

per. ApyN

per. cMat

- El incremento de la lectura se realiza dentro del **while** más interno. Allí es donde se avanza con la lectura de datos, que en este ejemplo serán:

per.nDNI

per.nDpto

per. nComi

3.3 Un Nivel de corte desde un archivo

En la Empresa “M. y M.” se quiere saber cuánto se abona de sueldo por cada Sector de Fábrica y en toda la Empresa. Los datos están ordenados por el campo clave “Número de Sector de Fábrica”, para lo cual se han analizado los datos que se encuentra en el archivo llamado “empleados.dat” y los registros del archivo tiene el siguiente formato:

- a. Número de Legajo (entero)
- b. Apellido y Nombre (hasta un máximo de 25 letras)
- c. Año de Ingreso a la empresa (entero)
- d. Número de Sector de Fábrica (entero)
- e. Sueldo (real)

Resolución del problema:

La propuesta es leer y no bajar a memoria los datos que tiene el archivo llamado “empleados.dat” porque se desconoce la cantidad de Registros que posee el archivo, y para lo cual se necesita crear una Estructura que contenga el formato del Registro del archivo, con lo cual se propone crear una estructura que se llamará PERSONA, que será la siguiente:

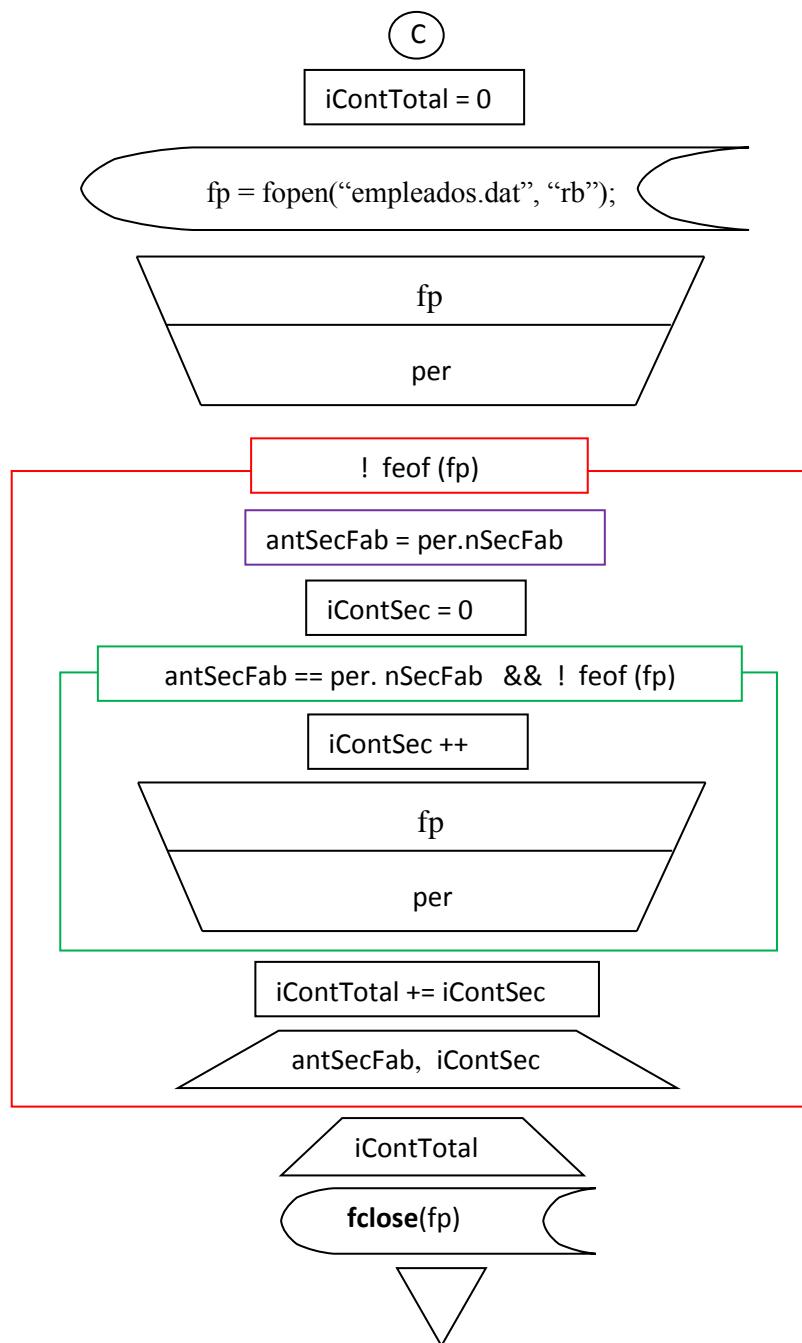
| Estructura PERSONA | | | | |
|--------------------|-------------------|-------------|--------------------|--------|
| Legajo | Apellido y Nombre | Año Ingreso | Nro.Sector Fábrica | Sueldo |

Diagrama de Lógica:

| struct PERSONA | | | | |
|----------------|----------------|-----------|-------------|---------------|
| int nLegajo | char ApyN [26] | int nAnio | int nSecFab | float rSueldo |

Variables de la función main():

```
struct PERSONA per;
FILE fp
int antSecFab,iContTotal, iContSec;
```



Codificación en Lenguaje C:

```

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

struct PERSONA
{
    int nLegajo;
    char ApyN [26];
    int nAnio;
    int nSecFab;
  
```

```
    float rSueldo;
};

int main ( )
{
    struct PERSONA per;
    int antSecFab, iContTotal, iContSec;
    FILE *fp;
    iContTotal = 0;
    if ( (fp = fopen("empleado.dat", "rb")) == NULL )
    {
        printf("\nNo se puede abrir.");
        getch();
        exit(1);
    }

    fread(&per, sizeof(struct PERSONA ),1,fp);
    while ( ! feof(fp) )
    {
        antSecFab = per.nSecFab;
        iContSec = 0;

        while (per.nSecFab == antSecFab && ! feof(fp) )
        {

            iContSec++;
            fread(&per, sizeof(struct PERSONA ),1,fp);

        }
        iContTotal += iContSec;
        printf("\nLa Cantidad de Empleados del Sector %d es %d",
               antSecFab, iContSec);
    }
    fclose(fp);

    printf("\nLa Cantidad de Empleados que tiene la empresa es %d",
           iContTotal);

    return 0;
}
```

3.4 Un Nivel de corte desde un archivo generando archivo resumen

Se dispone de un archivo llamado “ventas.dat” que tiene las ventas realizadas por una empresa a lo largo del mes. La empresa tiene distintas sucursales a lo largo del país. El archivo contiene registros con los siguientes datos:

- Código de sucursal (entero)
- Código de producto (entero)
- Cantidad Vendida (entero)
- Precio Unitario (real)

Para una misma sucursal se recibe un solo registro por cada producto vendido.

Se desea realizar un programa que calcule las ventas realizadas en cada sucursal generando un archivo llamado ventasXsuc.dat que contenga un registro sumarizado por cada sucursal con los siguientes datos:

- Código de sucursal (entero)

- Cantidad total de productos vendidos
- Importe total recaudado
- Código de producto con mayor cantidad de unidades vendidas (considerar único)

Resolución del problema:

Para resolver este problema será necesario trabajar con los dos archivos abiertos, el archivo ventas.dat se abrirá en modo lectura para ir leyendo registro a registro y hacer el corte de control sobre el campo código de sucursal. Al mismo tiempo se debe crear el archivo ventasXsuc.dat para que cada vez que se detecte un cambio en el código de sucursal se grabé un archivo sumarizado en dicho archivo.

Además de dos campos sumarizados (cantidad e importe), se nos pide determinar el código de producto con mayor cantidad de unidades vendidas por lo que se deberá aplicar el algoritmo para calcular el máximo para los productos vendidos en cada sucursal.

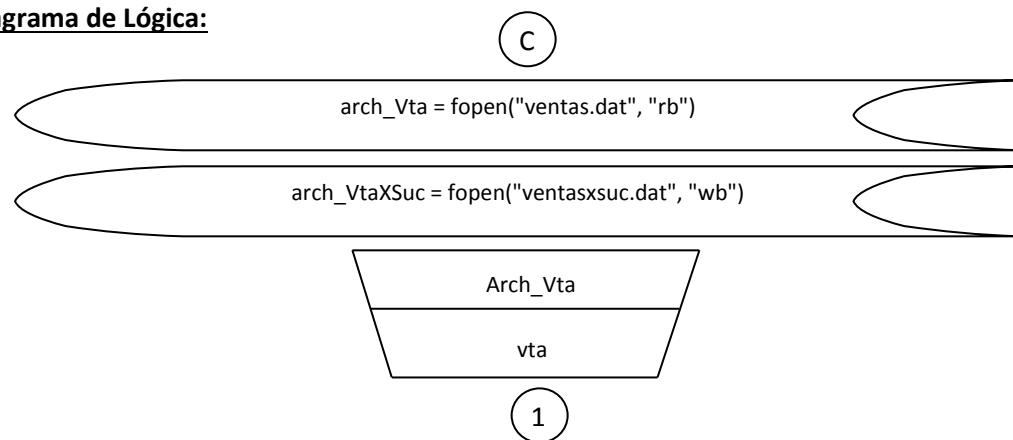
Se deben declarar dos estructuras de datos una por cada archivo ya que almacenan registros distintos.

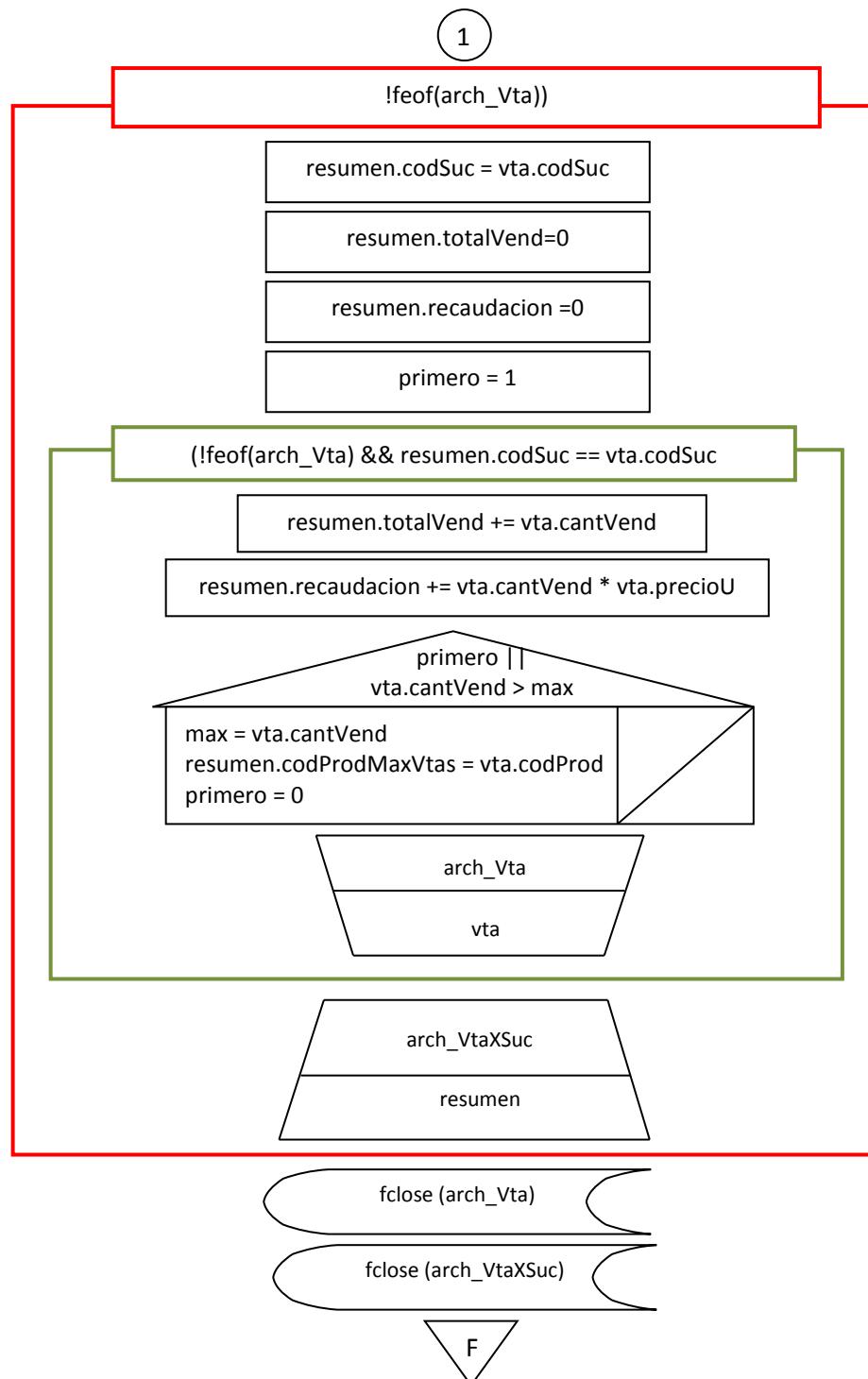
| Estructura sVenta | | | |
|-------------------|---------|----------|---------|
| codSuc | codProd | cantVend | precioU |

| Estructura sResumen | | | |
|---------------------|-----------|-------------|----------------|
| codSuc | totalVend | recaudacion | codProdMaxVtas |

Nótese que este programa no tiene salida por pantalla los resultados se guardan directamente en un archivo.

Diagrama de Lógica:





Codificación en Lenguaje C:

```

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

struct sVenta
{
    int codSuc;
    int codProd;
    int cantVend;
    float precioU;
};

```

```
struct sResumen
{
    int codSuc;
    int totalVend;
    float recaudacion;
    int codProdMaxVtas;
};

int main()
{
    FILE * arch_Vta, * arch_VtaXSuc;
    struct sVenta vta;
    struct sResumen resumen;
    int primero, max;

    arch_Vta = fopen("ventas.dat", "rb");
    arch_VtaXSuc = fopen("ventasxsuc.dat", "wb");
    if (arch_Vta==NULL || arch_VtaXSuc==NULL)
    {
        printf("Error al abrir los archivos");
        getch();
        exit(1);
    }

    fread(&vta, sizeof(vta), 1, arch_Vta);
    while (!feof(arch_Vta))
    {
        /*guarda la sucursal anterior e inicializa los acumuladores
        y contadores directamente en la variable del tipo estructura
        que se va a grabar el archivo por sucursal*/

        resumen.codSuc = vta.codSuc;
        resumen.totalVend=0;
        resumen.recaudacion =0;
        primero = 1; //para el maximo

        while (!feof(arch_Vta) && resumen.codSuc == vta.codSuc)
        {
            resumen.totalVend += vta.cantVend;
            resumen.recaudacion += vta.cantVend * vta.precioU;

            if(primerico || vta.cantVend > max)
            {
                max = vta.cantVend;
                resumen.codProdMaxVtas = vta.codProd;
                primero = 0;
            }
            fread(&vta, sizeof(vta), 1, arch_Vta);
        }
        fwrite (&resumen, sizeof( resumen),1, arch_VtaXSuc);
    }
    fclose(arch_Vta);
    fclose(arch_VtaXSuc);
}
```

3.5 Un Nivel de corte desde un archivo generando archivos dinámicamente

El sistema de control de ingreso y salida de empleados de una fábrica deja al final del mes un archivo ordenado por sector y summarizado por empleado llamado horasTrabajadas.dat con los siguientes datos:

- Sector (texto de 10 caracteres máximo)
- DNI del empleado (entero)
- Horas trabajadas

Por otro lado, se dispone del archivo empleados.dat con los empleados de la fábrica que contiene los siguientes datos:

- DNI del empleado (entero)
- Nombre y Apellido (texto de 30 caracteres máximo)
- Valor que cobra por hora (float)

No se sabe la cantidad exacta de empleados, pero sí se sabe que no son más de 100.

Se desea realizar un programa que utilizando ambos archivos genere para cada sector un archivo que contenga el detalle de sueldos que deba pagar a sus empleados. Se debe generar para cada sector un archivo distinto cuyo nombre sea nombreSector.dat y debe contener registros con la siguiente información:

- DNI del empleado (entero)
- Nombre y Apellido (texto de 30 caracteres máximo)
- Horas trabajadas (entero)
- Sueldo a Pagar (float)

Resolución del problema:

Inicialmente se deben descargar en memoria los datos los empleados de la empresa, que como se sabe que no son más de 100 se pueden descargar en un vector.

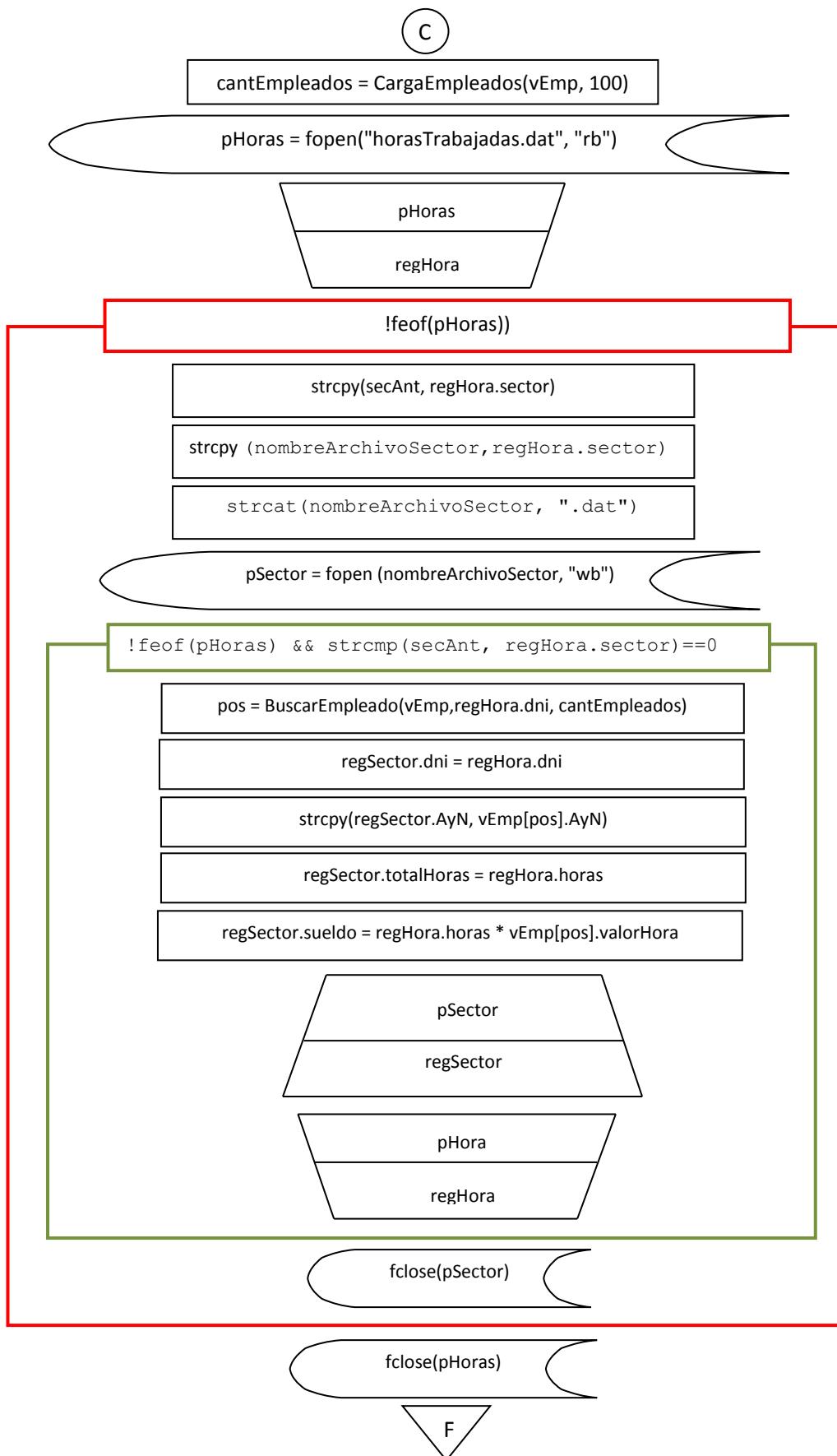
Luego se procesa el archivo horasTrabajadas.dat que está ordenado por sector. Por cada sector se debe generar dinámicamente un nuevo archivo con el nombre dicho sector y calcular en él la información solicitada para ello se debe buscar el empleado en el vector en memoria para obtener su nombre y valor que cobrar por hora.

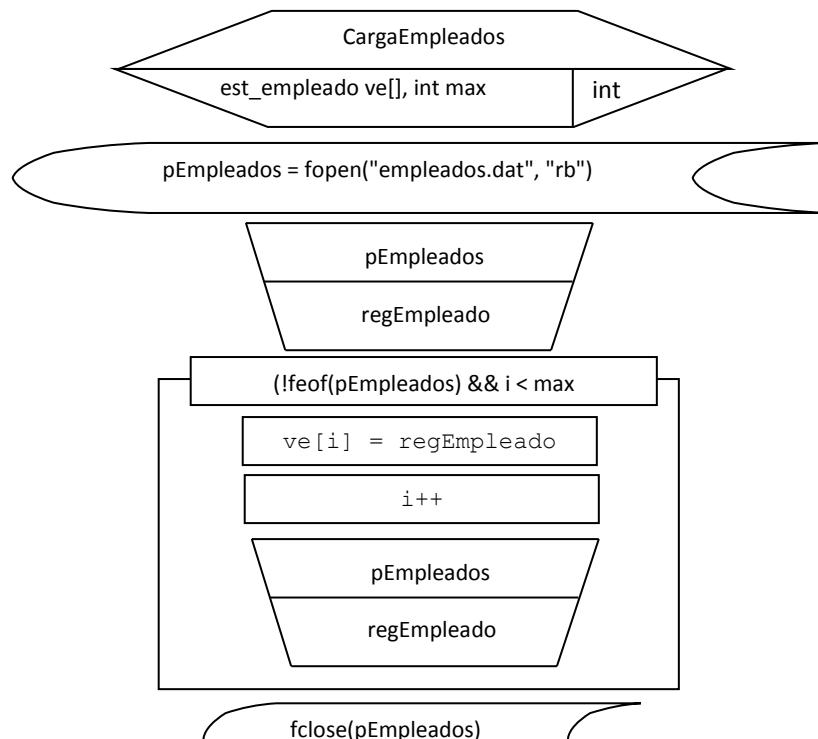
Se necesitan tres estructuras una por cada archivo (en realidad para los sectores se crean varios archivos, pero todos con la misma estructura):

| Estructura est_horas | | |
|----------------------|-----|-------|
| sector | dni | horas |
| | | |

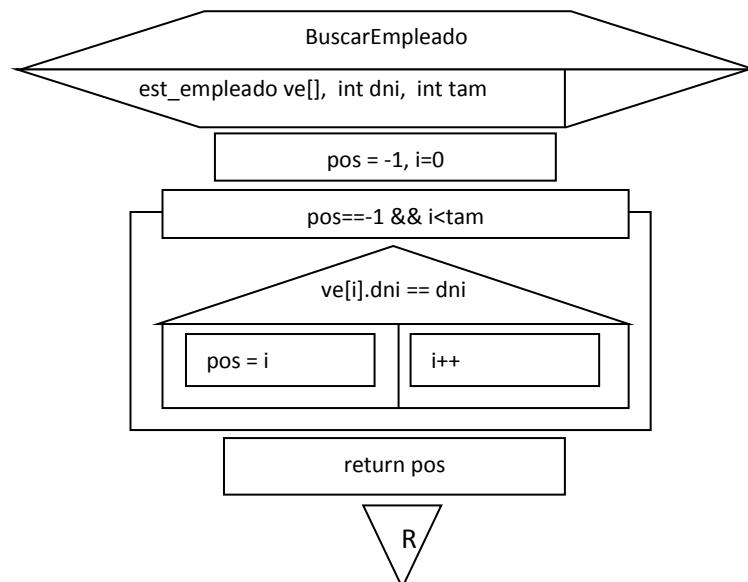
| Estructura est_empleado | | |
|-------------------------|-----|-----------|
| dni | AyN | valorHora |
| | | |

| Estructura est_sector | | | |
|-----------------------|-----|------------|--------|
| dni | AyN | totalHoras | sueldo |
| | | | |

Diagrama de Lógica:




`return i`



Codificación en Lenguaje C:

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

typedef struct
{
    char sector[11];
    int dni;
    int horas;
} est_horas;

typedef struct
{
    int dni;
    char AyN[31];
    float valorHora;
} est_empleado;

typedef struct
{
    int dni;
    char AyN[31];
    int totalHoras;
    float sueldo;
} est_sector;

int CargaEmpleados (est_empleado[],int);
int BuscarEmpleado (est_empleado[],int,int);

int main()
{
    FILE * pHoras, * pSector;
    int cantEmpleados, pos;
    est_empleado vEmp[100];
    est_horas regHora;
    est_sector regSector;
    char secAnt[11], nombreArchivoSector[15];

    cantEmpleados = CargaEmpleados (vEmp, 100);

    pHoras = fopen("horasTrabajadas.dat", "rb");
    if (pHoras == NULL)
    {
        printf ("No se pudo abrir el archivo de horas trabajadas");
        getch();
        exit (1);
    }

    fread(&regHora, sizeof(regHora), 1, pHoras);
    while (!feof(pHoras))
    {
        strcpy(secAnt, regHora.sector);
        /*Se genera el nombre del archivo basado en el nombre del sector */
        strcpy (nombreArchivoSector, regHora.sector);
        strcat(nombreArchivoSector, ".dat");
        pSector = fopen (nombreArchivoSector, "wb");
        if (pSector ==NULL)
        {
            printf ("No se pudo crear el archivo para el sector %s", secAnt);
            getch();
            exit(1);
        }

        while (!feof(pHoras) && strcmp(secAnt, regHora.sector)==0)
        {
            /*se busca el empleado y como se trabaja con archivos ya creados
```

```
        se asume que siempre lo encuentra*/
pos = BuscarEmpleado(vEmp, regHora.dni, cantEmpleados);
regSector.dni = regHora.dni;
strcpy(regSector.AyN, vEmp[pos].AyN);
regSector.totalHoras = regHora.horas;
regSector.sueldo = regHora.horas * vEmp[pos].valorHora;
fwrite(&regSector, sizeof(regSector), 1, pSector);

fread(&regHora, sizeof(regHora), 1, pHoras);
}

fclose(pSector);
fclose(pHoras);
}

int CargaEmpleados (est_empleado ve[], int max)
{
FILE *pEmpleados;
int i=0;
est_empleado regEmpleado;
pEmpleados = fopen("empleados.dat", "rb");
if (pEmpleados == NULL)
{
    printf ("No se pudo abrir el archivo de empleados.");
    getch();
    exit (1);
}

fread(&regEmpleado, sizeof(regEmpleado), 1, pEmpleados);
while (!feof(pEmpleados) && i < max)
{
    ve[i] = regEmpleado;
    i++;
    fread(&regEmpleado, sizeof(regEmpleado), 1, pEmpleados);
}
fclose(pEmpleados);
return i;
}

int BuscarEmpleado (est_empleado ve[], int dni, int tam)
{
int i =0 , pos =-1;
while (pos== -1 && i<tam)
{
    if (ve[i].dni == dni)
        pos = i;
    else
        i++;
}
return pos;
}
```