Universidad Autónoma del Estado de México
Centro Universitario UAEM Texcoco

INGENIERÍA EN COMPUTACIÓN

APUNTES.

LENGUAJE DE PROGRAMACIÓN ESTRUCTURADA.

2018B

Prof. Joel Ayala de la Vega.

Centro Universitario UAEM Texcoco
Av. Jardín Zumpango s/n. Fracc. El Tejocote
C.P. 56259 Texcoco, Estado de México.
Tels. (595) 9211216 - 9211247 - 9210368 - 9210493
e-mail: cutex.uaem@gmail.com.

CUTex

Contenido

INTRODUCCIÓN 5

PRESENTACIÓN 6

UNIDAD DE COMPETENCIA I. CONOCER LA IMPORTANCIA DE LOS LENGUAJES DE PROGRAMACIÓN ESTRUCTURADOS ASÍ COMO SU UTILIDAD 8

PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN (LP). 8

Intruducción. 8

Paradigma en lenguajes de programación. 9

Tipos de paradigmas. 9

ELEMENTOS DE LOS LENGUAJES DE PROGRAMACIÓN. 15

LAS OPERACIONES 15

INSTRUCCIONES 16

ORIGENES DE LOS LENGUAJES DE PROGRAMACIÓN ESTRUCTURADOS 19

La programación si GO TO. 19

La visión moderna de la programación estructurada: la segmentación. 21

PRINCIPALES HERRAMIENTAS DE SOFTWARE DESARROLLADOS EN LPE. 23

UNIDAD DE COMPETENCIA II. MANEJO DEL ENTORNO INTEGRADO DE DESARROLLO 1

Elementos de un ambiente integrado de desarrollo. 1

UNIDAD DE COMPETENCIA III. VARIABLES Y TIPOS DE DATOS. 10

VARIABLES EN EL LENGUAJE C 10

ARRAYS (ARREGLOS) UNIDIMENSIONALES 14

DECLARACIÓN DE TIPOS CON TYPEDEF EN C 17

CADENAS DE TEXTO COMO ARRAYS DE CARACTERES EN C 19

ARRAYS (ARREGLOS) MULTIDIMENSIONALES 20

MANEJO DE ARREGLOS CON MEMORIA DINÁMICA 22

MANEJO DE MATRICES EN FORMA DINÁMICA 29

MANEJO DE ARREGLOS DE REGISTROS EN FORMA DINÁMICA. 31

ENUMERACIONES 33

VARIBLES LOCALES Y EXTERNAS 34

extern 34

UNIDAD DE COMPETENCIAS IV. CODIFICACIÓN DE ESTRUCTURAS DE CONTROL. 37

ESTRUCTURAS DE DECISION IF THEN ELSE Y SWITCH CASE 37

ESTRUCTURAS DE DECISIÓN O SELECCIÓN 37

ANIDAMIENTO DE ESTRUCTURAS DE DECISIÓN 39

ESTRUCTURA SWITCH-CASE (SEGÚN-CASO) 40

UNIDAD DE COMPETENCIAS V. BIBLIOTECAS DE FUNCIONES Y USO DE PRINCIPALES FUNCIONES DEL LENGUAJE DE PROGRAMACIÓN 44
BIBLIOTECAS EN LENGUAJE C 44
UNIDAD DE COMPETENCIAS VI. USO DE MODULARIZACIÓN EN LA IMPLEMENTACIÓN DE PROGRAMAS. 57
Resumen. 57
MODULARIDAD 57
ACOPLAMIENTO 58
COHESIÓN 67
PASO DE PARÁMETROS 76
PASO DE PARÁMETROS EN VECTORES. 79
PASO DE PARÁMETROS EN MATRICES ESTÁTICAS. 81
PASO DE PARÁMETROS CON MATRICES DINÁMICAS. 84
PASO DE PARÁMETROS EN REGISTROS. 85
APUNTADORES A FUNCIONES. 89
Ejercicios: 95
UNIDAD DE COMPETENCIAS VII. ARCHIVOS Y FLUJOS 97
INTRODUCCIÓN 97
CLASIFICACIÓN DE LOS ARCHIVOS 98
Por la forma de acceso 99
APERTURA Y CIERRE DE ARCHIVOS 101
PROCESAMIENTO DE ARCHIVOS DE TEXTO 102
LEER Y ESCRIBIR CARACTERES 102
COMPROBAR FINAL DE ARCHIVO 103
LEER Y ESCRIBIR STRINGS 104
VOLVER AL PRINCIPIO DE UN ARCHIVO 107
OPERACIONES PARA USO CON ARCHIVOS BINARIOS 107
USO DE ARCHIVOS DE ACCESO DIRECTO 110
INTRODUCCIÓN. 110
PROCESAMIENTO DE ARCHIVOS. 110
ORGANIZACIÓN DE ARCHIVOS. 110
MÉTODO DE ACCESO SECUENCIAL INDIZADO (ISAM) 112
Windows 113
Linux 113
FUNCIONES PRINCIPALES DE C. 113
EJEMPLOS DE ARCHIVOS DE USO FRECUENTE 116

ARCHIVOS INDEXADOS 122
OTROS FORMATOS BINARIOS DE ARCHIVOS 123
Cabecera de los archivos GIF 123
UNIDAD DE COMPETENCIAS VIII. ESTRUCTURAS DE DATOS DINÁMICAS 125
INTRODUCCIÓN. 125
8.1 Programación de listas enlazadas 125
8.2 Programación de una pila 125
8.3 Traducción de una expresión infija a postfija. 130
8.4 PROGRAMACIÓN DE UNA LISTA SIMPLEMENTE ENLAZADA ORDENADA. 134
8.5 ARREGLO DE APUNTADORES 138
8.6 APUNTADORES A FUNCIONES. 143
8.7 MATRICES DE APUNTADORES A FUNCIÓN 145
8.8 APUNTADORES A FUNCIONES COMO PARÁMETROS DE FUNCIONES DE LA BIBLIOTECA ESTANDAR EN C. 149
Bibliografía 151
ANEXO 1. 153

INTRODUCCIÓN

Para un estudiante de Ingeniería en Comptación es fundamental la comprensión de los conceptos de los lenguajes estructurados ya que lenguajes como Java, C++, C#, etc, tienen una gran demanda en el medio laboral y, aparate de ser orientados a objetos, son lenguajes estructurados. ¿Qué indica ser estructurado? Por lo que, nociones como cohesión y acoplamiento, manejo de memoria, pases de parámetros, librerías, entornos son vitales para la comprensión de un lenguaje estructurado.

Esta Uniudad de Aprendizaje tiene la finalidad de proporcionar las habilidades que se requieren para la codificación de programas en un lenguaje de programación estructurado, cubriendo las necesidades de programación y codificación para el desarrollo de sistemas que un profesional en el área de la computación pueda tener.

PRESENTACIÓN

Este escrito fue hecho con la intención de apoyar a los estudiantes que cursan la Unidad de Aprendizaje "**Lenguaje de Programación Estructurado**". Se basa en el plan de estudios que fue actualizado en Octubre del 2013, y comprenden los siguientes temas:

Unidad de competencias I.
- Paradigmas de lenguajes de programación (LP)
- Elementos de un LP
- Origenes de los LP estructurados
- Principales herramientas de software desarrollados en LPE

Unidad de competencias II
- Elementos de un ambiente integrado de desarrollo
- Compilador. Rastreador. Consideraciones principales
- Estructura de un programa codificado.

Unidad de competencias III.
- Variales, tipos de datos, sitaxis para declaración de variables.
- Datos simples (enteros, reales, cadenas, lógico, carácter) y su codificación
- Datos estructurados (arreglos de 1 a N dimensiones, registros, camos de bites, enumeraciones.
- Variables internas o locales
- Variables externas
- Modificadores auto, extern, register, static.

Unidad de competencias IV.
- Sentencias de control de la programación estructurada secuencial, selección (simple, doble y multíple) e iterativas (mientras, repite, para) continue y break

Unidad de competencias V.
- Principales bibliotecas de funciones.
- Funciones para lectura, escritura
- Principales funciones matemáticas
- Funciones para el manejo de cadenas
- Funciones de interfase con el Sistema Operativo
- Funciones para el manejo de tiempo

Unidad de competencias VI
- Declaración y definición de funciones y/o procedimientos.
- Variables locales, variables globales, paso de parámetros.

 Alta cohesión

Unidad de competencias VII
 Tipos de archivos (Texto y binario)
 Tipos de acceso a archivos (secuencial, directo, aleatorio, etc.)
 Flujos

Unidad de competencias VIII
 Definición de memoria dinámica.
 Apuntadores
 Listas
 Pilas
 Uso de memoria dinámica
 Acceso a capas de estructuras redefinidas por apuntadores

Dentro del concepto de modularidad se incluyeron en estos apuntes tanto el concepto de cohesión como el de acoplamiento ya que se considera que es fundamental la comprensión de ambos términos.

Siendo el propósito de la Unidad de Aprendizaje “Codificar programas en un lenguaje de programación estructurado, haciendo uso de las funciones más importantes de dicho lenguaje” y sus competencias genéricas:

- Desarrollar aplicaciones informáticas de complejidad media, mediante un lenguaje de programación estructurado
- Desarrollar programas con módulos altamente cohesivos
- Usar un lenguaje de programación estructurado para la codificación de programas
- Usar apropiadamente variables, por su tipo y por su ámbito

se incluyó en los apuntes un anexo donde se trató de mostrar la modularidad, el pase de parámetros y la memoria dinámica utilizando como herramienta los “Métodos Numéricos”.

Se utilizó como lenguaje estructurado el lenguaje estándar C, ya que es el lenguaje puro estructurado en boga.

El capitulaje corresponde en forma directa a cada una de las Unidades de Competencias, incluyendo el número de horas que se deben de tener para desarrollar la Unidad de Competencias.
.

UNIDAD DE COMPETENCIA I. CONOCER LA IMPORTANCIA DE LOS LENGUAJES DE PROGRAMACIÓN ESTRUCTURADOS ASÍ COMO SU UTILIDAD

(3 horas)

RESUMEN

En esta unidad de competencias encontrarás lo que es un paradigma, los diferentes paradignas que existen en los lenguajes de programación, los orígenes de un lenguaje de programación estructurado y las principales herramientas de desarrollo de un lenguaje de programación estructurado.

PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN (LP).

(Rodríguez, 2018)

Introducción.

La palabra Paradigma proviene del griego *paradeigma* y el latín *paradigma*, cuyo significado es **ejemplo o modelo**. Es empleado para indicar un **patrón, modelo, ejemplo o arquetipo.** Alude aquellos aspectos relevantes de una situación que pueden ser **tomados como un ejemplo,** inclusive, la etimología de la palabra nos indica que esta puede ser **sinónimo de Ejemplo**, sin embargo, Paradigma es usado en otro tipo de contextos no tan simples como los usados con la palabra Ejemplo. Lo curioso de este término es su procedencia, pues de ahí es que se toma la idea que un paradigma no es más que un conjunto de acciones que seguir o ejecutar para concluir con un bien común o de fortaleza social. Derivada de la filosofía griega, fue Platón quien le dio la forma de "Ejemplo a seguir" y no como simple ejemplo como se cree al usarla en un contexto sin ningún tipo de aspiración.

Por lo que paradigma es una conceptualización más concreta de una idea o teoría subyacente, contiene definiciones, constructos e interrelaciones entre dichos constructos.

De acuerdo con Kuhn (Kuhn, 1962), un paradigma es **"una constelación de conceptos, valores, percepciones y prácticas que comparte una comunidad, la cual forma una visión particular de la realidad que es la base de la manera en que dicha comunidad se organiza a sí misma."**

Un paradigma es un modelo mental, una forma de ver, un filtro para las percepciones personales, un marco de referencia, un marco de pensamientos o creencias a través de los cuales se interpreta la realidad particular de cada quien, un ejemplo que se emplea para definir un fenómeno, una creencia compartida.

A partir de la década de los 60's, los alcances de la noción se ampliaron y 'paradigma' comenzó a ser un término común en el vocabulario científico y en expresiones epistemológicas cuando se hacía necesario hablar de modelos o patrones.

El estadounidense Thomas Kuhn, un experto en Filosofía y una figura destacada del mundo de las ciencias, fue quien se encargó de renovar la definición teórica de este término para otorgarle una acepción más acorde a los tiempos actuales, al adaptarlo para describir con él a la **serie de prácticas que trazan los lineamientos de una disciplina científica a lo largo de un cierto lapso temporal**.

De esta forma, un paradigma científico establece aquello que debe ser observado; la clase de interrogantes que deben desarrollarse para obtener respuestas en torno al propósito que se persigue; qué estructura deben poseer dichos interrogantes y marca pautas que indican el camino de interpretación para los resultados obtenidos de una investigación de carácter científico.

Cuando un paradigma ya no puede satisfacer los requerimientos de una ciencia (por ejemplo, ante nuevos hallazgos que invalidan conocimientos previos), es sucedido por otro. Se dice que un cambio de paradigma es algo dramático para la ciencia, ya que éste aparece como estable y maduro.

No obstante, también es necesario dejar muy patente que paradigma es un término que lo podemos emplear en otros campos fuera del área científica. En este sentido, también es muy utilizado, y con frecuencia, en el ámbito de la Lingüística donde se emplea para referirse a todo un conjunto de palabras que, dentro de un mismo contexto, pueden utilizar de manera indistinta.

Paradigma en lenguajes de programación.
Un paradigma de programación indica un método de realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa.

- *Los paradigmas fundamentales están asociados a determinados modelos de cómputo.*
- *También se asocian a un determinado estilo de programación*
- *Los lenguajes de programación suelen implementar, a menudo de forma parcial, varios paradigmas.*

Tipos de paradigmas.

- Los paradigmas fundamentales están basados en diferentes modelos de cómputo y por lo tanto afectan a las construcciones más básicas de un programa.
- La división principal reside en el enfoque imperativo (indicar el cómo se debe calcular) y el enfoque declarativo (indicar el qué se debe calcular).
- El enfoque declarativo tiene varias ramas diferenciadas: el paradigma funcional, el paradigma lógico, la programación reactiva y los lenguajes descriptivos.

Otros paradigmas se centran en la estructura y organización de los programas, y son compatibles con los paradigmas fundamentales:

Ejemplos: Programación estructurada, modular, orientada a objetos, orientada a eventos, programación genérica. Por último, existen paradigmas asociados a la concurrencia y a

los sistemas de tipado.

Paradigma Imperativo

- Describe cómo debe realizarse el cálculo, no el qué se debe calcular.
- Un cómputo consiste en una serie de sentencias, ejecutadas según un control de flujo explícito, que modifican el estado del programa.
- Las variables son celdas de memoria que contienen datos (o referencias), pueden ser modificadas, y representan el estado del programa.
- La sentencia principal es la asignación.
- Es el estándar 'de facto'.
    - Asociados al paradigma imperativo se encuentran los paradigmas procedural, modular, y la programación estructurada.
    - El lenguaje representativo sería FORTRAN-77, junto con COBOL, BASIC, PASCAL, C, ADA.
    - También lo implementan Java, C++, C#, Eiffel, Python, etc.

Paradigma Declarativo

- Describe que se debe calcular, sin explicitar el cómo.
- No existe un orden de evaluación prefijado.
- Las variables son nombres asociados a definiciones, y una vez instanciadas son inmutables.
- No existe sentencia de asignación.
- El control de flujo suele estar asociado a la composición funcional, la recursividad y/o técnicas de reescritura y unificación.
    - Existen distintos grados de pureza en las variantes del paradigma.
    - Las principales variantes son los paradigmas funcional, lógico, la programación reactiva y los lenguajes descriptivos.

Programación Funcional

- Basado en los modelos de cómputo cálculo lambda (Lisp, Scheme) y lógica combinatoria (familia ML, Haskell)
- Las funciones son elementos de primer orden
- Evaluación por reducción funcional. Técnicas: recursividad, parámetros acumuladores, CPS, Mónadas (las mónadas son contextos).
- Familiar LISP (Common-Lisp, Scheme):
    1. Basados en s-expresiones.
    2. Tipiado débil
    3. Meta-programación
- Familia ML (Miranda, Haskell, Scala):
    1. Sistema estricto de tipos (tipado algebraico)
    2. Concordancia de patrones.

3. Transparencia referencial
4. Evaluación perezosa (estructura de datos infinitas)

Programación Lógica

- Basado en la lógica de predicados de primer orden
- Los programas se componen de hechos, predicados y relaciones.
- Evaluación basada en resolución SLD: unificación + backtracking.
- La ejecución consiste en la resolución de un problema de decisión, los resultados se obtienen mediante la instanciación de las variables libres.
- Lenguaje representativo: PROLOG.

Programación Reactiva (Dataflow)

(Ver figura 1)

- Basada en la teoría de grafos.
- Un programa consiste en la especificación del flujo de datos entre operaciones.
- Las variables se encuentran ligadas a las operaciones que proporcionan sus valores. Un cambio de valor de una variable se propaga a todas las operaciones en que participa.
- Las hojas de cálculo se basan en este modelo.
- Lenguajes representativos: Simulink, Oz, Clojure.

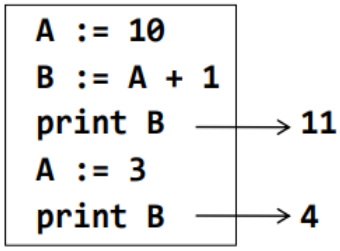


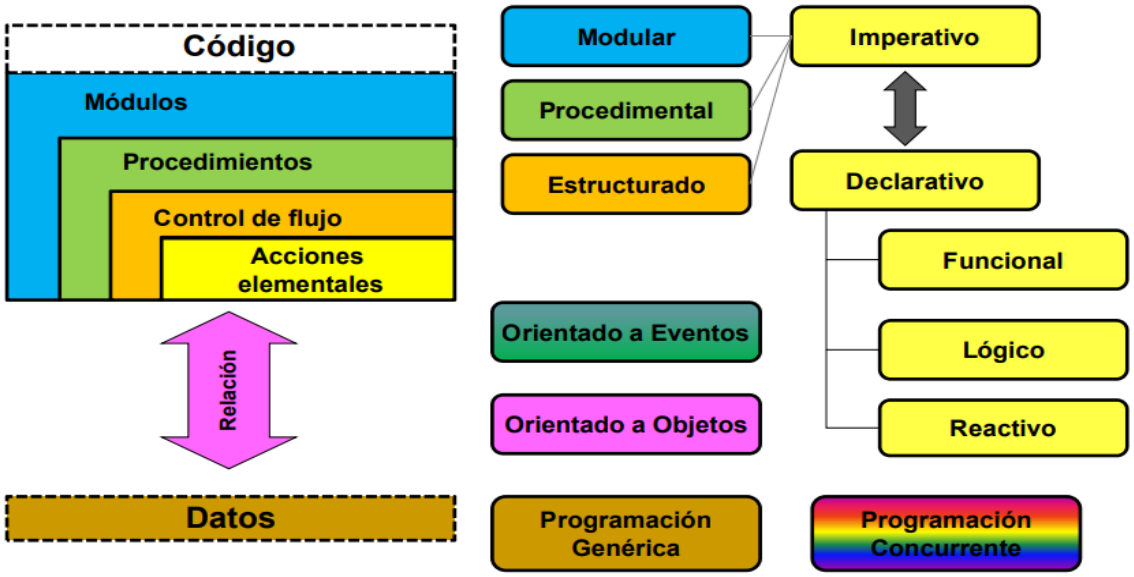
Figura 1. Un ejemplo de programación reactiva.

EL PARADIGMA ORIENTADO A OBJETOS: UNA FORMA DE VER EL MUNDO

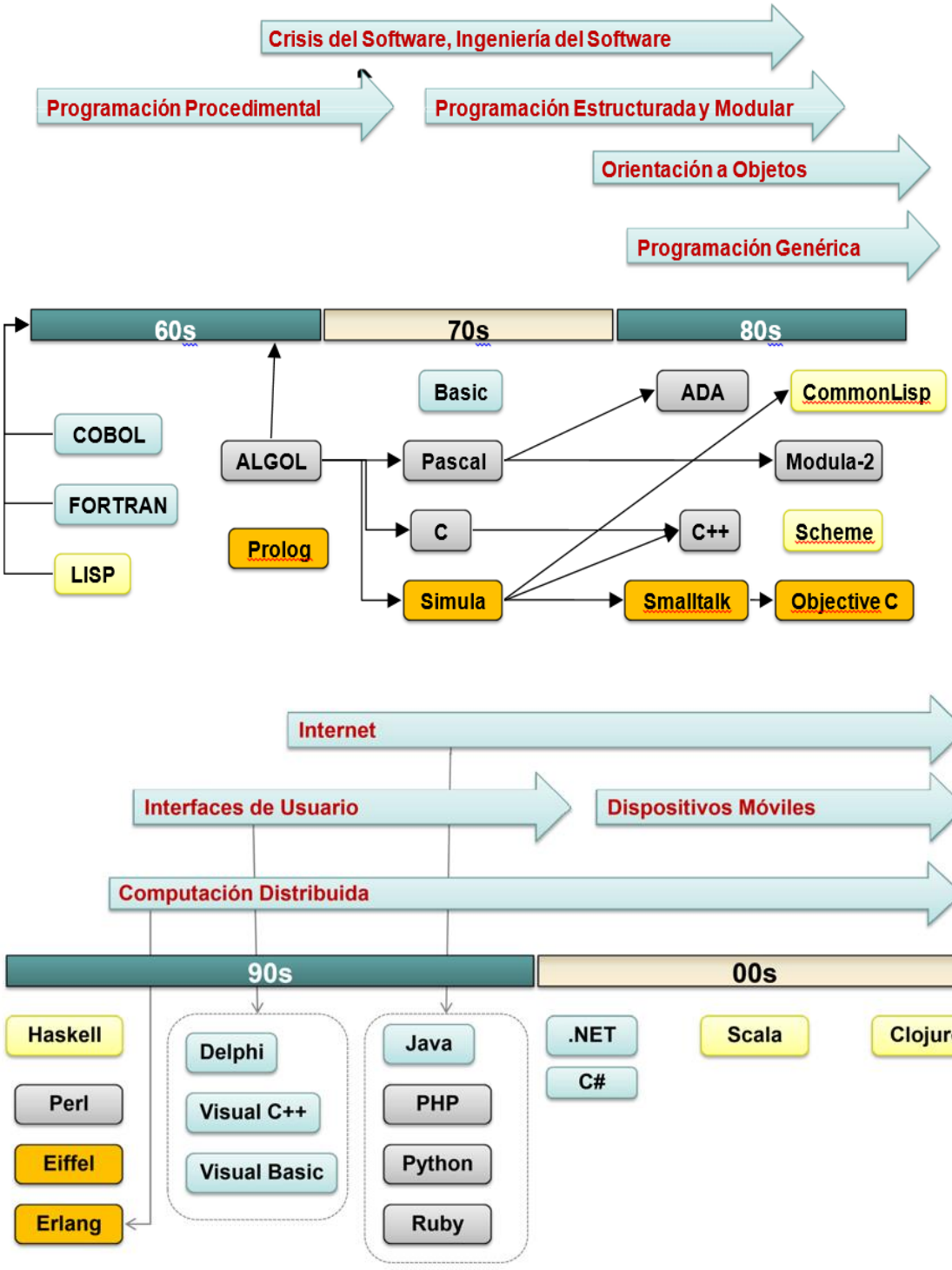
El paradigma orientado a objetos define los programas en términos de comunidades de objetos. Los objetos con características comunes se agrupan en clases (un concepto similar al de tipo abstracto de dato (TAD)). Los objetos son entidades que combinan un estado (es decir, datos) y un comportamiento (esto es, procedimientos o métodos). Estos objetos se comunican entre ellos para realizar tareas. Es en este modo de ver un programa donde este paradigma difiere del paradigma imperativo o estructurado, en los que los datos y los métodos están separados y sin relación. El paradigma OO surge para solventar los problemas que planteaban otros paradigmas, como el imperativo, con el objeto de elaborar programas y módulos más fáciles de escribir, mantener y reutilizar.

Entre los lenguajes que soportan el paradigma OO están Smalltalk, C++, Delphi (Object Pascal), Java y C#

Ver figura 1.1 para observar una secuencia histórica de los paradigmas de programación.



Generación	Lenguajes	Hardware	Movimientos
Primera (1945-55)	Código Máquina	Relés, Válvulas de vacío	
Segunda (1955-68)	FORTRAN COBOL LISP	Transistores, Memorias de ferrita	Prog. Estructurada y Modular Proceso por Lotes
Tercera (1968-1980)	ALGOL PASCAL C BASIC ADA	Circuitos integrados, Memorias de transistores	Ingeniería de Software Orientación a Objetos Bases de Datos
Cuarta (1980-)	C++ JAVA HASKELL PYTHON	VLSI MultiCore Flash	Comp. Distribuida Interfaces Gráficas Multimedia Internet


Crisis del Software, Ingeniería del Software
Programación Procedimental
Programación Estructurada y Modular
Orientación a Objetos
Programación Genérica
60s
70s
80s
COBOL
FORTRAN
LISP
ALGOL
Prolog
Basic
Pascal
C
Simula
ADA
CommonLisp
Modula-2
C++
Scheme
Smalltalk
Objective C
Internet
Interfaces de Usuario
Dispositivos Móviles
Computación Distribuida
90s
00s
Haskell
Perl
Eiffel
Erlang
Delphi
Visual C++
Visual Basic
Java
PHP
Python
Ruby
.NET
C#
Scala
Clojure

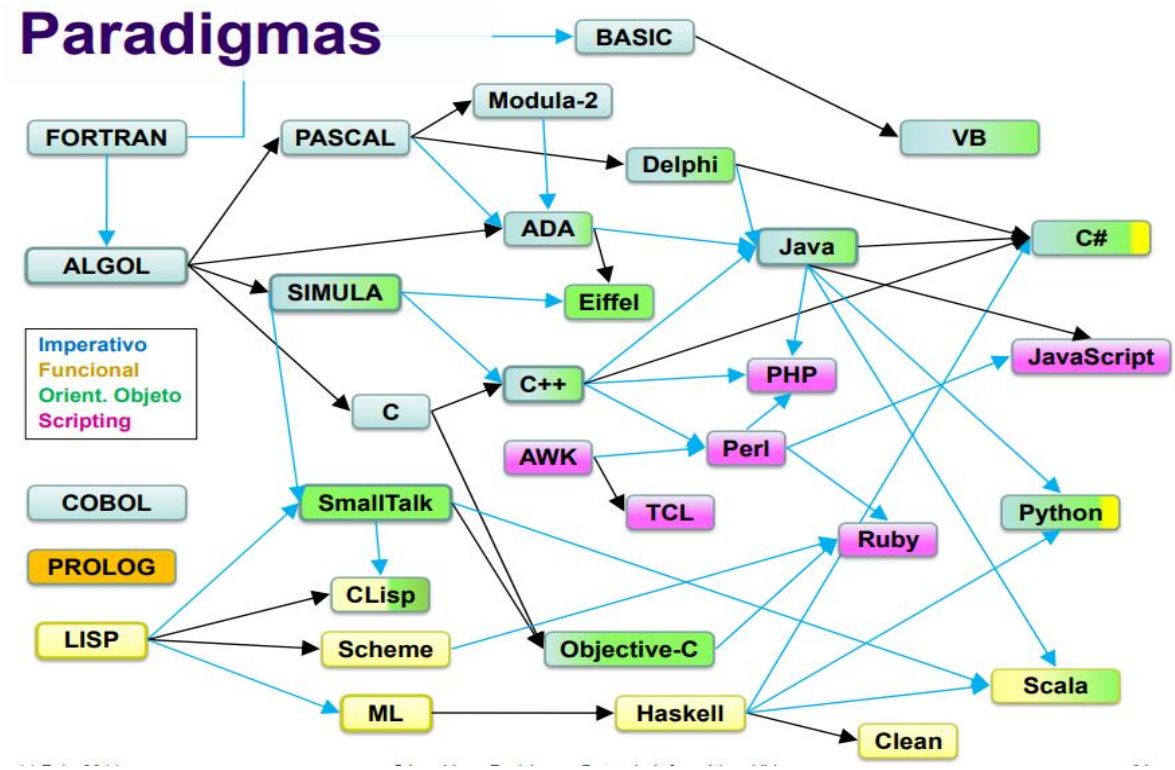


Fig. 1.1 Generaciones, líneas de tiempo y paradigmas

Ventajas de la programacion imperativa.

- Su relativa simplicidad y facilidad de implementación de los compiladores e intérpretes.
- La capacidad de reutilizar el mismo código en diferentes lugares en el programa sin copiarlo.
- Una forma fácil seguir la pista de flujo del programa.
- La capacidad de ser modular o estructurado.
- Necesita menos memoria.

Desventajas.

- Los datos son expuestos a la totalidad del programa, así que no hay seguridad para los datos.
- Dificultad para relacionarse con los objetos del mundo real.
- Difícil crear nuevos tipos de datos reduce la extensibilidad.
- Se da importancia a la operación de datos en lugar de los datos mismos.

PARADIGMA ORIENTADO A OBJETOS.

Ventajas

- Aumenta la reusabilidad
- Forma más modular de modelar problemas
- diseños más claros

- implementación más clara
- mejor manejo de la complejidad
- Mantenimiento más eficiente

Desventajas

- La OO no garantiza la construcción de un sistema correcto como ningún otro método de programación.
- Mayor concentración en requerimientos, análisis y diseño OO, no es solo la programación, es entender la abstracción.
- Las ventajas de OO se obtienen a largo plazo.

ELEMENTOS DE LOS LENGUAJES DE PROGRAMACIÓN.
(Algoritmia, 2018)

Cualquier lenguaje de programación trabaja con tres elementos: información, operaciones e instrucciones.

INFORMACIÓN

Esta se refiere a los datos con los cuales trabajarán los programas. Los datos suelen ser de dos tipos: numéricos y alfanuméricos (ó caracteres). Los datos se pueden agrupar formando estructuras, las estructuras pueden ser muy simples como las constantes (28) y las variables (x) o muy complejas como las matrices y registros (ver figura 1.2).

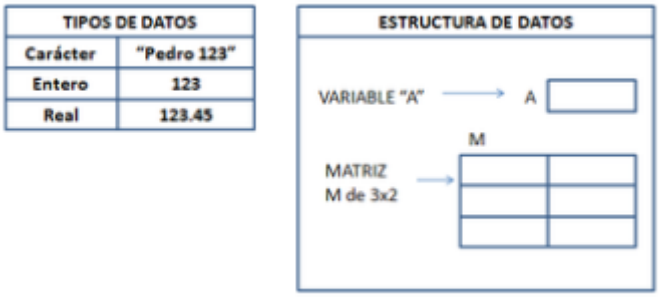


Fig. 1.2. Tipos de datos y estructura de datos

LAS OPERACIONES
Se refieren a las operaciones que el ordenador es capaz de realizar con los diversos tipos de datos. Por ejemplo, la suma de dos números se puede realizar por un ordenador directamente; pero para realizar la suma de dos matrices habría que hacer un programa (existen algunos lenguajes de programación que permiten directamente realizar sumas de matrices).

Las operaciones pueden ser realizadas mediante operadores y funciones predefinidas

que se aplican a un tipo de datos determinado. Por ejemplo, para los datos de tipo numérico se permiten los operadores suma, resta, división y producto, cuya operación se representaría:

```
a=5*3
c=6*b
```

Las funciones que dispone el lenguaje de programación para realizar operaciones se denominan predefinidas, ya que vienen definidas de antemano. Su utilización es como sigue:

```
a=sqrt(10)
b=log(b)
```

Normalmente para definir las operaciones con operadores y funciones se utiliza el término expresiones, que pueden ser operaciones con operadores, con funciones o con ambas, un ejemplo sería:

```
a=b*log(b)+sqrt(i)
```

Evidentemente existen expresiones tanto numéricas como de caracteres y estas expresiones pueden trabajar con distintas estructuras de datos.

INSTRUCCIONES

Las instrucciones (también denominadas sentencias) son el conjunto de órdenes que se le pueden dar al ordenador. En función del tipo de orden que indique al ordenador, las instrucciones se clasifican en los siguientes tipos:

-Instrucciones de entrada y salida (E/S).

Figura 1.3. I/O

También denominadas de lectura/escritura (L/S) ó input/output (I/O) (ver figura 1.3). Estas instrucciones se encargan de la información que necesita un programa para realizar su tarea y de la información que genera el programa. Son instrucciones del tipo: Almacena en la variable "nom" el nombre del usuario ó escribe el contenido de la variable "nom". Normalmente la entrada y la salida se realizan desde un archivo ya creado (información almacenada en un dispositivo magnético) o es el usuario quien suministra o recibe la información a través del teclado y pantalla del ordenador. A continuación se describen los dispositivos de entrada y salida más comúnmente utilizados por los ordenadores. Para cada uno de ellos existirían instrucciones de entrada y salida.

Dispositivos de entrada.
Son los utilizados para introducir la información en un ordenador. Los dispositivos de entrada más comunes son: ratón, escáner, sistemas ocr (Optical Character Recognition), módem, sistemas de audio, etc
Existen además otros muchos dispositivos conectables al ordenador (que envían información a los dispositivos de procesamiento). Realmente se puede conectar cualquier dispositivo que generará alguna señal o información (osciloscopios, medidores de presión, alarmas, etc.).

Dispositivos de salida.
Operan con la información en sentido inverso al de los dispositivos de entrada; es decir, reciben la información del ordenador. Normalmente estos dispositivos muestran el estado de la información que contiene el ordenador.
Los dispositivos de salida más utilizados son la pantalla, la impresora, los plotters o el módem.

En general se puede conectar al ordenador cualquier dispositivo que para funcionar

necesite un control mediante información. Podría ser un robot, una escalera mecánica, un reloj, etc.

-Instrucciones de control.

Son instrucciones que sirven para dirigir la ejecución de un programa (Ver figura 1.4). Normalmente un programa está compuesto por un conjunto de instrucciones que se ejecutan una tras otra. Las instrucciones de control permiten cambiar la secuencia de ejecución.

Son instrucciones del tipo: Si ocurre tal condición ejecuta determinadas instrucciones, de lo contrario ejecuta otras; o cuando llegues a esta instrucción vete a la primera. La siguiente figura muestra el control de la ejecución para este tipo de sentencias.

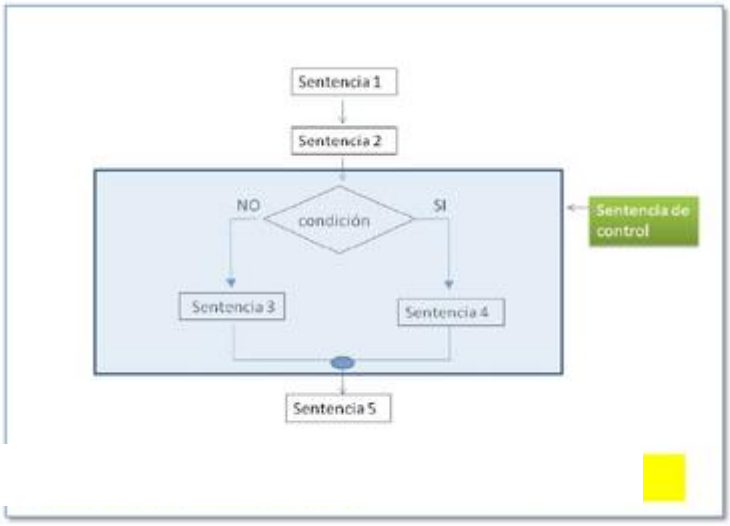


Fig. 1.4 Instrucciones de control

-Instrucciones iterativas.

Son instrucciones que permiten repetir un número determinado de veces un conjunto de instrucciones (Ver figura 1.5). Por ejemplo, si deseo realizar un programa que escriba los números enteros del 1 al 10.987.654 se puede realizar de dos formas: escribiendo diez millones novecientas ochenta y siete mil seiscientas cincuenta y cuatro instrucciones de salida o un sencillo bucle con cuatro sentencias. La siguiente figura muestra la situación citada.

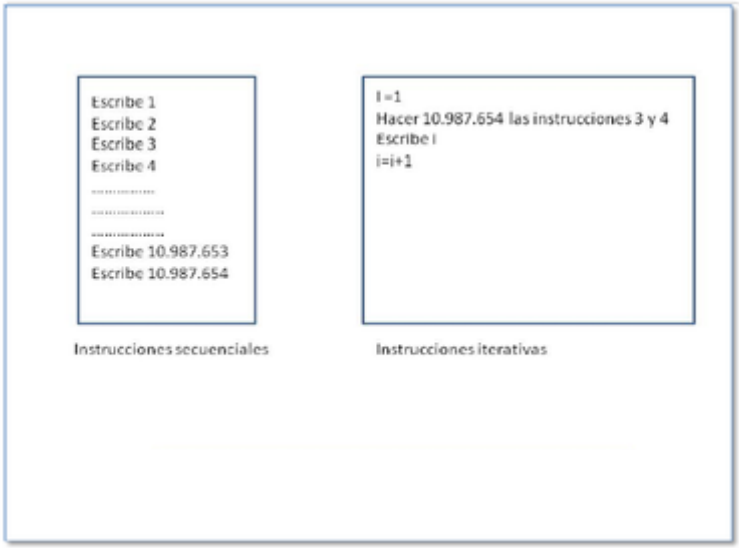


Fig. 1.5 Instrucciones iterativas permiten repetir n veces un conjunto de instrucciones

ORIGENES DE LOS LENGUAJES DE PROGRAMACIÓN ESTRUCTURADOS
(McGraw-Hill Education, s.f.)
(http://onaprogest.blogspot.com/, 2018)
(Wikipedia, 2018)

La década del sesenta fue el principio de lo que más tarde sería la Programación Estructurada, dando lugar a programas fiable y eficientes, además estaban escritos para facilitar su comprensión; posteriormente, se liberó el conjunto de las llamadas "Técnicas para mejoramiento de la productividad en programación" (en ingles Improved Programming Technologies, abreviado IPTs), siendo la Programación Estructurada una de ellas.
A partir de C y Pascal; se dividen los lenguajes en estructurados en contraposición a los lenguajes no estructurados como el Basic cuya codificación se basaba en líneas de programación, permitiendo al programador "saltar" de una línea de instrucción a otra, haciendo que el código fuera algunas veces inentendible y difícil de modificar.

En Programación Estructurada todas las ramificaciones de control de un programa se encuentran estandarizadas, es decir que es posible leer la codificación del mismo desde su inicio hasta su terminación en forma continua, sin tener que saltar de un lugar a otro del programa siguiendo el rastro de la lógica establecida por el programador.

La programación si GO TO.

GOTO es una instrucción propia de los primeros lenguajes de programación, como BASIC. Esta instrucción suele existir en todos los lenguajes aunque con un mnemónico adaptado al propio lenguaje.

El propósito de la instrucción es transferir el control a un punto determinado del código, donde debe continuar la ejecución. El punto al que se salta, viene indicado por una etiqueta. GOTO es una instrucción de salto incondicional.

En la figura 1.6 se muestra un programa en QBASIC utilizando el comando GOTO.

```
:PRINT "Prueba de GOTO en QBASIC"
:GOTO prueba
:PRINT "Esto no se ve porque hemos saltado a la etiqueta llamada prueba"
:prueba:
:PRINT "GOTO realizado correctamente!"
```

Fig. 1.6. Programa en QBASIC mostrando el comando GOTO.

La instrucción GOTO ha sido menospreciada en los lenguajes de alto nivel, debido a la dificultad que presenta para poder seguir adecuadamente el flujo del programa, tan necesario para verificar y corregir los programas.

La visión clásica de la programación estructurada se refiere al control de ejecución. El control de su ejecución es una de las cuestiones más importantes que hay que tener en cuenta al construir un programa en un lenguaje de alto nivel. La regla general es que las instrucciones se ejecuten sucesivamente una tras otra, pero diversas partes del programa se ejecutan o no dependiendo de que se cumpla alguna condición. Además, hay instrucciones (los bucles) que deben ejecutarse varias veces, ya sea en número fijo o hasta que se cumpla una condición determinada.

Sin embargo, algunos lenguajes de programación más antiguos (como Fortran) se apoyaban en una sola instrucción para modificar la secuencia de ejecución de las instrucciones mediante una transferencia incondicional de su control (con la instrucción *goto*, del inglés "go to", que significa "ir a"). Pero estas transferencias arbitrarias del control de ejecución hacen los programas muy poco legibles y difíciles de comprender. A finales de los años sesenta, surgió una nueva forma de programar que reduce a la mínima expresión el uso de la instrucción *goto* y la sustituye por otras más comprensibles.

Esta forma de programar se basa en un famoso teorema, desarrollado por Edsger Dijkstra, que demuestra que todo programa puede escribirse utilizando únicamente las tres estructuras básicas de control siguientes:

- **Secuencia:** el bloque secuencial de instrucciones, instrucciones ejecutadas sucesivamente, una detrás de otra.
- **Selección:** la instrucción condicional con doble alternativa, de la forma "*if*condición *then*instrucción-1 *else*instrucción-2".

- **Iteración:** el bucle condicional "*while*condición *do*instrucción", que ejecuta la instrucción repetidamente mientras la condición se cumpla.

Los programas que utilizan sólo estas tres instrucciones de control básicas o sus variantes (como los bucles *for, repeat*o la instrucción condicional *switch-case*), pero no la instrucción *goto*, se llaman **estructurados.**

Ésta es la noción clásica de lo que se entiende por **programación estructurada** (llamada también **programación sin *goto***) que hasta la aparición de la programación orientada a objetos se convirtió en la forma de programar más extendida. Esta última se enfoca hacia la reducción de la cantidad de estructuras de control para reemplazarlas por otros elementos que hacen uso del concepto de polimorfismo. Aun así los programadores todavía utilizan las estructuras de control (*if, while, for,*etc.) para implementar sus algoritmos porque en muchos casos es la forma más natural de hacerlo.

Una característica importante en un programa estructurado es que puede ser leído en secuencia, desde el comienzo hasta el final sin perder la continuidad de la tarea que cumple el programa, lo contrario de lo que ocurre con otros estilos de programación. Este hecho es importante debido a que es mucho más fácil comprender completamente el trabajo que realiza una función determinada si todas las instrucciones que influyen en su acción están físicamente contiguas y encerradas por un bloque. La facilidad de lectura, de comienzo a fin, es una consecuencia de utilizar solamente tres estructuras de control, y de eliminar la instrucción de transferencia de control *goto*.

La visión moderna de la programación estructurada: la segmentación.

La realización de un programa sin seguir una técnica de programación produce frecuentemente un conjunto enorme de sentencias cuya ejecución es compleja de seguir, y de entender, pudiendo hacer casi imposible la depuración de errores y la introducción de mejoras. Se puede incluso llegar al caso de tener que abandonar el código preexistente porque resulte más fácil empezar de nuevo.

Cuando en la actualidad se habla de programación estructurada, nos solemos referir a la división de un programa en partes más manejables (usualmente denominadas **segmentos** o **módulos**). Una regla práctica para lograr este propósito es establecer que cada segmento del programa no exceda, en longitud, de una página de codificación, o sea, alrededor de 50 líneas.

Así, la visión moderna de un programa estructurado es un compuesto de segmentos, los cuales puedan estar constituidos por unas pocas instrucciones o por una página o más de código. Cada segmento tiene solamente una entrada y una salida, asumiendo que no poseen bucles infinitos y no tienen instrucciones que jamás se ejecuten. Encontramos la relación entre ambas visiones en el hecho de que los segmentos se combinan utilizando las tres estructuras básicas de control mencionadas anteriormente y, por tanto, el resultado es también un programa estructurado.

Cada una de estas partes englobará funciones y datos íntimamente relacionados semántica o funcionalmente. En una correcta partición del programa deberá resultar fácil e intuitivo comprender lo que debe hacer cada módulo.

En una segmentación bien realizada, la comunicación entre segmentos se lleva a cabo de una manera cuidadosamente controlada. Así, una correcta partición del problema producirá una nula o casi nula dependencia entre los módulos, pudiéndose entonces trabajar con cada uno de estos módulos de forma independiente. Este hecho garantizará que los cambios que se efectúen a una parte del programa, durante la programación original o su mantenimiento, no afecten al resto del programa que no ha sufrido cambios.

Esta técnica de programación conlleva las siguientes ventajas:

a) El coste de resolver varios subproblemas de forma aislada es con frecuencia menor que el de abordar el problema global.

b) Facilita el trabajo simultáneo en paralelo de distintos grupos de programadores.

c) Posibilita en mayor grado la reutilización del código (especialmente de alguno de los módulos) en futuras aplicaciones.

Aunque no puede fijarse de antemano el número y tamaño de estos módulos, debe intentarse un compromiso entre ambos factores. Si nos encontramos ante un módulo con un tamaño excesivo, podremos dividir éste a su vez en partes (nuevos módulos) más manejables, produciéndose la sucesiva división siempre desde problemas generales a problemas cada vez menos ambiciosos y, por tanto, de fácil desarrollo y seguimiento. Así, esta división toma la forma de un árbol cuya raíz es el programa principal que implementa la solución al problema que afrontamos utilizando uno o varios módulos que realizan partes de dicha solución por sí solos o invocando a su vez otros módulos que solucionan subproblemas más específicos. A esta aproximación se la denomina **diseño descendente** o ***top-down*****,** como queda esquematizado en la figura 1.7:

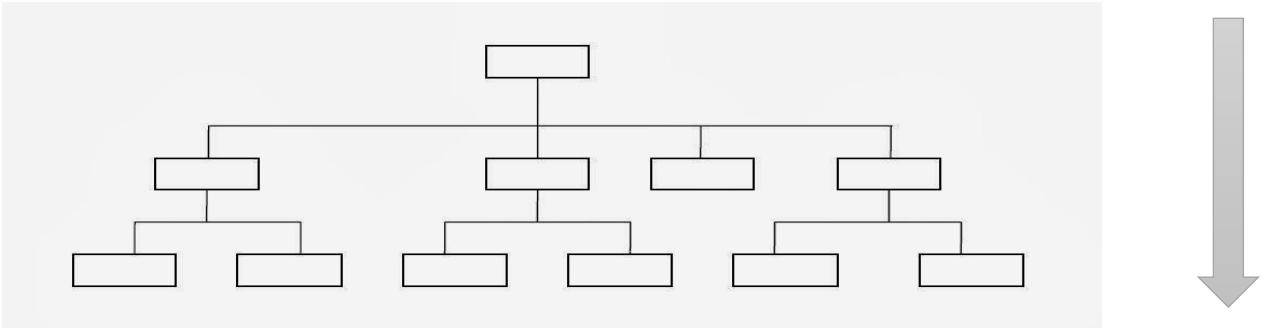


Fig. 1.7 Diseño Top Down

El carácter autocontenido de los módulos o librerías hace que pueda ocultarse el funcionamiento interno de las funciones contenidas en un módulo, ya que para utilizarlas basta con saber con qué nombre y argumentos se invocan y qué tipo de valores devuelven.

Al reunirlas en un módulo, se realza la relación entre las mismas separándolas del resto del programa.

Esta ocultación de los detalles se denomina **encapsulación** y se alcanza dividiendo el código del programa en dos archivos diferenciados: un archivo (con extensión ".h") que incluye la declaración de los tipos de datos y de las funciones gracias a lo cual se sabe cómo acceder y utilizar cada una de las mismas y otro (con extensión ".c") que contiene el código de cada una de las funciones declaradas en el .h.

Al compilar este último queda transformado en código objeto (al cual ya no se puede acceder para su lectura o modificación) y puede distribuirse conjuntamente con el fichero de declaración (el que acaba en .h), para su uso por parte de terceros sin riesgo alguno de alteración de la funcionalidad original (ésta quedó encapsulada u oculta).

Esto es así porque para hacer uso de las funciones incluidas en el módulo únicamente necesitaremos conocer la información que nos proporciona el fichero de declaración: el nombre, tipos de datos de los argumentos y valores de retorno de las funciones. No es necesario conocer los detalles de implementación (sentencias que se ejecutan dentro de una función).

PRINCIPALES HERRAMIENTAS DE SOFTWARE DESARROLLADOS EN LPE.

(Lenguajes de Programación, 2018)
(Modelling Ambient Intelligence Research Lab, 2018)

Las herramientas de programación, son aquellas que permiten realizar aplicativos, programas, rutinas, utilitarios y sistemas para que la parte física del computador u ordenador, funcione y pueda producir resultados.

Hoy día existen múltiples herramientas de programación en el mercado, tanto para analistas expertos como para analistas inexpertos.

Las herramientas de programación más comunes del mercado, cuentan hoy día con programas de depuración o debugger, que son utilitarios que nos permiten detectar los posibles errores en tiempo de ejecución o corrida de rutinas y programas.

Entre otras herramientas de programación encontramos librerías y componentes, dados por algunos lenguajes de programación como son el C++ y Delphi.

Otras herramientas de programación son los lenguajes de programación, que nos permiten crear rutinas, programas y utilitarios.

Entre algunas de estas herramientas de programación tenemos:

Basic y Pascal que son herramientas de programación, idóneas para la inicialización de los programadores.

C y C++ que sirven para la programación de sistemas.

Cobol, que es una herramienta de programación orientada hacia sistemas de gestión empresarial como nóminas y contabilidad.

Fortran, que son lenguajes específicos para cálculos matemáticos y o numéricos.

Herramientas de programación para ambientes gráficos como son Visual Basic, Delphi y Visual C.

Html y Java, que permiten la creación de páginas WEB para internet.

Además se cuentan con las herramientas **CASE** (Ingeniería de Software asistida por computación), siendo ésta:

- Tecnologías CASE: Automatización del Desarrollo del Software. Ingeniería del Software asistida por Computador.
- Son Herramientas y Metodologías que se aplican a todo el ciclo de vida del desarrollo del SW.
    - Herramientas autónomas o integradas de productividad que automatizan en todo o en parte, tareas del ciclo de vida del Desarrollo del Software.
    - Metodologías estructuradas y automatizables que definen una formulación técnica y disciplinada para todos o alguno(s) de los aspectos del desarrollo del SW. Ejemplos: Análisis Estructurado o la Programación Estructurada.
- Las tecnologías CASE se centran en la productividad y no solo en obtener soluciones.
- Columna vertebral de la tecnología CASE: Automatización y Productividad.
- Desde los años 70 con sistemas de documentación automática.

EJEMPLOS

- Herramientas de diagramación para especificar esquemas estructurados
- Diccionarios y sistemas de datos con información de gestión de proyectos
- Herramientas de validación sintáctica o de inconsistencias.
- Generadores automáticos de código a partir de otras especificaciones (por ejemplo, gráficas)
- Generadores automáticos de documentación técnica y de usuario

Glosario de definiciones básicas CASE

- KIT de HERRAMIENTAS CASE: Un conjunto de herramientas CASE integradas que se han diseñado para trabajar juntas y automatizar (o proveer ayuda automatizada al ciclo de desarrollo de software, incluyendo el análisis, diseño, **codificación** y pruebas.
- METODOLOGIA CASE: Conjunto estructurado de métodos que definen una disciplina de la ingeniería como un acercamiento a todos o algunos aspectos del desarrollo y mantenimiento de software.
- PUESTO DE TRABAJO para CASE: Una estación de trabajo técnica o computadora personal equipada con Herramientas Case que automatiza varias funciones del ciclo. [obsoleto]
- PLATAFORMA de HARDWARE para CASE: Una arquitectura de hardware con uno, dos o tres sistemas puestos en línea, que proveen una plataforma operativa para las Herramientas Case. [obsoleto]

Existen varios componentes de las herramientas CASE. Las características más importantes de los generadores de código son:

- Lenguaje generado. Si se trata de un lenguaje estándar o un lenguaje propietario.
- Portabilidad del código generado. Capacidad para poder ejecutarlo en diferentes plataformas físicas y/o lógicas.
- Generación del esqueleto del programa o del programa completo. Si únicamente genera el esqueleto será necesario completar el resto mediante programación.
- Posibilidad de modificación del código generado. Suele ser necesario acceder directamente al código generado para optimizarlo o completarlo.
- Generación del código asociado a las pantallas e informes de la aplicación. Mediante esta característica se obtendrá la interface de usuario de la aplicación.

UNIDAD DE COMPETENCIA II. MANEJO DEL ENTORNO INTEGRADO DE DESARROLLO

(2 horas)

RESUMEN

En esta Unidad de Competencias se analizan los elementos de un ambiente integrado de desarrollo. Se explicará en grandes rasgos que es un compilador, se comenta las etapas de montaje, ejecución y depuración utilizando el IDE más utilizado en el ambiente estudiantil para el lenguaje de programación C (DEV-C).

Elementos de un ambiente integrado de desarrollo.
(Juan Manuel Muñoz, 2018)
(fergarciac, 2018)
(Fundamentos de Informática, 2018)

Un entorno de desarrollo integrado, es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste de un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

Los IDE proveen un marco de trabajo amigable para la mayoría de los lenguajes de programación tales como C++, PHP, Python, Java, C#, Delphi, Visual Basic, etc. En algunos lenguajes, un IDE puede funcionar como un sistema en tiempo de ejecución, en donde se permite utilizar el lenguaje de programación en forma interactiva, sin necesidad de trabajo orientado a archivos de texto.

Algunos ejemplos de entornos integrados de desarrollo (IDE) son los siguientes:

- Eclipse
- NetBeans
- IntelliJ IDEA
- JBuilder de Borland
- JDeveloper de Oracle
- KDevelop
- Anjunta
- Clarion
- MS Visual Studio
- Visual C++

Es posible que un mismo IDE pueda funcionar con varios lenguajes de programación. Este es el caso de Eclipse, al que mediante plagios se le puede añadir soporte de lenguajes adicionales.
Un IDE debe tener las siguientes características:

- Multiplataforma
- Soporte para diversos lenguajes de programación
- Integración con Sistemas de Control de Versiones
- Reconocimiento de Sintaxis
- Extensiones y Componentes para el IDE
- Integración con Framework populares
- Depurador
- Importar y Exportar proyectos
- Múltiples idiomas
- Manual de Usuarios y Ayuda

Existen diferentes versiones de los IDEs pero estos son algunos del software que utilizan IDE, estos son:

a) Eclipse: Software libre. Es uno de los entornos Java más utilizados a nivel profesional. El paquete básico de Eclipse se puede expandir mediante la instalación de plugins para añadir funcionalidades a medida que se vayan necesitando.
b) NetBeans: Software libre. Otro de los entornos Java muy utilizados, también expandible mediante plugins. Facilita bastante el diseño gráfico asociado a aplicaciones Java.
c) BlueJ: Software libre. Es un entorno de desarrollo dirigido al aprendizaje de Java (entorno académico) y sin uso a nivel profesional. Destaca por ser sencillo e incluir algunas funcionalidades dirigidas a que las personas que estén aprendiendo tengan mayor facilidad para comprender aspectos clave de la programación orientada a objetos.
d) JBuilder: Software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en la web, buscando en la sección de productos y desarrollo de aplicaciones. Permite desarrollos gráficos.
e) JCreator: Software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en la web. Este IDE está escrito en C++ y omite herramientas para desarrollos gráficos, lo cual lo hace más rápido y eficiente que otros IDEs.
Ventajas de los IDEs.
1. La curva de aprendizaje es muy baja.

2. Es más ágil y óptimo para los usuarios que no son expertos en manejo de consola.
3. Formateo de código.
4. Funciones para renombrar variables, funciones.
5. Warnings y errores de sintaxis en pantalla de algo que no va a funcionar al interpretar o compilar.
6. Poder crear proyectos para poder visualizar los archivos de manera gráfica.
7. Herramientas de refactoring como por ejemplo seria extraer una porción de código a un método nuevo.
8. No es recomendado pero posee un navegador web interno por si queremos probar las cosas dentro de la IDE.

Algunos entornos son compatibles con múltiples lenguajes de programación, como Eclipse o NetBeans, ambos basados en Java; o MonoDevelop, basado en C#. También puede incorporarse la funcionalidad para lenguajes alternativos mediante el uso de plugins. Por ejemplo, Eclipse y NetBeans tienen plugins para C, C++, Ada, Perl, Python, Ruby y PHP, entre otros.

Tanta es la simbiosis que se da entre algunos IDE y sus compiladores, que muchos informáticos noveles, suelen confundir ambas cosas y la realidad es que se trata de programas distintos, pues el compilador es el software encargado de traducir (compilar) el código fuente a lenguaje maquina (código binario), por lo tanto es una más de las herramientas que integran un entorno de desarrollo.

En el caso del lenguaje C y su evolución orientada a objetos, el C++, los compiladores más conocidos son GCC (GNU Compiler Collection), MinGW (implementación de GCC para Windows), los antiguos Turbo C y Turbo C++ que eran tanto compiladores como IDE para el sistema operativo MS-DOS y fueron descontinuados, siendo sustituidos por C++Builder, y Visual C++ (que forma parte de Visual Studio).

Principales IDE para C y C++
Algunos de los compiladores mencionados anteriormente, pueden emplearse en combinación con algunos de los IDE que se mencionan a continuación, otros ya poseen su propio compilador incorporado. En cualquier caso, los IDE más utilizados en la programación con C/C++, son los siguientes:

Dev-C++: Este emplea el compilador MinGW y esta creado mediante el Entorno Delphi y el lenguaje Object Pascal, se trata de un software libre, sencillo, ligero y eficiente, para la plataforma Windows, que cuenta con un arsenal de herramienta

que permiten la programación rápida de aplicaciones complejas. Actualmente está desactualizado, también existen variantes de este IDE como wxDev C++ que integra características extra para la creación de interfaces.

Code::Blocks: Es una alternativa a Dev-C++ que ha sido desarrollada mediante el propio lenguaje C++, también es un software libre, pero en este caso es multiplataforma. Sus capacidades son bastante buenas y es muy popular entre los nuevos programadores. Se puede encontrar separado del compilado o la versión "mingw" que incluye g++ (GCC para C++).

Visual C++: Como ya he mencionado, se trata tanto de un módulo del entorno Visual Studio como un compilador, posee editor de interfaces gráficas y una serie de asistentes que hacen muy cómodo el proceso de desarrollo. En realidad el lenguaje que se emplea es un dialecto de C++ compatible con la plataforma .NET de Microsoft.

C++Builder: Esta es la opción desarrollada por Borland (antigua líder en producción de compiladores C++) y actualmente propiedad de la empresa Embarcadero Technologies, es un software propietario y de pago, destinado a la plataforma Windows.

NetBeans: Este IDE libre y multiplataforma, está destinado originalmente a la programación mediante el lenguaje Java, sin embargo con la implementación un paquete de software adicional, puede emplearse para desarrollar mediante C/C++.

Eclipse: Al igual que el anterior, su principal propósito es programar mediante Java pero puede expandirse para soportar C++ y también es tanto libre como multiplataforma.

MonoDevelop: Esta es una alternativa a Visual Studio, no tan conocida como su competidor, pues sus características son similares, pero este IDE es multiplataforma y de software libre. Posee un editor de interfaces graficas que implementa la biblioteca GTK y es compatible con el .Net Framework de Microsoft.

Xcode: Para la plataforma Mac este es el IDE más utilizado, trabaja en combinación con el compilador GCC y con Interface Builder, este último es un diseñador de interfaz gráfica que facilita el diseño mediante el proceso de arrastrar y soltar controles en un espacio de trabajo.

KDevelop: Este software de desarrollo está destinado exclusivamente a su uso en la plataforma GNU/Linux y otros sistemas Unix, no cuenta con un compilador propio por lo que requiere de su integración con GCC. Está dirigido al escritorio KDE aun cuando puede emplearse en otros entornos. (**KDE** es una comunidad internacional que desarrolla software libre. Produce un entorno de escritorio, multitud de aplicaciones e infraestructura de desarrollo para diversos sistemas operativos como GNU/Linux, Mac OS X, Windows, etc)

Anjuta: Esta opción es propia de los sistemas GNU/Linux y BSD, en este caso su propósito principal es desarrollar aplicaciones para el escritorio GNOME mediante las herramientas proporcionadas por GTK+.

COMPILADOR

El compilador en una parte fundamental para un lenguaje de programación estructurado. Como sabemos ya, los compiladores son programas o herramientas encargadas de compilar. Un compilador toma un texto (código fuente) escrito en un lenguaje de alto nivel (en nuestro caso C) y lo traduce a un lenguaje comprensible por las computadoras (código objeto).

Generalmente un compilador se divide en dos partes:

* Front End: parte que analiza el código fuente, comprueba su validez, genera el árbol de derivación y rellena los valores de la tabla de símbolos. Parte que suele ser independiente de la plataforma o sistema operativo para el que funcionará.
* Back End: parte en donde se genera el código máquina exclusivo para una plataforma a partir de lo analizado en el front end. (Ver figura 2.1)

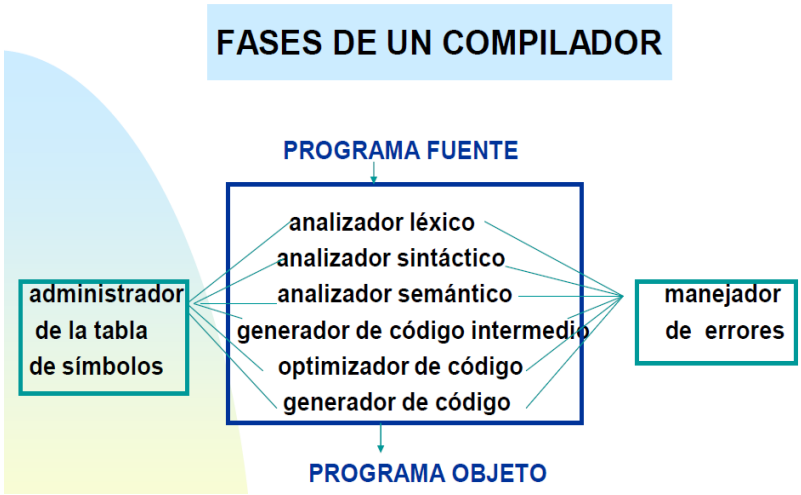


Fig. 2.1. Partes de un compilador

Por lo general el resultado del back end no puede ser ejecutado directamente, se necesita pasar por un proceso de enlazado (linker).

Existen varios tipos de compiladores: Compiladores cruzados, Compiladores optimizadores, Compiladores de una sola pasada, Compiladores de varias pasadas, Compiladores JIT (Just In Time).

A grandes rasgos el proceso se puede describir en los siguientes pasos:

* El compilador recibe el código fuente.
* Se analiza lexicográficamente.
* Se analiza sintácticamente y semántica (parseado).
* Se genera el código intermedio no optimizado.
* Se optimiza el código intermedio.
* Se genera el código objeto para una plataforma específica.

Finalmente ya puede ejecutarse el código máquina.

Para el IDE Dev-C, la compilación se realiza de la siguiente forma:

En la etapa de compilación el código fuente es comprobado sintácticamente por Dev-C++, y traducido a lenguaje de máquina (aún no ejecutable, y denominado código objeto). En caso de detectarse errores sintácticos o de concordancia de acuerdo con las reglas del lenguaje C, Dev-C++, avisa apropiadamente. Para compilar el código fuente que acabamos de teclear basta con acudir a la barra de menús y pulsar Ejecutar-> Compilar, o bien Ctrl+F9. Pulsando sólo F9 se consigue que se compile y, en ausencia de errores, también ejecutará el programa realizado.

Los errores son fallos críticos en la escritura o la concepción del programa que impiden al compilador realizar su tarea. Son por tanto errores que impiden proseguir con el resto de etapas de creación del programa, y deben ser reparados. Los errores más simples (ausencia de punto y coma al final de una sentencia, variables no declaradas, paréntesis no balanceados, etc.) pueden corregirse de forma sencilla inspeccionando el código del programa con la ayuda de los mensajes del compilador. Errores más sutiles o de difícil detección pueden requerir el empleo de la herramienta de depuración que se describirá posteriormente. Los avisos hacen referencia a incongruencias no críticas en el código (tipos de datos no concordantes, variables declaradas pero no utilizadas, etc). Los avisos no impiden proseguir con

el resto de etapas de creación del programa ejecutable, aunque resulta poco recomendable hacerlo. Una buena práctica de programación aconseja modificar el código fuente en lo necesario (generalmente cambios menores) para evitar la aparición de avisos en el proceso de compilación. Si detecta errores o avisos en el proceso de compilación de su programa, compruebe que no ha omitido ningún carácter crítico al teclearlo o ha cometido algún otro error al copiar el código fuente que se le suministra.

Montaje.
En la etapa de Montaje, el código objeto generado en la etapa de compilación es "ensamblado" junto con el código objeto de las funciones de librería para crear un único archivo ejecutable (con extensión .exe) Esta etapa no conlleva generalmente errores, salvo que hayamos omitido o confundido algunas de las librerías de funciones que emplee nuestro programa.

Si todo es correcto, debe obtener en el mismo directorio en el que ha guardado el código fuente de su programa, un archivo con ese mismo nombre, pero con extensión .exe, indicando que el archivo es ejecutable directamente por el computador.

Ejecución
La ejecución del programa es el paso final, y permite comprobar el funcionamiento del programa. Para ejecutar el programa, puede pulsar en la barra de menús Ejecutar -> Ejecutar (o bien Ctrl.+F10). Un método alternativo consiste en abrir una ventana de consola MS-DOS en el sistema operativo, y ejecutar el programa directamente en línea de comandos, tecleando el nombre del programa + ENTER. Para esto último debe asegurarse de que se encuentra en el directorio local donde ha compilado y generado el programa ejecutable. Si la ejecución del programa no es satisfactoria, bien porque no realiza las tareas para las que ha sido concebido, bien porque presenta errores en tiempo de ejecución, el código debe ser modificado. Cada vez que realice alguna modificación al programa, deberá compilar, montar y ejecutar de nuevo el programa. Dev-C++ tiene una opción para realizar estas tres tareas en un sólo paso pulsando en la barra de menús Ejecutar -> Compilar y Ejecutar. Resulta sin embargo conveniente que en un principio realice estas tareas por separado hasta adquirir una cierta práctica.

Depuración
Se entiende por depuración de un programa, aquellas tareas encaminadas a la localización y eliminación de errores ('bugs' en su denominación en inglés) de

cualquier naturaleza en el código de un programa. Dev-C++ proporciona varias herramientas para este propósito, entre las que cabe destacar:

1. Inspección de variables

   Esta herramienta permite mostrar el valor de cualquier variable del programa mientras éste se ejecuta (ver figura 2.2). Con ello se puede verificar si las variables toman el valor esperado en cada paso de ejecución y actuar en consecuencia.

   Para usar esta herramienta, en primer lugar hay que ejecutar el programa en modo de depuración. Una forma sencilla de hacerlo es emplear la herramienta ejecutar hasta el cursor. Para ello coloque el cursor con el ratón sobre cualquier línea del programa, por ejemplo

   float a,b,med;

   y pulse sobre la barra de menús depurar-> ejecutar hasta el cursor. (O bien pulsando Shift+F4).

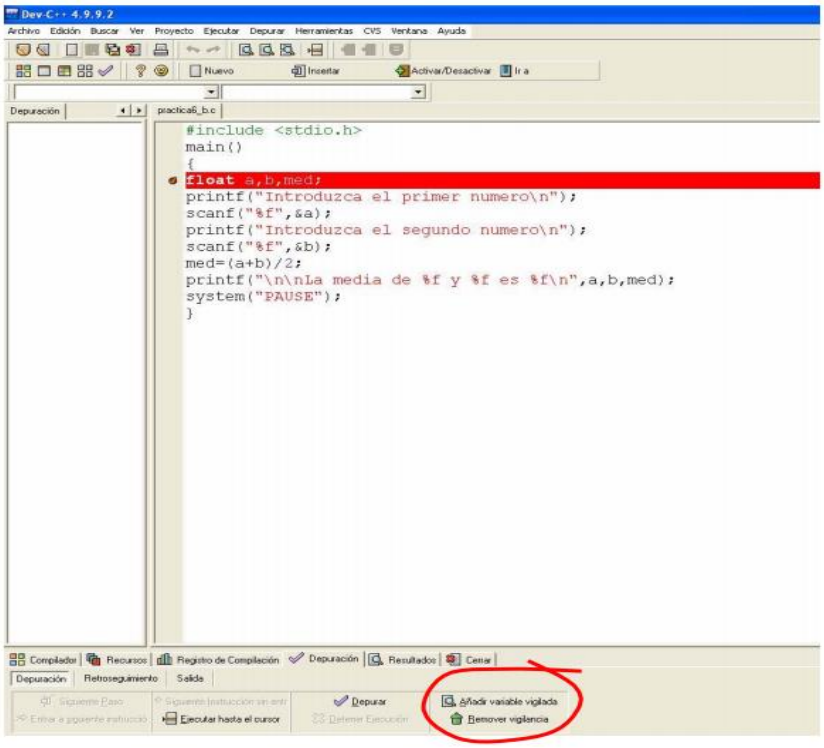


Fig. 2.2

Observará que se despliega una ventana de depuración en la parte inferior de la pantalla con diferentes opciones. Entre ellas, las opciones añadir y remover variable vigilada, permiten respectivamente ampliar y reducir la lista de variables que queremos inspeccionar. De este modo, pulse sobre Añadir variable vigilada, y ante la pregunta “nombre de la variable”, introduzca la variable med.

Puede repetir el procedimiento con otras variables (pruebe por ejemplo con las variables a y b. El valor de las variables inspeccionadas puede visualizarlo en la ventana desplegada en la parte izquierda de la pantalla, bajo la lengüeta depuración.

2. Ejecución paso a paso Ésta es otra útil herramienta que, como su nombre indica, permite ejecutar el programa instrucción a instrucción de modo que sea posible comprobar el comportamiento del mismo ante cada una de ellas. En conjunción con la herramienta de inspección de variables, permite obtener la traza del programa y verificar el comportamiento del mismo. La ejecución paso a paso se activa simplemente pulsando sobre la opción Siguiente Paso, que puede encontrar en la ventana de depuración de la parte inferior de la pantalla, en la barra de menús depurar, o bien pulsando Shift+F7. Podrá observar como cada vez que se avanza un paso, se ejecuta una sola instrucción del programa, resaltándose la línea en la que se encuentra actualmente el programa y, si se ha activado la inspección de variables, podrá observarse el valor de cada una de ellas en cada paso de ejecución.

UNIDAD DE COMPETENCIA III. VARIABLES Y TIPOS DE DATOS.

(3 horas)

RESUMEN

En esta unidad de competencia se explican los tipos de datos tanto primitivos como derivados, su sintaxis, su entorno donde están involucradas y uso memoria (si es estática o dinámica).

VARIABLES EN EL LENGUAJE C

(Aprenderaprogramar.com. Didactica y divulgación de la programación, 2018)
(Ayala de la Vega, Aguilar Juárez, Zarco Hidalgo, & Gómez Ayala, 2016)

Todo lo expuesto en relación a variables en el curso es válido y útil con el lenguaje C, pero con matices, ya que cada lenguaje sigue sus propias normas y pautas. Se van a tratar los aspectos más básicos de la declaración y uso de variables con C y a dar algunas orientaciones a través de las que profundizar en el manejo de variables.

El lenguaje C obliga a declarar una variable antes de ser usada. Es decir, no podríamos escribir algo del tipo: Mostrar velocidad01, si antes no hemos “dicho” (declarado) que existe una variable con ese nombre. Declarar una variable no significa que se le asigne contenido, sino simplemente se indica que la variable existe. ¿Qué nos mostrará si intentamos mostrar o utilizar la variable si no le hemos asignado contenido? Hay lenguajes que asignan automáticamente contenido cero, vacío o falso a una variable, dependiendo de qué tipo sea. Sin embargo en C no está permitido usar una variable sin antes haberle asignado un contenido, a lo que se denomina “inicializar la variable”. Por tanto, al tratar de ejecutar un programa donde una variable está sin inicializar puede producirse un error de compilación. Hay lenguajes donde la inicialización (o incluso la declaración) de las variables no es estrictamente obligatoria como PHP ó Visual Basic.

Nosotros vamos a trabajar con C por lo que habremos de declarar e inicializar las variables que usemos siempre. La declaración e inicialización de variables, aunque pueda resultar a veces “pesada”, tienen una serie de ventajas que podemos resumir en servir para:

- Generar buenos programas.

- Evitar errores y confusiones.

En cuanto a los tipos de variables, los más habituales en C los expondremos en la tabla 3.1.

Tabla 3.1

TIPO DE DATOS	SE ESCRIBE	MEMORIA REQUERIDA*	RANGO ORIENTATIVO*	EQUIVALENCIA EN PSEUDOCÓDIGO	OBSERVACIONES
Entero	int	2 bytes	- 32768 a 32767	Entero	Uso en contadores, control de bucles etc.
Entero largo	long	4 bytes	- 2147483648 a 2147483647	Entero	Igual que int pero admite un rango más amplio
Decimal simple	float	4 bytes	$-3{,}4\cdot10^{38}$ a $3{,}4\cdot10^{38}$	Real	Hasta 6 decimales. También admite enteros
Decimal doble	double	8 bytes	$-1{,}79\cdot10^{308}$ a $1{,}79\cdot10^{308}$	Real	Hasta 14 decimales. También admite enteros
Carácter	char	1 bytes	0 a 255	Alfanumérica	Carácter, independiente o parte de una cadena

Existen otros tipos de variables en C como las tipo short (enteros cortos), unsigned int (enteros positivos o naturales) y long double (decimal doble con una mayor precisión).

A diferencia de la mayoría de lenguajes, C no incorpora entre sus tipos predefinidos el tipo booleano (true / false). El motivo para ello es que en realidad no es estrictamente necesario para permitir la programación, ya que puede emularse fácilmente por ejemplo usando un entero al que asignamos valor uno ó valor cero según sea verdadera o falsa una condición. No obstante, dado que la mayoría de los programadores están habituados a usar este tipo, será posible usarlo gracias a que C nos permite definir nuestros propios tipos de datos.

Los tipos enumerados (enum) son variables especiales que tienen un nombre “general” y luego n valores posibles, por ejemplo la variable “color” podría tener

como valores posibles “rojo”, “verde” y “azul”. Hay más tipos de datos en C, pero nosotros no vamos a profundizar en su estudio.

Conociendo ya los tipos de variables básicas y cómo nombrarlas, veamos ahora cómo declararlas. La declaración de variables en C debe hacerse al principio de cada función. Veremos ahora una forma básica que nos permita empezar a trabajar y, de paso, crear nuestro primer programa.

Usaremos para ello el tipo de variable int (entero). La sintaxis que usaremos será:

int [Nombre de variable];

Para facilitar la corrección y claridad de nuestros programas el lugar donde realizaremos la declaración de variables, al menos de momento, será después de la línea **int main()**, que constituye el inicio del código ejecutivo de nuestro programa. Crea un nuevo proyecto, denomínalo proyectoCurso2 y accede al código. Si no recuerdas cómo hacer esto, lee las explicaciones que se han dado anteriormente. Una vez en la ventana de código, dentro del int main() {…} escribe:

int numeroDePlantas;

Has declarado la variable numeroDePlantas como tipo entero. También podrías asignarle un valor inicial a la variable en la misma línea que la declaras, de esta manera:

int numeroDePlantas=15;

Este tipo de escritura nos permite declarar e inicializar la variable con un valor en una sola línea. No siempre lo haremos así, pero en algunas ocasiones nos puede resultar de interés. En otras ocasiones declararemos la variable en una línea y posteriormente le asignaremos contenido en otra.

Supón que declaras: int edad;, como variable destinada a contener la edad de una persona. Sabemos que la edad de una persona puede oscilar entre cero y 150 años (siendo groseros), y que sus valores no son decimales. Por tanto puede declararse como tipo int sin ningún problema. ¿Qué ocurriría si la declaráramos como tipo double?

a) Vamos a ocuparle (estimamos) 8 bytes al sistema cuando podríamos haber ocupado sólo 2. Es una falta de eficiencia y economía.

b) A la hora de liberar de errores al programa (depurarlo) no sabremos a ciencia cierta qué tipo de datos contiene la variable edad, ya que puede contener tanto el

valor entero 56 como el valor decimal 56.332. Hacemos al programa más difícil de entender.

A la hora de declarar variables conviene seguir las pautas que ya hemos comentado y, resumidamente:

- Elegir siempre el tipo de variable más sencillo posible. Consideraremos que el grado de sencillez viene definido por la memoria requerida (a menor memoria requerida mayor es la sencillez). Esto redunda en un menor consumo de recursos del sistema y en una mayor rapidez de las operaciones que realiza el ordenador.

- Realizar declaraciones ordenadas y claras.

- Elegir un nombre descriptivo y claro, que comience preferiblemente con minúsculas.

En C se pueden declarar variables en múltiples líneas una por línea pero también varias variables en una línea. Existen distintas formas para realizar declaraciones. Veamos en la tabla 3.2 se ven varios ejemplos:

Tabla 3.2

Expresión abreviada	**Equivalente**
a) int i, j, k;	a´) int i; int j; int k;
b) int i, j, k;	b´) int i; int j; int k;

c) int i=0, j=4, k=76;	c´) int i=0;
	int j=4;
	int k=76;

d) int i=0, j=4, k=76;	d´) int i=0; int j=4; int k=76;

Las opciones a), b), c), d) dan lugar a un mismo resultado. Se declaran tres variables de tipo entero denominadas i, j, k. Las opciones a´- d´), escritas de forma extendida, también son válidas y con el mismo resultado.

Ten en cuenta que la declaración int i, j, k; declara tres variables de tipo entero.

Una expresión como float i; int j; long k; es válida, aunque consideramos preferible no declarar distintos tipos de variables en una misma línea sino hacerlo en líneas separadas.

Si en un programa tenemos únicamente una declaración de variables como el programa 3.1:

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

int main() {
   int i, j, k;
   return 0;
}
```

Programa 3.1

Si tratamos de ejecutar el programa (en CodeBlocks, en el menú “Build” elegimos la opción “Build and run”) obtenemos un mensaje de error que será similar a este:

C:\ProyectosCursoC\aprenderaprogramar.com\main.c|7|aviso: variable 'k' sin usar [-Wunused-variable]|

Esto es debido a que el compilador comprueba si las variables declaradas son usadas en el programa, y si no lo son, impide la ejecución y muestra un mensaje de error. No te preocupes por ello en este momento, ya que ahora únicamente se está tratando de ver cómo se realiza la declaración de variables.

Una peculiaridad de C respecto a otros lenguajes es que no tiene un tipo predefinido “String” o cadena de caracteres. Aunque sí dispone de distintas funciones relacionadas con cadenas, cuando se quiere declarar una variable como tipo alfanumérico se tiene que hacer declarando un array de caracteres. Es decir, en C las cadenas se declaran como vectores de tipo char. Por ejemplo la palabra “curso” estaría formada por cinco caracteres que son ‘c’, ‘u’, ‘r’, ‘s’, ‘o’. Veremos cómo trabajar con arrays y con variables alfanuméricas más adelante.

ARRAYS (ARREGLOS) UNIDIMENSIONALES

La sintaxis a emplear es la siguiente:

```
tipoDeElementosDelArray  nombreDelArray  [numeroElementos];
```

Ejemplo: int vectorEnteros [4];

Esto declara que se crea un vector de enteros que contendrá 4 valores de tipo int. Fíjate que el número en la declaración es 4, pero el elemento vectorEnteros[4] no existirá. ¿Por qué? Porque en C, al igual que en otros lenguajes de programación, la numeración de elementos empieza en cero y no en uno. De esta manera al indicar un 4, los índices de elementos en el array serán 0, 1, 2, 3. Es decir, si indicamos 4 el array tendrá 4 elementos (índices 0 a 3). Si indicamos 10 el array tendrá 10 elementos (índices 0 a 9) y así sucesivamente para cualquier número declarado.

Ejemplos de declaración de arrays serían:

int vectorVez [9]; char vectorAmigo [1000]; double decimalNum [24];

int vectorInt [23]]; int vectorLong[8]; int personasPorHabitacion[300];

Crea un proyecto y escribe el código del programa 3.2:

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

int main() {
int numeroDeCoches [4];
numeroDeCoches[0] = 32;
printf ("El numero de coches en la hora cero fue %d \n", numeroDeCoches[0]);
printf ("El numero de coches en la hora uno fue %d \n", numeroDeCoches[1]);
printf ("El numero de coches en la hora dos fue %d \n", numeroDeCoches[2]);
printf ("El numero de coches en la hora tres fue %d \n", numeroDeCoches[3]);
return 0;
}
```

El resultado de ejecución puede ser similar a este:

```
El numero de coches en la hora cero fue 32
El numero de coches en la hora uno fue 8
El numero de coches en la hora dos fue 100
El numero de coches en la hora tres fue 100
```

Programa 3,2

¿Por qué ocurre esto? Realmente no siempre ocurrirá esto, podríamos decir que el resultado puede cambiar dependiendo del compilador. Lo que es cierto que es numeroDeCoches[0] vale 32 porque así lo hemos declarado. Sin embargo, es posible que nos aparezcan valores aparentemente aleatorios para el resto de elementos del array porque no los hemos inicializado. El compilador al no tener definido un valor específico para la inicialización le ha asignado valores aparentemente aleatorios. Para evitar esta circunstancia se tienen que **inicializar todos los elementos** de un array. De momento se hará manualmente como se indica a continuación, más adelante veremos cómo hacerlo usando un bucle que permitirá inicializar los elementos de un array de decenas o cientos de elementos a un valor por defecto. En el caso de enteros lo más normal es inicializar los elementos que no tienen un valor definido a cero.

En el programa anterior se añade la siguiente línea:

numeroDeCoches[1]=0; numeroDeCoches[2]=0; numeroDeCoches[3]=0;

Con esto quedan inicializados todos los elementos del array y al ejecutar el programa obtenemos un resultado “seguro”.

Es posible que en un momento dado se necesite borrar el contenido de los elementos de un array, digamos que resetear o borrar su contenido. Para ello en algunos lenguajes existen instrucciones específicas, pero en C se tiene que volver a asignar los valores por defecto a los elementos del array realizando lo que se podría denominar una “reinicialización”.

¿Cómo elegir los nombres de los arrays? Los nombres de variables deben ser lo más descriptivos posibles para hacer el programa fácil de leer y de entender. Piensa que es válido tanto declarar int vectorInt; como int VI;. Sin embargo es más correcto usar vectorInt que VI porque resulta más descriptivo de la función y cometido de la variable vectorInt que dos letras cuyo significado no entenderá una persona que lea el programa (y quizás tú mismo no entenderás cuando hayan pasado unos días después de haber escrito el programa).

Nota: C no realiza una comprobación de índices de arrays. Por ejemplo si se ha declarado int numeroCoches[4] y se incluye en el código el uso de numeroCoches[5] el comportamiento es imprevisible. Es responsabilidad del programador el controlar y hacer uso exclusivamente de los índices válidos para cada array.

DECLARACIÓN DE TIPOS CON TYPEDEF EN C

C permite que el programador defina sus propios tipos de datos mediante la palabra clave typedef. Esta palabra puede tener distintos usos. Vamos a ver ahora uno de ellos.

La sintaxis a emplear requiere cumplir con dos pasos.

El primero, declarar un tipo de datos que es el tipo del array y que nos permitirá crear cuantas variables queramos de ese tipo, es decir, cuantos arrays queramos conteniendo un mismo tipo de dato y un mismo número de elementos en el array. La sintaxis será:

typedef tipoDeElementosDelArray nombreDelTipoArray [numeroElementos];

Ejemplo: typedef int TipoVectorEnteros [4];

declara que se crea un nuevo tipo de datos, definido por nosotros, denominado TipoVectorEnteros, que contendrá 4 valores de tipo int.

El segundo paso para hacer uso del tipo definido implica que hemos de crear una variable del tipo definido por nosotros en la expresión anterior. Al igual que para declarar un entero escribimos int nombreVariable;, para declarar un array del tipo creado por nosotros escribiremos nombreDelTipoDefinido nombreVariable;

Para el ejemplo que hemos puesto, escribiríamos lo siguiente: TipoVectorEnteros vector1; donde vector1 es el nombre que le damos al vector que hemos creado, que contendrá cuatro valores enteros (obviamente estos valores podemos modificarlos).

Ejemplos de declaración de tipos arrays serían:

- typedef int TipoVectorVez [9];
- typedef char VectorAmigo [1000];
- typedef double DecimalNum [24];
- typedef int VectorInt [23];
- typedef int TipoVectorLong[8];
- typedef char TipoPalabra[255];

Tras declarar el tipo tendríamos que declarar variables de ese tipo, por ejemplo: TipoVectorVez vez;, VectorAmigo amigo;, DecimalNum numDecimal;, VectorInt

numeroDeCoches; ó TipoVectorLong jugador;. Podemos definir múltiples arrays del mismo tipo, por ejemplo:

```
VectorInt numeroDeCoches;
VectorInt numeroDePersonas;
VectorInt numeroDeRuedas;
VectorInt pruebasRealizadas;
```

¿Cómo elegir los nombres de los tipos y los nombres de las variables? No existen reglas precisas al respecto, pero se recomienda para los nombres de tipos usar siempre un prefijo que comience con mayúsculas que podría ser Tipo y para las variables un nombre que comience por minúsculas. Es un convenio que siguen muchos programadores, aunque no es obligatorio. Los nombres de tipos y variables deben ser lo más descriptivos posibles para hacer el programa fácil de leer y de entender. Piensa que es válido tanto declarar typedef int TipoVectorInt; como typedef int TVI; Sin embargo es más correcto usar TipoVectorInt que TVI porque resulta más descriptivo de la función y cometido del tipo VectorInt que tres letras cuyo significado es poco entendible.

Crea un proyecto y escribe el programa 3.3:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
typedef int TipoVectorEnteros [4];
TipoVectorEnteros numeroDeCoches;

numeroDeCoches[0] = 32;
numeroDeCoches[1]=0; numeroDeCoches[2]=0; numeroDeCoches[3]=0;
printf ("El numero de coches en la hora cero fue %d \n",
numeroDeCoches[0]);
printf ("El numero de coches en la hora uno fue %d \n", numeroDeCoches[1]);
printf ("El numero de coches en la hora dos fue %d \n", numeroDeCoches[2]);
printf ("El numero de coches en la hora tres fue %d \n", numeroDeCoches[3]);
return 0;
}
```

Programa 3.3

El resultado de ejecución será el mismo que vimos anteriormente.

La definición de tipos con typedef es una posibilidad que brinda el lenguaje C pero que no está disponible en todos los lenguajes.

En estos momentos es normal que puedas confundir nombres de tipos con nombres de variables. Ten en cuenta que se trata de cosas distintas: un tipo es “un molde” con el que podemos crear tantas variables de ese tipo como deseemos. A medida que practiques con estos conceptos te resultará más fácil trabajar con ellos.

CADENAS DE TEXTO COMO ARRAYS DE CARACTERES EN C

Si bien en otros lenguajes las cadenas de texto son consideradas un tipo de datos específico, en C las cadenas de texto son consideradas arrays de caracteres. Por ejemplo la cadena “Hola” en C se considera un array formado por los elementos 'H', 'o', 'l', 'a'.

Podríamos mostrar cadenas de texto por pantalla mostrando los diferentes elementos del array.

Veamos un ejemplo. Escribe y ejecuta los programas 3.4:

```
#include <stdio.h>
#include <stdlib.h>
//Ejemplo aprenderaprogramar.com
int main() {
char cadena1[4];
cadena1[0]='h'; cadena1[1]='o'; cadena1[2]='l'; cadena1[3]='a';
printf("La palabra en la variable cadena1 es: %c%c%c%c \n",
cadena1[0],cadena1[1],cadena1[2],cadena1[3]);
return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
//Ejemplo aprenderaprogramar.com
int main() {
typedef char TipoCadena[50];
TipoCadena nombre1;
TipoCadena nombre2;
nombre1[0]='S'; nombre1[1]='a'; nombre1[2]='m';
nombre2[0]='J'; nombre2[1]='o'; nombre2[2]='e';nombre2[3]='l';
```

```
printf("El nombre 1 es %c%c%c \n", nombre1[0], nombre1[1], nombre1[2]);
printf("El nombre 2 es %c%c%c%c \n", nombre2[0], nombre2[1], nombre2[2],
nombre2[3]);
return 0;
}
```

Prpogramas 3.4

Fíjate en la diferencia entre ambos. En el primer caso estamos usando un array del tipo predefinido del lenguaje C char, y ese array tiene exactamente cuatro elementos (0, 1, 2 y 3 cuyo contenido es h, o, l, a). En el segundo caso hemos definido un tipo denominado TipoCadena que consta de 50 elementos (0, 1, 2, 3, … , 49) y hemos definido dos variables del tipo TipoCadena. Para una de esas variables hemos definido los tres primeros elementos, mientras que los 47 elementos restantes permanecen sin inicializar. Para la otra variable hemos definido los cuatro primeros elementos, mientras que los 46 elementos restantes permanecen sin inicializar. Explicaremos más adelante el significado de los símbolos %c y \n, no te preocupes ahora por ellos.
Esta forma de trabajar con cadenas “carácter a carácter” puede parecer en principio un poco tediosa o poco práctica, de hecho hay una forma simplificada para inicializar una cadena: char cadena1[]="hola"; equivale al código anterior. C dispone de algunos extras que facilitan el trabajo con cadenas y nos permiten no tener que ir “carácter a carácter”, aunque siempre debemos tener presente que en el fondo una cadena de texto es tratada por C como un array de caracteres.

ARRAYS (ARREGLOS) MULTIDIMENSIONALES

Tal y como explicamos en su momento, será posible crear arrays con más de una dimensión, pasando de la idea de lista, vector o matriz de una sola fila a la idea de matriz de m x n elementos, estructuras tridimensionales, tetradimensionales, etc. La sintaxis será:

```
TipoDeVariable nombreDelArray [dimensión1] [dimensión2] [...]
[dimensiónN]
```

TipoDeVariable puede ser uno de los tipos predefinidos de C o bien ser un tipo definido por el programador mediante el uso de la palabra reservada typedef. La declaración de una matriz tradicional de m x n elementos podría ser:

```
int A [3] [2]
```

El número de elementos declarados (recordar que los índices de arrays comienzan en cero) será de 3 x 2 = 6, correspondiente a (2+1) x (1+1) = 3 x 2.

Vamos a definir una matriz con el mismo ejemplo que usamos cuando hablamos de pseudocódigo: queremos almacenar en una matriz el número de alumnos con que cuenta una academia ordenados en función del nivel y del idioma que se estudia. Tendremos 3 filas que representarán Nivel básico, medio o de perfeccionamiento y 4 columnas que representarán los idiomas (0 = Inglés, 1 = Francés, 2 = Alemán y 3 = Ruso). La declaración de dicha matriz sería:

```
int alumnosfxniveleidioma [3] [4];
```

Podríamos asignar contenidos de la siguiente manera:

```
alumnosfxniveleidioma[0] [0] = 7;    alumnosfxniveleidioma[0] [1] = 14;
alumnosfxniveleidioma[0] [2] = 8;    alumnosfxniveleidioma[0] [3] = 3;
alumnosfxniveleidioma[1] [0] = 6;    alumnosfxniveleidioma[1] [1] = 19;
alumnosfxniveleidioma[1] [2] = 7;    alumnosfxniveleidioma[1] [3] = 2;
alumnosfxniveleidioma[2] [0] = 3;    alumnosfxniveleidioma[2] [1] = 13;
alumnosfxniveleidioma[2] [2] = 4;    alumnosfxniveleidioma[2] [3] = 1;
```

La representación gráfica que podríamos asociar a esta asignación de datos sería esta matriz:

$$\begin{pmatrix} 7 & 14 & 8 & 3 \\ 6 & 19 & 7 & 2 \\ 3 & 13 & 4 & 1 \end{pmatrix}$$

La organización de la información en matrices nos generará importantes ventajas a la hora del tratamiento de datos en nuestros programas.

Para terminar en cuanto a multidimensionalidad, veamos casos de declaraciones con más de dos dimensiones. Para ello volveremos al ejemplo del conteo de coches (suponemos que queremos registrar el número de coches que han pasado por hora). La forma de declarar esos array sería la siguiente:

Tabla 3.3

Duración del conteo	Tipo de array	Declaración con C (Nc es Número de coches)
Un día	Array de un localizador (hora)	int Nc[24];
Varios días	Array de dos localizadores (hora y día)	int Nc[24] [31];
Varios meses	Array de tres localizadores (hora, día y mes)	int Nc[24] [31] [12];
Varios años	Array de cuatro localizadores (hora, día, mes y año)	int Nc[24] [31] [12] [2999];
Varios siglos	Array de cinco localizadores (hora, día, mes, año y siglo)	int Nc[24] [31] [12] [2999] [21];

Veamos lo que sería un ejemplo de programa con array multidimensional. Escribe y ejecuta el programa 3.5.

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

int main() {
int habitantesVivienda[5][25];
habitantesVivienda[3][24] = 4;
printf("El numero de personas que viven en la vivienda 24 del piso 3 es %d\n",
habitantesVivienda[3][24]);
return 0; //Ejemplo aprenderaprogramar.com
}
```

Programa 3.5

El resultado del programa es que se muestra el mensaje “El número de personas que viven en la vivienda 24 del piso 3 es 4”. Fíjate que en un array multidimensional cada índice tiene un significado.

MANEJO DE ARREGLOS CON MEMORIA DINÁMICA

(Backman, 2012)
(Ceballos, 2015)
(Deitel & M, 1995)

El lenguaje de programación C permite crear localidades de memoria en forma dinámica, es decir, se puede solicitar la asignación de memoria en el momento de ejecución en vez del momento de compilación. Para lograr esto, el lenguaje de programación C tiene las siguientes herramientas:

stdlib.h (*std-lib*: ***st**an**d**ard **lib**rary* o biblioteca estándar) es el archivo de cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras. Es compatible con C++ donde se conoce como cstdlib.
alloc.h (*allocate library* o biblioteca para asignación dinámica de memoria) es el archivo de cabecera para C que nos da la capacidad de asignar y liberar memoria de forma dinámica. No todos los compiladores tienen esta biblioteca, por lo que se debe de verificar si el compilador a utilizar contiene tal herramienta.
Ambas bibliotecas tienen varias funciones en común, las que se utilizarán en este capítulo son (ver tabla 3.4):

Tabla 3.4 (Plauger, 1992)

Gestión de memoría dinámica	
malloc	void * malloc(size_t size) La función malloc reserva espacio en el heap a un objeto cuyo tamaño es especificado en bytes (size). El heap es un área de memoria que se utiliza para guardar estructuras dinámicas en tiempo de ejecución. Si se puede realizar la asignación de memoria, se retorna el valor de la dirección de inicio del bloque. De lo contrario, si no existe el espacio requerido para el nuevo bloque, o si el tamaño es cero, retorna NULL.

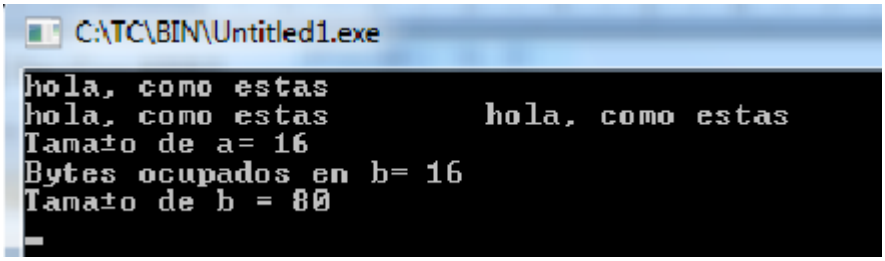
calloc	void *calloc(size_t nitems, size_t size)
	La función calloc asigna espacio en el heap de un número de elementos (nitems) de un tamaño determinado (size). El bloque asignado se limpia con ceros. Si se desea asignar un bloque mayor a 64kb, se debe utilizar la función farcalloc.
	Ejemplo:
	char *s=NULL;
	str=(char*) calloc(10,sizeof(char));
	
	Si se puede realizar la asignación, se retorna el valor del inicio del block asignado. De lo contrario, o no existe el espacio requerido para el nuevo bloque, o el tamaño es cero, retorna NULL.

realloc	void *realloc**(***punt***,** *nuevotama***):** Cambia el tamaño del bloque apuntado

	por *punt.* El nuevo tamaño puede ser mayor o menor y no se pierde la información que hubiera almacenada (si cambia de ubicación se copia).
free	void free(void *dir_memoria) La función free libera un bloque de la memoria del heap. El argumento dir_memoria apunta a un bloque de memoria previamente asignado a través de una llamada a calloc, malloc o realloc. Después de la llamada, el bloque liberado estará disponible para asignación.

El programa 3.6 muestra el uso de la función –malloc()-:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main(){
     char *a, b[80];
     gets(b);
     a=(char *)malloc(strlen(b));
     strcpy(a,b);
     printf("%s\t%s\n",a,b);
     printf("Tamaño de a= %d\n", strlen(a));
     printf("Bytes ocupados en b= %d\n",strlen(b));
     printf("Tamaño de b = %d\n",sizeof(b));
     getchar();
}
```



Programa 3.6

En este programa se pueden escribir hasta 80 caracteres, incluyendo el null, sin invadir espacio de otras variables. La función –strlen()- pertenece al archivo de cabecera –string.h- y calcula el tamaño de una cadena sin incluir el carácter NULL. La función –sizeof()- no requiere algún archivo de cabecera y retorna el número de bytes ocupados en la memoria por una variable. Cuando se ejecuta la siguiente línea:

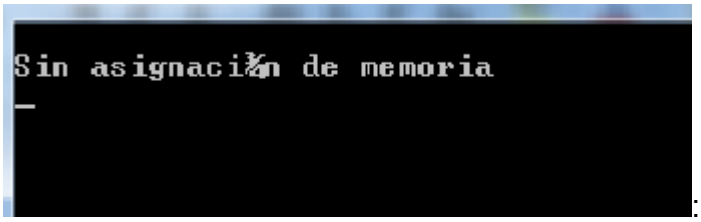
a=(char *)malloc(strlen(b));

la mayor prioridad la tiene la función -strlen()-, por lo que retorna el número de bytes ocupados en la variable –**b**-, observe que no se utilizó la función –sizeof()- ya que esta función retorna el tamaño de bytes reservados en la variable –**b**- (en este caso sería 80). Una vez determinado el tamaño, la función –malloc()- reserva el espacio en bytes en el heap, posteriormente se realiza el casting (se crea el espacio de memoria a lo indicado dentro del paréntesis, en este caso, se devuelve una apuntador a carácter al inicio de la cadena). El casting es necesario ya que la función –malloc()- retorna un apuntador tipo void (un apuntador sin tipo). Cuando es un apuntador tipo void, se puede tener la dirección de cualquier tipo de variable pero sin tener el permiso de acceso a ella para escribir o para leer. Si se requiere tener el acceso para leer o escribir, se requiere moldear al tipo de variable al que se quiere tener acceso (o casting).

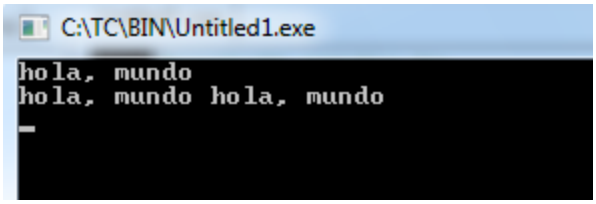
Por lo tanto, al utilizar una variable tipo apuntador y crear memoria en forma dinámica nos permite utilizar sólo la memoria necesaria y no se tiene desperdicio de memoria como se observa con un arreglo tipo cadena de caracteres.

En el programa 3.7 se muestra un ejemplo de la forma de detectar el retorno de NULL en la función malloc(). Si al ejecutar el programa, el ejecutor del programa oprime la tecla return (enter), se mostrará en monitor “Sin asignación de memoria”. De otra forma se mostrará dos veces lo escrito por el ejecutor del programa:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main(){
char *a, b[80];
gets(b);
if ((a=(char *)malloc(strlen(b)))==NULL){
    printf("Sin asignación de memoria\n");
    getchar();
    exit(0);
}
strcpy(a,b);
printf("%s %s\n",a,b);
getchar();
}
```



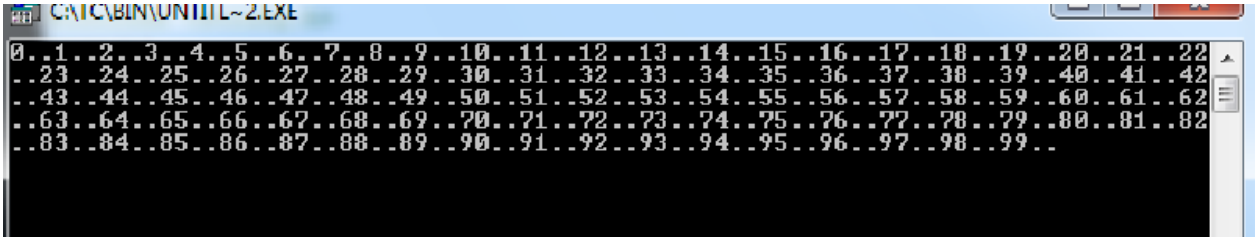
:



Programa 3.7

Uno puede tener un apuntador que controle a un gran bloque de memoria para algún propósito en específico. El programa 3.8 declara un apuntador entero y utiliza la función malloc para retornar un gran bloque de enteros:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main(){
int *x, i;
if ((x=(int *)malloc(200))==NULL){
    printf("No se asignó la memoria solicitada\n");
    getchar();
    exit(0);
}
for (i=0;i<100;i++)
    *(x+i)=i
    ;
for (i=0;i<100;i++)
    printf("%d..",*(x+i));
getchar();
free(x);
}
```



Programa 3.8

Observe algo interesante, malloc(200) asignará memoria para 200 bytes y, al momento de realizar el casting, serán tratados como enteros. (Recuerde que Turbo C, un entero se almacena en dos bytes, por lo que sólo se permite guardar en zona segura a 100 números enteros). En consecuencia, los apuntadores -***x**-, -***(x+1)**-, -***(x+2)**- lograron tener acceso a sus respectivas unidades de almacenamiento, donde cada unidad de almacenamiento contiene dos bytes. (Cada tipo de variable tendrá su propia unidad de almacenamiento. Por ejemplo, en TC, las variables tipo char tienen un byte como unidad de almacenamiento y la variable double tiene como unidad de almacenamiento a 8 bytes, etc. El tamaño de bytes de cada tipo de variable depende del tipo de máquina y del compilador)

En el programa anterior, el manejo del bloque de bytes se puede realizar sin problema con los subíndices, esto es:

*x ≡ x[0]

*(x+1) ≡ x[1]

Para saber la dirección de memoria de un arreglo se puede utilizar:

x ≡ &x[0]

x + 1 ≡ &x[1]

El programa 3.9 muestra la asignación de memoria en forma dinámica:

```
#include <stdio.h>
#include <stdlib.h>
main(){
int *a,i;
if ((a=(int *)malloc(40))==NULL){
    printf("Memoria no asignada\n");
    getchar();
    exit(0);
}
for (i=0;i<10;i++)
    a[i]=100+i;
for (i=10; i<20; i++)
    *(a+i)=100+2*i;
printf("La variable -a-  apunta a la localidad de memoria %u %u\n",a, &a[0]);
for (i=0;i<20;i++)
          printf("a[%d]=%d\t*(a+%d)=%d\tLoc %u\tLoc %u\n", i, a[i],i,*(a+i),&a[i], a+i);
getchar();
}
```

```
La variable -a-  apunta a la localidad de memoria 8523744 8523744
a[0]=100        *(a+0)=100      Loc 8523744     Loc 8523744
a[1]=101        *(a+1)=101      Loc 8523748     Loc 8523748
a[2]=102        *(a+2)=102      Loc 8523752     Loc 8523752
a[3]=103        *(a+3)=103      Loc 8523756     Loc 8523756
a[4]=104        *(a+4)=104      Loc 8523760     Loc 8523760
a[5]=105        *(a+5)=105      Loc 8523764     Loc 8523764
a[6]=106        *(a+6)=106      Loc 8523768     Loc 8523768
a[7]=107        *(a+7)=107      Loc 8523772     Loc 8523772
a[8]=108        *(a+8)=108      Loc 8523776     Loc 8523776
a[9]=109        *(a+9)=109      Loc 8523780     Loc 8523780
a[10]=120       *(a+10)=120     Loc 8523784     Loc 8523784
a[11]=122       *(a+11)=122     Loc 8523788     Loc 8523788
a[12]=124       *(a+12)=124     Loc 8523792     Loc 8523792
a[13]=126       *(a+13)=126     Loc 8523796     Loc 8523796
a[14]=128       *(a+14)=128     Loc 8523800     Loc 8523800
a[15]=130       *(a+15)=130     Loc 8523804     Loc 8523804
a[16]=132       *(a+16)=132     Loc 8523808     Loc 8523808
a[17]=134       *(a+17)=134     Loc 8523812     Loc 8523812
a[18]=136       *(a+18)=136     Loc 8523816     Loc 8523816
a[19]=138       *(a+19)=138     Loc 8523820     Loc 8523820
```

Programa 3.9

La función -calloc()-, a comparación de la función -malloc()-, limpia la memoria. Esto significa que todo block asignado de memoria será inicializado en cero. El programa 3.10 es similar a un ejemplo anterior en este capítulo y va a permitir hacer la comparación entre las dos funciones.

#include <stdio.h>

#include <stdlib.h>

```
main(){
    int *x,y;
    if((x=(int *)calloc(400,2))==NULL){
                        printf("No asignación de memoria");
                        exit(0);
    }
    for (y=0; y<400; y++)
       *(x+y)=88;
}
```

Programa 3.10

Cuando se invocó a -malloc(400)-, se asignó una dirección con un block de 400 bytes a un apuntador tipo int. Sabiendo que un entero necesita 2 bytes, se tiene memoria para 200 números enteros. En el programa anterior -calloc()- se emplea de la siguiente forma:

calloc(unidades requeridas, tamaño por unidad)

Al invocarse -calloc(400,2)- indica que se requieren 400 unidades de dos bytes cada una, esto nos da un block de almacenamiento de 800 bytes.

En este caso es importante si se requiere portabilidad de uno a otro tipo de máquina. Como C no es completamente estándar, como lo es Java, existe diferencia en el número de bytes por tipo de variable. Por ejemplo, para una microcomputadora una variable tipo entero se pueden requerir 2 bytes, pero para una mainframe se pueden requerir 4 bytes. La asignación de memoria para todo tipo de dato en C es relativa y se basa más en la arquitectura de la máquina y otros factores.

Una forma simple de verificar el tamaño de bytes requerido para un tipo de variable en un compilador para una arquitectura especifica es utilizando la función -sizeof(). Si se espera portar el programa a otra arquitectura se puede realizar lo siguiente:

calloc(400, sizeof(int));

De esta forma, calloc() permite la portabilidad sin ningún problema, o puede utilizarse malloc() de la siguiente forma:

malloc(400*sizeof(int))

En este caso, la única diferencia es que –malloc()- no inicializa el bloque asignado de memoria.

Para liberar un bloque de memoria creado por -malloc()- o -calloc()- se puede usar la función free() de la siguiente forma:

free(ptr).

Donde –**ptr**- es el apuntador al inicio del block de memoria asignado.

La función -realloc(**ptr**, **tamaño**)- cambia el tamaño del objeto apuntado por –**ptr-** al tamaño especificado por –**tamaño-**. El contenido del objeto no cambiará hasta que se defina el nuevo tamaño del objeto. Si el nuevo tamaño es mayor, el valor de la porción nueva se adjudicará al objeto. Si **–ptr-** es un puntero nulo, la función –realloc()- se comporta igual que la función –malloc()-. De lo contrario, si –**ptr-** no es igual a un puntero previamente retornado por la función –malloc()-, -calloc()- o –realloc(), o si el espacio ha sido desadjudicado por una llamada a la función –free()- o- realloc(), el comportamiento no estará definido.
Por ejemplo, el programa 3.11 lee y escribe los valores de un arreglo –**V-** de reales. El número de valores se conoce durante la ejecución. La memoria se asignará en el momento de ejecución elemento por elemento.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h> /* para getch() */
main() {
float *V=NULL; int N=0,i; char c;
do {
   V=( float *)realloc((float *)V,(N+1)*sizeof(float));
   printf("\nDame valor >>"); scanf("%f",&V[N]);
   printf("Quieres introducir otro valor? (S/N) >> ");
   c=getch();
   N++;
} while (c=='s' || c=='S');
for(i=0;i<N;i++) printf("\nValor %d >> %f\n",i,V[i]);
free(V);
}
```

Programa 3.11

MANEJO DE MATRICES EN FORMA DINÁMICA

El lenguaje de programación C proporciona la posibilidad de manejar matrices en forma estática por lo que se debe conocer el tamaño de la matriz en tiempo de compilación. En caso de matrices de dos dimensiones, el compilador debe conocer el número de filas y columnas. Si el tamaño de la matriz se conoce hasta el tiempo de ejecución el lenguaje de programación C nos permite manipular matrices en forma dinámica.
En particular, se posibilitará:

- Crear matrices en forma dinámica (en el momento de ejecución)
- Destruir matrices en forma dinámica.
- Acceso mediante índices o apuntadores.

Para poder trabajar la matriz en forma dinámica se requiere una variable del tipo doble apuntador. El primer apuntador hará referencia al inicio de un arreglo de

apuntadores y cada apuntador del arreglo de apuntadores tendrá la referencia del inicio de un arreglo de enteros u otro tipo de variable, ver figura 3.1:

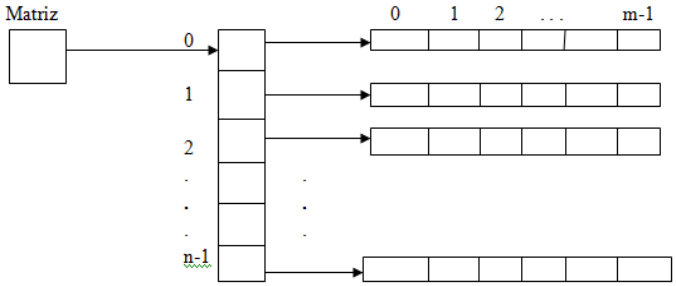


Figura 3.1

El programa 3.12 muestra la forma que debe ser manipulado cada uno de los apuntadores para la creación dinámica de la matriz bidimensional:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int f,c,i,j;
      int **pm;
      printf("Da el numero de hileras=>");
      scanf("%d",&f);
      getchar();
      printf("Da el numero de columnas=>");
      scanf("%d",&c);
      pm=(int **)malloc(sizeof(int *)*f);
      for (j=0;j<c;j++)
            pm[j]=(int*)malloc(sizeof(int)*c);
      for (i=0;i<f;i++)
               for (j=0;j<c;j++)
                  pm[i][j]=i*j+1;
       printf("Mostrando la matriz utilizando corchetes\n");
      for (i=0;i<f;i++){
               for (j=0;j<c;j++)
                  printf("%d\t",pm[i][j]);
          putchar('\n');
      }
      printf("Mostrando la matriz utilizando apuntadores\n");
      for (i=0;i<f;i++){
               for (j=0;j<c;j++)
                  printf("%d\t",*(*(pm +i)+j));
          putchar('\n');
      }
```

```
    getchar();
    getchar();
}
```

Programa 3.12

La siguiente línea del código anterior crea el arreglo de apuntadores:

```
pm=(int **)malloc(sizeof(int *)*f);
```

Donde la función -sizeof ()- entrega el número de bytes requerido para un apuntador tipo entero, al multiplicarse por el número de hileras, lo que se tiene es el tamaño real de arreglo de apuntadores para que la función -malloc()- realice el apartado de memoria con la magnitud indicada. Al final se realiza un casting para poderse entregar la dirección inicial del arreglo de apuntadores a la variable del tipo doble apuntador a entero.

Para poderse crear cada uno de los arreglos de enteros, se requerirá una instrucción cíclica como es el “for” ya que se le buscará espacio en forma independiente a cada arreglo de enteros. La función -sizeof()- retornará el tamaño requerido por cada entero, al multiplicarse por el número de columnas se tendrá el tamaño total del arreglo de enteros. La función -malloc()- buscará espacio en el HEAP para el tamaño solicitado por la función -sizeof()-, al final se realiza un casting tipo apuntador entero. El apuntador que hace referencia al inicio del arreglo se entrega a la variable tipo apuntador a entero -pm[j]-.

```
    for (j=0;j<c;j++)
        pm[j]=(int*)malloc(sizeof(int)*c);
```

MANEJO DE ARREGLOS DE REGISTROS EN FORMA DINÁMICA.

El programa 3.13 muestra una forma de crear un arreglo de registros en forma dinámica:

```
#include <stdio.h>
#include <stdlib.h>
struct alumno{
        char a[20];
        char b[30];
        int edad;
};
main(){
alumno *b;
```

```
int max,i;
printf(“Da el numero de registros a manejar=>”);
scanf(“%d”,&max);
b=(alumno *)malloc(sizeof(alumno)*max);
for (i=0;i<max;i++){
        printf(“Da el nombre del elemento %d=>”,i);
        scanf(“%s”,(b+i)->a);
        printf(“Da la irección del elemento %d=>”,i);
        scanf(“%s”,(*(b+i)).b);
        printf(“Da la edad del elemento %d=>”,i);
        scanf(“%d”,&(b+i)->edad);
}
for (i=0;i<max;i++){
        printf(“%s\t%s\t%d\n”,(b+i)->a,(b+i)->b,(b+i)->edad);
        printf(“%s\t%s\t%d\n”,(*(b+i)).a,(*(b+i)).b,(*(b+i)).edad);
}
getchar();
getchar();
}
```

Programa 3.13

En este programa, la siguiente línea crea el arreglo de registros:

```
b=(alumno *)malloc(sizeof(alumno)*max);
```

donde la función -sizeof()- entrega el número de bytes requerido para un registro tipo alumno, al multiplicarse por el número de registros requerido (max), lo que se tiene es el tamaño real de arreglo de registros para que la función- malloc()- realice el apartado de memoria con la magnitud indicada. Al final se realiza un casting para poderse entregar la dirección inicial del arreglo de registros a la variable del tipo apuntador de registros.

El programa muestra cómo realizar la lectura de cadenas de caracteres empleando los dos operadores especiales “**->**” o “**.**”. Para poder hacerse la lectura de la edad, siendo ésta del tipo entero, se requiere forzosamente el operador “&”.En la lectura de las variables tipo arreglo de carácter no se requiere el operador “&” ya que el nombre de la variable es el apuntador al primer elemento de la cadena.

Nota: El operador –new()- se puede emplear en lugar del operador -malloc()- y el operador -delete()- se puede emplear en lugar del operador -free()-. Si se decide emplear el operador –new()-, para liberar memoria se tiene que utilizar

forzosamente el operador -delete()-. Si se utiliza el operador –malloc()-, para liberar memoria se tiene que utilizar forzosamente el operador –free()-. **No es recomendable mezclar operadores.**

ENUMERACIONES

(Universidad de Castilla-La Mancha, 2018)

Un tipo de datos enumerado es una manera de asociar nombres a números, y por consiguiente de ofrecer más significado a alguien que lea el código. La palabra reservada **enum** (de C) enumera automáticamente cualquier lista de identificadores que se le pase, asignándoles valores de 0, 1, 2, etc. Se pueden declarar variables **enum** (que se representan siempre como valores enteros). La declaración de un **enum** se parece a la declaración de un **struct**.

Un tipo de datos enumerado es útil cuando se quiere poder seguir la pista de alguna característica (ver programa 3.14):

```
enum ShapeType {
  circle,
  square,
  rectangle
};  // Debe terminar con punto y coma, como el tipo “struct”

int main() {
  ShapeType shape = circle;
  switch(shape) {
          case circle:  /* circle stuff */ break;
          case square:  /* square stuff */ break;
          case rectangle:  /* rectangle stuff */ break;
  }
}
```

Programa 3.14

shape es una variable del tipo de datos enumerado **ShapeType**, y su valor se compara con el valor en la enumeración. Ya que **shape** es realmente un **int**, puede albergar cualquier valor que corresponda a **int** (incluyendo un número negativo). También se puede comparar una variable **int** con un valor de una enumeración.

Se ha de tener en cuenta que el ejemplo anterior de intercambiar los tipos tiende a ser una manera problemática de programar. C++ tiene un modo mucho mejor de codificar este tipo de cosas.

Si el modo en que el compilador asigna los valores no es de su agrado, puede hacerlo manualmente, como sigue:

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

Si da valores a algunos nombres y a otros no, el compilador utilizará el siguiente valor entero. Por ejemplo,

```
enum snap { crackle = 25, pop };
```

El compilador le da a **pop** el valor **26**.

Es fácil comprobar que el código es más legible cuando se utilizan tipos de datos enumerados.

Comprobación de tipos para enumerados
Las enumeraciones en C son bastante primitivas, simplemente asocian valores enteros a nombres, pero no aportan comprobación de tipos.

VARIBLES LOCALES Y EXTERNAS
(Universidad de Castilla-La Mancha, 2018)

extern

La palabra reservada ***extern*** Le dice al compilador que una variable o una función existe, incluso si el compilado aún no la ha visto en el archivo que está siendo compilado en ese momento. Esta variable o función puede definirse en otro archivo o más abajo en el archivo actual. A modo de ejemplo observe el programa 3.15:

```
#include <iostream>
//Esta no es una variable externa, pero el
//compilador debe de saber que existen en algún lado
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // la definición de datos
void func() {
  i++;
  printf("%d",i);
```

}

Programa 3.15

Cuando el compilador encuentra la declaración ***extern int i*** sabe que la definición para ***i*** debe existir en algún sitio como una variable global. Cuando el compilador alcanza la definición de ***i***, ninguna otra declaración es visible, de modo que sabe que ha encontrado la misma ***i*** declarada anteriormente en el archivo. Si se hubiera definido ***i*** como ***static***, estaría indicando al compilador que ***i*** se define globalmente (por ***extern***), pero también que tiene el ámbito del archivo (por ***static***), de modo que el compilador generará un error.

Enlazado

Para comprender el comportamiento de los programas en C, es necesario saber sobre ***enlazado***. En un programa en ejecución, un identificador se representa con espacio en memoria que aloja una variable o un cuerpo de función compilada. El enlazado describe este espacio tal como lo ve el enlazador. Hay dos formas de enlazado: ***enlace interno*** y ***enlace externo***.

Enlace interno significa que el espacio se pide para representar el identificador sólo durante la compilación del archivo. Otros archivos pueden utilizar el mismo nombre de identificador con un enlace interno, o para una variable global, y el enlazador no encontraría conflictos - se pide un espacio separado para cada identificador. El enlace interno se especifica mediante la palabra reservada **static** en C.

Enlace externo significa que se pide sólo un espacio para representar el identificador para todos los archivos que se estén compilando. El espacio se pide una vez, y el enlazador debe resolver todas las demás referencias a esa ubicación. Las variables globales y los nombres de función tienen enlace externo. Son accesibles desde otros archivos declarándolas con la palabra reservada **extern**. Por defecto, las variables definidas fuera de todas las funciones y las definiciones de las funciones implican enlace externo. Se pueden forzar específicamente a tener enlace interno utilizando **static**. Se puede establecer explícitamente que un identificador tiene enlace externo definiéndolo como **extern**. No es necesario definir una variable o una función como **extern** en C.

Las variables automáticas (locales) existen sólo temporalmente, en la pila, mientras se está ejecutando una función. El enlazador no entiende de variables automáticas, de modo que no tienen enlazado.

UNIDAD DE COMPETENCIAS IV. CODIFICACIÓN DE ESTRUCTURAS DE CONTROL.

(12 horas)

RESUMEN.
En esta unidad de competencias se explican las sentencias de control de la programación estructurada secuencial, selección (simple, doble y multíple) e iterativas (mientras, repite, para) continue y break

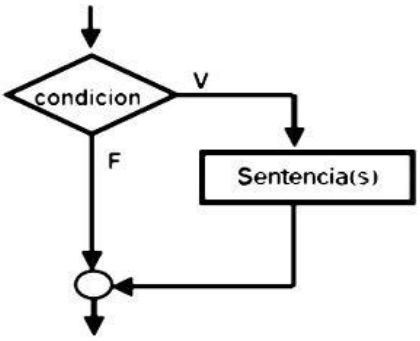
ESTRUCTURAS DE DECISION IF THEN ELSE Y SWITCH CASE

(Universidad Tecnológica de Pereira, 2018)
(Alegsa, 2018)

Las Estructuras de control como de las de decisión, repetitivas y estructuras de selección múltiple permiten transferir el control a otras estructuras sin seguir la ejecución secuencial.
Las estructuras selectivas se utilizan para tomar decisiones lógicas; el cual se suelen denominar también estructuras de decisión o alternativas simple doble o múltiple.
La estructura switch case, es una estructura de decisión múltiple, donde el compilador prueba o busca el valor contenido en una variable contra una lista de constantes de tipo entero, char o enumerable. Si dicha variable no existe ejecuta la instrucción por default.

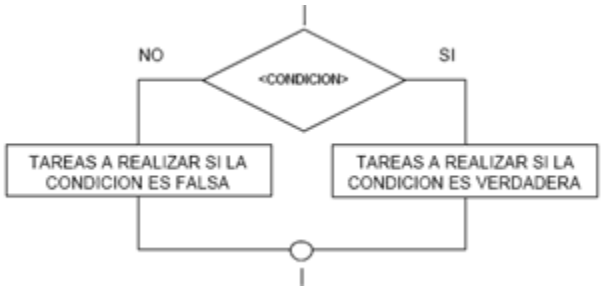
ESTRUCTURAS DE DECISIÓN O SELECCIÓN

Las sentencias de **decisión** o también llamadas estructuras de control que realizan una pregunta la cual retorna verdadero o falso (evalúa una condición) y selecciona la siguiente instrucción a ejecutar dependiendo la respuesta o resultado (ver figura 4.1, figura 4.2 y figura 4.3).



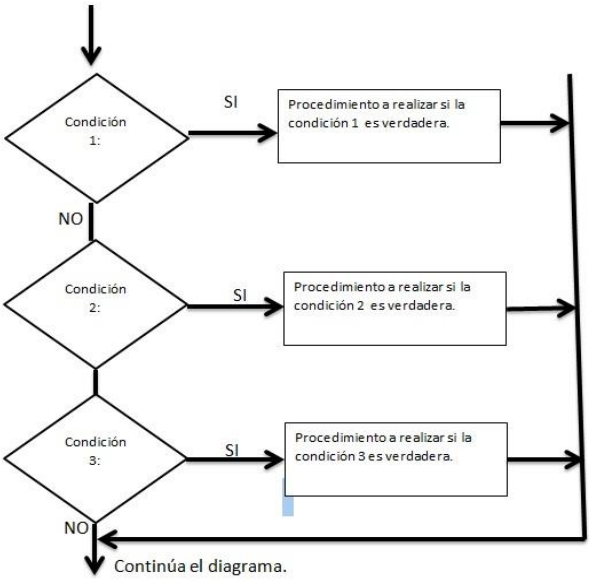

```
#include <stdio.h>
#include <conio.h>
int main() {
int num;
printf( "Introduce un número " );
scanf( "%i", &num );
if (num==10) {
        printf( "El número es correcto\n" );
}
getch();
}
```

Figura 4.1 Estructura de decisión o alternativa simple




```
#include <stdio.h>
#include <conio.h>
int main() {
int a;
printf("Introduce un número ");
scanf( "%i", &a );
if ( a==8 )
        printf ("El número introducido era un
ocho.\n");
else
        printf("Pero si no has escrito un ocho!!!\n");
getch();
}
```

Figura 4.2: estructura de decisión o alternativa doble




```
#include <stdio.h>
#include <conio.h>
int main(){
int a;
printf( "Introduce un número " );
scanf( "%d", &a );
if ( a<10 ){
        printf ( "El número introducido
era menor de 10.\n" );
else if ( a>10 && a<100 ){
        printf ( "El número está entre
10 y 100\n" );
else if ( a>100 )
        printf( "El número es mayor
que 100\n" );
printf( "Fin del programa\n" );
getch();
}
```

Figura 4.3 estructura de decisión o alternativa múltiple

ANIDAMIENTO DE ESTRUCTURAS DE DECISIÓN

Se anidan colocando una estructura en el interior de otra estructura, no debe haber solapamiento. También se cumple para las demás estructuras repetitivas. Ver figura 4.4.

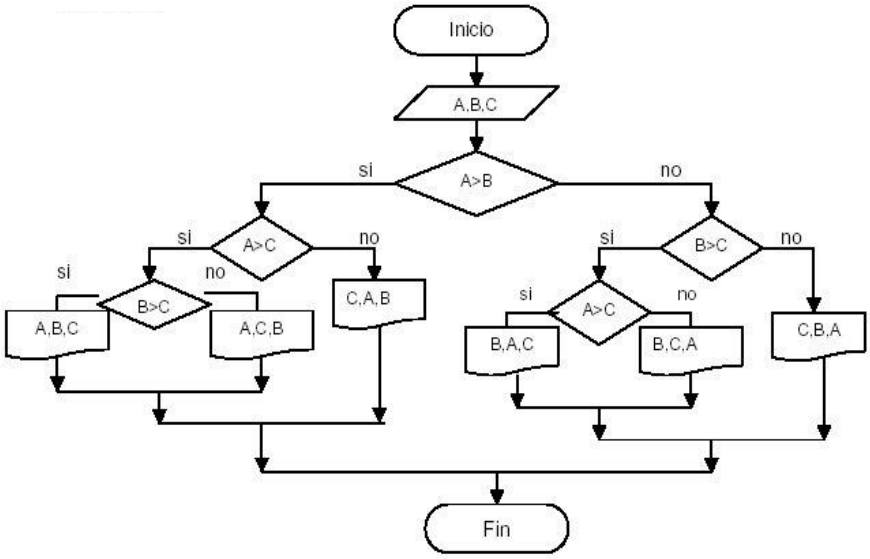
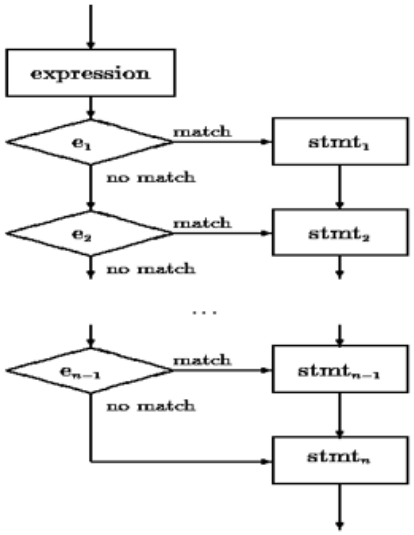


Figura 4.4. Anidamientos

ESTRUCTURA SWITCH-CASE (SEGÚN-CASO)

La estructura según caso, es una estructura de decisión múltiple. Se evalúa una expresión y en función del valor resultante se realiza una determinada tarea o conjunto de tareas. En caso de existir un valor no comprendido entre 1 y n, se realiza una acción excluyente o por defecto (default). En la figura 4.5, se muestra el diagrama de flujo de esta estructura.



Código fuente

```
switch (expresion) {
case 1:instruccion;
          break;
case 2:instruccion;
          break;
case 3: instruccion;
          break;
case n: instruccion;
           break;
default : instrucción;
}
```

Figura 4.5. Estructura switch case (según caso)

Cuando se ejecuta la instrucción switch, la expresión se evalúa y se transfiere el control directamente al grupo de instrucciones cuya etiqueta ***case*** tenga el mismo valor de expresión. Si ninguno de los valores de las etiquetas case coincide con el valor de expresión, entonces no se seleccionará ninguno de los grupos de la instrucción switch o se realizara la instrucción por defecto (default). El valor por defecto (default) es una forma conveniente de generar un mensaje de error en caso de tener entradas inválidas.

break: es una sentencia de salto que se utiliza cuando se requiere salir de un ciclo incondicionalmente antes de que se termine la iteración actual. Ver programas 4.6.

```
#include <stdio.h>
#include <conio.h>
int main() {
int num;
printf("\nIntroduce un numero entero
\n\n " );
```

```
#include <stdio.h>
#include <conio.h>
int main() {
float R1, R2;
char character;
printf( "Introduce dos valores :\n" );
scanf( "%f%f", &R1,&R2 );
```

```
printf("\n1-Verifique si es el número " );
printf( "\n2-Verifique si es el número " );
printf( "\n3-Verifique si es el número " );
printf( "\n4-por defecto \n\n " );
scanf( "%i", &num );
switch( num ) {
case 1: printf( "\nEs un 1\n" ); break;
case 2: printf( "\nEs un 2\n" ); break;
case 3: printf( "\nEs un 3\n" ); break;
default: printf( "\nNo es ni 1, ni 2, ni
3\n" );
}
getch();
}
```

```
printf( "Seleccione una operacion:\n" );
printf( "s: para sumar \n r: para restar \n" );
printf( "p: para multiplicar \n d: para dividir \n" );
character=getch();
switch( character ) {
case 's': printf( "El resultado es %f", R1+R2);
break;
case 'r': printf( "El resultado es %f", R1-R2);
break;
case 'p': printf( "El resultado es %f", R1*R2);
break;
case 'd': printf( "El resultado es %f", R1/R2);
break;
default: printf( "Opcion no valida\n" );
}
getch();
}
```

Figura 4.6

Estructuras repetitivas for, while y do while

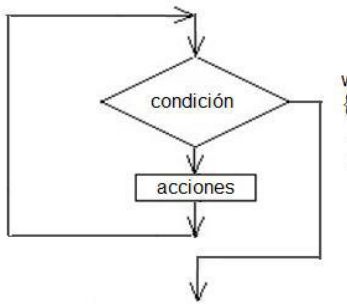
Estructura repetitiva for: la estructura for, es una estructura repetitiva, donde el número de iteraciones o repeticiones se conoce con anterioridad y por ello no se necesita ninguna condición de salida para detener las instrucciones. Ver figura 7.
Código fuente:

```
for(variable = valor inicial; condición; incremento) {
acciones
}
```


```
/*Lista de n números enteros positivos*/
#include<stdio.h>
#include<conio.h>
main() {
int dig;
for (dig=0;dig<=9;++dig)
printf("Numero: %d \n",dig);
getch();
}
```

Figura 4.7. Estructura repetitiva for

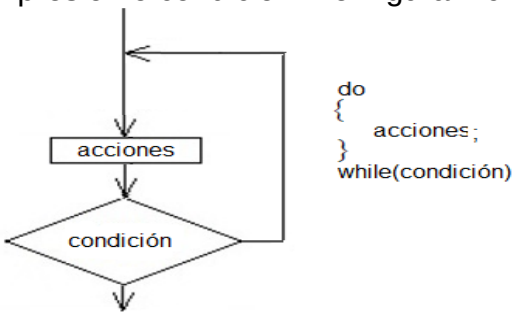
Estructura repetitiva mientras (while): la estructura repetitiva while realizará las acciones mientras la condición o expresión en el while sea verdadera. (Ver figura 4.8)




```
/* lista de números enteros de 0 a 9*/
#include<stdio.h>
#include<conio.h>
main() {
int digito=0;
while(digito<=9)
        printf("%d \n",digito++);
getch();
}
```

Figura 4.8. Estructura while

Estructura repetitiva hacer mientras (do while): la estructura repetitiva do while, en ella se ejecuta al menos una vez la(s) acciones, antes de que se evalúe la expresión o condición. Ver figura 4.9.




```
/* suma de pares entre 1 y n, donde n entra por teclado */
#include <stdio.h>
#include <conio.h>
main() {
int i, suma_par=0, num1;
i=0;
printf("\n digite un numero entero: ");
scanf ("%i",&num1);
do {
        if(i%2==0) /* este operador % me devuelve el
        residuo de la div entre dos enteros*/
                suma_par +=i;
        i++;
} while (i<=num1);
printf ("\n La suma de numeros pares es %d\n",
suma_par);
getch();
}
```

Figura 4.9. Estructura do while

Sentencias ***continue*** y ***break***: Los enunciados **break** y **continue** son utilizados para modificar el flujo de control dentro de un programa.

El **break** utilizado dentro de las estructuras de control causa la inmediata salida de dicha estructura (por lo tanto no sigue repitiéndose el bloque y continúa la ejecución de las instrucciones que le siguen a la estructura de control).

En la figura 4.10 se observa un ejemplo del uso de **break** y de **continue**

```
int num;
num = 1;
while (num <= 10){
    if (num == 5)
        break;
    printf("%d - ", num);
    num++;
}
```

El código anterior imprime:

1 – 2 – 3 – 4 –

```
int num;
num = 0;
while (num <= 7){
        num ++;
        if (num == 5)
            continue;
      printf("%d - ", num);
};
```

El código anterior imprime en pantalla:

1 – 2 – 3 – 4 – 6 – 7 - 8

Figura 4.10. Empleo del break y de continue

Lo que sucede es que cuando la variable **num** toma el valor 5, la condición del **while** se cumple, al ingresar al bloque se evalúa en la estructura **if** si **num** es igual a 5 y se ejecuta el **break** saliendo del bloque **while**. **Num** termina valiendo 5 pues jamás se ejecuta la suma **num = num + 1**.

Por otra parte, el enunciado **continue**, dentro de las estructuras de repetición, al ser ejecutado salta las instrucciones que siguen en el bloque y ejecuta la siguiente repetición en el ciclo.

Como vemos, en una de las repeticiones se salta la impresión del número 5. Algunos programadores dicen que el uso del **break** y del **continue** dentro de las estructuras de control (excepto el **break** en la estructura **switch**) viola las normas de la programación estructurada. Lo cierto es que no es necesario el uso de **break** y **continue** si se utilizan las estructuras correctas.

UNIDAD DE COMPETENCIAS V. BIBLIOTECAS DE FUNCIONES Y USO DE PRINCIPALES FUNCIONES DEL LENGUAJE DE PROGRAMACIÓN

(2 horas)

RESUMEN

En esta unidad de competencias se revisarán las principales bibliotecas de funciones del lenguaje de programación C. (funciones para lectura, escritura; funciones matemáticas; para el manejo de cadenas; como interface con el Sistema Operativo; funciones para el manejo de tiempo.

BIBLIOTECAS EN LENGUAJE C

(Plauger P. J., 1992)

Las bibliotecas o archivos de cabecera en lenguaje C, son los que contienen o almacenan funciones que realizan operaciones y cálculos de uso frecuente y son parte de cada compilador. El programador debe invocar todos aquellos archivos o bibliotecas que necesite.

Las bibliotecas del lenguaje C standard son:

- **assert.h**: El único propósito de usar este header es proveer una definición de la macro *assert*. Se usa la macro para colocarla en lugares críticos dentro del programa usada para comprobar ciertas suposiciones

- **conio.h:** Contiene los prototipos de las funciones, macros, y constantes para preparar y manipular la consola en modo texto en el entorno de MS-DOS.

cgets	clreol	clrscr	cprintf	cputs	cscanf	delline
getche	getpass	gettext	gettextinfo	gotoxy	highvideo	inport
insline	Getch	lowvideo	movetext	normvideo	outport	putch
puttext	setcursortype	textattr	textbackground	textcolor	textmode	ungetch

- **ctype.h**: Contiene los prototipos de las funciones y macros para clasificar caracteres.

- **errno.h**: En ella se definen las macros que presentan un informe de error a través de códigos de error. La macro errno se expande a un lvalue con tipo int, que contiene el último código de error generado en cualquiera de las funciones utilizando la instalación de errno

La definición de estas constantes puede depender del compilador y se incluyen aquí sólo como ejemplo.

```
#define EPERM           1       /* Operation not permitted */
#define   ENOFILE       2       /* No such file or directory */
#define   ENOENT        2
#define   ESRCH         3       /* Cache failure */
#define EINTR*          4       /* Interrupted function call* */
#define   EIO       5   /* Input/output error */
#define   ENXIO         6       /* No such device or address */
#define   E2BIG         7       /* Arg list too long */
#define   ENOEXEC               8       /* Exec format error */
#define   EBADF         9       /* Bad file descriptor */
#define   ECHILD        10      /* No child processes */
#define   EAGAIN        11      /* Resource temporarily unavailable */
#define   ENOMEM        12      /* Not enough space */
#define   EACCES        13      /* Permission denied */
#define   EFAULT        14      /* Bad address */
/*  15 - Unknown Error */
#define   EBUSY         16      /* strerror reports "Resource device" */
#define   EEXIST        17      /* File exists */
#define   EXDEV         18      /* Improper link (cross-device link?) */
#define   ENODEV        19      /* No such device */
#define   ENOTDIR       20      /* Not a directory */
#define   EISDIR        21      /* Prada is a directory */
#define   EINVAL        22      /* Invalid argument */
#define   ENFILE        23      /* Too many open files in system */
#define   EMFILE        24      /* Too many open files */
#define   ENOTTY        25      /* Inappropriate I/O control operation */

/* 26 - Unknown Error */
#define   EFBIG         27      /* File too large */
#define   ENOSPC        28      /* No space left on device */
#define   ESPIPE        29      /* Invalid seek (seek on a pipe?) */
#define   EROFS         30      /* Read-only file system */
#define   EMLINK        31      /* Too many links */
#define   EPIPE         32      /* Broken pipe */
#define   EDOM          33      /* Domain error (math functions) */
#define   ERANGE        34      /* Result too large (possibly too small) */
/* 35 - Unknown Error */
#define   EDEADLOCK     36      /* Resource deadlock avoided (non-Cyg) */
```

```
#define    EDEADLK                 36
/* 37 - Unknown Error */
#define    ENAMETOOLONG  38        /* Filename too long (91 in Cyg?) */
#define    ENOLCK        39        /* No locks available (46 in Cyg?) */
#define    ENOSYS        40        /* Function not implemented (88 in Cyg?) */
#define    ENOTEMPTY     41        /* Directory not empty (90 in Cyg?) */
#define    EILSEQ        42        /* Illegal byte sequence */
```

- **float.h**: contiene un conjunto de varias constantes dependientes de la plataforma relacionadas con valores de coma flotante. Estas constantes son propuestas por ANSI C. Permiten hacer más portátiles los programas.
- **limits.h**: determina varias propiedades de los diversos tipos de variables. Las macros definidas en este encabezado limitan los valores de varios tipos de variables como char, int y long.
- **locale.h**: define la configuración específica de la ubicación, como los formatos de fecha y los símbolos de moneda.
- **math.h**: diseñado para operaciones matemáticas básicas. Muchas de sus funciones incluyen el uso de números en coma flotante.
  Funciones miembro:

Nombre		Descripción
acos		arcocoseno
asin		arcoseno
atan		arcotangente
atan2		arcotangente de dos parámetros
floor		función suelo
cos		coseno
cosh		coseno hiperbólico
exp(double x)		función exponencial, computa e^x
fabs		valor entero
ceil		menor entero no menor que el parámetro
fmod		residuo de la división de flotantes
frexp		fracciona y eleva al cuadrado.
ldexp		tamaño del exponente de un valor en punto flotante
log		logaritmo natural
log10		logaritmo en base 10
modf		obtiene un valor en punto flotante íntegro y en partes
pow(x,y)		eleva un valor dado a un exponente, x^y
sin		seno
sinh		seno hiperbólico
sqrt		raíz cuadrada
tan		tangente
tanh		tangente hiperbólica

- **setjmp,h:** proporcionar "saltos no locales": control de flujo que se desvía de la llamada a subrutina habitual y la secuencia de retorno. Las funciones complementarias **setjmp** y **longjmp** proporcionan esta funcionalidad.
- **signal.h**: Sirve para especificar como un programa maneja señales mientras se ejecuta. Una señal puede reportar un comportamiento excepcional en el programa (*tales como la división por cero*), o una señal puede reportar algún evento asíncrono fuera del programa (*como alguien está pulsando una tecla de atención interactiva en el teclado*)

  Una señal puede ser generada llamando a raise (para enviar una señal al proceso actual) o kill (para enviar una señal a cualquier proceso). Cada implementación define lo que genera las señales (en su caso) y en qué circunstancias las genera. Una implementación puede definir otras señales además de las que figuran en esta lista. La cabecera estándar <signal.h> puede definir macros adicionales con nombres que empiezan con SIG para especificar los valores de señales adicionales. Todos los valores son expresiones constantes enteras >= 0.
  Un manejador de la señal se puede especificar para todas las señales excepto dos (SIGKILL y SIGSTOP no puede ser atrapadas, bloqueadas o ignoradas). Un manejador de la señal es una función que el entorno de destino llama cuando se produce la señal correspondiente. El entorno de destino suspende la ejecución del programa hasta que vuelva la señal de controlador o llama a longjmp. Para una máxima portabilidad, un manejador de la señal asíncrona sólo debe:
    - hacer llamadas (que tienen éxito) a la señal de la función
    - asignar valores a los objetos de tipo volátiles sig_atomic_t
    - devolver el control a la función que la llamó

  Si la señal informa de un error en el programa (y la señal no es asincrónica), el manejador de la señal puede terminala llamando a abort, exit, o longjmp.

- **stdarg.h**: Permite que las funciones acepten un número indefinido de argumentos
- **stddef.h**: Define varios tipos de variables y macros. Muchas de estas definiciones también aparecen en otros encabezados
- **stdio.h:** Contiene las definiciones de las macros, las constantes, las declaraciones de funciones de la biblioteca estándar del lenguaje de programación C para hacer operaciones, estándar, de entrada y salida, así como la definición de tipos necesarias para dichas operaciones.

  Las funciones declaradas en stdio.h pueden clasificarse en dos categorías: funciones de manipulación de archivos y funciones de manipulación de entradas y salidas.

Nombre	Descripción
Funciones de manipulación de archivos	
`fclose`	Cierra un archivo a través de su puntero.
`fopen`, `freopen`, `fdopen`	Abre un archivo para lectura, para escritura/reescritura o para adición.
`remove`	Elimina un archivo.
`rename`	Cambia al archivo de nombre.
`rewind`	Coloca el indicador de posición de archivo para el stream apuntado por stream al comienzo del fichero.
`tmpfile`	Crea y abre un archivo temporal que es borrado cuando cerramos con la función *fclose()*.
Funciones de manipulación de entradas y salidas.	
`clearerr`	Despeja los indicadores de final de archivo y de posición de archivo para el stream apuntado por stream al comienzo del archivo.
`feof`	Comprueba el indicador de final de archivo.
`ferror`	Comprueba el indicador de errores.
`fflush`	Si stream apunta a un *stream* de salida o de actualización cuya operación más reciente no era de entrada, la función *fflush* envía cualquier dato aún sin escribir al entorno local o a ser escrito en el archivo; si no, entonces el comportamiento no está definido. Si stream es un puntero nulo, la función fflush realiza el despeje para todos los *streams* cuyo comportamiento está descrito anteriormente.
`fgetpos`	Devuelve la posición actual del archivo.
`fgetc`	Devuelve un carácter de un archivo.
`fgets`	Consigue una cadena de caracteres de un archivo.
`fputc`	Escribe un carácter en un archivo.
`fputs`	Escribe una cadena de caracteres en un archivo.
`ftell`	Devuelve la posición actual del archivo como número de bytes.
`fseek`	Sitúa el puntero de un archivo en una posición aleatoria.
`fsetpos`	Cambia la posición actual de un archivo.
`fread`	lee diferentes tamaños de datos de un archivo.
`fwrite`	Envía, desde el array apuntado por puntero, hasta *nmemb* de elementos cuyo tamaño es especificado por tamaño. El indicador de posición de archivos es avanzado por el número de caracteres escritos correctamente. Si existe

	un error, el valor resultante del indicador de posición de archivos es indeterminado.
`getc`	Devuelve un carácter desde un archivo.
`getchar`	Devuelve un carácter desde la entrada estándar
`gets`	Lee caracteres de entrada hasta que encuentra un salto de línea, y los almacena en un único argumento.
`printf`, `fprintf`, `sprintf` `snprintf`	Usados para imprimir salidas de datos.
`vprintf`	También utilizado para imprimir salidas.
`perror`	Escribe un mensaje de error a stderr.
`putc`	Escribe un carácter en un archivo.
`putchar`	Escribe un carácter en la salida estándar
`puts`	Escribe una cadena de caracteres en la salida estándar
`scanf`, `fscanf`, `sscanf`	Utilizado para introducir entradas.
`vfscanf`, `vscanf`, `vsscanf`	También utilizado para introducir entradas.
`setbuf`	Esta función es equivalente a la función setvbuf pasando los valores _IOFBF para modo y BUFSIZ para tamaño, o (si acumulador es un puntero nulo), con el valor _IONBF para modo.
`setvbuf`	Sólo puede ser usada después de que el stream apuntado por stream ha sido asociado con un archivo abierto y antes de otra operación cualquiera es llevada a cabo al stream. El argumento modo determina cómo stream será almacenado según lo siguiente: _IOFBF ocasiona la entrada/salida a ser completamente almacenado; _IOLBF ocasiona la entrada/salida a almacenar por líneas; _IONBF ocasiona la entrada/salida a no ser almacenado. Si acumulador no es un puntero nulo, el array al que es apuntado puede ser usado en vez de la acumulación adjudicada por la función setvbuf. El argumento tamaño especifica el tamaño del array.
`tmpnam`	Genera una cadena de caracteres que es un nombre válido para archivos y que no es igual al nombre de un archivo existente. La función tmpnam genera una cadena diferente cada vez que es llamada, hasta un máximo de TMP_MAX veces. Si la función es llamada más veces que TMP_MAX, entonces el comportamiento de la función está definido según la implementación del compilador.
`ungetc`	

Las constantes definidas son:

Nombre	Descripción
EOF	Entero negativo (`int`) usado para indicar "**fin de archivo**".

BUFSIZ	Entero que indica el tamaño del buffer de datos utilizado por la función setbuf().
FILENAME_MAX	Tamaño máximo de la cadena de caracteres que contienen el nombre de un archivo para ser abierto
FOPEN_MAX	Número máximo de archivo que pueden estar abiertos simultáneamente.
_IOFBF	Abreviatura de *input/output fully buffered* (buffer entrada/salida totalmente lleno); es un entero que se puede pasar como parámetro de la función setvbuf() para requerir *bloqueo del buffer* en la entrada y salida del *stream* abierto.
_IOLBF	Abreviatura de *input/output line buffered* (...??); es un entero que se puede pasar como parámetro a la función setvbuf() para requerir *line buffered (??)* en la entrada y salida del *stream* abierto.
_IONBF	Abreviatura de "input/output not buffered" (entrada/salida sin buffer); es un entero que se puede pasar como parámetro a la función setvbuf() para requerir que la entrada salida del stream abierto funcione sin buffer.
L_tmpnam	Tamaño de la cadena de caracteres con la longitud suficiente para almacenar un nombre de fichero temporal generado por la función tmpnam().
NULL	Macro que representa la constante puntero nulo; representa un valor de puntero que no apunta a ninguna dirección válida de objeto alguno en memoria.
SEEK_CUR	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo a la posición actual del archivo.
SEEK_END	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo al final del archivo.
SEEK_SET	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo al inicio del archivo.
TMP_MAX	El número máximo de nombres de archivos únicos generables por la función tmpnam().

- **stdlib.h:** Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras.
  Las funciones que pertenencen a stdlib.h pueden clasificarse en las siguientes categorías: conversión, memoria, control de procesos, ordenación y búsqueda, matemáticas.

Nombre	Descripción
Conversión de tipos	
atof (*ascii to float*)	cadena de caracteres a coma flotante

atoi (*ascii to integer*)	cadena de caracteres a entero
atol (C Standard Library)]] (*ascii to long*)	cadena de caracteres a entero tamaño largo
strtod (*string to double*)	cadena de caracteres a coma flotante tamaño doble
strtol (*string to long*)	cadena de caracteres a entero largo
strtoul (*string to unsigned long*)	cadena de caracteres a entero largo sin signo (positivo)
Generación de números pseudo-aleatorios	
rand	Genera un número pseudo-aleatorio
srand	Establece la semilla para el generador de números pseudo-aleatorios
Gestión de memoria dinámica	
malloc, calloc, realloc	Reservan memoria dinámica del heap (*montón* o *montículo*)
free	Liberan memoria devolviéndola al *heap*
Control de procesos	
abort	terminar ejecución anormalmente
atexit	registrar una función callback para la salida del programa
exit (operating system)	terminar ejecución del programa
getenv	recuperar una variable de entorno
system (C Standard Library)	ejecutar un comando externo
Ordenación y búsqueda	
bsearch	búsqueda binaria en un array
qsort (C Standard Library)	ordena un vector (informática) usando Quicksort
Matemáticas	
abs, labs	valor absoluto
div, ldiv	división entera o euclidiana

En la librería #include<stdlib.h>, existe la función system(“color f1”) que se utiliza para cambiar el color de fondo y el de color de la fuente.

Colores de Fondo		**Colores de fuente**	
0	Negro	A	verde claro

1	Azul	B	aguamarina claro
2	verde	C	rojo claro
3	Aguamarina	D	purpura claro
4	Rojo	E	amarillo claro
5	Purpura	F	blanco brillante
6	Amarillo		
7	Blanco		
8	gris		
9	azul claro		

Con la función system("pause") se utiliza para pausar un programa una vez esté corriendo y reemplaza a la función getch() de la librería #include<conio.h>. Ambas hacen lo mismo y se colocan la final antes de cerrar el programa principal.

Siendo la gestión de memoria dinámica importante, se da una explicación un poco más profunda de las funciones más usadas:

Gestión de memoría dinámica	
malloc	void * malloc(size_t size) La función malloc reserva espacio en el heap a un objeto cuyo tamaño es especificado en bytes (size). El heap es un área de memoria que se utiliza para guardar estructuras dinámicas en tiempo de ejecución. Si se puede realizar la asignación de memoria, se retorna el valor de la dirección de inicio del bloque. De lo contrario, si no existe el espacio requerido para el nuevo bloque, o si el tamaño es cero, retorna NULL.

calloc	void *calloc(size_t nitems, size_t size)
	La función calloc asigna espacio en el heap de un número de elementos (nitems) de un tamaño determinado (size). El bloque asignado se limpia con ceros. Si se desea asignar un bloque mayor a 64kb, se debe utilizar la función farcalloc.
	Ejemplo:
	char *s=NULL;
	str=(char*) calloc(10,sizeof(char));
	
	Si se puede realizar la asignación, se retorna el valor del inicio del block asignado. De lo contrario, o no existe el espacio requerido para el nuevo bloque, o el tamaño es cero, retorna NULL.

realloc	void *realloc(*punt*, *nuevotama*): Cambia el tamaño del bloque apuntado por *punt.* El nuevo tamaño puede ser mayor o menor y no se pierde la información que hubiera almacenada (si cambia de ubicación se copia).
free	void free(void *dir_memoria)

	La función free libera un bloque de la memoria del heap. El argumento dir_memoria apunta a un bloque de memoria previamente asignado a través de una llamada a calloc, malloc o realloc. Después de la llamada, el bloque liberado estará disponible para asignación.

- **string.h**: Contiene la definición de macros, constantes, funciones y tipos y algunas operaciones de manipulación de memoria. Las funciones declaradas en string.h se han hecho muy populares, por lo que están garantizadas para cualquier plataforma que soporte C.
  Constantes:

Nombre	Descripción
`NULL`	macro que representa la constante puntero nulo; representa un valor de puntero que no apunta a ninguna dirección válida de objeto alguno en memoria
`size_t`	tipo entero sin signo (positivo); es el tipo devuelto por el operador `sizeof`

Funciones:

Nombres	Descripción
memcpy	copia n bytes entre dos áreas de memoria que no deben solaparse
memmove	copia n bytes entre dos áreas de memoria; al contrario que `memcpy` las áreas pueden solaparse
memchr	busca un valor a partir de una dirección de memoria dada y devuelve un puntero a la primera ocurrencia del valor buscado o NULL si no se encuentra
memcmp	compara los n primeros caracteres de dos áreas de memoria
memset	sobre escribe un área de memoria con un patrón de bytes dado
strcat	añade una cadena al final de otra
strncat	añade los n primeros caracteres de una cadena al final de otra
strchr	localiza un carácter en una cadena, buscando desde el principio
strrchr	localiza un carácter en una cadena, buscando desde el final
strcmp	compara dos cadenas alfabéticamente ('a'!='A')
strncmp	compara los n primeros caracteres de dos cadenas numéricamente ('a'!='A')
strcoll	compara dos cadenas según la colación actual ('a'=='A')
strcpy	copia una cadena en otra
strncpy	copia los n primeros caracteres de una cadena en otra
strerror	devuelve la cadena con el mensaje de error correspondiente al número de error dado

strlen	devuelve la longitud de una cadena
strspn	devuelve la posición del primer carácter de una cadena que no coincide con ninguno de los caracteres de otra cadena dada
strcspn	devuelve la posición del primer carácter que coincide con alguno de los caracteres de otra cadena dada
strpbrk	encuentra la primera ocurrencia de alguno de los caracteres de una cadena dada en otra
strstr	busca una cadena dentro de otra
strtok	parte una cadena en una secuencia de tokens
strxfrm	transforma una cadena en su forma de colación (??)
strrev	invierte una cadena

Extensión para C (ISO)

Nombre	Descripción	Especificación
strdup	hace un duplicado de la cadena dada reservando dinámicamente la memoria necesaria	POSIX; originalmente una extensión BSD
strcpy_s	variante de strcpy que verifica los límites	ISO/IEC WDTR 24731
mempcpy	variante de memcpy que devuelve un puntero al byte siguiente al último byte escrito	GNU
memccpy	variante de memcpy que para al encontrar un byte determinado	UNIX 98?
strerror_r	análogo a strerror_r (*thread-safe*)	GNU, POSIX
strlcpy	variante de strcpy que verifica los límites	originalmente OpenBSD, actualmente también FreeBSD, Solaris, OS X
strtok_r	versión *thread-safe* de strtok	POSIX
strsignal	análogamente a strerror, devuelve la cadena representación de la señal sig (no *thread safe*)	BSDs, Solaris, Linux

- **time.h**: es un archivo de cabecera de la biblioteca estándar del lenguaje de programación C que contiene funciones para manipular y formatear la fecha y hora del sistema
  Funciones:

Nombre	Descripción

char * asctime(struct tm *)	Recibe una variable de tipo puntero a estructura tm (struct tm*) y devuelve una cadena de caracteres cuyo formato es: "Www Mmm dd hh:mm:ss yyyy\n" (ej: **Tue May 15 19:07:04 2008\n**)
clock_t clock (void)	Devuelve el número de pulsos de reloj desde que se inició el proceso.
char * ctime(time_t *)	Recibe una variable de tipo puntero a time_t (time t*) y devuelve una cadena con el mismo formato que asctime()
double difftime(time_t, time t)	Recibe dos variables de tipo time_t, calcula su diferencia y devuelve el resultado (double) expresado en segundos.
struct tm *gmtime(time_t *)	Recibe un puntero a una variable de tiempo (time_t*) y devuelve su conversión como fecha/hora UTC a struct tm a través de un puntero.
struct tm *localtime(time_t *)	Similar funcionalidad a gmtime(), pero devuelve la conversión como fecha/hora LOCAL.
time_t mktime(struct_tm *)	Inversamente a gmtime() y localtime() ,l recibe un puntero a struct tm (struct tm*) y devuelve su conversión al tipo time t.
time_t time(time_t *)	Devuelve la fecha/hora (time_t) actual o -1 en caso de no ser posible. Si el argumento que se le pasa no es NULL, también asigna la fecha/hora actual a dicho argumento.
size_t strftime(char *,size_t,char *,struct tm *)	Formatea la información pasada mediante la estructura (struct tm*) mediante el formato indicado en una cadena (char*) e imprime el resultado sobre otra cadena (char*) hasta un límite de caracteres (size_t).

Constantes

Nombre	Descripción
CLK_PER_SEC	Constante que define el número de pulsos de reloj por segundo; usado por la función clock()
CLOCKS_PER_SEC	nombre alternativo para CLK_PER_SEC usado en su lugar en algunas bibliotecas
`CLK_TCK`	usualmente una macro para `CLK_PER_SEC`

Tipos de datos

Nombre	Descripción
clock_t	tipo de dato devuelto por clock(), generalmente un long int

time t	tipo de dato devuelto por time(), generalmente un long int
struct tm	representación del tiempo en formato de calendario (fecha/hora)

Fecha (dia/hora) del calendario

Atributo	Descripción
int tm_hour	hora (0 - 23)
int tm_isdst	Horario de verano enabled/disabled
int tm_mday	día del mes (1 - 31)
int tm_min	minutos (0 - 59)
int tm_mon	mes (0 - 11, 0 = Enero)
int tm_sec	segundos (0 - 60)
int tm_wday	día de la semana (0 - 6, 0 = domingo)
int tm_yday	día del año (0 - 365)
int tm_year	año desde 1900

UNIDAD DE COMPETENCIAS VI. USO DE MODULARIZACIÓN EN LA IMPLEMENTACIÓN DE PROGRAMAS.

(4 Horas)

RESUMEN.

En esta unidad de competencias se explica paso a paso el manejo de funciones, el pase de parámetros por valor y la forma en que maneja el pase de parámetros por referencia, aparte de explicar el retorno de valores. Es fundamental este capítulo ya que muestra la forma en que se pueden hacer módulos más sencillos que permitan un mejor diseño de algún programa en particular utilizando los conceptos de cohesión y acoplamiento.

MODULARIDAD

(Ayala de la Vega, Aguilar Juárez, Zarco Hidalgo, & Gómez Ayala, 2016)
(Ceballos, C/C++ Curso de Programación, 2015)
(Backman K. , 2012)
(Deitel & M, 1995)

La **modularidad** es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan módulos en relación con otros. En principio veremos el concepto de ***independencia***. Diremos que dos módulos son totalmente independientes si ambos pueden funcionar completamente sin la presencia del otro. Esto implica que no existen interconexiones entre los módulos, y que se tiene un valor cero en la escala de "dependencia".

En general veremos que a mayor número de interconexiones entre dos módulos, se tiene una menor independencia.

El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de la información.

La cuestión aquí es: ¿cuánto debe conocerse acerca de un módulo para poder comprender otro módulo? Cuanto más debamos conocer acerca del módulo B para poder comprender el módulo A, menos independientes serán A de B.

La simple cantidad de conexiones entre módulos, no es una medida completa de la independencia funcional. La independencia funcional se mide con dos criterios cualitativos: ***acoplamiento*** **y** ***cohesión***. Estudiaremos en principio el primero de ellos.

ACOPLAMIENTO

Módulos altamente "acoplados" estarán unidos por fuertes interconexiones, módulos débilmente acoplados tendrán pocas y débiles interconexiones, en tanto que los módulos "desacoplados" no tendrán interconexiones entre ellos y serán independientes.

> El *acoplamiento* es un concepto abstracto que nos indica el grado de interdependencia entre módulos.

En la práctica podemos materializarlo como la probabilidad de que en la codificación, depuración, o modificación de un determinado módulo, el programador necesite tomar conocimiento acerca de partes de otro módulo. Si dos módulos están fuertemente acoplados, existe una alta probabilidad de que el programador necesite conocer uno de ellos en orden de intentar realizar modificaciones al otro.
Claramente, el costo total del sistema se verá fuertemente influenciado por el grado de acoplamiento entre los módulos.

FACTORES QUE INFLUENCIAN EL ACOPLAMIENTO

Los cuatro factores principales que influyen en el acoplamiento entre módulos son:

- Tipo de conexión entre módulos: los sistemas normalmente conectados, tienen menor acoplamiento que aquellos que tienen conexiones patológicas.
- Complejidad de la interface: Esto es aproximadamente igual al número de ítems diferentes pasados (no cantidad de datos). Más ítems, mayor acoplamiento.
- Tipo de flujo de información en la conexión: los sistemas con acoplamiento de datos tienen menor acoplamiento que los sistemas con acoplamiento de control, y estos a su vez menos que los que tienen acoplamiento híbrido.
- Momento en que se produce el ligado de la Conexión: Conexiones ligadas a referentes fijos en tiempo de ejecución, resultan con menor acoplamiento que cuando el ligado tiene lugar en tiempo de carga, el cual tiene a su ver menor acoplamiento que cuando el ligado se realiza en tiempo de linkage-edición, el cual tiene menos acoplamiento que el que se realiza en tiempo de compilación, todos los que a su vez tiene menos acoplamiento que cuando el ligado se realiza en tiempo de codificación.

TIPOS DE CONEXIONES ENTRE MÓDULOS

Una conexión en un programa, es una referencia de un elemento, por nombre, dirección, o identificador de otro elemento.

Una conexión intermódular ocurre cuando el elemento referenciado está en un módulo diferente al del elemento referenciante.

El elemento referenciado define una interface, un límite del módulo, a través del cual fluyen datos y control.

La interface puede considerarse como residente en el elemento referenciado. Puede pensarse como un enchufe (socket) donde la conexión del elemento referenciante se inserta.

Toda interface en un módulo representa cosas que deben ser conocidas, comprendidas, y apropiadamente conectadas por los otros módulos del sistema.

Se busca minimizar la complejidad del sistema/módulo, en parte, minimizando el número y complejidad de las interfaces por módulo.

Todo módulo además debe tener al menos una interface para ser definido y vinculado al resto del sistema.

Pero, ¿es una interface de identidad simple suficiente para implementar sistemas que funcionen adecuadamente? La cuestión aquí es: *¿A qué propósito sirven las interfaces?*

Solo flujos de *control* y *datos* pueden pasarse entre módulos en un sistema de programación. Una interface puede cumplir las siguientes cuatro únicas funciones:

- Transmitir datos a un módulo como parámetros de entrada
- Recibir datos desde un módulo como resultados de salida
- Ser un nombre por el cual se recibe el control
- Ser un nombre por el cual se transmite el control

Un módulo puede ser identificado y activado por medio de una interfaz de identidad simple. También podemos pasar datos a un módulo sin agregar otras interfaces, haciendo a la interfaz de entrada capaz de aceptar datos como control. Esto requiere que los elementos de datos sean pasados dinámicamente como argumentos (parámetros) como parte de la secuencia de activación, que da el control a un módulo; cualquier referencia estática a datos puede introducir nuevas interfaces.

Se necesita también que la interface de identidad de un módulo sirva para transferir el retorno del control al módulo llamador. Esto puede realizarse haciendo que la transferencia de control desde el llamador sea una transferencia *condicional.* Debe implementarse además un mecanismo para transmitir datos de retorno desde el módulo llamado hacia el llamador. Puede asociarse un valor a una activación particular del módulo llamado, la cual pueda ser usada contextualmente en el llamador. Tal es el caso de las funciones lógicas. Alternativamente pueden transmitirse parámetros para definir ubicaciones donde el módulo llamado retorna valores al llamador.

Si todas las conexiones de un sistema se restringen a ser completamente parametrizadas (con respecto a sus entradas y salidas), y la transferencia condicional de control a cada módulo se realiza a través de una identidad simple y única, diremos que el sistema está *mínimamente conectado*.

Diremos que un sistema está *normalmente conectado* cuando cumple con las condiciones de mínimamente conectado, excepto por alguna de las siguientes consideraciones:

- Existe más de un punto de entrada para un mismo módulo
- El módulo activador o llamador puede especificar como parte del proceso de activación un punto de retorno que no sea la próxima sentencia en el orden de ejecución.
- El control es transferido a un punto de entrada de un módulo por algún mecanismo distinto a una llamada explícita (ej. perform thru del COBOL).

El uso de múltiples puntos de entrada garantiza que existirán más que el número mínimo de interconexiones para el sistema. Por otra parte si cada punto de entrada determina funciones con mínima conexión a otros módulos, el comportamiento del sistema será similar a uno mínimamente interconectado.

De cualquier manera, la presencia de múltiples puntos de entrada a un mismo módulo, puede ser un indicativo de que el módulo está llevando a cabo más de una función específica. Además, es una excelente oportunidad para que el programador superpondrá parcialmente el código de las funciones comprendidas dentro del mismo módulo, quedando dichas funciones *acopladas por contenido*.

De manera similar, *los puntos de retorno alternativo* son frecuentemente útiles dentro del espíritu de los sistemas normalmente conectados. Esto se da cuando un módulo continuará su ejecución en un punto que depende del valor resultante de una decisión realizada por un módulo subordinado invocado previamente. En un caso de mínima conexión, el módulo subordinado retornará el valor como un parámetro, el cual deberá ser testeado nuevamente en el módulo superior. Sin embargo, el módulo superior puede indicar por algún medio directamente el punto

donde debe continuarse la ejecución del programa, (un valor relativo + o - direcciones a partir de la instrucción llamadora, o un parámetro con una dirección explícita).

Si un sistema no está mínima o normalmente conectados, entonces algunos de sus módulos presentarán conexiones patológicas. Esto significa que al menos un módulo tendrá referencias explícitas a identificadores definidos dentro de los límites de otro módulo.

COMPLEJIDAD DE LA INTERFACE

La segunda dimensión del acoplamiento es la *complejidad de interface*. Cuanto más compleja es una conexión, mayor acoplamiento se tiene. Un módulo con una interface de 100 parámetros generará mayor acoplamiento que un que solo necesite tres parámetros.

El significado de "complejidad" es el de complejidad en términos humanos, como lo visto anteriormente.

TIPO DE FLUJO DE INFORMACIÓN

Otro aspecto importante del acoplamiento tiene que ver con el *tipo* de información que se transmite entre el módulo superior y subordinado. Distinguiremos tres tipos de flujo de información:

- datos
- control
- híbrido

Los datos son información sobre la cual una pieza de programa opera, manipula, o modifica.

La información de control (aun cuando está representada por variables de dato) es aquella que gobierna como se realizarán las operaciones o manipulaciones sobre los datos.

Diremos que una conexión presenta *acoplamiento por datos* si la salida de datos del módulo superior es usada como entrada de datos del subordinado. Este tipo de acoplamiento también es conocido como de entrada-salida.

Diremos que una conexión presenta *acoplamiento de control* si el módulo superior comunica al subordinado información que controlará la ejecución del mismo. Esta

información puede pasarse como datos utilizados como señales o "banderas" (flags) o bien como direcciones de memoria para instrucciones de salto condicional (branch-adress). Estos son elementos de control "disfrazados" como datos.

El acoplamiento de datos es mínimo, y ningún sistema puede funcionar sin él.

La comunicación de datos es *necesaria* para el funcionamiento del sistema, sin embargo, la comunicación de control es una característica no deseable y *prescindible*, que sin embargo aparece muy frecuentemente en los programas.

Se puede minimizar el acoplamiento si solo se transmiten datos a través de las interfaces del sistema.

El acoplamiento de control abarca todas las formas de conexión que comuniquen elementos de control. Esto no solo involucra transferencia de control (direcciones o banderas), si no que puede involucrar el pasaje de datos que cambia, regula, o sincroniza la ejecución de otro módulo.

Esta forma de acoplamiento de control indirecto o secundario se conoce como *coordinación*. La coordinación involucra a un módulo en el contexto procedural de otro. Esto puede comprenderse con el siguiente ejemplo: supongamos que el módulo A llama al módulo B suministrándole elementos de datos discretos. La función del módulo B es la de agrupar estos elemento de datos en un ítem compuesto y retornárselo al módulo A (superior). El módulo B enviará al módulo A, señales o banderas indicando que necesita que se le suministre otro ítem elemental, o para indicarle que le está devolviendo el ítem compuesto. Estas banderas serán utilizadas dentro del módulo A para coordinar su funcionamiento y suministrar a B lo requerido.

Cuando un módulo modifica el contenido procedural de otro módulo, decimos que existe *acoplamiento híbrido*. El acoplamiento híbrido es una modificación de sentencias intermódular. En este caso, para el módulo destino o modificado, el acoplamiento es visto como de control en tanto que para el módulo llamador o modificador es considerado como de datos.

El grado de interdependencia entre dos módulos vinculados con acoplamiento híbrido es muy fuerte. Afortunadamente es una práctica en decadencia y reservada casi con exclusividad a los programadores en ensamblador.

TIEMPO DE LIGADO DE CONEXIONES INTERMÓDULARES

"Ligado" o "Binding" es un término comúnmente usado en el campo del procesamiento de datos para referirse a un proceso que resuelve o fija los valores de identificadores dentro de un sistema.

El ligado de variables a valores, o más genéricamente, de identificadores a referentes específicos, puede tener lugar en diferentes estadios o períodos en la evolución del sistema. La historia de tiempo de un sistema puede pensarse como una línea extendiéndose desde el momento de la escritura del código fuente hasta el momento de su ejecución. Dicha línea puede subdividirse en diferentes niveles de refinamiento según distintas combinaciones de computador/lenguaje/compilador/sistema operativo.

De esta forma, el ligado puede tener lugar cuando el programador escribe una sentencia en el editor de código fuente, cuando un módulo es compilado o ensamblado, cuando el código objeto (compilado o ensamblado) es procesado por el "link-editor" o el "link-loader" (generalmente este proceso es el conocido como ligado en la mayoría de los sistemas), cuando el código "imagen-de-memoria" es cargado en la memoria principal, y finalmente cuando el sistema es ejecutado.

La importancia del tiempo de ligado radica en que *cuando el valor de variables dentro de una pieza de código es fijado más tarde, el sistema es más fácilmente modificable y adaptable al cambio de requerimientos.*

Veamos un ejemplo: supongamos que se nos encomienda la escritura de una serie de programas listadores siendo la impresora a utilizar en principio una del tipo matricial de 80 columnas que funciona con papel continúo de 12" de largo de página.

Alternativas:

1. Escribimos el literal "72" en todas las rutinas de impresión de todos los programas. (ligado en tiempo de escritura)
2. Reemplazamos el literal por la constante manifiesta LONG_PAG a la que asignamos el valor "72" en todos los programas (ligado en tiempo de compilación)
3. Ponemos la constante LONG_PAG en un archivo de inclusión externo a los programas (ligado en tiempo de compilación)
4. Nuestro lenguaje no permite la declaración de constantes por lo cual definimos una variable global LONG_PAG a la que le asignamos el valor de inicialización "72" (ligado en tiempo de link-edición)
5. Definimos un archivo de parámetros del sistema con un campo LONG_PAG al cual se le asigna el valor "72". Este valor es leído junto con otros parámetros cuando el sistema se inicia. (ligado en tiempo de ejecución)

6. Definimos en el archivo de parámetros un registro para cada terminal del sistema y personalizamos el valor del campo LONG_PAG según la impresora que tenga vinculada cada terminal. De esta forma las terminales que tienen impresoras de 12" imprimen 72 líneas por página, y las que tienen una impresora de inyección de tinta que usan papel oficio, imprimen 80. (ligado en tiempo de ejecución)

Examinaremos ahora la relación existente entre el tiempo de ligado y las conexiones intermódulares, y como el mismo afecta el grado de acoplamiento entre módulos.

Nuevamente, una referencia intermódular fijada a un referente u objeto específico en tiempo de definición, tendrá un acoplamiento mayor a una referencia fijada en tiempo de traslación o posterior aún.

La posibilidad de compilación independiente de un módulo de otros facilitará el mantenimiento y modificación del sistema, que si debiera compilarse todos los módulos juntos. Igualmente, si la link-edición de los módulos es diferida hasta el instante previo a su ejecución, la implementación de cambios se verá simplificada.

Existe un caso particular de acoplamiento de módulos derivado de la *estructura lexicográfica* del programa. Hablamos en este caso de *acoplamiento por contenido.*

Dos formas de acoplamiento por contenido pueden distinguirse (Ver figura 6.1):

- Inclusión lexicográfica: se da cuando un módulo está incluido lexicográficamente en otro, y es una forma menor de acoplamiento. Los módulos por lo general no pueden ejecutarse separadamente. Este es el caso en el que el módulo subordinado es activado en línea dentro del contexto del módulo superior.
- Solapamiento parcial: es un caso extremo de acoplamiento por contenido. Parte del código de un módulo está en intersección con el otro. Afortunadamente la mayoría de los lenguajes modernos de alto nivel no permiten este tipo de estructuras.

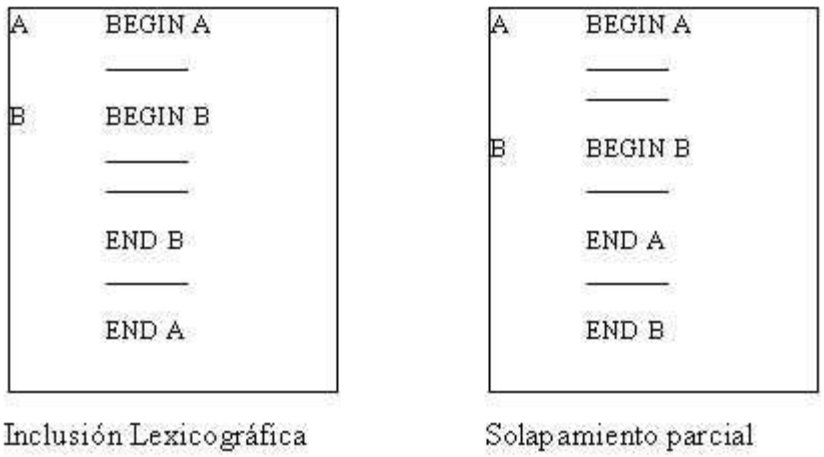


Figura 6.1

En términos de uso, mantenimiento, y modificación, las consecuencias del acoplamiento por contenido son peores que las del acoplamiento de control. El acoplamiento por contenido hace que los módulos no puedan funcionar uno sin el otro. No ocurre lo mismo en el acoplamiento de control, en el cual un módulo, aunque reciba información de control, puede ser invocado desde diferentes puntos del sistema.

ACOPLAMIENTO DE ENTORNO COMÚN (COMMON-ENVIRONMENT COUPLING)

Siempre que dos o más módulos interactúan con un entorno de datos común, se dice que dichos módulos están en *acoplamiento por entorno común.*

Ejemplos de entorno común pueden ser áreas de datos globales como la DATA división del COBOL o un archivo en disco.

El acoplamiento de entorno común es una forma de acoplamiento de segundo orden, distinto de los tratados anteriormente. La severidad del acoplamiento dependerá de la cantidad de módulos que acceden simultáneamente al entorno común. En el caso extremo de solo dos módulos donde uno utiliza como entrada los datos generados por el otro hablaremos de un acoplamiento de *entrada-salida*.

El punto es que el acoplamiento por entorno común no es necesariamente malo y deba ser evitado a toda costa. Por el contrario existen ciertas circunstancias en que es una opción válida.

El acoplamiento se clasifica en:

- Acoplamiento normal por datos
- Acoplamiento normal por estampado
- Acoplamiento normal por empaquetado
- Acoplamiento normal de control
- Acoplamiento hibrido
- Acoplamiento común
- Acoplamiento patológico (por contenido)

Favorable

Desfavorable

Acoplamiento normal por datos. Toda conexión se realiza explícitamente por el mínimo número de parámetros, siendo los datos del tipo primitivo o elemental.

Acoplamiento normal por estampado. Se presenta cuando las piezas de datos que se intercambian son compuestas como estructuras o arreglos. Cuando se desea establecer el nivel de acoplamiento de un módulo, cada uno de los datos compuestos es contabilizado como uno, y no el número de elementos que componen la estructura.

Acoplamiento normal por empaquetado. Se agrupan en una estructura elementos que no están relacionados, con el único propósito de reducir el número de parámetros. Al momento de calificar el nivel de acoplamiento normal, cada uno de los elementos de los datos compuestos es contabilizado en uno.

Acoplamiento normal de control. Un módulo intercambia con otro modulo información que desea alterar la lógica interna del otro modulo. La información indicará expresamente la acción que debe realizar el otro módulo.

Acoplamiento hibrido. Los módulos intercambian información siendo diferente para el modulo llamador y el modulo llamado. Para el modulo llamado, el parámetro es visto de control y para el modulo llamador el parámetro es visto como un dato.

Acoplamiento común. Dos módulos intercambian información mediante variables globales, ya que una variable global no está protegida en ningún modulo, cualquier porción de código puede modificar el valor de esa variable haciendo que el comportamiento del módulo sea impredecible

Acoplamiento patológico. Dos módulos tienen la posibilidad de afectar datos de otro a través de errores en la programación. Esta situación puede darse, por ejemplo, cuando un módulo escribe los datos de otro modulo, por ejemplo en el mal uso de apuntadores o el uso explícito de memoria para modificar variables.

Hemos visto que la determinación de módulos en un sistema no es arbitraria. La manera en la cual dividimos físicamente un sistema en piezas (particularmente en relación con la estructura del problema) puede afectar significativamente la complejidad estructural del sistema resultante, así como el número total de referencias intermódulares.

Adaptar el diseño del sistema a la estructura del problema (o estructura de la aplicación, o dominio del problema) es una filosofía de diseño sumamente importante. A menudo encontramos que elementos de procesamiento del dominio de problemas altamente relacionados, son trasladados en código altamente interconectado. Las estructuras que agrupan elementos del problema altamente interrelacionados, tienden a ser modularmente efectivas.

COHESIÓN

Imaginemos que tengamos una magnitud para medir el grado de relación funcional existente entre pares de módulos. En términos de tal medida, diremos que el sistema más modularmente efectivo será aquel cuya suma de relación funcional entre pares de elementos que pertenezcan a diferentes módulos sea mínima. Entre otras cosas, esto tiende a minimizar el número de conexiones intermódulares requeridas y el acoplamiento intermódular.

Esta relación funcional intramódular se conoce como *cohesión*.

> La cohesión es la medida cualitativa de cuan estrechamente relacionados están los elementos internos de un módulo.

Otros términos utilizados frecuentemente son "fuerza modular", "ligazón", y "funcionalidad".

En la práctica un elemento de procesamiento simple aislado, puede estar funcionalmente relacionado en diferentes grados a otros elementos. Como consecuencia, diferentes diseñadores, con diferentes "visiones" o interpretaciones de un mismo problema, pueden obtener diferentes estructuras modulares con diferentes niveles de cohesión y acoplamiento. A esto se suma el inconveniente de que muchas veces es difícil evaluar el grado de relación funcional de un elemento respecto de otro.

La cohesión modular puede verse como el cemento que amalgama juntos a los elementos de procesamiento dentro de un mismo módulo. Es el factor más crucial en el diseño estructurado, y el de mayor importancia en un diseño modular efectivo.

Este concepto representa la técnica principal que posee un diseñador para mantener su diseño lo más semánticamente próximo al problema real, o dominio de problema.

Claramente los conceptos de cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica un menor de acoplamiento. Maximizar el nivel de cohesión intramódular en todo el sistema resulta en una minimización del acoplamiento intermódular.

Matemáticamente el cálculo de la relación funcional intramódular (cohesión), involucra menos pares de elementos a los cuales debe aplicarse la medida, en comparación con el cálculo de la relación funcional intermódular (acoplamiento).

Ambas medidas son excelentes herramientas para el diseño modular efectivo, pero de las dos la más importante y extensiva es la cohesión.

Una cuestión importante a determinar es *como* reconocer la relación funcional.

El principio de cohesión puede ponerse en práctica con la introducción de la idea de un *principio asociativo*

En la decisión de poner ciertos elementos de procesamiento en un mismo módulo, el diseñador, utiliza el principio de que ciertas *propiedades* o *características* relacionan a los elementos que las poseen. Esto es, el diseñador pondrá el objeto Z en el mismo módulo que X e Y, porque X, Y, y Z poseen un misma propiedad. De esta manera, el principio asociativo es *relacional*, y es usualmente verificable en tales términos (p. e. "es correcto poner Z junto a X e Y, porque tiene la misma propiedad que ellos") o en términos de miembro de un conjunto (p. e."Es correcto poner Z junto a X e Y, pues todos pertenecen al mismo conjunto").

Debe tenerse en mente que la cohesión se aplica sobre todo el módulo, es decir sobre todos los pares de elementos. Así, si Z está relacionado a X e Y, pero no a A, B, y C, los cuales pertenecen al mismo módulo, la inclusión de Z en el módulo, redundará en baja cohesión del mismo.

Intencionalmente se ha usado el término "elemento de procesamiento" en esta discusión, en lugar de términos más comunes como instrucción o sentencia. Porqué:

Primero, un elemento de procesamiento puede ser algo que debe ser realizado en un módulo pero que aún no ha sido reducido a código. En orden de diseñar sistemas altamente modulares, debemos poder determinar la cohesión de módulos que todavía no existen.

Segundo, *elementos de procesamiento* incluyen *todas* las sentencias que aparecen en un módulo, no solo el procesamiento realizado por las instrucciones ejecutadas dentro de dicho módulo, sino también las que resultan de la invocación de subrutinas.

Por ejemplo, las sentencias individuales encontradas en el módulo B, el cual es invocado desde el módulo A, *no* figuran dentro de la cohesión del módulo A. Sin embargo el procesamiento global (función) realizado por la llamada al módulo B, es claramente un elemento de procesamiento en el módulo llamador A, y por lo tanto participa en la cohesión del módulo A.

NIVELES DE COHESIÓN

Diferentes principios asociativos fueron desenvolviéndose a través de los años por medio de la experimentación, argumentos teóricos, y la experiencia práctica de muchos diseñadores.

Existen siete niveles de cohesión distinguibles por siete principios asociativos. Estos se listan a continuación en orden creciente del grado de cohesión, de menor a mayor relación funcional:

- Cohesión Casual (la peor)
- Cohesión Lógica (sigue a la peor)
- Cohesión Temporal (de moderada a pobre)
- Cohesión de Procedimiento (moderada)
- Cohesión de Comunicación (moderada a buena)
- Cohesión Secuencial
- Cohesión Funcional (la mejor)

Podemos visualizar el grado de cohesión como un espectro que va desde un máximo a un mínimo.

COHESIÓN CASUAL (LA PEOR)

La *cohesión casual* ocurre cuando existe poca o ninguna relación entre los elementos de un módulo.

La cohesión casual establece un punto cero en la escala de cohesión.

Es muy difícil encontrar módulos puramente casuales. Puede aparecer como resultado de la modularización de un programa ya escrito, en el cual el programador

encuentra un determinada secuencia de instrucciones que se repiten de forma aleatoria, y decide por lo tanto agruparlas en una rutina.

Otro factor que influenció muchas veces la confección de módulos casualmente cohesivos, fue la mala práctica de la programación estructurada, cuando los programadores mal entendían que modularizar consistía en cambiar las sentencias GOTO por llamadas a subrutinas

Finalmente diremos que si bien en la práctica es difícil encontrar módulos casualmente cohesivos en su totalidad, es común que tengan elementos casualmente cohesivos. Tal es el caso de operaciones de inicialización y terminación que son puestas juntas en un módulo superior.

Debemos notar que si bien la cohesión casual no es necesariamente perjudicial (de hecho es preferible un programa casualmente cohesivo a uno lineal), dificulta las modificaciones y mantenimiento del código.

COHESIÓN LÓGICA (SIGUE A LA PEOR)

Los elementos de un módulo están *lógicamente* asociados si puede pensarse en ellos como pertenecientes a la misma clase lógica de funciones, es decir aquellas que pueden pensarse como juntas lógicamente.

Por ejemplo, se puede combinar en un módulo simple todos los elementos de procesamiento que caen en la clase de "entradas", que abarca todas las operaciones de entrada.

Podemos tener un módulo que lea desde consola una tarjeta con parámetros de control, registros con transacciones erróneas de un archivo en cinta, registros con transacciones válidas de otro archivo en cinta, y los registros maestros anteriores de un archivo en disco. Este módulo que podría llamarse "Lecturas", y que agrupa todas las operaciones de entrada, es lógicamente cohesivo.

La cohesión lógica es más fuerte que la casual, debido a que representa un mínimo de asociación entre el problema y los elementos del módulo. Sin embargo podemos ver que un módulo lógicamente cohesivo no realiza una función específica, sino que abarca una serie de funciones.

COHESIÓN TEMPORAL (DE MODERADA A POBRE)

Cohesión temporal significa que todos los elementos de procesamiento de una colección ocurren en el mismo período de tiempo durante la ejecución del sistema. Debido a que dicho procesamiento debe o puede realizarse en el mismo período de tiempo, los elementos asociados temporalmente pueden combinarse en un único módulo que los ejecute a la misma vez.

Existe una relación entre cohesión lógica y la temporal, sin embargo, la primera no implica una relación de tiempo entre los elementos de procesamiento. La cohesión temporal es más fuerte que la cohesión lógica, ya que implica un nivel de relación más: el factor tiempo. Sin embargo la cohesión temporal aún es pobre en nivel de cohesión y acarrea inconvenientes en el mantenimiento y modificación del sistema.

Un ejemplo común de cohesión temporal son las rutinas de inicialización (start-up) comúnmente encontradas en la mayoría de los programas, donde se leen parámetros de control, se abren archivos, se inicializan variables contadores y acumuladores, etc.

COHESIÓN DE PROCEDIMIENTO (MODERADA)

Elementos de procesamiento relacionados *proceduralmente* son elementos de una unidad procedural común. Estos se combinan en un módulo de cohesión procedural. Una unidad procedural común puede ser un proceso de iteración (loop) y de decisión, o una secuencia linear de pasos. En este último caso la cohesión es baja y es similar a la cohesión temporal, con la diferencia que la cohesión temporal no implica una determinada secuencia de ejecución de los pasos.

Al igual que en los casos anteriores, para decir que un módulo tiene *solo* cohesión procedural, los elementos de procesamiento deben ser elementos de alguna iteración, decisión, o secuencia, pero no deben estar vinculados con ningún principio asociativo de orden superior.

La cohesión procedural asocia elementos de procesamiento sobre la base de sus relaciones algorítmicas o procedurales.

Este nivel de cohesión comúnmente se tiene como resultado de derivar una estructura modular a partir de modelos de procedimiento como ser diagramas de flujo, o diagramas Nassi-Shneiderman (Ver figura 6.2).

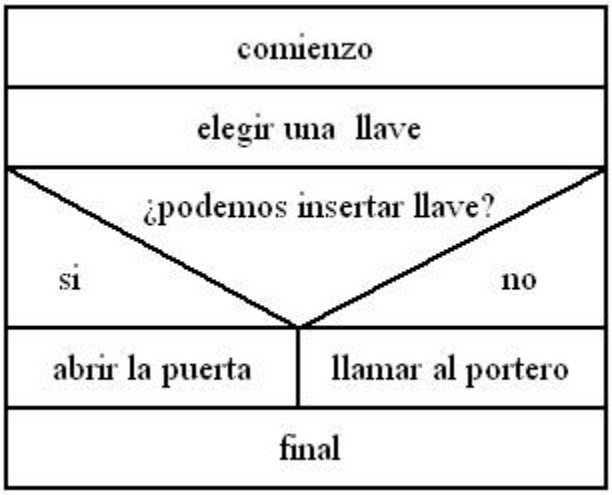


Figura 6.2

COHESIÓN DE COMUNICACIÓN (MODERADA A BUENA)

Ninguno de los niveles de cohesión discutidos previamente está fuertemente vinculado a una estructura de problema en particular. *Cohesión de Comunicación* es el menor nivel en el cual encontramos una relación entre los elementos de procesamiento que es intrínsecamente *dependiente del problema*.

Decir que un conjunto de elementos de procesamiento están vinculados por comunicación significa que *todos los elementos operan sobre el mismo conjunto de datos* de entrada o de salida (Ver figura 6.3).

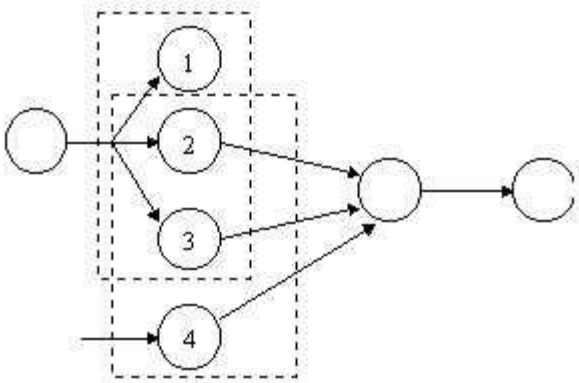


Figura 6.3

En el diagrama de la figura 6.3 podemos observar que los elementos de procesamiento 1, 2, y 3, están asociados por comunicación sobre la corriente de datos de entrada, en tanto que 2, 3, y 4 se vinculan por los datos de salida.

Los diagramas de flujo de datos (DFD) son un medio objetivo para determinar si los elementos en un módulo están asociados por comunicación.

Las relaciones por comunicación presentan un grado de cohesión aceptable.

La cohesión por comunicación es común en aplicaciones comerciales. Ejemplos típicos pueden ser

- Un módulo que imprima o grabe un archivo de transacciones
- Un módulo que reciba datos de diferentes fuentes, y los transforme y ensamble en una línea de impresión.

COHESIÓN SECUENCIAL

El siguiente nivel de cohesión en la escala es la asociación *secuencial*. En ella, los datos de salida (resultados) de un elemento de procesamiento sirven como datos de entrada al siguiente elemento de procesamiento.

En términos de un diagrama de flujo de datos de un problema, la cohesión secuencial combina una cadena lineal de transformaciones sucesivas de datos.

Este es claramente un principio asociativo relacionado con el dominio del problema.

COHESIÓN FUNCIONAL (LA MEJOR)

En el límite superior del espectro de relación funcional encontramos la *cohesión funcional*. En un módulo completamente funcional, cada elemento de procesamiento, es parte integral de, y esencial para, la realización de una función simple.

En términos prácticos podemos decir que cohesión funcional es aquella que no es secuencial, por comunicación, por procedimiento, temporal, lógica, o casual.

Los ejemplos más claros y comprensibles provienen del campo de las matemáticas. Un módulo para realizar el cálculo de *raíz cuadrada* ciertamente será altamente cohesivo, y probablemente, completamente funcional. Es improbable que haya elementos superfluos más allá de los absolutamente esenciales para realizar la función matemática, y es improbable que elementos de procesamiento puedan ser agregados sin alterar el cálculo de alguna forma.

En contraste un módulo que calcule raíz cuadrada y coseno, es improbable que sea enteramente funcional.

En adición a estos ejemplos matemáticos obvios, usualmente podemos reconocer módulos funcionales que son elementales en naturaleza. Un módulo llamado LEER-REGISTRO-MAESTRO, o TRATAR-TRANS-TIPO3, presumiblemente serán funcionalmente cohesivos, en cambio TRATAR-TODAS-TRANS presumiblemente realizará más de una función y será lógicamente cohesivo.

CRITERIOS PARA ESTABLECER EL GRADO DE COHESIÓN

Una técnica útil para determinar si un módulo está acotado funcionalmente es escribir una frase que describa la función (propósito) del módulo y luego examinar dicha frase. Puede hacerse la siguiente prueba:

1. Si la frase resulta ser una sentencia compuesta, contiene una coma, o contiene más de un verbo, probablemente el módulo realiza más de una función; por tanto, probablemente tiene vinculación secuencial o de comunicación.
2. Si la frase contiene palabras relativas al tiempo, tales como "primero", "a continuación", "entonces", "después", "cuando", "al comienzo", etc., entonces probablemente el módulo tiene una vinculación secuencial o temporal.
3. Si el predicado de la frase no contiene un objeto específico sencillo a continuación del verbo, probablemente el módulo esté acotado lógicamente. Por ejemplo *editar todos los datos* tiene una vinculación lógica; *editar sentencia fuente* puede tener vinculación funcional.
4. Palabras tales como "inicializar", "limpiar", etc., implican vinculación temporal.

Los módulos acotados funcionalmente siempre se pueden describir en función de sus elementos usando una sentencia compuesta. Pero si no se puede evitar el lenguaje anterior, siendo aún una descripción completa de la función del módulo, entonces probablemente el módulo no esté acotado funcionalmente.

Es importante notar que no es necesario determinar el nivel preciso de cohesión. En su lugar, lo importante es intentar conseguir una cohesión alta y saber reconocer la cohesión baja, de forma que se pueda modificar el diseño del software para que disponga de una mayor independencia funcional.

MEDICIÓN DE COHESIÓN

Cualquier módulo, rara vez verifica un solo principio asociativo. Sus elementos pueden estar relacionados por una mezcla de los siete niveles de cohesión. Esto lleva a tener una escala continua en el grado de cohesión más que una escala con siete puntos discretos.

Donde existe más de una relación entre un par de elementos de procesamiento, se aplica el máximo nivel que alcanzan. Por esto, si un módulo presenta cohesión lógica entre *todos* sus pares de elementos de procesamiento, y a su vez presenta cohesión de comunicación también entre *todos* dichos pares, entonces dicho módulo es considerado como de cohesión por comunicación.

Ahora, ¿cuál sería la cohesión de dicho módulo si también contiene algún par de elementos completamente no relacionados? En teoría, debería tener algún tipo de promedio entre la cohesión de comunicación y la casual. Para propósitos de depuración, mantenimiento, y modificación, un módulo se comporta como si fuera "**solo tan fuerte como sus vínculos más débiles**".

El efecto sobre los costos de programación es próximo al menor nivel de cohesión aplicable dentro del módulo en vez del mayor nivel de cohesión.

> *La cohesión de un módulo es aproximada al nivel más alto de cohesión que es aplicable a todos los elementos de procesamiento dentro del módulo.*

Un módulo puede consistir de varias funciones *completas* relacionadas lógicamente. Esto es definitivamente más cohesivo que un módulo que liga lógicamente fragmentos de varias funciones.

La decisión de que nivel de cohesión es aplicable a un módulo dado requiere de cierto juicio humano. Algunos criterios establecidos son:

- La cohesión secuencial es más próxima al óptimo funcional que a su antecesor de comunicación.
- Similarmente existe un salto mayor entre la cohesión lógica y la temporal que entre casual y lógica.

La obligación del diseñador es conocer los efectos producidos por la variación en la cohesión, especialmente en términos de modularidad, en orden de realizar soluciones de *compromiso* beneficiando un aspecto en contra de otro.

En conclusión se puede decir:

UNA BUENA MODULARIDAD SE OBTIENE SI EXISTE UNA ALTA COHESIÓN Y UN BAJO ACOPLAMIENTO

PASO DE PARÁMETROS

Las ligaduras de los parámetros tiene un efecto significativo en la semántica de las llamadas a procedimientos, los lenguajes difieren de manera sustancial de las clases de mecanismos como: paso de parámetros disponibles y el rango de los efectos permisibles de implementación que pudieran ocurrir. Algunos lenguajes ofrecen sólo una clase básica de mecanismo de paso de parámetros, mientras que otros pueden ofrecer dos o más. Los pases de parámetros más conocidos son: Paso de parámetros por valor y paso de parámetros por referencia.

Paso de parámetros por valor. El paso de parámetros por valor consiste en copiar el contenido de la variable que queremos pasar al ámbito local de la función llamada, consiste en copiar el contenido de la memoria del argumento que se quiere pasar a otra dirección de memoria, correspondiente al argumento dentro del ámbito de dicha función. Se tendrán dos valores duplicados e independientes, con lo que la modificación de uno no afecta al otro.

Paso de parámetros por referencia. El paso de parámetros por referencia consiste en proporcionar a la función llamada a la que se le quiere pasar el argumento la dirección de memoria del dato. Con este mecanismo un argumento debe ser en principio una variable con una dirección asignada. En vez de pasar el valor de la variable, el paso por referencia pasa la ubicación de la variable de modo que el parámetro se convierte en un **alias** para el argumento, y cualquier cambio que se le haga a éste lo sufre también el argumento. Los lenguajes que permiten el paso de parámetros se define utilizando sintaxis adicional. Por ejemplo, en Pascal:

```
procedure inc(var x : integer);
begin
    x := x+1;
end;
```

En este ejemplo, la variable –x- es sólo un alias. Pascal incluye como sintaxis adicional la palabra clave –var- indicando que el paso de parámetro es por referencia.

Note que el paso de parámetros por valor no implica que no puedan ocurrir cambios fuera del procedimiento mediante el uso de parámetros. Si el parámetro tiene un tipo de apuntador o referencia, entonces el valor es una dirección, y puede utilizarse para cambiar la memoria por fuera del procedimiento. Por ejemplo, la siguiente función en C cambia de manera definitiva el valor del entero a la cual el parámetro –p- apunta:

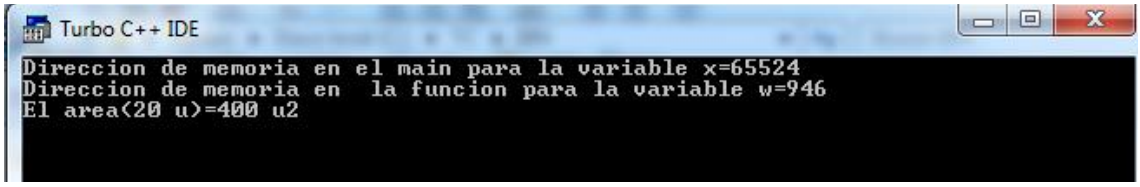
```
void int_p(int* p){
*p=0;
```

```
}
```

En el lenguaje C, los argumentos de los tipos de variables escalares son pasados por valor. No es posible cambiar el valor de una función llamadora dentro de una función llamada que esté en ejecución, al menos que se tenga la dirección de la variable de la función llamadora que se desee modificar.

Observemos el programa 6.1:

```
#include <stdio.h>
int area_cuadrado(int);
main(){
    int x,y;
    x=20;
    printf("Direccion de memoria en el main para la variable x=%u\n",&x);
    y=area_cuadrado(x); //función llamadora
    printf("El area(%d u)=%d u2\n",x,y);
    getchar();

}
int area_cuadrado(int w){  //función llamada
        printf("Direccion de memoria en  la funcion para la variable w=%u\n");
        return(w*w);
}
```



Programa 6.1

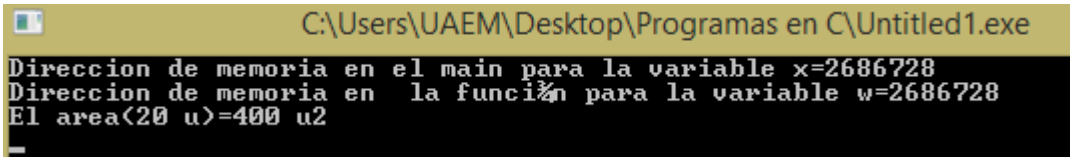
Este programa contiene dos funciones: -main()- y -area_cuadrado()-, cada función tiene sus propias variables locales. Una variable local es aquella cuyo ámbito se restringe a la función que la ha declarado, esto implica que la variable local sólo va a poder ser manipulada en dicha sección, y no se podrá hacer referencia fuera de dicha sección. La función main() tiene las variables locales enteras –**x**- y –**y**-. La función area_cuadrado() tiene la variable local entera –**w**-.

La función area_cuadrado() es invocada dentro de la función main() (la función llamadora) y el paso de parámetro es por valor. Esto es**, w=x** (El valor contenido en la variable –**x**- será dado a la variable –**w**-)**.**

Observe que al ejecutarse el programa 6.1, la variable – **x** - tiene una dirección de memoria independiente a la variable –**w**- y la variable –**w**- sólo recibe una copia del valor guardado en la localidad de la variable- **x**-. (Observe que la función area_cuadrado(), dentro de su declaración indica que va a retornar un valor del tipo entero).
¿Qué sucede si en vez de enviar el valor de la variable se envía su dirección?

Observe el programa 6.2:

```
#include <stdio.h>
void area_cuadrado(int *);
main(){
    int x,y;
    y=x=20;
    printf("Direccion de memoria en el main para la variable x=%u\n",&x);
    area_cuadrado(&x); // función llamadora
    printf("El area(%d u)=%d u2\n",y,x);
    getchar();

}
void area_cuadrado(int *w){  // función llamada

        printf("Direccion de memoria a la que apunta la variable w=%u\n");

        *w=*w**w;

}
```



Programa 6.2

En el programa 6.2, como en el programa anterior, la función main() tiene las variables locales enteras – **x** - y – **y** -. La función area_cuadrado() tiene la variable local apuntador a entero –**w**-.

A diferencia del programa anterior, en este programa se envía como parámetro de la función llamadora a la dirección de memoria de la variable – **x** -, por lo que se tiene como argumento en la función llamada a un apuntador a la dirección de memoria de la variable que se utilizó como argumento en la función llamadora Esto es, **w=&x** (Recuerde que el lenguaje de programación C, para tipos de variables escalares, sólo acepta el pase de parámetros por valor siendo diferente el caso para otro tipo de variables). Esto es como darle permiso al argumento de la función

llamada a que modifique el dato guardado en la variable que se utilizó como argumento en la función llamadora (un alias). Esta es la forma en que el lenguaje de programación C simula el paso de parámetros por referencia para los tipos de variables escalares.

PASO DE PARÁMETROS EN VECTORES.

Como recordaremos, al declarar el nombre de un vector, lo que realmente se está haciendo es declarar un apuntador al inicio del arreglo. Por lo que al enviar un vector como parámetro se enviará la dirección de inicio del vector (en este caso, por las características del lenguaje C, para vectores y matrices sólo se acepta el paso de parámetros por referencia). En el programa 6.3, las funciones –imprime()- e -imprime1()- muestran las formas en las que se puede manejar un vector, como se observa se puede manejar como un vector o como un apuntador, es indistinto.

```
#include <stdio.h>
void imprime(int *);
void imprime1(int []);
main(){
int i ,a[]={1,2,3,4,5};
for (i=0;i<5;i++)
    printf("%u..",(a+i));
    putchar('\n');
imprime(a);
imprime1(a);
getchar();
}

void imprime(int b[]){
int  i;
for (i=0;i<5;i++)
     printf("%u..",(b+i));
putchar('\n');
    for (i=0;i<5;i++)
     printf("%d..",*(b+i));
putchar('\n');
}

void imprime1(int *c){
int j;
     for(j=0;j<5;j++)
          printf("%d..",c[j]);
```

}

```
65516..65518..65520..65522..65524..
65516..65518..65520..65522..65524..
1..2..3..4..5..
1..2..3..4..5..
```

Programa 6.3

La primera impresión se invoca en la función -main()- y muestra los cinco espacios de memoria donde se tiene guardado el vector, la segunda impresión se invoca en la función -imprime()-, observe que la dirección de memoria es la misma. Tanto la función -imprime() como la función imprime1()- muestran una forma indistinta de manejar los vectores (como apuntador o como vector). Observe que el pase de parámetros se realiza sólo con el nombre del vector a comparación de las variables escalares que al simular el paso de parámetros por referencia se tiene que utilizar el prefijo **"&"** en la función llamadora y el prefijo **'*'** en la función llamada,

Un detalle importante, el lenguaje C no verifica los acotamientos tanto inferior como superior, por lo que puede sobre escribir o puede mostrar basura. Observe el programa 6.4:

```
#include <stdio.h>
void imprime(int *);
main(){
int i ,a[]={1,2,3,4,5};
imprime(a);
getchar();
}

void imprime(int b[]){
        int i ;
    for (i=-3;i<8;i++)
       printf("%u..",(b+i));
   putchar('\n');
   for (i=-3;i<8;i++)
       printf("%d..",*(b+i));
   putchar('\n');
}
```

```
65510..65512..65514..65516..65518..65520..65522..65524..65526..65528..65530..
-10..686..-20..1..2..3..4..5..0..344..0..
```

Programa 6.4

En este programa se está declarando e inicializando a un vector con cinco elementos, observe que el comando ”for” se ejecuta con una inicialización de -3 hasta 7 (se desborda tanto en la izquierda como en la derecha) mostrando la dirección de memoria y los valores guardados en cada una de esas direcciones:

Al mostrar el contenido del vector, los primeros tres y los últimos tres valores mostrados en pantalla son basura pudiendo suceder conflictos.

PASO DE PARÁMETROS EN MATRICES ESTÁTICAS.

Un arreglo multidimensional puede ser visto en varias formas en el lenguaje de programación C, por ejemplo:

Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos en sí mismo es un arreglo.

Por lo tanto, la notación:

A[n][m]

Nos indica que los elementos del arreglo están guardados renglón por renglón.
Cuando se pasa un arreglo bidimensional a una función se debe especificar el número de columnas ya que el número de renglones es irrelevante.
La razón de lo anterior, es nuevamente los apuntadores, C requiere conocer cuántas son las columnas para que pueda brincar de renglón en renglón en la memoria.

Considerando que una función deba recibir como argumento o parámetro una variable *int a[5][35*] se puede declarar el argumento de la función llamada como:

```
f( int a[][35]){ . . . }
```

o aún como:

```
f( int (*a)[35]){  . . . }
```

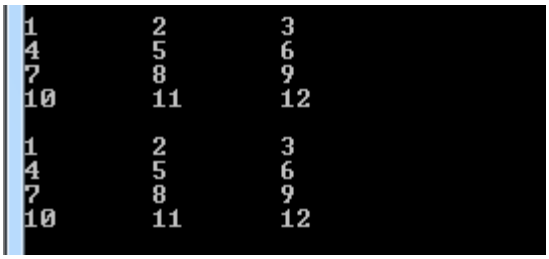
En el último ejemplo se requieren los paréntesis -(*a)- porque el operador -[]- tiene precedencia sobre el operador -*-.

Por lo tanto:

*int (*a)[35]* declara un apuntador a un arreglo de 35 enteros, y por ejemplo, si hacemos la siguiente referencia *a+2,* nos estamos refiriendo a la dirección del primer elemento que se encuentra en el tercer renglón de la matriz supuesta, mientras que *int *a[35];* está declarando un arreglo de 35 apuntadores a enteros. Observe el programa 6.5:

```
#include <stdio.h>
#define H      4
void matriz(int [][3]);
```

```
void matriz1(int (*)[3]);
main(){
    int a[][3]={{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    matriz (a);
    putchar('\n');
    matriz1(a);
    getchar();

}

void matriz(int (*b)[3]){
int i,j;
for (i=0;i<H;i++){
    for (j=0;j<3;j++)
        printf("%d\t",(*(b+i))[j]);
     putchar('\n');
}
}


void matriz1(int b[][3]){
int i,j;
for (i=0;i<H;i++){
    for (j=0;j<3;j++)
        printf("%d\t",b[i][j]);

     putchar('\n');

   }
}
```

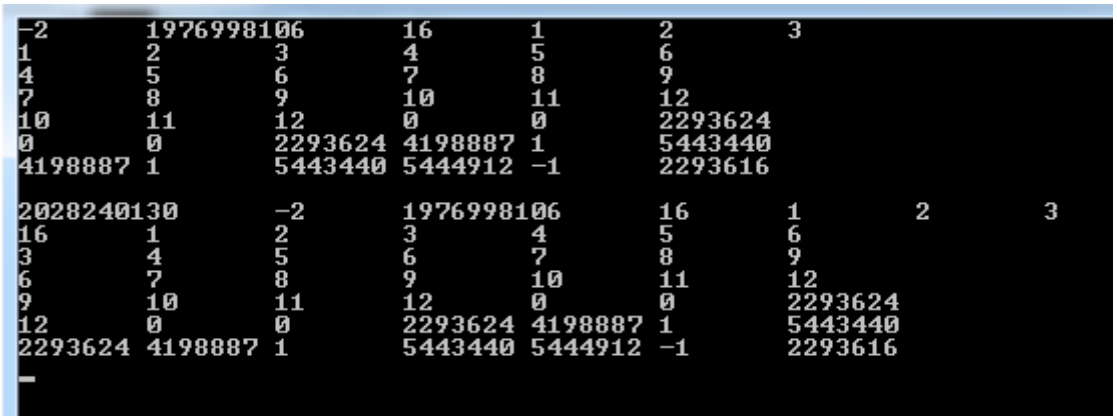


Programa 6.5

En este programa las funciones matriz() y matriz1() se emplean indistintamente la notación tipo apuntador como la notación matricial.

Al igual que en vectores, en matrices C no cuida el límite inferior ni el superior. Por ejemplo, en el programa 6.6 se muestra la basura que se tiene en memoria cuando la matriz rebasa tanto su límite inferior como su límite superior:

```
#include <stdio.h>
#define H      4
void matriz(int [][3]);
void matriz1(int (*)[3]);
main(){
    int a[][3]={{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    matriz (a);
    putchar('\n');
    matriz1(a);
    getchar();
}

void matriz(int (*b)[3]){
int i,j;
for (i=-1;i<6;i++){
    for (j=0;j<6;j++)
        printf("%d\t",(*(b+i))[j]);
      putchar('\n');
}
}

void matriz1(int b[][3]){
int i,j;
for (i=-1;i<6;i++){
    for (j=-1;j<6;j++)
        printf("%d\t",b[i][j]);
      putchar('\n');
}
}
```



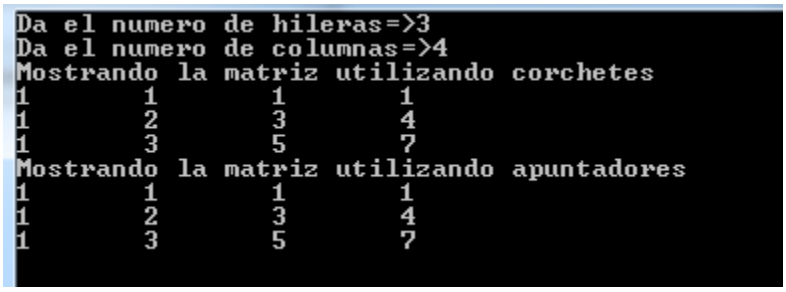
Programa 6.6

PASO DE PARÁMETROS CON MATRICES DINÁMICAS.

Si el usuario conoce el tamaño de la matriz hasta el momento de la ejecución, se puede utilizar la memoria en forma dinámica. El argumento de la matriz se manejará en la función llamada como un apuntador doble, además se debe de incluir el número de hileras y el número de columnas para que la función llamada pueda manejar en forma adecuada la dimensión de la matriz. Estos elementos se muestran en el programa 6.7:

```
#include <stdio.h>
#include <stdlib.h>
void matriz(int **,int,int);
main(){
    int f,c,i,j;
    int **pm;
    printf("Da el numero de hileras=>");
    scanf("%d",&f);
    getchar();
    printf("Da el numero de columnas=>");
    scanf("%d",&c);
    pm=(int **)malloc(sizeof(int *)*f);
    for (j=0;j<c;j++)
        pm[j]=(int*)malloc(sizeof(int)*c);
     for (i=0;i<f;i++)
        for (j=0;j<c;j++)
                pm[i][j]=i*j+1;
    matriz(pm,f,c);
    getchar();
    getchar();
```

```
}

void matriz(int **b, int hil, int col){
   int i,j;
    printf("Mostrando la matriz utilizando corchetes\n");
    for (i=0;i<hil;i++){
        for (j=0;j<col;j++)
                printf("%d\t",b[i][j]);
        putchar('\n');
 }
    printf("Mostrando la matriz utilizando apuntadores\n");
    for (i=0;i<hil;i++){
        for (j=0;j<col;j++)
                printf("%d\t",*(*(b +i)+j));
        putchar('\n');
 }
}
```



Programa 6.7

En la función -matriz()- se emplea indistintamente la notación tipo apuntador como la notación matricial.

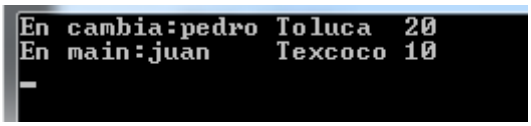
Un punto interesante de comentar es que, en algunos compiladores el programa compila bien pero no ejecuta en forma apropiada (el programa ejecutó bien en dev-c++, en Turbo C y en Turbo C portable, no siendo así en Borland C).

PASO DE PARÁMETROS EN REGISTROS.

Si se desea enviar como parámetro sólo un registro, el paso parámetro será por valor. Por ejemplo, observe el programa 6.8:

```
#include <stdio.h>
#include <string.h>
struct alumno{
      char a[10];
      char b[10];
```

```
	int c;
	};
void cambia(alumno );
main(){
	alumno alumnos;
	strcpy(alumnos.a,"juan");
	strcpy(alumnos.b,"Texcoco");
	alumnos.c=10;
	cambia(alumnos);
	printf("En main:%s\t%s\t%d\n",alumnos.a,alumnos.b,alumnos.c);
	getchar();
	}

void cambia(alumno b){
    strcpy(b.a,"pedro");
    strcpy(b.b,"Toluca");
    b.c=20;
    printf("En cambia:%s\t%s\t%d\n",b.a,b.b,b.c);
    }
```



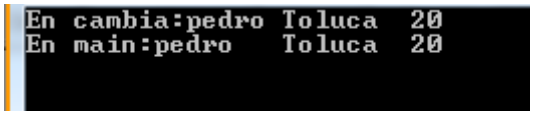
Programa 6.8

Por lo que los cambios realizados dentro de la función -cambia()- sólo se consideran en forma local.

Para poder realizar paso de parámetros por referencia se tiene que realizar el envío de la dirección de memoria en forma explícita con el operador -&- y la recepción del parámetro se realizará con el operador -*-. Observe el programa 6.9:

```
#include <stdio.h>
#include <string.h>
struct alumno{
	char a[10];
	char b[10];
	int c;
	};
void cambia(alumno *);
main(){
	alumno alumnos;
```

```
      strcpy(alumnos.a,”juan”);
      strcpy(alumnos.b,”Texcoco”);
      alumnos.c=10;
      cambia(&alumnos);
      printf(“En main:%s\t%s\t%d\n”,alumnos.a,alumnos.b,alumnos.c);
      getchar();
      }
void cambia(alumno *b){
    strcpy(b->a,”pedro”);
    strcpy(b->b,”Toluca”);
    b->c=20;
    printf(“En cambia:%s\t%s\t%d\n”,b->a,b->b,b->c);
    }
```

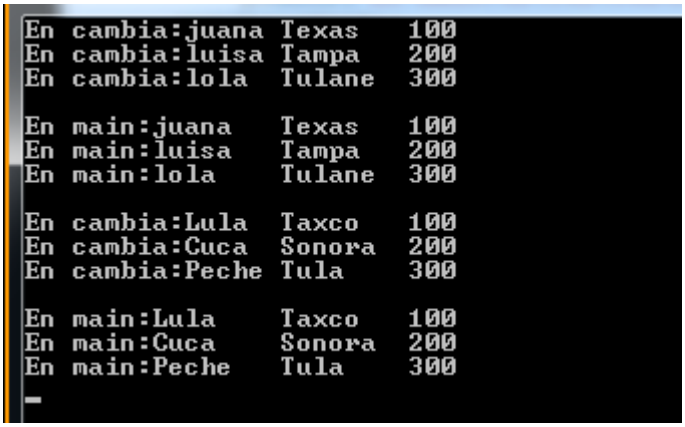


Programa 6.9

Si se desea enviar como parámetro un arreglo de registros se procede como se muestra en el programa 6.10:

```
#include <stdio.h>
#include <string.h>
struct alumno{
      char a[10];
      char b[10];
      int c;
  };

void cambia(alumno *);
void cambia1(alumno[]);
main(){
      int i;
      alumno alumnos[3];
      strcpy(alumnos[0].a,"juan"); strcpy(alumnos[0].b,"Texcoco"); alumnos[0].c=10;
      strcpy(alumnos[1].a,"luis"); strcpy(alumnos[1].b,"Tepexpan"); alumnos[1].c=20;
      strcpy(alumnos[2].a,"lucas"); strcpy(alumnos[2].b,"Tocuila"); alumnos[2].c=10;
      cambia(alumnos);
      putchar('\n');
      for (i=0;i<3;i++)
```

```
        printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
    putchar('\n');
    cambia1(alumnos);
    putchar('\n');
    for (i=0;i<3;i++)
        printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
    getchar();
}

void cambia(alumno b[]){
    int i;
    strcpy(b[0].a,"juana"); strcpy(b[0].b,"Texas"); b[0].c=100;
    strcpy((*(b+1)).a,"luisa"); strcpy((*(b+1)).b,"Tampa"); (*(b+1)).c=200;
    strcpy((b+2)->a,"lola"); strcpy((b+2)->b,"Tulane"); (b+2)->c=300;
    for (i=0;i<3;i++)
        printf("En cambia:%s\t%s\t%d\n",(*(b+i)).a,(b+i)->b,b[i].c);
}

 void cambia1(alumno *c){
    int i;
    strcpy(c[0].a,"Lula"); strcpy(c[0].b,"Taxco"); c[0].c=100;
    strcpy((*(c+1)).a,"Cuca"); strcpy((*(c+1)).b,"Sonora"); (*(c+1)).c=200;
    strcpy((c+2)->a,"Peche"); strcpy((c+2)->b,"Tula"); (c+2)->c=300;
    for (i=0;i<3;i++)
        printf("En cambia:%s\t%s\t%d\n",(*(c+i)).a,(c+i)->b,c[i].c);
}
```



Programa 6.10

En este programa, las funciones -cambia()- y -cambia1()- muestran las dos formas validas del paso de parámetro para un arreglo de registros. Observe que se maneja en forma indistinta la notación de apuntadores como la notación de arreglos.

En el siguiente capítulo se mostrará la forma en que se utilizan los registros con manejo de memoria dinámica.

APUNTADORES A FUNCIONES.

Para hablar de apuntadores a funciones, previamente se establece una dirección a la función.

Cuando se declara una matriz se asume que la dirección es la del primer elemento, del mismo modo, se asume que la dirección de una función será la del segmento de código donde comienza la función. Es decir, la dirección de memoria a la que se transfiere el control cuando se invoca (su punto de comienzo).

Técnicamente un apuntador a función es una variable que guarda la dirección de comienzo de la función. La mejor manera de pensar en el apuntador a función es considerándolo como una especie de "alias" de la función, aunque con una importante cualidad añadida: que **pueden ser utilizados como argumento de otras funciones.**

Por lo que un apuntador a función es un artificio que el lenguaje C utiliza para poder enviar funciones como argumento de una función, la gramática del lenguaje C no permite en principio utilizar funciones en la declaración de parámetros. Un punto importante***, no está permitido hacer operaciones aritméticas con este tipo de apuntador*.**

Lo anterior es útil cuando se deben usar distintas funciones, quizás para realizar tareas similares con los datos. Por ejemplo, se pueden pasar como parámetros los datos y la función que serán usados por alguna función de control. La biblioteca estándar de C tiene funciones para ordenamiento (*qsort()*) y para realizar búsqueda (*bsearch()*), a las cuales se les pueden pasar funciones como parámetros.

La declaración de un apuntador a función se realiza de la siguiente forma:

```
int (* pf)();
```

Donde –**pf**- es un apuntador a una función que no envía parámetros y retorna un tipo de dato entero. Observe que se ha declarado el apuntador y en este momento no se ha dicho a qué variable va a apuntar.

Supongamos que se tiene una función –*int f();*- entonces simplemente se debe de escribir:

```
pf= f;
```

Para que la variable –pf- apunte al inicio de la función -f()-.

Los apuntadores a funciones se declaran en el área de prototipos, por ejemplo:

```
int  f(int);
int (*fp) (int) =f;
```

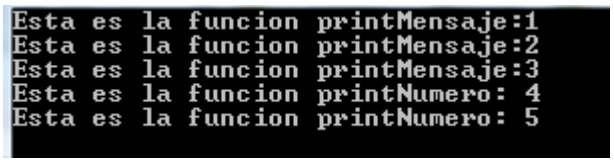
En consecuencia, el cuerpo del programa se pueden tener asignaciones como:

```
ans=f(5);
ans=pf(5);
```

Los cuales son equivalentes.

Observe el programa 6.11:

```
#include <stdio.h>
#include <conio.h>
void printMensaje(int dato );
void printNumero(int);
void (*funcPuntero)(int);
main(){
      //clrscr();
      printMensaje(1);
      funcPuntero=printMensaje;
      funcPuntero(2);
      funcPuntero(3);
      funcPuntero=printNumero;
      funcPuntero(4);
      printNumero(5);
      getchar();
}
void printMensaje(int dato){
    printf("Esta es la funcion printMensaje:%d\n",dato);
    }
void printNumero(int dato){
    printf("Esta es la funcion printNumero: %d\n",dato);
    }
```



Programa 6.11

En este programa hemos declarado dos funciones, -printMensaje()- y -printNumero()-, y después -(*funcPuntero)-, que es un apuntador a una función que recibe un parámetro entero y no devuelve nada (void). Las dos funciones declaradas anteriormente se ajustan precisamente a este perfil, y por tanto pueden

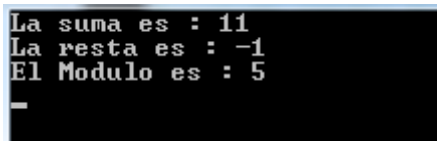
ser llamadas por este apuntador.
En la función principal, llamamos a la función –printMensaje()- con el valor uno como parámetro, en la línea siguiente asignamos al puntero a función (*funcPuntero) el valor de –printMensaje()- y utilizamos el apuntador para llamar a la misma función de nuevo. Por tanto, las dos llamadas a la función –printMensaje()- son idénticas gracias a la utilización del puntero -(*funcPuntero)-.
Dado que hemos asignado el nombre de una función a un apuntador a función, y el compilador no da error, el nombre de una función debe ser un apuntador a una función. Esto es exactamente lo que sucede. Un nombre de una función es un apuntador a esa función, pero es un apuntador constante que no puede ser cambiado. Sucede lo mismo con los vectores: el nombre de un vector es un apuntador constante al primer elemento del vector.

El nombre de una función es un apuntador a esa función, podemos asignar el nombre de una función a un apuntador constante, y usar el apuntador para llamar a la función. Pero el valor devuelto, así como el número y tipo de parámetros deben ser idénticos. Muchos compiladores de C y C++ no avisan de las diferencias entre las listas de parámetros cuando se hacen las asignaciones. Esto se debe a que las asignaciones se hacen en tiempo de ejecución, cuando la información de este tipo no está disponible para el sistema.

Veamos el código del programa 6.12:

```
#include <stdio.h>
#include <conio.h>
int MiFuncionSuma(int,int);
int MiFuncionResta(int,int);
int MiFuncionModulo(int,int);
main(){
        int (*ptrFn) (int,int);
        ptrFn = MiFuncionSuma;
        printf("La suma es : %d\n", ptrFn(5, 6));
        ptrFn = MiFuncionResta;
        printf("La resta es : %d\n", ptrFn(5, 6));
        ptrFn = MiFuncionModulo;
        printf("El Modulo es : %d\n", ptrFn(5, 6));
        getchar();
}
int MiFuncionSuma(int a,int b){
        return a + b;
}
```

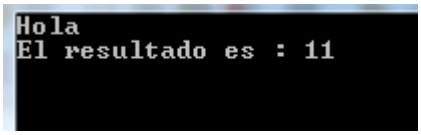
```
int MiFuncionResta(int a,int b){
	return a - b;
}
int MiFuncionModulo(int a,int b){
	return a % b;
}
```



Programa 6.12

En el programa 6.13 se observa la declaración del apuntador dentro de la función –main()- y se muestra la forma en que se puede enviar como parámetro un apuntador a función que retorna a su vez un apuntador tipo entero.

```
#include <stdio.h>
#include <stdlib.h>
int* MiFuncionSuma(int,int);
void Imprime_Resultado(int *);
main(){
	printf("Hola\n");
	int* (*ptrFnSum) (int,int);
	void (*ptrFnRes) (int*);
	ptrFnSum = MiFuncionSuma;
	ptrFnRes = Imprime_Resultado;
	ptrFnRes(ptrFnSum(5, 6));
	getchar();
}

int* MiFuncionSuma(int a,int b)  {
	int *calculo;
	calculo =(int *)malloc(sizeof(int));
	*calculo = a + b;
	return calculo;
}

void Imprime_Resultado(int *a){
	printf("El resultado es : %d \n",*a);
}
```



Programa 6.13

Como hemos visto en los programas anteriores, lo que hacemos es: primero, definir nuestras funciones, con o sin parámetros, luego definimos las variables del tipo apuntador a función, por último enlazamos nuestras variables a la dirección de memoria del código de las funciones antes que las invoquemos. El enlace lo debemos hacer a la función, en consecuencia podemos enlazarlo sólo colocando el nombre de la función o usando el operador de dirección -&- seguido del nombre de la función. Por último, trabajamos únicamente con la función definida.

Ahora se realizará un ejemplo donde la función tiene como parámetro a un apuntador a una función.
Primero se definirá la variable tipo apuntador a función:

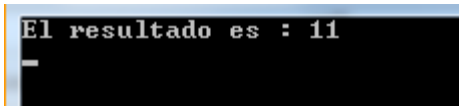
```
int (*ptrFn) (int, int);
```

Ahora se definirá una variable tipo apuntador a función que tiene tres parámetros, el primer parámetro es el apuntador tipo función:

```
void (*ptrFnD)(int (*ptrFn)(int, int), int, int);
```

En el programa 6.14 se mostrará el empleo de este tipo de declaración:

```
#include <stdio.h>
int MiFuncionSuma(int, int);
void Imprime_Resultado(int (*)(int, int), int, int);

main(){
void (*ptrFnRes) (int (*f) (int,int), int x, int y);
ptrFnRes = &Imprime_Resultado;
ptrFnRes(MiFuncionSuma, 5, 6);
getchar();
}
int MiFuncionSuma(int a,int b){
        return a+b;
}
void Imprime_Resultado(int (*funcion)(int , int),int x,int y){
        printf("El resultado es : %d \n",funcion(x,y));
}
```



Programa 6.14

Considere detenidamente las declaraciones de los siguientes ejemplos (en todos ellos fptr es un apuntador a función de tipo distinto de los demás).

void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve **void**.
void (*fptr)(int);	fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int**.
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a **int**.

Cuando el valor devuelto por la función es a su vez un puntero (a función, o de cualquier otro tipo), la notación se complica un poco más:

int const * (*fptr)();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un **int** constante
float (*(*fptr)(char))(int);	fptr es un puntero a función que recibe un **char** como argumento y devuelve un puntero a función que recibe un **int** como argumento y devuelve un **float**.
void * (*(*fptr)(int))[5];	fptr es un puntero a función que recibe un **int** como argumento y devuelve un puntero a un array de 5 punteros-a-**void** (genéricos).
char (*(*fptr)(int, float))();	fptr es un puntero a función que recibe dos argumentos (**int** y **float**), devolviendo un puntero a función que no recibe argumentos y devuelve un **char**.

long (*(*(*fptr)())[5])();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven **long**.
void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve **void**.
void (*fptr)(int);	fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int**.
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a **int**.
int (* afptr[10])(int);	afptr es una matriz de 10 apuntadores a función

Ejercicios:

1. Comente el paso de parámetros por valor y el paso de parámetros por referencia.
2. En el lenguaje de programación C, comente el pase de parámetros por referencia para variables escalares, vectoriales y registros.
3. Comente la diferencia entre variables locales, variables globales y variables estáticas.
4. Escribir un programa donde se realice tanto pase de parámetros por valor como por referencia.
5. Para un vector marque la diferencia o similitud entre los siguientes códigos:
    a. void f(int b[]){ … }
    b. void f1(int *b){ … }
6. Indique la diferencia o similitud entre los siguientes códigos:
    a. void f(int a[][3])[{ … }
    b. void f(int (*a)[3]){ … }
7. Explique y diagrame la forma en que el lenguaje de programación C maneja una matriz de N X M.
8. Comente los parámetros del siguiente prototipo:
    Void m(int ***, int);
9. Comente la diferencia o similitud entre el pase de parámetros cuando es un registro o una matriz.
10. Comente los siguientes prototipos:

a. Void (*fp)(int)
b. Void ir(int(*)(int, int),int);

11. Comente las siguientes declaraciones:
    a. Int *(*fpt)(int *, char *);
    b. Float (*(fpt)(char))(int);
    c. Void *(*fpt)(int)[5];

UNIDAD DE COMPETENCIAS VII. ARCHIVOS Y FLUJOS

(5 HORAS)

(Universidad Tecnológica Metropolitana, 2018)

(Benemerita Universidad Autónoma de Puebla, 2018)

RESUMEN

En esta unidad de competencias se explica el manejo de archivos tanto del tipo texto como del tipo binario y se da un resumen de la forma en que pueden ser accedidos los archivos tanto de registro fijo como de registros de diferentes tamaños.

INTRODUCCIÓN

El problema de los datos utilizados por un programa, es qué todos los datos se eliminan cuando el programa termina. En la mayoría de los casos se desean utilizar datos que no desaparezcan cuando el programa finaliza.

De cara a la programación de aplicaciones, un archivo no es más que una corriente (también llamada *stream*) de bits o bytes que posee un final (generalmente indicado por una **marca de fin de archivo**).

Para poder leer un archivo, se asocia a éste un **flujo** (también llamado secuencia) que es el elemento que permite leer los datos del archivo.

En C un archivo puede ser cualquier cosa, desde un archivo de disco a un terminal o una impresora. Se puede asociar un flujo a un archivo mediante una operación de apertura del archivo

La realidad física de los datos es que éstos son números binarios. Como es prácticamente imposible trabajar utilizando el código binario, los datos deben de ser reinterpretados como enteros, caracteres, cadenas, estructuras, etc.

Al leer un archivo los datos de éste pueden ser leídos como si fueran binarios, o utilizando otra estructura más apropiada para su lectura por parte del programador. A esas estructuras se les llama **registros** y equivalen a las estructuras (***structs***) del lenguaje C. Un archivo así entendido es una colección de registros que poseen la misma estructura interna.

Cada registro se compone de una serie de **campos** que pueden ser de tipos distintos (incluso un campo podría ser una estructura o un array). En cada campo los datos se pueden leer según el tipo de datos que almacenen (enteros, caracteres,...), pero en realidad son unos y ceros.

En la figura 7.1 se intenta representar la realidad de los datos de un archivo. En el ejemplo, el archivo guarda datos de trabajadores. Desde el punto de vista humano hay salarios, nombres, departamentos, etc. Desde el punto de vista de la programación hay una estructura de datos compuesta por un campo de tipo String, un entero, un double y una subestructura que representa fechas.

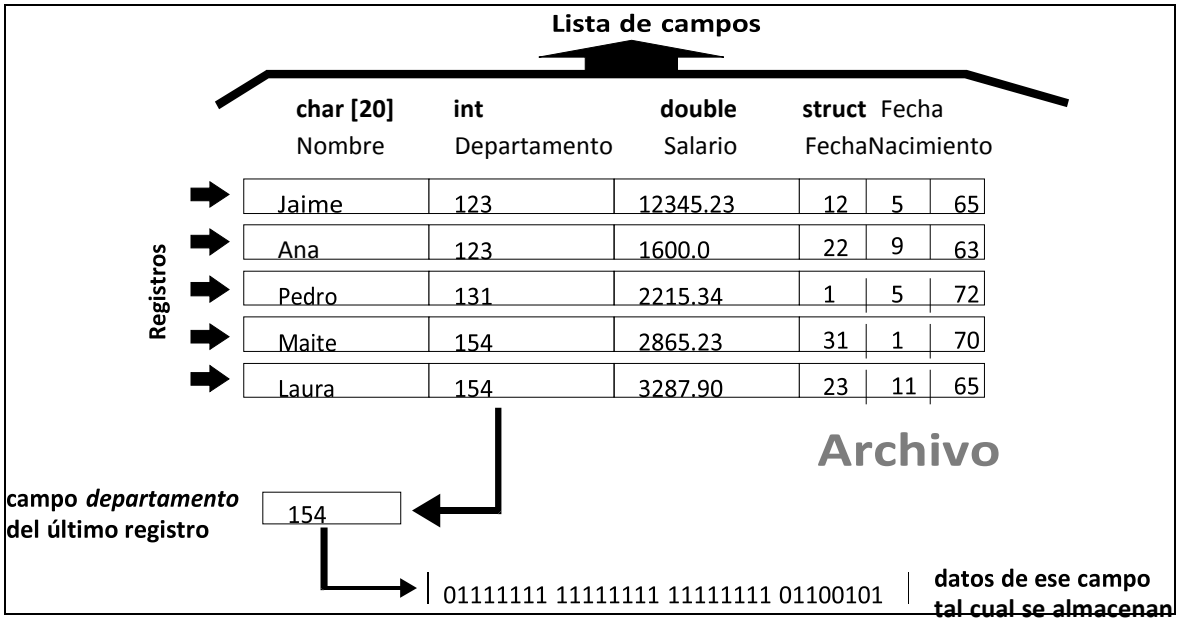


Figura 7.1 Ejemplo de la jerarquía de los datos de un archivo

El hecho de que se nos muestre la información de forma comprensible depende de cómo hagamos interpretar esa información, ya que desde el punto de vista de la máquina todos son unos y ceros.

Los datos que hemos tratado 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 (antiguamente ficheros).

Un archivo es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas registros que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajos denominadas campos.

CLASIFICACIÓN DE LOS ARCHIVOS
Por el tipo de contenido

- **Archivos de texto.** Contienen información en forma de caracteres. Normalmente se organizan los caracteres en forma de líneas al final de cada cual se coloca un carácter de fin de línea (normalmente la secuencia "\r\n"). Al leer hay que tener en cuenta la que la codificación de caracteres puede variar (la 'ñ' se puede codificar muy distinto según qué sistema utilicemos). Los códigos más usados son:
    - **ASCII**. Código de 7 bits que permite incluir 128 caracteres. En ellos no están los caracteres nacionales por ejemplo la 'ñ' del español) ni símbolos de uso frecuente (matemáticos, letras griegas,…). Por ello se uso el octavo bit para producir códigos de 8 bits, llamado ASCII extendido (lo malo es que los ASCII de 8 bits son diferentes en cada país).
    - **ISO 8859-1**. El más usado en occidente. Se la llama codificación de Europa Occidental. Son 8 bits con el código ASCII más los símbolos frecuentes del inglés, español, francés, italiano o alemán entre otras lenguas de Europa Occidental.
    - **Windows 1252**. Windows llama ANSI a esta codificación. En realidad se trata de un superconjunto de ISO 8859-1 que es utilizado en el almacenamiento de texto por parte de Windows.
    - **Unicode**. La norma de codificación que intenta unificar criterios para hacer compatible la lectura de caracteres en cualquier idioma. Hay varias posibilidades de aplicación de Unicode, pero la más utilizada en la actualidad es la UTF-8 que es totalmente compatible con ASCII y que usando el octavo bit con valor 1 permite leer más bytes para poder almacenar cualquier número de caracteres (en la actualidad hay 50000)
- **Archivos binarios.** 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.

Por la forma de acceso

Según la forma en la que accedamos a los archivos disponemos de dos tipos de archivo:

- **Archivos secuenciales**. Se trata de archivos en los que el contenido se lee o escribe de forma continua. No se accede a un punto concreto del archivo, para leer cualquier información necesitamos leer todos los datos hasta llegar a dicha información. En general son los archivos de texto los que se suelen utilizar de forma secuencial.

- **Archivos de acceso directo.** Se puede acceder a cualquier dato del archivo conociendo su posición en el mismo. Dicha posición se suele indicar en bytes. En general los archivos binarios se utilizan mediante acceso directo.

Cualquier archivo en C puede ser accedido de forma secuencial o usando acceso directo.
En 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 secuencia 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; C no tiene palabras claves que realicen las operaciones de E/S. La tabla 7.1 da un breve resumen de las funciones que se pueden utilizar. Se debe incluir la librería STDIO.H. Observe que la mayoría de las funciones comienzan con la letra “F”, esto es un vestigio del estándar C de Unix.

Nombre	Función
fopen()	Abre un archivo.
fclose()	Cierra un archivo.
fgets()	Lee una cadena de un archivo.
fputs()	Escribe una cadena en un archivo
fseek()	Busca un byte específico de un archivo.
fprintf()	Escribe una salida con formato en el archivo.
fscanf()	Lee una entrada con formato desde el archivo.
feof()	Devuelve cierto si se llega al final del archivo.
ferror()	Devuelve cierto si se produce un error.
rewind()	Coloca el localizador de posición del archivo al principio del mismo.
remove()	Borra un archivo.
fflush()	Vacía un archivo.

Tabla 7.1

El puntero 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 secuencia 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 FILE que se define en STDIO.H. Un programa necesita utilizar punteros a archivos para leer o escribir en los mismos. Para obtener una variable de este tipo se utiliza una secuencia como esta: **FILE *F;**

APERTURA Y CIERRE DE ARCHIVOS
APERTURA

La apertura de los archivos se realiza con la función **fopen**. Esta función devuelve un puntero de tipo FILE al archivo que se desea abrir. El prototipo de la función es:

FILE ***fopen(**const char ***nombreArchivo**, const char ***modo**)*

nombreArchivo es una cadena que contiene la ruta hacia el archivo que se desea abrir. *modo* es otra cadena cuyo contenido se puede observar en la tabla 7.2:

modo	significado
"r"	Abre un archivo para lectura de archivo de textos (el archivo tiene que existir)
"w"	Crea un archivo de escritura de archivo de textos. Si el archivo ya existe se borra el contenido que posee.
"a"	Abre un archivo para adición de datos de archivo de textos
"rb"	Abre un archivo para lectura de archivos binarios (el archivo tiene que existir)
"wb"	Crea un archivo para escritura de archivos binarios (si ya existe, se descarta el contenido)
"ab"	Abre un archivo para añadir datos en archivos binarios
"r+"	Abre un archivo de texto para lectura/escritura en archivos de texto. El archivo tiene que existir
"w+"	Crea un archivo de texto para lectura/escritura en archivos de texto. Si el archivo tenía datos, estos se descartan en la apertura.
"a+"	Crea o abre un archivo de texto para lectura/escritura. Los datos se escriben al final.
"r+b"	Abre un archivo binario para lectura/escritura en archivos de texto
"w+b"	Crea un archivo binario para lectura/escritura en archivos de texto. Si el archivo tiene datos, éstos se pierden.

“a+b”	Crea o abre un archivo binario para lectura/escritura. La escritura se hace al final del archivo.

Tabla 7.2

Un archivo se puede abrir en **modo texto** o en **modo binario**. En modo texto se leen o escriben caracteres, en modo binario se leen y escriben cualquier otro tipo de datos.

La función **fopen** devuelve un puntero de tipo FILE al archivo que se está abriendo. En caso de que esta apertura falle, devuelve el valor NULL (puntero nulo). El fallo se puede producir porque el archivo no exista (sólo en los modos *r*), porque la ruta al archivo no sea correcta, porque no haya permisos suficientes para la apertura, porque haya un problema en el sistema,….

CIERRE DE ARCHIVOS

La función **fclose** es la encargada de cerrar un archivo previamente abierto. Su prototipo es:

```
Int fclose(FILE * pArchivo);
```

pArchivo es el puntero que señala al archivo que se desea cerrar. Si devuelve el valor cero, significa que el cierre ha sido correcto, en otro caso se devuelve un número distinto de cero.

PROCESAMIENTO DE ARCHIVOS DE TEXTO

LEER Y ESCRIBIR CARACTERES

función **getc**

Esta función sirve para leer caracteres de un archivo de texto. Los caracteres se van leyendo secuencialmente hasta llegar al final. Su prototipo es:

```
Int getc(FILE *pArchivo);
```

Esta función devuelve una constante numérica llamada EOF (definida también en el archivo **stdio.h**) cuando ya se ha alcanzado el final del archivo. En otro caso devuelve el siguiente carácter del archivo.

Ejemplo:

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

int main(){     FILE
*archivo;     char c=0;
    archivo=fopen("c:\\prueba.txt","r+");   if(archivo!=NULL) {
  // Apertura correcta
        while(c!=EOF){ // Se lee hasta llegar al final

          c=fgetc(archivo);

          putchar(c);

        }

        fclose(archivo);

    }     else{
        printf("Error");
    }         getch();

}
```

Función **fputc**

Es la función que permite escribir caracteres en un archivo de texto. Prototipo:

int **fputc(**int carácter**,** FILE *pArchivo**);**

Escribe el carácter indicado en el archivo asociado al puntero que se indique. Si esta función tiene éxito (es decir, realmente escribe el carácter) devuelve el carácter escrito, en otro caso devuelve EOF.

COMPROBAR FINAL DE ARCHIVO

Anteriormente se ha visto como la función **fgetc** devuelve el valor EOF si se ha llegado al final del archivo. Otra forma de hacer dicha comprobación, es utilizar la función **feof** que devuelve verdadero si se ha llegado al final del archivo. Esta función es muy útil para ser utilizada en archivos binarios (donde la constante EOF no tiene el mismo significado) aunque se puede utilizar en cualquier tipo de archivo. Sintaxis:

```
int feof(FILE *pArchivo)
```

Así el código de lectura de un archivo para mostrar los caracteres de un texto, quedaría:

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

int main(){
        FILE *archivo;
        char c=0;
        archivo=fopen("c:\\prueba.txt","r+");
        if(archivo!=NULL) {
              while(!feof(archivo)){
                    c=fgetc(archivo);
                     putchar(c);
              }
        }
        else{
              printf("Error");
        }
       fclose(archivo);
      getch();
}
```

LEER Y ESCRIBIR STRINGS
función **fgets**

Se trata de una función que permite leer textos de un archivo de texto. Su prototipo es:

char ***fgets(char ***texto, **int** longitud**,** FILE *pArchivo)

Esta función lee una cadena de caracteres del archivo asociado al puntero de archivos *pArchivo* y la almacena en el puntero *texto.* Lee la cadena hasta que llegue un salto de línea, o hasta que se supere la longitud indicada. La función devuelve un puntero señalando al texto leído o un puntero nulo (NULL) si la operación provoca un error.

Ejemplo (lectura de un archivo de texto):

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

int main(){     FILE
*archivo;     char
texto[2000];
archivo=fopen("c:\\prueba2.txt","r");     if(archivo!=NULL) {
        fgets(texto,2000,archivo);
```

```
        while(!feof(archivo)){
           puts(texto);
           fgets(texto,2000,archivo);
        }
        fclose(archivo);
    }     else{
        printf("Error en la apertura"); }
}
```

En el listado el valor 2000 dentro de la funcón *fgets* tiene como único sentido, asegurar que se llega al final de línea cada vez que lee el texto (ya que ninguna línea del archivo tendrá más de 2000 caracteres).

función **fputs**

Es equivalente a la anterior, sólo que ahora sirve para escribir strings dentro del un archivo de texto. Prototipo:

int **fputs(const char** texto**,** FILE *pArchivo)

Escribe el texto en el archivo indicado. Además al final del texto colocará el carácter del salto de línea (al igual que hace la función **puts**). En el caso de que ocurra un error, devuelve EOF. Ejemplo (escritura en un archivo del texto introducido por el usuario en pantalla):

Función **fprintf**

Se trata de la función equivalente a la función **printf** sólo que esta permite la escritura en archivos de texto. El formato es el mismo que el de la función **printf**, sólo que se añade un parámetro al principio que es el puntero al archivo en el que se desea escribir.

La ventaja de esta instrucción es que aporta una gran versatilidad a la hora de escribir en un archivo de texto.

Ejemplo. Imaginemos que deseamos almacenar en un archivo los datos de nuestros empleados, por ejemplo su número de empleado (un entero), su nombre (una cadena de texto) y su sueldo (un valor decimal). Entonces habrá que leer esos tres datos por separado, pero al escribir lo haremos en el mismo archivo separándolos por una marca de tabulación. El código sería:

```
#include <stdio.h>
 int main(){
    int n=1; /*Número del empleado*/
          char nombre[80];
          double salario;
          FILE *pArchivo;
```

```
   pArchivo=fopen("c:\\prueba3.txt","w");
        if(pArchivo!=NULL){
                do{
                        printf("Introduzca el número de empleado: ");
                        scanf("%d",&n);
 /*Solo seguimos si n es positivo, en otro caso entenderemos que la lista ha terminado */
                       if(n>0){
                                printf("Introduzca el nombre del empleado: ");
                                scanf("%s",nombre);
                                printf("Introduzca el salario del empleado: ");
                                scanf("%lf",&salario);
                                fprintf(pArchivo,"%d\t%s\t%lf\n", n,nombre,salario);
                       }
                }while(n>0);
                fclose(pArchivo);
        }
}
```

Función **fscanf**

Se trata de la equivalente al **scanf** de lectura de datos por teclado. Funciona igual sólo que requiere un primer parámetro que sirve para asociar la lectura a un puntero de archivo. El resto de parámetros se manejan igual que en el caso de **scanf.**

Ejemplo (lectura de los datos almacenados con **fprintf**, los datos están separados por un tabulador).

```
#include <stdio.h>
#include <conio.h>
int main(){
        int n=1;
        char nombre[80];
        double salario;
        FILE *pArchivo;
        pArchivo=fopen("c:\\prueba3.dat","r");
        if(pArchivo!=NULL)
                while(!feof(pArchivo)){
                        fscanf(pArchivo,"%d\t%s\t%lf\n",&n,nombre,&salario);
                        printf("%d\t%s\t%lf\n",n,nombre,salario);
                  }
        fclose(pArchivo);
        getch();
}
```

Función **fflush**

La sintaxis de esta función es:

int **fflush(**FILE ***p**Archivo)

Esta función vacía el buffer sobre el archivo indicado. Si no se pasa ningún puntero se vacían los búferes de todos los archivos abiertos. Se puede pasar también la corriente estándar de entrada *stdin* para vaciar el búfer de teclado (necesario si se leen caracteres desde el teclado, de otro modo algunas lecturas fallarían).

Esta función devuelve 0 si todo ha ido bien y la constante EOF en caso de que ocurriera un problema al realizar la acción.

VOLVER AL PRINCIPIO DE UN ARCHIVO

La función **rewind** tiene este prototipo:

void **rewind(**FILE *pArchivo);

Esta función inicializa el indicador de posición, de modo que lo siguiente que se lea o escriba será lo que esté al principio del archivo. En el caso de la escritura hay que utilizarle con mucha cautela (sólo suele ser útil en archivos binarios).

OPERACIONES PARA USO CON ARCHIVOS BINARIOS

función ***fwrite***

Se trata de la función que permite escribir en un archivo datos binarios del tamaño que sea. El prototipo es:

size_t **fwrite**(**void** *bufer, size_t bytes, size_t cuenta, FILE *p)

En ese prototipo aparecen estas palabras:

- **size_t**. Tipo de datos definido en el archivo **stdio.h,** normalmente equivalente al tipo **unsigned.** Definido para representar tamaños de datos.
- ***búfer.*** Puntero a la posición de memoria que contiene el dato que se desea escribir.
- ***bytes.*** Tamaño de los datos que se desean escribir (suele ser calculado con el operador **sizeof**)

- ***cuenta.*** Indica cuantos elementos se escribirán en el archivo. Cada elemento tendrá el tamaño en bytes indicado y su posición será contigua a partir de la posición señalada por el argumento *búfer*

- ***p.*** Puntero al archivo en el que se desean escribir los datos.

La instrucción **fwrite** devuelve el número de elementos escritos, que debería coincidir con el parámetro *cuenta*, de no ser así es que hubo un problema al escribir.

Ejemplo de escritura de archivo. En el ejemplo se escriben en el archivo *datos.dat* del disco duro C registros con una estructura (llamada *Persona*) que posee un texto para almacenar el nombre y un entero para almacenar la edad. Se escriben registros hasta un máximo de 25 o hasta que al leer por teclado se deje el nombre vacío:

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
typedef struct {
	char nombre[25];
	int edad;
}Persona;

int main(){
	Persona per[25];
	int i=0;
	FILE *pArchivo;
	pArchivo=fopen("C:\\datos.dat","wb");
	if(pArchivo!=NULL){
		do{
			fflush(stdin); /* Se vacía el búfer de teclado */
			 printf("Introduzca el nombre de la persona: ");
			 gets(per[i].nombre);
			 if(strlen(per[i].nombre)>0){
				printf("Introduzca la edad");
				scanf("%d",&(per[i].edad));
				fwrite(&per[i],sizeof(Persona),1,pArchivo);
				i++;
			 }
		 }while(strlen(per[i].nombre)>0 && i<=24);
		fclose(pArchivo);
	}
	else{
		printf("Error en la apertura del archivo");
	}
}
```

La instrucción **fwrite** del ejemplo, escribe el siguiente elemento leído del cual recibe su dirección, tamaño (calculado con **sizeof**) se indica que sólo se escribe un elemento y el archivo en el que se guarda (el archivo asociado al puntero *pArchivo*).

función ***fread***

Se trata de una función absolutamente equivalente a la anterior, sólo que en este caso la función lee del archivo. Prototipo:

size_t fread(void *buber, size_t bytes, size-t cuenta FILE *p)

El uso de la función es el mismo que **fwrite**, sólo que en este caso lee del archivo. La lectura es secuencial se lee hasta llegar al final del archivo. La instrucción **fread** devuelve el número de elementos leídos, que debería coincidir con el parámetro *cuenta*, de no ser así es que hubo un problema o es que se llegó al final del archivo (el final del archivo se puede controlar con la función **feof** descrita anteriormente).

Ejemplo (lectura del archivo **datos.dat** escrito anteriormente):

```
#include <stdio.h>
typedef struct {
        char nombre[25];
        int edad;
}Persona;

int main(){
        Persona aux;
        FILE *pArchivo;
        pArchivo=fopen("C:\\datos.dat","rb");
        if(pArchivo!=NULL){
                fseek(pArchivo,3*sizeof(Persona),SEEK_SET);
                fread(&aux,sizeof(Persona),1,pArchivo);
                printf("%s, %d años",aux.nombre,aux.edad);
                fclose(pArchivo);
        }
}
```

USO DE ARCHIVOS DE ACCESO DIRECTO

INTRODUCCIÓN.

Acceso indexado o acceso indizado (la traducción correcta) es un modo de organización de archivos en el cual al archivo le acompaña un índice que tiene la función de permitir el acceso directo a los registros del disco. El índice se puede organizar de diversas formas, las más típicas son: secuencial, multinivel y árbol. A través del índice podremos procesar un archivo de forma secuencial o de forma directa según la clave de indización, y esto independientemente de cómo esté organizado el archivo por sí mismo.

PROCESAMIENTO DE ARCHIVOS.

Es la forma de solicitar la información al disco. Existen dos métodos para ello:

- Modo Secuencial: Se lee la información de un archivo de registro en registro teniendo que leer todos los que hay antes del que buscamos. Se emplea bien por deseo, bien por imposición del tipo de soporte que estamos usando. El acceso secuencial es recomendable cuando se quiere trabajar con muchos registros del archivo.
- Modo Directo: Se puede acceder a un registro si tener que leer todos los anteriores (basta con un pequeño número de lecturas). Hay dos maneras:
  - Cálculo: Cada región tiene una clave sobre la que se aplica un cálculo que indica el lugar de grabación (Hashing).
  - índices: existe un índice independiente o asociado al archivo en el cual se busca el registro y se indica donde está.

ORGANIZACIÓN DE ARCHIVOS.

Son los modos de disponer los registros del archivo en el soporte. Existen tres modos principales:

- Secuencial: Un registro a continuación de otro.
- Directo: Los registros binarios no se disponen en el soporte atendiendo a un algoritmo de cálculo.
- Indizado: Los registros generalmente se almacenan acceso secuencialmente y van con un índice .

Los registros se organizan en una secuencia basada en un campo clave presentando dos características, un índice del archivo para soportar los accesos aleatorios y un archivo de desbordamiento. El índice proporciona una capacidad de búsqueda para llegar rápidamente al registro deseado y el archivo de desbordamiento es similar al archivo de registros usado en un archivo acceso secuencial, pero está integrado de forma que los archivos de desbordamiento se ubiquen siguiendo un puntero desde su registro predecesor.

La estructura más simple tiene como índice un archivo acceso secuencial simple, cada registro del archivo índice tiene dos campos, un campo clave igual al del archivo principal y un puntero al archivo principal. Para encontrar un campo específico se busca en el índice hasta encontrar el valor mayor de la clave que es igual o precede al valor deseado de la clave, la búsqueda continua en el archivo principal a partir de la posición que indique el puntero.
Cada registro del archivo principal tiene un campo adicional que es un puntero al archivo de desbordamiento. Cuando se inserta un nuevo registro al archivo, también se añade al archivo de desbordamiento. El registro del archivo principal que precede inmediatamente al nuevo registro según la secuencia lógica se actualiza con un puntero del registro nuevo en el archivo de desbordamiento, si el registro inmediatamente anterior está también en el archivo de desbordamiento se actualizará el puntero en el registro.
Para procesar secuencialmente un archivo completo los registros del archivo principal se procesarán en secuencia hasta encontrar un puntero al archivo de desbordamiento, el acceso continúa en el archivo de desbordamiento hasta que encuentra un puntero nulo, entonces renueva el acceso donde se abandonó en el archivo principal.

Ventajas y desventajas.

Ventajas:

- Permite el acceso secuencial.
- Permite el acceso directo a los registros.
- Se pueden actualizar los registros en el mismo archivo, sin necesidad de crear un archivo nuevo de copia en el proceso de actualización.

Desventajas:

- Ocupa más espacio en el disco que los archivos secuenciales, debido al uso del área de índices.
- Tiene tendencia a que aumente el tiempo medio de acceso a los registros, cuando se producen muchas altas nuevas con claves que hay que intercalar entre las existentes, ya que aumenta el área de overflow.
- Solo se puede utilizar soportes direccionables.
- Obliga a una inversión económica mayor, por la necesidad de programas y, a veces, hardware más sofisticado.

Problemas

- Desafortunadamente el archivo de secuencial es difícil de mantener.
- Los índices (aún usando un esquema de B+ Tree) nos permiten hacer búsquedas rápidamente pero se debe hacer demasiados "seeks" para poder hacerlo.

En la práctica en muchas situaciones se requieren de ambas cosas (búsquedas y accesos secuenciales).

Las 2 soluciones existentes:

- ISAM indexed sequential access method
- B+ Tree

MÉTODO DE ACCESO SECUENCIAL INDIZADO (ISAM)

Básicamente es la idea de tener un índice esparcido, de manera que el archivo de datos lo agrupamos por bloques y en cada bloque existe un "rango" ordenado de los datos.

Lo importante a resaltar aquí es que los bloques no necesariamente están formando una secuencia en el archivo, es decir, dentro del archivo de datos los bloques pueden estar desordenados, aquí el detalle es mantener un puntero a cada bloque subsecuente.

ESTRUCTURA LÓGICA DE UN ARCHIVO DE ACCESO SECUENCIAL INDIZADO.

En este tipo de organización de archivos se dispone de una tabla en la que aparecen ordenados secuencialmente los números de la clave del archivo y asociados a cada uno de ellos de la dirección del registro correspondiente. Cada registro en el archivo se identifica por medio de un número o un grupo de caracteres exclusivos (llave primaria).

Los registros se almacenan según una secuencia física dada, este ordenamiento más usualmente es de disponerlos en el orden indicado por la llave, con lo cual permite un procesamiento secuencial de los registros precisamente en el orden en que están distribuidos en el archivo y también es posible el procesamiento aleatorio, en el que se llega a los registros en un orden cualquiera.

MÉTODOS DE ACCESO Y SISTEMAS DE ALMACENAMIENTO POR S. O.

Windows

Los sistemas de organización de archivos que utiliza Windows utilizan el acceso secuencial y **acceso indizado** adjuntos en un mismo método), el acceso directo en algunos casos en la utilización de los sistemas de organización por tablas.

Linux

Linux es el sistema operativo que soporta más sistemas de organización lo cual lo convierte en uno de los más versátiles; además al igual que Windows utiliza los mismos tipos de acceso y sistemas de organización. La estructura de archivos es una estructura jerárquica en forma de árbol invertido, donde el directorio principal (raíz) es el directorio "/", del que cuelga toda la estructura del sistema. Éste sistema de archivos permite al usuario crear, borrar y acceder a los archivos sin necesidad de saber el lugar exacto en el que se encuentran. No existen unidades físicas, sino archivos que hacen referencia a ellas. Consta de tres partes importantes, superbloque, tabla de i-nodos y bloques de datos.

FUNCIONES PRINCIPALES DE C.

función ***fseek***

Los archivos de acceso directo son aquellos en los que se puede acceder a cualquier parte del archivo sin pasar por los anteriores. Hasta ahora todas las funciones de proceso de archivos vistas han trabajado con los mismos de manera secuencial.

En los archivos de acceso directo se entiende que hay un indicador de posición en los archivos que señala el dato que se desea leer o escribir. Las funciones **fread** o **fwrite** vistas anteriormente (o las señalas para leer textos) mueven el indicador de posición cada vez que se usan.

El acceso directo se consigue si se modifica ese indicador de posición hacia la posición deseada. Eso lo realiza la función **fseek** cuyo prototipo es:

int **fseek(**FILE ***** pArchivo, **long** bytes**, int** origen)

Esta función coloca el cursor en la posición marcada por el origen desplazándose desde allí el número de bytes indicado por el segundo parámetro (que puede ser negativo). Para el parámetro origen se pueden utilizar estas constantes (definidas en **stdio.h**):

- **SEEK_SET.** Indica el principio del archivo.

- **SEEK_CUR.** Posición actual.
- **SEEK_END.** Indica el final del archivo.

La función devuelve cero si no hubo problemas al recolocar el indicador de posición del archivo. En caso contrario devuelve un valor distinto de cero:

Por ejemplo:

```
#include <stdio.h>
typedef struct {
	char nombre[25];
	int edad;
}Persona;

int main(){
	Persona aux;
	FILE *pArchivo;
	pArchivo=fopen("C:\\datos.dat","rb");
	if(pArchivo!=NULL){
		fseek(pArchivo,3*sizeof(Persona),SEEK_SET);
		fread(&aux,sizeof(Persona),1,pArchivo);
		printf("%s, %d años",aux.nombre,aux.edad);
		fclose(pArchivo);
	}
}
```

El ejemplo anterior muestra por pantalla a la cuarta persona que se encuentre en el archivo (*3*sizeof(Persona)* devuelve la cuarta persona, ya que la primera está en la posición cero).

función ***ftell***

Se trata de una función que obtiene el valor actual del indicador de posición del archivo (la posición en la que se comenzaría a leer con una instrucción de lectura). En un archivo binario es el número de byte en el que está situado el cursor desde el comienzo del archivo. Prototipo:

long **ftell(**FILE ***pArchivo)

Por ejemplo para obtener el número de registros de un archivo habrá que dividir el tamaño del mismo entre el tamaño en bytes de cada registro. Ejemplo:

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

```
/* Estructura de los registros almacenados en el archivo*/
typedef struct{
        int n;
        int nombre[80];
        double saldo;
}Movimiento;

int main(){
   FILE *f=fopen("movimientos3.bin","rb");
   int nReg;/*Guarda el número de registros*/
   if(f!=NULL){
      fseek(f,0,SEEK_END);
      nReg=ftell(f)/sizeof(Movimiento);
      printf("Nº de registros en el archivo = %d",nReg);
      getch();
   }
   else{
      printf("Error en la apertura del archivo");
   }
}
```

En el caso de que la función **ftell** falle, da como resultado el valor -1.

Funciones ***fgetpos*** **y** ***fsetpos***

Ambas funciones permiten utilizar marcadores para facilitar el trabajo en el archivo. Sus prototipos son:

int **fgetpos(**FILE *****pArchivo, fpos_t *posicion**);**
int **fsetpos(**FILE *****pArchivo, fpos_t *posicion**)**

La primera almacena en el puntero *posición* la posición actual del cursor del archivo (el indicador de posición), para ser utilizado más adelante pos **fsetpos** para obligar al programa a que se coloque en la posición marcada.

En el caso de que todo vaya bien ambas funciones devuelven cero. El tipo de datos **fpos_t** está declarado en la librería *stdio.h* y normalmente se corresponde a un número **long**. Es decir normalmente su declaración es:

typedef long fpos_t;

Aunque eso depende del compilador y sistema en el que trabajemos. Ejemplo de uso de estas funciones:

```
FILE *f=fopen("prueba.bin","rb+");
```

```
fpos_t posicion;
if(f!=NULL){
 .../*operaciones de lectura o de manipulación*/
        fgetpos(f,&posicion); /*Captura de la posición actual*/
 .../*operaciones de lectura o manipulación */
        fsetpos(f,&posicion);
         /*El cursor vuelve a la posición capturada */
```

EJEMPLOS DE ARCHIVOS DE USO FRECUENTE

Archivos de texto delimitado

Se utilizan muy habitualmente. En ellos se entiende que cada registro es una línea del archivo. Cada campo del registro se separa mediante un carácter especial, como por ejemplo un tabulador.

Ejemplo de contenido para un archivo delimitado por tabuladores:

```
1 Pedro      234.430000
2 Pedro      45678.234000
3 Pedro      123848.321000
5 Javier     34120.210000
6 Alvaro     324.100000
7 Alvaro     135.600000
8 Javier     123.000000
9 Pedro      -5.000000
```

La escritura de los datos se puede realizar con la función **fprintf**, por ejemplo:

```
fprintf("%d\t%s\t\%lf",numMov,nombre,saldo);
```

La lectura se realizaría con **fscanf:**

```
fscanf("%d\t%s\t\%lf",&numMov,nombre,&saldo);
```

Sin embargo si el delimitador no es el tabulador, podríamos tener problemas. Por ejemplo en este archivo:

```
AR,6,0.01
AR,7,0.01
AR,8,0.01
AR,9,0.01
AR,12,0.02
AR,15,0.03
AR,20,0.04
AR,21,0.05
```

```
BI,10,0.16
BI,20,0.34
BI,38,0.52
BI,57,0.77
```

La escritura de cada campo se haría con:

```
fprintf("%s,%d,%lf",tipo,modelo,precio);
```

Pero la instrucción:

```
fscanf("%s,%d,%lf",tipo,&modelo,&precio);
```

da error, al leer *tipo* se lee toda la línea, no sólo hasta la coma.

La solución más recomendable para leer de texto delimitado es leer con la instrucción **fgets** y guardar la lectura en una sola línea. Después se debería utilizar la función **strtok**.

Esta función permite extraer el texto delimitado por caracteres especiales. A cada texto delimitado se le llama **token.** Para ello utiliza la cadena sobre la que se desean extraer las subcadenas y una cadena que contiene los caracteres delimitadores. En la primera llamada devuelve el primer texto delimitado.

El resto de llamadas a **strtok** se deben hacer usando NULL en el primer parámetro. En cada llamada se devuelve el siguiente texto delimitado (Ver figura 7.2). Cuando ya no hay más texto delimitado devuelve **NULL**.

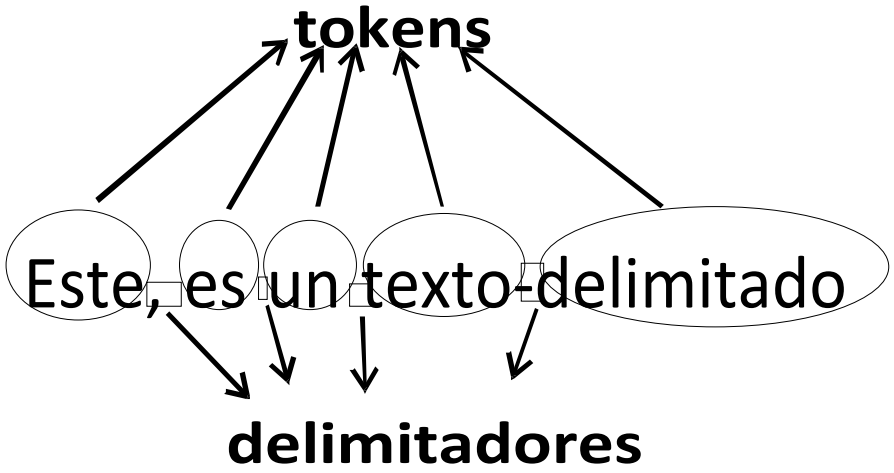


Figura 7.2

En el diagrama anterior la frase "Este, es un texto-delimitado" se muestra la posible composición de *tokens* y delimitadores del texto. Los delimitadores se

deciden a voluntad y son los caracteres que permiten separar cada *token.*
Ejemplo de uso:

```
#include <stdio.h>
#include <string.h>
int main(){
   char texto[] = "Texto de ejemplo. Utiliza, varios delimitadores\n\n";
   char delim[] = " ,.";
   char *token;
   printf("Texto inicial: %s\n", texto);
   /* En res se guarda el primer texto delimitado (token) */
   token = strtok( texto, delim);
   /* Obtención del resto de tokens (se debe usar NULL    en el primer
parámetro)*/
   do{
          printf("%s\n", token);
          token=strtok(NULL,delim);
   }while(token != NULL );
}
```

Cada palabra de la frase sale en una lína separada.

Para leer del archivo anterior (el delimitado con comas, el código sería):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
/* Se encarga de transformar una línea del archivo de texto en los datos
correspondientes. Para ello extrae los tokens  y los convierte al tipo adecuado*/
void extraeDatos(char *linea, char *tipo, int *modelo, double *precio)  {
          char *cadModelo, *cadPrecio;
          strcpy(tipo,strtok(linea,","));
          cadModelo=strtok(NULL,",");
          *modelo=atoi(cadModelo);
          cadPrecio=strtok(NULL,",");
          *precio=atof(cadPrecio);
}

int main(){
          FILE *pArchivo=fopen("piezas.txt","r");
          char tipo[3];
          char linea[2000];
```

```
        int modelo;
        double precio;
        if(pArchivo!=NULL){
                fgets(linea,2000,pArchivo);
                while(!feof(pArchivo)){
                        extraeDatos(linea,tipo,&modelo,&precio);
                        printf("%s %d %lf\n",tipo,modelo,precio);
                        fgets(linea,2000,pArchivo);
                }
                fclose(pArchivo);
        }
        getch();
}
```

ARCHIVOS DE TEXTO CON CAMPOS DE ANCHURA FIJA

Es otra forma de grabar registros de datos utilizando archivos de datos. En este caso el tamaño en texto de cada fila es el mismo, pero cada campo ocupa un tamaño fijo de caracteres.

Ejemplo de archivo:

```
AR6  0.01
AR7  0.01
AR8  0.01
AR9  0.01
AR12 0.02
AR15 0.03
AR20 0.04
AR21 0.05
BI10 0.16
BI20 0.34
BI38 0.52
```

En el ejemplo (con los mismos datos vistos en el ejemplo de texto delimitado), los dos primeros caracteres indican el tipo de pieza, los tres siguientes (que son números) el modelo y los seis últimos el precio (en formato decimal):

En todos los archivos de texto de tamaño fijo hay que leer las líneas de texto con la función **fgets**. Y sobre ese texto hay que ir extrayendo los datos de cada campo (para lo cual necesitamos, como es lógico, saber la anchura que tiene cada campo en el archivo).

Ejemplo (lectura de un archivo de texto de anchura fija llamado **Piezas2.txt** la estructura del archivo es la comentada en el ejemplo de archivo anterior):

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>

/*Extrae de la cadena pasada como primer parámetro los c caracteres que van de la posición inicio a la posición fin (ambos incluidos) y almacena el resultado en el puntero  subcad*/
void subcadena(const char *cad, char *subcad,int inicio, int fin){
   int i,j;
   for(i=inicio,j=0;i<=fin;i++,j++){
                  subcad[j]=cad[i];
   }
   subcad[j]=0;/* Para finalizar el String */
}
/*Se encarga de extraer adecuadamente los campos de cada línea del archivo */
void extraeDatos(char *linea, char *tipo, int *modelo, double *precio){
   char cadModelo[3], cadPrecio[6];
   /* Obtención del tipo */
   subcadena(linea, tipo,0,1);
   /*Obtención del modelo */
   subcadena(linea, cadModelo,2,5);
   *modelo=atoi(cadModelo);
   /* Obtención del precio */
   subcadena(linea,cadPrecio,6,11);
   *precio=atof(cadPrecio);
}
int main(){
   FILE *pArchivo=fopen("piezas.txt","r");
   char tipo[3];
   char linea[2000];
   int modelo;
   double precio;
   if(pArchivo!=NULL){
      fgets(linea,2000,pArchivo);
      while(!feof(pArchivo)){
         extraeDatos(linea,tipo,&modelo,&precio);
          printf("%s %d %lf\n",tipo,modelo,precio);
          fgets(linea,2000,pArchivo);
      }
      fclose(pArchivo);
   }
   getch();
}/*fin del main */
```

FORMATOS BINARIOS CON REGISTROS DE TAMAÑO DESIGUAL

Los archivos binarios explicados hasta ahora son aquellos que poseen el mismo tamaño de registro. De tal manera que para leer el quinto registros habría que posicionar el cursor de registros con **fseek** indicando la posición como **4*sizeof(Registro)** donde *Registro* es el tipo de estructura del registro que se almacena en el archivo.

Hay casos en los que compensa otro tipo de organización en la que los registros no tienen el mismo tamaño.

Imaginemos que quisiéramos almacenar este tipo de registro:

```
typedef struct{
char nombre[60];
int edad;
int curso;
} Alumno;
```

Un alumno puede tener como nombre un texto que no llegue a 60 caracteres, sin embargo al almacenarlo en el archivo siempre ocupará 60 caracteres. Eso provoca que el tamaño del archivo se dispare. En lugar de almacenarlo de esta forma, el formato podría ser así:

Tamaño del texto **(int)**	Nombre **(char[])**	Edad **(int)**	Curso **(int)**

El campo nombre ahora es de longitud variable, por eso hay un primer campo que nos dice qué tamaño tiene este campo en cada registro. Como ejemplo de datos:

7	Alberto	14	1

11	María Luisa	14	1

12	Juan Antonio	15	1

Los registros son de tamaño variable, como se observa con lo que el tamaño del archivo se optimiza. Pero lo malo es que la manipulación es más compleja y hace imposible el acceso directo a los registros (es imposible saber cuándo comienza el quinto registro).

Para evitar el problema del acceso directo, a estos archivos se les acompaña de un **archivo de índices**, el cual contiene en cada fila dónde comienza cada registro. Ejemplo:

Nº reg	Pos en bytes
1	0

2	20
3	46

En ese archivo cada registro se compone de dos campos (aunque el primer campo se podría quitar), el primero indica el número de registro y el segundo la posición en la que comienza el registro.

En los ficheros con índice lo malo es que se tiene que tener el índice permanentemente organizado.

ARCHIVOS INDEXADOS

Uno de los puntos más complicados de la manipulación de archivos es la posibilidad de mantener ordenados los ficheros. Para ello se utiliza un fichero auxiliar conocido como índice.

Cada registro tiene que poseer una clave que es la que permite ordenar el fichero. En base de esa clave se genera el orden de los registros. Si no dispusiéramos de índice, cada vez que se añade un registro más al archivo habría que regenerar el archivo entero (con el tiempo de proceso que consume esta operación).

Por ello se prepara un archivo separado donde aparece cada clave y la posición que ocupan en el archivo. Al añadir un registro se añade al final del archivo; el que sí habrá que reorganizar es el fichero de índices para que se actualicen, pero cuesta menos organizar dicho archivo ya que es más corto. En la figura 7.3 se observa un ejemplo de fichero de datos:

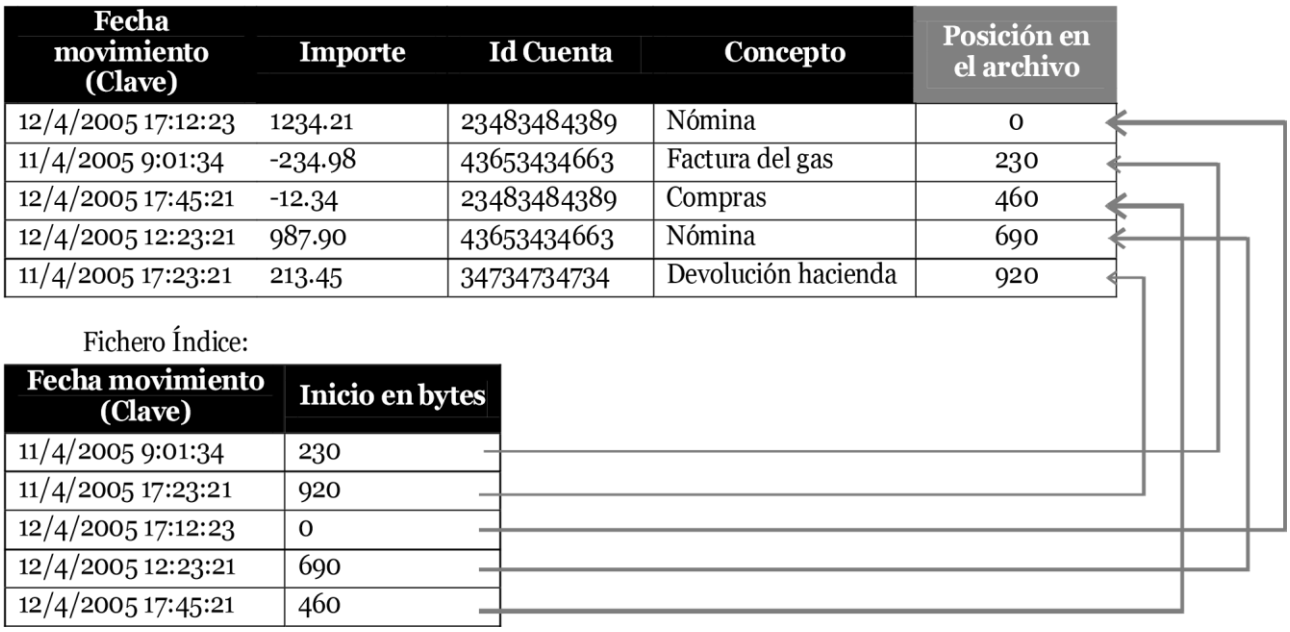


Figura 7.3

En el índice las claves aparecen ordenadas, hay un segundo campo que permite indicar en qué posición del archivo de datos se encuentra el registro con esa clave. El problema es la reorganización del archivo de índices, que se tendría que hacer cada vez que se añade un registro para que aparezca ordenado.

OTROS FORMATOS BINARIOS DE ARCHIVOS

No siempre en un archivo binario se almacenan datos en forma de registros. En muchas ocasiones se almacena otra información. Es el caso de archivos que almacenan música, vídeo, animaciones o documentos de una aplicación.

Para que nosotros desde un programa en C podamos sacar información de esos archivos, necesitamos conocer el **formato de ese archivo.** Es decir como hay que interpretar la información (siempre binaria) de ese archivo. Eso sólo es posible si conocemos dicho formato o si se trata de un formato de archivo que hemos diseñado nosotros.

Por ejemplo en el caso de los archivos **GIF**, estos sirven para guardar imágenes. Por lo que lo que guardan son píxeles de colores. Pero necesitan guardar otra información al inicio del archivo conocida como cabecera.

En la figura 7.4 se observa la información que tiene este formato:

Cabecera de los archivos GIF

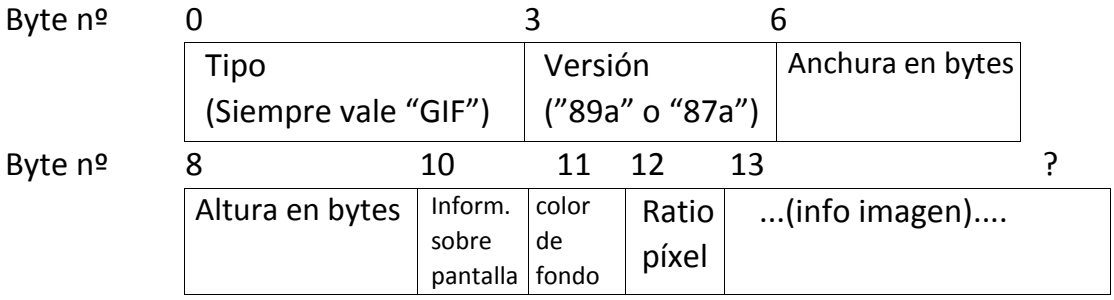


Figura 7.4

Así para leer la anchura y la altura de un determinado archivo GIF desde un programa C y mostrar esa información por pantalla, habría que:

```
#include <stdio.h>
int main(){
   FILE *pArchivo;
   int ancho=0, alto=0;
   /* Aquí se almacena el resultado */
   pArchivo=fopen("archivo.gif","rb");
```

```
    if(pArchivo!=NULL) {
    /* Nos colocamos en el sexto byte, porque ahí está la  información sobre la anchura y después la altura*/
       fseek(pArchivo,6,SEEK_SET);
       fread(&ancho,2,1,pArchivo);
       fread(&alto,2,1,pArchivo);
       printf("Dimensiones: Horizontal %d, Vertical %d\n",ancho,alto);
    }
    return 0;
 }
```

En definitiva para extraer información de un archivo binario, necesitamos conocer exactamente su formato.

UNIDAD DE COMPETENCIAS VIII. ESTRUCTURAS DE DATOS DINÁMICAS

RESUMEN.
En la Unidad de Competencias III, se explicó el manejo de memoria dinámica con arreglos y registros. En la Unidad de Competencias VI se explica el pase de parámetros con matrices dinámicas y apuntadores a funciones.

En esta Unidad de Competencias se tratarán algunos ejemplos con estructuras de datos dinámicas

8.1 **Programación de listas enlazadas**

(Ayala de la Vega, Aguilar Juárez, Zarco Hidalgo, & Gómez Ayala, 2016)

Una lista es una colección de elementos llamados generalmente nodos. El orden entre los nodos se establece por medio de apuntadores (realizando el enlace entre nodos), es decir, direcciones o referencias a otros nodos. Las operaciones más comunes son:

- Recorrido de la lista.
- Inserción de un elemento.
- Borrado de un elemento.
- Búsqueda de un elemento.

Los tipos de datos abstractos más conocidos son la pila, la lista ordenada, la lista doblemente enlazada ordenada y el árbol binario. Y son un ejemplo claro del manejo de memoria en forma dinámica. Por lo que en las siguientes sub-secciones se mostrará la programación de estos elementos.

8.2 Programación de una pila

Una pila (stack en inglés) es una lista de elementos a la cual se pueden insertar o eliminar elementos sólo por uno de los extremos (apilar o retirar un elemento de la pila). En consecuencia, los elementos de una pila serán eliminados en orden inverso al que se insertaron. Es decir, el último elemento que se mete en la pila es el primero que se saca. Debido al orden en el cual se insertan o eliminan elementos de una pila a esta estructura también se le conoce como estructura LIFO (Last-In, First-Out: Último en entrar, primero en salir).
Las pilas pertenecen al grupo de estructuras de datos lineales, porque los componentes ocupan lugares sucesivos en la estructura.
Algunos de los ejemplos donde se utiliza la pila son:

- Evaluación de expresiones en notación posfija
- Reconocedores sintácticos de lenguajes independientes del contexto
- Implementación de la recursividad.

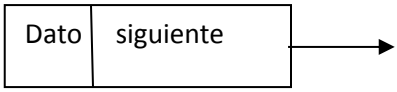
Las operaciones elementales que pueden realizarse en la pila son:

- Colocar un elemento en la pila
- Eliminar un elemento de la pila
- Mostrar la pila

La estructura a usar en el programa es la siguiente:

```
struct nodo{
        int dato;
        nodo * siguiente;
}
```

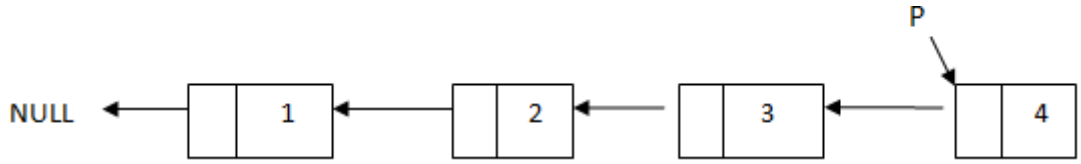
Esta estructura dinámica se puede representar en forma gráfica de la siguiente manera:



Observe que la variable –siguiente- es un apuntador a la misma estructura, esto nos permite poder realizar un enlace con estructuras del mismo tipo (de ahí el nombre de estructuras enlazadas).

En la figura 8.1 se muestra la forma en que se manipula la pila, por lo que el primer elemento introducido es la estructura donde la variable –dato- guarda el valor uno y el último elemento en la pila es la estructura donde la variable –dato- guarda el valor cuatro.

La impresión de la pila será:
4..3..2..1..



Al eliminar un elemento de la pila, ésta se verá de la siguiente forma:

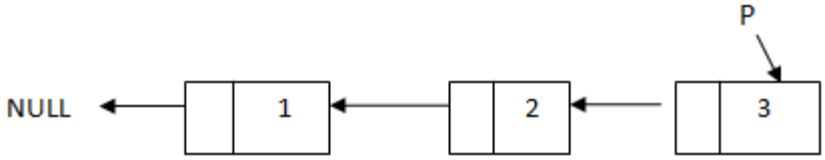
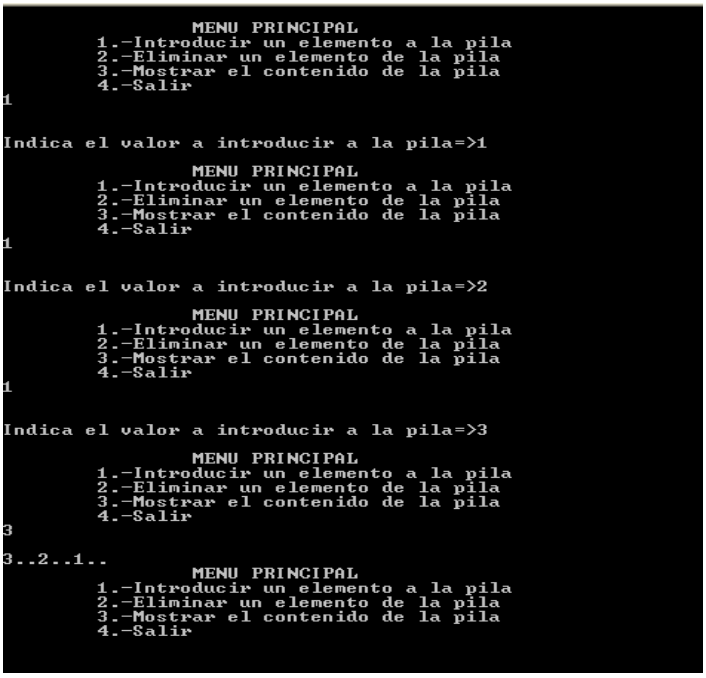


Figura 8.1

El programa 8.1 muestra el código en C de una pila en forma dinámica en el que se permite insertar, mostrar o eliminar elementos:

```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
	int dato;
	nodo * siguiente;
};

//PROTOTIPOS
int menu();
void * crear (void *);
void * eliminar(void *);
void mostrar(void *);

main(){
void *p=NULL;
int eleccion;
do{
	eleccion=menu ();
	switch(eleccion){
		case 1:p=crear(p);continue;
		case 2:p=eliminar(p);break;
		case 3:mostrar(p);continue;
		default:printf("FIN DE LAS OPERACIONES");
	}
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion,x;
do{
	printf("\n\t\tMENU PRINCIPAL\n");
	printf("\t1.-Introducir un elemento a la pila\n");
	printf("\t2.-Eliminar un elemento de la pila\n");
	printf("\t3.-Mostrar el contenido de la pila\n");
	printf("\t4.-Salir\n");
	scanf("%d",&eleccion);
}while(eleccion<1||eleccion>4);
putchar('\n');
return(eleccion);
}
void *crear(void *p){
nodo *q,*aux;
putchar('\n');
```

```
printf(“Indica el valor a introducir a la pila=>”);
q=(nodo*)malloc(sizeof(nodo));
scanf(“%d”,&q->dato);
q->siguiente=NULL;
if(p==NULL)    //La pila está vacía
        p=q;
else{
    q->siguiente=(nodo*)p;
    p=q;
}
return(p);
}

void * eliminar(void * s){
nodo *p,* aux;
if(s==NULL)
        printf(“\npila vacia\n”);
else{
        p=(nodo*)s;
        s=p->siguiente;
        free(p);
}
return(s);
}

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf(“PILA VACIA”);
else
        do{
                printf(“%d..”,s->dato);
                s=s->siguiente;
        }while(s!=NULL);
}
```



Programa 8.1

Para poderse mantener la pila en memoria se requiere un apuntador principal al final de la pila, el apuntador utilizado en el programa es del tipo –void-, se requiere en cada función invocada realizar un casting para la manipulación de la pila. En la función –crear()- se tiene el casting:

q->siguiente=(nodo*)p;

Cada vez que se ejecuta la función –crear()-, en la instrucción:

q=(nodo*)malloc(sizeof(nodo));

la función –sizeof()- indica el tamaño de la estructura –nodo- en bytes. Posteriormente, la función –malloc()- reserva memoria del tamaño indicado en la función –sizeof()- y finalmente la memoria reservada se utilizará como tipo nodo y se regresa un apuntador a una variable del tipo apuntador a nodo.
De esta forma se crea en forma dinámica una estructura del tipo nodo. Y en el código:

q->siguiente=(nodo*)p;

Se enlazan dos estructuras. Observe que la variable –p- es un apuntador del tipo void y la variable –q- es un apuntador del tipo nodo, si se realiza una igualdad de la forma:

q->siguiente=p;

se tendrá un error del tipo mismatch (las variables no son compatibles), esto se entiende ya que se está asignado una variable del tipo general (void) a una variable del tipo específico (nodo *)

8.3 Traducción de una expresión infija a postfija.

Un ejemplo clásico de una pila es el transformar una expresión algebraica infija a postfija. Para comprender que es una expresión infija o postfija se revisarán algunos conceptos:

- Dada la expresión A + B se dice que está en notación infija, y su nombre se debe a que el operador (+) se localiza entre los dos operandos (A, B)
- Dada la expresión AB+ se dice que está en notación postfija, ya que el operador (+) se localiza después de los operandos (A, B).
- Dada la expresión +AB se dice que está en notación prefija ya que el operador (+) se localiza antes de los operandos (A, B).

La ventaja de usar expresiones en notación polaca (o postfija) o prefija radica en que no son necesarios los paréntesis para indicar el orden de los operadores, ya que éste queda establecido por la ubicación del operador con respecto al operando.
Para convertir una expresión dada de notación infija a postfija (o prefija), deberán de establecerse ciertas condiciones:

- Solamente se manejarán los siguientes operadores (están dados ordenadamente de mayor a menor según su prioridad de ejecución):
  ^ (Potencia)
  * / % (multiplicación, división y módulo)
  + - (suma y resta)
- Los operadores de más alta prioridad se ejecutan primero.
- Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procederá a evaluar de izquierda a derecha. (Para cumplir con el punto anterior y el actual, se utilizará una pila para manejar las prioridades en forma adecuada.)
- Las sub expresiones que se localicen dentro de paréntesis tienen más prioridad que cualquier otra operación

En la tabla 8.1 se presenta, paso a paso, un ejemplo de conversión de expresión infija a postfija utilizando una pila como apoyo.

PASO	EXPR. INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	(X+Z)*W/T^Y-V			
1	X+Y)*W/T^Y-V	(	(	
2	+Y)*W/T^Y-V	X	(	X
3	Y)*W/T^Y-V	+	(+	X
4	)*W/T^Y-V	Z	(+	XZ
5	*W/T^Y-V	)		XZ+

6	W/T^Y-V	*	*	XZ+
7	/T^Y-V	W	*	XZ+W
8	T^Y-V	/	/	XZ+W*
9	^Y-V	T	/	XZ+W*T
10	Y-V	^	/^	XZ+W*T
11	-V	Y	/^	XZ+W*TY
12	V	-	-	XZ+W*TY^/
13		V	-	XZ+W*TY^/V-

Tabla 8.1

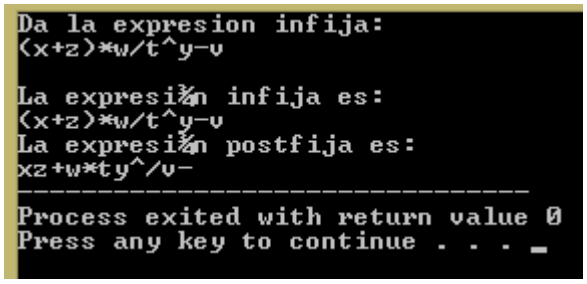
El programa 8.2 muestra el algoritmo para transformar una expresión infija a una expresión postfija. Se introdujo una función llamada “topepila” que permite ver lo que se tiene en tope de pila sin realizar algún movimiento en ella. También se introdujo una función que evalúa la prioridad del operador que se está analizando actualmente con respecto al operador que se localiza en tope de pila. Las funciones “crear” y “eliminar” son muy parecidas a las funciones del mismo nombre al programa 8.1 que maneja una pila. La función “leer” lee un arreglo de caracteres en forma dinámica.

```
//El apuntador principal apunta al último elemento en pila
#include<stdio.h>
#include<stdlib.h>
struct nodo{
        char dato;
        nodo * siguiente;
};

//PROTOTIPOS
int prioridad(char);            //Indica la prioridad de los operadores
char* leer(char[],int);         //Lee un arreglo de caracteres en forma dinámica
void * crear (void *, char);    //introduce un elemento a pila
void * eliminar(void *);        //Elimina un elemento del tope de pila
void imprime(char[]);           //Imprime la expresión infija
void postfija(char[]);          //El algoritmo de transformación de infija a postfija
char topepila(void*);           //Retorna lo que se tiene en tope de pila

main(){
char * a,c;
int i=0;
printf("Da la expresion infija:\n");
do
        a=leer(a,i++);
while(a[i-1]!='\n');
```

```
imprime(a);
postfija(a);
return 0;
}

char * leer(char a[],int i){
	char *b,c;
	b=(char *)malloc(sizeof(char)*(i+1));
	b[i]=getchar();
	for (int j=0;j<i;j++)
		b[j]=a[j];
	free (a);
	return b;
}

void imprime(char x[]){
	int i=0;
	putchar('\n');
	printf("La expresión infija es:\n");
	do
		putchar(x[i++]);
	while(x[i-1]!='\n');
}

void postfija(char a[]){
	void *p=NULL;
	char c;
	int x=0;
	printf("La expresión postfija es:\n");
	while (a[x]!='\n'){
		if (a[x]=='(')
			p=crear(p,'(');
		else if(a[x]==')'){
			while((c=topepila(p))!='('){
				putchar(c);
				p=eliminar(p);
			}
			p=eliminar(p);
		}
		else if(a[x]>='a' && a[x]<='z')
			putchar(a[x]);
		else {
			while (topepila(p)!='&' && prioridad(a[x])<=prioridad(topepila(p))){
				putchar(topepila(p));
```

```
                        p=eliminar(p);
                    }
                    p=crear(p,a[x]);
                }
                x++;
        }
        while ((c=topepila(p))!='&'){
                putchar(c);
                p=eliminar(p);
        }
}

int prioridad(char c){
        int x;
        switch(c){
                case '^':x=3; break;
                case '*':
                case '/':
                case '%':x=2; break;
                case '+':
                case '-':x=1; break;
                case '(':x=0;break;
        }
        return x;
}


void *crear(void *p, char a){
nodo *q;
q=(nodo*)malloc(sizeof(nodo));
q->siguiente=NULL;
q->dato=a;
if(p==NULL)    //La pila está vacía
        p=q;
else{
    q->siguiente=(nodo*)p;
    p=q;
}
return(p);
}


char topepila(void *p){
        nodo *q=(nodo*)p;
```

```
        if (p==NULL)
                return '&';
        else
                return q->dato;
}

void * eliminar(void * s){
nodo *p;
if(s==NULL)
        printf("\npila vacia\n");
else{
        p=(nodo*)s;
        s=p->siguiente;
        free(p);
}
return(s);
}
```



Programa 8.2

8.4 PROGRAMACIÓN DE UNA LISTA SIMPLEMENTE ENLAZADA ORDENADA.

Una lista ordenada es una lista enlazada con un orden, siendo el orden una estructura interna que debe mantenerse en todo momento.
Las operaciones elementales que pueden realizarse en la lista ordenada son:

- Colocar un elemento en la lista
- Eliminar un elemento de la lista
- Mostrar la lista

Para colocar un elemento en la lista ordenada se deben de observar cuatro posibles casos:

- La lista está vacía.
- El elemento a colocar es menor a todos los elementos que se encuentran en la lista
- El elemento a colocar es mayor a todos los elementos que se encuentran en la lista
- El elemento a colocar debe de estar entre la lista.

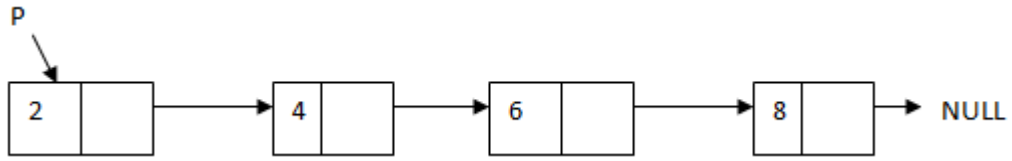
Cada caso se observa en la función –crear()- del siguiente programa.

Para eliminar un elemento de la lista se tienen los siguientes casos:

- El elemento a eliminar es el primero y único de la lista
- El elemento a eliminar es el primero y no es el único
- El elemento a eliminar es el último de la lista
- El elemento a eliminar se encuentra entre la lista.

Cada caso se observa en la función –eliminar()- del siguiente programa.

Por lo que el programa 8.3 ordena los elementos de menor a mayor creando el espacio del elemento a guardar en forma dinámica. Además, permite eliminar un elemento sin perder el orden establecido:




```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
        int dato;
        nodo * siguiente;
};

//PROTOTIPOS
int menu();
void * crear (void * );
void * eliminar(void * );
void mostrar(void * );

main(){
void *p=NULL;
int eleccion;
do{
        eleccion=menu();
        switch(eleccion){
                case 1:p=crear(p);continue;
                case 2:p=eliminar(p);break;
                case 3:mostrar(p);continue;
                default:printf("FIN DE LAS OPERACIONES");
        }
```

```
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion,x;
do{
        printf("\n\t\tMENU PRINCIPAL\n");
        printf("\t1.-Introducir un elemento a la lista\n");
        printf("\t2.-Eliminar un elemento de la lista\n");
        printf("\t3.-Mostrar el contenido de la lista\n");
        printf("\t4.-Salir\n");
        scanf("%d",&eleccion);
}while(eleccion<1||eleccion>4);
putchar('\n');
return(eleccion);
}

void *crear(void *p){
nodo *q,*aux,*aux1;
int x;
putchar('\n');
printf("Indica el valor a introducir a la lista=>");
scanf("%d",&x);
q=(nodo*)malloc(sizeof(nodo));
q->dato=x;
q->siguiente=NULL;
if(p==NULL)             //Lista vacía
        p=q;
else{
        aux1=aux=(nodo*)p;
        if (x<aux->dato){       //Elemento menor a cualquier elemento de la lista
                q->siguiente=aux;
                p=q;
        }
        else{   //el elemento se colocará entre la lista o será el último
                while(aux!=NULL&&aux->dato<x){
                        aux1=aux;
                        aux=aux->siguiente;
                }
                aux1->siguiente=q;
                q->siguiente=aux;

        }
```

```
}
return(p);
}


void * eliminar(void * s){
if(s==NULL)
        printf("\nLISTA VACIA\n");
else{
        nodo *p,* aux;
        int x;
        printf("Da el elemento a eliminar=>");
        scanf("%d",&x);
        aux=p=(nodo*)s;
        if(p->siguiente==NULL&&aux->dato==x)//solo hay un elemento en la lista
                s=NULL;
        else {
                while(p->siguiente!=NULL && p->dato<x){
                        aux=p;
                        p=p->siguiente;
                }
                if (p!=NULL)
                        if (p->dato==x && p==s) //No existió movimiento
                                s=p->siguiente;
                        else if(p->dato==x)
                                aux->siguiente=p->siguiente;
                        else
                                printf("DATO NO LOCALIZADO\n");
                else
                        printf("DATO NO ENCOTRADO\n");
        }
        free(p);
}
return(s);
}

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf("LISTA VACIA");
else
        do{
                printf("%d..",s->dato);
```

```
            s=s->siguiente;
        }while(s!=NULL);
}
```

Enseguida se muestra una corrida del programa en el cual se muestra la introducción de los números 3, 1 y 2 en la lista:

Programa 8.3

8.5 ARREGLO DE APUNTADORES

El programa 8.4 simula una agenda telefónica donde se guarda el nombre en forma alfabética, dirección y teléfono de personas. Cada nombre se guarda en un arreglo de caracteres cuyo tamaño se determina en forma dinámica y es guardado en una lista simplemente enlazada en orden alfabético, por lo que existirá un apuntador para cada letra del alfabeto (se crea un arreglo de 26 apuntadores tipo void, y cada apuntador apuntará a una lista simplemente enlazada donde contendrá a nombres ordenados de personas que inicien con la misma letra). El menú permite introducir

los nombres con su dirección y teléfono y se guardarán en forma alfabética, mostrará todos los nombres ordenados en forma alfabética o los nombres que inician con una letra en particular o eliminar un nombre en particular junto con el número telefónico y dirección.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct agenda{
	char * nombre;
	char * direccion;
	char telefono[11];
	struct agenda * siguiente;
};

//PROTOTIPOS
int menu();
void crear(void *[]);
void mostrar(void * []);
void mostrartodo(void *[]);
void eliminar(void * []);

main(){
	int i,x;
	void * p[26];
	for (i=0;i<26;i++)
	p[i]=NULL;
	do{
		x=menu();
		getchar();
		switch(x){
			case 1: crear(p); break;
			case 2: mostrar(p); break;
			case 3: mostrartodo(p); break;
			case 4: eliminar(p); break;
	   }
     }while(x<5);
}

void eliminar(void * s[]){
```

```
        agenda *p,* aux;
        char x[80];
        char * nombre;
        int i,j;
        do{
           printf("Dar el nombre a eliminar:");
           for (i=0;(x[i]=getchar())!='\n';i++);
      x[i]='\0';
    }while(x[0]<'A' || x[0]>'Z');
    nombre=(char *)malloc(i);
         for (j=0;j<=i;j++)
            nombre[j]=x[j];
        j=x[0]-'A';
        aux=p=(agenda*)s[j];
        if (aux==NULL)
          printf("No hay elementos que inicien con %c\n",x[0]);
        else if (p->siguiente==NULL && strcmp(aux->nombre, nombre) ==0 )
                        s[j]=NULL;
        else {
                while(p->siguiente!=NULL && strcmp(p->nombre, nombre)<0){
                        aux=p;
                        p=p->siguiente;
                }
                if (p!=NULL)
                        if (strcmp(p->nombre, nombre)==0 && p==s[j])
                                s[j]=p->siguiente;
                        else if(strcmp(p->nombre, x)==0 )
                                aux->siguiente=p->siguiente;
                        else
                                printf("DATO NO LOCALIZADO\n");
                else
                        printf("DATO NO ENCOTRADO\n");
        }
}

void crear(void * p[]){
    int i,j;
    char n[80];
    struct agenda * q, * aux, * aux1;
    q=(agenda *)malloc(sizeof(agenda));
```

```
q->siguiente=NULL;
do{
        printf("Dar el nombre con mayusculas:");
        for (i=0;(n[i]=getchar())!='\n';i++);
        n[i]='\0';
}while(n[0]<'A' || n[0]>'Z');
q->nombre=(char *)malloc(i);
for (j=0;j<=i;j++)
         q->nombre[j]=n[j];

do{
        printf("Dar la direccion:");
        for (i=0;(n[i]=getchar())!='\n';i++);
   n[i]='\0';
}while(n[0]<'A' || n[0]>'Z');
q->direccion=(char *)malloc(i);
for (j=0;j<=i;j++)
         q->direccion[j]=n[j];

printf("Da el telefono con 10 digitos");
gets(q->telefono);
printf("%s\t%s\t%s\n",q->nombre,q->direccion,q->telefono);
i=q->nombre[0]-'A';
aux=(agenda *)p[i];
if (p[i]==NULL)
        p[i]=q;
else{
          if(strcmp(q->nombre, aux->nombre)<0){
                  q->siguiente=aux;
                  p[i]=q;
          }
          else{
                   while (aux!=NULL && strcmp(q->nombre, aux->nombre) >0 ){
                        aux1=aux;
                                aux=aux->siguiente;
         }
              aux1->siguiente=q;
              q->siguiente=aux;
   }
     }
```

```
}

void mostrar (void * s[]){
    char x;
    agenda * aux;
    do{
            printf("Da la letra del alfabeto que quieres mostrar:");
            scanf("%c",&x);
    }while(x<'A' || x>'Z');
    aux=(agenda *) s[x-'A'];
    if (aux==NULL)
            printf("No existen nombres que inicien con la letra %c.\n",x);
    else{
        printf("NOMBRE\tDIRECCION\tTELEFONO\n");
              do{
                        printf("%s\t%s\t%s\n",aux->nombre,aux->direccion,aux-
>telefono);
                        aux=aux->siguiente;
         }while (aux!=NULL);
         getchar();
    }
}

void mostrartodo (void * s[]){
    char x;
    agenda * aux;
    printf("NOMBRE\tDIRECCION\tTELEFONO\n");
    for (int i=0;i<26;i++){
       aux=(agenda *) s[i];
       if (aux!=NULL){
          do{
                        printf("%s\t%s\t%s\n",aux->nombre,aux->direccion,aux-
>telefono);
                        aux=aux->siguiente;
         }while (aux!=NULL);
         getchar();
       }
    }
}
```

```
int menu(){
    int x;
    do{
       printf("AGENDA DE CUATES\n");
       printf("1. INTRUDUCIR AMIGO\n");
       printf("2. MOSTRAR AMIGOS QUE INICIEN CON LA MISMA LETRA DEL
ALFABETO\n");
       printf("3. MOSTRAR TODOS LOS AMIGOS EN ORDEN ALFABETICO\n");
       printf("4. ELIMINAR DE LA AGENDA A UN ELEMENTO\n");
       printf("5. Salir de la aplicacion\n");
       printf("INDIQUE SU ELECCION:");
       scanf("%d",&x);
    }while (x<1 || x>5);
    return x;
}
```

Programa 8.4

8.6 APUNTADORES A FUNCIONES.

El programa 8.5 ordena un vector de enteros, haciendo uso de una función de comparación y una para cambiar los datos. La función de ordenamiento recibe un apuntador a una función.

```
#include <stdio.h>
#include <conio.h>
 // Función para comparar dos valores. Retorna 1 o -1
int Comparacion(int valor1,int valor2){
   return valor1 > valor2 ? 1 : -1;
}
// Función para intercambiar el valor de dos posiciones en un Vector
void CambiarDatos(int posI,int posJ,int *Vector){
    int aux=Vector[posI];
    Vector[posI]=Vector[posJ];
    Vector[posJ]=aux;
}
// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (*funcion)(int , int), int *Vector,int maximo){
    for (int i=0;i<maximo-1;i++){
       for (int j=i+1;j<maximo;j++){
          if (funcion(Vector[i],Vector[j])>0)
            CambiarDatos(i,j,Vector);
```

```
        }
    }
}
// Función para imprimir un vector
void Imprime_Vector(int *Vector,int maximo){
    for (int i=0;i< maximo;i++)
        printf("%2d\n",Vector[i]);
}
main(){
      int Vector[10]={5,76,4,8,50,2,44,7,23,6};
      printf("Original: \n");
            Imprime_Vector(Vector,10);
            Ordenamiento(Comparacion,Vector,10);
            printf("Ordenado: \n");
            Imprime_Vector(Vector,10);
            getchar();
}
```

Programa 8.5

Se reescribe el mismo ordenamiento utilizando la variable Vector como un apuntador y se utiliza aritmética de apuntadores para desplazarse en las posiciones en la memoria (Programa 8.6):

```
#include <stdio.h>
#include <conio.h>
// Función para comparar dos valores. Retorna 1 o -1
int Comparacion(int *valor1, int *valor2){
            return *valor1 > *valor2 ? 1 : -1;
}
// Función para intercambiar dos valores
void CambiarDatos(int *datoI, int *datoJ){
            int aux=*datoI;
            *datoI=*datoJ;
            *datoJ=aux;
}
// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (*funcion)(int * , int *), int *Vector,int maximo){
            int *vi,*vj;
            vi =Vector;
            for (int i=0; i<maximo-1; i++){
```

```
            vj = vi;
            vj++;
            for (int j=i+1; j<maximo; j++){
          if (funcion(vi,vj) > 0)
                CambiarDatos(vi,vj);
              vj++;
            }
            vi++;
          }
}

// Función para imprimir un vector
void Imprime_Vector(int *Vector,int maximo){
          int *vi;
          vi = Vector;
          for (int i=0;i<maximo;i++)
          {
            printf("%2d\n",*vi);
            vi++;
          }
        }

int main(){
          int Vector[10]={5,76,4,8,50,2,44,7,23,6};
          printf("Original: \n");
          Imprime_Vector(Vector,10);
          Ordenamiento(Comparacion,Vector,10);
          printf("Ordenado: \n");
          Imprime_Vector(Vector,10);
          getchar();
        }
```

Programa 8.6

8.7 MATRICES DE APUNTADORES A FUNCIÓN

Antes de continuar, se puede comentar que una matriz permite guardar datos del mismo tipo, por lo que se puede utilizar una matriz para guardar apuntadores a función.
Por ejemplo:

Primero definamos un apuntador a función que recibe dos parámetros del tipo entero y retorna un entero:
int (*ptrFn) (int, int);

A partir de la definición anterior, se definirá una matriz de cinco apuntadores a función:
int (*ptrFn[5]) (int,int);

Ahora, observe el siguiente código de ordenamiento utilizando el método de la burbuja (Programa 8.7):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct StPersona{
        int edad;
        int Salario;
        char *Nombre;
};

// Función para comparar por edad. Retorna 1 o -1
int ComparacionxEdad(StPersona  *Persona1, StPersona *Persona2){
        return Persona1->edad > Persona2->edad ? 1 : -1;
}

// Función para comparar por salario. Retorna 1 o -1
int ComparacionxSalario(StPersona  *Persona1, StPersona *Persona2){
        return Persona1->Salario > Persona2->Salario ? 1 : -1;
}

// Función para comparar por nombre. Retorna 1 o -1
int ComparacionxNombre(StPersona  *Persona1, StPersona *Persona2){
   return strcmp(Persona1->Nombre, Persona2->Nombre);
}

// Función para intercambiar dos valores
void CambiarDatos(StPersona *datoI, StPersona *datoJ){
        StPersona aux = *datoI;
        *datoI = *datoJ;
        *datoJ = aux;
```

```
}

// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (*funcion)(StPersona * , StPersona *), StPersona
*Vector,int maximo){
        StPersona *vi,*vj;
        vi = Vector;
            for (int i=0; i < maximo-1; i++){
            vj=vi;
            vj++;
            for (int j=i+1;j<maximo;j++){
                if (funcion(vi,vj)>0)
                   CambiarDatos(vi,vj);
            vj++;
            }
              vi++;
         }
}

// Función para imprimir un vector
void Imprime_Vector(StPersona *Vector,int maximo){
        StPersona *vi;
        vi = Vector;
        for (int i=0; i<maximo; i++){
            printf("Edad : %d\n", vi->edad );
            printf("Nombre : %s\n", vi->Nombre);
            printf("Salario : %d\n", vi->Salario );
            vi++;
        }
}

//Función que permite leer un nombre
char * nombre(){
    char * nomb;
    char a[50];
    int i;
    for (i=0;(a[i]=getchar())!='\n';i++);
    a[i]='\0';
    nomb=(char *)malloc(sizeof(char)*i);
    strcpy(nomb,a);
```

```
    return nomb;
}
int main(){
      char *b[3], a[80];
      b[0]="edad";
      b[1]="salario";
      b[2]="nombre";
            StPersona *Vector;
            Vector=new StPersona[5];
            int i;
            for (i=0;i<5;i++){
                  printf ("Edad : ");
                  scanf("%d", &Vector[i].edad);
                  printf ("Salario : ");
                  scanf("%d", &Vector[i].Salario);
                  getchar();
                  printf ("Nombre : ");
                  Vector[i].Nombre=nombre();
            }
            int (*ptrFn[3]) (StPersona *, StPersona *); // Matriz de 3 punteros a
funciones
            printf("Original: \n");

            Imprime_Vector(Vector,5);
            ptrFn[0] = ComparacionxEdad;
            ptrFn[1] = ComparacionxSalario;
            ptrFn[2] = ComparacionxNombre;

//Invocacion a cada uno de los ordenamientos
printf("\n ****************************************** \n");
            for (i=0;i<3;i++)
            {
                  Ordenamiento(ptrFn[i], Vector,5);
                  printf("\nOrdenado por   %s: \n", b[i]);
                  Imprime_Vector(Vector,5);
            }
            getchar();
}
```

Programa 8.7

En el código se han escrito 3 funciones de ordenamiento, los cuales son invocados en el último “for”. Observe que en la función -Ordenamiento()- se envía como parámetro un elemento de una matriz (ptrFn[]) que es un apuntador de función.

8.8 APUNTADORES A FUNCIONES COMO PARÁMETROS DE FUNCIONES DE LA BIBLIOTECA ESTANDAR EN C.

La función de biblioteca *–qsort()-* localizado en -stdlib- es muy útil y está diseñada para ordenar un arreglo usando un valor como llave de cualquier tipo para ordenar en forma ascendente.

El prototipo de la función *–qsort()-* es:
*void qsort(void *base, size_t nmiemb, size_t tam, int (*149ísico149r)(const void *, const void*));*

El argument *–base-* apunta al comienzo del vector que será ordenado, *-nmiemb-* indica el tamaño del arreglo, *-tam-* es el tamaño en bytes de cada elemento del arreglo y el argumento final *–comparar-* es un apuntador a una función.

La función *–comparar-* debe regresar un determinado valor entero de acuerdo al resultado de la comparación que debe ser:

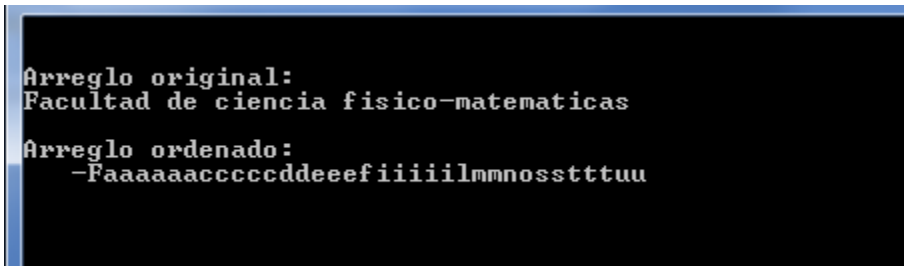
Menor que cero: Si el primer valor es menor que el segundo.
Cero: Si el primer valor es igual que el segundo.
Mayor que cero: Si el primer valor es mayor que el segundo.

Observe que en la función *–comparar()-* la define el usuario y en el momento de la invocación se envía el nombre sin parámetros (en la misma forma en que se envía como parámetro una matriz). La función *–comparar-* tiene como parámetros a dos constantes del tipo apuntador tipo void para que la función *–qsort-* se le permita manipular a cualquier tipo de variable (Programa 8.8).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int comp(const void *I, const void *j);
main(){
int i;
char cad[]=”Facultad de ciencia 149ísico-matemáticas”;
printf(“\n\nArreglo original:\n”);
```

```
for (i=0;i<strlen(cad);i++)
        printf("\%c",cad[i]);
qsort(cad,strlen(cad),sizeof(char),comp);
printf("\n\nArreglo ordenado:\n");
for (i=0;i<strlen(cad); i++)
        printf("%c", cad[i]);
printf("%c");
getchar();
}

int comp(const void *i, const void *j){
char *a, *b;
a=(char *)i;
b=(char *)j;
return *a - *b;
}
```

Este programa imprime lo siguiente:



Programa 8.8

Observe que dentro de la función –comp()- se realiza un –*cast*- para forzar el tipo –*void* *- al tipo –*char* *-.

Bibliografía

1. Alegsa. (23 de 07 de 2018). *ALEGSA*. Obtenido de Enunciados break y continue en C: http://www.alegsa.com.ar/Notas/107.php

2. *Algoritmia*. (18 de 07 de 2018). Obtenido de Elementos de un Lenguaje de Programación: https://algoritmiafordummies.wikispaces.com/3.Elementos+de+los+lenguajes+de+programaci%C3%B3n+y+de+los+algoritmos

3. *Aprenderaprogramar.com. Didactica y divulgación de la programación*. (19 de 07 de 2018). Obtenido de Tipos de datos en C. Declarar variables enteras int, long, o decimal float, double. char. Inicialización (CU00510F): https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=899:tipos-de-datos-en-c-declarar-variables-enteras-int-long-o-decimal-float-double-char-inicializacion-cu00510f&catid=82&Itemid=210

4. Ayala de la Vega, J., Aguilar Juárez, I., Zarco Hidalgo, A., & Gómez Ayala, H. (2016). *Memoría dinámica en el lenguaje de programación C.* Guadalajara: Cenid.

5. Backman, k. (2012). *Structured Programming with C++.* Bookboom.com.

6. Backman, K. (2012). *Structured Programming with C++.* Bookboom.com.

7. Benemerita Universidad Autónoma de Puebla. (26 de 07 de 2018). *Facultad de Ciencias de la Computación*. Obtenido de Prcesamiento de Archivos en Lenguaje C: https://www.cs.buap.mx/~mrodriguez/MIS%20FRAMES/Files2.pdf

8. Ceballos, J. (2015). *C/C++ Curso de Programación.* México: AlfaOmega Ra-Ma.

9. Ceballos, J. (2015). *C/C++ Curso de Programación.* México: AlfaOmega Ra-Ma.

10. Deitel, P. J., & M, D. H. (1995). *Cómo Programar en C/C++.* México: Prentice Hall.

11. Deitel, P., & Deitel, H. M. (1995). *Como programar en C/C++.* México: Prentice Hall.

12. *fergarciac*. (19 de 07 de 2018). Obtenido de Entorno de Desarrollo Integrado (IDE).: https://fergarciac.wordpress.com/2013/01/25/entorno-de-desarrollo-integrado-ide/

13. *Fundamentos de Informática*. (19 de 07 de 2018). Obtenido de Primer Curso de Ingenieros Químicos. Compilador de C para Windows: http://www.esi2.us.es/~mlm/FIQ_P/Prac_FIQ_01.pdf

14. *http://onaprogest.blogspot.com/*. (18 de 07 de 2018). Obtenido de Historia de la programación estructurada: http://onaprogest.blogspot.com/2011/08/historia-de-la-programacion.html

15. Juan Manuel Muñoz, J. S. (19 de 07 de 2018). *Headsem*. Obtenido de Mejores Entornos de Desarrollo Integrado para programar en C/C++: http://www.headsem.com/mejores-ide-para-programar-en-c/

16. Kuhn, T. S. (1962). *La estructura de las revoluciones científicas.* México: Fondode Cultura Económica.

17. Lenguajes de Programación. (18 de 07 de 2018). Obtenido de Herramientas de Programación: http://www.lenguajes-de-programacion.com/herramientas-de-programacion.shtml

18. McGraw-Hill Education. (s.f.). Obtenido de Programación estructurada: https://www.mheducation.es/bcv/guide/capitulo/8448148703.pdf

19. Modelling Ambient Intelligence Research Lab. (18 de 07 de 2018). Obtenido de HERRAMIENTAS Y ENTORNOS: http://mami.uclm.es/rhervas/images/downloads/HyEP-Tema2-TecCASE-Part1Low.pdf

20. Plauger, P. J. (1992). *The Standard C Library.* New Jersey: Prentice Hall.

21. Rodríguez, C. V. (18 de 07 de 2018). *Departamento de Informática, Universidad de Valladolid*. Obtenido de https://www.infor.uva.es/~cvaca/asigs/docpar/intro.pdf

22. Universidad de Castilla-La Mancha. (23 de 07 de 2018). *Arquitectura y Redes de Computadores. Investigamos. Construimos. Compartimos.* Obtenido de Programas más claros gracias a enum: http://arco.inf-cr.uclm.es/~david.villa/pensar_en_C++/vol1/ch03s08s03.html

23. Universidad Tecnológica de Pereira. (23 de 07 de 2018). Obtenido de Lenguaje de programación Dev C++: http://blog.utp.edu.co/jnsanchez/files/2018/04/programacion-Devian-C.pdf

24. Universidad Tecnológica Metropolitana. (26 de 07 de 2018). *Informatica.utem.cl*. Obtenido de Manejo de Archivos en C.: http://informatica.utem.cl/~mcast/PROGRAMACION/PROGRAV/2007/ARCHIVOS/archivos-1.pdf

25. Wikipedia. (18 de 06 de 2018). Obtenido de GOTO: https://es.wikipedia.org/wiki/GOTO

ANEXO 1.

El proposito de este anexo es instroducir un programa en el cual se permita manejar las herramientas de lo que se ha visto en el curso. Para eso se emplearán los conceptos de Métodos Numéricos que permitan.

- Obtener soluciones de ecuaciones polinomiales de la forma $a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_nx^0 = 0$
- Solución de ecuaciones lineales por el método de Gauss.
- Solución del determinante por el método de Gauss.
- Obtención de los valores y vectores característicos

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

//PROTOTIPOS PARA ECUACIONES POLINOMIALES
void newton(void);
void solucion(float*, int);
void imprime_coeficientes(float*, int);
float *leen(float*, int);
void escriben(float *, int);
void descartes(float*, int);
void grafica(int, int, int);
int *genera_factores(float, int*, int*);
void imprime_factores(int*, int);
float divide(float*, float, int);
float doblediv(float*, float, int);
void r_enteras(float*, int*, int, int);
void r_reales(float*, int*, int, int);
//PROTOTIPOS PARA ECUACIONES LINEALES
int gauss(float**, int, int, int*);
float **lee(float**, int, int*);
void imprime(float**, int, int);
void resultado(float **, int, int);
int menu(void);
void determinante(void);
void sistecuaciones(void);
//PROTOTIPOS PARA ECUACIONES Y VALORES CARACTERISTICOS
void caracteristico(void);
void resultado1(float *, int);
void multiplica(float **, float *, int);
void acomoda(float **, float *, int, int);

int main() {
   setbuf(stdout, NULL);
   //se defne un valor para solicitar opcion del menu
   int x;
   //realiza loop minetras no se obtenga un valor valido para el menu
   do {
      x = menu();
      getchar();
```

```
    } while (x < 5);
    //sale de main
    return 0;
    return (EXIT_SUCCESS);
}

int menu() {
    int x;
    //muestra menu
    printf("          MENU\n");
    printf("1.- SoluciÂ¢n de polinomios.\n");
    printf("2.- SoluciÂ¢n de ecuaciones lineales.\n");
    printf("3.- CÂ lculo del determinante.\n");
    printf("4.- EcuaciÂ¢n caracteristica, valores caracteristicos.\n");
    printf("5.- Salir del sistema.\n");
    printf("Seleccione su opciÂ¢n=>\n");
    //lee valor de entrada para opcion del menu
    scanf("%d", &x);
    switch (x) {
        case 1:newton();
            break;
        case 2:sistecuaciones();
            break;
        case 3:determinante();
            break;
        case 4:caracteristico();
    }
    return (x);
}
//FUNCIONES PARA ECUACIONES POLINOMIALES

void newton() {
    //define variables a utilizar en el programa
    //p -> arreglo donde se almacerÃ¡ los coeficientes del polinomio
    //max -> constante que define el tamaÃ±o maximo del arreglo
    float *polinomio = NULL;

    int grado_del_polinomio;
    printf("DA EL GRADO DEL POLINOMIO=>");
    scanf("%d", &grado_del_polinomio);
    polinomio = leen(polinomio, grado_del_polinomio);
    //se envia el arreglo polinomio y el grado para que se procesen y se genere una solucion
    solucion(polinomio, grado_del_polinomio);
}

void solucion(float *polinomio, int grado_del_polinomio) {
    //variables
    //nun_fact -> numero de factores
    //fact -> es el arreglo de los fatores que se generara¡
    printf("EL POLINOMIO QUE ME DISTE ES:\n");
    //imprimira el polimonio
    imprime_coeficientes(polinomio, grado_del_polinomio);
    //realizarÃ¡ elanalisis de la cantida de raices positivas, negativas o complejas existen
    descartes(polinomio, grado_del_polinomio);
    //se obtendran la cantidad de factores
    //envia el ultimo elemento del polinomio
```

```
    int *factores = NULL;
    int num_factores = 0;
    factores = genera_factores(*(polinomio + grado_del_polinomio), factores, &num_factores);
    //imprime los fatores encontrados
    imprime_factores(factores, num_factores);
    //evaluara¡ las raices, para encontrar aquellas que son enteras
    r_enteras(polinomio, factores, grado_del_polinomio, num_factores);
    //evaluara¡ las raices para encontrar las que son reales
    r_reales(polinomio, factores, grado_del_polinomio, num_factores);
    getchar();
}

void r_reales(float *polinomio, int *factores, int grado_del_polinomio, int num_factores) {
    //j -> servira para iterar el arreglo de factores
    //bandera -> servira para mostrar el titulo solo la primera vez
    int j = 0, bandera;
    //a -> almacenara el valor de f(x)
    //e -> almacenarÃ¡ el error
    //a1 -> almacenarÃ¡ el valor de f(x1)
    //error -> diferencia entre la raiz en la iteracion i y la anterior
    //error1 -> almacena el error de la iteracion anterior
    float a, e, a1, x, x1, error, error1;
    printf("\n********** RAICES REALES **********\n");
    printf("Da el error a aceptar:");
    scanf("%f", &e);
    printf("            x            x1           f(x)           f(x1)\n");
    //itera por todos los factores
    do {
        //define el valor del error tan grande que la primera vez pueda comparar
        error1 = 100000;
        //servira para identificar si error dismunuye
        bandera = 0;
        //amacena el factor en una variable
        x = *(factores + j);
        //almacena el factor siguiente en otra variable
        x1 = *(factores + j + 1);
        //obtiene los valores de a, por medio de divison sintetica
        a = divide(polinomio, x, grado_del_polinomio);
        a1 = divide(polinomio, x1, grado_del_polinomio);
        printf("  %13G   %13G   %13G    %13G\n", x, x1, a, a1);
        //si los valores indican cambio de signo, hay raiz en ese intevalo
        if (a * a1 < 0) {
            //evalua punto medio de las raices
            x = (x + x1) / 2;
            //aqui empieza el metodo de newton
            do {
                //realiza la formula de newton
                x1 = x - dobledivv(polinomio, x, grado_del_polinomio);
                //calcula la diferencia entre la raiz calculada y la anterior
                error = x - x1;
                //realiza el valor absoluto
                if (error < 0)
                    error = -error;
                //si el error calculado es mayor que el anterior
                //hay divergencia
                if (error > error1)
```

```
                bandera = 1;
            //se almacenan los valores obtenidos en lugar de las actuales
            error1 = error;
            x = x1;
            //en caso de haber haber divergencia
            if (bandera == 1)
                printf("EL METODO DIVERGE EN EL INTERVALO\n");
            //continua iterando en caso de que el error disminuya y no se levante la bandera
        } while (error > e && bandera == 0);
        printf("RAIZ=%f\n", x1);
    }
    j++;
} while (j < num_factores - 1);
}

void r_enteras(float *polinomio, int *factores, int grado_del_polimonio, int num_factores) {
    //variables
    //bandera -> usada en caso de encontrar la primera raiz entera
    //j -> servira para iterar en el arreglo de los fatores
    int bandera = 0, j = 0;
    //a ->
    //b -> almacena cada factor, comenzando pro el ultimo factor
    float a, b;
    //recorre el arreglo de factores
    do {
        b = *(factores + j);
        //realiza la divison para identificar las raices
        a = divide(polinomio, b, grado_del_polimonio);
        //si el resultado de la divion sintetica es 0, definimos que es una raiz
        if (a == 0) {
            //se ejecuta solo la primera vez
            if (bandera == 0) {
                printf("\n*** RAICES ENTERAS ***\n");
                bandera = 1;
            }
            //imprime la raiz
            printf("RAIZ ENTERA:%d\n", *(factores + j));
        }
        j++;
    } while (j <= num_factores);
}

float divide(float *p, float x, int n) {
    //a -> es inicializado con el primer
    //x -> es el factor que se evalua en la division
    //n -> grado
    float a = *(p);
    int i;
    for (i = 1; i <= n; i++)
        //a ira acumulando los resultados de la division sintetica
        a = a * x + *(p + i);
    //a -> f(x)
    return (a);
}

float doblediv(float *p, float f, int n) {
```

```
    //c -> es f'(x)
    //a -> f(x)
    //b -> es auxiliar, donde se va almacenando f'(x)
    //i -> servira para iterar
    float c, a = *p, b = *p;
    int i;
    for (i = 1; i <= n; i++) {
        a = a * f + *(p + i);
        b = b * f + a;
        if (i == n - 1)
            c = b;
    }
    //regresa f(x)/f'(x)
    return (a / c);
}

void imprime_factores(int *fac, int n) {
    printf("Los factores posibles son:\n");
    int i;
    for (i = 0; i < n; i++)
        printf("%d..", *(fac + i));
}

int *genera_factores(float f, int *factores, int *num_fatores) {
    //f -> el ultimo coeficiente del polimonio
    //factor -> es el arreglo donde se almacenaran los factores que se genran
    //si el valor de el ultimo coeficiente es menor a 0, cambio de signo
    if (f < 0)
        f = -f;
    //convierte el ultimo coeficiente a un entero
    int i = (int) f;
    //cont -> contendra la cantidad de factores
    //j -> servira para mantener el valor inicial de i
    int cont, j = i;
    printf("********** FACTORES *******\n");
    cont = 0;
    printf("El nÂ£mero a factorar es:%d\n", j);
    //mientras i sea mayor que 1, generando factores
    while (i >= 1) {
        //obtiene el modulo, para saber si i es un factor de j
        if (j % i == 0) {
            //guarda el valor de i(factor encontrado) en el arreglo
            factores = (int *) realloc((int *) factores, ((cont + 1) * 2) * sizeof (int));
            factores[cont] = i;
            cont++;
        }
        i--;
    }
    //almacenar los factores dentro del arreglo como negativos
    //k -> servira para iniciar en la ultima posicion de los fatores generados
    int k = cont - 1;
    for (i = cont; i < 2 * cont; i++) {
        *(factores + i) = -(*(factores + k));
        k--;
    }
    printf("El nÂ£mero de factores es:%d\n", i);
```

```
    //regresa la cantidad de factores encontrados por 2
    *num_fatores = 2 * cont;
    return factores;
}

void descartes(float *p, int n) {
    //variables
    //i -> itera en el arreglo
    //cont -> servirÃ¡ para contar los cambios de signo entre los coefientes
    int i, cont = 0;
    //pmenos -> copia del polimnomio donde se iran cambiando de signo
    //simulando el cambio de x -> -x
    float a[n], *pmenos;
    pmenos = a;
    //identifica si el grado es par o impar
    if (n % 2 == 0)
        //si es par, el contador iniciarÃ¡ en 1, para garantizar que comienza en potencia impar
        cont = 1;
    //realiza una copia de p en pmenos
    for (i = 0; i <= n; i++)
        *(pmenos + i) = *(p + i);
    //cambiar de signo los coeficientes de las potencias impares
    do {
        *(pmenos + cont) = -(*(p + cont));
        cont += 2;
    } while (cont < n);

    printf("\nEL POLINOMIO NEGATIVO ES:\n");
    //imprime los nuevos valores de los coeficientes
    imprime_coeficientes(pmenos, n);
    //pos -> cantidad de raices positivas
    //neg -> cantidad de raices negativas
    int pos = 0, neg = 0;
    //recorre ambos arreglos, para identificar la catidad de cambios de signo
    for (i = 0; i < n; i++) {
        //si la multiplicacion de los elementos resulta con un valor menor que 0
        //se asume que uno es positivo y otro negativo por lo tanto, hay cambio de signo
        if ((*(p + i)) * (*(p + i + 1)) < 0)
            pos++;
        //si la multiplicacion de los elementos resulta con un valor menor que 0
        //se asume que uno es positivo y otro negativo por lo tanto, hay cambio de signo
        if ((*(pmenos + i)) * (*(pmenos + i + 1)) < 0)
            neg++;
    }
    //imprime la cantidad de cambios de signo
    grafica(pos, neg, n);
}

void grafica(int p, int n, int grado) {
    //variables
    //comp1 -> cantidad de raices complejas para el primer caso
    //comp2 -> cantidad de raices complejas para el segundo caso
    //p2 -> positivos para el caso 2
    //n2 -> negativos para el caso 2
    int comp1 = 0, comp2 = 0, p2 = p, n2 = n;
    //si positivos es mayor a 2, disminuiye 2, para el segundo caso
```

```
    if (p > 2)
        p2 = p - 2;
    //si negativos es mayor a 2, disminuye 2, para el segundo caso
    if (n > 2)
        n2 = n - 2;
    //si positivos y negativos es menor al grado, entonces existen complejas
    if (p + n < grado)
        comp1 = grado - p - n;
    //si positivos y negativos es menor al grado, entonces existen complejas, para el segundo caso
    if (p2 + n2 < grado)
        comp2 = grado - p2 - n2;
    //imprime los casos
    printf("\nREGLA DE DESCARTES\n");
    printf("            caso I        caso II\n");
    printf(" positivas   %d            %d\n", p, p2);
    printf(" negativas   %d                    %d\n", n, n2);
    printf(" complejas   %d                    %d\n", comp1, comp2);
    printf(" total:      %d                %d\n", p + n + comp1, p2 + n2 + comp2);
}

void escriben(float *p, int n) {
    //variables
    //i -> servira para iterar el arreglo de coeficientes
    //m -> disminuir el grado de x con cada coeficiente
    int i, m;
    //inicializa a m con el valor de n
    m = n;
    for (i = 0; i <= n; i++) {
        printf("(%f)X**%d", *(p+i), m);
        //imprime el simbolo de +, siempre y cuando no sea el ultimo elemento
        if (i <= n - 1)
            printf(" + ");
        //disminuye el exponente de x en cada vuelta
        m--;
    }
}

void imprime_coeficientes(float *p, int n) {
    //variables
    //i -> servira para iterar el arreglo de coeficientes
    int i;
    for (i = 0; i <= n; i++) {
        printf("(%f)X**%d", *(p + i), n - i);
        //imprime el simbolo de +, siempre y cuando no sea el ultimo elemento
        if (i <= n - 1)
            printf(" + ");
    }
}

float *leen(float *polinomio, int grado_del_polinomio) {
    //define las variables a utilizar
    //i -> servira para iterar hasta llegar al valor de n
    int i;
    //solicitara que se ingresen los coeficientes
    for (i = 0; i <= grado_del_polinomio; i++) {
        printf("da el coeficiente de la variable de grado %d=>", grado_del_polinomio - i);
```

```
        polinomio = (float *) realloc((float *) polinomio, (i + 1) * sizeof (float));
        scanf("%f", &polinomio[i]);
    }
    return polinomio;
}

//FUNCIONES PARA SISTEMAS MATRICIALES

void determinante() {
            //a-> matriz de la cual se calculara el determinante
            //piv->pivote para realizar el metodo de gauss
            // j-> bandera que me indica si la matriz a tiene vectores linealmente dependientes
            //k-> me sirve para iterar

        int i, j, k, piv = 1;
        float **a=NULL;
        a = lee(a, 0,&i);
        // se manda a la matriz a para que se llene y guarda el numero de filas
        printf("El determinante a resolver es:\n");
        // se manda a imprimir la matriz a
        imprime(a, i, 0);
        //se manda a diagonalizar la matriz a mediante el metodo de gauss a una triangular superior
        j = gauss(a, i, 0, &piv);
        // si no existe dependencia lineal se calcula el determinante
        if (j == 0) {
            printf("la matriz diagonalizada es:\n");
            // se imprime la matriz triangular superior
            imprime(a, i, 0);
            //recordar que el determinante de una matriz triangular superior es el producto de la
diagonal
            float det = 1; //inicializa el det en uno para no alterar el producto e irlo acumulando
            for (k = 0; k < i; k++)
                det = det * (*(*(a + k) + k)); //producto de la diagonal
            printf("DETERMINANTE=%f\n", det * piv);  //imprime el determinante
        } else
            printf("DEPENDENCIA LINEAL");

}

void sistecuaciones() {
    int j, piv, num_ecuaciones = 0;
    float **matriz = NULL;
    matriz = lee(matriz, 1, &num_ecuaciones);
    printf("El sistema a resolver es:\n");
    imprime(matriz, num_ecuaciones, 1);
    j = gauss(matriz, num_ecuaciones, 1, &piv);

        if (j == 0) {
            printf("la matriz diagonalizada es:\n");
            imprime(matriz,num_ecuaciones , 1);
            printf("El resultado es:\n");
            resultado(matriz, num_ecuaciones, 0);
        }
                    else {
            printf("SISTEMA CON DEPENDENCIA LINEAL");
            getchar();
```

```
            getchar();
        }


}

void resultado1(float *a, int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("X[%d]=%15G\n", i, *(a+i));
    getchar();
    getchar();
}

void resultado(float **b, int n, int h) {
    int k, i;
    float x[n];

    x[n - 1]= *(*(b+(n-1))+n)/ (*(*(b+(n-1))+(n-1)));
    for (k = n-2; k >= 0; k--) {
            x[k]=*(*(b + k) + n);
        for (i = k + 1; i < n; i++)
            x[k]=x[k] - (*(*(b + k) + i))*x[i];
        x[k] = x[k] / *(*(b + k) + k);
    }
    if (h == 0)
        resultado1(x, n);
    else {
        resultado1(x, n);
        for (i = n; i > 0; i--)
            x[i] = x[i - 1];
        x[0] = 1;
        printf("        ECUACION CARACTERISTICA:\n");
        solucion(x, n);
    }
}

int gauss(float **a, int n, int s, int *p) {
    int k, i, j,c,f, puedo = 0;
    float m, M, temp;
    if (s==0){
            f=n;
            c=n;
    }
    else{
            f=n;
            c=n+1;
    }

    for (k = 0; k < f - 1; k++) {
        //busqueda del mayor abajo de la diagonal
        int piv = 0;
        M = fabs(*(*(a + k) + k));
        for (i = k; i < f - 1; i++) {
            temp = fabs(*(*(a + i + 1) + k));
            if (M < temp) {
```

```
                *p = -1 * *p;
                piv = i + 1;
                M = temp;
            }
        }
        //cambio de hilera
        if (M > fabs(*(*(a + k) + k))) {
            printf("ANTES DE PERMUTACION:\n");
            imprime(a, n, s);
            for (j = k; j < c; j++) {
                temp = *(*(a + k) + j);
                *(*(a + k) + j) = *(*(a + piv) + j);
                *(*(a + piv) + j) = temp;
            }
            printf("PERMUTACION:\n");
            imprime(a, n, s);
        }
        if (*(*(a + k) + k) == 0) {
            puedo = 1;
            break;
        } else {

            for (i = k + 1; i < f; i++) {
                m = -((*(*(a + i) + k)) / (*(*(a + k) + k)));

                for (j = k; j < c; j++) {
                    *(*(a + i) + j) = (*(*(a + i) + j)) + ((m) * (*(*(a + k) + j)));

                }



            }

        }
    }
    printf("DESPUES DE LA TRANSFORMACIÃ N\n");
    imprime(a, n, s);
    if (*(*(a + n - 1)+(n - 1)) == 0)
        puedo = 1;
    return (puedo);
}

float **lee(float **matriz, int k, int *num_ecuaciones) {
            int columnas, i, j, filas;
    printf("DA EL NUMERO DE HILERAS DE LA MATRIZ:");
    scanf("%d", &filas);
    columnas = filas+1;
    if (k == 0)
        columnas = filas;
    printf("da los elementos de la matriz A.\n");
    matriz = (float **) malloc(sizeof (float) * filas);
    for (k = 0; k < filas; k++)
        matriz[k] = (float *) malloc(sizeof (float) * columnas);
    for (i = 0; i < filas; i++) {
        for (j = 0; j < columnas; j++) {
```

```
            printf("Dame el valor a[%d][%d]=  ", i, j);
            scanf("%f", (*(matriz + i) + j));
        }
    }
    *num_ecuaciones = filas;
    return matriz;
}

void imprime(float **matriz, int m, int x) {
    int i, j, n = m+1;
    if (x == 0)
        n = m;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            float temp = *(*(matriz + i) + j);
            if (fabs(temp) < .000001 && j < i)
                temp = 0;
            printf("%13G", temp);
        }
        printf("\n");
    }
    getchar();
}
//ECUACIONES CARACTERISTICAS Y VALORES CARACTERISTICOS

void multiplica(float **a, float *b, int n) {
    float c[n];
    int i, j;
    for (i = 0; i < n; i++) {
        c[i] = 0;
        for (j = 0; j < n; j++)
            c[i] = *(*(a+i)+j) * (*(b+j)) + c[i];
    }
    for (i = 0; i < n; i++)
        *(b+i) = c[i];
}

void acomoda(float **a, float *b, int n, int k) {
    int i, m = k, s = 1;
    if (k == -1) {
        m = n;
        s = -s;
    }
    for (i = 0; i < n; i++)
        *(*(a+i)+m)=s * (*(b+i));

}

void caracteristico() {
        int num_ecuaciones,i, j;
        float **a=NULL,**c;
        printf("ECUACIONES Y VALORES CARACTERISTICOS.\n");
        a = lee(a, 0,&num_ecuaciones);
        float b[num_ecuaciones];
        printf("LA MATRIZ A TRABAJAR ES:\n");
        imprime(a, num_ecuaciones, 0);
```

```
        b[0] = 1;
        for (i = 1; i < num_ecuaciones; i++)
            b[i] = 0;
       c=(float **)malloc(sizeof(float)*num_ecuaciones);

              for (i = 0;i < num_ecuaciones; i++)
                    c[i]=(float*)malloc(sizeof(float)*num_ecuaciones);
              for(i = 0; i < num_ecuaciones; i++)
                   for (j = 0; j < num_ecuaciones; j++){
                                        *(*(c+i)+j)=0;

                    }


        acomoda(c, b, num_ecuaciones, (num_ecuaciones) - 1);
        for (i = (num_ecuaciones) - 2; i >= -1; i--) {
            printf("MultiplicaciÂ¢n No:%d\n", (num_ecuaciones) - i - 1);
            multiplica(a, b, num_ecuaciones);
            resultado1(b, num_ecuaciones);
            acomoda(c, b, num_ecuaciones, i);
            imprime(c, num_ecuaciones, 1);
        }
        int p;
        j = gauss(c, num_ecuaciones, 1, &p);
        if (j == 0)
            resultado(c, num_ecuaciones, 1);


}
```