JUAN MANUEL TORRES MORENO

Métodos numéricos con software en *C*

COLECCION
LIBRO DE TEXTO
1993

La razón es el método lento y tortuoso mediante el cual quien no conoce la verdad la descubre. El corazón tiene sus propias razones que la razón no comprende.

La raison est la méthode lente et tortueuse au moyen de laquelle quelqu'un qui ne connaît pas la vérité la découvre. Le cœur a ses raisones que la raison ne comprend pas.

Blaise PASCAL

AGRADECIMIENTOS

Este libro hubiera sido imposible sin las colaboraciones siguientes: en primer lugar a mi 80386 DX que permitió editar el texto durante meses y meses en noches incansables, así como a la LaserJet IIIp para el impreso final. A las máquinas 80286 y 80386SX del *Departamento de Sistemas*. A la impresora HP-LaserJet IIIsi donde pude imprimir diversas versiones. En fechas más recientes a la 80486 a 33 MHz, que (haciéndola trabajar de 10 am a 8 pm, sábado y domingo) me permitió entregar la versión final del documento. Igual reconocimiento a la *rapidísima* HP 80486 DX2 a 66MHz con la que he hecho una verdadera amistad. WordPerfect 5.1© se merece un especial agradecimiento porque todo el texto (5 Mbytes) fueron procesados con él. La ayuda inapreciable de ARJ versión 2.21© de *Robert K. Jung*, que me permitió compactar todo el material en 600 Kb para poderlo transportar en un solo disco de 3.5 plg.

A *Laura Caballero Pascual-Leone* por las pruebas y generación de varios ejemplos del software. A *Javier Ramírez Rodríguez*, *Ana Lilia Laureano* y *Diego Luna Gálvez* por la revisión cuidadosa del material. A mi mamá por el apoyo de tantos años.

INDICE DE MATERIAS

PARTE II. METODOS NUMERICOS

ANALISIS NUMERICO VS. MATEMATICAS SIMBOLICAS

LISTA DE PROGRAMAS

CAPITULO 1

CAPITULO 2

CAPITULO 3

CAPITULO 4

CAPITULO 5

CAPITULO 6

CAPITULO 7

CAPITULO 8

ANEXO B

ANEXO C

Todos los programas del texto se han probado con el compilador Borland ***C/C++***, bajo MS-DOS en una computadora personal 80486 DX2 a 66 Mhz, con 4Mb de RAM.

MARCAS REGISTRADAS

ARJ©	Robert K. Jung.
AT©	International Business Machines Co.
CATCH©	Dexxa International Inc.
MS-DOS©	Microsoft Co.
IBM©	International Business Machines Co.
IBM PC©	International Business Machines Co.
Paint Brush®	Microsoft Co.
Turbo C©	Borland International, Inc.
UNIX©	AT&T.
WordPerfect©	WordPerfect Co.

El manuscrito del texto fue transferido
en forma electrónica.

Texto, tipografía y ortografía producidos con **Word Perfect 5.1**
Imágenes rastreadas con un scanner **Scanman Plus** y producidas
con **Paint Brush (Windows)** y **Draw Perfect**.
Edición en **NOYS** (80386DX / 40 MHz), AT 286 y 386SX.

Impresión en una **HP Laser-Jet III**, en la **UAM-Azcapotzalco**
Departamento de Sistemas.
México, 1992.

1
PROGRAMACION EN *C*

ESQUELETO DE PROGRAMAS
TIPOS DE DATOS EN C
ENTRADA Y SALIDA
VARIABLES Y CONSTANTES
OPERADORES, CASTS Y TIPOGRAFIA
FUNCIONES
LOGICA BASICA Y CONTROL DE PROGRAMA
RECURSIVIDAD
APUNTADORES
ARCHIVOS EN C
GRAFICAS EN C

C es un lenguaje que deja al programador una gran libertad de acción dentro de los límites impuestos por una sintaxis concisa y a la vez sencilla. Esta libertad permite crear software poderoso, pero puede sin embargo ser fuente de errores inesperados para los programadores novatos (o acostumbrados a lenguajes demasiado paternalistas) porque ignoran la filosofía de ***C***, de *"el que manda es el programador"*.

En este primer capítulo se hace una recorrido del lenguaje para habituar al lector en la sintaxis del mismo y presentarle de manera clara y con ejemplos las partes que pudieran ocasionar ciertos problemas. Al terminarlo, usted tendrá una visión concreta de que cosas puede o no puede hacer con ***C***, y estará preparado a continuar después para programar algoritmos numéricos en forma clara y eficiente.

PREFACIO

Se han escrito muchos libros sobre análisis y métodos numéricos; pero existen menos libros sobre métodos numéricos usando software; y no he encontrado alguno que use al lenguaje ***C*** como base para desarrollarlos. Y lo que es peor todavía: la mayor parte de la literatura es *traducida* de textos en inglés, lo que ocasiona que no esté adecuada a nuestra realidad.

A veces el estudiante se confunde entre textos de *Análisis Numérico* y *Métodos Numéricos*. El análisis numérico inventa algoritmos para resolver los problemas que *no pueden* ser resueltos en forma analítica. También se encarga de calcular los errores asociados a los métodos. El análisis numérico se conoce desde hace mucho tiempo, pero hasta la aparición de las computadoras digitales estuvo relegado a pocos usos prácticos.

En realidad los métodos numéricos son la expresión formal del análisis numérico. Y estos métodos sólo tienen aplicación práctica cuando se acompañan de un software adecuado para implantarlos. El software es la expresión real -y última- de los algoritmos. Este es un libro de *métodos numéricos con software*. Es este punto el cual muchos textos dejan a un lado: la realización práctica de los métodos numéricos. Sin los resultados producidos por la computadora el mejor método queda empolvado. De nada sirve llenarle la cabeza de ideas matemáticas a un futuro ingeniero si éste no va a saberlas aplicar correctamente en la realidad.

Un detalle acerca de los textos que sí usan software para implantar métodos numéricos es el referente al lenguaje: normalmente no es el correcto por distintas razones. La mayor parte de los textos (y los programadores veteranos) emplean **FORTRAN**, que es un lenguaje bastante rápido en ejecución y con una extensa biblioteca de funciones numéricas (como la *International Mathematical and Statistical Library IMSL*, con alrededor de 240 subrutinas escritas precisamente en **FORTRAN**). No obstante, si se toma en cuenta el *aspecto formativo*, el hecho de que miles de líneas de código estén hechas en **FORTRAN** *no ayuda a programar mejor*: *programar sólo se logra programando* y no únicamente usando bibliotecas; además puede argumentarse que *IMSL* (*comercializada* por IMSL Inc. Houston, Texas) se encuentra en E.U., y no siempre está disponible -en el momento requerido- para universidades y estudiantes de otros países. Otras veces se han hecho implantaciones de métodos numéricos en **BASIC** o en **PASCAL**, pero no es fácil con ellos lograr una programación poderosa.

He escrito programas en varios lenguajes (**FORTRAN IV y 77**, **BASIC**, **PASCAL**, **PROLOG**, **FORTH** y **ensamblador**), pero los sistemas más importantes los he realizado en lenguaje ***C***, ya sea **Turbo *C/C++*** © en **MS-DOS**© o en **ANSI-*C*** bajo sistema operativo **UNIX**©Ä. Resulta que ***C*** cumple con todas las expectativas que se esperan de él como programador: *rapidez de compilación y ejecución, código eficiente, pequeño y reutilizable, portabilidad, modularidad, inclusión de archivos, compilación separada, gráficas, manejo de apuntadores y precisión*. Elementos todos para desarrollar proyectos en serio en el área de métodos numéricos. Mi decisión se basa también en la experiencia con ***C***: *programar se aprende programando* y definitivamente, el lenguaje ***C*** está construido para satisfacer múltiples necesidades informáticas de los desarrolladores, desde los novatos hasta los más exigentes, puesto que es un lenguaje construido por y para programadores.

• TOPICOS Y OBJETIVOS

El texto está basado en el lenguaje **ANSI *C*** en su mayor parte, y particularmente en el compilador de **Turbo *C***© en la cuestión gráfica, aplicado a métodos numéricos. Este compilador es extremadamente rápido y produce un código compacto y eficiente en microcomputadoras. Todos los programas que se incluyen han sido probados con el compilador **Borland *C/C++*** © versión 2.0 en una computadora 80486, PC compatible con 640 Kb de RAM en memoria base y tarjeta gráfica VGA, usando el sistema operativo **MS-DOS**© versión 5.

Además, como se ha programado bajo el estándar **ANSI**, es posible correr los programas en una computadora **UNIX**© que posea el compilador **ANSI-*C*** (usando el argumento **-A** en la orden **cc**). Los únicos problemas pueden presentarse en el programa **"Grafica.c"** que visualiza gráficas de funciones *f(x)* en $\mathbf{R^2}$, pero ello es debido a que -desafortunadamente- el estándar **ANSI** no llegó a tocar este punto; y en la forma de limpiar la pantalla en **Turbo *C***©, a través de la función **clrscr( )** que no forma parte de la biblioteca estándar pero que puede omitirse en **UNIX**©.

El material está dividido en dos partes: la *Parte I* presenta un capítulo dedicado al estudio de ***C*** y la *Parte II* se enfoca a aplicar métodos numéricos programándolos en ***C***. Debe mencionarse sin embargo, que el texto no es un libro para enseñar a programar: se asume que el lector está familiarizado con ciertos temas de computadoras como *bits, bytes, software, algoritmos, tipos de datos, direcciones de memoria* y conoce al menos un lenguaje de programación de alto nivel como **PASCAL**, **BASIC**, **MODULA, FORTRAN** o ***C***.

La *Primera parte* enseña a programar en ***C***, desde la óptica de los métodos numéricos; esta es la razón de que la mayoría de los ejemplos se enfoquen en esa dirección. El *Capítulo 1* presenta una introducción al lenguaje ***C*** (y a **Turbo *C***© en particular) dedicando el estudio a la lógica básica: ciclos, estructuras y control de flujo; funciones, apuntadores, paso de parámetros por valor y referencia y también

recursividad. Los archivos para entrada y salida de datos en disco son igualmente analizados. Por último se hace un estudio de las gráficas en **Turbo *C*©**.

La *Segunda parte* del texto presenta un panorama general introductorio de los métodos numéricos.

En el *Capítulo 1* se hace énfasis en la superioridad de éstos en relación a la solución analítica de diversos problemas. Se estudian los diferentes tipos de error y la manera de controlarlos.

El *Capítulo 3* muestra algunos métodos para hallar las soluciones (raíces) de ecuaciones no lineales del tipo $f(x) = 0$. También se toca el tema de las gráficas de funciones en el plano, que permiten estimar cualitativamente las raíces.

Los *Capítulos 4* y *5* tratan los temas de diferenciación e integración numérica respectivamente, empleando distintos métodos para ello. También se consideran las integrales múltiples y su evaluación por el método de Montecarlo.

El *Capítulo 6* considera el tema de los sistemas de ecuaciones lineales *(SEL)* a través de métodos directos e iterativos. Se incluye el excelente método del Ing. Francisco Montante de la *Universidad Autónoma de Nuevo León (UANL)*. En los métodos iterativos se expone el método de Jacobi, de Gauss-Seidel y el método de sobrerrelajaciones sucesivas *(SOR)* que son ampliamente usados en sistemas provenientes de ecuaciones diferenciales parciales.

El *Capítulo 7* trata la teoría de interpolación y aproximación polinomial: aquí se ven los temas de interpolación de Lagrange, se desarrolla mínimos cuadrados de orden *n*, usando el método de Montante y la aproximación exponencial y logarítmicas.

Finalmente el *Capítulo 8* trata a las ecuaciones diferenciales con condiciones iniciales. En esta parte se incluyen las secciones: método de Euler, métodos de Runge-Kutta, de Runge-Kutta-Fehldberg y métodos predictivos-correctivos como Milne, además de tratar el tema de ecuaciones de orden superior y sistemas de ecuaciones diferenciales.

Todos los capítulos incluyen programas completos que fueron probados y que el lector debería verificar en la computadora. También se incluye al final de cada uno, una serie de ejercicios selectos que ayudarán al lector a comprender mejor el material presentado. Algunos de ellos son teóricos y la mayoría le piden el desarrollo de software en computadora.

Al final del libro se incluyen cinco apéndices:

A.- Funciones, operadores y tipos del lenguaje.
B.- Tipos y definiciones estándar para el software de métodos numéricos.
C.- Este apéndice expone la biblioteca de métodos numéricos usada en el texto;
D.- Expone la complejidad algorítmica del método de Montante; y finalmente en
E.- Se deduce la ecuación de Laplace en diferencias finitas.

El material del texto es suficiente para cubrirse en cursos trimestrales omitiendo ciertos tópicos (como integrales triples, aplicaciones como distribución de temperaturas o sistemas de ecuaciones diferenciales) o en cursos semestrales de licenciatura para estudiantes de ingeniería o ciencias, que se supone han seguido cursos de matemáticas como álgebra lineal, cálculo y ecuaciones diferenciales; y además han llevado uno o más cursos de computación y diseño de algoritmos.

Se ha recopilado material de las notas del autor a lo largo de diversos cursos de materias como **COMPUTACION I**, **COMPUTACION II**, **PROGRAMACION AVANZADA**, **TEMAS SELECTOS DE SISTEMAS** y **DE ELECTRONICA** impartidos en la **Universidad Autónoma Metropolitana-Azcapotzalco** (**UAM-A**) durante varios trimestres. Se incluyen más de 60 programas completamente probados que pueden servir como base para desarrollar una biblioteca completa de métodos numéricos, además de dejar como ejercicios otros algoritmos.

Es necesario indicar que al implantar un algoritmo en un programa de computadora a veces se tiene la disyuntiva entre la *claridad del algoritmo o la eficiencia del método*. Pienso que la claridad es importante para entender el principio de funcionamiento, pero la *eficiencia* debe ser el criterio último al momento de implantar los algoritmos numéricos en una computadora. Por lo mismo he preferido mantenerla a costa de un pequeño precio en la claridad, porque ésta debe buscarse en los textos de Análisis Numérico.

PARTE I

HERRAMIENTAS DE PROGRAMACION: El lenguaje *C*

UN LARGO RECORRIDO: DESDE ALGOL 60 A C++
TURBO C
***C*: LENGUAJE DE NIVEL INTERMEDIO**

Los lenguajes de programación constituyen un tema fascinante, en el cual la comunicación hombre-máquina a sufrido cambios radicales. La evolución de los lenguajes informáticos refleja la capacidad del hombre para interactuar de manera inteligente con sus propias creaciones.

Ya hace algún tiempo desde el alambrado directo y el ensamblador. **FORTRAN** se ha ganado un lugar respetable en la historia de la programación al cabo de décadas de uso, al igual que **ALGOL**, **LISP** y más recientemente **PASCAL** y **MODULA**. Ahora, corresponde a la generación de ***C*** y sus sucesores realizar un papel digno en esta importante área del quehacer humano.

• UN LARGO RECORRIDO: DESDE ALGOL 60 A *C*++

Los orígenes de ***C*** son antiguos: se remontan tiempo atrás desde el histórico **Algol 60**. El árbol de lenguajes derivados de él (**CPL**, **Algol 68**, **PASCAL**[1] y **SIMULA 67**) son una muestra de la potencia de este lenguaje original (a su vez derivado del **FORTRAN**). **CPL** ideado por Strachey alrededor de 1966 (siglas de *Combined Programming Language*) fue un lenguaje usado en el laboratorio con propósitos de estudio sobre una teoría coherente y formal de lenguajes de programación y nunca se implantó realmente.

M. Richards en 1969 creó **BCPL** (siglas de *Basic CPL*) como una herramienta para el desarrollo de compiladores y programación de sistemas, que aún sigue de uso en Europa. **BCPL** influyó en otro lenguaje llamado **B** (eliminando las siglas **CPL** por simplicidad) que fue inventado por Thompson[2] y éste a su vez impactó para llegar al **lenguaje *C***.

Dennis Ritchie inventó el primer compilador de ***C*** en una computadora **DEC PDP-11** con el sistema operativo **UNIX©** (que en ese entonces había sido recientemente creado por Ken Tomphson) en 1972, con el propósito de constituir un lenguaje de implantación para programas asociados a este sistema operativo.

C se hizo famoso cuando con él pudo reescribirse en 1973 el software del sistema operativo **UNIX** casi en su totalidad, logrando de esta manera hacerlo *portable* entre distintas plataformas de hardware. La pequeña parte de **UNIX** que se siguió construyendo en ensamblador fue la correspondiente a las partes de más bajo nivel del núcleo (o *kernel* del sistema), pero ello no representó más del 5% del total de líneas de código de aquel entonces; rompiendo así con el viejo mito de que *"los sistemas operativos deben ser construidos totalmente en ensamblador"*.

Existió desde 1971 hasta 1983 el lenguaje ***C*** definido por Brian Kernighan y Ritchie (denominado por simplicidad ***C K&R***) que se consideró un estándar semioficial del mismo, hasta que en el verano de 1983 el comité **ANSI**[3] de estándares se reunió y comenzó a trabajar para unificar criterios y diversas versiones y dialectos de ***C*** que ya pululaban por todas partes. Después de un largo período de evaluaciones, el **ANSI-*C*** se estableció a finales de 1988.

---

[1] ***Blaise Pascal***. 1623-1662. Matemático y filósofo nacido en Clermont-Ferrand, Francia.Inventó una de las primeras máquinas de cálculo: la Pascalina. Contribuyó a las teorías previas del cálculo y junto con Pierre Fermat tiene el crédito de la teoría de la probabilidad. El lenguaje PASCAL fue creado por el suizo ***Niklaus Wirth*** y le bautizó con ese nombre en honor al matemático Blaise.

[2] ***Ken Thompson***, ***Brian Kernighan*** y ***Dennis Ritchie***. Famosos informáticos de AT&T Bell Laboratories en Murray Hill, New Jersey que desarrollaron el lenguaje ***C*** y el sistema operativo UNIX.

[3] **ANSI** son las siglas del *American National Standards Institute*.

En 1986, el sueco Bjarne Stroustrup desarrolló ***C++*** (en un principio algunos programadores decían que debió haberse llamado ***D***, este nuevo lenguaje) derivado de **Simula 67** y ***C*** y como una extensión de éste último, en sus propias palabras para proporcionar: *"... recursos flexibles y eficientes para definir tipos nuevos"*; lo que originó después el lenguaje orientado a objetos en ***C***.

En la figura I.1 puede observarse una evolución gráfica de los lenguajes de programación correspondientes a la rama de **ALGOL**.

Una disertación amena e interesante del tema de la evolución y los paradigmas de los lenguajes de programación se encuentra por ejemplo en *Seti, [1992]*; donde se trata -entre otras cosas- de los derroteros seguidos por las diferentes ramas del árbol y sus alcances, como la que hicieron los lenguajes funcionales derivados de **LISP**[4] o de la programación lógica con **PROLOG**[5] en inteligencia artificial, escrito por una autoridad en la materia.

• TURBO C

La compañía Borland© lanzó la versión 2.0 del compilador *Turbo C*© [6], con el propósito de incorporar todas las funciones del estándar **ANSI-*C*** y otras más (en particular gráficos, sonido y manejo de los recursos del DOS y de la PC) y ofrecer gran velocidad en microcomputadoras, tanto en tiempo de compilación como en tiempo de ejecución.

El resultado de este trabajo fue uno de los compiladores más rápidos, eficientes y populares de lenguaje ***C***. Posteriormente Borland introdujo la versión orientada a objetos con ***C++*** en sus versiones **1.0, 2.0** y **3.0**

Sin embargo el compilador **Borland *C/C++*** tiene completa compatibilidad con **Turbo *C* 2.0** y es posible compilar y ejecutar cualquier programa escrito en lenguaje ***C*** en el compilador ***C++*** si se le añade al nombre del archivo la extensión **".C"** (en lugar de **".CPP"**, que viene precisamente de las siglas de ***C Plus Plus***).

Este fue el compilador utilizado para probar todos los programas ejemplo del texto; pero puede emplearse para este mismo propósito, **Turbo *C* 2.0**.

---

[4] ***LISP*** Lenguaje de procesamiento de listas, inventado por el norteamericano ***John McCarty*** para programación en inteligencia artificial.

[5] **PROLOG** fue desarrollado en 1970 por ***Alain Colmeraur*** en **l'Université d'Aix-Marseille II**, France.

[6] **Turbo *C*** fue creado por el matemático ***Robert Jarvis***.

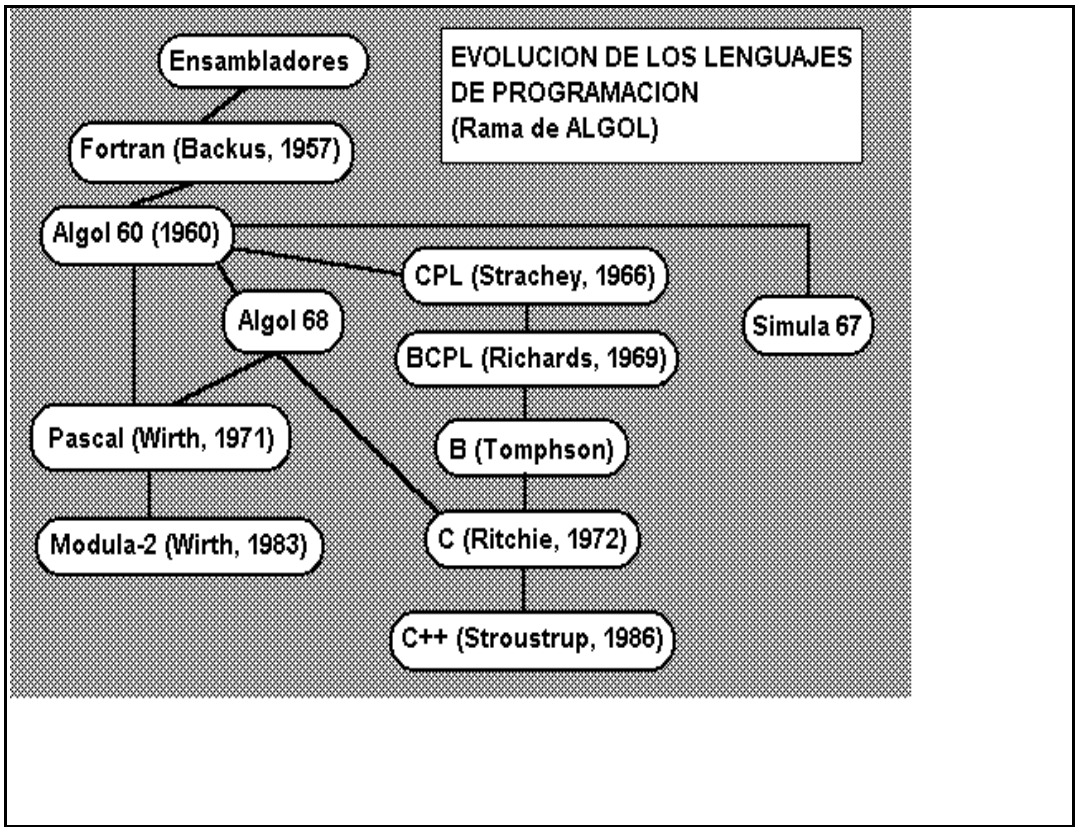


Fig. I.1 ***Lenguajes de la rama de ALGOL***

• *C*: LENGUAJE DE NIVEL INTERMEDIO

El nivel computacional de un lenguaje de programación suele ser una manera de clasificación de los mismos. Es tradicional dividir los lenguajes en alto, medio y bajo nivel. Al decir que ***C*** es un lenguaje de nivel medio, se quiere dar a entender que permite manejar y programar la computadora en forma directa y tan eficientemente como el lenguaje **ensamblador**, además de que fue pensado para realizar programación estructurada.

C posee tipos de datos en forma parecida a **PASCAL** y permite además manipular bits, bytes, números y direcciones de memoria, aunque carece de la capacidad de manipular *"objetos compuestos, tales como cadenas de caracteres, conjuntos, listas o arreglos" [KER90]*, ni tampoco la entrada y salida

en forma implícita; aunque todo lo anterior puede ser manipulado a través de funciones explícitas. Gracias a ello los compiladores de ***C*** son pequeños porque el lenguaje es pequeño, lo que permite profundizar mucho en su estudio.

Los programas escritos en ***C*** son estructurados a nivel de funciones y éstas permiten la *recursividad*. En síntesis, ***C*** es un lenguaje de propósito general, tiene una sintaxis concisa y proporciona un acceso eficiente a la máquina.

Por todo lo anterior, no debe tomarse la ubicación de nivel medio en un sentido peyorativo, por el contrario el lector puede pensar que ***C*** posee las ventajas del ensamblador: velocidad, código pequeño; y al mismo tiempo las ventajas de los lenguajes de alto nivel como PASCAL o MODULA, es decir tipos de datos, estructuras de control de flujo y modularización por funciones; o al igual que FORTRAN permite la compilación separada.

En la tabla I.1 se muestra la ubicación de ***C*** en relación a otros lenguajes de acuerdo a su nivel computacional

Tabla I.1 Ubicación del lenguaje *C*

Nivel computacional	Lenguajes
Bajo nivel	**Ensamblador** **Macroensamblador**
Nivel medio	**Forth** ***C*** ***C*++**
Alto nivel	**Basic** **Fortran** **Cobol** **Pascal** **Modula-2** **Ada**

Los compiladores de ***C*** se han utilizado ampliamente en la programación de sistemas y en proyectos importantes de software durante los últimos 17 años. Esta programación incluye aspectos importantes como:

a) *Construcción de sistemas operativos*, por ejemplo: **MS-DOS**©, **UNIX**©, **XENIX**©, y otros.
b) *Construcción de intérpretes y compiladores de lenguajes*, por ejemplo: **FORTRAN, BASIC, PASCAL**, **PROLOG**, (el mismo ***C***) y otros.

c) *Desarrollo de hojas de cálculo electrónicas y bases de datos*, como **ORACLE**©.
d) *Elaboración de editores y procesadores de textos*.

Sin embargo, puede utilizarse ***C*** virtualmente en cualquier aplicación, incluyendo -por supuesto- métodos numéricos.

Por último cabe mencionar que ***C*** es un lenguaje portable; esto es: un programa escrito en una microcomputadora personal con sistema operativo **MS-DOS** puede ser fácilmente trasladado a una minicomputadora que opere bajo **UNIX**, o a un *mainframe* (IBM©, DIGITAL©, HP©, etc.) con sistema operativo propio. Esta portabilidad se debe -en parte- a la estandarización **ANSI** y representa una herramienta poderosa para adaptar o actualizar un programa dentro de un entorno distinto del que lo vio nacer. No se prevé un cambio radical de esta situación: la tendencia seguirá probablemente con la construcción de software en ***C*** (o en ***C*++** orientado a objetos, de Bjarne Stroustup) para el futuro próximo.

§1.1 ESQUELETO DE PROGRAMAS

En ***C*** a diferencia de **PASCAL**, **FORTRAN** o **BASIC**, existe una *distinción* importante entre escribir con letras mayúsculas o minúsculas, porque el compilador las trata de diferente manera: por conveniencia las constantes simbólicas se deben escribir con mayúsculas y el resto del código (variables y palabras reservadas) en minúsculas. De esta manera, los programas resultan ser más legibles para una posterior modificación.

Es recomendable también que los programas sigan una estructura denominada **esqueleto estándar**, lo que contribuye a un mejor mantenimiento cuando tenga lugar. Los prototipos de las funciones a ser usadas deben escribirse antes del programa principal, y las funciones inmediatamente abajo de éste. Esta manera de escribir obliga al programador a ser más disciplinado. Un esqueleto estándar de código en ***C*** se presenta inmediatamente:

Listado 1.0. Programa.c

```
#include <stdio.h>          /* sección de              */
#include <stdlib.h>         /* archivos incluidos      */
#include <math.h>
#include "archivo..."

#define ...                 /* constantes simbólicas   */
#define ...                 /* y macroprocesador       */

funcion_1( );               /* prototipos ANSI         */
funcion_2( );
    ...
funcion_n( );
lista de variables y constantes globales;
main( )
{
     /* variables y procesos */;
}

funcion_1( )
{
     /* variables y procesos */;
}

funcion_2( )
{
     /* variables y procesos */;
}
     ...
funcion_n( )
{
     /* variables y procesos */;
```

```
}
```

Los comentarios del programa deben iniciar con diagonal-asterisco **/*** y terminar con asterisco-diagonal ***/**, pueden ser situados en cualquier parte del código e incluso abarcar varias líneas. Normalmente el estándar **ANSI** no permite los comentarios anidados, pero el compilador Borland ***C/C++***© sí lo permite a través de una opción del entorno.

Se pueden incluir archivos separados en la compilación a través de la directiva del compilador **`#include <archivo>`** o **`#include "archivo"`**. Los archivos pueden ser códigos fuente escritos por el usuario o archivos de encabezado (*archivos.h*)[7] de funciones estándar que vienen como parte del compilador. La compilación condicional deja la libertad de elegir dichos archivos al momento de compilar dependiendo de una bandera, ver por ejemplo el **Anexo A**.

En el *macroprocesador* se definen constantes simbólicas globales o macro funciones como será explicado más adelante. El estándar **ANSI** indica que debe escribirse los *prototipos* de las funciones a utilizar antes de que sean invocadas. Todos los programas deben empezar con una función especial llamada **`main( )`** y esta función debe ser única. Además, un programa estará construido por bloques separados denominados ***funciones***.

De acuerdo a la estructura anterior, el programa más simple que puede *compilarse* y *ejecutarse* en cualquier compilador de ***C*** es el siguiente:

```
main( )
   {
   }
```

C tiene un número relativamente corto de palabras reservadas en relación a otros lenguajes de programación, lo que implica tener una sintaxis concisa. Ver el **Anexo A** para las palabras reservadas del lenguaje.

• Directiva al compilador `#define`

`#define` se usa para definir un *identificador* y una *cadena de símbolos*. El compilador sustituirá esta cadena de símbolos por el identificador cada vez que encuentre dichas ocurrencias en el archivo fuente.

El estándar **ANSI** identifica a **`#define`** con el nombre de *macro*. La directiva **`#define`** se conoce como *macroprocesador*. No debe existir ningún punto y coma (**;**) después de la macrodefinición. Observe los ejemplos que están a continuación.

Ejemplo 1.1

[7] Los archivos de encabezado son conocidos en inglés como *archivos header*, denominación comúnmente usada en la programación en *C*.

```
#define VERDADERO 1
#define FALSO 0

      . . .

printf("%d %d",VERDADERO,FALSO);

      . . .

#define ERROR  "Error en la lectura\n"
printf(ERROR);
#define AVOGADRO 6.022e23
```

Las macros realizadas a través de **#define** pueden tener argumentos, lo que genera una *macro función*. Esto es ejemplificado con las funciones **MAX( )** y **MIN( )** que sirven para encontrar el valor máximo y mínimo respectivamente de dos entidades que pueden ser números o expresiones. Una ventaja evidente de las macro funciones es que pueden hacerse independientes del tipo de los argumentos enviados; es decir, **MIN( )** y **MAX( )** funcionan igual con argumentos enteros, reales, de doble precisión o incluso caracteres. Otra ventaja importante es que **#define** aumenta la velocidad de ejecución del programa, pero ésto se tiene que pagar con un incremento obvio en el tamaño del código fuente, puesto que el macroprocesador realizará una sustitución de la definición en todos los lugares del programa donde se utilice la macro función.

Note que además se requiere el uso de paréntesis **( )** en la definición y manipulación de los argumentos de una macro lo que logra evitar ambigüedades de interpretación por parte del preprocesador, cuando dichos argumentos involucran expresiones complejas y no únicamente variables simples. En seguida se reproducen las dos macros relatadas anteriormente:

Ejemplo 1.2 Macros MIN( ) y MAX( )

```
#include <stdio.h>
#define MIN(a,b)  ((a)<(b))?(a):(b)
#define MAX(a,b)  ((a)>(b))?(a):(b)

...

void main(void)
{
      int x, y;
      int w;
      x = 5;
      y = 4;
      printf("el mínimo es: %d",MIN(x,y));
      w = MAX(x*10, MIN(x-5,y+7);
      printf("El máximo es %d\n",w);
}
```

§1.2 TIPOS DE DATOS EN C

Existen cinco tipos de datos básicos en lenguaje ***C***, denotados por: **`char, int, float, double`** y **`void`**; que permiten manipular caracteres alfanuméricos, números enteros, reales, reales en doble precisión y tipos nulos respectivamente. Cada tipo tiene un rango asociado que depende del número de bits de su representación (Vea por ejemplo el **Anexo A**). La palabra de máquina de una computadora personal 8086 es de 16 bits, y -por ejemplo- el tipo de dato **`double`** requiere cuatro palabras en memoria para poder ser almacenado.

• Modificadores de tipo

Una lista de modificadores permite cambiar el rango de los tipos, excepto para **`void`**. Estos modificadores se enlistan a continuación:

Tabla 1.1 Modificadores de tipos

Modificador	Significado
`signed`	con signo
`unsigned`	sin signo
`long`	Grande
`short`	pequeño

Los modificadores se usan para generar nuevos rangos en los tipos básicos como puede verse en la tabla **1.2**:

Tabla 1.2 Rangos modificados

Tipo modificado	Nuevo rango
`unsigned char`	**`0 a 255`**
`unsigned int`	**`0 a 65535`**
`long int`	**`-2147483648 a 2147483647`**
`unsigned long int`	**`0 a 4294967295`**
`long double`	**`3.4E-4932 a 1.1E+4932`**

§1.3 ENTRADA Y SALIDA

Existen dos funciones básicas para el manejo de entrada y salida: **`printf( )`** y **`scanf( )`**.

En particular, la función **printf( )** escribe información en la pantalla y **scanf( )** lee datos desde el teclado. No forman parte del lenguaje en sí, sino que constituyen parte de la biblioteca de **<stdio.h>**.

Sintaxis

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

La **"cadena de control"** contiene la *información a desplegar* o el *código de formato* con el que deben leerse o imprimirse los argumentos. Por ejemplo, **%d** se asocia a enteros, **%f** y **%e** a reales tanto en **printf( )** como en **scanf( )**; El formato **%lf** se asocia a datos en doble precisión únicamente en **scanf( )**. Es necesario anteponer un símbolo **&** (*ampersand*) a los argumentos que representan variables a leer con **scanf( )**, y en **§1.6** se explicará por qué. Otros códigos de formato para manipular entrada y salida se muestran más ampliamente en el **Anexo A.**

§1.4 VARIABLES Y CONSTANTES

• Variables

Las variables son localidades de memoria que almacenan valores que pueden cambiar durante la ejecución. El nombre de las variables debe empezar con un carácter alfabético o con el carácter de subrayado '_'. Se permite el uso del subrayado en cualquier lugar del nombre, pero sólo los primeros 32 caracteres son significativos para diferenciar una variable de otra. Lo anterior significa que:

```
contador, ContadoR, cONTADOR, C_O_N_T_A_D_O_R  y __CONTADOR__
```

son consideradas como cinco variables válidas, pero distintas. Todas las variables en un programa en ***C*** deben ser declaradas siempre antes de usarse. Esta característica es deseable en un lenguaje porque ayuda a tener disciplina y a no inventar variables innecesarias simplemente porque "son gratis". La declaración puede ser realizada dentro de bloques de código; un bloque es un grupo de líneas encerradas entre llaves. De esta manera, una variable "vive" solamente dentro del bloque correspondiente y se destruye al salir de él.

Sintaxis

```
tipo lista de variables;
```

La *lista de variables* indica los nombres de las variables que serán declaradas del mismo **tipo**, separadas entre sí por comas.

Ejemplo 1.3

```
int i, j, k;
unsigned int x;
double epsi, tolerancia;

{     /* Bloque de código */
      long int iteraciones;
      int i, j, k;
      . . .
}
```

Las variables se denominan *automáticas* o locales en caso de ser declaradas dentro de una función o un bloque de código entre llaves, *nunca* están inicializadas por omisión, lo que significa que pueden contener *basura*. Por otra parte, las variables *globales* siempre se inicializan a cero o a un string nulo y serán conocidas a lo largo de todo el código de un mismo programa. Observe las declaraciones en el ejemplo siguiente y trate de responder ¿qué imprimen los **`printf( )`**?:

Ejemplo 1.4

```
#include <stdio.h>

int i = 1, j = 2, k = 3;    /* Globales */

void main( )
{
  int i = 4,
      j = 5,
      k = 6;
  {                    /* Bloque */
    int i = 7, j = 8, k = 9;
    printf("Locales al bloque: %d %d %d\n",i, j, k);
  }
  printf("Locales a main( ): %d %d %d\n",i, j, k);
}
```

• Constantes

Las constantes en ***C*** pueden ser de cualquier tipo básico. Las constantes de caracteres deben estar encerradas entre apóstrofes.

Sintaxis

```
const constante = valor;
```

El valor original de las constantes no puede ser alterado durante la ejecución normal de un programa.

Ejemplo 1.5

Tipo de dato	Constante
char	`const alfa = 'a',` `beta = '\n';`
int	`const rango = 1,` `valor = 123;`
double	`const n = 12345678.0` `const epsi = 1e-70;`

Una cadena es un conjunto de caracteres encerrados entre comillas. Las cadenas se conocen comúnmente con el nombre de *string*. Por otra parte, un único carácter constante se encierra entre apóstrofes: **`"a"`** es una cadena con la letra **`a`**, y **`'a'`** es un carácter simple. Estos dos ejemplos son conceptualmente distintos y en **§1.8** se desarrolla el tema de las cadenas.

• Constantes de barra invertida

Existen algunos caracteres no imprimibles como el carácter campana (**bell**), y para manipularlos se usan las constantes de barra invertida como **`'\a'`** (bell), **`'\b'`** (backspace), **`'\n'`** (retorno), **`'\t'`** (tabulador) y otras mostradas en el **Anexo A**. La barra invertida debe denotarse por **`'\\'`**. Esta barra invertida será particularmente útil en la búsqueda de archivos a través del árbol de directorios del disco (lo que se conoce como *path*).

§1.5 OPERADORES, CASTS Y TAQUIGRAFIA

El lenguaje ***C*** posee diversos operadores aritméticos, relacionales, lógicos y operadores a nivel de bits. Los operadores se evalúan de acuerdo a ciertas reglas de precedencia (es decir, de acuerdo a un orden de evaluación de una expresión matemática) ya establecidas que son mostradas en la tabla **1.4.** Cuando la precedencia de dos operadores es la misma, la expresión dada se evalúa siempre *de izquierda a derecha*.

• Operador de asignación

El operador de asignación corresponde al signo **`=`** de igualdad. Este operador puede ser usado dentro de sentencias condicionales tipo **`if( )`** (que se verán más adelante) o en sentencias de impresión a través de **`printf( )`**. Además este es un operador secuencial, por lo que es válida la notación matemática convencional:

`a = b = c = d = ... =` *`expresión`*`;`

• Operadores aritméticos

Los operadores aritméticos del lenguaje ***C*** permiten realizar las cinco operaciones básicas siguientes:

- suma,
- resta,
- multiplicación,
- división y
- módulo[8].

Además existen un par de operadores especiales para incrementos y decrementos representados por **++** y **--**, normalmente no encontrados en otros lenguajes de programación.

Tabla 1.3 Operadores aritméticos

Operador	Operación
+	Suma
-	Resta y menos unario
*	Multiplicación
/	División
%	Módulo de la división
++	Incremento
--	Decremento

Notas:

1. El operador módulo **%** no se permite en los tipos **float** o **double**.
2. El operador división **/** sobre algún operando **int** produce siempre un resultado de tipo **int**.

La precedencia de los operadores aritméticos se muestra en la tabla 1.4 reproducida a continuación:

Tabla 1.4 Precedencia de operadores

Precedencia	Operador
Máxima	+, - unarios,

[8] El módulo aritmético representa lo siguiente:
$x\overline{)yz}$ *division* *w=modulo*

	( )
Intermedia	*, /, %
Mínima	+, - suma y resta

• Operadores de incremento y decremento

El operador `++` añade `1` a su operando y se conoce como **incremento**; por otra parte, el operador `--` resta `1` a su operando y representa un **decremento**. De tal manera que la expresión:

```
x++; y--;
```

es equivalente a:

```
x = x + 1;  y = y + 1;
```

Ambos operadores pueden preceder o seguir al operando. Por ejemplo considere:

```
x = x + 1;
```

es equivalente a: `x++;` o a: `++x;`

pero esto es diferente cuando se emplea una expresión. Considere ahora:

```
x = 10; y = ++x;
```

en este caso se pone: `y = 11` porque se pre-incrementa la `x`, y luego se asigna a `y`. En cambio en:

```
x = 10; y = x++;
```

`y` toma el valor `10`, después se post-incrementa la `x`. En ambos casos la `x` es puesta a `11` pero la diferencia con la `y` depende de un pre o post-incremento en la `x`. En todo caso el programador debe saber elegir cuál de los operadores (post o pre operador) debe usar de acuerdo a sus necesidades, considerando además los criterios anteriores.

• Operadores lógicos y relacionales

Al trabajar en ***C***, cualquier expresión distinta de cero equivale a *verdadero*, mientras que *falso* equivale a un cero. Este hecho junto con las decisiones condicionales y los operadores relacionales y lógicos, son frecuentemente usados en los programas escritos de manera profesional.

Tabla 1.5 Operadores relacionales

Operador	Acción
>	Mayor que

>=	**Mayor o igual**
<	**Menor que**
<=	**Menor o igual**
==	**Igual**
! =	**Diferente**

No debe existir espacios en blanco entre los operadores con dos caracteres.

Tabla 1.6 Operadores lógicos

Operador	Acción
&&	AND
||	OR
!	NOT

Los operadores lógicos y relacionales tienen menor precedencia que los aritméticos. Por ejemplo la expresión:

```
15 < 7 + 21;
```

se evalúa como si realmente estuviera codificada de la manera siguiente:

```
15 < (7 + 21);
```

que produce por supuesto un resultado verdadero. Más aún, todas las operaciones lógicas y relacionales producen necesariamente `0` ó `1`.

• Operadores sobre bits

Los operadores sobre bits actúan como lo hace el lenguaje ensamblador a bajo nivel: es decir, tratan a las variables con su representación en binario para poder manipular bits en forma individual.

Corrimientos de bits.

Permiten correr **n** bits del número **x** a la izquierda o a la derecha del mismo. Lo que se conoce como *shift*.

```
x >> n;    /* shift a la derecha */
x << n;    /* shift a la izquierda */
```

Operadores lógicos de bits.

Con dichos operadores se pueden realizar las operaciones tradicionales lógicas de **and**, **or**, **xor** y `not` pero con bits individuales.

Tabla 1.7 Operadores lógicos a nivel de bits

Operador	Acción
&	AND
|	OR
^	OR EXLUSIVO

!	NOT

Al usar los operadores sobre bits, tanto **x** como **n** deben ser variables o constantes del tipo **char** o **int** necesariamente.

• El operador , (coma)

Este operador se usa para encadenar varias expresiones que dependen unas de otras; por ejemplo en control de ciclos (que se estudiarán posteriormente) o en expresiones tales como:

```
x = ( y = 3, z = ++y, z + 1);
```

donde se asigna **3** a la variable **y**, luego se pre-incrementa la variable **y**, asignándose a **z**, por lo que **z=4**; y finalmente se asigna **5** a **x**.

• El operador ?: (interrogación-dos puntos)

Es un interesante operador ternario (porque requiere de tres operandos) usado en sentencias condicionales.

Sintaxis

var = *expresión_1***?** *expresión_2***:** *expresión_3***;**

Se evalúa *expresión_1*, si es cierta, **var** toma el valor de *expresión_2*; si no, toma el valor de la *expresión_3*. **?:** es un operador ternario y es usado para reemplazar sentencias del tipo:

SI(condición)
ENTONCES expresion1;
OTRO CASO expresion2;

Ejemplo 1.6

```
x = 10;
. . .
y = x>9? 100: 200;
```

En este ejemplo, la variable **y** toma el valor 200 al resultar verdadera la *condición* **x > 9.**

• `(casts)` El operador conversión forzada de tipos

Se puede *forzar* a una expresión a pertenecer a un tipo específico de dato usando una conversión forzada a través del operador **(cast)**.

Sintaxis

```
(tipo) expresión;
```

Aquí, *tipo* corresponde a cualquier tipo estándar (modificado o creado por el usuario) al cual será convertida la expresión.

Ejemplo 1.7

```
int   x = 3,
      y = 2;

      . . .
printf("%10.5f\n",(float) x/y);
      . . .
```

El **(cast)** provoca que el valor de la variable `x` sea convertido al número real `3.0`, se divida entre la variable `y` convertida a **2.0** y se obtenga el resultado real **1.50**.

Debe ser cuidadoso el uso de los paréntesis al manejar **(cast)**, ya que sustituyendo la sentencia **printf( )** anterior por:

```
printf("%3.2f\n",(float) (x/y));
```

que parecer ser lo mismo, pero en cambio ahora se efectúa una división entera entre las dos variables, luego este resultado se convierte a real y finalmente se imprime el número `1.00` que es totalmente diferente a como se deseaba.

• Taquigrafía en C

La taquigrafía del lenguaje ***C*** simplifica ciertas sentencias de asignación. Es muy recomendable su uso pues reduce el tiempo de ejecución y proporciona una visión más consistente del código. Los programas profesionales en ***C*** siempre hacen uso de esta taquigrafía. En la tabla **1.8** se muestra la conversión de expresiones algebraicas convencionales en su correspondiente expresión taquigráfica.

Tabla 1.8 Taquigrafía en C

Expresión convencional	Expresión taquigráfica
`x=x+a`	`x+=a`
`x=x-b`	`x-=b`
`x=x*c`	`x*=c`
`x=x/d`	`x/=d`
`x=x%e`	`x%=e`
`x=x>>f`	`x>>=f`
`x=x<<g`	`x<<=g`

- **`typedef`**

Con **`typedef`** se permite definir nuevos nombres de tipos de datos de manera explícita. No se crea un nuevo tipo de dato, sino solamente un nuevo *nombre de definición*, lo que puede llegar a ser muy útil, entre otras cosas por razones nemotécnicas.

Sintaxis

```
typedef tipo definición;
```

Ejemplo 1.8

```
typedef double real;
real  x = M_PI,
      y = -15.98788, z;
```

De esta manera se define el tipo **`real`** de una manera más intuitiva para el programador que puede ser cambiada por el tipo **`float`** cuando sea necesario. En el software básico de métodos numéricos del texto se declaran de esta forma los tipos de datos. Ver el **Anexo B**.

- **El especificador `register`**

Modificador interesante de almacenamiento aplicable sólo en variables locales de caracteres y enteros. Provoca que el compilador trate de mantener una variable en registros de la CPU en lugar de la memoria, donde residen normalmente las variables; ello agiliza enormemente el desempeño de cualquier programa. En situaciones normales la velocidad de acceso de las variables register se ve incrementada en forma dramática hasta en un 50%.

Sintaxis

`register` *tipo variable*;

`tipo` solamente puede ser **`char`** o **`int`** y **`register`** es únicamente aplicable en variables locales.

Ejemplo 1.9

```
void main(void)
     {

          register int x = 0;
          register char op;
          int y=0;
          . . .
          x++; y++;
          . . .
     }
```

La manipulación de las variables **`x`** y **`op`** tendrá un acceso más rápido que con la variable **`y`**, en el ejemplo anterior.

Las operaciones realizadas con variables tipo **`register`** resultan ser más rápidas que las operaciones hechas con variables normales. Esta especificación se debe emplear en contadores de ciclos o índices de arreglos (ya sean vectores o matrices) para acelerar su desempeño; sin embargo debe considerar que declarar variables **`register`** resulta ser más bien una *sugerencia* al compilador en vez de una orden: finalmente éste determinará cuándo usar variables tipo **`register`** y cuándo no, ignorándose las declaraciones excesivas.

• Tipos enumerados

Pueden enumerarse los tipos de dato **`integer`** o **`char`** con la sentencia **`enum`**, que permite definir nombres simbólicos a las constantes y -posiblemente- inicializarlas en un valor predeterminado. El uso de la sentencia **`enum`** es recomendable en aplicaciones que involucren menús porque se puede hacer referencia a las opciones del mismo como si fueran etiquetas en lugar de números.

Sintaxis

```
enum <nombre de enumeración> { VALOR_1 < =inicio >,
                               VALOR_2,
                               VALOR_3,
                               . . .
                               VALOR_n
                               };
```

< nombre de enumeración > es opcional.

< =**inicio** > es un valor opcional de inicio en la enumeración. Si se omite, el valor por defecto se considera igual a cero.

Ejemplo 1.10

```
#define INICIO 48                /* ASCII de '0' */
void main()
{
      enum { GAUSS = INICIO,/* Inicializción a 0 */
             MONTANTE,
             SEIDEL,
             JACOBI,
             SALIR
            };
      char opcion;
           . . .
     printf("Solución de sistemas lineales\n");
     printf("0.- Eliminación de Gauss\n");
     printf("1.- Método de Montante\n");
     printf("2.- Gauss-Seidel\n");
     printf("3.- Iteración de Jacobi\n");
     printf("4.- Fin\n");
     printf("Teclee opción de 0-4\n");
 opcion = getchar();
      if (opcion == GAUSS)    gauss();
      if (opcion == MONTANTE) montante();
      if (opcion == SEIDEL)   seidel();
      if (opcion == JACOBI)   jacobi();
      if (opcion == SALIR)    exit(0);
           . . .
}
```

Más adelante se presentará una manera de realizar menús más elegante que la indicada hasta este momento, pero para ello será necesario recurrir a sentencias condicionales, específicamente a la sentencia **`switch( )`**.

`enum` también se usará en el software básico de métodos numéricos (ver **Anexo B**) para definir el tipo **`status`**, que representa un cierto tipo de dato booleano para algunas funciones: si la función no detectó un error de procesamiento retornará el "valor" **`BIEN`**, en otro caso retornará **`ERROR`**. Observe la definición de **`status`** a través de **`enum`** y **`typedef`**:

```
typedef enum {BIEN, ERROR} status;
```

En realidad el "valor" **BIEN** equivale numéricamente a **0** y **ERROR** a **1**. En ejemplos posteriores se hará uso de esta definición de tipo.

§1.6 FUNCIONES

Una *función* consiste en un bloque de código con un nombre único creado con un propósito específico. La tarea que deba realizar debe ser coherente y completa. Existen *funciones estándar*, que forman parte de la biblioteca (y no del lenguaje) y *funciones definidas* por el usuario con finalidades particulares.

• Funciones matemáticas

Las *funciones matemáticas* requieren el archivo **<math.h>**, que contiene diversos prototipos y constantes (π, $\pi/2$, $\pi/4$, e, $log(\pi)$ y otras). Las funciones aceptan argumentos tipo **double** y regresan un valor **double**; además los ángulos se representan en radianes. Un *error de dominio* ocurre si los argumentos caen fuera del dominio de la función. Un *error de rango* ocurre si el resultado no se puede representar como un **double**. En el **Anexo A** se muestran las funciones matemáticas más importantes. Las **macro funciones** también pueden ser consideradas de cierta manera como funciones del usuario. Una macro útil para calcular ángulos en grados en lugar de radianes es:

```
#define GRADO_RAD(n) (n)*M_PI/180.0
```

que acepta un número **n** en grados y efectúa la conversión automática a radianes por medio de la relación matemática:

$$\frac{\pi}{180} : \frac{x_{RADIANES}}{n_{GRADOS}}$$

de donde se tiene que:

$$x_{RADIANES} = n_{GRADOS}\left(\frac{\pi}{180}\right)$$

A la macros también se les puede incorporar un **(cast)** para garantizar que el valor de retorno sea de un tipo de dato específico requerido.

Ejemplo 1.11

```
#define GRADO_RAD(n) ((double) (n)*M_PI/180.0)
void main( )
{
      double x, y, z;
      scanf("%lf%lf",&x,&z);
      printf("%10.5f", (y=sin(GRADO_RAD( 2*x/(z+2.0) )) );
}
```

• Funciones definidas por el usuario

Una *función definida por el usuario* es un bloque de código creada con un propósito específico. La tarea que este bloque realice debe ser coherente y completa. Las funciones pueden regresar valores al sitio desde donde se les invocó o pueden alterar los valores de variables globales o de los argumentos enviados por referencia.

Sintaxis

```
tipo nombre_de_función(lista de parámetros)
{
      /* cuerpo de código; */
}
```

tipo es el valor que la función retornará; ***lista de parámetros*** son los parámetros que recibirá la función al ser llamada con los argumentos apropiados.

• La sentencia return

Las funciones pueden *no retornar* ningún valor (funciones tipo **void**) o *regresar un valor* de un cierto tipo al lugar que las invocó a través de la sentencia **return**.

Sintaxis

return *expresión*;

El siguiente programa de ejemplo calcula el paralelo de cuatro resistencias llamadas r1, r2, r3 y r4. Recuerde que el paralelo de dos resistencias *a* y *b* está dado por:

$$a \| b = \frac{a \cdot b}{a + b}$$

Listado 1.1. Paralelo.c

```
#include <stdio.h>
double paral(double, double);                    /* prototipo a º b
      */

void main(void)
{
        double paralelo;
        double r1, r2, r3, r4;
        printf("Déme las resistencias r1 y r2 en •: ");
        scanf("%lf%lf",&r1, &r2);
        printf("Déme ahora las resistencias r3 y r4 en •: ");
        scanf("%lf%lf",&r3, &r4);
        paralelo = paral(paral(r1,r2), paral(r3, r4));
        printf("  -> %fº%fº%fº%f = %10.5f •\n",r1,r2,r3,r4,paralelo);
```

```
}

double paral(double a, double b)
{
        return a*b/(a+b);
}
```

Por supuesto, el valor regresado por **return** podría ser ignorado completamente (lo que no causa ningún error de sintaxis salvo, quizá un mensaje de *warning* del compilador, pero lógicamente no sirve de nada) o ser utilizado adecuadamente en sentencias de asignación o en impresiones. Considere por ejemplo:

Ejemplo 1.12

```
#include <stdio.h>

int mul(int, int );

void main( )
{
      int x = 2, y = -8;
      int z;

      z = mul(x, y);         /* Línea 1 */
      printf("%d",mul(x, y));     /* Línea 2 */
      mul(x, y);             /* Línea 3 */
}

int mul(int a, int b)
{
      return a*b;
}
```

En el ejemplo anterior, en la línea 1 el valor de **x*y** se asigna a **z**, en la línea número 2 el valor se usa como salida de la función **`printf( )`**; y en la última línea el valor calculado se ignora y se pierde.

• Variables globales

Las *variables globales* se conocen a través del programa entero y se pueden usar en cualquier parte. Se escriben fuera de cualquier función o bloque de código. Cuando se encuentra una variable global y una local a una función con el mismo nombre, se usará la variable local (o local al bloque de código) sin afectar el valor de la variable global.

• Variables locales o automáticas

Las variables que se declaran dentro de una función o dentro de un bloque entre llaves se denominan **locales** o más técnicamente, ***automáticas***.

• Argumentos, parámetros y direcciones de memoria

Cuando se declara una función, se declaran también los *parámetros* que ésta recibirá para trabajar. Estos son llamados *parámetros formales* o simplemente *parámetros*. En el momento en que se invoca una función, estos parámetros son sustituidos por sus correspondientes *argumentos*, también conocidos como *parámetros actuales*.

Ejemplo 1.13

```
void main(void)
{
     int a = -5.0,
          b =  15.75;
                  ---------------- Argumentos
                  |  |
     c =integral(a, b);

}
                       ------------------- Parámetros
                       |        |            formales
float integral(float x, float y)
     {
        return( );
     }
```

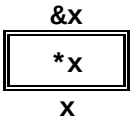
Los argumentos deben ser *consistentes* con los parámetros. Esta consistencia significa tener el *mismo tipo y orden posicional*, aunque ***C*** tratará de trabajar con argumentos de tipo erróneo aún sin avisar al programador, lo que causará inesperados errores en los cálculos.

• Llamadas por valor y por referencia

Existen dos formas de pasar argumentos a una función, por valor y por referencia. La *llamada por valor* ***copia*** el valor del argumento en el parámetro formal; y los cambios que se hagan en los parámetros no tienen efecto en los argumentos. La *llamada por referencia* ***copia la dirección*** de un argumento en el parámetro, esto significa que lo que se hace con el parámetro ***afectará*** al argumento.

Al declarar la función se deben usar apuntadores antecedidas por el operador *****; y al llamar la función, a cada argumento se le antepondrá el símbolo ampersand **&** (que indica **referencia** a la **dirección de la variable**). Es esa la razón por la cual **`scanf()`** requiere el **&** antepuesto a las variables que se leen. **&** es el operador de referencia a una variable. El operador ***** se conoce como **desreferenciador** de una

localidad de memoria: es decir manipula el contenido de la localidad. El **&** es un **referenciador** a dirección de memoria. ***** aplicado a una variable significa *"el valor contenido en la variable"*; y es -por tanto- el operador inverso de **&**. Así, dada una variable **x** se tiene:



donde:

- **x** : Nombre de la variable.
- ***x**: Valor contenido en x.
- **&x**: Dirección en memoria de x.

Ejemplo 1.14

El siguiente programa usa las llamadas por referencia para intercambiar (realizar un *swap*) los valores de las variables **x, y**.

```
void swap(int *, int *);    /* intercambio   */

void main( )
{
      int   x=10,
            y=20;

      printf("x = %3d, y = %3d\n",x,y);
      swap(&x, &y);
      printf("x = %3d , y = %3d\n",x,y);
}

void swap(int *a, int *b)
{
      int temp;

      temp=*a;         /* guarda el contenido de x */
      *a=*b;           /* y = x */
      *b=temp;         /* x = y */
}
```

---

§1.7 LOGICA BASICA Y CONTROL DE PROGRAMA

• La sentencia if ( )

La sentencia **if( )** proporciona la forma de realizar una bifurcación condicional que depende

del valor de una *condición* predeterminada. La condición a verificar puede ser cualquier expresión válida en ***C*** que no necesariamente debe contener operadores lógicos. El flujo de la ejecución dependerá entonces del valor lógico resultante de esta expresión.

Sintaxis

```
if (condición) sentencia_1;
        < else sentencia_2; >
```

La parte correspondiente a la sentencia **`else`** es opcional. Si la *condición* es **verdadera** (o sea **distinta de `0`**) entonces se ejecutará la *sentencia_1*, en caso contrario se ejecutará la *sentencia_2* o el código inmediato de abajo.

Ejemplo 1.15

```
int   x, y,
      producto;

scanf("%d%d",&x,&y);

if ( (producto = x * y) < 0 )
     printf("número negativo\n");
else printf("Producto = %d\n",producto);
```

Por supuesto pueden usarse **`if( )`** anidados para tener sentencias como:

```
if(a)
      if(b)
            if(c)
                  printf("Sentencia 1");
else printf("Sentencia 2");
```

El else corresponde al último **`if( )`** que no tenga una sentencia **`else`**, en este caso al **`if(c)`**. Cualquier expresión válida en ***C*** puede controlar un **`if( )`**, no necesariamente deben ser expresiones que impliquen operadores relacionales o lógicos.

Ejemplo 1.16

```
. . .
printf("Déme 2 números: ");
scanf("%d%d",&a,&b);

if (b)
      printf("División = %f\n", (float) a/b);
else printf("No se puede dividir entre 0.\n");
```

Es innecesario hacer:

```
if(b!=0)
     printf("División = %d\n",a/b);
else printf("No se puede dividir entre 0.\n");
```

porque la sentencia **if(b)** realiza la pregunta:

"si b existe (o sea si b diferente de cero) entonces..."

Escribir:

```
if(b!=0)
  . . .
```

además de ser mal visto es redundante e ineficaz desde el punto de vista de la programación.

La solución de una ecuación cuadrática del tipo:

$$ax^2 + bx + c$$

puede resolverse usando la fórmula:

$$x_{1,2} = \frac{-b +\_ \sqrt{b^2 - 4ac}}{2a}$$

que produce dos raíces x_1 y x_2. Debe considerarse que el valor del discriminante:

$$D = b^2-4ac$$

determina si las raíces serán reales o complejas por el criterio:

$D \geq 0$: *raíces reales*, en otro caso implica *raíces complejas.*

Listado 1.2. Ecuacion.c

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

#define D(a,b,c) ((double) (b)*(b)-4*(a)*(c))
void ec_cuad(double a, double b, double c);

void main( )
{
     double a, b, c;
```

```
      printf("Déme los coeficientes a, b y c: ");
      scanf("%lf%lf%lf",&a,&b,&c);
      printf("Resolviendo: ");
      printf("%+5.5f x² %+5.5f x %+5.5f = 0\n",a,b,c);
      if (!a) printf("No es ec. cuadrática...\n");
           else ec_cuad(a,b,c);
} /* main */

void ec_cuad(double a, double b, double c)
{
  double x1, x2;                      /* reales  */
  double real, imaginaria;       /* complejas    */

  if (D(a,b,c)<0)
      {
          printf("\n\n*** Raíces complejas:\n");
          real  = -b/(2*a);
          imaginaria =  sqrt(-D(a,b,c))/(2*a);

          printf("X1 = %8.4f %+8.4fi\n", real,  imaginaria);
          printf("X2 = %8.4f %+8.4fi\n", real, -imaginaria);
      }
      else
      {
           printf("\n\n*** Raíces reales:\n");
           x1 = (-b + sqrt(D(a,b,c)))/(2*a);
           x2 = (-b - sqrt(D(a,b,c)))/(2*a);
           printf("X1 = %+f\nX2 = %+1f\n",x1,x2);
      }
}
```

- **La sentencia `break`**

Sintaxis

```
break;
```

Es una instrucción usada para terminar o salir de un bloque de sentencias en forma incondicional. En ciclos, decisiones o sentencias **`switch( )`** será comúnmente empleada como se verá más delante.

- **La sentencia `switch ( )`**

A través de sentencias **`if( )-else-if( )`** anidados se pueden hacer comprobaciones múltiples pero en una forma poco elegante; por ello ***C*** tiene incorporada una sentencia de multibifurcación condicional que permite realizar dicho trabajo de una forma más adecuada.

Sintaxis

```
switch(variable)
{
  case constante_1: sentencia 1; break;
  case constante_2: sentencia 2; break;
     .      . . .        .          .
  case constante_n: sentencia n; break;
  <default: sentencia n+1;>
}
```

Si el valor de la **variable** coincide con algún valor predefinido en la i-ésima constante, se ejecuta el bloque correspondiente de instrucciones, si no se continúa verificando con otra constante. Un **case** puede no tener ninguna sentencia asociada. La parte correspondiente a **default** es opcional e indica que la **sentencia n+1** será ejecutada cuando la variable no coincida con el valor de alguna de las constantes.

Ejemplo 1.17 Menu de opciones

```
enum { TRAP = 48, SIMPSON, CUADRATURA, FIN };
char op;
printf("**** Integración numérica ***\n");
printf("     Escoja una opción de 1 - 4\n\n");
printf("1.Método del Trapecio\n");
printf("2.Método de Simpson  \n");
printf("3.Método de Cuadratura Gaussiana\n");
printf("4.Fin del programa\n");
op=getchar();    /* lee carácter del teclado */
switch(op)
    {
      case TRAP      : trapecio();  break;
      case SIMPSON   : simpson();   break;
      case CUADRATURA: gaussiana(); break;
      case FIN :{
                      printf("Fin de uso del programa.\n");
           exit(0);
           } break;

      default: printf("Opción errónea\n"); break;
    }  /* switch op */
```

- **Ciclos iterativos `for( )`, `while( )` y `do-while( )`**

Existen tres tipos de ciclos iterativos: el ciclo **`for( )`**, el ciclo **`while( )`** y el ciclo **`do-while( )`** que permiten ejecutar múltiples veces grupos de instrucciones. A continuación se explicará

el funcionamiento de cada uno de ellos.

• El ciclo `for( )`

Este ciclo se usa cuando se conoce el número de veces (o la condición exacta) en que finalizará un grupo de sentencias.

Sintaxis

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

- La ***inicialización*** es una expresión usada para establecer el inicio del ciclo.
- La ***condición*** es una expresión que comprueba la variable de control: si la condición es igual a cero (o falsa), el ciclo termina.
- Finalmente, el ***incremento*** es una expresión que define la manera en que cambiará la variable de control.

Cabe mencionar que las tres partes que constituyen una iteración `for( )` son opcionales. Es posible incrementar el desempeño de la iteración `for( )` realizando combinaciones adecuadas de las tres componentes.

Ejemplo 1.18

```
#define LIMITE 10

int x;
    . . .

printf("Lista ascendente:\n");
for(x=0; x<LIMITE; x++)
    printf("%5d\n",x);

printf("Lista descendente:\n");
for(x=LIMITE; x>0; x--)
    printf("%5d\n",x);
    . . .
```

Ejemplo 1.19 **Factorial iterativo**

El *factorial* de un número entero *n* se define por:

```
n! = n.(n-1).(n-2)...1
```

con `0! = 1`. La siguiente función permite su cálculo:

```
long double fact_iter(int n)
```

```
{
      double resultado = 1.0;

      if(n<0) return 0;
      for( ; n; n--)
            resultado *= n;
      return (resultado);
}
```

Ejemplo 1.20 **Fibonacci iterativo**

La siguiente función permite calcular la serie:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

que es una secuencia conocida como serie de *Fibonacci* (vea en §1.8 la definición de esta secuencia). Verifique que la función realmente la genera:

```
double fib_iterativo(int n)
{
       double   primero, segundo, temp;
       register int i;

       primero = segundo = 1.0;
       for(i=1;i<=(n-2);++i)
           {
             temp=primero;
             primero+=segundo;
             segundo=temp;
           }
       return(primero);
}
```

• Variantes del ciclo for ( )

El ciclo **for( )** del lenguaje ***C*** es un ciclo mucho más poderoso y flexible que su equivalente en otros lenguajes de programación. De hecho constituye la iteración más completa que pueda encontrarse en cualquiera de ellos. La flexibilidad de su desempeño se basa en las variantes que pueden presentar sus componentes como se verá a continuación.

1) Permite *argumentos múltiples* usando el operador coma (**,**):

```
for (x=0, y=LIMITE; x < y; x++, y--)
```

2) Se puede tener *condición implícita de terminación* si se omite alguna de las piezas de definición, típicamente el **incremento** y/o **inicio**:

```
for(x=1; x>0; )
   {
      scanf("%d",&x);
      printf("√%5.2f = %5.2f",(float) x, sqrt(x));
   }
```

3) *Ciclos infinitos*:

```
for( ; ; )
  {
   ch=getche();
   . . .
   if(ch == 'F') break;
      } printf("Usted tecleó una %c\n",ch);
```

4) *Ciclos sin cuerpo*:

```
#define TIEMPO 10000

 . . .

for(i=0; i<TIEMPO; i++);  /* retardo de tiempo */
```

5) La variable de control puede ser de *cualquier tipo*:

```
for( c = 'a';c < 'z'; c++)
```

6) La condición de prueba *no tiene que hacerse sobre el contador* necesariamente:

```
for( i = 0; x <LIMITE; i++)
```

7) La condición puede ser *cualquier expresión*:

```
for(c = 'a'; c & 0x0F; c++)
```

8) La variable de control *conserva su valor* dentro y fuera del ciclo.

9) La inicialización *no tiene que ser una asignación*:

```
for(printf("Dame un dato");(c=getch()) != '\'; printf("%c",c))
```

10) Si el incremento no se define, *se supone igual a 1*.

• El ciclo `while( )`

La iteración **`while( )`** se puede utilizar cuando no se sabe exactamente cuántas veces se ejecutará la sentencia correspondiente, porque ésto depende de un criterio condicional.

Sintaxis

```
while (condición) sentencia;
```

La *condición* es una expresión que determina cuál es el momento exacto para detener la ejecución del ciclo. El ciclo **while( )** trabajará *mientras la condición sea verdadera*; y si ésta es falsa al principio, podría no ejecutarse nunca.

Ejemplo 1.21

La suma:

$$\sum_{i=1}^{n} i = 1+2+3+...+n$$

en algún momento pasa de **100**. Realice un programa que determine el primer entero que hace que la suma rebase **100**.

```
#define MAX 100  /* Límite de términos */
void main( )
{
  int    n = 0,
      suma = 0;

  while(suma <= MAX)
       {
            n++;
            suma += n;
        }   /* while */

  printf("La • pasa de %d cuando se agrega %d\n",MAX, n);
  printf("• = %10d\n",suma);
}
```

Un ciclo **while( )** podría no tener ninguna sentencia ejecutable, como se muestra a continuación:

```
while ((ch = getche( )) !='A');
```

lo que es equivalente a:

```
ch ='\0';  /* inicializa ch */

while (ch !='A')
    ch = getche( );
```

en ambos casos el ciclo **while( )** trabaja hasta que se teclee una **'A'** mayúscula, solo que es preferible el uso de la primera construcción porque es más consistente y elegante.

• El ciclo `do-while( )`

Al contrario del ciclo **`while( )`** que prueba la condición *antes* de ejecutar la sentencia, la iteración **`do-while( )`** *ejecuta al menos una vez* la sentencia y posteriormente verifica la condición.

Sintaxis

```
do
      sentencia;
while (condición);
```

Ejemplo 1.22

```
int num;
do {
     scanf("%d",&num);
         printf("³%d³ = %5.5f\n",fabs(num));
   } while(num>300);
. . .
```

Este ciclo se ejecutará mientras la entrada en la variable **`num`** sea menor de 300.

• Ciclos anidados

Naturalmente es posible anidar cualesquiera de las sentencias de ciclos para obtener ciclos dentro de ciclos. Esta es la característica poderosa de los lenguajes de programación porque permite repetir múltiples veces los cálculos variando ciertas condiciones en cada cada ocasión. En métodos numéricos el manejo de ciclos anidados es esencial cuando se trabaja con matrices y vectores. Un ejemplo de ciclos anidados es mostrado a continuación:

Ejemplo 1.23

```
#define MAX 100

for(i=0; i<MAX; i++)
   for(j=0; j<MAX; j++)
       printf("%d %d\n",i, j);
```

El ciclo anterior se ejecutará **`MAX*MAX`** veces, que en este caso corresponden a 10,000 iteraciones

• Arreglos y cadenas de caracteres

Un ***arreglo*** es un conjunto de variables del mismo tipo que se accesan usando un nombre

común. En ***C***, un arreglo puede tener una o varias dimensiones. Para tener acceso a un elemento específico del arreglo se utiliza un *índice*. Un arreglo de **n** elementos empieza a direccionarse con el elemento **0** y finaliza con el índice en **n-1**. Es una buena idea declarar índices en **`register`**, para acelerar su desempeño. Un arreglo común es el de caracteres: ya que ***C*** no tiene incorporado el tipo de dato *cadena*, se debe usar un *arreglo de caracteres*.

Sintaxis

`tipo arreglo[tamaño`$_1$`][tamaño`$_2$`]...[tamaño`$_n$`];`

`tipo` es el tipo de dato base del arreglo.
`tamaño`$_1$**`,..., tamaño`**$_n$ definen el número de dimensiones que tendrá el arreglo en cuestión. Este número está limitado por la memoria de la máquina.

Ejemplo 1.24

```
#define MAX 10

int x[MAX];
register int i;

for(i=0; i<MAX; i++)
     printf("%3d ",( x[i]=i ) );
```

A través de **`typedef`**, es más fácil declarar matrices. Unas declaraciones que se utilizarán a menudo en la parte de métodos numéricos serán precisamente, las relacionadas con el manejo de matrices y vectores:

```
#define N 20
#define M N
typedef double MATRIZ[N][M];
typedef double VECTOR[N];
```

Con estas definiciones puede escribirse declaraciones para variables tipo **`MATRIZ`** y **`VECTOR`** de manera intuitiva, por ejemplo:

```
MATRIZ     a, b, c;
VECTOR     x;
VECTOR     y;
```

Ejemplo Estadística elemental

Suponga que *n* números reales se leerán dentro de un vector *X* en la memoria. Determine la *media* de **n** números $\mathbf{X_i}$, la *varianza* σ^2 y la *desviación estándar* σ de dicho vector **X**. De estadística se sabe que:

$$\overline{X} = \frac{1}{n}\sum_{i-1}^{n} X_i \qquad \sigma = \sqrt{\sigma^2}$$

$$\sigma^2 = \frac{1}{n\text{-}1}\sum_{i=1}^{n} (X_i - \overline{X})^2$$

El vector $\mathbf{X_i}$ puede ser declarado como:

```
#define N 20

typedef double VECTOR[N];
VECTOR X;
```

Puede resolverse el problema propuesto en una sola función principal pero ello no es muy elegante. Mas adelante se realizará un programa que calcule los valores de la media, varianza y desviación estándar usando vectores como argumentos entre funciones.

• Comprobación de fronteras

Los compiladores de ***C*** no comprueban las fronteras de un arreglo: el intento de acceso de elementos por encima o por debajo de la frontera probablemente ocasionará la caída del sistema, errores de cálculo o eventualmente funcionará bien. Este hecho puede resultar poco agradable a muchos programadores acostumbrados a que el compilador revise el manejo de arreglos en las fronteras; sin embargo esto tiene un objetivo: al no comprobar de manera continua las fronteras del arreglo, se incrementa mucho la velocidad de ejecución del programa (situación siempre deseable en métodos numéricos) al liberar al compilador de la pesada tarea de comprobar índices y variables. La responsabilidad del programador al verificar que se respeten las fronteras de los arreglos al manipularlos tiene un premio: la programación disciplinada en el manejo de índices y un código rápido por parte del compilador se traducen en programas elegantes al leerlos y eficientes al momento de ejecutarse.

• Paso de arreglos a funciones

Es posible pasar la dirección de un arreglo como argumento entre funciones, usando su nombre. Los arreglos siempre son pasados **por referencia** entre las llamadas, lo que ocasiona que lo que la función haga con el arreglo sí modificará los elementos del mismo, pues lo que en realidad se pasa es la dirección en memoria del primer elemento del arreglo.

Ejemplo 1.25

```
#define N 10
#define M 5
#define O M
```

```
void main(void)
     {
       int vector[N][M][O];
           . . .
       funcion(vector);
           . . .
     }
```

Si una función recibe un arreglo, se puede declarar el parámetro indicando el nombre del arreglo y sus dimensiones u omitiendo la primera de ellas:

```
funcion(int x[N][M][O])
     {
          . . .
     }
```

ó bien como:

```
funcion(int x[][M][O])
     {
          . . .
     }
```

El programa que resuelve problemas de estadística elemental usando paso de arreglos como parámetros a diversas funciones es mostrado en el listado 1.3:

Listado 1.3. Estadist.c

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

#define N 20

typedef float real;
typedef real VECTOR[N];

real            media(VECTOR);
real            varianza(VECTOR);
real            desv_std(void);
void            lectura(VECTOR);

int   n;        /* Número de elementos*/
real xm;        /* media              */
real var;       /* •²                 */
real ds;        /* •                  */


void main( )
```

```
{
        VECTOR X;          /* X[i]; i=1,2,...,LIMITE            */

        lectura(X);
        xm = media(X);
        var = varianza(X);
        ds = desv_std();
        printf("Xm = %f   •² = %f  • = %f\n",xm,var,ds);
} /* main */

void lectura(VECTOR A)
{
        int i;

        printf("Número de datos: ≤ %d ",N);
        scanf("%d",&n);
        for(i=0; i<n; i++)
           {
                printf("Déme elemento A[%d]: ",i);
                scanf("%f",&A[i]);
           }
} /* lectura */

real media(VECTOR A)
{
        real suma = 0;
        register int i;

        for(i=0; i<n; i++)
                 suma+=A[i];
        return(suma/(real) n);
} /* media */

real varianza(VECTOR A)
{
        real suma = 0;
        register int i;

        for(i=0; i<n; i++)
                 suma+=pow((A[i] - xm),2);
        return( suma/ (real) (n-1));
} /* varianza */

real desv_std(void)
{
        return(sqrt(var));

} /* desv_estándar */
```

• Cadenas de caracteres

Un uso común de los arreglos se presenta en las cadenas de caracteres. Dichas cadenas son definidas como un *vector de caracteres* terminados con el carácter nulo **'\0'**.

Sintaxis

```
char cadena[longitud];
```

Por ejemplo, una cadena de caracteres como **"Hola, qué tal"**, se almacena internamente como:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
H	o	l	a	,		q	u	é		t	a	l	\0

La cadena anterior requiere 14 bytes para almacenar los caracteres de información, incluyendo al carácter terminador **'\0'**.

• Funciones de cadenas

El archivo de encabezado **<string.h>** tiene incorporadas varias funciones especiales para el manejo de cadenas de caracteres. Estas cadenas pueden ser tratadas en forma similar a vectores, con elementos individuales tipo carácter que se pueden manipular por separado. Esto significa que si se declara:

```
char ALFA[10];
```

se tienen espacio en memoria para almacenar caracteres en los sitios:

```
ALFA[0], ALFA[1],..., ALFA[8]
```

Todas esas localidades contienen información, y finalmente:

```
ALFA[9]='\0'
```

contiene al terminador nulo.

La siguiente tabla **1.9** muestra algunas funciones de cadenas de caracteres del archivo **<string.h>**.

Tabla 1.9 Funciones de cadenas

Función	Descripción
strcpy(c1,c2)	Copia la cadena c2 en c1

strcat(c1,c2)	**Concatena c2 al final de c1**
strlen(c1)	**Devuelve longitud de c1**
strcmp(c1,c2)	**Devuelve 0 si c1=c2** **Devuelve >0 si c1>c2** **Devuelve <0 si c1<c2**
puts(c)	**Escribe c en la consola y añade un carácter \n al final.**
gets(c1)	**Lee c1 del teclado y añade \0 al final**

Ejemplo 1.26

Una forma de verificar si dos cadenas son iguales (es decir, tienen los mismos caracteres ASCII en el mismo orden posicional) puede realizarse con el siguiente fragmento de código (suponga que tanto **c1** como **c2** son variables de tipo cadena):

```
if(!strcmp(c1, c2))
      printf("iguales");
else printf ("distintas");
```

• Estructuras

En ***C*** una *estructura* es un conjunto de variables (de tipos probablemente diferentes) que se accesan bajo un nombre común. La estructuras son similares en cierta forma al **record** de PASCAL.

Sintaxis

```
struct <nombre de estructura>
  {
      tipo variable_1;
      tipo variable_2;
            . . .
      tipo variable_n;
  }  <lista de variables>;
```

En la sintaxis es posible omitir el **nombre de estructura** o la **lista de variables**, pero no ambas.

Ejemplo 1.27

```
struct complejo
  {
      double re;
      double im;
```

```
} u, v;
```

El ejemplo anterior define una estructura y crea dos variables **u, v** del tipo **`struct complejo`**.

Ejemplo 1.28

```
struct {   double real;
           double imaginario;
       }  u;
```

En este caso, como sólo se declara una variable, no es necesario incluir el nombre de la estructura.

• Acceso de elementos en las estructuras

Se puede accesar los elementos individuales de cualquier estructura a través del operador **.** (punto). Este operador permite manipular los campos de la estructura como si fueran variables sencillas.

Ejemplo 1.29

```
struct complejo
     {
          double re;
          double im;
     } u, v;
. . .
u.real = 7.566;
u.imaginario = -1.4142;
```

para imprimir el número complejo **u** puede usar:

```
printf("x = %10.5f %+10.5f\n" ,u.real, u.imaginario);
```

• Arreglos de estructuras

Cuando se necesitan diversas variables de tipo estructura, puede crearse un arreglo de estructuras con la siguiente sintaxis:

`struct nombre variable[tamaño`$_1$`][tamaño`$_2$`]...[tamaño`$_n$`];`

Ejemplo 1.30

```
#define N 100

struct complejo vector[N];
      . . .
for (i=0;i<N;i++)
    {
     vector.real[i]=sin(i);
      vector.imaginario[i]=cos(i);
    }
```

§1.8 RECURSIVIDAD

Una función en ***C*** puede realizar un *llamado recursivo* a sí misma. La recursión puede ser *total* o *mutua*. La recursión total tiene lugar cuando una función **A( )** se llama a sí misma. La recursión mutua es cuando una función **A( )** llama a otra función **B( )**, y **B( )** a su vez llama a **A( )**. La recursión se da de manera natural en funciones matemáticas y en la naturaleza: un ejemplo de la recursividad total tiene lugar en la función **factorial**. Esta función puede expresarse en forma iterativa como ya se había mencionado:

$$n! = n(n-1)(n-2)...1$$

con $0! \equiv 1$. Si se desea calcular **4!**, debe efectuarse **4! = 4.3.2.1** para obtener **4! = 24**.

Pero también es posible definir el factorial de un número en términos de sí mismo como:

$$n! = n(n-1)!$$

con $0! \equiv 1$; expresando *n!* en términos más sencillos de *(n-1)!*. De esta forma, se tiene para **4!**:

```
4! = 4.3!
       3! = 3.2!
              2! = 2.1!
                     1! = 1.0!
                            0! = 1
```

y el resultado final equivale a multiplicar los resultados parciales de cada llamada recursiva, en este caso:

```
4.(3!=3.(2!=2.(1!=1.(0!=1)))) = 4.3.2.1.1 = 24.
```

Por supuesto las llamadas sucesivas (recursivas) a la misma función con un argumento distinto cada vez van obteniendo resultados parciales que deben ser guardados para poder hallar el resultado final. Afortunadamente el compilador se encarga de llevar la administración de todo ello, y el programador solamente necesita definir *claramente* la función a emplear. Una restricción importante al empleo de la

recursividad en el software es la rapidez de ejecución: los programas recursivos resultan ser mucho más lentos que los iterativos. En métodos numéricos la recursividad es poco utilizada por este hecho, pero en algunos casos resulta ser la solución más sencilla y elegante: ver por ejemplo las funciones especiales en la parte VI del **Anexo D**, en particular los polinomios ortogonales.

Ejemplo 1.31 Factorial recursivo

La función para calcular el factorial recursivo es presentada a continuación. Observe la simplicidad de la misma y su programación directa de la definición y compare con la versión iterativa presentada **§1.7**, en el ejemplo **1.22**.

```
long double fact_recur(int n)
{
      if(n<=0)
            return 1;  /* 0! ≡ 1 */
      return ( n*fact_recur(n-1) );
}
```

Ejemplo 1.32 La serie de Fibonacci

Los *números de Fibonacci* son una secuencia famosa de términos: los dos primeros son iguales a uno; de ahí en adelante cada número es la suma de los dos anteriores. Los siguientes son los primeros 12 números de Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

La sucesión de Fibonacci está determinada por las ecuaciones:

$$Fib_1 = 1;$$
$$Fib_2 = 1;$$
$$Fib_n = Fib_{n-2} + Fib_{n-1};$$

También existe una fórmula para determinar el *n*-ésimo número de Fibonacci, si $n \geq 5$:

$$Fib_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

Hay por tanto, tres formas de evaluar esta serie: por fórmula, por iteración y por recursión. Inmediatamente se muestra la programación del método por fórmula y el método recursivo, el cálculo iterativo se realizó ya en **§1.7**, en el ejemplo **1.23** cuando se estudiaron los ciclos.

```
double fib_formula(int n)
{
  return((pow((1+sqrt(5))/2.0,n+1)-
          pow((1-sqrt(5))/2.0,n+1))/sqrt(5));
}
double fib_recursivo(int n)
{
        if(n<3)
            return(1);
        return ( fibonacci(n-1)+fibonacci(n-2) );
```

```
}
```

§1.9 APUNTADORES

Un *apuntador* es una variable que apunta a un objeto en la memoria. Esto es, contiene la *dirección* de una localidad de la memoria de la máquina donde se encuentra un dato. Por tal motivo, es posible manejar tanto direcciones de memoria (en forma transparente para el programador), como los contenidos de dichas localidades, para lograrlo, ***C*** posee los operadores: **&** y *****

• Operadores & y *

& Este operador se refiere siempre a localidades de memoria (es decir, a direcciones en la memoria).
***** Se refiere a los contenidos de las localidades de memoria.

Sintaxis

```
tipo *variable_apuntador;
```

Ejemplo 1.33

En cierta dirección en memoria se tiene el entero 136, que ocupa dos bytes de espacio. Se puede definir una variable *apuntador* para manipular a este dirección (y por ende al contenido de la misma) como:

```
int   *p;   /* definición de un apuntador a un entero */
```

Que se debe leer: *"p es una variable apuntador, que apunta a un entero"* y no *"es una variable entera que apunta a una dirección"*.

Ejemplo 1.34

```
int a=20, b=0;
int *p;

...

p=&a;   /* asigna a p, la dirección donde se encuentra a */
b=*p;   /* el contenido de la dirección de p, se asigna a b */
```

• Aritmética de apuntadores

Ciertos operadores pueden ser usados con apuntadores para realizar algunas operaciones aritméticas con ellos:

- El operador incremento **++** no incrementará al apuntador en 1, sino en *una localidad de memoria*, que depende del *tipo* de la variable al cual se apunta.
- El operador decremento **--** es casi el mismo caso anterior excepto que decrementa una localidad de memoria.
- El operador suma **+** sirve para sumar enteros a un apuntador, pero el resultado es el incremento del número de localidades del tipo al que se apunta.
- El operador resta **-** es un caso parecido al anterior sólo que se disminuyen localidades apuntadas.

No son permitidas ni la multiplicación ni la división; tampoco la suma o resta entre apuntadores, operadores de bits, operaciones lógicas y tampoco sumas y restas con tipos float o double.

• Apuntadores y arreglos

Normalmente un arreglo es manejado por *indexación*, esto es: se manipulan los elementos con el índice del arreglo; sin embargo, también puede manejarse los elementos a través de apuntadores y generalmente esta opción proporciona un acceso más rápida y resulta preferida por los programadores profesionales.

Ejemplo 1.35

```
#define N 100
char vector[N];
char *c;
```

La asignación:

```
c = &vector[0];
```

hace que **c** apunte al primer elemento del arreglo **vector**, que es exactamente igual a:

```
c=vector;
```

dado que el valor de una variable arreglo es la dirección del elemento **0** del mismo. Existe sin embargo, una diferencia: un apuntador es una variable; en consecuencia:

```
c = nom;
```

ó:

```
c++;
```

son expresiones válidas, pero el nombre de un arreglo *no es una variable* y

```
nom = c;
```

ó

```
nom++;
```

no son expresiones válidas.

• Apuntadores a funciones

Dado que todas las funciones tienen el mismo alcance y no pueden *anidarse* unas dentro de otras, ***C*** admite apuntadores a funciones que pueden pasar como argumentos. Los apuntadores son -como se había mencionado- una herramienta realmente poderosa del lenguaje. Los apuntadores pueden aplicarse a variables sencillas, a direcciones de arreglos y también a funciones; porque a pesar de que una función *no es una variable*, posee una dirección física en memoria.

Por ejemplo, se quiere calcular:

$$\sum_{i=0}^{i=n} f^3(i)$$

para un número variado de funciones ***f***. Especialmente quizá, ***f*** pueden ser alguna de las siguientes:

$$f(i): seno(i), \frac{1}{i}, \sqrt{i^2\text{-}i+1}, \sum_{k=0}^{10} \ln(i)$$

En lugar de escribir varias funciones en ***C*** específicas, es preferible construir una función en ***C*** que permita llamar la sumatoria con distintas funciones matemáticas como argumentos en cada ocasión.

Sintaxis

```
tipo (*funcion)();
```

`tipo` es algún tipo de datos y el **`(*funcion)( )`** indica apuntador a la función **`*funcion`**. Los paréntesis en **`(*funcion)`** son necesarios por sintaxis. En la declaración anterior, **`funcion`** es el apuntador a función, **`*funcion`** es la función, y finalmente **`(*funcion)(x)`** es la llamada a la función con el argumento **`x`**.

Siguiendo el ejemplo anterior, para hallar la sumatoria de funciones elevadas al cubo se debe construir:

```
double suma(double (*f)(),
            int n
           )
{
  double   k,
           suma = 0;

  for(k=1; k<n; ++k)
      suma += pow((*f)(k), 3);
```

```
  return(suma);
}
```

Un programa manejador para el problema planteado podría ser algo como:

```
void main()
{
      printf("Usando función seno   : %10.5f\n", suma(sin , 100));
      printf("Usando función inversa: %10.5f\n", suma(inv , 20));
      printf("Usando un polinomio   : %10.5f\n", suma(poli, 10));
      ...
}
```

Observe como la función **suma( )** acepta dos argumentos: por una parte el *nombre* de la función matemática a procesar y por otra el *número* de términos a sumar. La definición de las otras funciones: inversa, polinomio y logaritmo puede ser realizada con el código que se indica a continuación:

```
double inv(double x)
{
      return 1/x;  }

double poli(double x)
{
      return sqrt(x*x - x + 1);  }

double loga(double x)
{
   int i=0;
   double suma=0;

   for(i=0; i<10; i++)
       suma+=log(x);
   return suma;
}
```

El programa completo incluyendo archivos de encabezado y prototipos se reproduce a continuación en el listado 1.4:

Listado 1.4. Apuntar.c

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

double inv (double);                  /* Prototipos      */
double poli(double);
double loga(double);
double suma( double (*f)(), int n );

void main( )
{
```

```
        printf("Usando función seno   : %10.5f\n", suma(sin , 100));
        printf("Usando función inversa: %10.5f\n", suma(inv , 20));
        printf("Usando un polinomio   : %10.5f\n", suma(poli, 10));
        printf("Usando la sumatoria   : %10.5f\n", suma(loga, 75));
}


double suma( double (*f)(), int n )
{
  double k,
      suma = 0;

  for(k=1; k<n; ++k)
        suma += pow((*f)(k), 3);

  return(suma);
}


double inv(double x)
{
        return 1/x;
}


double poli(double x)
{
        return sqrt(x*x - x + 1);
}

double loga(double x)
{
   int i=0;
   double suma=0;

   for(i=0; i<10; i++)
       suma+=log(x);
   return suma;
}
```

§1.10 ARCHIVOS EN C

C es un lenguaje de programación poderoso que permite manejar la entrada y salida (E/S) de una manera sencilla y transparente para el programador. A pesar de que el lenguaje en sí carece de

funciones tan elementales como leer de teclado o escribir en pantalla, estas son construidas de modo que puedan residir como funciones de bibliotecas separadas. De esta manera los compiladores de C poseen una poderosa biblioteca estándar de E/S para ser usada por los programas del usuario.

• Streams y archivos

Conceptualmente el lenguaje ***C*** proporciona una interfase consistente e independiente del dispositivo físico real denominada *stream*, que representa un flujo de datos. Existen dos tipos de *stream*:

a) textuales y
b) binarios.

Los *stream textuales* son secuencias de caracteres agrupados en líneas terminadas por el carácter ´**\n**´. Normalmente estos *streams* son almacenados en algún código estándar como **ASCII** y son legibles por un usuario en forma directa. Los *stream binarios*, por otra parte son secuencias de bytes que corresponden a programas ejecutables o programas objeto y no son legibles. Ambos tipos de *streams* son terminados por el carácter **`EOF`**[9] que marca el fin del archivo. Cada *stream* es asociado a un archivo usando una estructura tipo **`FILE`**. Se debe declarar entonces un apuntador al archivo a través del cual poder manipularlo. Todas las funciones requeridas para el manejo adecuado de archivos se hallan en **`<stdio.h>`**.

Tabla 1.10 Funciones de manejo de archivos.

Función	Significado
`fopen( )`	Abre un stream
`fclose( )`	Cierra un stream
`fprintf( )`	Análogo a printf( )
`fscanf( )`	Análogo a scanf( )
`putc( )`	Escribe un carácter en un archivo
`getc( )`	Lee un carácter desde un archivo
`fseek( )`	Posiciona el apuntador a registros dentro de un archivo en operaciones de acceso aleatorio.
`feof( )`	Devuelve 1 si se alcanza la marca EOF (fin de archivo)

• El apuntador a archivo

El ***apuntador a archivo*** es el medio a través del cual un programa tiene acceso a un *stream*.

[9] **EOF** son las siglas en inglés de End Of File

Para cada uno de ellos existe un *apuntador a archivo* único que establece la liga lógica con el archivo físico.

Sintaxis

```
FILE *apuntador;
```

• Funciones de manipulación de archivos

La función fopen( )

La función **`fopen( )`** sirve para dos propósitos, primero abre un *stream* para usarse y posteriormente enlaza dicho *stream* con un archivo físico.

Sintaxis

```
apuntador_archivo = fopen("archivo","modo");
```

El "**modo**" corresponde a una forma específica de abrir un archivo: sólo para lectura, sólo para escritura, modo para agregar datos a un archivo existente o combinado. Dichas formas son presentadas en la tabla **1.11** siguiente.

Tabla 1.11 Tabla de "modo"

Modo	Significado
`"r"`	Sólo lectura
`"w"`	Para escritura
`"a"`	Añadir al final de un archivo
`"r+"`	Lectura y escritura
`"w+"`	Lectura y escritura

`"archivo"` también puede tomar el nombre del dispositivo impresora, que corresponde a **`"prn"`**, para utilizarla como salida de datos. Por supuesto, el nombre del archivo puede estar contenido en una variable de tipo cadena de caracteres que previamente se asignó o se leyó desde teclado.

Ejemplo 1.35+1

```
FILE *fp;
. . .
```

```
fp = fopen("A:\\programa\\prueba.txt","r");
```

Normalmente no debe usarse la declaración anterior a menos que se esté *completamente seguro* que existe el archivo llamado **"prueba.txt"** en la unidad A: y en el subdirectorio **"\programa"** (¡el estar completamente seguro de tres cosas simultáneas como las anteriores es cosa rara en la mayoría de los programadores!). Observe el uso de la doble barra invertida **'\\'** para denotar correctamente el *path* del árbol del directorio: usar únicamente **'\'** no basta pues representa el caracter escape (vea constantes de barra invertida en §1.4).

Es *sumamente peligroso* usar apuntadores nulos: puede ocasionar fácilmente la caída del sistema. Por ello antes de usar un *apuntador a archivo* debe verificarse que **NO** sea nulo. Por lo tanto se debe cambiar la declaración del ejemplo anterior a:

Ejemplo 1.36+1

```
#define PATH "A:\\PROGRAMA\\prueba.txt"

typedef enum {CORRECTO, ERROR} status;

status abre_archivo(void)
     {
     FILE *fp;
     char archivo[]=PATH;
          . . .
      if((fp=fopen(archivo,"r")==NULL)
            {
                  printf("Imposible abrir: %s\n",archivo);
                  return(ERROR);
            }
            else return(CORRECTO);
     }
```

• Las funciones fprintf( ) y fscanf( )

Estas dos funciones trabajan de manera similar que las correspondientes **printf( )** y **scanf( )**, excepto que en vez de hacerlo con la entrada y salida estándar, lo hacen a través de archivos previamente abiertos en modo escritura o lectura respectivamente.

Sintaxis

```
fprintf(fp,"cadena de control",argumentos);

fscanf(fp,"cadena de control",&argumentos);
```

fp es un apuntador a archivo asignado por **fopen( )**.

Si se desea que en lugar de que la información sea enviada a disco, lo haga a impresora, debe sustituirse el nombre de archivo por **"prn"** al abrirlo con **fopen( )** en modo **"w"**. De manera similar, **"con"** proporciona el acceso de datos desde la consola.

• Macros putc( ) y getc( )

Sintaxis

```
putc(carácter,apuntador);

getc( )
```

La macro **putc( )** permite escribir un carácter en un archivo que haya sido abierto con el modo **"w"**. **getc( )** es usada para leer caracteres de un archivo abierto en el modo **"r"**.

Sintaxis

```
c = getc(apuntador_a_archivo);
```

El carácter se almacena en la variable **c**.

• Función fclose( )

Permite cerrar un *stream* que ya no se va a continuar utilizando.

Sintaxis

```
fclose(apuntador_a_archivo);
```

Si no se cierra un *stream* en modo **"w"** o **"w+"** adecuadamente, se corre el riesgo de que no se escriban al archivo los últimos datos enviados, pues no hay garantía de que se vacíe la memoria temporal (buffer) que almacenaba los datos.

§1.11 GRAFICAS EN C

• Modos de texto y gráfico

El manejo de gráficas no se definió en el estándar **ANSI** porque la diversidad de hardware existente no permite uniformizar criterios. Por lo tanto ninguna de las funciones discutidas a continuación corresponde al estándar **ANSI**. En lugar de ello se hará una reseña de las funciones gráficas que maneja el compilador *Turbo **C*** y se aplicarán exclusivamente para las generación de gráficas de ecuaciones. En el *modo gráfico* se puede manipular una tarjeta de video para generar imágenes de objetos de alta calidad a través de los prototipos y macros contenidas en el archivo de biblioteca **`<graphics.h>`** de **Turbo *C***.

• Texto con ventanas

Algunas funciones específicas de *Turbo **C*** que permiten manipular la consola como **`conio()`**, **`getche()`** y **`gotoxy()`** se hallan definidas en **`<conio.h>`.** A continuación se explicará cada una de ellas.

• Función `clrscr( )`

Sintaxis

```
clrscr();
```

Propósito: Limpia la ventana de texto que es toda la pantalla por defecto.

• Función `gotoxy(int, int)`

Sintaxis

```
gotoxy(x,y);
```

Propósito: Posiciona el cursor en la columna **x**, renglón **y**. Tanto **x** como **y** deben ser variables o constantes **int**.

• Función getche( )

Sintaxis

```
< simbolo = > getche( );
```

Propósito: Lee un carácter con eco a la pantalla. Dicho carácter se lee automáticamente y se puede asignar a cualquier variable de tipo **`char`**. **`getche( )`** permite suspender la ejecución de un programa hasta que se presione una tecla.

La tabla 1.12 muestra algunas otras funciones importantes de este archivo de encabezado.

Tabla 1.12 Funciones de manejo de consola

`cprintf( )`	**Escribe salida formateada en la ventana.**
`getche( )`	**Lee un carácter de la ventana.**
`cgets( )`	**Lee una cadena de la ventana.**
`clrscr( )`	**Limpia la ventana.**
`clreol( )`	**Borra desde cursor hasta final de la línea**
`gotoxy( )`	**Posiciona el cursor.**

Por supuesto, los atributos de texto pueden modificarse con algunas funciones, lo que permite tener presentaciones en alta o baja intensidad y el cambio de los colores de fondo y primer plano.

Tabla 1.13 Atributos del texto

Función *C*	
`highvideo( )`	Alta intensidad
`lowvideo( )`	Baja intensidad
`textbackground( )`	Color del fondo
`textcolor( )`	Color del texto

Tabla 1.14 Macros de color

`BLACK`	`0`	`LIGHTBLUE`	`9`
`BLUE`	`1`	`LIGHTGREEN`	`10`
`GREEN`	`2`	`LIGHTCYAN`	`11`
`CYAN`	`3`	`LIGHTRED`	`12`
`RED`	`4`	`LIGHTMAGENTA`	`13`
`MAGENTA`	`5`	`YELLOW`	`14`
`BROWN`	`6`	`WHITE`	`15`
`LIGHTGRAY`	`7`	`BLINK`	`128`
`DARKGRAY`	`8`		

Ejemplo 1.37+1

```
#include <conio.h>

. . .
highvideo( );

textcolor(GREEN|BLINK);
. . .
```

Estas dos funciones permiten que se trabaje en video en alta intensidad, con letras en color verde y parpadeando.

• Gráficos

La computadora debe poseer un adaptador gráfico de video tipo VGA, CGA o EGA para poder realizar despliegues gráficos. Cada uno de ellos posee una resolución en pixeles diferente dependiendo del modo en el que se active.

Tabla 1.15 Controladores gráficos

Controlador	Modo	Resolución en pixeles (x,y)
CGA	CGAC0	320 x 200
	CGAC1	320 x 200
	CGAC2	320 x 200
	CGAC3	320 x 200
	CGAHI	640 x 200
VGA	VGALO	640 x 200
	VGAMED	640 x 350
	VGAHI	640 x 480

Tabla 1.16 Tipos de resolución.

X max	Y max	Tarjeta gráfica
640	200	CGA

640	480	VGA
640	200	EGA
720	348	Hércules

• Funciones de control de video

Antes de intentar usar funciones gráficas, debe inicializarse la tarjeta de video en un *modo gráfico* usando **initgraph(char *controlador, char *ruta)**. Esta función carga en memoria el controlador gráfico apuntado por ***controlador**. La **ruta** especifica el camino a través del árbol de directorios donde está el controlador especificado en un archivo **.BGI**. **<Graphics.h>** tiene varias macros a usar para esta función:

Tabla 1.17 Macros de video

Macro	Adaptador	Equivalente
DETECT	Autodetección	0
CGA	CGA	1
EGA	EGA	3
HERCMONO	Hércules	7
VGA	VGA	9

DETECT se encarga de detectar automáticamente el hardware del que se disponga.

El programa del listado 1.5 se encarga de realizar las primitivas gráficas como inicialización del video, creación de ventanas lógicas, físicas y construcción de puntos, líneas, círculos y posición de textos que son necesarios en programas para crear gráficas de ecuaciones. El controlador gráfico usado será el **VGA** y el modo el **VGAHI** que corresponde a alta resolución. La *ventana lógica* es el rango de datos en las variables *x* y *y* donde se encuentran los valores extremos de cierto objeto en el plano. Pensando en una gráfica de ecuaciones en R^2, corresponde al intervalo de graficación en *x*, y a los máximos y mínimos que toma la función evaluada. La *ventana física* es un área del monitor de la computadora que se usará para representar la salida visual. Es evidente que la ventana lógica puede abarcar números reales positivos y negativos. La ventana física, por otra parte se mide a través de unidades enteras llamadas pixeles. Existe una relación matemática entre ambas que permite "mapear" los cálculos de la ventana lógica en la física. Vea la figura 1.1.

Estudie el código detenidamente y vea después el capítulo siguiente, raíces de ecuaciones para entender su funcionamiento.

Listado 1.5. Fun_graf.h

```
#ifndef __FUN_GRAF_H
#define __FUN_GRAF_H
#include <graphics.h>
#define PATH ""       /* Ruta al driver .BGI */

real W1, W3, W2, W4;  /* ventana lógica */
int  V1, V2, V3, V4;  /* ventana física */
real U1, U2;

int Manejador, Modo=VGAHI, codigo;     /* Manejadores de video */


/* La ventana fisica indica las coordenadas de la pantalla */
void ventana_fisica(int x1, int y1, int x2, int y2)
{
  V1=x1;    /*                 */
  V4=y2;    /* (V1,V3)         */
  V2=x2;    /*                 */
  V3=y1;    /*         (V2,V4) */
  graphdefaults();
  setviewport(V1,V3,V2,V4,1);
}

/* La ventana logica indica las coordenadas reales del objeto */
void ventana_logica(real x1, real y1, real x2, real y2)
{
  W1=x1;     /*            (W2,W4) */
  W3=y1;     /*                    */
  W2=x2;     /*                    */
  W4=y2;     /* (W1,W3)            */

  U1=(V2-V1)/(W2-W1); /* FISICO/LOGICO */
  U2=(V4-V3)/(W4-W3);
}


/* Inicializa el modo gráfico en VGA */
status inicializa(int x1, int y1, int x2, int y2)
{
  detectgraph(&Manejador, &Modo);/* Detecta el hardware */
  if(Manejador != VGA)
     return ERROR;
  initgraph(&Manejador, &Modo, PATH);
  codigo = graphresult();
  if(codigo != grOk)
     return ERROR;
  ventana_fisica(x1,y1,x2,y2);
```

```
  return BIEN;
}

/* Transforma coordenadas */
void transforma(real x, real y, int *a, int *b)
{
  *a =           (x-W1)*U1;
  *b = (V4-V3) - (y-W3)*U2;
}


/* Escribe un mensaje de texto */
void letrero(real x, real y, char *mensaje, int color, int DIR)
{
  int a,b;
  settextstyle(DEFAULT_FONT,DIR,1);
  setcolor(color);
  transforma(x, y, &a, &b);
  outtextxy(a, b, mensaje);
}

/* Traza un cuadro de (x1,y1)-(x2,y2) con color */
void cuadro(real x1, real y1, real x2, real y2, int color)
{
  int a1,b1,a2,b2;
  setcolor(color);
  transforma(x1,y1, &a1, &b1);
  transforma(x2,y2, &a2, &b2);
  rectangle(a1,b1,a2,b2);
  setfillstyle(SOLID_FILL, color);
  transforma( x1 + (x2-x1)/2.0, y1 + (y2-y1)/2.0, &a1, &b1);
  floodfill(a1, b1, color);
}

/* Traza un rectángulo de (x1,y1)-(x2,y2) con color */
void rectángulo(real x1, real y1, real x2, real y2, int color)
{
  int a1,b1,a2,b2;
  setcolor(color);
  transforma(x1,y1, &a1, &b1);
  transforma(x2,y2, &a2, &b2);
  rectangle(a1,b1,a2,b2);
}

/* Traza una línea de (x0,y0)-(x1,y1) */
void linea(real x0, real y0, real x1, real y1, int color )
{
   int xi, xf, yi, yf;
   setcolor(color);
```

```
    transforma(x0, y0, &xi, &yi);
    transforma(x1, y1, &xf, &yf);
    line(xi, yi, xf, yf);
}

#endif
```

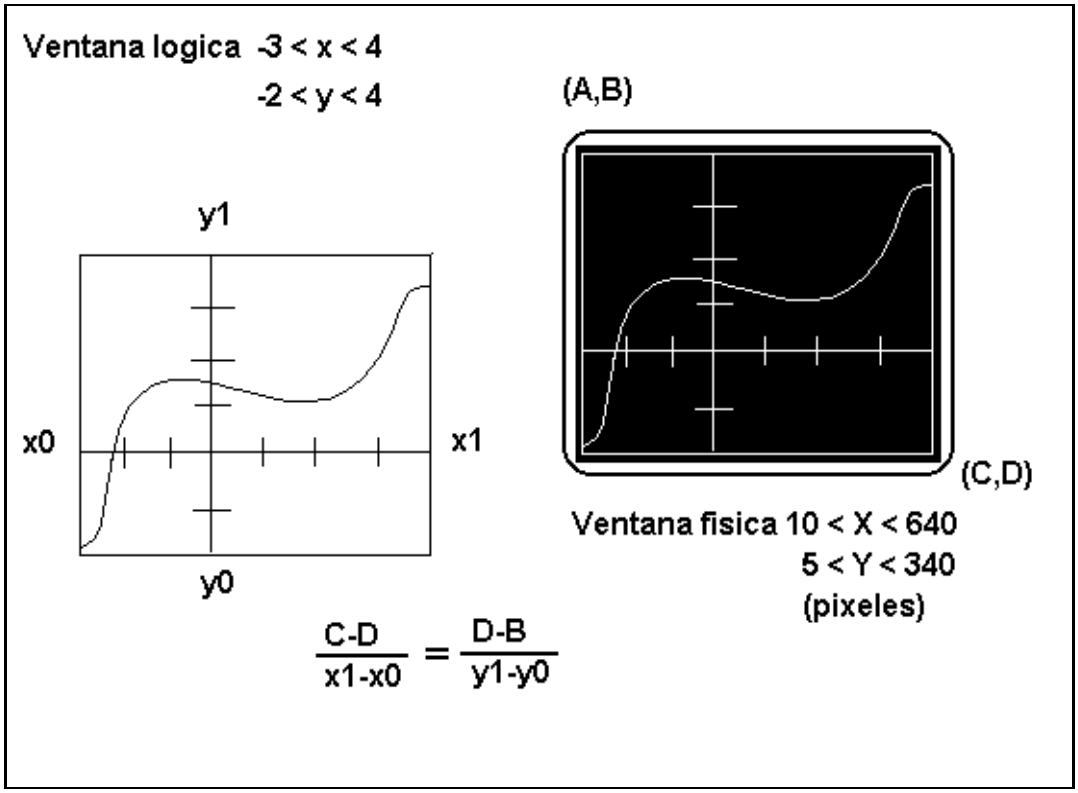


Fig. 1.1 Ventanas física y lógica

Por supuesto, generar gráficas complejas requiere de ciertos conocimientos de geometría analítica para construir funciones de transformación, recorte, rotación y otras. Además hay que considerar que el estándar **ANSI** no define ninguna función gráfica, por esa razón en el texto únicamente se usarán el modo gráfico para obtener gráficas de funciones *f(x)* con el fin de evaluar cualitativamente el comportamiento de *f* y de sus raíces. Las funciones para graficar curvas del tipo *f(x)=0* serán estudiadas en el siguiente capítulo.

Un bello ejemplo de lo que puede hacerse con gráficas por computadora se muestra en la figura 1.2, que muestra una imagen fractal conocida como conjunto de Mandelbrot. Las imágenes fractales poseen entre otras características la autosimilitud, es decir, dentro de una pequeña parte de la imagen se puede

encontrar toda la imagen misma.

Para el lector interesado en el tema de gráficas en ***Turbo C/C++***, una excelente referencia la puede hallar en *Ezzel, [1990]*, en la que se programa en ***C*** y en objetos con ***C++***, además de estudiar la interfase con el ratón (*mouse*), la salida a impresora y un capítulo dedicado a imágenes fractales.

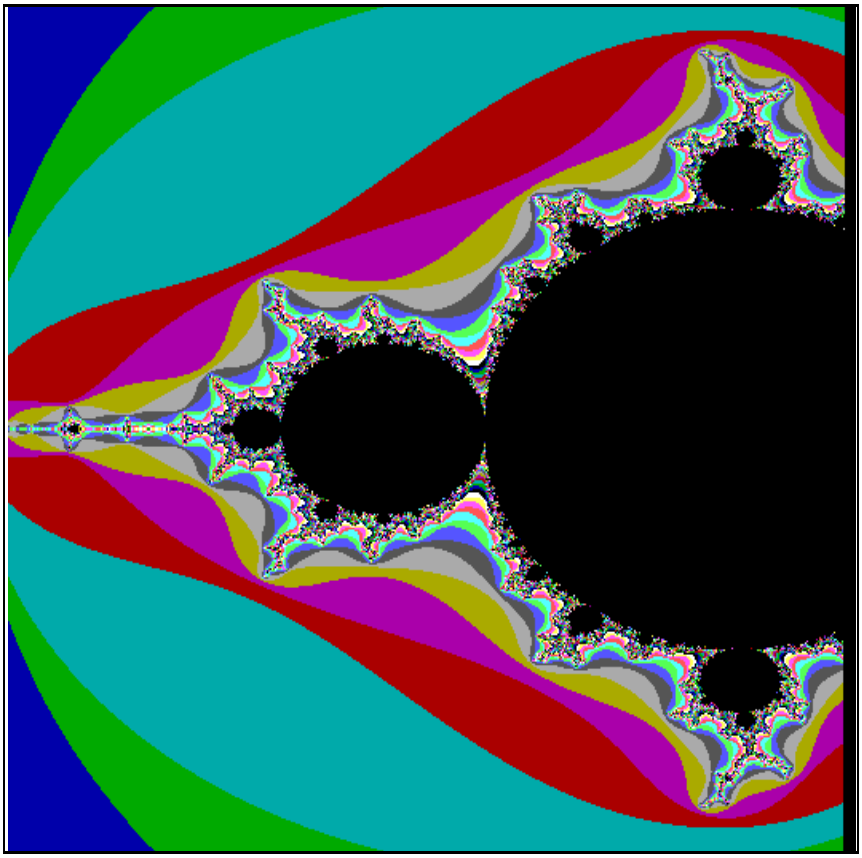


Fig. 1.2 Conjunto de Mandelbrot