INTRODUCCIÓN A LA PROGRAMACIÓN ESTRUCTURADA Y AL LENGUAJE C

ING. J. FERNANDO SOLÓRZANO PALOMARES

FEBRERO DE 1995

VOLUMEN II

METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA

PRÓLOGO

DEBIDO A LOS CAMBIOS EN LOS PLANES Y PROGRAMAS DE LAS CARRERAS DE INGENIERÍA Y A LA NECESIDAD DE ACTUALIZAR A LOS ALUMNOS, SOBRE TODO EN LAS BASES DE UNA CULTURA INFORMÁTICA Y LOS ELEMENTOS DE LA PROGRAMACIÓN ESTRUCTURADA QUE LES PERMITAN UTILIZAR A LA COMPUTADORA COMO HERRAMIENTA DE APOYO PARA CUALQUIER ÁREA DE LA INGENIERÍA, SE CONSIDERÓ CONVENIENTE ELABORAR ESTAS NOTAS PARA AYUDAR A LA TRANSICIÓN DE PROGRAMAS Y CONTAR CON EL MATERIAL DE APOYO PARA EL NUEVO PROGRAMA DE LA ASIGNATURA DE COMPUTADORAS Y PROGRAMACIÓN.

SE DIVIDIÓ EL CONTENIDO EN DOS GRANDES PARTES, LA PRIMERA COMPRENDE UNA INTRODUCCIÓN A LA COMPUTACIÓN Y LA SEGUNDA INCLUYE LA PARTE BÁSICA DE LA METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA, DE TAL FORMA QUE SE CUBRE LO MÁS IMPORTANTE DE LOS REQUERIMIENTOS DE ENSEÑANZA DEL PROGRAMA ANTERIOR DE LA ASIGNATURA Y DEL NUEVO; LA SEGUNDA PARTE INCLUYE EXCLUSIVAMENTE LO BÁSICO DE LOS LENGUAJES DE PROGRAMACIÓN QUICK BASIC Y LENGUAJE C. EN AMBOS PROGRAMAS SE INDICA ADEMÁS ENSEÑAR EN FORMA ELEMENTAL UN PROCESADOR DE TEXTOS, UNA BASE DE DATOS Y UNA HOJA ELECTRÓNICA, SIN EMBARGO SE CONSIDERÓ CONVENIENTE PARA ESA PARTE TRABAJAR FASCÍCULO SEPARADOS DE ESTE TEXTO.

PARA EL DESARROLLO DE LA TEORÍA SE DA LA INFORMACIÓN MEDIANTE DESARROLLOS FORMALES BREVES Y DIRECTOS CONSIDERANDO EJEMPLOS Y APLICACIONES EN GENERAL CON LA IDEA DE UNAS NOTAS QUE DESTACAN LO MÁS RELEVANTE, PERO NO EXTENSAS O CON DETALLES PORMENORIZADOS, POR TAL MOTIVO SE RECOMIENDA CONSULTAR TEXTOS ESPECIALIZADOS QUE COMPLEMENTEN LA CURIOSIDAD DEL ESTUDIANTE.

EL CONTENIDO DEL TEXTO Y REDACCIÓN ES RESPONSABILIDAD DEL ING. FERNANDO SOLÓRZANO PALOMARES; LA REVISIÓN CORRESPONDIÓ AL ING. JOEL VILLAVICENCIO CISNEROS, QUIEN APORTÓ EJEMPLOS Y SUGERENCIAS PARA UNA MEJOR COMPRENSIÓN.

SE AGRADECE LA REVISIÓN Y ADAPTACIÓN PEDAGÓGICA DE LAS LICENCIADAS MARÍA CUAIRÁN RUIDÍAZ Y ANDREA AYALA HERNÁNDEZ. ASIMISMO SE AGRADECEN LA LECTURA, COMENTARIOS Y SUGERENCIAS DE LAS SRITAS. LETICIA MARAVILLA FRANCO, PILAR ALBARRAN MIER Y DEL SR. JORGE CALDERÓN PÉREZ.

EL AUTOR.

INDICE

INTRODUCCIÓN 153

7 DIAGRAMAS DE FLUJO 156

7.1 Conceptos Generales 157

7.1.1 Concepto de programas: fuente, objeto y ejecutable 157

7.1.2 Algoritmo no numérico y algoritmo numérico 158

7.2 Elementos de un lenguaje de programación 159

7.2.1 Operadores 159

7.2.2 Declaraciones 160

7.2.3 Constantes 160

7.2.4 Variables 162

7.2.5 Funciones de biblioteca 163

7.3 Simbología básica de diagramas de flujo 166

7.4 Diagramación tradicional (Diagramación a partir de estructuras básicas) 166

7.5 El ciclo iterativo 169

7.6 Instrucción de reemplazo o concepto de almacén 172

7.7 Aplicaciones del ciclo iterativo 172

7.7.1 Primera aplicación: el concepto de contador 172

7.7.2 Segunda aplicación: el concepto de sumador (sumatorias) 174

7.7.3 Tercera aplicación: el concepto de multiplicación reiterada (factoriales) 177

7.8 Diagramación compuesta o estructurada 183

7.8.1 Ciclo iterativo controlado por un contador 184
7.8.2 Ciclos anidados (con contador incluido) 191
7.8.3 Ciclos iterativos controlados por condición 198
7.8.3.1 Ciclo con condición al inicio (ciclo **WHILE** o mientras) . 198
7.8.3.2 Ciclos con condición **WHILE** o **UNTIL** 202

7.8.4 Diagramas estructurados de selección (ramificaciones o transferencias) 213
7.8.4.1 Diagrama estructurado **IF - THEN - ELSE** 213
7.8.4.2 Elemento estructurado de ramificación encadenado **IF - THEN - ELSE IF - THEN - ELSE IF ... ELSE** 215
7.8.4.3 Selección de casos específicos 217
7.9 Subprogramas 221
7.9.1 Funciones 223
7.9.2 Subrutinas 228

8 **PSEUDOCÓDIGO 231**

8.1 Características generales del pseudocódigo 232
8.2 Elementos estructurados de repetición (ciclos) 233
8.2.1 Ciclo controlado por contador (ciclo desde o para) 233
8.2.1.1 Ciclos anidados (con contador incluido) 234
8.2.2 Ciclo controlado por condición al inicio: **WHILE** 235
8.3 Ciclos repetir 236
8.3.1 Primera opción **(DO WHILE LOOP o REPETIR MIENTRAS_QUE)** 237
8.3.2 Segunda opción **(DO UNTIL - LOOP o REPETIR HASTA_QUE** 239
8.3.3 Tercera opción **(DO - LOOP WHILE)** . . . 241
8.3.4 Cuarta opción **(DO - LOOP UNTIL)** 243
8.4 Elementos estructurados de selección 245
8.4.1 Elemento estructurado **IF THEN - ELSE** (si_entonces - en_caso_contrario) 245
8.4.2 Elemento estructurado de selección encadenado **IF THEN - ELSE IF - ELSE IF - ELSE IF - ELSE** . . 247
8.4.3 Elemento estructurado para seleccionar casos específicos 251
8.5 Subprogramas 259
8.5.1 Funciones 260
8.5.2 Subrutinas 264

9 **EL LENGUAJE DE PROGRAMACIÓN C 269**

9.1 Configuración de un programa 269
9.1.1 Declaraciones 271
9.1.2 Instrucciones ejecutables 271
9.2 Directivas de preprocesamiento 278
9.3 Tipos de declaraciones de variables y su ámbito de acción en el diseño estructurado de programas.
9.4 Elementos de programación del lenguaje C 282

9.4.1 Constantes de C 283
9.4.1.1 Constantes enteras 283
9.4.1.2 Constantes reales 283
9.4.1.3 Constantes alfanuméricas 283
9.4.2 Variables 284
9.4.2.1 Variables enteras 284
9.4.2.2 Variables reales 284
9.4.2.3 Variables alfanuméricas o de cadena 285
9.4.2.4 Variables con índice 285
9.4.2.5 Parámetros asociados a las variables 286
9.5 Funciones de biblioteca 287
9.6 Codificación 288
9.7 Operadores abreviados típicos del lenguaje C . . . 291
9.8 Entrada - Salida 294
9.9 Pregunta lógica **if - else** sencilla 310
9.10 El operador ? como pregunta **if - else** simple 313
9.11 Elemento estructurado **for** 316
9.11.1 Ciclos iterativos **for** anidados 324
9.12 Ciclos iterativos controlados por condición . 329
9.12.1 Ciclo con condición al inicio del ciclo: elemento estructurado **while** 329
9.12.2 Ciclo **do** con opción **while** 338
9.12.2.1 Ciclo infinito: elemento estructurado **do - while(1)** 343
9.13 Elemento estructurado **if else** compuesto (ramificaciones o transferencias) 345
9.14 Elemento estructurado de ramificación encadenado o anidado **if - else if - else if ... else** 348
9.15 Ramificación múltiple o selección de casos específicos 357
9.16 Apuntadores 366
9.17 Enumeraciones 381
9.18 Estructuras 382
9.19 Uniones 386
Apéndice 1: Algoritmos numéricos escritos en C 393
Apéndice 2: Introducción al ambiente gráfico de C . . . 417

INTRODUCCIÓN

Estos apuntes están divididos en dos grandes áreas: el primer volumen refiere un panorama del desarrollo de las computadoras y lenguajes; el segundo volumen presenta lo más importante de la metodología de la programación estructurada, que proporciona al estudiante de ingeniería en general, los elementos para emplear la computadora como herramienta en la obtención de la solución de algoritmos numéricos.

La metodología de la programación estructurada es el conjunto de técnicas o elementos que nos permiten plantear, probar y depurar un algoritmo para por último obtener la solución mediante un lenguaje de alto nivel procesado con la moderna computadora digital. En este segundo volumen se desarrolla la metodología de la programación estructurada bajo un enfoque general a todas las áreas de la ingeniería, haciendo énfasis en los elementos de programación estructurados que permiten un más fácil desarrollo de algoritmos para la computadora. Se pretende pues dar los conocimientos básicos para todo tipo de estudiante de ingeniería, partiendo prácticamente desde cero e intentando reunir en estos textos lo más relevante del programa actual de la asignatura de tal manera que el estudiante cuente en un solo texto lo que normalmente se presenta en diversos libros. Se intenta dar las bases necesarias para el estudiante de ingeniería no especializado, de esa manera se presenta una herramienta de uso general que permita al estudiante aplicar la programación en cualquier área de la ingeniería y la mayoría de sus asignaturas; Con estos textos se considera que el estudiante contará con las bases para proseguir con temas especializados. Esta segunda parte del curso se dedica a lo más importante del programa de la asignatura, es decir a la programación. Para la elaboración también se consideró la heterogeneidad de los alumnos y que una muy buena parte de ellos no traen antecedentes de programación, por lo cual se exponen los conceptos más importantes paso a paso. La metodología comprende las herramientas básicas que permiten la formulación, depuración y documentación formal de algoritmos aplicables a las distintas áreas de las carreras de ingeniería. Todos los temas incluyen al inicio una tabla o cuadro sinóptico cuya finalidad es la de permitir al estudiante recapitular sobre los conocimientos que deberán dominarse al finalizar el tema respectivo.

El primer elemento de la metodología es el tema de diagramas de flujo que comienza con la diagramación tradicional con el fin de que el alumno comprenda la simbología básica y cómo ha evolucionado hasta una simbología estructurada. Se considera a este tema como un lenguaje formal teóricamente independiente de los lenguajes de alto nivel de las computadoras, que permite expresar de forma gráfica la solución de problemas relacionados con la ingeniería. Por tal motivo se describen también los elementos comunes a los lenguaje de programación, el manejo de la aritmética de tipo entero y de tipo

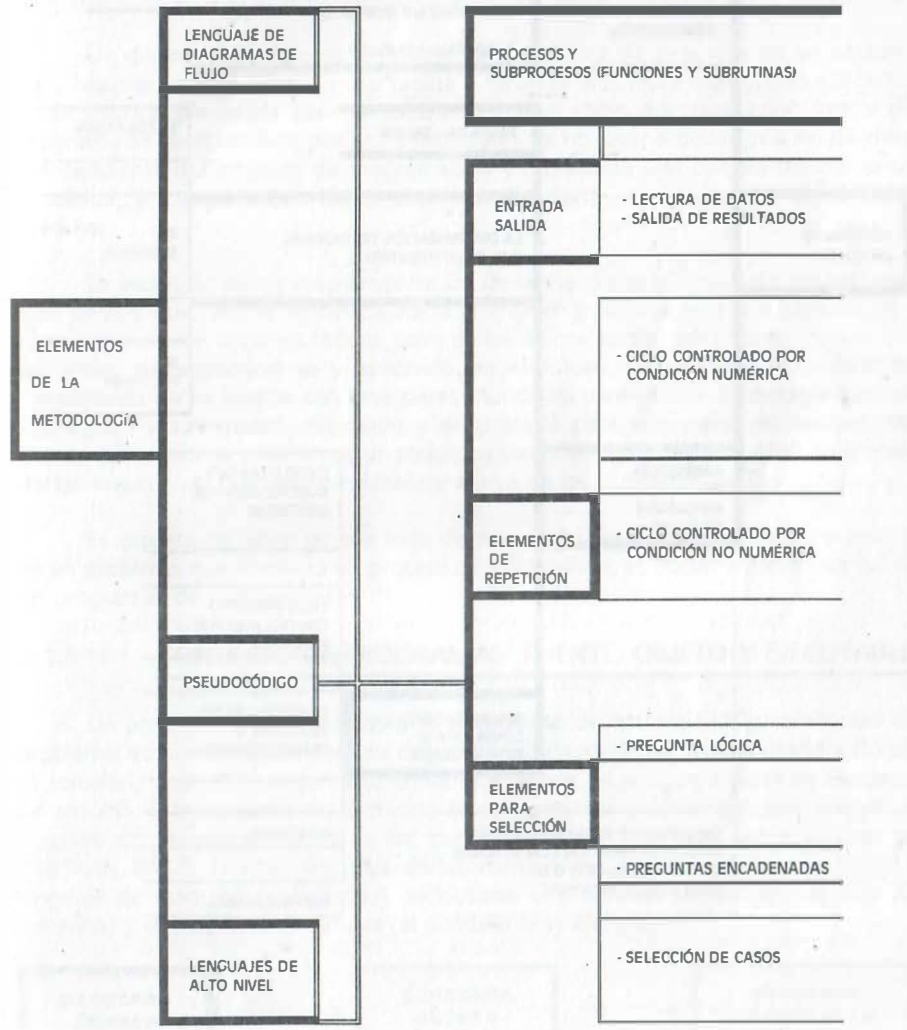
real. A continuación se describe paso a paso el funcionamiento de los elementos de repetición (ciclos iterativos), su uso, aplicaciones para generar contadores y sumatorias, así como los distintos tipos de ciclos que se emplean en la programación estructurada. A continuación se describen los elementos de selección: preguntas, preguntas encadenadas y selección de casos específicos. Por último se da la simbología para los subprocesos (funciones y subrutinas), indicando sus principales características, aplicaciones y ejemplos de uso.

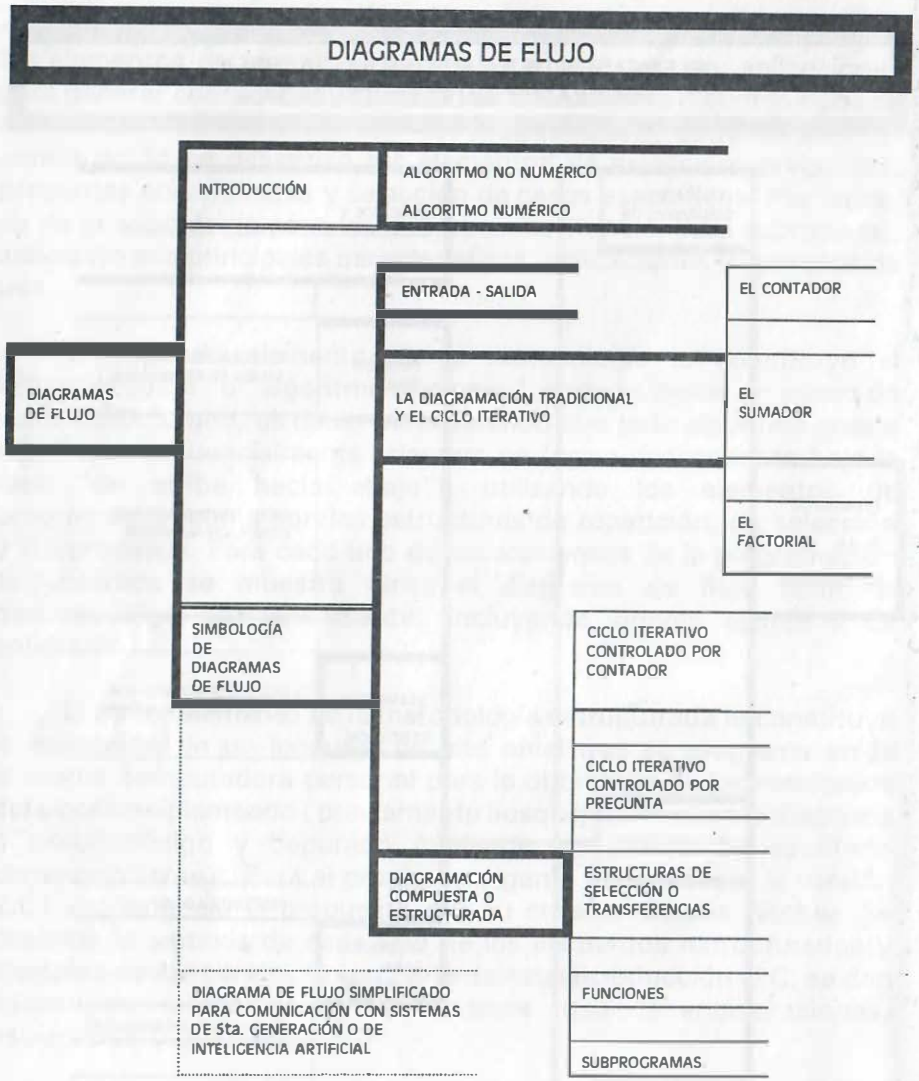
El segundo elemento de la metodología lo constituye el "Pseudocódigo" o "algoritmo platicado", tratado desde un punto de vista estructurado, es decir, considerando que todo algoritmo puede resolverse secuencialmente (siempre en forma descendente bajo la idea "de arriba hacia abajo"), utilizando los elementos de programación como son las estructuras de repetición, de selección y subprocesos. Para cada uno de los elementos de la programación estructurada se muestra tanto el diagrama de flujo como el pseudocódigo correspondiente, incluyendo breves ejemplos de aplicación.

El tercer elemento de la metodología estructurada lo constituye la aplicación de un lenguaje de alto nivel que se programa en la moderna computadora personal para la obtención de los resultados del algoritmo planteado (previamente bosquejado mediante diagrama o pseudocódigo y depurado mediante su prueba de escritorio correspondiente). Para el programa vigente se consideró la versión 2.01 del lenguaje C propuesta por su creador Dennis Ritchie. Se describe la sintaxis de cada uno de los elementos estructurados y ejemplos de aplicación. Como parte de esta introducción al C, se dan bases para el manejo de apuntadores, uso de enumeraciones, estructuras y uniones.

Por último se incluye dos apéndices, uno con ejemplos de algoritmos numéricos procesados en la versión antes indicada de C (misma que puede instalarse en computadoras de escasos recursos), otro con una introducción al ambiente gráfico de C con diversos ejemplos y sus resultados.

ELEMENTOS DE LA METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA





7 DIAGRAMAS DE FLUJO

7.1 CONCEPTOS GENERALES

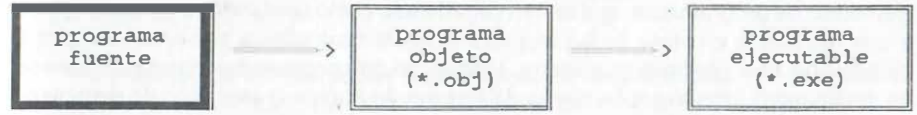
Un diagrama de flujo es la representación gráfica de cada una de las acciones para resolver un problema. Su uso facilita a terceros la comprensión y/o modificación de la solución planteada para un problema y sirve como documentación formal del programa de computadora que se ha elaborado. Es sin lugar a dudas una ayuda visual independiente del lenguaje de programación y representa una posible solución a un problema, en términos de lógica expresada mediante una secuencia de símbolos gráficos.

La evolución de los diagramas ha ido de la mano con el desarrollo del software y es posible que, con la tendencia hacia interfases gráficas y lenguaje hablado para comunicarnos con la computadora, sean de los únicos medios para comunicación que continúen perfeccionándose y aplicando en el futuro. A pesar de todo lo antes mencionado no se cuenta con estándares mundiales para utilizar simbología que, sin embargo, ha aumentado, mejorado y simplificado para convertirse en herramienta formal que explica la solución de un problema dado. Recientemente la ANSI ha emitido ciertas reglas cuyo objetivo es normalizar el uso de los diagramas de flujo.

Es importante remarcar que todo diagrama de flujo obtenido a partir del análisis de un problema que involucra un proceso de información, es documentación formal de los programas de computadora.

7.1.1 CONCEPTO DE PROGRAMAS: FUENTE, OBJETO Y EJECUTABLE

Un programa de computadora es el conjunto de instrucciones para resolver un problema, escritas en algún lenguaje de programación, para que la computadora calcule su solución, tomando en cuenta todas las alternativas de proceso a partir de los datos de entrada. Este conjunto de instrucciones se denomina programa fuente cuando el lenguaje de programación es de los denominados de alto nivel como el caso de FORTRAN, BASIC, PASCAL, etc. Cuando las instrucciones anteriores se han traducido a lenguaje de máquina (compiladas), se obtiene el programa objeto (en lenguaje de máquina) y el programa ejecutable (al nombrarse se ejecuta).



7.1.2 ALGORITMO NO NUMÉRICO Y ALGORITMO NUMÉRICO

Un algoritmo no numérico es el conjunto de instrucciones ordenadas, que indican paso a paso, cómo obtener el resultado de un problema que no involucra procesos matemáticos. Por ejemplo, las instrucciones para cocinar alimentos.

```
1.  Inicio
2.  Adquirir los ingredientes del platillo que se va a cocinar
3.  Seguir los pasos indicados en la receta
4.  Fin
```

Observemos que un algoritmo no numérico puede hacerse tan detallado o general como se requiera para obtener el resultado deseado. Para ejemplificar lo anterior sería necesario seleccionar un platillo en particular, sus ingredientes y cantidades, así como cada uno de los pasos para cocinar.

Un algoritmo numérico es el conjunto de instrucciones ordenadas para resolver un problema que involucra procesos matemáticos (con cálculo de fórmulas de manera repetida). Este tipo de algoritmos no admite ambigüedades y debe darse cada uno de los pasos para su solución. Este tipo de algoritmos se analizarán con detalle a continuación.

ANTECEDENTES

Aunque la idea de solución de un problema paso a paso mediante símbolos gráficos se hace pensando en no particularizar en algún lenguaje de programación, es necesario conocer la interpretación y uso de algunos elementos de programación que en su definición de comportamiento, se diferencian de lo que podría entenderse desde el punto de vista algebraico o matemático. Además, dado que por lo general, el diagrama de flujo debe ser escrito posteriormente en algún lenguaje de programación, resulta útil conocer algunas reglas de funcionamiento lógico y de interpretación comunes a la mayoría de los lenguajes de alto nivel, con el fin de manejar adecuadamente la prioridad de operaciones y agrupamiento en las instrucciones ejecutables por la computadora.

Es conveniente incluir como antecedente al tema de diagramas de flujo, aquellos elementos de programación que se han considerado como comunes a algunos lenguajes y que facilitan la escritura de las expresiones aritméticas que se presentan durante la elaboración de algoritmos numéricos; además, se ha considerado importante conocer de antemano la interpretación que se da en programación a la expresión de reemplazo, a fin de facilitar también su correcta interpretación en la solución de algún problema;

veamos lo mínimo sobre los elementos de programación comunes a un buen número de lenguajes de programación y por último los diagramas de flujo estructurados más comunes a los lenguajes de alto nivel.

NOTA IMPORTANTE:

Por razones didácticas, en estas notas se hará un enfoque de diagramas de flujo orientado principalmente hacia el lenguaje BASIC y su equivalente en lenguaje C, que involucre algoritmos numéricos.

7.2 ELEMENTOS DE UN LENGUAJE DE PROGRAMACIÓN

Los elementos esenciales de todo lenguaje de programación son:

OPERADORES
DECLARACIONES
CONSTANTES
VARIABLES
FUNCIONES DE BIBLIOTECA

Es mediante ellos, y algunos otros elementos de programación básicos, que podemos definir correctamente un buen número de instrucciones o pasos para la solución de un problema con la computadora, desde la entrada de información, hasta la salida de resultados.

7.2.1 OPERADORES

Los operadores básicos son los siguientes:

+	SUMA
-	RESTA
-	NEGACIÓN
MOD	RESIDUO DE DIVISIÓN ENTERA
*	MULTIPLICACIÓN
\	DIVISIÓN ENTERA (EN C SE USA EL CARACTER %)
/	DIVISIÓN
^	EXPONENCIACIÓN (EN C SE USA UNA FUNCIÓN)

La prioridad de los operadores en casi todos los lenguajes para la evaluación de las expresiones aritméticas, involucrando funciones del programa y de biblioteca, de izquierda a derecha es la siguiente:

FUNCIONES
EXPONENCIACIÓN
MULTIPLICACIÓN O DIVISIÓN REALES
DIVISIÓN ENTERA
MOD
SUMA O RESTA

Los paréntesis redondos se utilizan como elementos de agrupamiento y siempre que existe un par o más de dichos paréntesis, el contenido de éstos se evalúa primero, desde el más interno, hasta el más externo. Un ejemplo de aplicación es cuando en una expresión existe un numerador y un denominador que constan de más de un término.

7.2.2 DECLARACIONES

Las instrucciones de un programa se dividen en ejecutables y no ejecutables. Estas últimas (las declaraciones no ejecutables) proveen a la computadora la información necesaria sobre las características de las variables, especificaciones para formatos de salida, etc.

7.2.3 CONSTANTES

Los tipos de constantes se dividen en:

ENTERAS
REALES
ALFANUMÉRICAS

A su vez, dentro de los dos primeros grupos, las constantes pueden catalogarse en:

DE PRECISIÓN SENCILLA
DE DOBLE PRECISIÓN

DEFINICIONES:

Constante entera. Cantidad que carece de punto y dígitos decimales, sólo admite signos (positivo o negativo).

Constante real. Cantidad que incluye punto y dígitos decimales, sólo admite signos (positivo o negativo).

Por lo general una constante entera o real de precisión sencilla se almacena utilizando el mínimo de palabras de memoria posible, esto lo fija la tecnología de la

marca y modelo de computadora que se va a usar.

Las constantes reales permiten representar algunos números racionales o cantidades con fracción definida exactamente, en razón de las limitaciones de su memoria y de la forma en que se almacenan. Curiosamente una fracción real se almacena en forma parecida a una cantidad representada en notación exponencial, es decir, hay una parte entera y un exponente. La precisión simple de las cantidades reales garantiza por lo general seis dígitos decimales exactos, la doble precisión, el doble más o menos.

Una computadora almacena datos en su memoria primaria utilizando dígitos binarios (cero o uno) agrupados en bytes para formar con esto, a su vez, palabras de computadora. De esta manera, una constante entera requiere de uno o más bytes para la representación de su valor. Es así que la cantidad puede imaginarse representada por un conjunto de focos que presentan individualmente uno de dos estados prendido o apagado, es decir, uno o cero. De ese grupo de bytes la primera posición de la izquierda está reservada para indicar si la cantidad es negativa (valor unitario) o positiva (valor nulo), los demás bits indican un valor posicional:

De esta característica resulta que la cantidad entera más grande permitida por una computadora depende del número de bits que puede utilizar para su representación y que se calcula con la expresión:

$$2^{(n-1)} - 1$$

en la cual *n* significa el número de bits empleados por la computadora para representación de cantidades enteras.

Por ejemplo, si una cantidad entera se representa mediante 16 bits, la cantidad máxima que podrá representarse es de $2^{(16-1)} - 1 = 32,767$

Las constantes enteras se utilizan en la programación para definir contadores, índices de arreglos, valores máximos en cuanto número de elementos en arreglos, tamaño de variables alfanuméricas, etc, y se prefiere hacerlo con variables debido a que

sus valores se almacenan utilizando menos localidades de memoria y, por lo tanto un empleo más eficiente de la memoria de la computadora.

La aritmética de modo entero tiene características particulares, cuando se realizan operaciones aritméticas (suma, resta, multiplicación y división) con constantes enteras se produce como resultado siempre una constante entera. Veamos el ejemplo clásico de la división:

7\2 produce como resultado 3
31\49 produce como resultado 0

En algunos lenguajes de programación se indica la división entera con el símbolo \ (diagonal inversa), en otros mediante el uso de variables y cantidades definidas específicamente como de tipo entero.

Constante alfanumérica. Son valores que involucran no solamente dígitos, sino también letras o caracteres y que en su conjunto tienen asignado un valor numérico para almacenarse en la memoria de la computadora.

7.2.4 VARIABLES

Una variable es un nombre que empieza con una letra y puede estar formada con varios caracteres más (por lo general sólo son válidos dígitos, letras y el subguión _).

Las variables están clasificadas en:

ENTERAS
REALES
ALFANUMÉRICAS

Y se emplean para almacenar los valores definidos por constantes de tipo:

ENTERO
REAL
ALFANUMÉRICO

Las variables enteras son aquellas cuyo nombre está mencionado en una declaración de este tipo o que se dan por definición en algún lenguaje o por que incluyen algún caracter específico que actúa como identificador de tipo, como en el caso de lenguaje BASIC, por ejemplo: I%,J%,K%,L%,M%,N%, (el caracter % identifica

a variables tipo entero). Estas variables sólo almacenan valores enteros por lo que si se les da valores con fracción lo redondearán al entero más próximo para almacenarlo. Algunos lenguajes no los redondean y sólo toman la parte entera; en lenguaje C se definen mediante declaraciones.

Las variables reales son aquellas cuyo nombre se incluye en una declaración de tipo (lenguaje C) o que son consideradas así por omisión (lenguaje BASIC), por ejemplo: A,B,C,X,Y,Z. En el caso de BASIC se puede adicionar un caracter como identificador de tipo, por ejemplo: A!,B!, etc.

La clasificación en variables de tipo alfanumérico requiere la presencia de una declaración de tipo al principio del programa fuente (lenguaje C) o de un identificador como se muestra a continuación para el caso de BASIC: RESP$,X$,W$

7.2.5 FUNCIONES DE BIBLIOTECA

Una función de biblioteca es un subproceso preprogramado, identificado con un nombre de una letra y de uno a cinco caracteres más, seguido de argumento(s) entre paréntesis que asocia a él o a los valores definidos por argumento(s), un valor único en respuesta al llamado de la función.

Todo lenguaje de programación cuenta con una cantidad básica de funciones conocidas como de biblioteca, que ayudan al programador en sus labores y evitan definirlas con subprocesos adicionales. Su número varía de lenguaje a lenguaje y de computadora a computadora, sin embargo, podemos decir que dicha lista básica y sintaxis es prácticamente común a todos los lenguajes. Las funciones pueden emplearse en todas las expresiones aritméticas, órdenes de escritura y de asignación. Como ejemplo considere las siguientes funciones:

FUNCIONES BÁSICAS MÁS USUALES:

VALOR ABSOLUTO DE X	ABS(X)
LOGARITMO NATURAL DE X	LOG(X)
SENO TRIGONOMÉTRICO DE X (en radianes)	SIN(X)
COSENO TRIGONOMÉTRICO DE X (en radianes)	COS(X)
FUNCIÓN e^x	EXP(X)
RAÍZ CUADRADA DE X	SQR(X)
TANGENTE DE X (en radianes)	TAN(X)
ARCO TANGENTE DE X	ATN(X)

APLICACIÓN DE LOS ELEMENTOS DE PROGRAMACIÓN EN LA GENERACIÓN DE EXPRESIONES

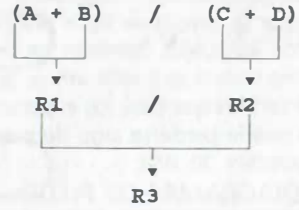
Todos los elementos de programación mencionados nos ayudan en la definición exacta de la solución de algún problema, puesto que la sintaxis (correcta escritura) de todas las instrucciones de un programa no puede presentar errores para compilar el programa fuente (conjunto de instrucciones del programa escritas por ejemplo en BASIC). Empecemos con una metodología de codificación de expresiones sencillas y su evaluación por la computadora (sin particularizar en algún lenguaje), con la prioridad de evaluación de operadores aritméticos siguiente:

operador	operación	prioridad
^	exponenciación	máxima o 1
-	negación	2
* /	multiplicación/división	3
\	división entera	4
a MOD b	residuo división entera	5
+ -	suma/resta	6

Cuando se requiere evaluar primero parte de una instrucción, se agrupa con paréntesis redondos: ( ).

EXPRESIÓN ALGEBRAICA	CODIFICACIÓN
$\frac{a+b}{c+d}$	`(A + B)/(C + D)`
$a+\frac{b}{c+d}$	`A + B/(C + D)`
$a+\frac{b}{c}+d$	`A + B/C + D`
$\frac{a+b}{c}+d$	`(A + B)/C + D`

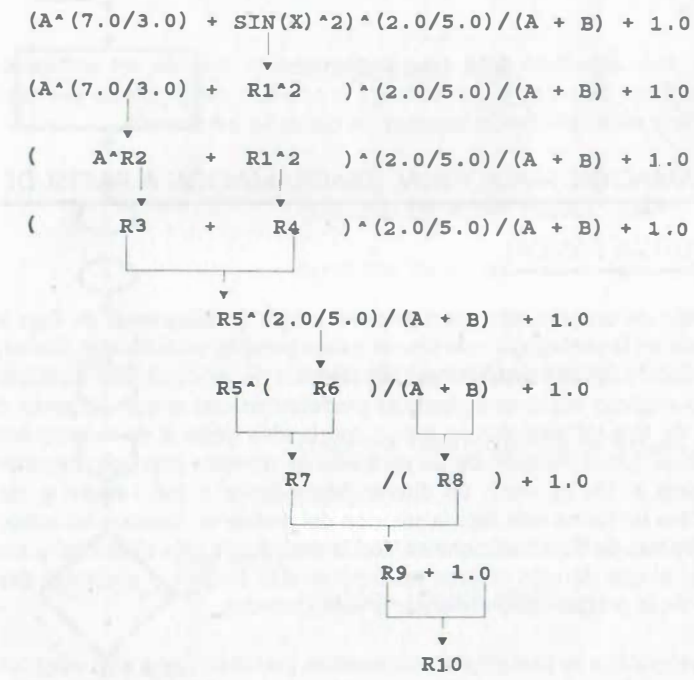
Por codificación se entiende la escritura en algún lenguaje de programación de las expresiones aritméticas dadas en la parte izquierda de la hoja. En la codificación indicada a continuación de ellas, se nota la semejanza de las cuatro expresiones y que dicha diferencia radica en la presencia o ausencia de paréntesis redondos, que sin embargo, permiten una evaluación muy distinta. En efecto, la primera de las codificaciones se evalúa, siguiendo los resultados parciales R's como sigue:



Veamos otro ejemplo:

$$\frac{[a^{7/3} + sen^2(x)]^{2/5}}{a+b} + 1$$

La cual se codifica y evalúa apoyándose en resultados parciales (R's) como sigue:



NOTA:

Observe que las funciones de biblioteca deben escribirse tal como se definen y no se puede alterar su sintaxis (escritura).

Observe que es conveniente evitar la mezcla de tipos distintos de variables para evitar errores de precisión en algunos lenguajes, también se recomienda que todo exponente que pueda conservarse como entero se quede así, ya que para su evaluación se usa el concepto de multiplicación, mientras que para los exponentes reales el método general de evaluación es por series y puede perderse algo de precisión.

7.3 SIMBOLOGÍA BÁSICA DE DIAGRAMAS DE FLUJO

Todo tipo de algoritmo se puede representar gráficamente mediante símbolos previamente definidos que indican las características básicas de un algoritmo:

1. Inicio
2. Datos de entrada
3. Proceso de la información de entrada
4. Salida
5. Fin

Además, todo algoritmo debe estar perfectamente definido, sin ambigüedades, con un número finito de pasos y que obtenga la solución del problema planteado de manera eficiente y eficaz utilizando recursos de cómputo y humanos.

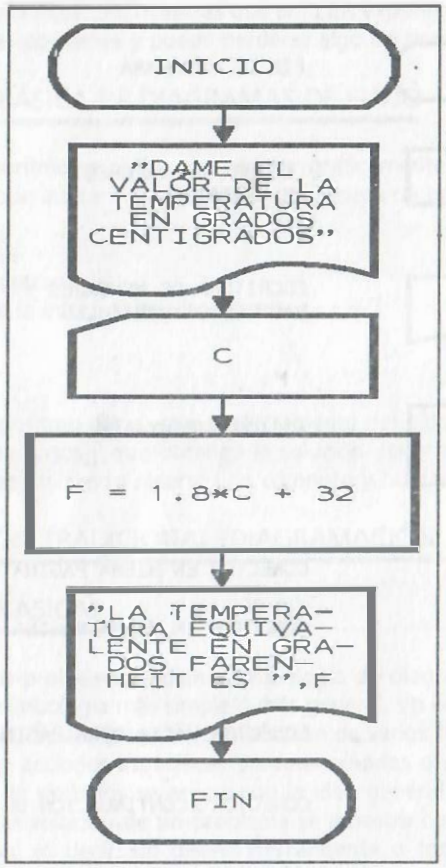
7.4 DIAGRAMACIÓN TRADICIONAL (DIAGRAMACIÓN A PARTIR DE SIMBOLOGÍA BÁSICA)

La solución de un problema mediante simbología de diagramas de flujo puede hacerse con base en la simbología más simple o más general, sin embargo, derivado de ella se han producido figuras compuestas (con la unión de varios símbolos básicos) que en su conjunto implican acciones específicas predeterminadas o que permiten definir subprocesos. Este tipo de símbolos se asocia con la idea general de la programación estructurada en la que la solución de un problema se muestra con una trayectoria de solución del inicio al fin, es decir, un diseño descendente o *top - down* y, de esta manera se plantea de forma más fácil la solución del problema. Veamos las estructuras básicas de diagramas de flujo tradicionales, con la simbología más elemental y, una vez dominado esto, el uso de esas nuevas estructuras más simples y eficientes desde el punto de vista de la programación (diagrama estructurado).

La simbología que se presenta a continuación pretende ser la más aceptada, sin embargo, como ya se indicó, no existe un estándar mundial que norme el uso de figuras geométricas en la elaboración de diagramas de flujo.

SIMBOLOGÍA BÁSICA

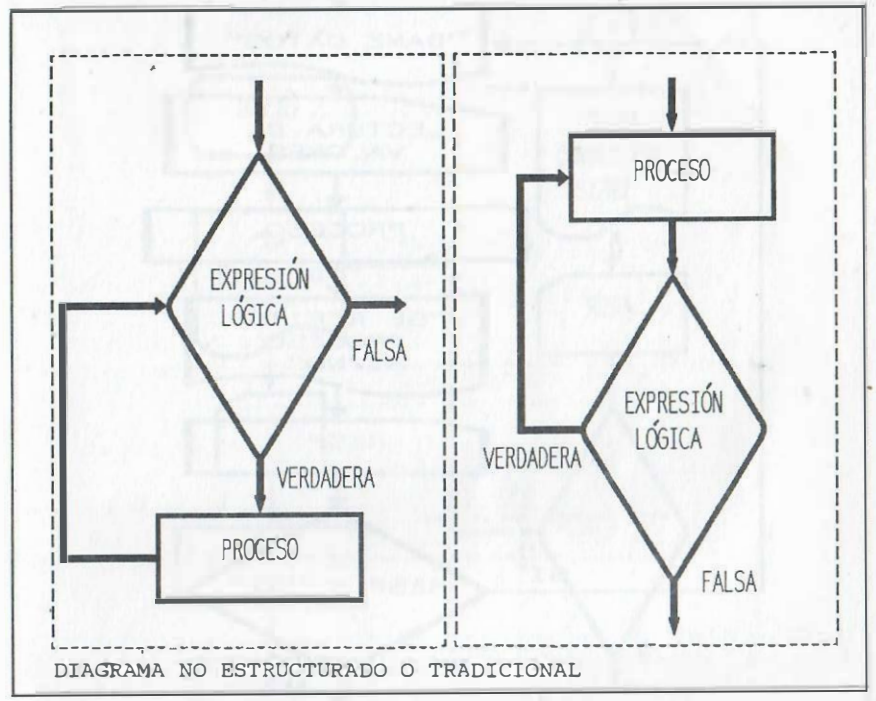
El ejemplo más simple de algoritmo numérico representado mediante diagrama de flujo puede ser el cálculo de una fórmula, por ejemplo: $F = 1.8^0C + 32$



La computadora se emplea para resolver cualquier algoritmo numérico, pero es realmente útil cuando facilita la tarea tediosa de procedimientos que se llevan a cabo una y otra vez. De ahí que el concepto más importante en la programación es el diagrama estructurado para el ciclo iterativo que, de hecho, permite calcular todo proceso que se repite o realiza de forma cíclica, por ejemplo: calcular la nómina quincenal de una empresa, la aplicación recursiva de fórmulas algebraicas para la obtención de valores aproximados de raíces de ecuaciones, el ordenamiento alfabético del padrón de electores, o procesar la información de un censo poblacional, etc.

7.5 EL CICLO ITERATIVO

Se entiende por ciclo iterativo la unión de símbolos gráficos que en su conjunto indica cómo repetir cierto proceso; las estructuras básicas más comunes son:



Observamos que mediante una pregunta, se opta por repetir n veces el proceso que se indica en el rectángulo. Es evidente que en las labores humanas existe la necesidad de repetir una y otra vez una serie de actividades en forma cíclica. Usar un ciclo es básico o esencial en la solución de múltiples problemas que repiten procesos. Por ejemplo el pago quincenal de la nómina de una empresa, la clasificación de información, etc.

Es conveniente realizar la solución de problemas con la computadora de manera conversacional, es decir, que el programa de computadora nos vaya indicando mediante mensajes las opciones y si se desea repetir el proceso con otros datos. El bosquejo general de un diagrama de flujo tradicional de tipo conversacional puede ser el siguiente:

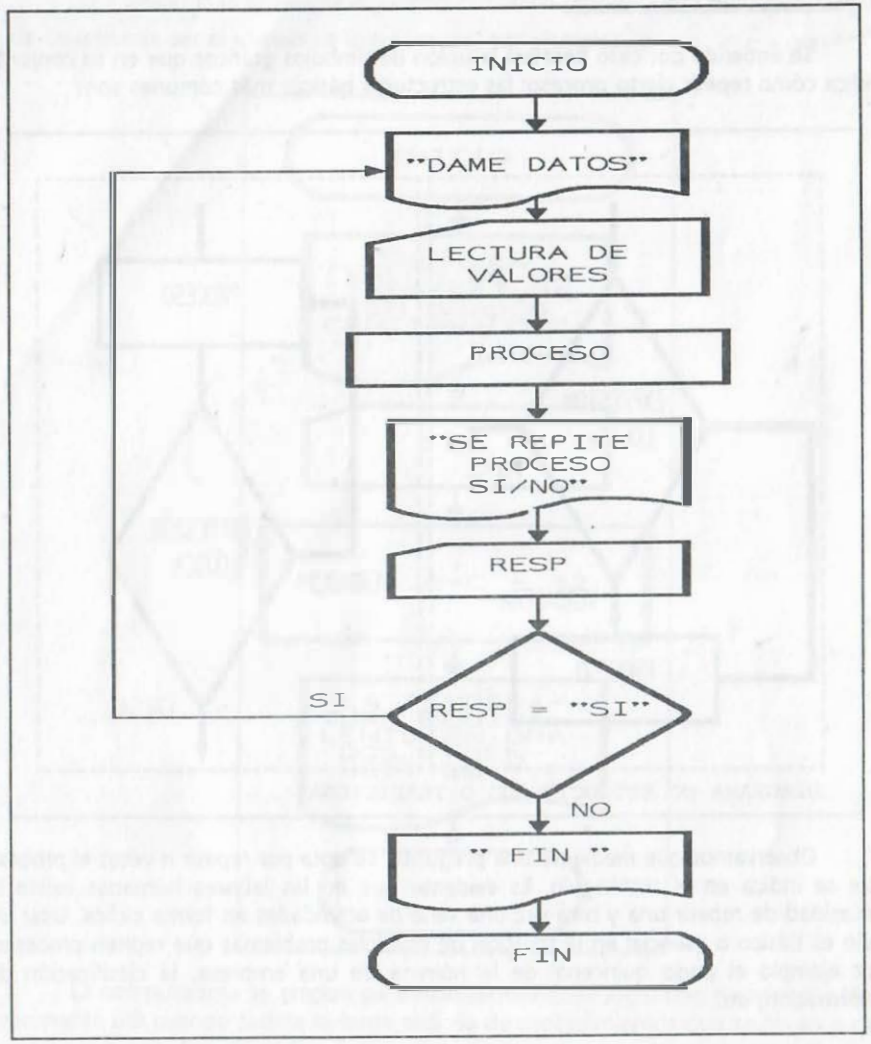


DIAGRAMA TRADICIONAL

Observe cómo se ha conformado un ciclo iterativo que permite repetir un procesamiento de información y cuya ejecución se controla con el valor que se da a la variable alfanumérica denominada RESP; cuando se alimenta el valor SI la condición se cumple y se repite el ciclo (ciclo controlado por condición alfanumérica).

El ejemplo de diagrama de flujo conversacional para el cálculo de la fórmula $F = 1.8°C + 32$, es el siguiente:

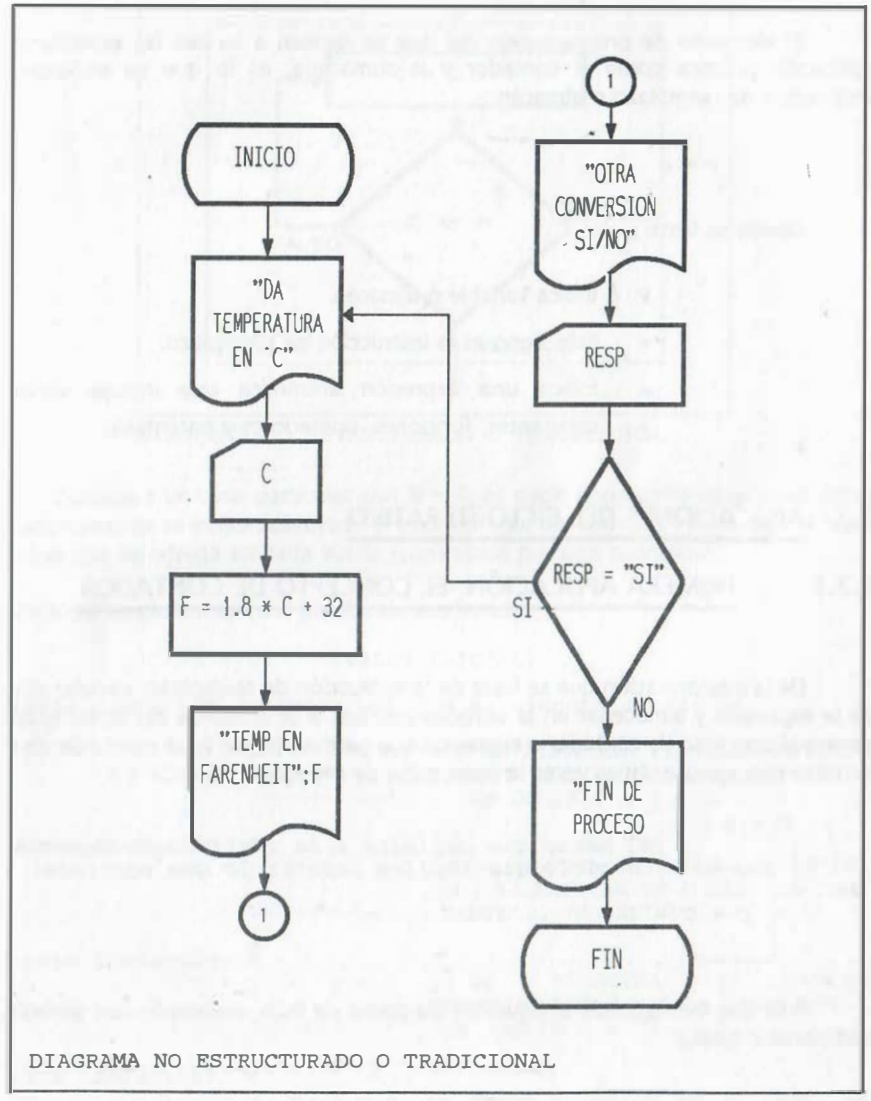


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

En este caso la repetición se controla con el valor que se alimenta a la variable alfanumérica RESP, cuando se lee algo distinto de SI, el proceso termina.

7.6 INSTRUCCIÓN DE REEMPLAZO O CONCEPTO DE ALMACÉN

El elemento de programación del que se derivan a su vez las estructuras de aplicación práctica como el contador y la sumatoria, es lo que se entiende por instrucción de reemplazo o almacén:

$$V = e$$

Donde se tiene que:

- *V* Indica variable o almacén.
- = Este signo es la instrucción de reemplazo.
- *e* Indica una expresión aritmética que incluye variables, constantes, funciones, operadores y paréntesis.

7.7 APLICACIONES DEL CICLO ITERATIVO

7.7.1 PRIMERA APLICACIÓN: EL CONCEPTO DE CONTADOR

De la interpretación que se hace de la instrucción de reemplazo: calcular el valor de la expresión y almacenar en la variable indicada a la izquierda del signo igual, se genera el concepto de contador o expresión que permite contar (que consta de un ciclo iterativo que ejecuta varias veces la instrucción de reemplazo: C = C+ 1):

```
C = 0
        (El marco que delimita a la instrucción representa
         un ciclo que implica repetir lo que contiene)
C = C + 1
```

A lo que corresponde el siguiente diagrama de flujo, elaborado con simbología tradicional o básica:

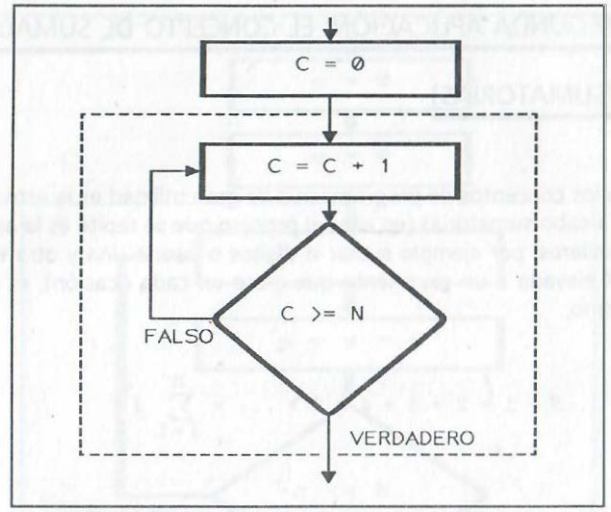


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

Considere un caso particular con N = 4, es decir, con cuatro iteraciones donde notacionalmente se indica subrayado el valor anterior del almacén y enseguida el nuevo término que se agrega en cada vuelta (controlada por una condición).

Análisis del funcionamiento o prueba de escritorio:

```
C = 0      (VALOR INICIAL)

primera iteración (primera vez que se ejecuta el ciclo):

C = 0 + 1 = 1    ( SE ALMACENA EL VALOR ANTERIOR
                   DE C (CERO) MÁS LA CONSTANTE I,
                   ES DECIR C = C + 1 ).

segunda iteración:
C = 1 + 1 = 2    ( SE ALMACENA EL VALOR QUE SE ACA-
                   BA DE CALCULAR DE C MÁS LA CONS-
                   TANTE 1, ES DECIR C = C + 1).

tercera iteración:
C = 2 + 1 = 3    ( SE ALMACENA EL VALOR
                   INMEDIATO DE C MÁS LA CONSTANTE 1,
                   ES DECIR C = C + 1 ).

cuarta iteración:
C = 3 + 1 = 4    ( VALOR ANTERIOR DE C MÁS LA
                   CONSTANTE 1: C = C + 1 ).
```

7.7.2 SEGUNDA APLICACIÓN: EL CONCEPTO DE SUMADOR (SUMATORIAS)

Otro de los conceptos de programación de gran utilidad es la estructura que nos permite llevar a cabo sumatorias (en ellas el proceso que se repite es la acumulación de expresiones similares, por ejemplo sumar *n* dígitos o sumar una y otra vez el valor de una variable X elevada a un exponente que crece en cada ocasión), es decir, calcular expresiones como:

$$S = 1 + 2 + 3 + 4 + 5 + \ldots = \sum_{i=1}^{n} i$$

$$W = X^1 + X^2 + X^3 + X^4 + X^5 + \ldots = \sum_{i=1}^{n} X^i$$

$$Z = X^1 + X^2 + X^3 + X^5 + X^8 + \ldots$$

NOTA:
En esta última expresión, los exponentes a partir del tercero se obtienen con la suma de los dos exponentes anteriores.

El concepto de sumador también involucra un ciclo iterativo que contiene una instrucción de reemplazo que en este caso acumula el valor de una variable.

Esta aplicación por lo general también controla el número de veces que se ejecuta el ciclo mediante un contador.

Veamos el siguiente ejemplo donde se suman los dígitos que genera un contador:

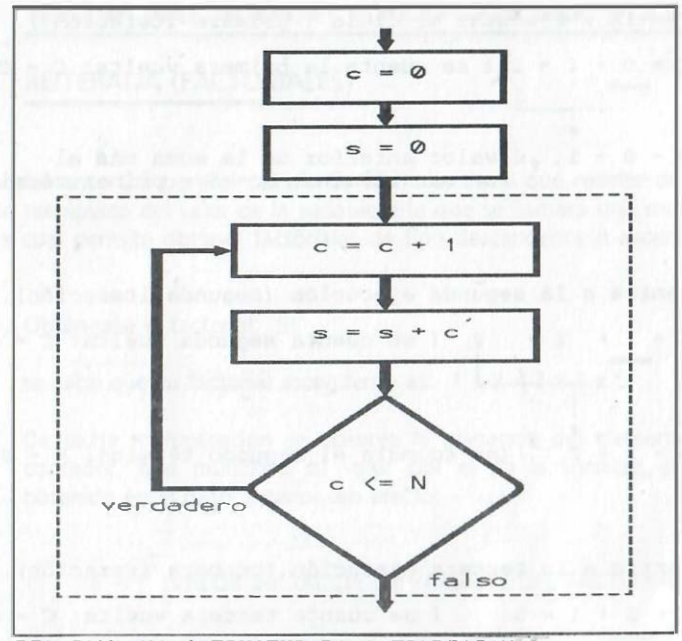


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

```
C = 0
S = 0

   C = C + 1
   S = S + C
```

La definición de sumador estructuralmente es similar a la del contador, con la diferencia de que ahora se acumula en cada iteración un nuevo término dado por una variable o contador (considérese el ciclo iterativo que permite calcular la sumatoria siguiente: S = 1 + 2 + 3 + 4 ; y veamos su análisis de funcionamiento o prueba de escritorio).

```
valores iniciales para entrar al ciclo iterativo.

C = 0    (el contador no ha contado nada)
S = 0    (valor inicial de la suma)
```

Notacionalmente se indica subrayado el valor anterior del almacén y enseguida el nuevo término que se agrega al valor anterior.

entra a ejecutar el ciclo (primera iteración).

C = 0 + 1 = 1 (se cuenta la primera vuelta: C = C + 1)

S = 0 + 1 (valor anterior de la suma más el primer término: S = S + primer término)

entra a la segunda ejecución (segunda iteración).

C = 1 + 1 = 2 (se cuenta segunda vuelta: C = C + 1)

S = 1 + 2 (se acumula el segundo término: S = S + C)

entra a la tercera ejecución (tercera iteración).

C = 2 + 1 = 3 (se cuenta tercera vuelta: C = C + 1)

S = 1 + 2 + 3 (valor anterior de la suma más tercer término: S = S + C)

entra a la cuarta ejecución (cuarta iteración).

C = 3 + 1 = 4 (se cuenta cuarta vuelta: C = C + 1)

S = 1 + 2 + 3 + 4 (valor anterior de la suma más el cuarto término: S = S + C)

De igual manera se sigue repitiendo (iterando) para calcular *n* términos, según se indicó mediante el diagrama de flujo anterior.

7.7.3 TERCERA APLICACIÓN: EL CONCEPTO DE MULTIPLICACIÓN REITERADA (FACTORIALES)

Otro elemento útil para el cálculo de fórmulas es el que resulta de aplicar el concepto de reemplazo del valor de la variable a lo que se llamará una multiplicación reiterada, la cual permite obtener factoriales de tipo descendente o ascendente. Por ejemplo:

Obténgase el factorial: 5!

se sabe que su factorial ascendente es: 1 x 2 x 3 x 4 x 5

De dicha multiplicación se observa la presencia del elemento llamado contador, que multiplica su valor por el de la variable del factorial obtenido en el paso anterior, en efecto:

F = 1 (VALOR INICIAL DE LA VARIABLE DEL FACTORIAL)

F = 1 x 2 (VALOR ANTERIOR DE F MULTIPLICADO POR 2, ES DECIR, F = F x 2)

F = 1 x 2 x 3 (VALOR ANTERIOR DE F MULTIPLICADO POR 3, ES DECIR, F = F x 3)

F = 1 x 2 x 3 x 4 (VALOR ANTERIOR DE F MULTIPLICADO POR 4, ES DECIR, F = F x 4)

F = 1 x 2 x 3 x 4 x 5 (VALOR ANTERIOR DE F MULTIPLICADO POR 5, ES DECIR, F = F x 5)

Apoyándose en el concepto de almacén y en la estructura previamente obtenida para el contador, se obtiene el algoritmo para la multiplicación sucesiva o factorial para enteros positivos a partir de la unidad, que en forma simplificada se representa:

```
K = 0
F = 1
    ( Indicamos que el proceso se ejecutará n
      veces al iterar )
    K = K + 1
    F = F x K
```

lo cual representado en diagrama de flujo queda:

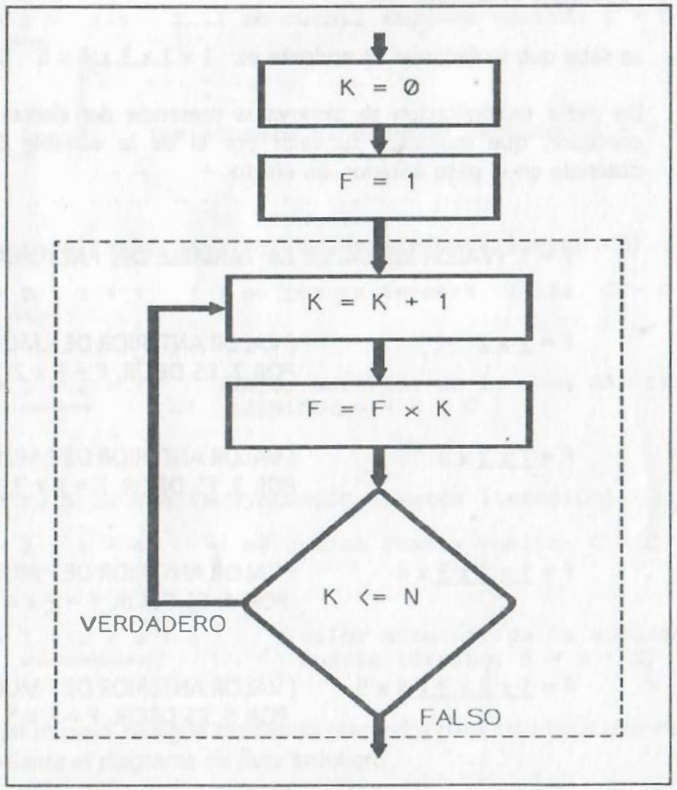


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

En el diagrama la comparación: K <= N, permite repetir el proceso; considere ahora el diagrama completo para obtener factoriales de números enteros positivos no nulos (discuta, corrobore o establezca la operación lógica para repetir).

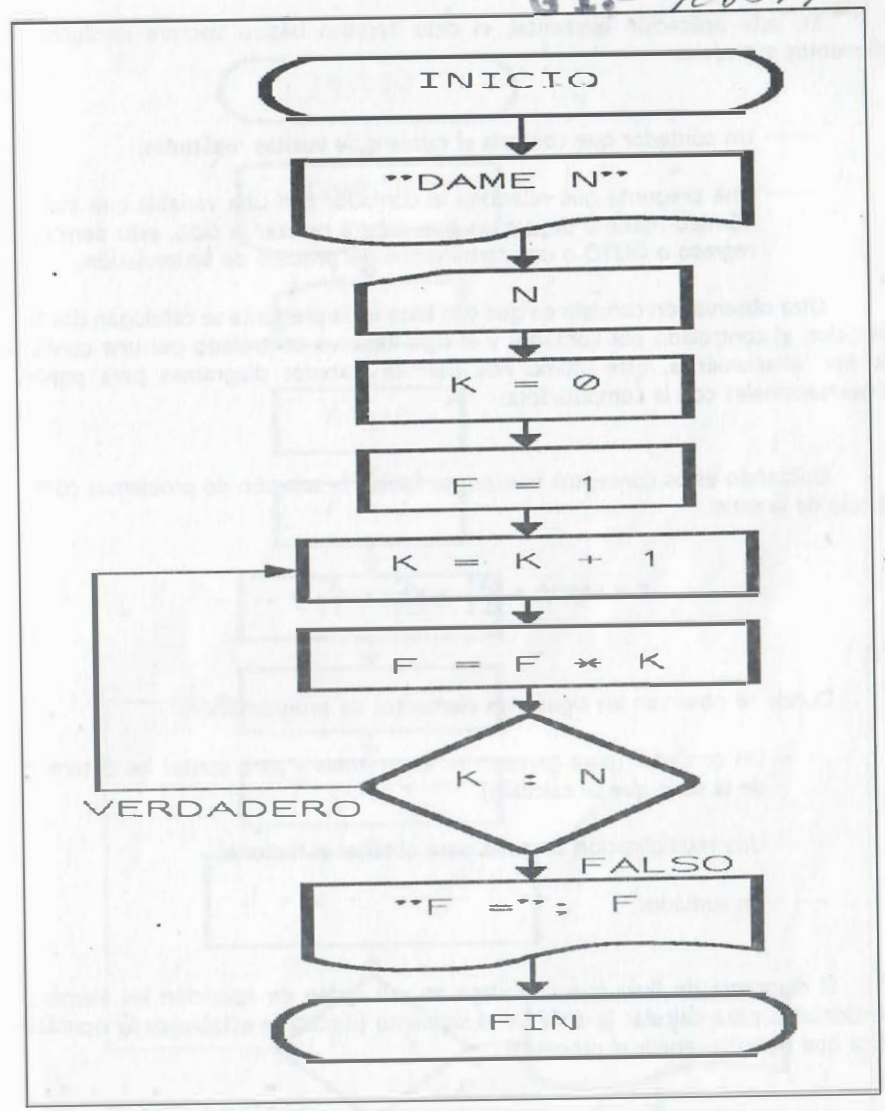


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

En esta aplicación elemental, el ciclo iterativo básico siempre involucra dos elementos esenciales:

—— Un contador que controla el número de vueltas realizadas.

—— Una pregunta que relaciona al contador con una variable que indica el número máximo de vueltas que deberá realizar el ciclo, esto permite un regreso o GOTO o una terminación del proceso de acumulación.

Otra observación consiste en que con base en la pregunta se catalogan dos tipos de ciclos: el controlado por contador y el ciclo iterativo controlado por una condición de tipo alfanumérica, este último nos permite elaborar diagramas para procesos conversacionales con la computadora.

Utilizando estos conceptos básicos, se facilita la solución de problemas como el cálculo de la serie:

$$Z = -1000 + \frac{X^2}{1!} + \frac{X^4}{2!} + \frac{X^6}{3!} + \ldots$$

Donde se observan los siguientes elementos de programación:

—— Un contador (para generar los exponentes y para contar los ***N*** términos de la serie que se calculan).

—— Una multiplicación sucesiva para obtener el factorial.

—— Un sumador.

El diagrama de flujo que considera en ese orden de aparición los elementos mencionados para calcular la serie es el siguiente (discuta y establezca la operación lógica que permita repetir el proceso).

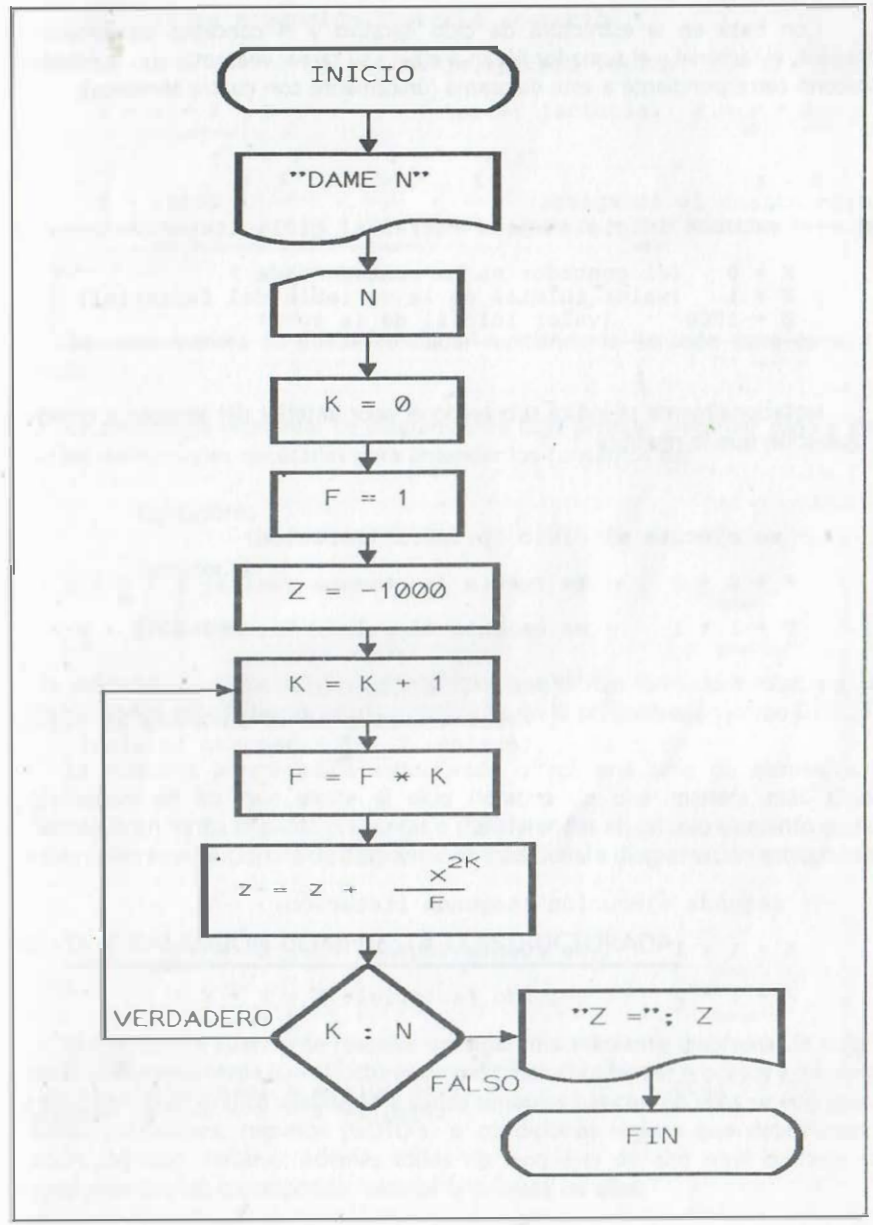


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

Con base en la estructura de ciclo iterativo y el concepto de almacén, el contador, el factorial y el sumador llevan a efecto su tarea; veámoslo con la prueba de escritorio correspondiente a este diagrama (únicamente con cuatro términos):

```
N = 4

  valores iniciales para entrar al ciclo iterativo
  K = 0    (el contador no ha contado nada )
  F = 1    (valor inicial de la variable del factorial)
  Z = -1000      (valor inicial de la suma)
```

Notacionalmente se indica subrayado el valor anterior del almacén y enseguida la operación que lo modifica:

```
  se ejecuta el ciclo (primera iteración)
  K = 0 + 1     ( se cuenta la primera vuelta: K = K + 1 )
  F = 1 * 1     ( se calcula el primer factorial: F = F * K )
                  2(1)
                 X
  Z = -1000 + ------    ( valor anterior de la suma más el segundo
                 1!       término: Z = Z + segundo término)
```

```
  segunda ejecución (segunda iteración).
  K = 1 + 1     ( se cuenta segunda vuelta: K = K + 1 )
  F = 1 * 2     ( segundo factorial: F = F * K )
                   2       2(2)
                  X        X
  Z = -1000 + ------  + ------   ( se agrega a la suma el tercer
                 1!       2!       término: Z = Z + tercer término )
```



```
  tercera ejecución (tercera iteración).
  K = 2 + 1     ( se cuenta la tercera vuelta: K = K + 1 )
  F = 1 * 2 * 3          ( tercer factorial: F = F * K )
                   2       4       2(3)
                  X       X       X
  Z = -1000 + ------  + ------  + ------  (agregamos el cuarto térmi-
                 1!       2!      3!      no: Z = Z + cuarto término)
```

De igual manera se puede continuar repitiendo la iteración para calcular *n* términos.

La simbología elemental de diagramas de flujo permite presentar paso a paso todas las instrucciones necesarias para entender los conceptos de:

- Contadores
- Sumatorias
- Factoriales

con la utilización de ciclos iterativos elementales que se han formado mediante el uso de instrucciones específicas de regreso conocidas en la programación como GOTOS.

La moderna programación estructurada ofrece una serie de elementos de diagramación en los que existe el ciclo iterativo de una manera más simple, involucrando en forma implícita preguntas o transferencias en un solo elemento gráfico. En estas notas se evolucionará de diagramación tradicional a diagramación estructurada.

7.8 DIAGRAMACIÓN COMPUESTA O ESTRUCTURADA

Normalmente cuando se resuelve un algoritmo mediante diagrama de flujo se emplean preferentemente las estructuras simplificadas para repetir procesos y preguntas que en un símbolo gráfico sustituyen a varios símbolos básicos. En ellos se encuentran implícitos contadores, regresos (GOTO's) o condiciones lógicas que determinan la duración del ciclo iterativo; además todos los lenguajes de alto nivel cuentan con instrucciones que les corresponde; veamos la primera de ellas:

7.8.1 CICLO ITERATIVO CONTROLADO POR UN CONTADOR

EL CICLO ESTRUCTURADO

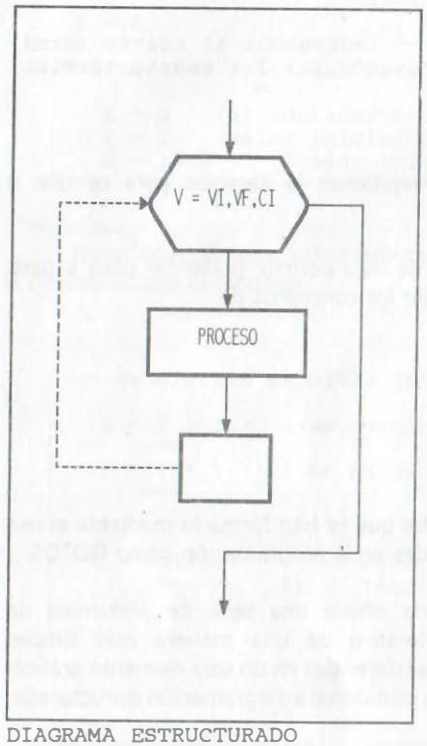


DIAGRAMA ESTRUCTURADO

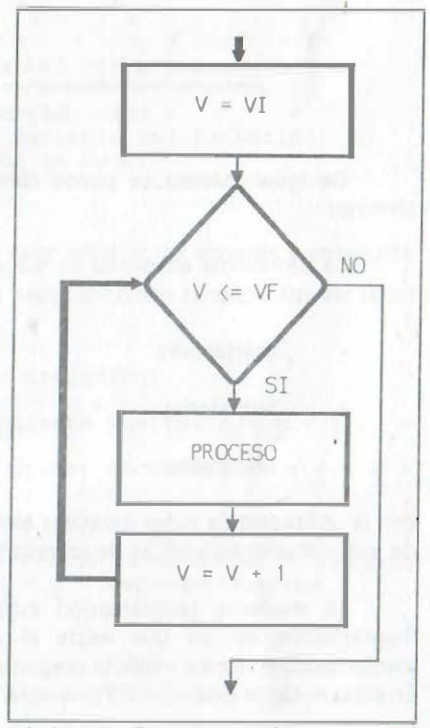


DIAGRAMA TRADICIONAL

El diagrama de la izquierda indica que el proceso enmarcado en el rectángulo se ejecutará tantas veces como sea necesario para que la variable V cambie de un valor inicial VI a un valor final VF (la repetición del ciclo es automática y el ciclo es controlado por un contador asociado a este símbolo compuesto), al utilizar un incremento indicado en la variable CI por cada vez que se ejecuta dicho proceso.

El cuadro indicado en blanco, como última instrucción del ciclo, podrá asociarse con alguna instrucción específica del lenguaje en que se programe y, por tanto, siempre se dibujará en blanco; de ese recuadro, también se realiza un regreso automático

indicado por la línea punteada de la izquierda. Al efectuar dicho regreso, la variable del ciclo se incrementa automáticamente y se pregunta si el contador implícito no ha excedido el valor límite, en caso de ser menor, nuevamente se ejecuta el proceso; en caso de ser mayor, se sale del ciclo según indica la línea de la derecha.

En forma reducida, podemos indicar lo anterior como:

```
V = VI,VF,CI

PROCESO
```

Los valores de las variables *VI,VF,* y *CI* pueden ser fraccionarios o enteros, además también se acostumbra colocar valores numéricos directamente en lugar de las variables. Cuando el incremento es unitario, en muchos lenguajes se sobrentiende y se puede eliminar junto con la coma que le precede; en el caso del lenguaje C el incremento debe indicarse siempre. Este ciclo se usa en la lectura de arreglos y en todo aquello que se repite un número prestablecido de veces.

En resumen se puede comentar que el ciclo toma un valor inicial con:

V = VI

a continuación se hace la pregunta implícita:

V <= VF (para verificar que no se haya excedido el límite de variación *VF)*

y por último se incrementa el contador implícito, es decir se ejecuta:

V = V + CI

En algunos casos como en el del lenguaje C es necesario especificar claramente la condición o pregunta y el contador que en otros lenguajes son implícitos, por ejemplo:

```
(v = vi; v <= vf; v = v + ci)

PROCESO
```

Bajo ciertas circunstancias, el diagrama estructurado del ciclo controlado por contador puede ser equivalente al siguiente ciclo tradicional:

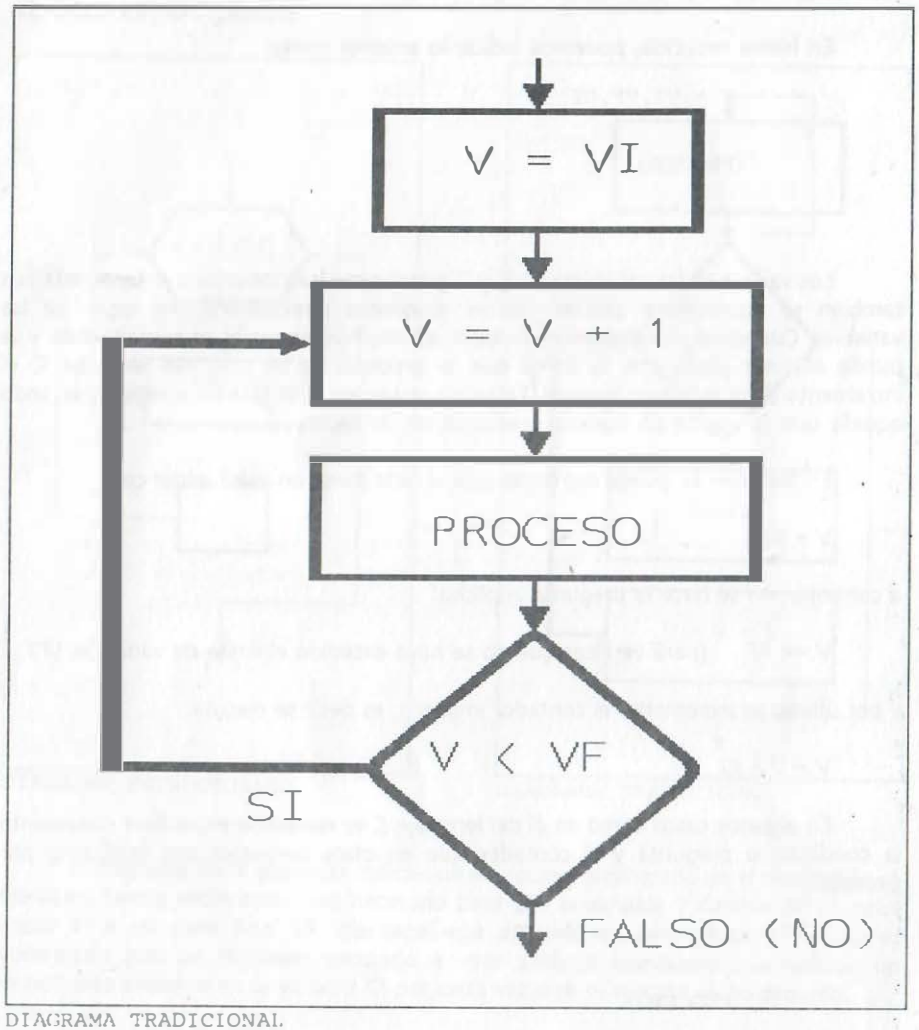


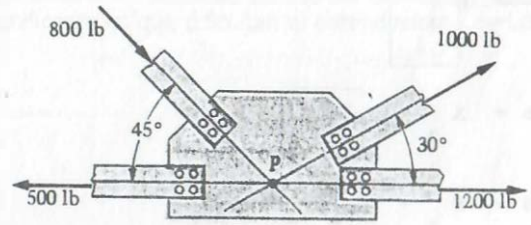
DIAGRAMA TRADICIONAL

En el diagrama anterior se observa la definición clásica del ciclo controlado por un contador.

En problemas de ingeniería es frecuente el uso de arreglos de una o más dimensiones. Un arreglo es un conjunto de valores que se asocian a un mismo nombre de variable, pero que se diferencian por medio de índices. Ejemplo:

De una dimensión:

El conjunto de fuerzas que actúan sobre un cuerpo y que en estática se identifican con *F(1), F(2), F(3), ... , F(n).*



De dos dimensiones:

Un arreglo matricial.

$$Z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Z(1,1) & Z(1,2) & Z(1,3) \\ Z(2,1) & Z(2,2) & Z(2,3) \\ Z(3,1) & Z(3,2) & Z(3,3) \end{bmatrix}$$

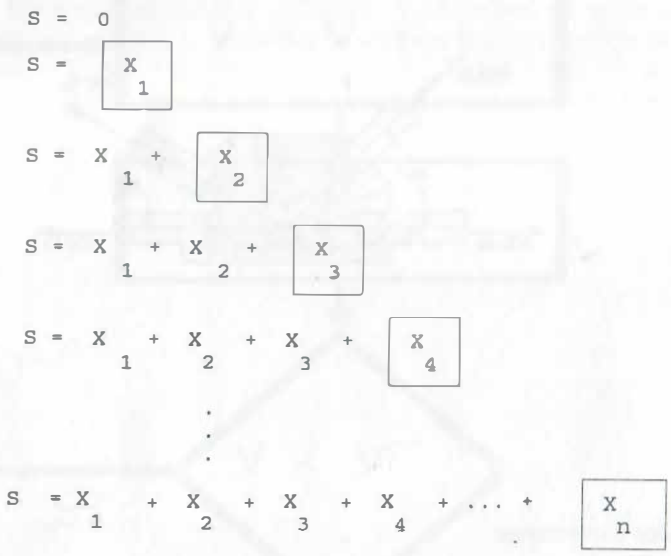
APLICACIONES

La manipulación de arreglos se facilita con el uso de estructuras simplificadas de ciclos iterativos. Por ejemplo, obténgase la sumatoria siguiente:

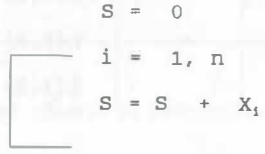
$$S = X_1 + X_2 + X_3 + X_4 + \ldots + Xn$$

Notamos la necesidad de un ciclo iterativo en la lectura de valores y en el proceso de la suma, en virtud de que se repite una y otra vez la misma actividad, esta es: leer y acumular en la variable S un elemento del vector X .

Esquemáticamente el proceso de acumulación de un nuevo término en cada iteración es el marcado con un recuadro:



que, de acuerdo con la definición de almacén para la variable, se puede representar en forma abreviada con:



lo cual indica que ese proceso se repite tantas veces como sea necesario para que el valor de *i* cambie de su valor unitario inicial, hasta el valor *n*, utilizando, en este caso, un incremento unitario. Veamos el diagrama de flujo detallado que incluye además este contador definido con la variable *i*.

NOTA:

El ciclo controlado por contador es de las estructuras más empleadas en la programación, debido a que una gran cantidad de procesos normalmente se controlan con valores numéricos de contadores. Los contadores aparecen tanto en problemas físicos, matemáticos y de manejo de información, es conveniente entender y dominar su uso.

El diagrama de flujo estructurado del ciclo iterativo estructurado constituye una herramienta simplificada, que brinda una forma sencilla para la repetición de procesos sin necesidad de pensar en la unión de símbolos gráficos elementales para su construcción y que elimina la necesidad del uso del GOTO, permitiendo un diseño más elegante y sin ramificaciones que dificultan el entendimiento de la solución planteada en el diagrama.

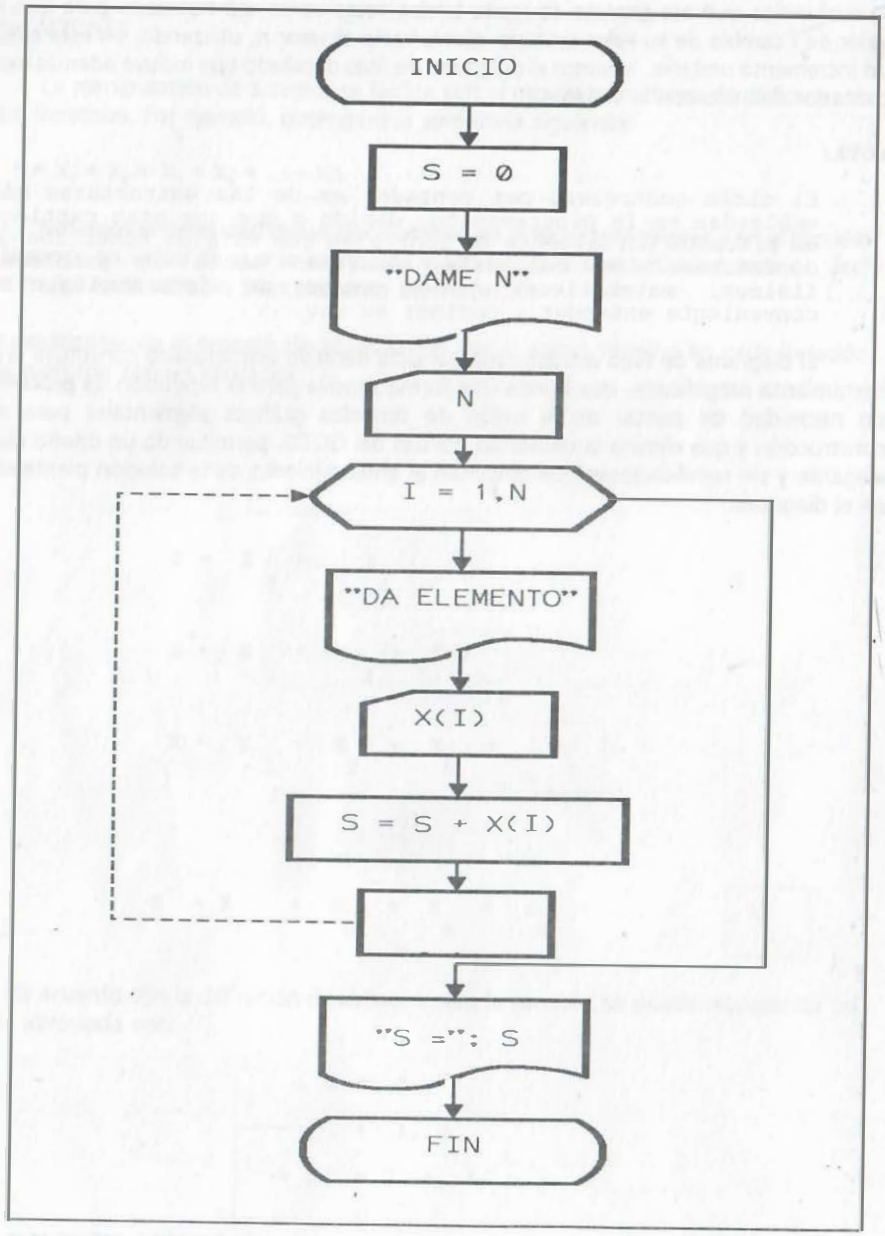


DIAGRAMA ESTRUCTURADO

7.8.2 CICLOS ANIDADOS (CON CONTADOR INCLUIDO)

MANEJO DE ARREGLOS DE DOS DIMENSIONES

Cuando una de estas estructuras iterativas contiene a otra, se dice que son ciclos anidados. En ellos, por cada valor de la variable del ciclo externo, la variable del ciclo interno varía completamente, generándose dos tipos de variación en los contadores: la externa o lenta y la interna o rápida. Este tipo de situación resulta adecuada para el manejo de arreglos de varias dimensiones, por ejemplo, la lectura de un arreglo matricial por renglones:

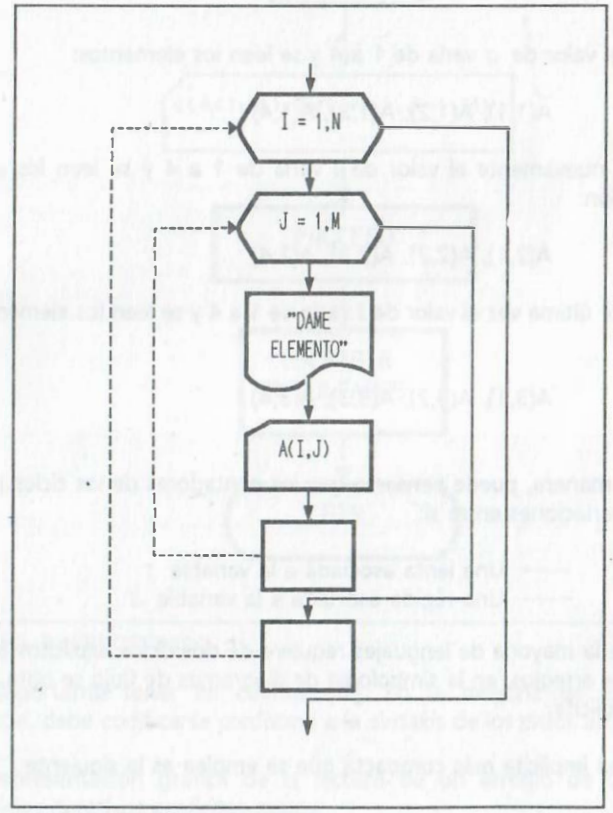


DIAGRAMA ESTRUCTURADO

```
I = 1, 3
J = 1, 4
A(I,J)
```

En este caso, como ya se mencionó, por cada valor de I, la variable J toma los valores de 1 a 4, es decir se tiene:

Con I = 1, el valor de J varía de 1 a 4 y se leen los elementos:

A(1,1), A(1,2), A(1,3), A(1,4)

Con I = 2, nuevamente el valor de J varía de 1 a 4 y se leen los elementos del segundo renglón:

A(2,1), A(2,2), A(2,3), A(2,4)

Con I = 3 por última vez el valor de J varía de 1 a 4 y se leen los elementos del tercer renglón:

A(3,1), A(3,2), A(3,3), A(3,4)

De esa manera, puede pensarse que los contadores de los ciclos han generado dos tipos de variaciones entre sí:

- Una lenta asociada a la variable I
- Una rápida asociada a la variable J

Aunque la mayoría de lenguajes requiere de dos ciclos explícitos para la lectura de este tipo de arreglos, en la simbología de diagramas de flujo se opta por indicarlos de manera implícita.

La forma implícita más compacta que se emplea es la siguiente:

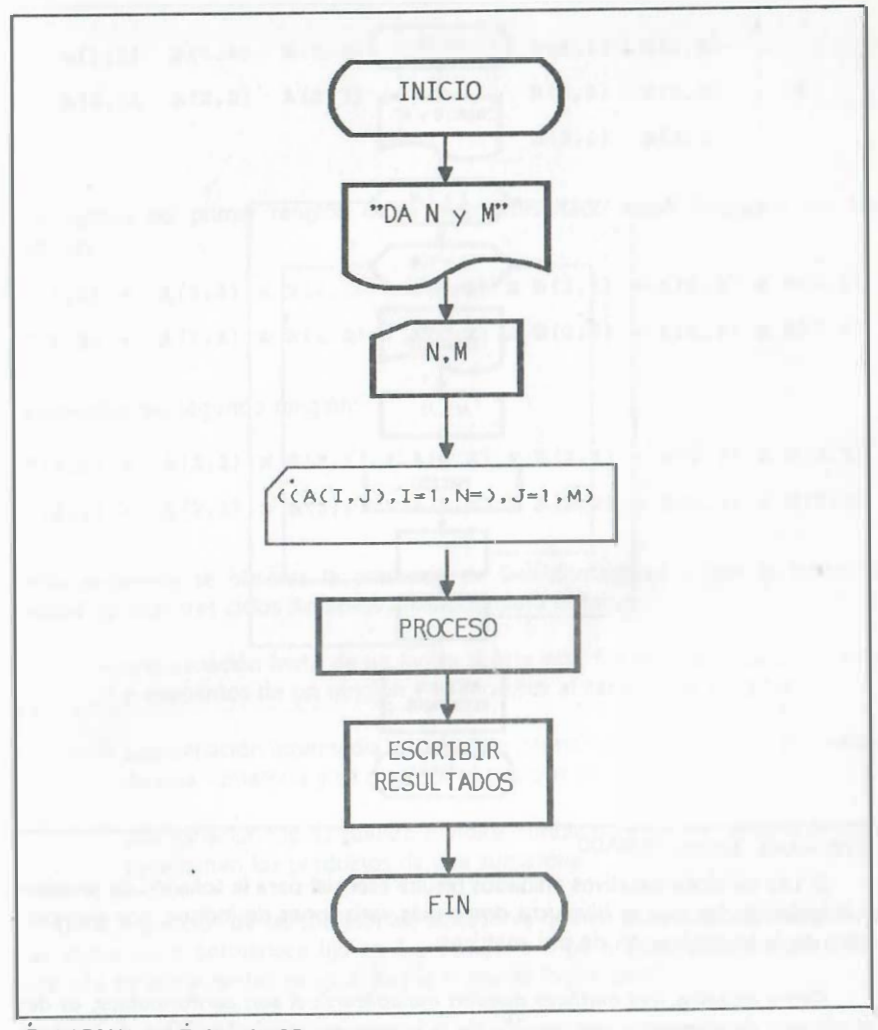


DIAGRAMA ESTRUCTURADO

Es importante tener en cuenta que, en la mayoría de los lenguajes de programación, debe codificarse conforme a la sintaxis de los ciclos anidados explícitos.

La representación gráfica de la lectura de un arreglo de dos dimensiones mediante ciclos iterativos explícitos queda:

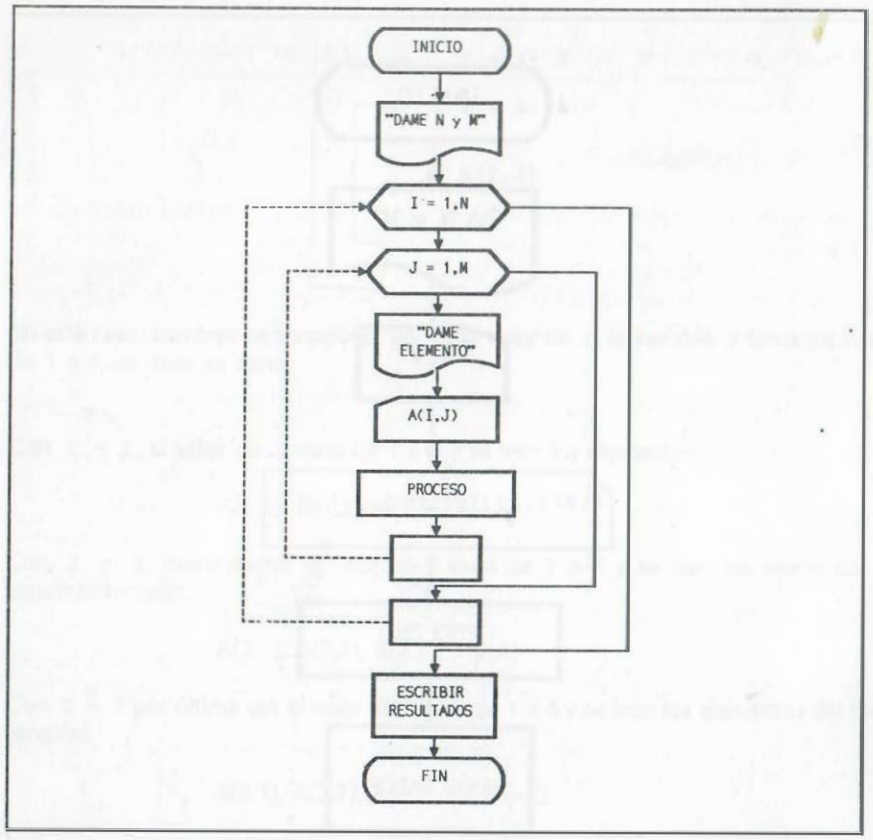


DIAGRAMA ESTRUCTURADO

El uso de ciclos iterativos anidados resulta esencial para la solución de problemas con arreglos en los que se involucra dos o más variaciones de índices, por ejemplo en el caso de la multiplicación de dos matrices:

Como se sabe, dos matrices pueden multiplicarse si son conformables, es decir, si el número de elementos por renglón de la primera es igual al número de elementos por columna de la segunda. Con estas condiciones considere el producto de dos matrices de 2 x 3 y de 3 x 2.

$$\begin{bmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \end{bmatrix} \times \begin{bmatrix} B(1,1) & B(1,2) \\ B(2,1) & B(2,2) \\ B(3,1) & B(3,2) \end{bmatrix} =$$

Los elementos del primer renglón de la matriz resultado están formados por las sumatorias:

$$C(1,1) = A(1,1) \times B(1,1) + A(1,2) \times B(2,1) + A(1,3) \times B(3,1)$$

$$C(1,2) = A(1,1) \times B(1,2) + A(1,2) \times B(2,2) + A(1,3) \times B(3,2)$$

Los elementos del segundo renglón:

$$C(2,1) = A(2,1) \times B(1,1) + A(2,2) \times B(2,1) + A(2,3) \times B(3,1)$$

$$C(2,2) = A(2,1) \times B(1,2) + A(2,2) \times B(2,2) + A(2,3) \times B(3,2)$$

De este desarrollo se observa la presencia de tres contadores y, por lo tanto, la necesidad de usar tres ciclos iterativos anidados para obtener:

- una variación lenta de un índice si éste está fijo en todas las sumatorias o elementos de un renglón y se modifica al cambiar de renglón;
- una variación intermedia si un índice permanece fijo entre los elementos de una sumatoria y se modifica al cambiar de sumatoria;
- una variación rápida cuando el índice cambia para definir las variables que determinan los productos de una sumatoria.

De la inspección de las sumatorias, se observa que en todo el primer renglón, el primer índice de A permanece fijo en 1 y cambia a 2 para el segundo renglón. Esto implica una variación lenta; de igual forma el primer índice de C:

Primer renglón de la matriz:

$A(1,)\times B(,)+A(1,)\times B(,)+A(1,)\times B(,)$ $\quad$ $A(1,)\times B(,)+A(1,)\times B(,)+A(1,)\times B(,)$

Segundo renglón:

$A(2,)\times B(,)+A(2,)\times B(,)+A(2,)\times B(,)$ $\quad$ $A(2,)\times B(,)+A(2,)\times B(2,)+A(2,)\times B(,)$

Los índices de variación intermedia son los que varían de sumatoria a sumatoria en el mismo renglón. Estos son el segundo índice de B y el segundo de C:

Primer renglón:

$$A(,1) \times B(,1) + A(,) \times B(,1) + A(,) \times B(,1) \qquad A(,) \times B(,2) + A(,) \times B(,2) + A(,) \times B(,2)$$

Segundo renglón:

$$A(,) \times B(,1) + A(,) \times B(,1) + A(,) \times B(,1) \qquad A(,) \times B(,2) + A(,) \times B(,2) + A(,) \times B(,2)$$

Por último, se observa que el segundo índice de A cambia de 1 a 3 en cada una de las sumatorias, al igual que el primer índice de B; esto implica una variación rápida:

Primer renglón:

$$A(,1) \times B(1,) + A(,2) \times B(2,) + A(,3) \times B(3,) \qquad A(,1) \times B(1,) + A(,2) \times B(2,) + A(,3) \times B(3,)$$

Segundo renglón:

$$A(,1) \times B(1,) + A(,2) \times B(2,) + A(,3) \times B(3,) \qquad A(,1) \times B(1,) + A(,2) \times B(2,) + A(,3) \times B(3,)$$

Es más o menos obvio que con el ciclo que presenta la variación rápida, también se obtiene la sumatoria con la cual se calcula cada elemento de la matriz resultante, por lo tanto el algoritmo que define la multiplicación es:

```
I = 1,2
J = 1,2
K = 1,3
C(I,J) = C(I,J) + A(I,J) * B(K,J)
```

Por lo que el diagrama de flujo para el caso general es:

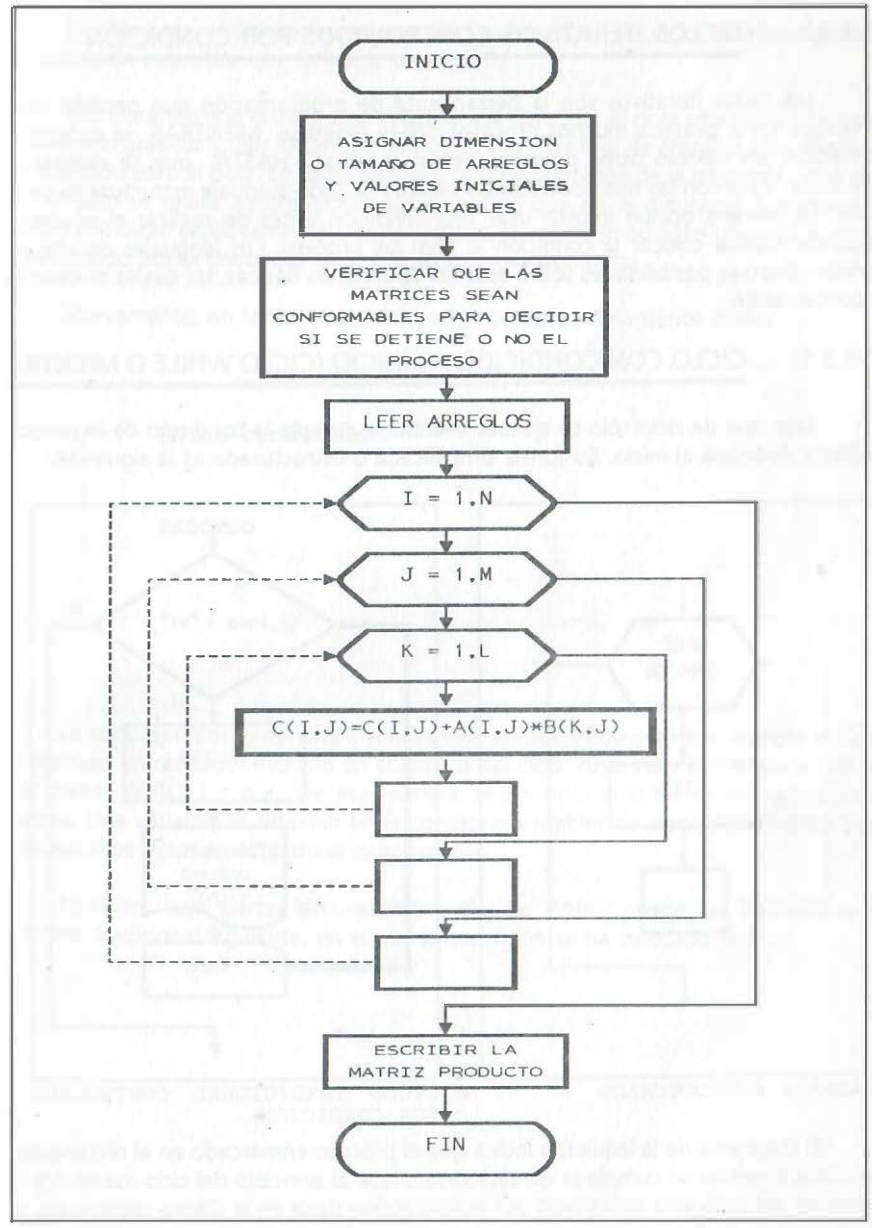


DIAGRAMA ESTRUCTURADO

7.8.3 CICLOS ITERATIVOS CONTROLADOS POR CONDICIÓN

Los ciclos iterativos son la herramienta de programación que permite repetir procesos. En la práctica muchos procesos deben repetirse MIENTRAS se cumpla una condición, en cambio otros procesos deben repetirse HASTA que se cumpla una condición. Esas son las dos posibilidades básicas de todo lenguaje estructurado de alto nivel. La primera opción implica usar una condición antes de realizar el proceso, la segunda implica colocar la condición al final del proceso. Los lenguajes de alto nivel ofrecen diversas posibilidades sobre esas construcciones básicas, las cuales se describen a continuación.

7.8.3.1 CICLO CON CONDICIÓN AL INICIO (CICLO WHILE O MIENTRAS)

Este tipo de ciclo sólo se ejecuta cuando se cumple la condición de la pregunta implícita colocada al inicio. Su forma simplificada o estructurada es la siguiente:

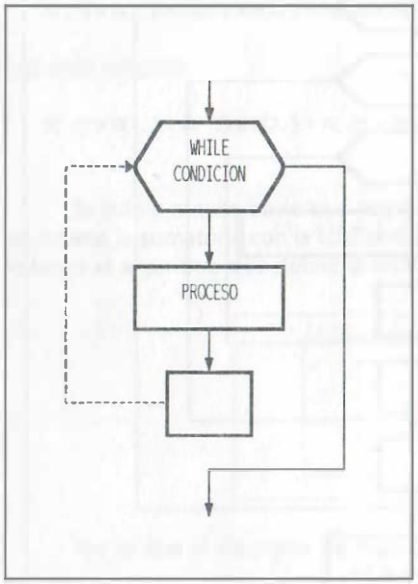
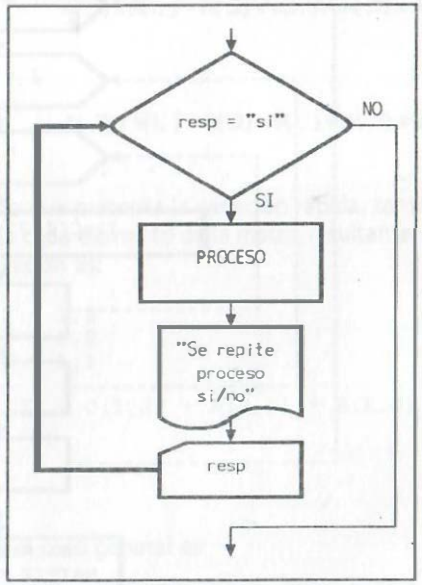


DIAGRAMA ESTRUCTURADO



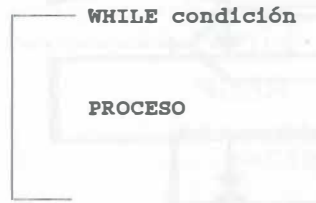
CICLO TRADICIONAL CONTROLADO POR CONDICIÓN

El diagrama de la izquierda indica que el proceso enmarcado en el rectángulo se ejecutará mientras se cumpla la condición señalada al principio del ciclo iterativo y que el ámbito del ciclo está delimitado por la instrucción dada en el último rectángulo, que en diagramas de flujo siempre se dejará en blanco (fin del ámbito del WHILE).

En este caso, el último recuadro o última instrucción del ciclo, está asociada con la instrucción específica que define al ciclo WHILE.

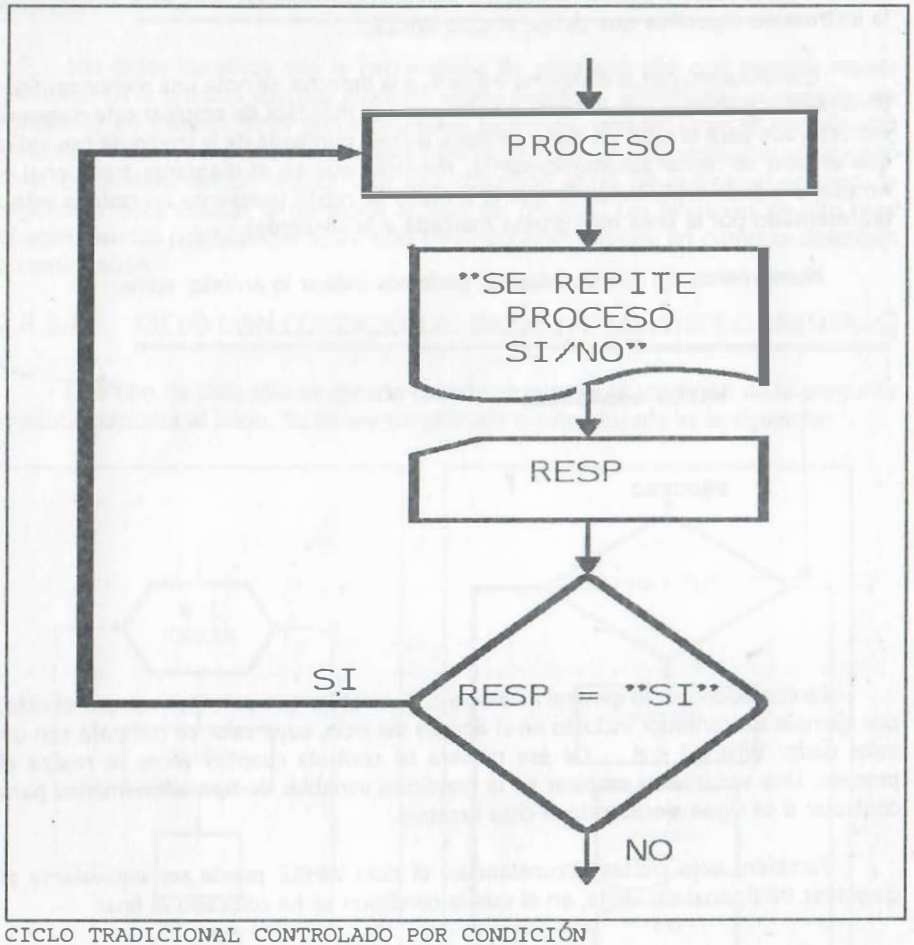
Comparando con el diagrama indicado a la derecha, se nota una menor cantidad de símbolos gráficos y, por lo tanto creemos que es más fácil de emplear este diagrama estructurado para el ciclo. En dicho símbolo, la línea punteada de la izquierda nos indica que el ciclo se repite automáticamente, mientras que en el diagrama tradicional es necesario indicar explícitamente que el proceso se repite (mediante un goto o vete a representado por la línea más gruesa marcada a la izquierda).

Nuevamente, en forma reducida, podemos indicar lo anterior como:



La condición por lo general involucra dos variables que permiten romper el ciclo, por ejemplo un contador incluido en el ámbito del ciclo, cuyo valor se compara con un valor dado: WHILE i < n . De esa manera se controla cuantas veces se realiza el proceso. Una variante es emplear en la condición variables de tipo alfanumérico para controlar si se sigue ejecutando el ciclo iterativo.

También, bajo ciertas circunstancias, el ciclo WHILE puede ser equivalente al diagrama tradicional siguiente, en el que la condición se ha colocado al final:



CICLO TRADICIONAL CONTROLADO POR CONDICIÓN

Como ejemplo de aplicación, considere el diagrama de flujo que permite tabular 101 puntos de la función:

$$y = SIN(x) + x$$

en el intervalo de -3.1416 a 3.1416 radianes.

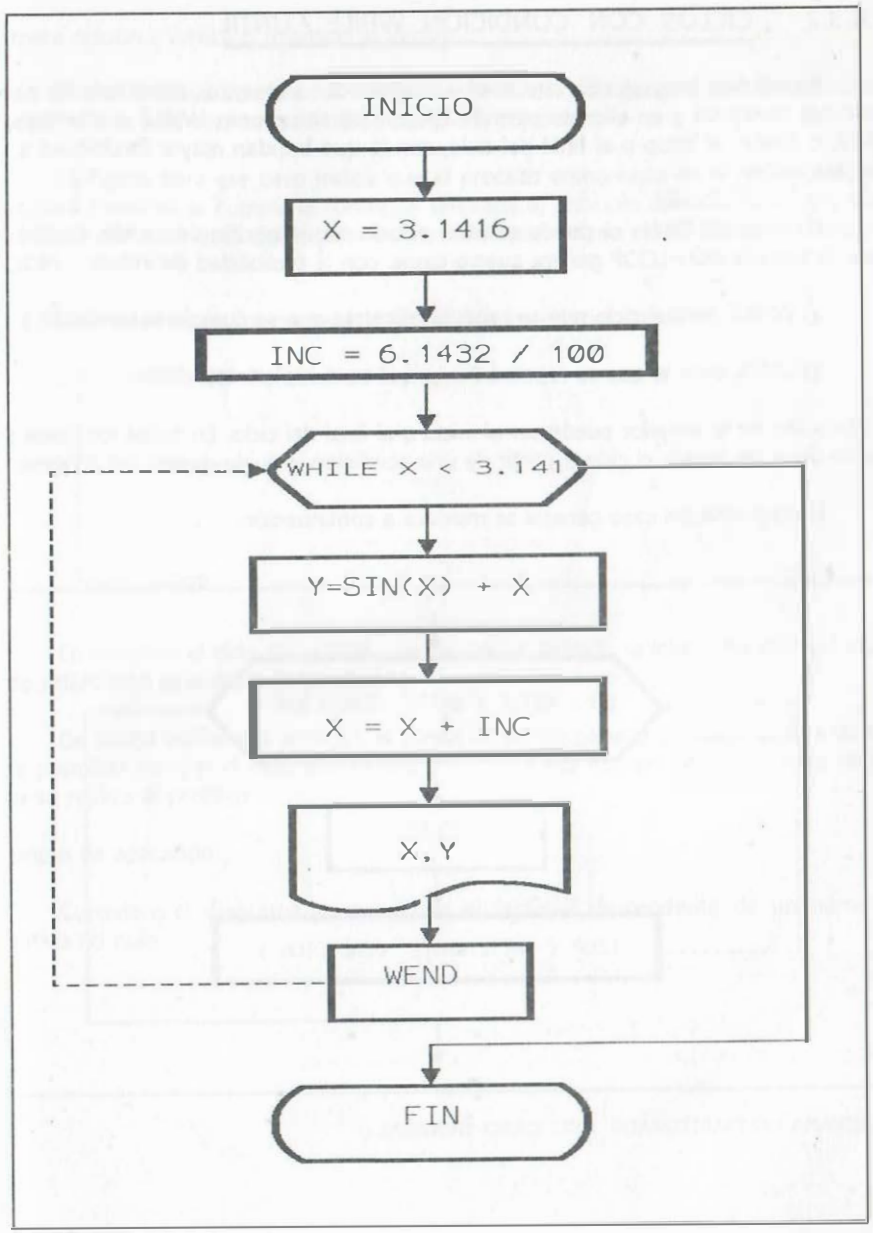


DIAGRAMA ESTRUCTURADO

7.8.3.2 CICLOS CON CONDICIÓN WHILE / UNTIL

En algunos lenguajes de alto nivel, se cuenta con elementos estructurados para los ciclos iterativos y en ellos se permite colocar las situaciones WHILE o mientras y UNTIL o hasta al inicio o al final del ciclo, con lo que brindan mayor flexibilidad a la programación.

En lenguaje C sólo se puede emplear el ciclo con la opción do - while. En Quick Basic la sintaxis DO - LOOP genera cuatro casos, con la posibilidad de indicar:

a) WHILE para el ciclo que se repetirá mientras que se cumpla la condición y

b) UNTIL para el que se repetirá hasta que se cumpla la condición

la ubicación de lo anterior puede ser al inicio o al final del ciclo. En todos los casos se puede dejar de repetir el ciclo a partir de una condición incluida dentro del proceso.

El diagrama del caso general se muestra a continuación:

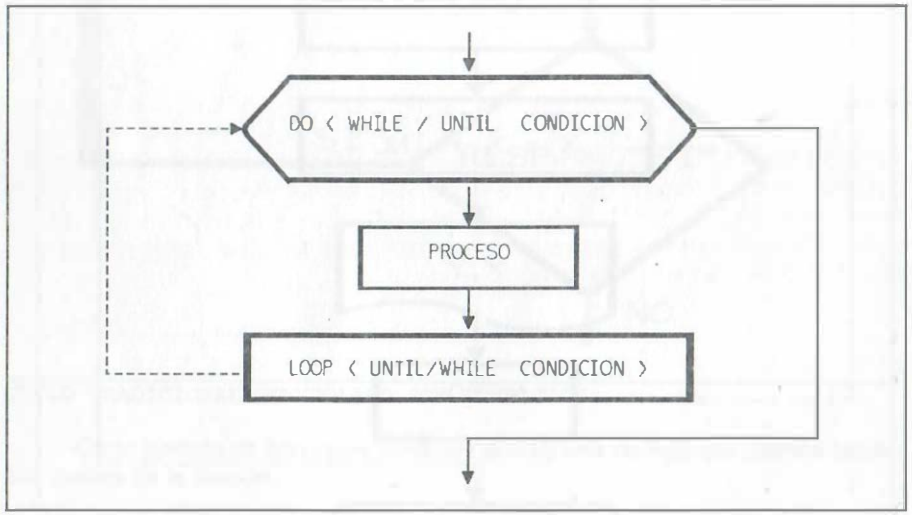


DIAGRAMA ESTRUCTURADO DEL CASO GENERAL

Primera opción (WHILE o mientras al inicio)

El ciclo iterativo con la opción WHILE al inicio del ciclo permite que el ciclo no se ejecute ni una sola vez, cuando no se cumple la condición.

La figura para ese caso indica que el proceso enmarcado en el rectángulo, se ejecutará mientras se cumpla la condición señalada al principio del ciclo iterativo y que el ámbito del ciclo está delimitado por la instrucción dada en el último rectángulo (LOOP).

Nuevamente, en forma reducida, se indica lo anterior como:

```
DO WHILE condición

PROCESO

LOOP
```

En resumen el ciclo DO LOOP , con la opción WHILE, se interpreta como: hacer ciclo MIENTRAS se cumpla la condición.

De forma similar a la anterior, la condición por lo general involucra dos variables que permiten romper el ciclo: DO WHILE i < n . De esa manera se controla las veces que se realiza el proceso.

Ejemplo de aplicación:

Considere el diagrama para calcular el factorial descendente de un número positivo no nulo.

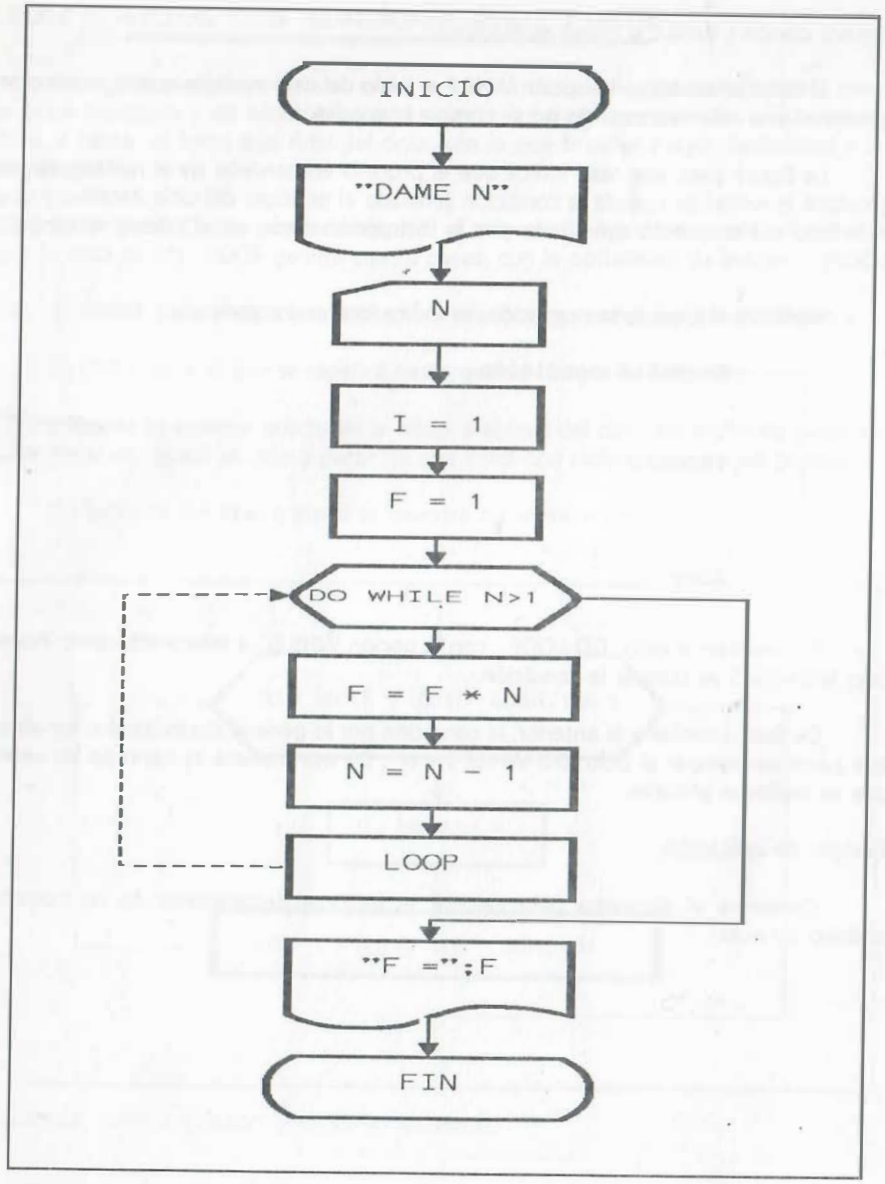


DIAGRAMA ESTRUCTURADO

Segunda opción (DO UNTIL-LOOP)

El ciclo iterativo con la opción UNTIL al inicio del ciclo se muestra a continuación:

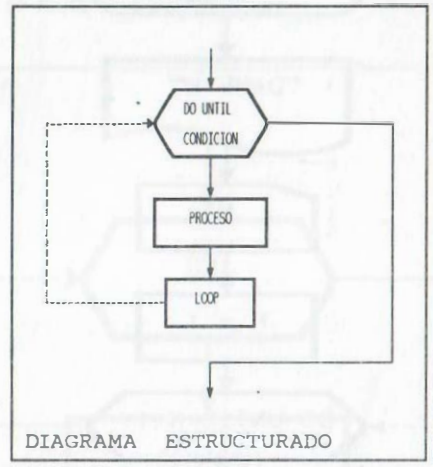


DIAGRAMA ESTRUCTURADO

La figura indica que el proceso enmarcado en el rectángulo se ejecutará hasta que se cumpla la condición señalada al principio del ciclo iterativo y que el ámbito del ciclo está delimitado por la instrucción dada en el último rectángulo (LOOP).

En forma reducida se indica lo anterior como:

```
DO UNTIL condición
PROCESO
LOOP
```

El ciclo DO LOOP, con la opción UNTIL, puede interpretarse como: hacer ciclo HASTA QUE se cumpla la condición.

De forma similar a la anterior, la condición por lo general involucra dos variables que permiten romper el ciclo como se sugirió para el caso anterior: DO UNTIL *i* = *n* (hacer el ciclo hasta que *i* sea igual a *n*), así se controla cuantas veces se realiza el proceso.

Veamos el caso del ejemplo anterior resuelto con UNTIL:

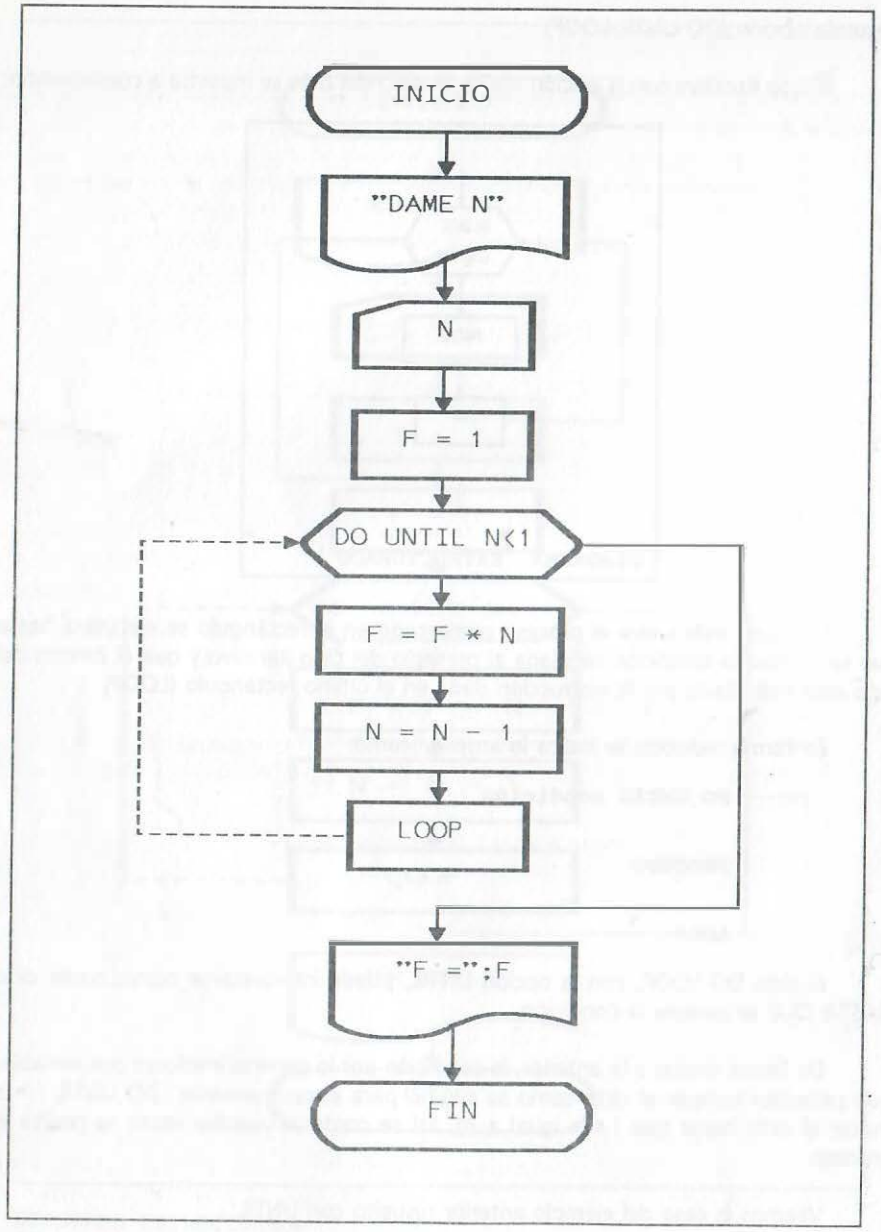


DIAGRAMA ESTRUCTURADO

Tercera opción (DO-LOOP WHILE)

La tercera forma simplificada para un ciclo iterativo es la opción WHILE al final del ciclo DO _ LOOP y se muestra a continuación:

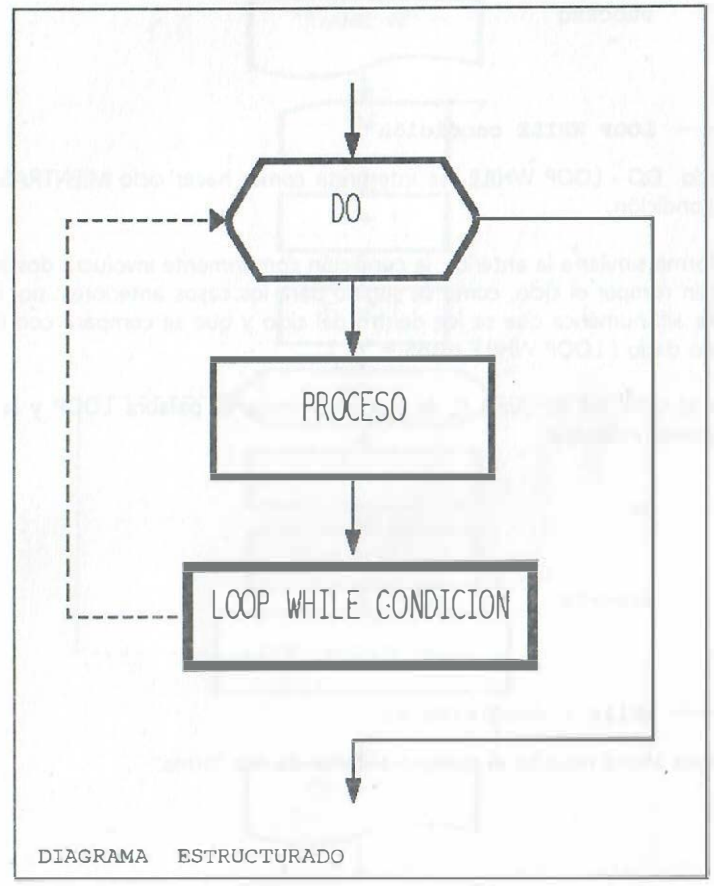


DIAGRAMA ESTRUCTURADO

La figura indica que el proceso enmarcado en el rectángulo se ejecutará al menos una vez y se repetirá mientras se cumpla la condición indicada al final del ciclo iterativo y que el ámbito del ciclo está delimitado precisamente por esa instrucción que involucra la condición (LOOP WHILE).

En forma reducida se indica lo anterior como:

```
DO

PROCESO

LOOP WHILE condición
```

El ciclo DO - LOOP WHILE se interpreta como: hacer ciclo MIENTRAS que se cumpla la condición.

De forma similar a la anterior, la condición comúnmente involucra dos variables que permiten romper el ciclo, como se sugirió para los casos anteriores: por ejemplo una variable alfanumérica que se lee dentro del ciclo y que se compara con un valor alfanumérico dado (LOOP WHILE resp$ = "n") .

Para el caso del lenguaje C, la sintaxis elimina la palabra LOOP y la opción abreviada puede indicarse:

```
do

proceso

while ( condición );
```

Veamos ahora resuelto el ejemplo anterior de esa forma:

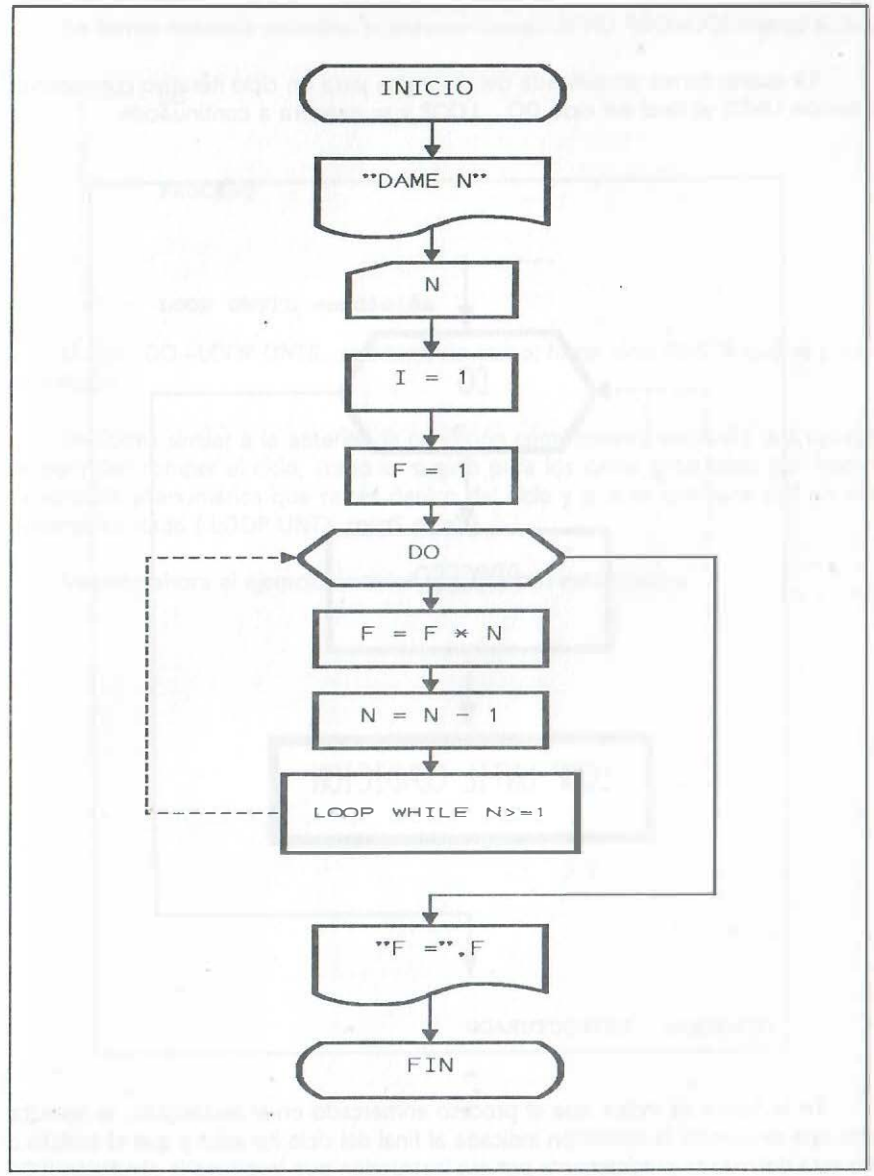


DIAGRAMA ESTRUCTURADO

Cuarta opción (DO-LOOP UNTIL)

La cuarta forma simplificada del diagrama para un ciclo iterativo corresponde a la opción UNTIL al final del ciclo DO _ LOOP y se muestra a continuación:

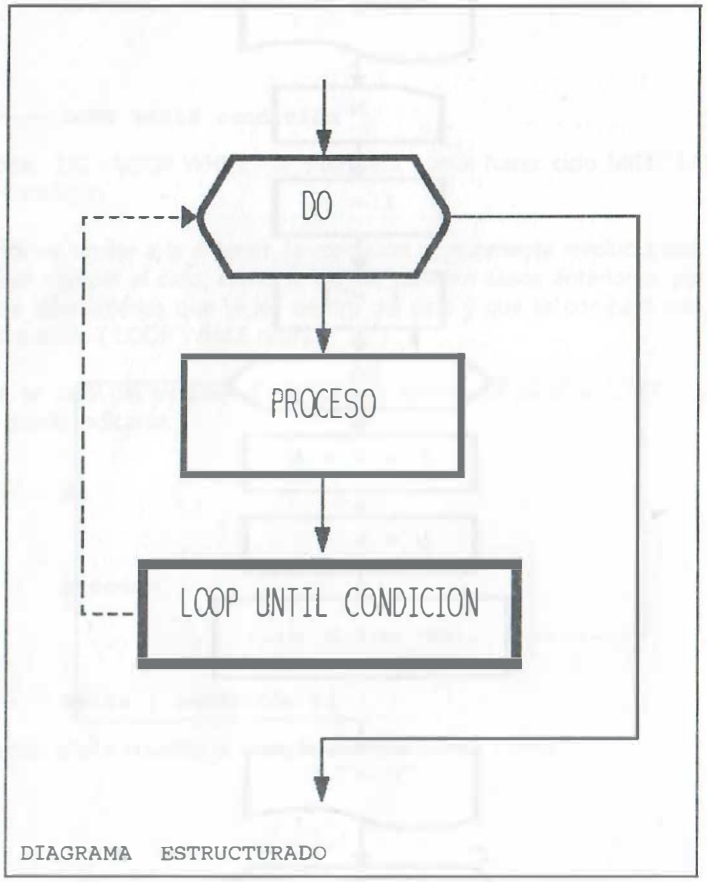


DIAGRAMA ESTRUCTURADO

En la figura se indica que el proceso enmarcado en el rectángulo, se ejecutará hasta que se cumpla la condición indicada al final del ciclo iterativo y que el ámbito del ciclo está delimitado precisamente por esa instrucción que involucra la condición (LOOP UNTIL).

En forma reducida se indica lo anterior como:

```
DO

PROCESO

LOOP UNTIL condición
```

El ciclo DO - LOOP UNTIL se interpreta como: hacer ciclo HASTA que se cumpla la condición.

De forma similar a la anterior, la condición comúnmente involucra dos variables que permiten romper el ciclo, como se sugirió para los casos anteriores: por ejemplo una variable alfanumérica que se lee dentro del ciclo y que se compara con un valor alfanumérico dado (LOOP UNTIL resp$ = "n") .

Veamos ahora el ejemplo anterior resuelto con esta opción:

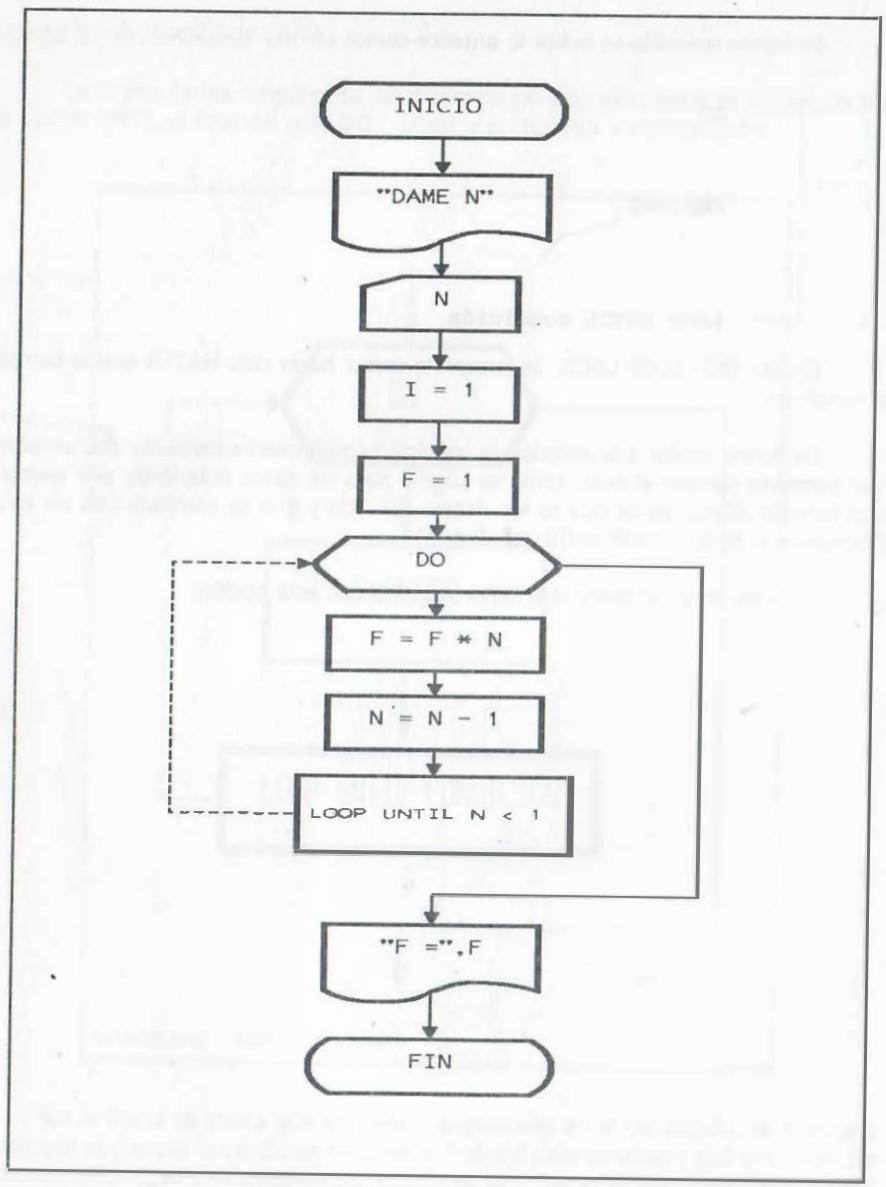


DIAGRAMA ESTRUCTURADO

7.8.4 DIAGRAMAS ESTRUCTURADOS PARA SELECCIÓN (RAMIFICACIONES O TRANSFERENCIAS)

7.8.4.1 DIAGRAMA ESTRUCTURADO IF - THEN - ELSE

Cuando el resultado de una pregunta o (IF) considera por un lado ejecutar un conjunto de acciones específicas y en caso contrario ejecutar otro conjunto de instrucciones, se emplea un diagrama que corresponde al elemento IF - THEN - ELSE en Quick Basic (THEN para el caso afirmativo ó verdadero y ELSE para el caso contrario) o el elemento if - else en el lenguaje C. La figura es:

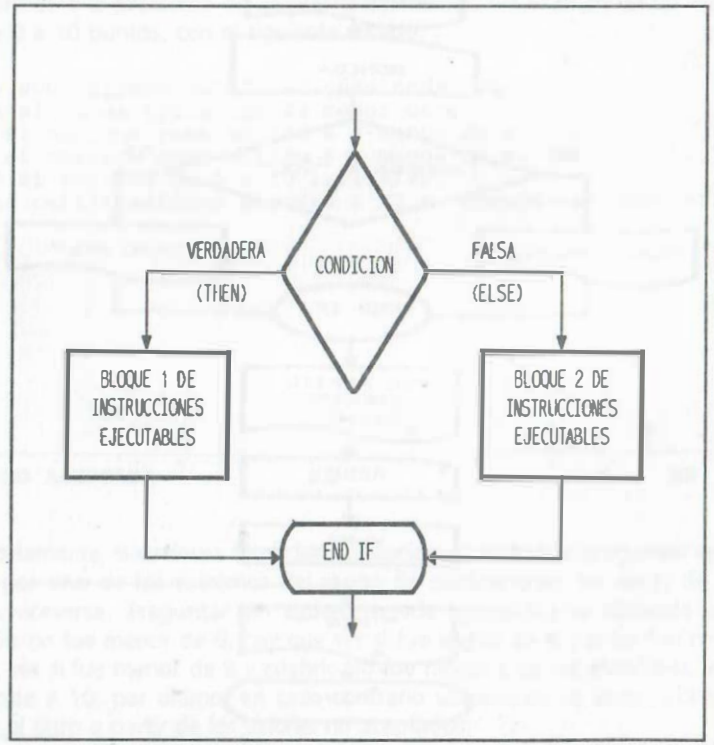


DIAGRAMA ESTRUCTURADO

Ejemplo de aplicación:

Un estudiante lanza al aire una moneda n veces, cada vez que cae águila se indica con un 1 y cuando cae sol se indica con un 2. Elabore el diagrama de flujo que muestre como salida en la pantalla el letrero "Águila" o el letrero "Sol", según el caso.

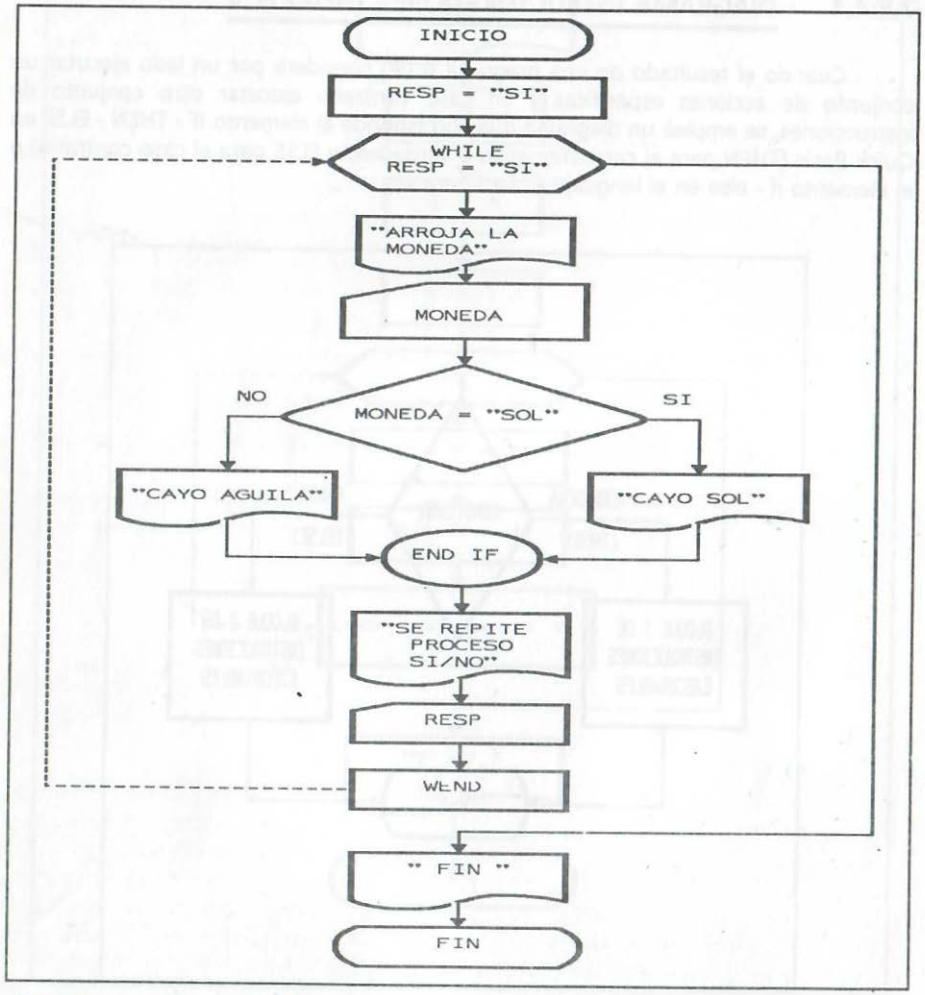


DIAGRAMA ESTRUCTURADO

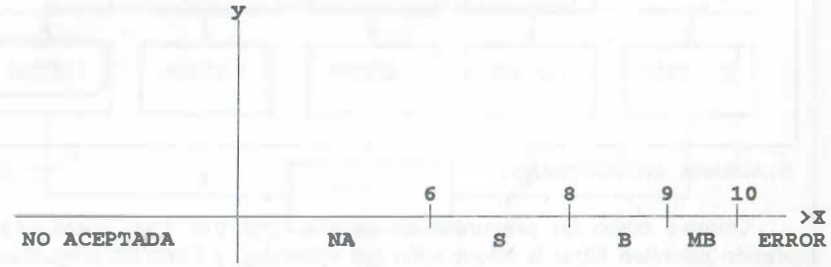
7.8.4.2 ELEMENTO ESTRUCTURADO DE RAMIFICACIÓN ENCADENADO:

IF - THEN - ELSE IF - THEN - ELSE IF ... ELSE

Cuando se presentan preguntas en serie, es decir, una tras otra sobre una expresión, se puede establecer categorías, intervalos o clases (de mayor a menor o viceversa), a manera de un filtro que separa en distintas clases los valores por la posición que ocupan en un rango de comparación para una expresión dada: precios, intervalos, clases, etc. La primera pregunta captura los datos de la primera categoría, la segunda establece el segundo intervalo con valores mayores y acotados por la condición establecida y así sucesivamente. Por esta razón, se dice que se tiene una estructura de preguntas encadenadas o de selección de clases. Veamos el diagrama del siguiente ejemplo:

Considere el problema de asignar la calificación del examen de un alumno, en escala de 0 a 10 puntos, con el siguiente criterio:

```
No son válidas calificaciones negativas.
NA si la calificación es menor de 6.
S si obtiene como mínimo 6 y menos de 8.
B si obtiene como mínimo 8 y menos de 9.
MB si obtiene de 9 a 10 inclusive.
Las calificaciones mayores a 10 se consideran como errores.
```



Obviamente, si se desea filtrar las calificaciones realizando preguntas en cadena, se inicia por uno de los extremos del rango de calificaciones, es decir, de menor a mayor o viceversa. Preguntar en cadena puede ejemplificarse diciendo que si la calificación no fue menor de 6, hay que ver si fue menor de 8 y si no fue ninguna de esas dos, ver si fue menor de 9 y cuando no fue ninguna de las anteriores, verificar si corresponde a 10; por último, en caso contrario ubicarla como error. Consideremos entonces el filtro a partir de los valores no aceptados:

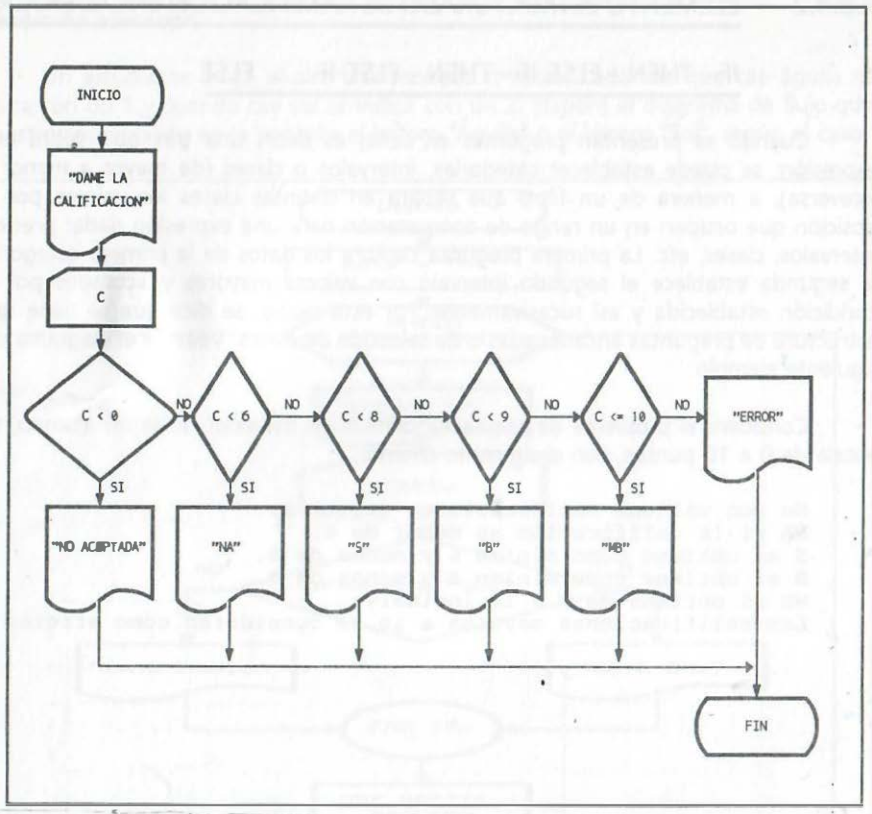


DIAGRAMA ESTRUCTURADO

Observe cómo las preguntas en cadena (una tras otra) sobre una misma expresión permiten filtrar la información por intervalos, y como las preguntas van de menor a mayor o viceversa, no hay necesidad de acotar ambos extremos y sólo se delimita uno de ellos.. De esta manera cada pregunta establece una magnitud límite para cada intervalo y establece un filtro para las preguntas encadenadas que restan (en este caso las preguntas previas capturan los valores menores y sólo dejan pasar los valores de los intervalos de las siguientes preguntas que actuarán de la misma manera hasta el fin del encadenamiento).

Tenga en cuenta que para el lenguaje C el encadenamiento se denominará úicamente if - else if - else if ... else

7.8.4.3 SELECCIÓN DE CASOS ESPECÍFICOS

El diagrama que indica selección de un caso específico de un conjunto de opciones dadas, se considera similar al caso de las preguntas encadenadas y corresponde a SELECT CASE y switch, del Quick Basic y lenguaje C respectivamente. En el diagrama se pregunta por una expresión única que se compara con ***N + 1*** opciones posibles (empezando de izquierda a derecha).

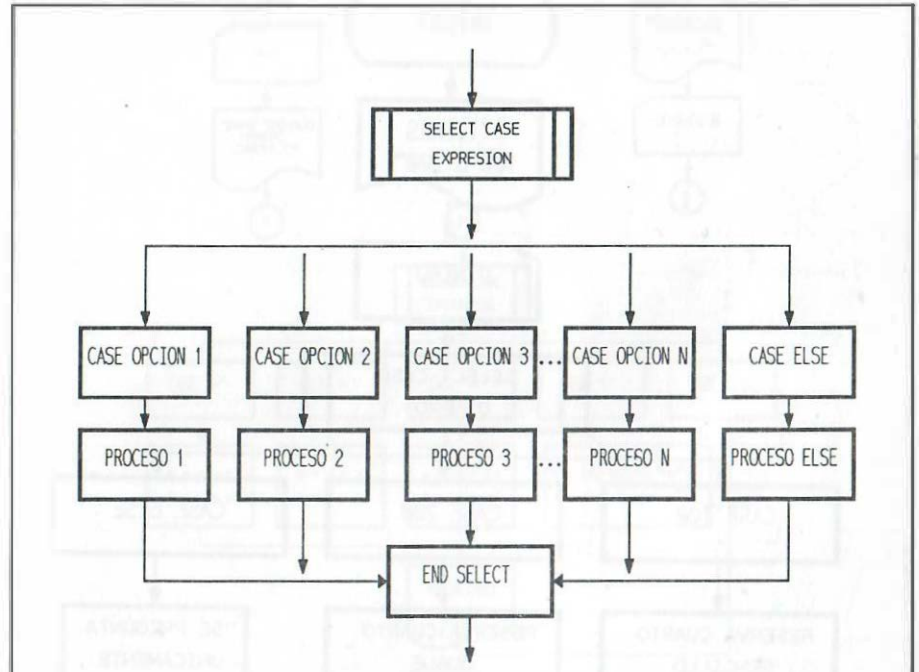


DIAGRAMA ESTRUCTURADO

En lenguaje C la expresión sólo se puede comparar con un valor dado, mientras que en Quick Basic las opciones son de la siguiente forma:

a) Expresiones separadas por comas.

b) Un intervalo que se delimita con: expresión TO expresión.

c) Una condición mediante: IS operador relacional expresión.

d) Combinaciones de las formas anteriores.

Debido a ello, se optó por ejemplificar diagramas para el caso del Quick Basic; como ejemplo de aplicación considere reservar un cuarto de hotel dependiendo del dinero que se tiene:

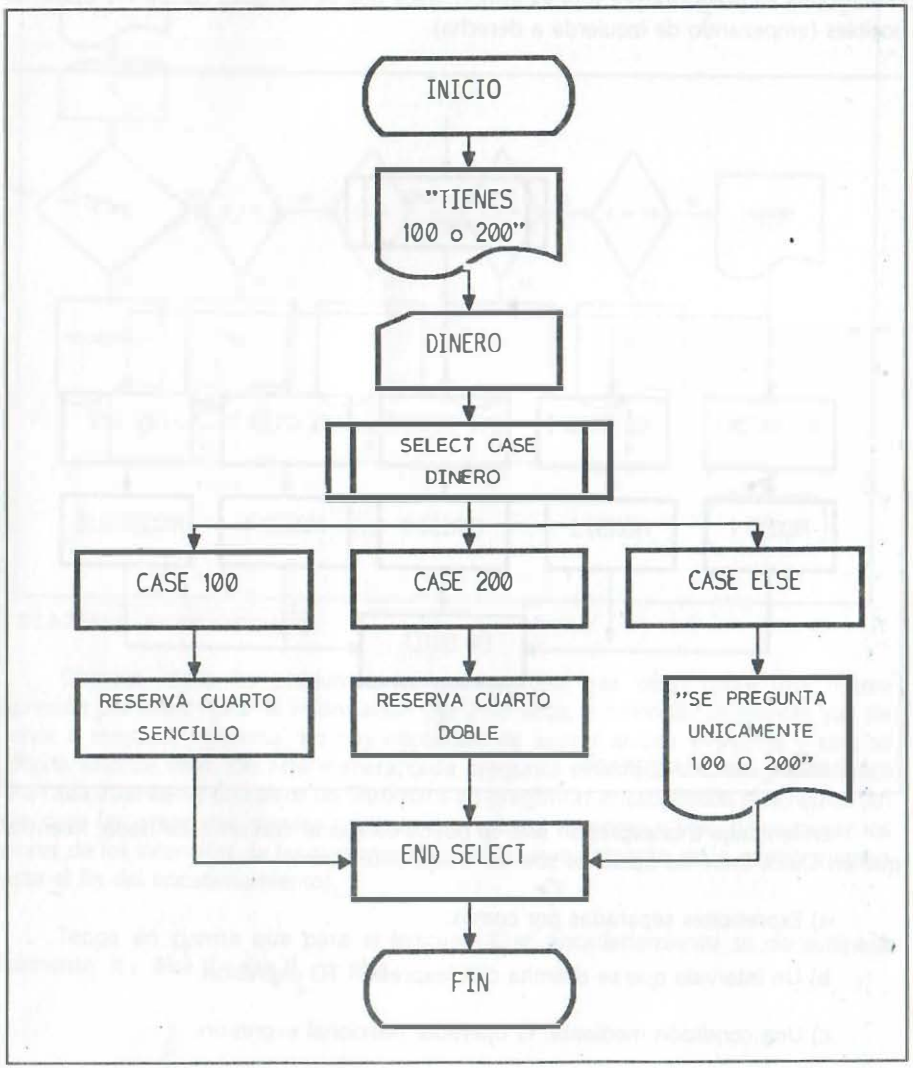


DIAGRAMA ESTRUCTURADO

Ejemplo para seleccionar operaciones con números complejos:

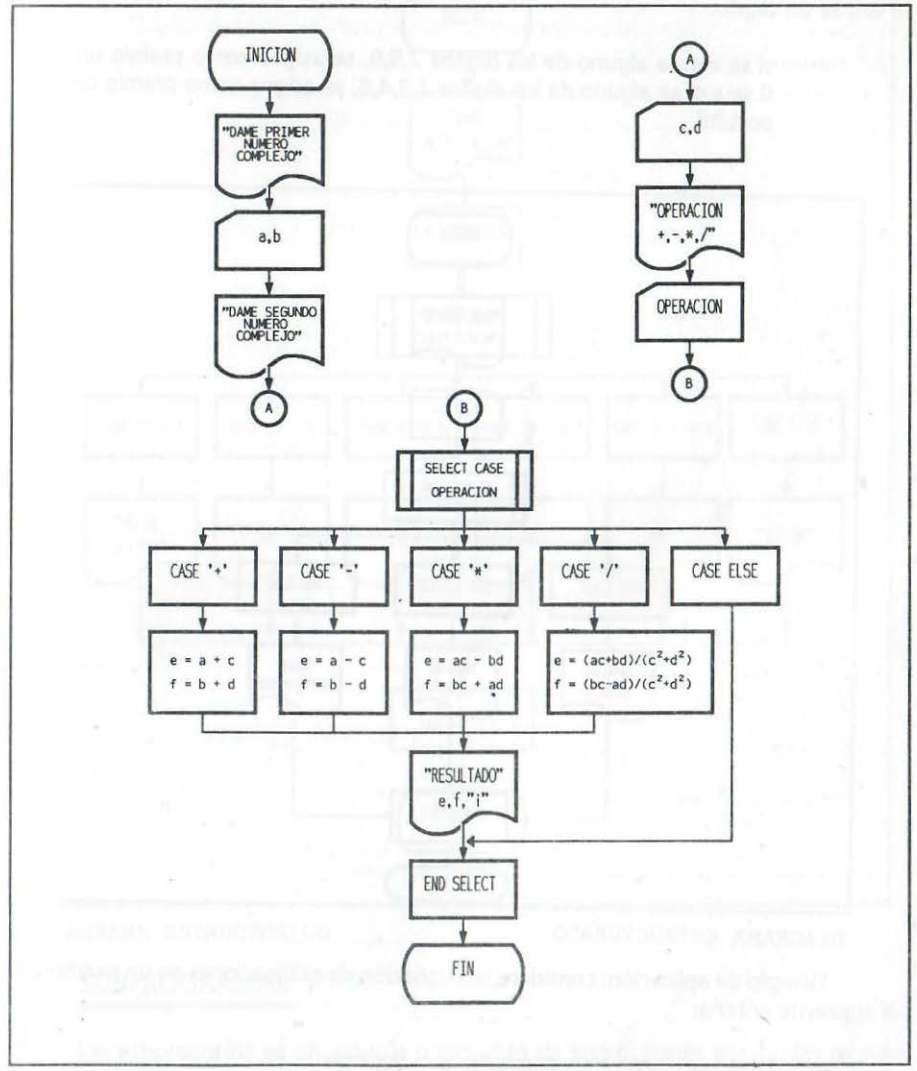


DIAGRAMA ESTRUCTURADO

Ejemplo de aplicación: considere la asignación de premios en un sorteo en donde se extrae un dígito:

—— si se extrae alguno de los dígitos 2,5,9, se asigna como premio un reloj.
—— si se extrae alguno de los dígitos 1,3,4,8, se asigna como premio un radio portátil.

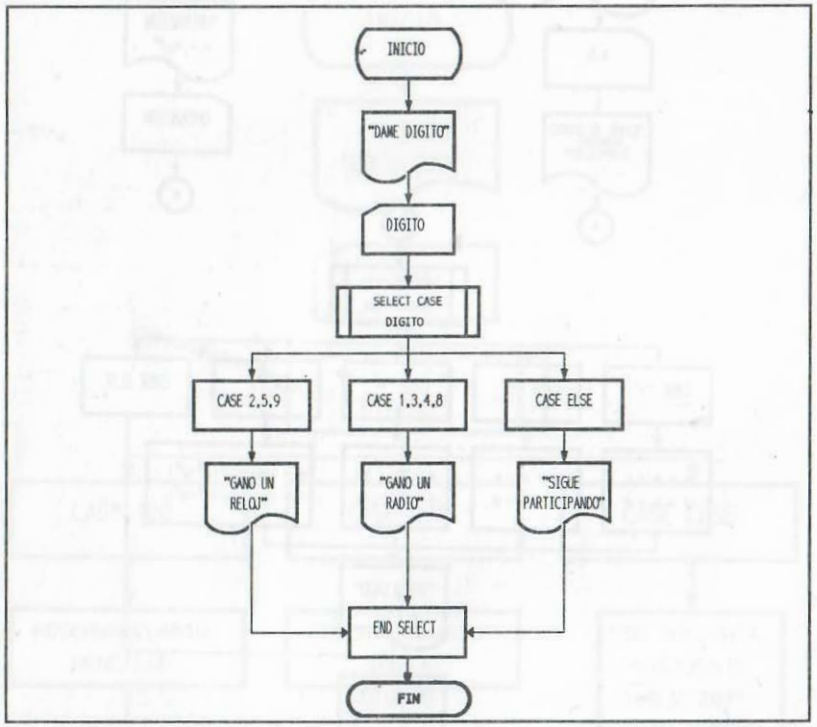


DIAGRAMA ESTRUCTURADO

Ejemplo de aplicación: considere la asignación de calificaciones en un examen con el siguiente criterio:

—— las fracciones más pequeñas son décimas.
—— si se obtiene un puntaje menor de 6 se asigna NA
—— si se obtiene un puntaje de 6.0 a 8 se asigna S
—— si se obtiene un puntaje menor de 9.3 se asigna B
—— si se obtiene un puntaje de 9.3 a 10 se asigna MB

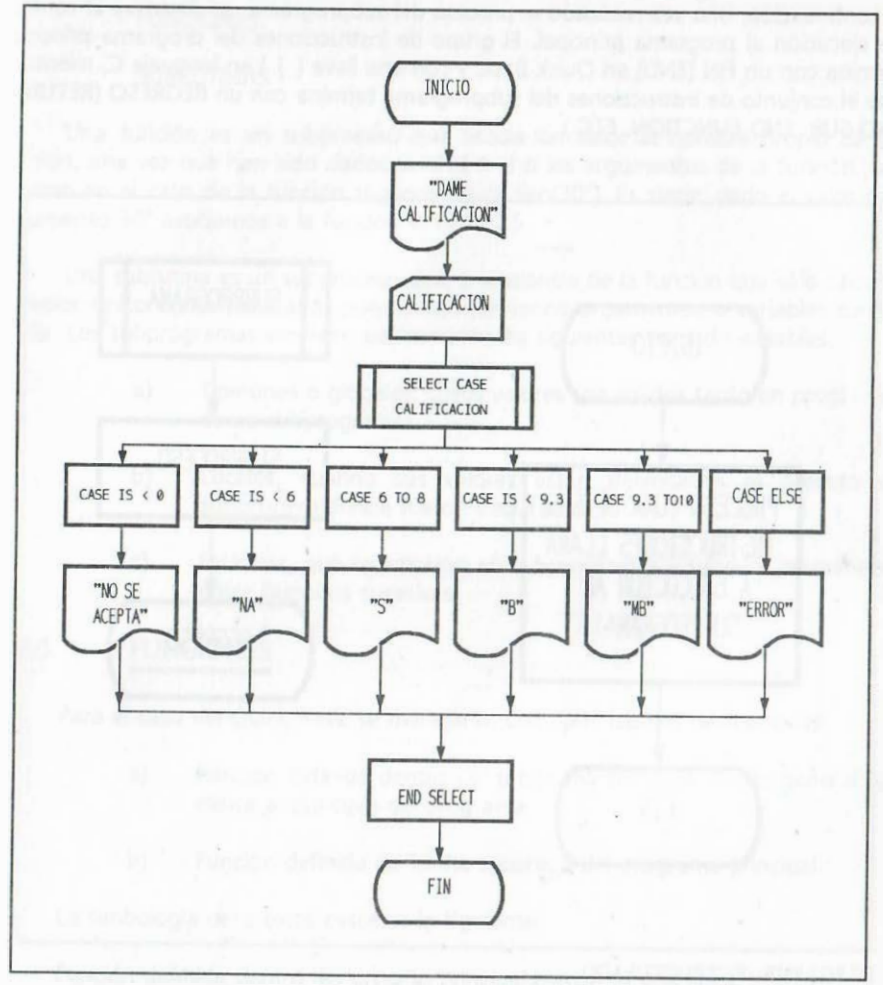


DIAGRAMA ESTRUCTURADO

7.9 SUBPROGRAMAS

Un subprograma es un módulo o conjunto de instrucciones agrupadas que por lo general se consideran separadas del programa o procedimiento principal. La ejecución de dichas instrucciones se realiza mediante un llamado del subprograma en algún lugar del programa principal, con lo cual el control de ejecución se transfiere al subprograma.

A continuación, una vez realizado el proceso del subprograma, se devuelve el control de ejecución al programa principal. El grupo de instrucciones del programa principal termina con un FIN (END) en Quick Basic y con una llave (}) en lenguaje C, mientras que el conjunto de instrucciones del subprograma termina con un REGRESO (RETURN, END SUB, END FUNCTION, ETC.).

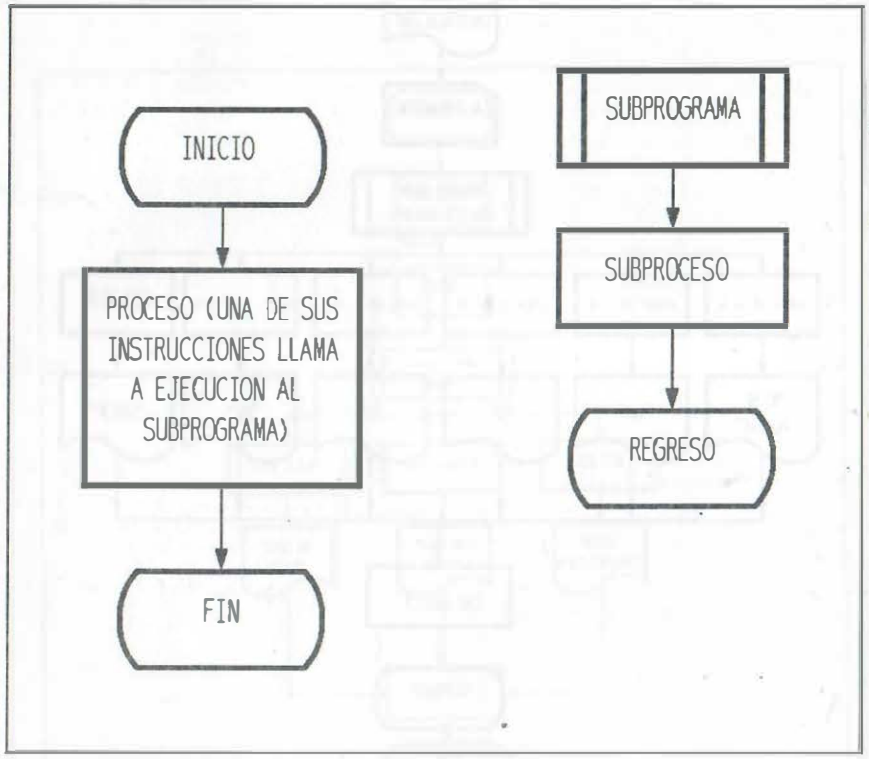


DIAGRAMA ESTRUCTURADO

Por lo tanto los subprogramas facilitan la programación al dividir un problema extenso en pequeñas partes, además, si el programa requiere repetir en varios lugares una serie de instrucciones, es conveniente definirlas dentro de un subprograma y llamarlo a ejecución donde se necesite.

Dependiendo del lenguaje de programación, hay distintos nombres para los subprogramas, sin embargo para el tema de diagramas de flujo consideraremos sólo dos:

FUNCIONES

SUBRUTINAS

Una función es un subproceso que asocia un valor al nombre propio de la función, una vez que han sido dados valores a el o los argumentos de la función, tal y como en el caso de la función trigonométrica sen(30°). Es decir, dado el valor del argumento 30° asociamos a la función el valor 0.5.

Una subrutina es un subproceso que, a diferencia de la función que sólo calcula un valor único como resultado, puede manejar varios argumentos o variables como salida. Los subprogramas emplean básicamente los siguientes tipos de variables:

a) Comunes o globales, cuyos valores son válidos tanto en programa como subprogramas.

b) Locales, cuando sus valores están delimitados al proceso o subproceso donde fueron declaradas.

c) Estáticas, que se emplean en subprogramas y su valor permanece entre llamadas sucesivas.

7.9.1 FUNCIONES

Para el caso del Quick Basic se manejarán dos tipos básicos de funciones:

a) Función definida dentro del programa principal. Por lo general se indica al principio del programa.

b) Función definida de forma separada del programa principal.

La simbología para estos casos es la siguiente:

1) Función definida dentro del proceso principal mediante una línea

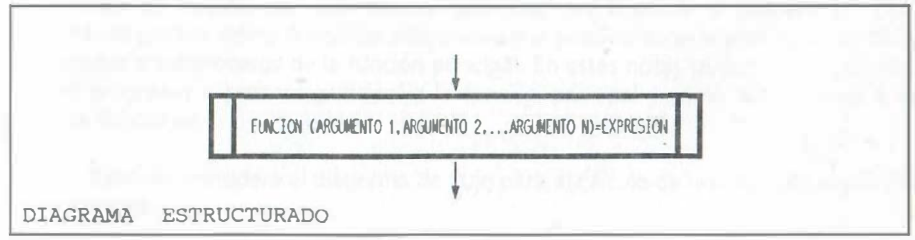


DIAGRAMA ESTRUCTURADO

2) Función definida dentro del proceso principal mediante varias líneas

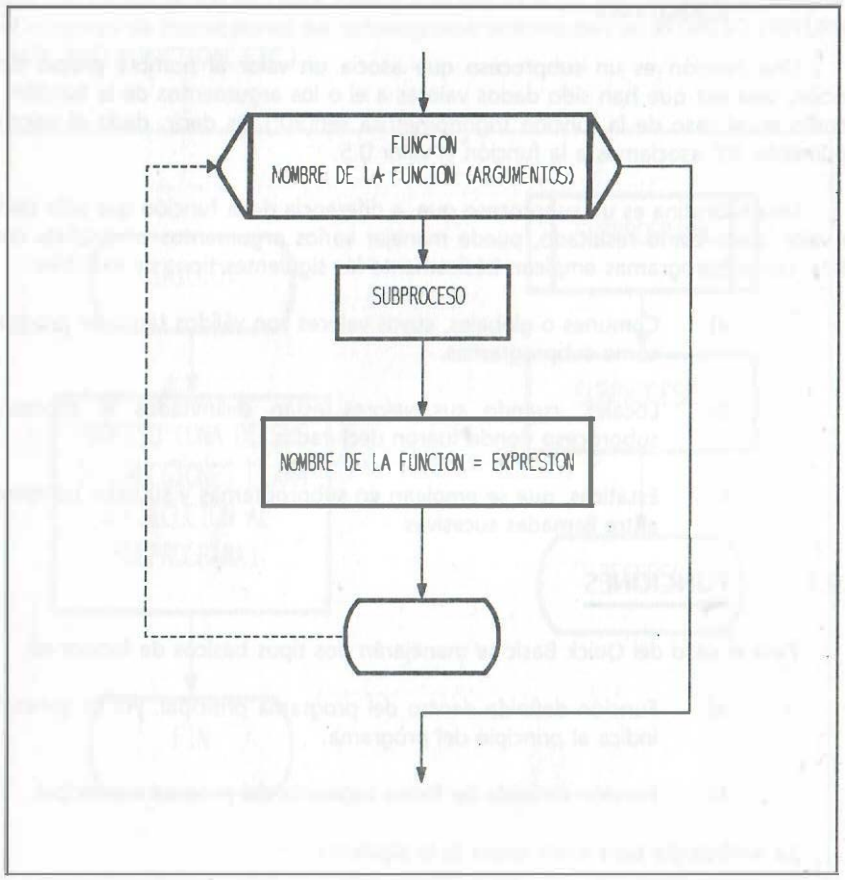


DIAGRAMA ESTRUCTURADO

3) Función separada del proceso principal

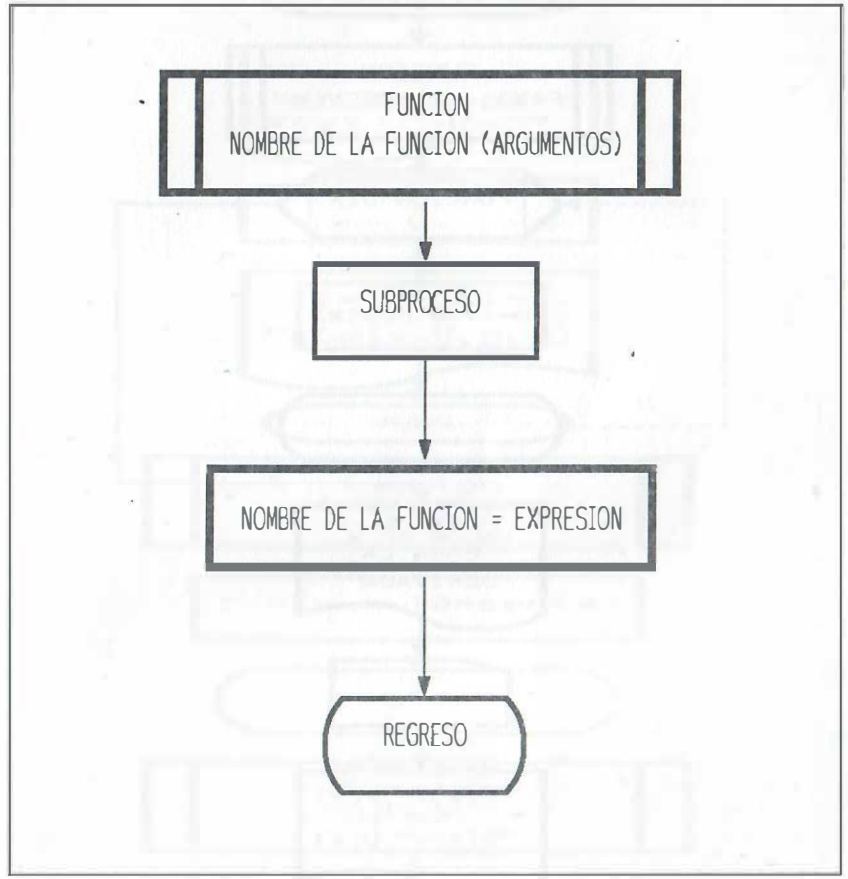


DIAGRAMA ESTRUCTURADO

En el caso del lenguaje C los programas se definen exclusivamente por medio de funciones. Se cuenta con una función principal que equivale al proceso principal; además es posible definir funciones adicionales que pueden considerarse como módulos separados o subprocesos de la función principal. En estas notas se optó por considerar como programa o proceso principal a la función principal y como subprocesos a las demás funciones.

Ejemplo: considere el diagrama de flujo para el cálculo de una función algebraica y su derivada.

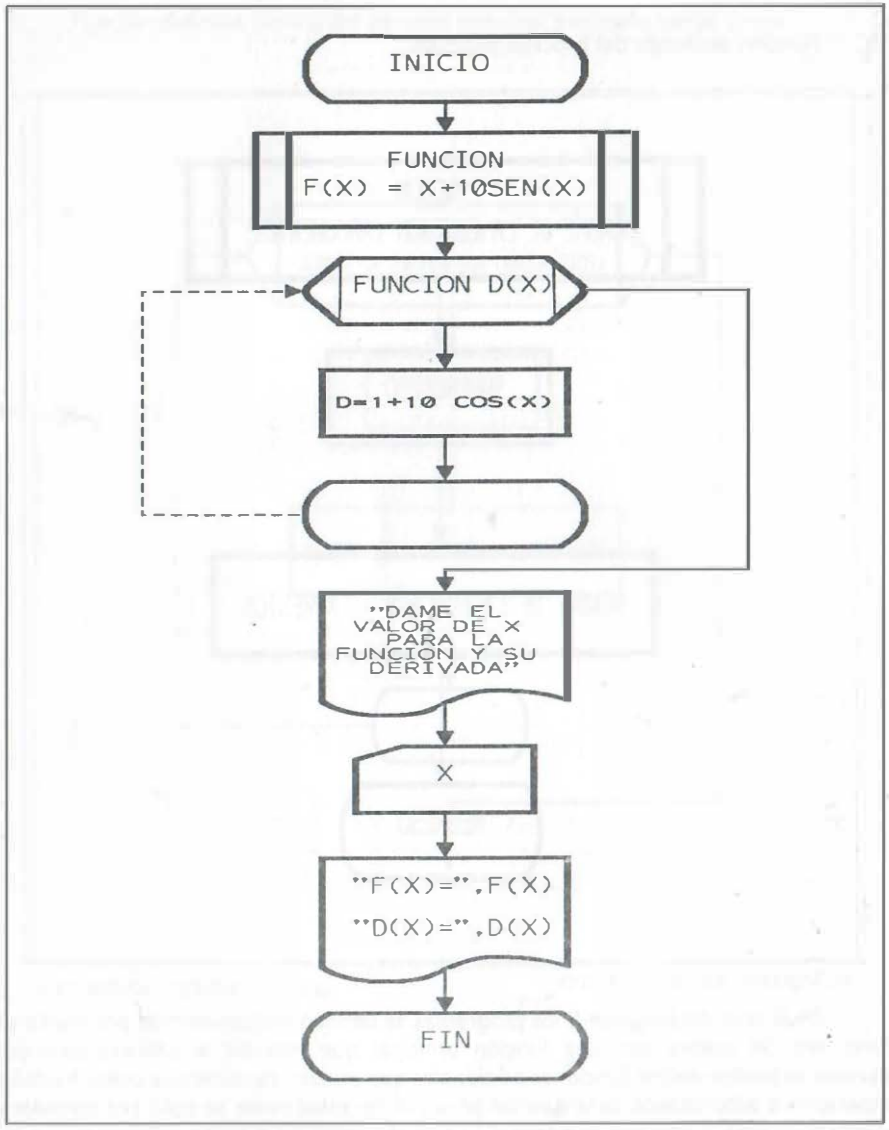


DIAGRAMA TÍPICO PARA QUICK BASIC, QUE EMPLEA FUNCIONES DE UNA Y VARIAS LÍNEAS EN EL MISMO PROGRAMA

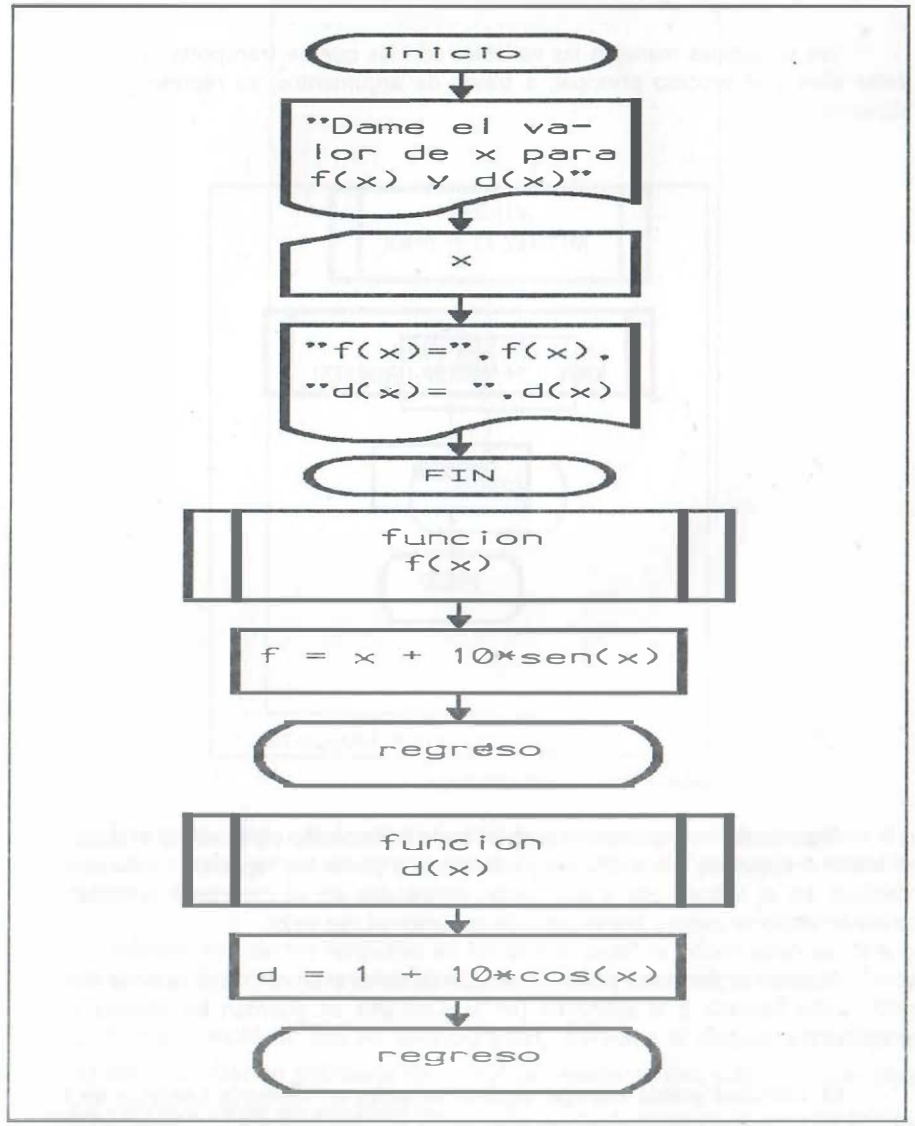


DIAGRAMA TIPICO PARA LENGUAJE C, EMPLEANDO FUNCIONES SEPARADAS. EL VALOR CALCULADO DEBE ASOCIARSE DIRECTAMENTE A LA INSTRUCCIÓN RETURN EN LUGAR DE ASIGNARSE AL NOMBRE DE LA FUNCIÓN

7.9.2 SUBRUTINAS

Las subrutinas manejan las variables con las que se transporta la información entre ellas y el proceso principal, a través de argumentos; su representación es la siguiente:

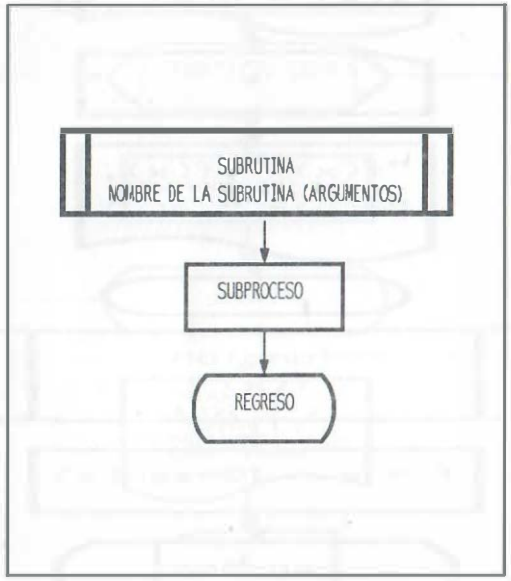


DIAGRAMA ESTRUCTURADO

Algunos de los argumentos proporcionan información para realizar el subproceso al llamar a ejecución a la subrutina y los demás argumentos regresarán valores que se calculan en el subproceso y que serán empleados en el programa principal. Este procedimiento se conoce como paso de parámetros por valor.

Algunos argumentos pueden manejar variables estáticas, cuyo valor se conserva entre cada llamada a la subrutina (en la subrutina se guardan los últimos valores empleados).

La subrutina puede manejar algunas variables en memoria común o de forma global, de tal manera que los valores que cambie el subproceso, automáticamente están cambiados para su uso en el programa principal y viceversa, sin necesidad de mencionarse como argumentos; esta opción se representa a continuación:

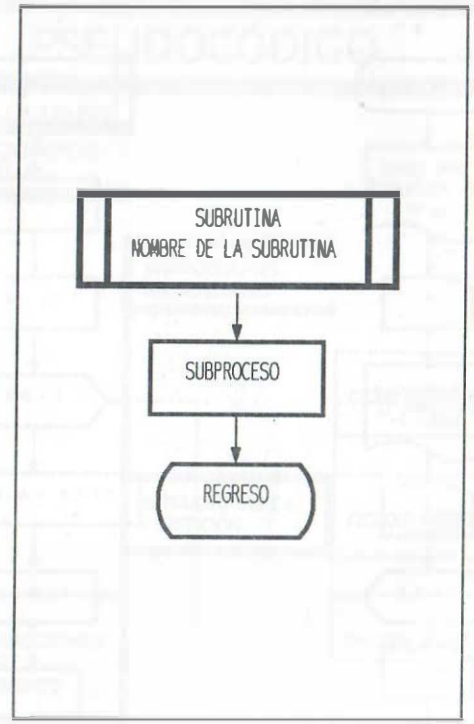


DIAGRAMA ESTRUCTURADO

En algunos lenguajes de programación existe cierto tipo de subrutina para el cual todas las variables están de forma global o común, y no se requiere mencionar argumento alguno.

Además hay ciertos lenguajes en los que el paso de información se hace con referencia a la dirección de las variables (paso de valores por referencia o direccción).

Como ejemplo de uso de subprogramas, considere el diagrama de flujo que indica cómo calcular un polinomio de grado n mediante una subrutina que envía y recibe valores a través de argumentos:

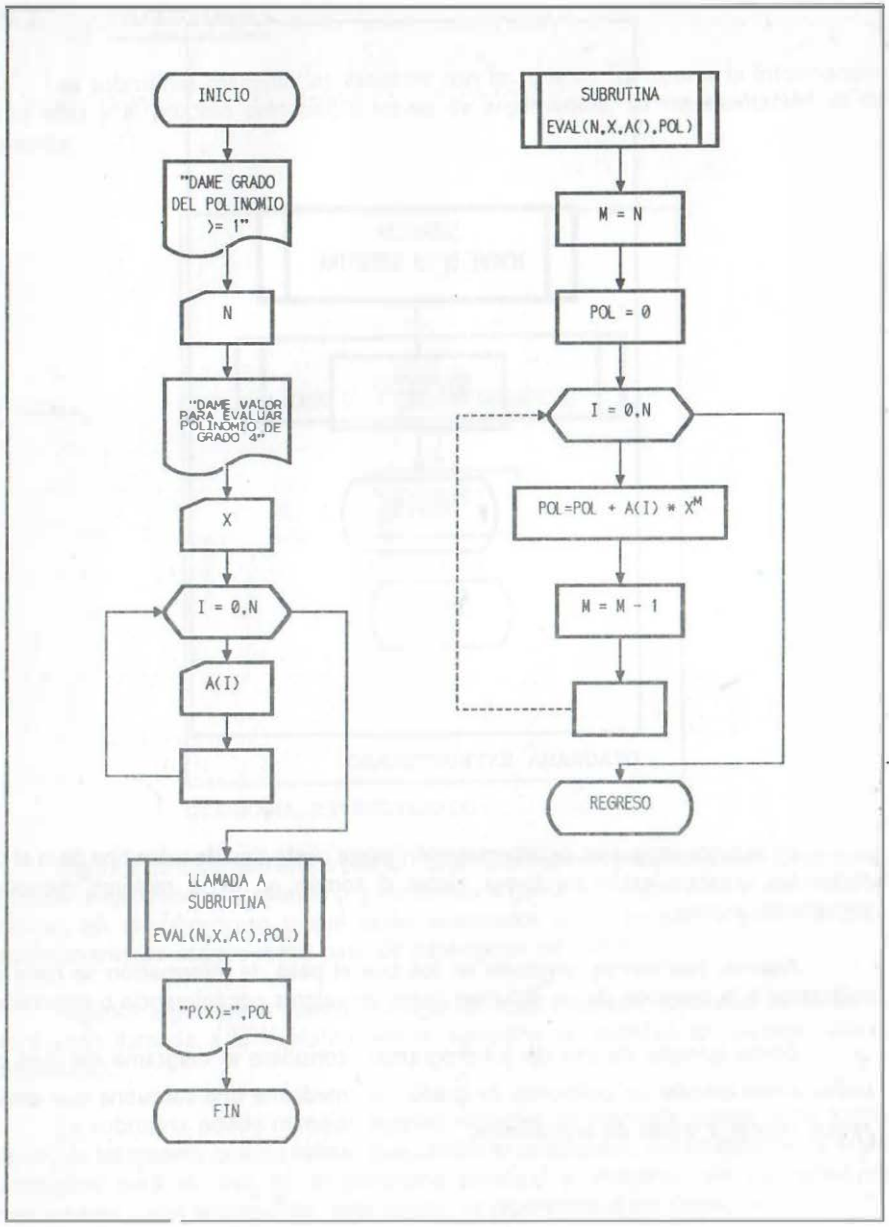


DIAGRAMA ESTRUCTURADO

PSEUDOCÓDIGO

- CARACTERÍSTICAS GENERALES
- EL PSEUDOCÓDIGO EMPLEA BÁSICAMENTE:
  - INSTRUCCIONES SECUENCIALES
  - ESTRUCTURAS DE REPETICIÓN
    - CICLO DESDE/PARA
    - CICLO(S) MIENTRAS/HASTA
    - CICLO(S) REPETIR
  - ESTRUCTURAS DE SELECCIÓN
    - PREGUNTA LÓGICA
    - PREGUNTAS ENCADENADAS
    - SELECCIÓN DE CASOS
  - SUBPROCESOS
    - FUNCIONES
    - SUBRUTINAS

PSEUDOCÓDIGO EVOLUCIONADO Y SIMPLIFICADO PARA COMUNICACIÓN CON SISTEMAS CON INTELIGENCIA ARTIFICIAL

8 PSEUDOCÓDIGO

8.1 CARACTERÍSTICAS GENERALES DEL PSEUDOCÓDIGO

Toda instrucción de lenguajes estructurados se expresa mediante órdenes escritas que corresponden a acciones indicadas en un lenguaje parecido a nuestro modo de hablar cotidiano (pseudocódigo). La idea del pseudocódigo fue originalmente desarrollada en inglés, sin embargo, en nuestro caso expondremos su equivalente en español (de uso común en nuestro medio), con palabras que indican las acciones por desarrollar, como leer, escribir, etc.

La representación de algoritmos mediante pseudocódigo y diagrama de flujo se considera útil desde la etapa de depuración del proceso, debido a que en ellos se facilita la prueba de escritorio y su modificación para corresponder de manera eficiente a la solución planteada, mediante instrucciones simplificadas de control comunes a cualquier lenguaje de alto nivel (programación estructurada), sin que importe la sintaxis o escritura específica de alguno de ellos. En esta etapa primaria lo importante es concentrarse en la lógica de solución, expresando ideas tal como las pensamos y decimos, y a continuación probándolas. Además, una vez afinada la lógica de solución, el planteamiento así expresado puede constituir parte de la documentación final del programa que se elabore.

De acuerdo con las consideraciones anteriores, el pseudocódigo resulta ser un lenguaje de especificación de algoritmos que representa toda acción ejecutable mediante palabras específicas (seleccionadas entre las palabras usadas en el ambiente de computación), que fácilmente tienen su equivalente en lenguajes como C, Cobol, Fortran 77, Quick Basic, Pascal, etc.

Todo algoritmo indicado mediante pseudocódigo incluirá las instrucciones: INICIO y FIN, que delimitan el ámbito del conjunto de acciones del algoritmo; todo grupo de instrucciones secuenciales usará cierta sangría a la izquierda, la cual se aumentará para las instrucciones dentro de los diagramas estructurados (CICLOS ITERATIVOS Y PREGUNTAS DE SELECCIÓN), por ejemplo el algoritmo para viajar en metro por la ciudad de México:

```
INICIO
        trasladarse a una estación del metro
        adquirir boleto
        abordar metro con dirección a la estación deseada
        viajar hasta llegar a la estación deseada
        descender del metro
        abandonar la estación
FIN
```

Los algoritmos de tipo numérico incluyen las declaraciones de tipo de variables

y sus posibles valores iniciales, antecediendo al algoritmo. Las instrucciones secuenciales de asignación se indicarán utilizando una flecha izquierda, que significa que el valor que se calcule para la expresión de la derecha se asigna a la variable indicada a la izquierda de dicha flecha. Por ejemplo considere el algoritmo para calcular la temperatura farenheit que corresponde a un valor centígrado dado:

```
Algoritmo: cálculo de temperatura en grados farenheit
C,F reales
INICIO
        escribir "Dame temperatura en grados centígrados"
        leer C
        F <— 1.8 * C + 32
        escribir "temperatura equivalente en farenheit ", F
FIN
```

A continuación se menciona el pseudocódigo para los elementos de repetición (ciclos iterativos) y de selección empleados en programación estructurada:

8.2 ELEMENTOS ESTRUCTURADOS DE REPETICIÓN (CICLOS)

8.2.1 CICLO DESDE O PARA (CONTROLADO POR CONTADOR)

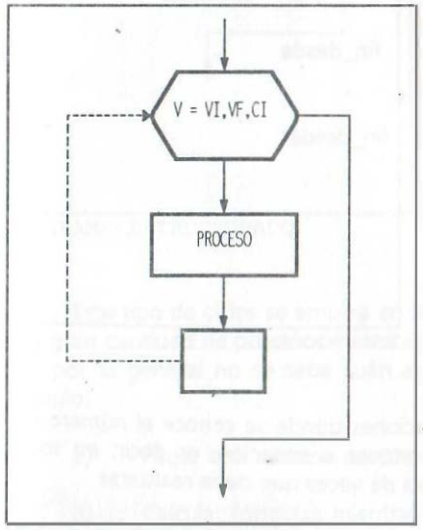


DIAGRAMA ESTRUCTURADO

```
desde v = vi hasta vf incrementar ci

        proceso
        (instrucciones ejecutables)

fin_desde
```

PSEUDOCÓDIGO

En el caso en el que el incremento sea unitario, podrá omitirse la palabra incrementar y la variable o valor del incremento (en este caso ci).

8.2.1.1 CICLOS ANIDADOS (CON CONTADOR INCLUIDO)

MANEJO DE ARREGLOS DE DOS DIMENSIONES

Por ejemplo, la lectura de un arreglo matricial por renglones:

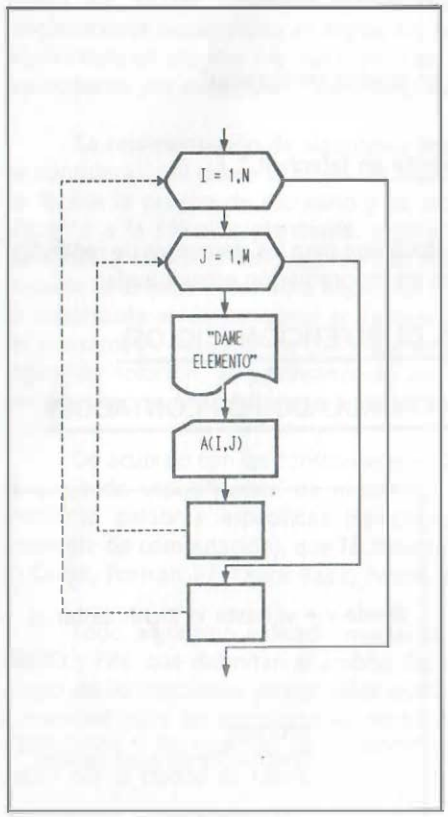


DIAGRAMA ESTRUCTURADO

```
desde I = 1 hasta N incrementar 1
 desde J = 1 hasta M incrementar 1
                escribir "Dame elemento"
                leer A(I,J)
 fin_desde
fin_desde
```

PSEUDOCÓDIGO

Este tipo de ciclos son útiles en situaciones donde se conoce el número de veces que se repite cierto procedimiento, lectura o situación, es decir, en todo aquello que sabemos exactamente el número de veces que debe realizarse.

8.2.2 CICLO CONTROLADO POR CONDICIÓN AL INICIO: WHILE

Este tipo de ciclo sólo se ejecuta **MIENTRAS** se cumple la condición colocada al inicio del ciclo:

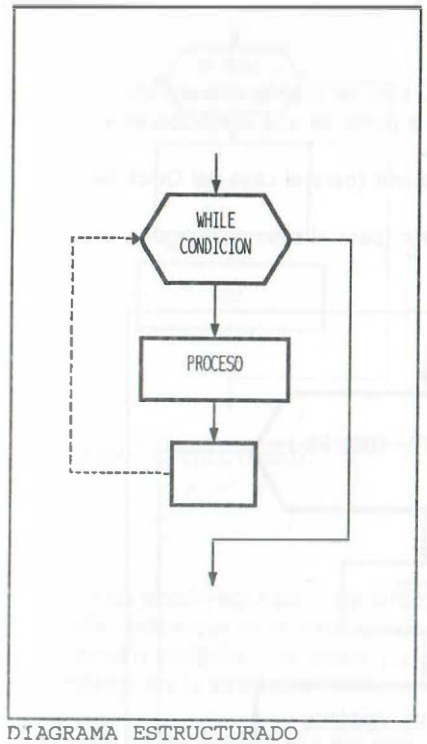


DIAGRAMA ESTRUCTURADO

```
MIENTRAS <condición>
PROCESO
FIN_MIENTRAS
```

PSEUDOCÓDIGO

Este tipo de ciclos se emplea en situaciones en las cuales es necesario repetir una gran cantidad de procesos mientras se cumple una condición dada y el caso es que por lo general no se sabe cuántas veces hay que repetir dicho proceso, por ejemplo:

a) Pagar una nómina mientras se tenga registro de horas trabajadas.

b) Calcular fórmulas mientras se está en cierto intervalo.

8.3 CICLOS REPETIR

En Quick Basic este tipo de ciclo iterativo corresponde a la sintaxis DO - LOOP y puede incluir:

a) la situación WHILE (mientras_que) o

b) la situación UNTIL o (hasta_que),

al inicio o al final del ciclo; en lenguaje C sólo se cuenta con el ciclo do - while. Además es posible dejar de repetir el ciclo a partir de una condición en el proceso:

c) condición EXIT DO o salida_de_repetir (para el caso del Quick Basic)

d) condición break o salida_de_repetir (para el caso del lenguaje C)

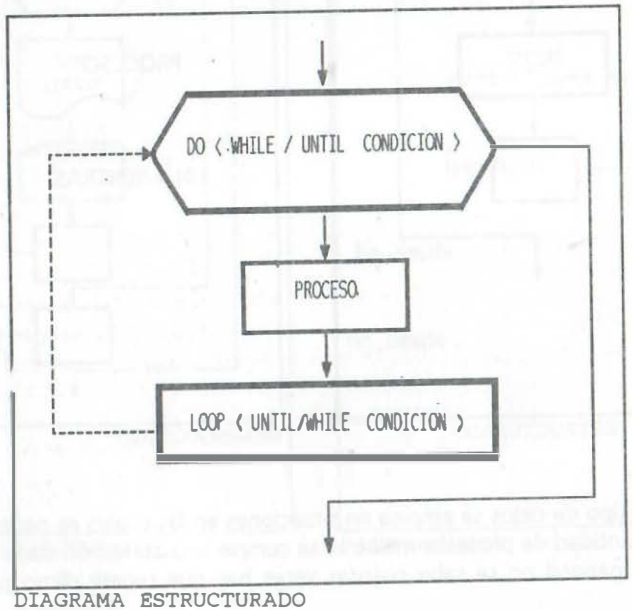


DIAGRAMA ESTRUCTURADO

De esta instrucción se generan cuatro casos particulares para el lenguaje Quick Basic para los cuales en forma resumida, se indicará su pseudocódigo a continuación:

8.3.1 PRIMERA OPCIÓN (DO WHILE-LOOP O REPETIR MIENTRAS_QUE)

El ciclo iterativo con condición al inicio del ciclo se muestra a continuación:

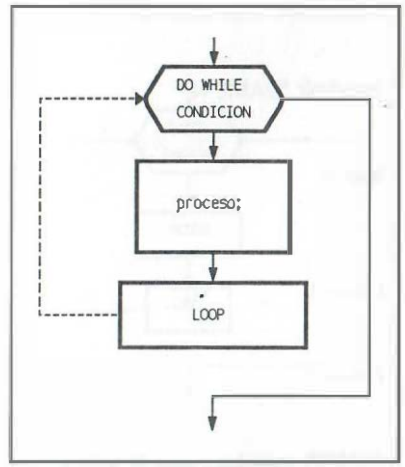


DIAGRAMA ESTRUCTURADO

```
REPETIR mientras_que condición
    PROCESO
FIN_REPETIR
```

PSEUDOCÓDIGO

Esta opción equivale a la instrucción MIENTRAS, vista anteriormente y, al igual que ella, indica que el proceso enmarcado en el rectángulo, se ejecutará mientras se cumpla la condición indicada al principio del ciclo iterativo; el ámbito del ciclo está delimitado por la instrucción dada en el último rectángulo (LOOP o FIN_REPETIR).

De forma similar a lo antes expuesto, la condición por lo general involucra dos variables que permiten romper el ciclo: REPETIR mientras_que i < n . De esa manera se puede controlar las veces que se realiza el proceso.

Ejemplo de aplicación:

Considere el pseudocódigo del diagrama para calcular el factorial descendente de un número positivo no nulo.

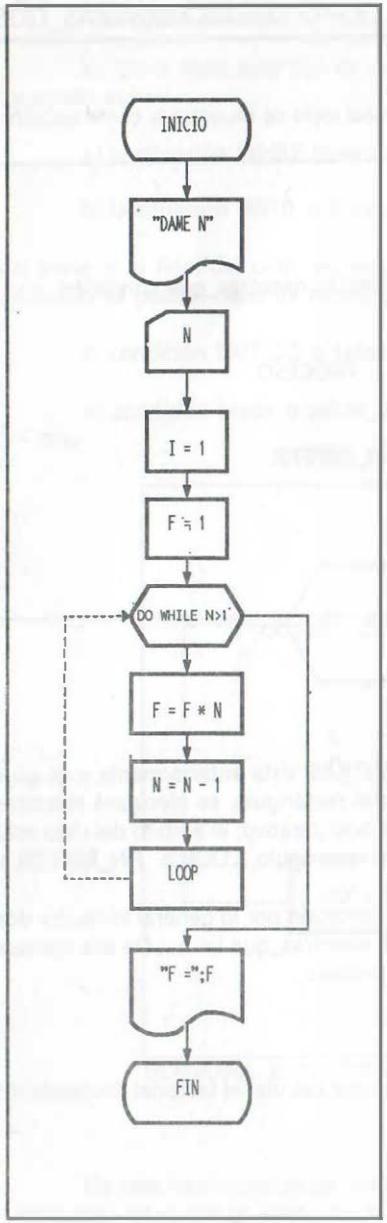


DIAGRAMA ESTRUCTURADO

```
INICIO

   imprimir "DAME N "

   leer N

   I <--- 1

   F <--- 1

   REPETIR mientras_que N > 1

      F <---- F * N

      N <---- N - 1

   FIN_REPETIR

   escribir "F = ", F

FIN
```

PSEUDOCÓDIGO

8.3.2 SEGUNDA OPCIÓN (DO UNTIL-LOOP) O REPETIR HASTA_QUE

Este ciclo iterativo con condición al inicio del ciclo se muestra a continuación:

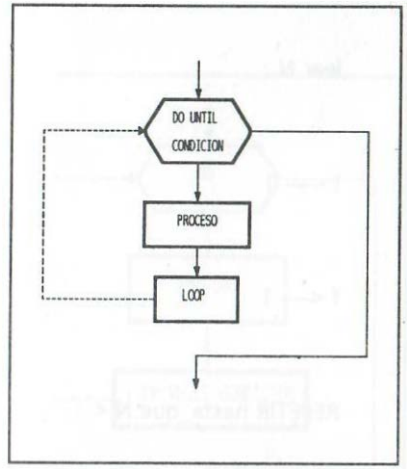


DIAGRAMA ESTRUCTURADO

```
REPETIR hasta_que condición

   PROCESO

FIN_REPETIR
```

PSEUDOCÓDIGO

En este tipo de ciclo el proceso indicado en el recuadro interno se ejecutará hasta que se cumpla la condición indicada al principio del ciclo iterativo y el ámbito del ciclo está delimitado por la última instrucción dada (LOOP o FIN_REPETIR).

En resumen este ciclo REPETIR, con la opción hasta, se interpreta como: hacer ciclo hasta que se cumpla la condición.

De forma similar a la anterior, la condición por lo general involucra dos variables que permiten romper el ciclo, como se sugirió para el caso anterior: REPETIR hasta_que i = n (hacer el ciclo hasta que i sea igual a n), así se controla las veces que se realiza el proceso.

Caso del ejemplo anterior resuelto con UNTIL:

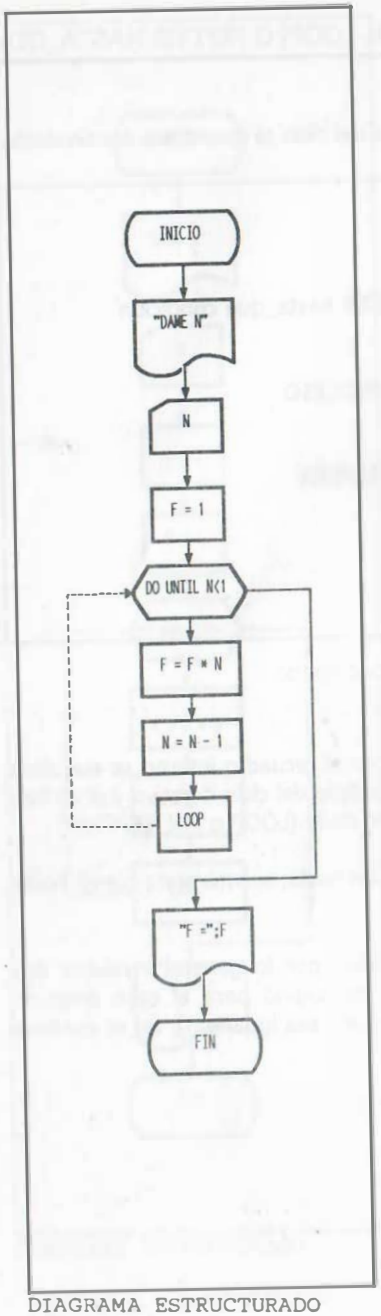


DIAGRAMA ESTRUCTURADO

```
INICIO

  leer N

  I <— 1

  f <— 1

  REPETIR hasta_que N < 1

    F <— F * N

    N <— N - 1

  FIN_REPETIR

  escribir "F = ", F

FIN
```

PSEUDOCÓDIGO

8.3.3 TERCERA OPCIÓN (DO-LOOP WHILE)

La tercera forma simplificada para un ciclo iterativo es la opción WHILE al final del ciclo DO _ LOOP y se muestra a continuación:

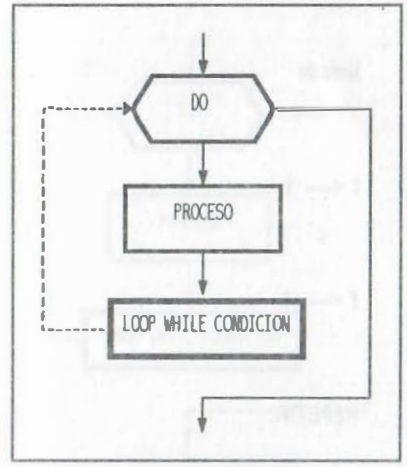


DIAGRAMA ESTRUCTURADO

```
REPETIR

    PROCESO

mientras_que  <condición>
```

PSEUDOCÓDIGO

La figura indica que el proceso enmarcado en el rectángulo se ejecutará al menos una vez y se repetirá mientras que se cumpla la condición indicada al final del ciclo iterativo y que el ámbito del ciclo está delimitado precisamente por esa instrucción que involucra una comparación (LOOP WHILE <condición> o hasta_que <condición>).

De forma similar a la anterior, la condición comúnmente involucra dos variables que permiten romper el ciclo, como se sugirió para los casos anteriores: por ejemplo una variable alfanumérica que se lee dentro del ciclo y que se compara con un valor alfanumérico dado (LOOP WHILE resp$ = "n") .

Esta opción es aplicable al caso del lenguaje C, eliminando la palabra LOOP al final del ciclo, es decir, se tiene en ese caso la opción do - while(condición);

El ejemplo anterior resuelto con esta opción queda así:

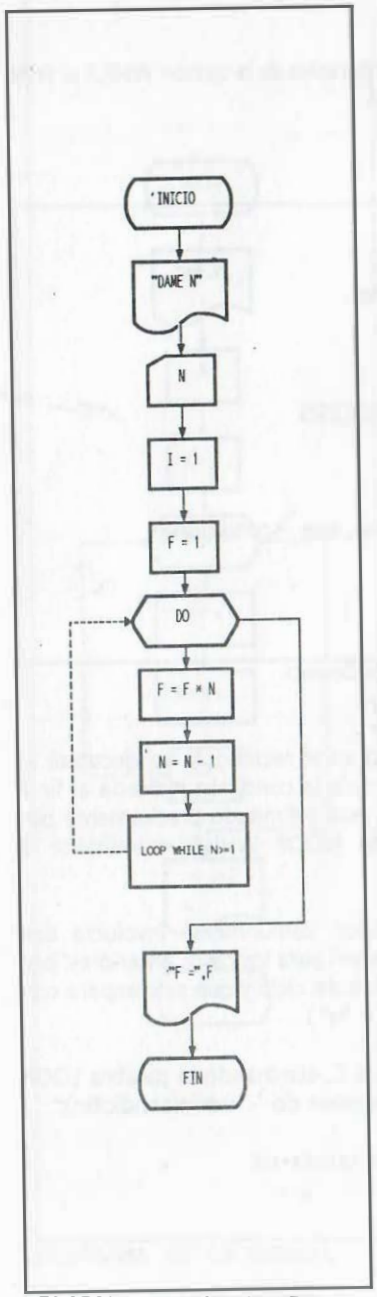


DIAGRAMA ESTRUCTURADO

```
INICIO

   leer N

   I <— 1

   f <— 1

   REPETIR

      F <— F * N

      N <— N - 1

   MIENTRAS_QUE N > = 1

   escribir "F = ", F

FIN
```

PSEUDOCÓDIGO

8.3.4 CUARTA OPCIÓN (DO-LOOP UNTIL)

La cuarta forma simplificada para un ciclo iterativo es la opción UNTIL al final del ciclo DO _ LOOP y se muestra a continuación:

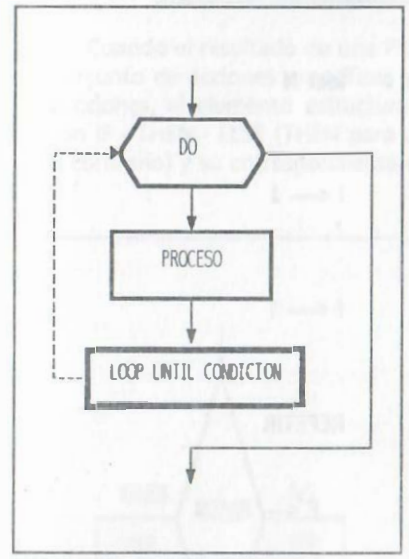


DIAGRAMA ESTRUCTURADO

```
REPETIR

   PROCESO

   HASTA_QUE  condición
```

PSEUDOCÓDIGO

La figura indica que el proceso enmarcado en el rectángulo se ejecutará hasta que se cumpla la condición indicada al final del ciclo iterativo y que el ámbito del ciclo está delimitado precisamente por esa instrucción que involucra la condición (LOOP UNTIL).

El ciclo DO - LOOP UNTIL o REPETIR HASTA_QUE puede interpretarse como: hacer ciclo HASTA QUE se cumpla la condición.

De forma similar a la anterior, la condición comúnmente involucra dos variables que permiten romper el ciclo, como se sugirió para los casos anteriores: por ejemplo una variable alfanumérica que se lee dentro del ciclo y que se compara con un valor alfanumérico dado (LOOP UNTIL resp$ = "n") .

El ejemplo anterior resuelto con esta opción queda así:

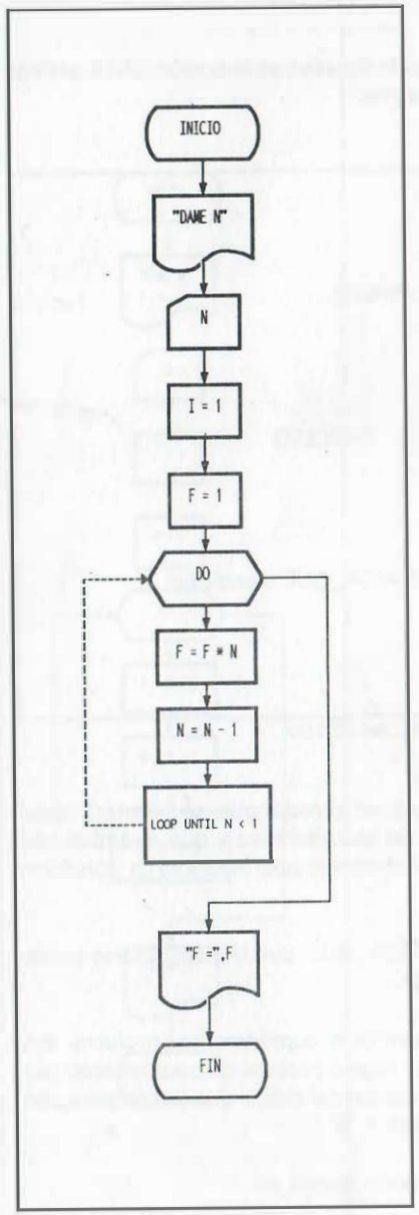


DIAGRAMA ESTRUCTURADO

```
INICIO

   leer N

   I <— 1

   f <— 1

   REPETIR

      F <— F * N

      N <— N - 1

   HASTA_QUE N < 1

   escribir "F = ", F

FIN
```

PSEUDOCÓDIGO

8.4 ELEMENTOS ESTRUCTURADOS DE SELECCIÓN

8.4.1 ELEMENTO ESTRUCTURADO IF - THEN - ELSE (SI - ENTONCES - EN_CASO_CONTRARIO)

Cuando el resultado de una PREGUNTA o (IF) considera por un lado ejecutar un conjunto de acciones específicas y en caso contrario ejecutar otro conjunto de instrucciones, el elemento estructurado que le corresponde en Quick Basic es la opción IF - THEN - ELSE (THEN para el caso afirmativo ó verdadero y ELSE para el caso contrario) y su correspondiente en lenguaje C es la if - else :

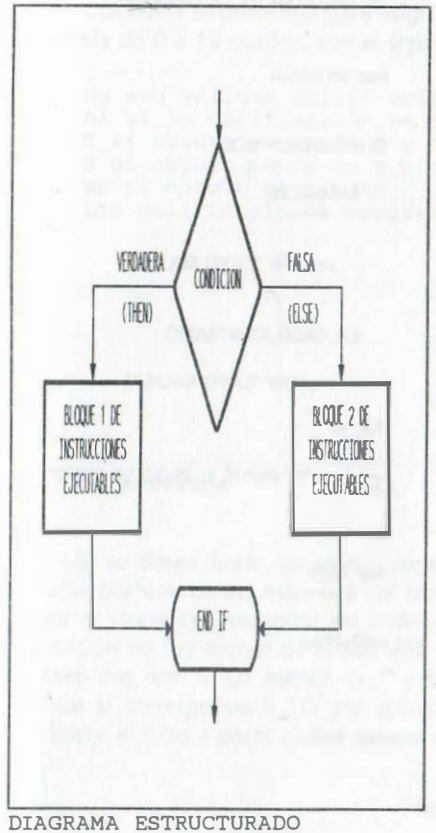


DIAGRAMA ESTRUCTURADO

```
SI <condición>

ENTONCES

      bloque de
      instrucciones 1

EN_CASO_CONTRARIO

      bloque de
      instrucciones 2

FIN_SI
```

PSEUDOCÓDIGO

Ejemplo de aplicación:

Un estudiante lanza al aire una moneda n veces, cada vez que cae águila se indica con un 1 y cada vez que cae sol se indica con un 2. Elabore el diagrama de flujo que muestre como salida en la pantalla el letrero "CAYO ÁGUILA" o el letrero "CAYO SOL", según el caso.

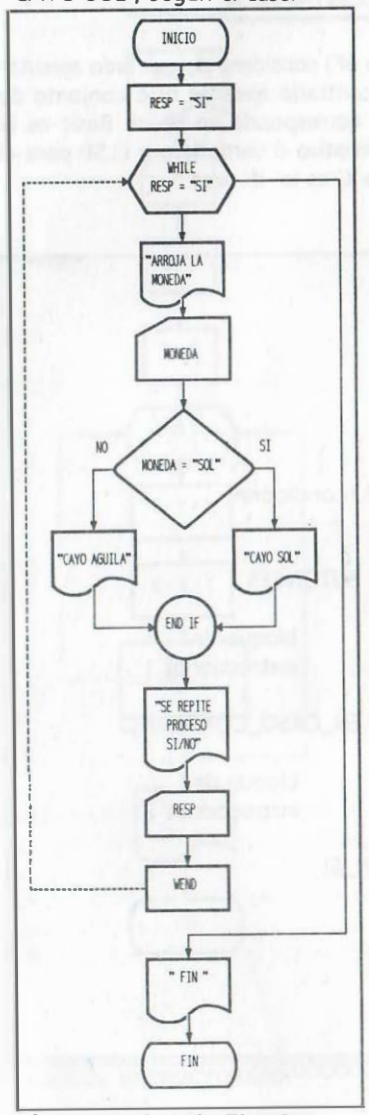


DIAGRAMA ESTRUCTURADO

```
INICIO

RESP <— "SI"

MIENTRAS RESP = "SI"

    imprimir "ARROJA LA MONEDA"

    leer MONEDA

    SI MONEDA = "SOL"
        ENTONCES

            imprimir "CAYO SOL"

        EN_CASO_CONTRARIO

            imprimir "CAYO AGUILA"

    FIN_SI

    imprimir "SE REPITE EL PROCESO SI/NO"

    leer RESP

FIN_MIENTRAS

imprimir "FIN"

FIN
```

PSEUDOCÓDIGO

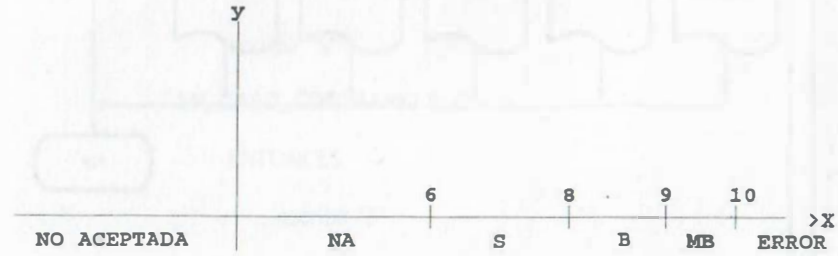
8.4.2 ELEMENTO ESTRUCTURADO DE SELECCIÓN ENCADENADO

IF - THEN - ELSE IF - THEN - ELSE IF ... ELSE

Este encadenamiento de preguntas en serie realizadas sobre una misma expresión para establecer categorías, intervalos o clases (de mayor a menor o viceversa), a manera de un filtro que separa en distintas clases los valores por la posición que ocupan en un rango de comparación para una expresión dada: precios, intervalos, clases, etc., se conoce normalmente como opción de preguntas encadenadas o de selección de clases; su aplicación se ejemplifica mediante un ejemplo a continuación.

Considere el problema para asignar la calificación del examen de un alumno, en escala de 0 a 10 puntos, con el siguiente criterio:

```
No son válidas calificaciones negativas
NA si la calificación es menor de 6
S si obtuvo menos de 8 y como mínimo 6
B si obtuvo menos de 9 y como mínimo 8
MB si obtuvo de 9 a 10
Las calificaciones mayores a 10 se consideran como errores
```



Si se desea filtrar las calificaciones realizando *preguntas en cadena*, debe iniciarse por uno de los extremos del rango de calificaciones, es decir, de menor a mayor o viceversa. Preguntar en cadena puede ejemplificarse diciendo que si la calificación no fue menor de 6, hay que ver si fue menor de 8 y si no fue ninguna de esas dos, ver si fue menor de 9 y cuando no fue ninguna de las anteriores, verificar si corresponde a 10; por último, en caso contrario ubicarla como error. Considere el filtro a partir de los valores no aceptados:

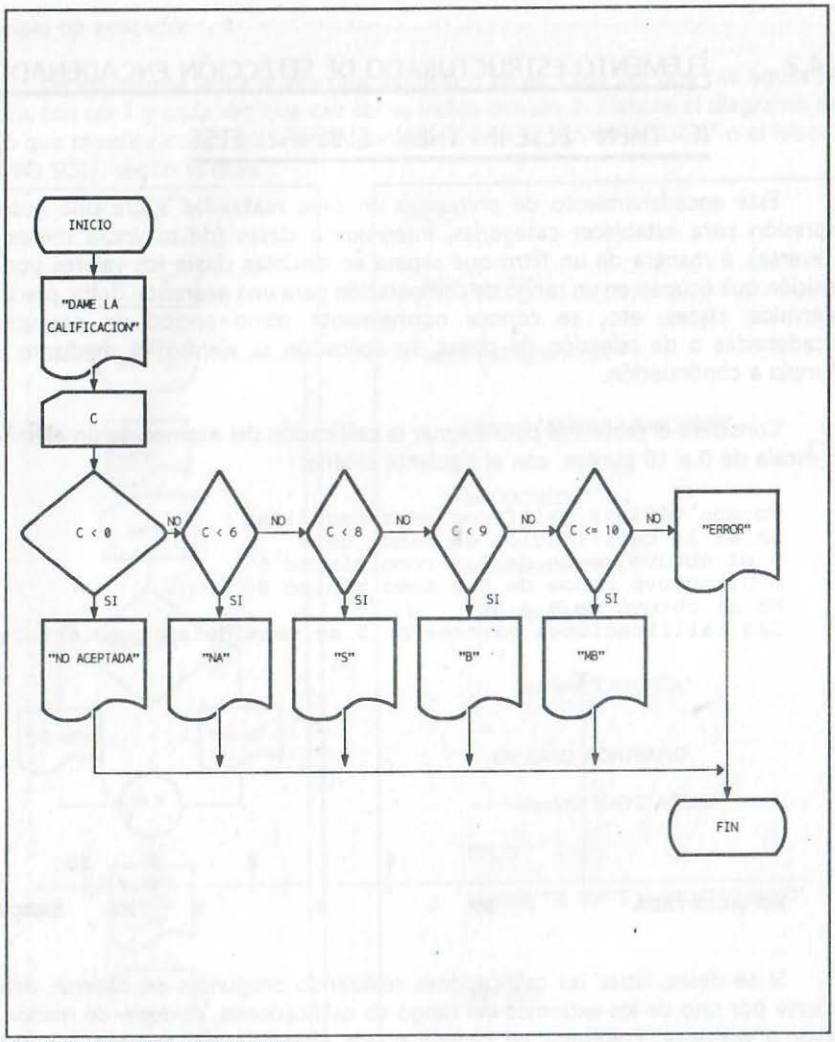


DIAGRAMA ESTRUCTURADO

Observe cómo las preguntas en cadena (una tras otra) sobre una misma expresión, permiten filtrar por intervalos sin necesidad de acotar ambos extremos, y únicamente delimitando uno de ellos. Cada pregunta establece una magnitud límite para cada intervalo y establece un filtro para las preguntas encadenadas que restan (en este caso las preguntas previas capturan los valores menores y sólo dejan pasar los valores de los intervalos de las siguientes preguntas que actuarán de la misma manera hasta el fin del encadenamiento).


```
INICIO
  imprimir "DAME CALIFICACION"
  leer C
  SI C < 0
        ENTONCES
            escribir "NO ACEPTADA"
        EN_CASO_CONTRARIO SI C < 6
            ENTONCES
                escribir "NA"
            EN_CASO_CONTRARIO SI C < 8
                ENTONCES
                    escribir "S"
                EN_CASO_CONTRARIO SI C < 9
                    ENTONCES
                        escribir "B"
                    EN_CASO_CONTRARIO SI C < 10
                        ENTONCES
                            escribir "MB"
                        EN_CASO_CONTRARIO
                            escribir "ERROR"
  FIN_SI
FIN
```

Observe que el encadenamiento genera un solo elemento estructurado que se inicia con un SI y finaliza con un solo FIN_SI, que cierra a todas las preguntas en cascada sin necesidad de cerrar cada una de ellas. A continuación se muestra una variante del mismo problema, colocando pregunta tras pregunta sin encadenar y por consiguiente, cerrando con un FIN_SI cada una de ellas:

```
INICIO

  imprimir "DAME CALIFICACION"
  leer C

  SI C < 0
      ENTONCES
          escribir "NO ACEPTADA"
      EN_CASO_CONTRARIO
          SI C < 6
              ENTONCES
                  escribir "NA"
              EN_CASO_CONTRARIO
                  SI C < 8
                      ENTONCES
                          escribir "S"
                      EN_CASO_CONTRARIO
                          SI C < 9
                              ENTONCES
                                  escribir "8"
                              EN_CASO_CONTRARIO
                                  SI C < 10
                                      ENTONCES
                                          escribir "MB"
                                      EN_CASO_CONTRARIO
                                          escribir "ERROR"
                                  FIN_SI
                              FIN_SI
                      FIN_SI
              FIN_SI
  FIN_SI

FIN
```

El uso adecuado de sangría a la izquierda de cada instrucción permite visualizar, en un solo golpe de vista, las estructuras anidadas que se requirieron en el algoritmo. Note que es más sencillo con preguntas encadenadas.

8.4.3 ELEMENTO ESTRUCTURADO PARA SELECCIONAR CASOS ESPECÍFICOS

Este elemento selecciona un caso específico del conjunto de opciones dadas y se denomina SELECT CASE en Quick Basic y switch en lenguaje C, su funcionamiento es similar al del las preguntas encadenadas. En él una expresión única se compara con ***N + 1*** opciones posibles, y se selecciona aquel caso específico que le corresponde (la comparación se efectúa de izquierda a derecha).

En lenguaje C la expresión sólo se puede comparar con un valor dado, mientras que en Quick Basic las opciones pueden ser de la siguiente forma:

a) Expresiones separadas por comas.

b) Un intervalo acotado con: expresión TO expresión.

c) Una condición lógica indicada con: IS operador relacional expresión.

d) Combinaciones de las formas anteriores.

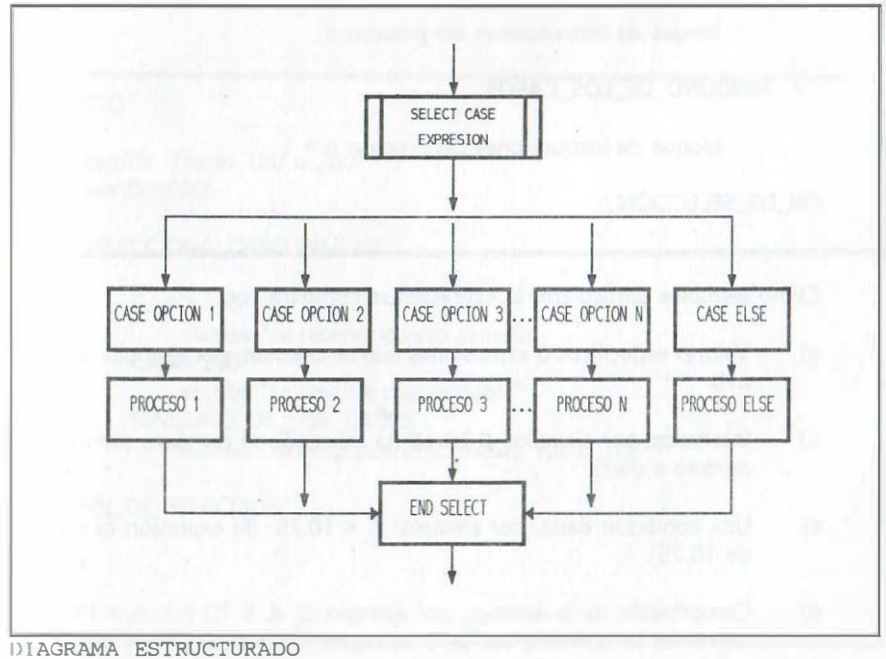


DIAGRAMA ESTRUCTURADO

A lo que corresponde el pseudocódigo siguiente:

```
SELECCIONA_CASO expresión
    CASO opción_1
        bloque de instrucciones del proceso 1
    CASO opción_2
        bloque de instrucciones del proceso 2
    CASO opción_3
        bloque de instrucciones del proceso 3
    .
    .
    .
    CASO opción_n
        bloque de instrucciones del proceso n
    NINGUNO_DE_LOS_CASOS
        bloque de instrucciones del proceso n + 1
FIN_DE_SELECCIÓN
```

Como ejemplos particulares la expresión se compara con:

a) Valores específicos o expresiones que se calculan, por ejemplo: 3, 4, 5, a+b.

b) Intervalos, por ejemplo: 0 TO 10 (la expresión se compara con valores de cero a diez).

c) Una condición dada, por ejemplo: IS < 10.75 (la expresión es menor de 10.75).

d) Combinación de lo anterior, por ejemplo: 2, 4, 5 TO 7.3, IS > 13.5 (la expresión se compara con 2, 4; cualquier valor de 5 a 7.3 y cualquier valor mayor de 13.5).

Como ejemplo de aplicación considere reservar un cuarto de hotel en función de una pregunta específica acerca del dinero con que se cuenta:

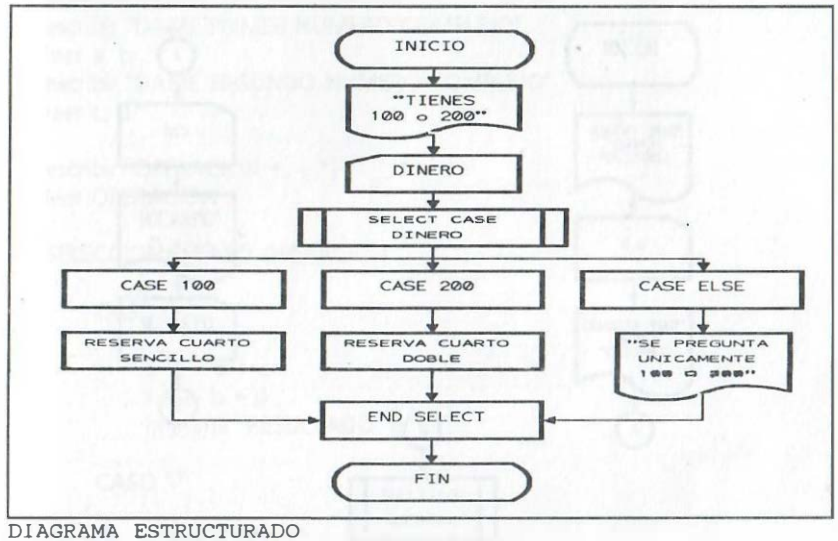


DIAGRAMA ESTRUCTURADO

```
INICIO

  escribir "Tienes 100 o 200"
  leer DINERO

  SELECCIONA_CASO DINERO

      CASO 100
          escribir "se reserva cuarto sencillo"
      CASO 200
          escribir "se reserva cuarto doble"
      NINGUNO_DE_LOS_CASOS
          escribir "se pregunta unicamente 100 o 200"

  FIN_DE_SELECCIÓN

FIN
```

SELECT CASE utilizando operaciones con números complejos:

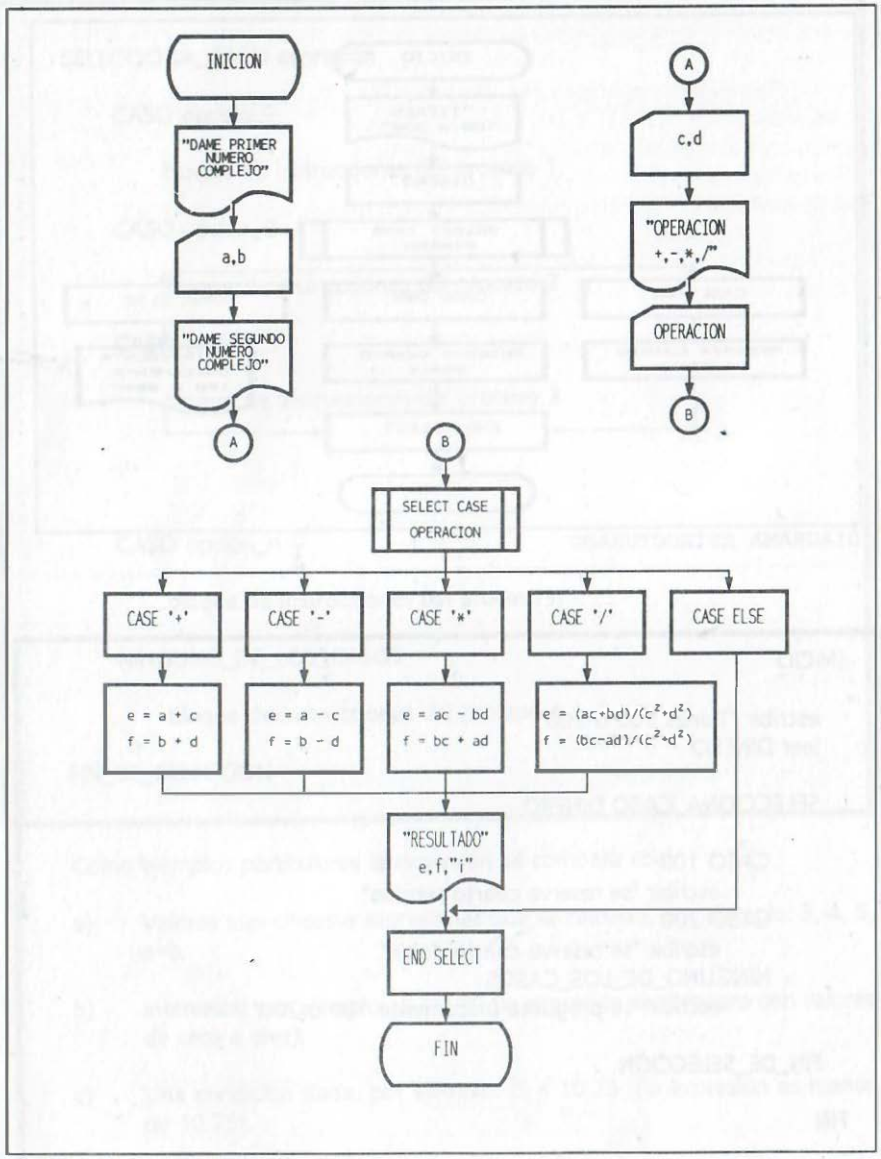


DIAGRAMA ESTRUCTURADO


```
INICIO

  escribir "DAME PRIMER NUMERO COMPLEJO"
  leer a, b
  escribir "DAME SEGUNDO NUMERO COMPLEJO"
  leer c, d

  escribir "OPERACION +, -, *, /"
  leer OPERACION

  SELECCIONA_CASO OPERACION

      CASO "+"

          e <— a + c
          f <— b + d
          imprimir "RESULTADO" e, f, "i"

      CASO "-"

          e <— a - c
          f <— b - d
          imprimir "RESULTADO" e, f, "i"

      CASO "*"

          e <— ac - bd
          f <— bc + ad
          imprimir "RESULTADO" e, f, "i"

      CASO "/"

          e <— (ac + bd)/(c² + d²)
          f <— (bc - ad)/(c² + d²)
          imprimir "RESULTADO" e, f, "i"

      NINGUNO_DE_LOS_CASOS
          continuar
  FIN_DE_SELECCIÓN

FIN
```

Note que en este caso no se requiere la opción NINGUNO_DE_LOS_CASOS, la cual se puede omitir.

Ejemplo de aplicación: considere la asignación de premios en un sorteo donde se extrae un dígito:

- Si se extrae alguno de los dígitos 2,5,9, el premio es un reloj.
- Si se extrae alguno de los dígitos 1,3,4,8, se asigna como premio un radio portátil.

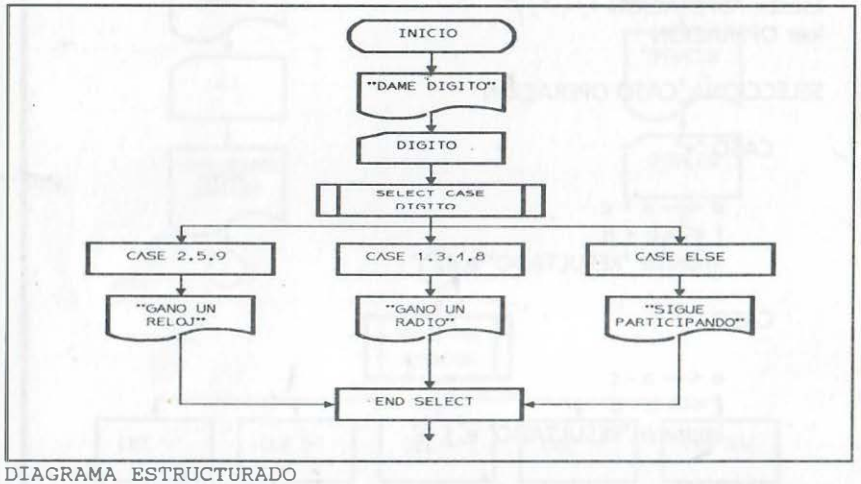


DIAGRAMA ESTRUCTURADO

```
INICIO

escribir "DAME DIGITO"
leer DIGITO

SELECCIONA_CASO DIGITO

    CASO 2, 5, 9
        escribir "GANO UN RELOJ"
    CASO 1, 3, 4, 8
        escribir "GANO UN RADIO"

    NINGUNO_DE_LOS_CASOS

        ESCRIBIR "SIGUE PARTICIPANDO"

FIN_DE_SELECCIÓN

FIN
```


Ejemplo de aplicación: considere la asignación de calificaciones en un examen bajo el siguiente criterio:

- No se aceptan calificaciones negativas.
- Las fracciones más pequeñas son décimas.
- Si se obtiene un puntaje menor de 6 se asigna NA.
- Si se obtiene un puntaje de 6.0 a 8 se asigna S.
- Si se obtiene un puntaje mayor de 8 y menor de 9.3 se asigna B.
- Si se obtiene un puntaje de 9.3 a 10 se asigna MB.

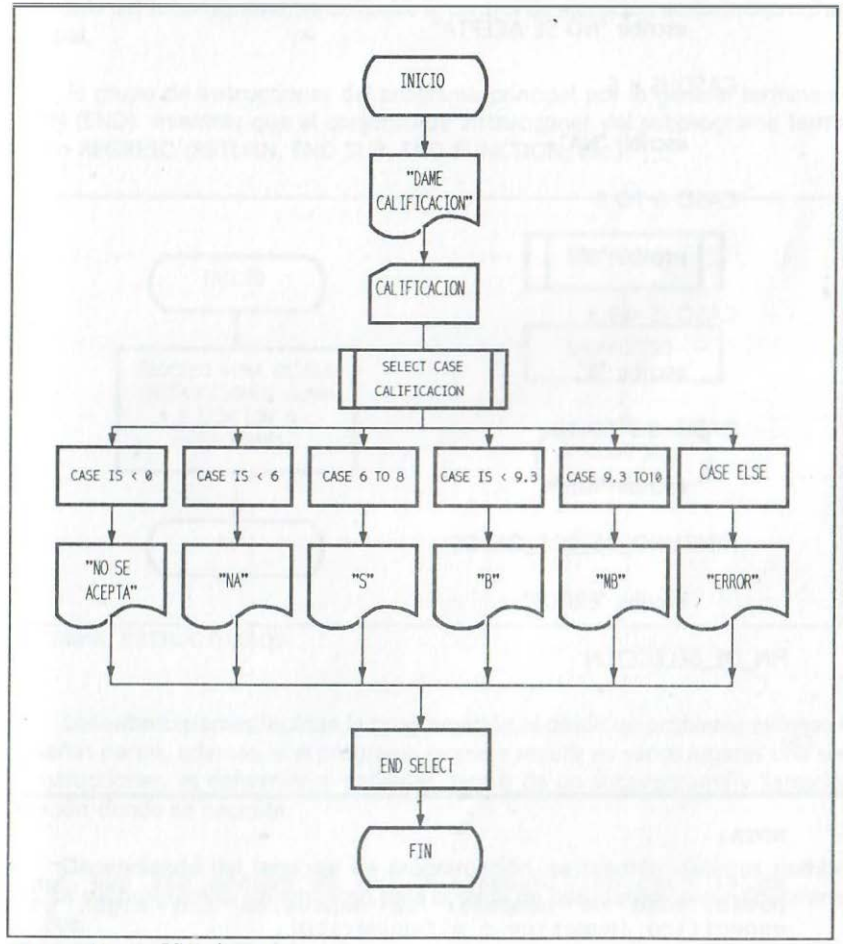


DIAGRAMA ESTRUCTURADO

```
INICIO

  escribir "DAME CALIFICACION"
  leer CALIFICACION

  SELECCIONA_CASO CALIFICACION

      CASO IS < 0

          escribir "NO SE ACEPTA"

      CASO IS < 6

          escribir "NA"

      CASO 6 TO 8

          escribir "S"

      CASO IS < 9.3

          escribir "B"

      CASO  9.3 TO 10

          escribir "MB"

      NINGUNO_DE_LOS_CASOS

          escribir "ERROR"

  FIN_DE_SELECCIÓN


FIN
```

NOTA:

En el caso del lenguaje C, no se cuenta más que con la posibilidad de comparar la expresión con algún valor específico (numérico o alfanumérico).

8.5 SUBPROGRAMAS

Los subprogramas son conjuntos de instrucciones agrupadas, que por lo general son módulos de un programa o procedimiento principal, en conjunto constituyen un problema largo o complicado con partes más sencillas.

La ejecución de las instrucciones de cada módulo se realiza mediante un llamado del subprograma en algún lugar del programa principal, con lo cual el control de ejecución se transfiere al subprograma. A continuación, una vez realizado el proceso del subprograma, se devuelve el control de ejecución a nuestro programa principal.

El grupo de instrucciones del programa principal por lo general termina con un FIN (END) mientras que el conjunto de instrucciones del subprograma termina con un REGRESO (RETURN, END SUB, END FUNCTION, etc.).

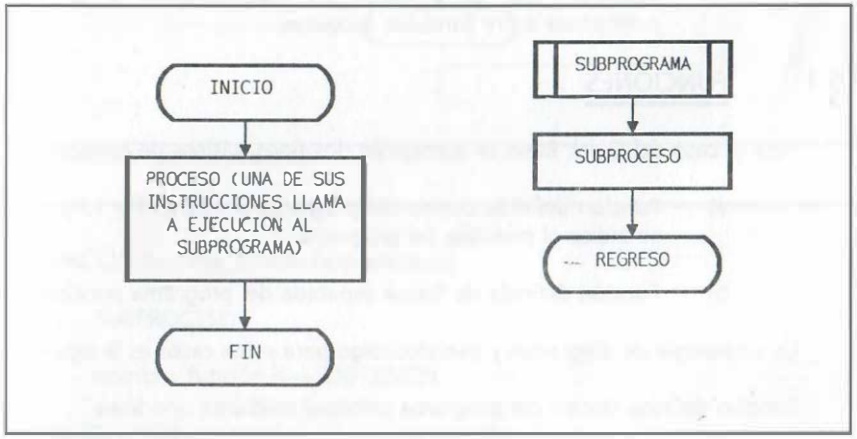


DIAGRAMA ESTRUCTURADO

Los subprogramas facilitan la programación al dividir un problema extenso en pequeñas partes, además, si el programa requiere repetir en varios lugares una serie de instrucciones, es conveniente definirlas dentro de un subprograma y llamarlo a ejecución donde se necesite.

Dependiendo del lenguaje de programación, se tendrán distintos nombres para los subprogramas, sin embargo para el tema de pseudocódigo consideraremos sólo dos:

FUNCIONES

SUBRUTINAS

Una función es un subproceso que asocia un valor al nombre propio de la función, una vez que han sido dados valores a las variables empleadas como argumentos de la función, tal y como en el caso de la función trigonométrica sen(30°). Es decir, dado el valor del argumento 30° se sabe que a la función sen se le asocia el valor 0.5

Una subrutina es un subproceso que, a diferencia de la función que sólo calcula un valor único, puede manejar varios argumentos de entrada y varias variables como salida. Los subprogramas emplean básicamente los siguientes tipos de variables:

a) Comunes o globales, cuyos valores son válidos tanto en programa como subprogramas.

b) Locales, cuando sus valores están delimitados al proceso o subproceso donde fueron declaradas.

c) Estáticas, que se emplean en subprogramas y su valor permanece entre llamadas sucesivas.

8.5.1 FUNCIONES

Para el caso del Quick Basic se manejarán dos tipos básicos de funciones:

a) Función definida dentro del programa principal. Por lo general se indica al principio del programa.

b) Función definida de forma separada del programa principal.

La simbología de diagramas y pseudocódigo para estos casos es la siguiente:

1. Función definida dentro del programa principal mediante una línea

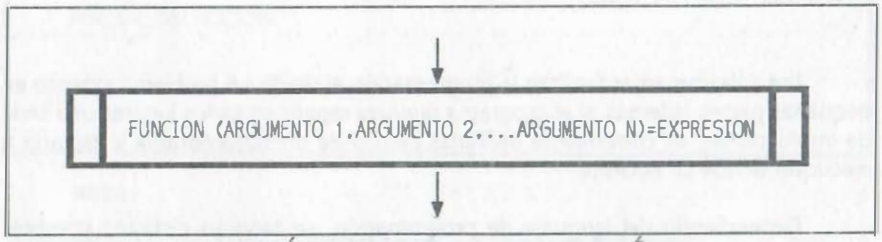


DIAGRAMA DE LA FUNCIÓN DEFINIDA EN UNA SOLA LÍNEA

FUNCIÓN nombre_función(argumentos) = expresión

PSEUDOCÓDIGO

2. Función definida dentro del programa principal mediante varias líneas

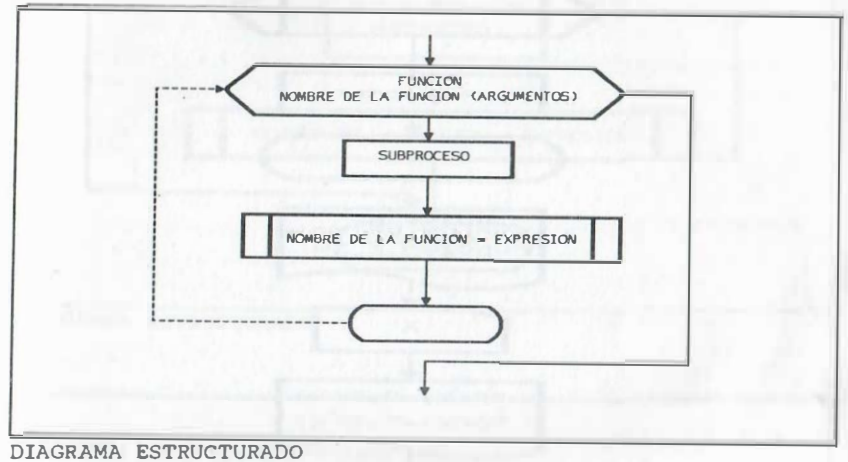


DIAGRAMA ESTRUCTURADO

```
FUNCIÓN nombre_función(argumentos)
    SUBPROCESO
    nombre_función <— EXPRESIÓN
FIN_FUNCIÓN
```

PSEUDOCÓDIGO

Ejemplo: considere el diagrama de flujo y pseudocódigo que indican cómo utilizar una función de una sola línea para el cálculo de una función y una función de varias líneas para calcular la derivada de esa misma función:

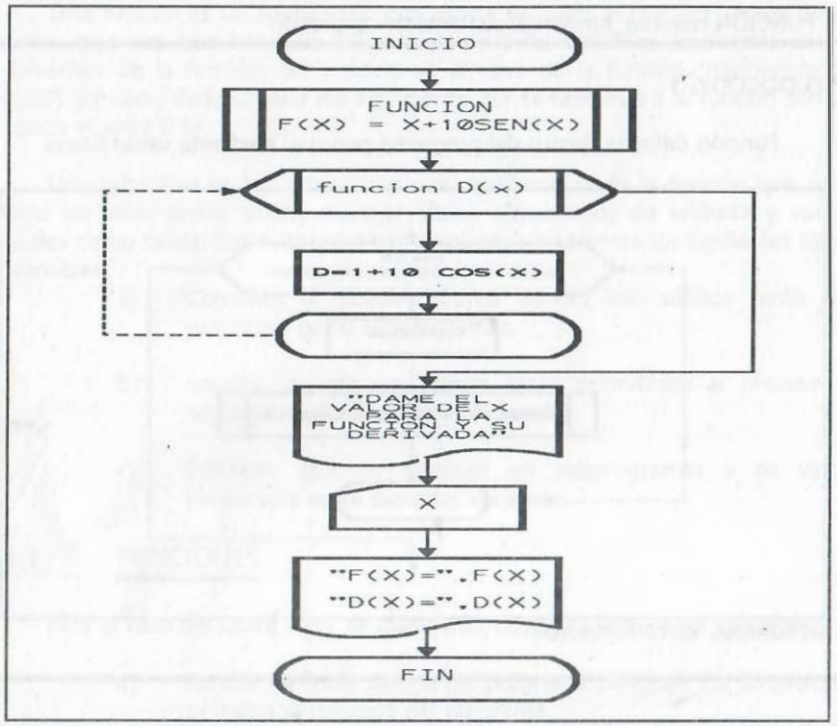


DIAGRAMA ESTRUCTURADO

```
INICIO

    FUNCIÓN F(X) = X + 10 * SEN(X)

    FUNCIÓN D(X)
        D = 1 + 10 * COS(X)
    FIN_FUNCIÓN

    escribir "DAME EL VALOR DE X PARA LA FUNCION Y SU DERIVADA"
    leer X

    escribir "F(X) = ", F(X), "D(X) = ", D(X)
FIN
```

PSEUDOCÓDIGO

3. Diagrama y pseudocódigo para la función separada del proceso principal

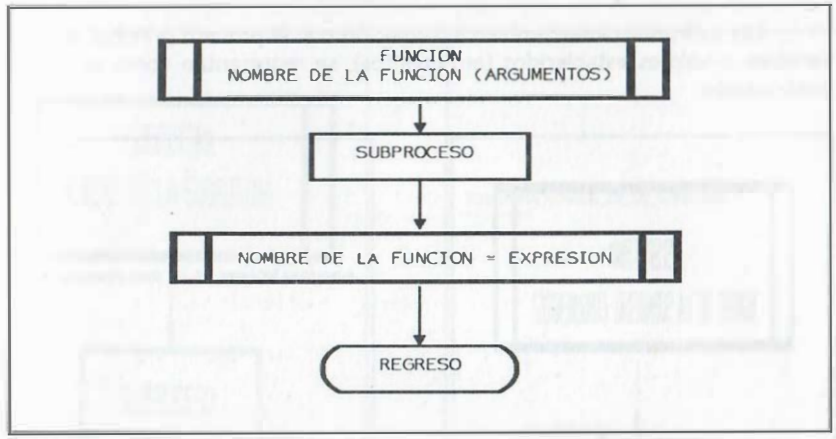


DIAGRAMA ESTRUCTURADO

```
FUNCIÓN nombre_función(argumentos)

    SUBPROCESO

    nombre_función <— EXPRESIÓN

REGRESA_FUNCIÓN
```

PSEUDOCÓDIGO

NOTA:

Como ya se comentó, los programas en lenguaje C se construyen exclusivamente empleando funciones. La función principal corresponde al procedimiento principal y los subprocesos se definen mediante funciones adicionales que, en estas notas, se consideran como funciones separadas de la función principal.

En Quick Basic una función regresa un valor asignándolo al nombre dado a la función; en lenguaje C la función regresa un valor por medio de la instrucción: return (expresión);

Note que, con el fin de diferenciar las funciones dentro del programa de las funciones separadas de él, en las primeras el ámbito de ellas se delimita con un FIN_FUNCION y en estas últimas con un REGRESA_FUNCION.

8.5.2 SUBRUTINAS

Las subrutinas intercambian información con el proceso principal a través de variables o valores establecidos (argumentos), se representan como se muestra a continuación:

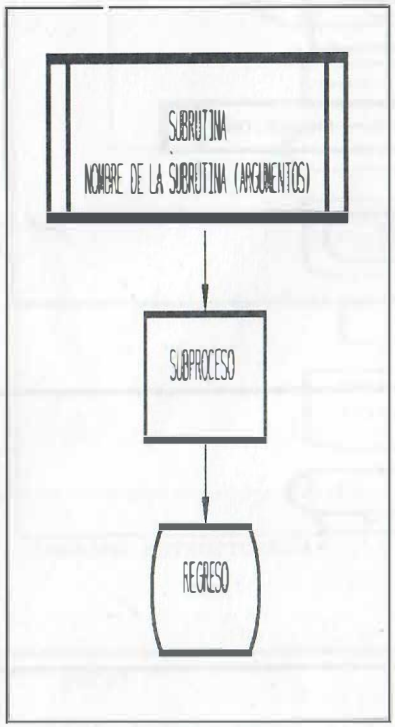


DIAGRAMA ESTRUCTURADO

```
SUBRUTINA NOMBRE_DE_LA_SUBRUTINA(argumentos)

    subproceso

    REGRESO
```

PSEUDOCÓDIGO

Algunos argumentos proporcionan información para realizar el subproceso al llamar a ejecución a la subrutina y otros regresan valores que se calculan en el subproceso, y son empleados en el programa principal. Este proceso se conoce como paso de parámetros por valor.

Otros argumentos emplean variables estáticas, cuyo valor se conserva entre cada llamada a la subrutina (en la subrutina se guardan los últimos valores empleados).

Además la subrutina puede utilizar algunas variables en memoria común o de forma global, de esta manera los valores cambiados por el subproceso automáticamente se modifican para usarlos en el programa principal y viceversa sin

necesidad de mencionarlos como argumentos. A continuación se muestra su diagrama y pesudocódigo:

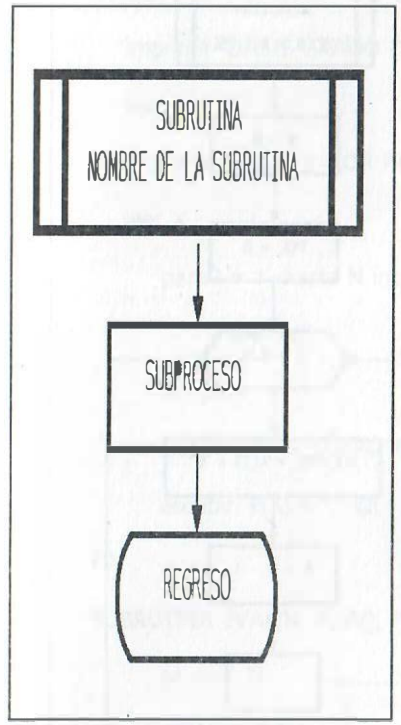


DIAGRAMA ESTRUCTURADO

```
SUBRUTINA NOMBRE_DE_LA_SUBRUTINA

    subproceso

    REGRESO
```

PSEUDOCÓDIGO

Observe que en el caso de las variables que se definen como globales o comunes, no se requiere mencionar argumento alguno.

Además hay ciertos lenguajes en los que el paso de información se hace con referencia a la dirección de las variables (paso de valores por referencia o dirección).

Ejemplo: considere el diagrama de flujo que indica cómo calcular un polinomio de grádo n mediante una subrutina que envía y recibe valores a través de argumentos:

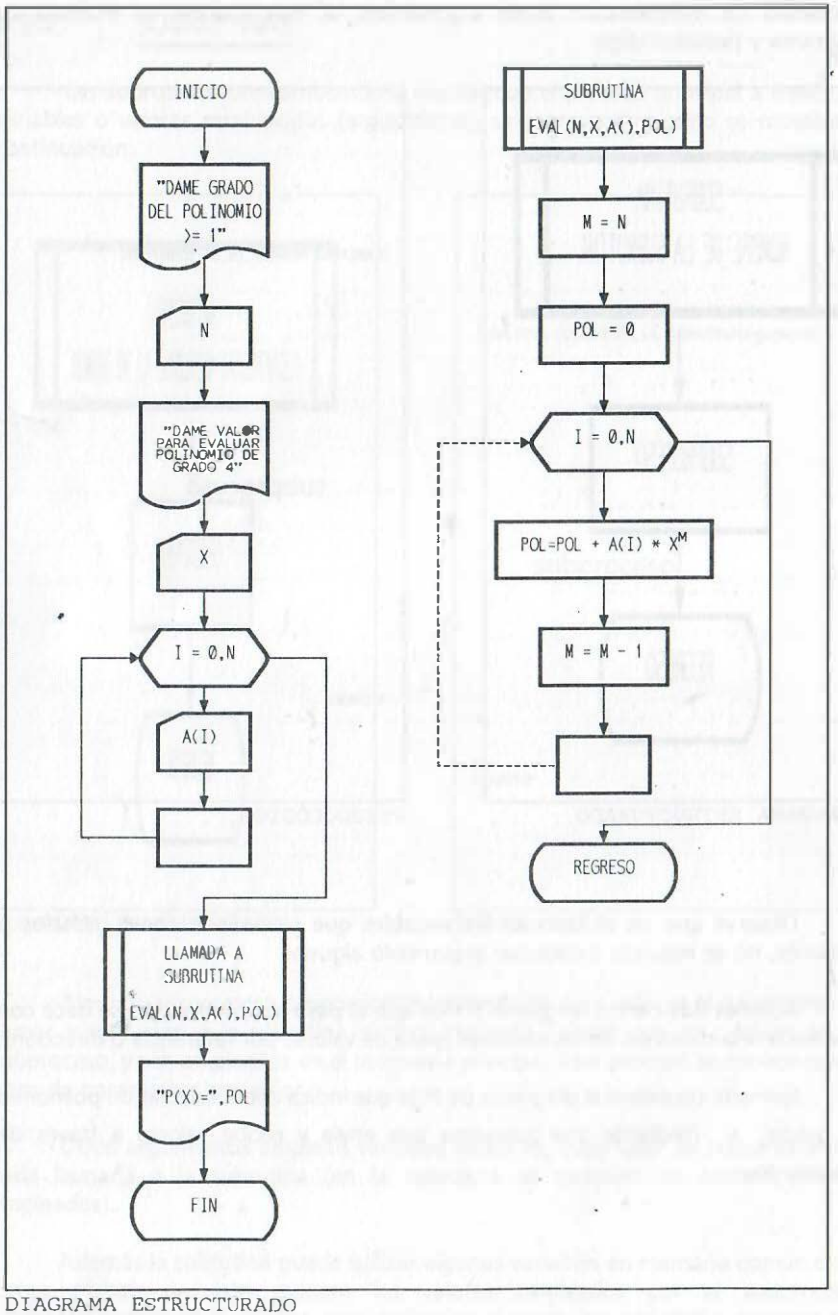
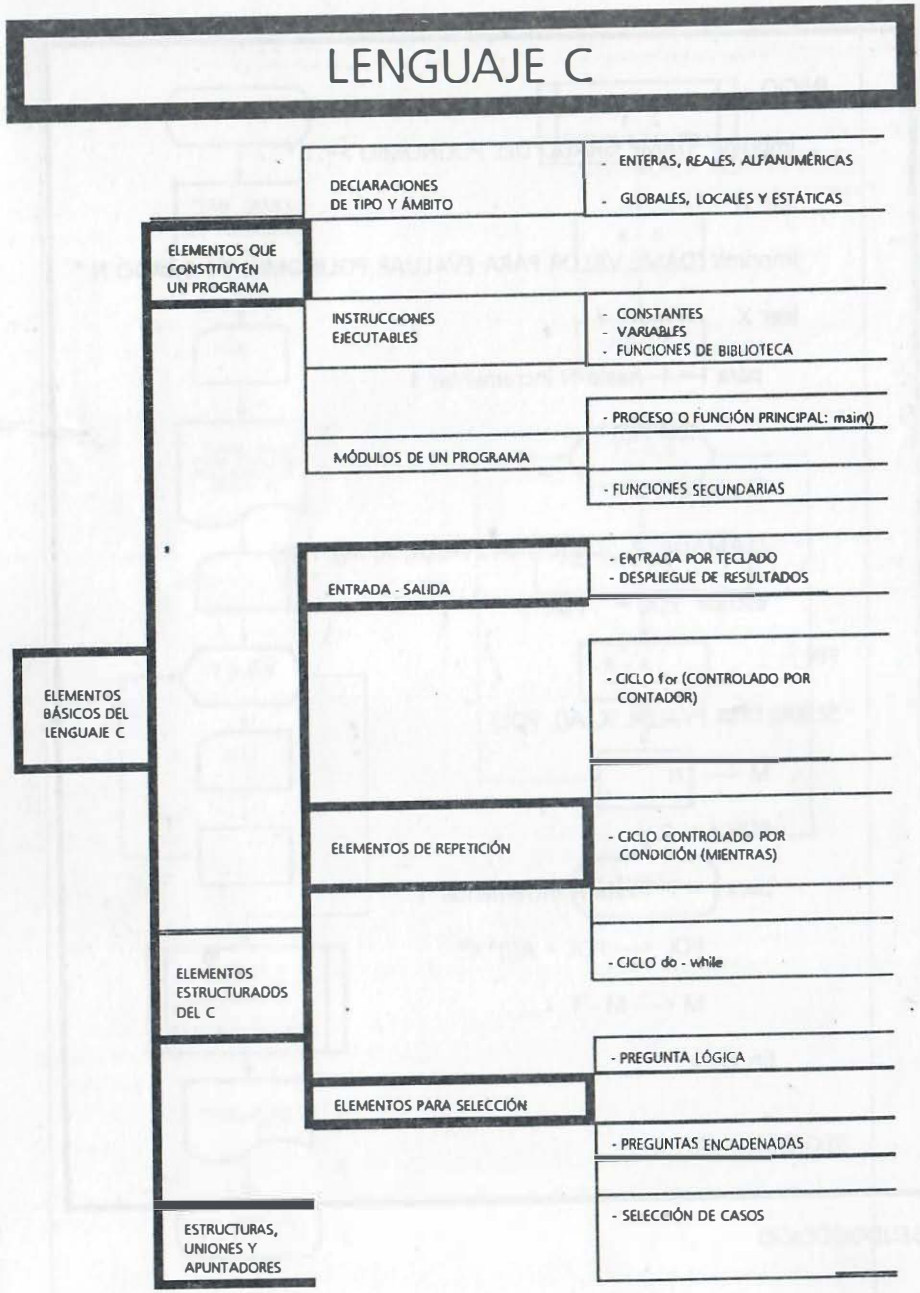


DIAGRAMA ESTRUCTURADO


```
INICIO
    imprimir "DAME GRADO DEL POLINOMIO >= 1 "
    leer N
    imprimir "DAME VALOR PARA EVALUAR POLINOMIO DE GRADO N "
    leer X
        para I = 1  hasta N incrementar 1
            leer A(I)
        fin_para
        LLAMADA_A_SUBRUTINA EVAL(N, X, A(), POL)
        escribir "P(X) = ", POL
FIN
SUBRUTINA EVAL(N, X, A(), POL)
        M <— N
        POL <— 0
        para I = 1  hasta N incrementar 1
            POL <— POL + A(I)*X^M
            M <— M - 1
        fin_para I
REGRESO
```

PSEUDOCÓDIGO



9 EL LENGUAJE DE PROGRAMACIÓN C

El lenguaje C reúne características de programación intermedia entre los lenguajes ensambladores y los lenguajes de alto nivel; con un gran poderío basado en sus operaciones a nivel de bits (propias de ensambladores) y la mayoría de los elementos de la programación estructurada de los lenguajes de alto nivel, por lo que resulta ser el lenguaje preferido para desarrollo de software de sistemas y aplicaciones de profesionales de la programación de computadoras.

En 1970 Ken Thompson de los laboratorios Bell se había propuesto desarrollar un compilador para el lenguaje Fortran que corría en la primera versión del sistema operativo UNIX tomando como referencia el lenguaje BCPL; el resultado fue el lenguaje B (orientado a palabras) que resultó adecuado para programación de software de sistemas. Este lenguaje tuvo la desventaja de producir programas relativamente lentos.

En 1971 Dennis Ritchie, con base en el lenguaje 8 desarrolló **NB** que luego cambió su nombre por C; en un principio sirvió para mejorar el sistema **UNIX** por lo que se le considera su lenguaje nativo. Su diseño incluyó una sintaxis simplificada, la aritmética de direcciones de memoria (permite al programador manipular bits, bytes y direcciones de memoria) y el concepto de apuntador; además, al ser diseñado para mejorar el software de sistemas, se buscó que generase códigos eficientes y una portabilidad total, es decir el que pudiese correr en cualquier máquina. Logrados los objetivos anteriores, C se convirtió en el lenguaje preferido de los programadores profesionales.

En 1980 Bjarne Stroustrup de los laboratorios Bell de Murray Hill, New Jersey, inspirado en el lenguaje Simula67 adicionó las características de la programación orientada a objetos (incluyendo las ventajas de una biblioteca de funciones orientada a objetos) y lo denominó *C con clases*. Para 1983 dicha denominación cambió a la de C++. Con este nuevo enfoque surge una nueva metodología que aumenta de nuevo las posibilidades de programación bajo nuevos conceptos.

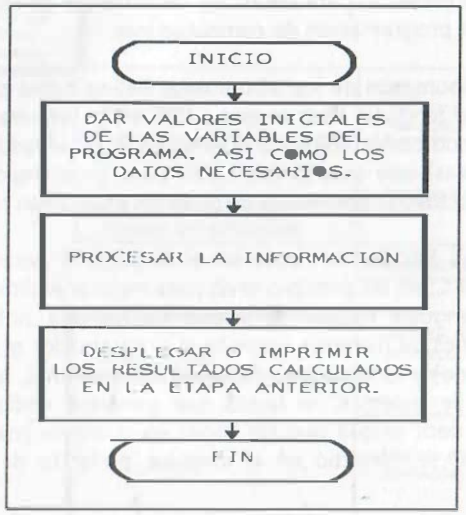
9.1 CONFIGURACIÓN DE UN PROGRAMA

En este tema describiremos los elementos que conforman un programa típico desde el punto de vista de la programación en lenguaje C.

Un programa de computadora en un lenguaje estructurado tiene un esquema que se puede describir para todos los casos, independientemente del lenguaje que se seleccione. Este esquema general, de todo programa de computadora, tiene dos grupos de instrucciones:

DECLARACIONES
INSTRUCCIONES EJECUTABLES

que conforman el grupo de líneas de un programa, desde el algoritmo más simple, con una entrada de información, un proceso de la misma y una salida de resultados:



En los lenguajes de alto nivel un algoritmo sencillo, conformado por una cuantas instrucciones, se denomina procedimiento o programa principal, en lenguaje C equivale a un módulo denominado función principal, según se muestra a continuación:

```
main()          /* Función principal */
{
  Declaraciones locales;

  INSTRUCCIONES EJECUTABLES (lectura, proceso y despliegue de resultados);

}
```

El ámbito de la función principal está delimitado por medio de llaves que encierran a las declaraciones y a las instrucciones ejecutables; toda información entre /* ... */ es un comentario que no se procesa.

9.1.1 DECLARACIONES

Estas son un conjunto de instrucciones que por lo general son conocidas en la mayoría de los lenguajes de alto nivel como *no ejecutables*; su objetivo es dar a la computadora la información sobre tipos de variables, arreglos, y características diversas, pero, en el caso del lenguaje C, también incluye la posibilidad de dar valores iniciales de variables.

9.1.2 INSTRUCCIONES EJECUTABLES

Las instrucciones ejecutables son aquellas expresiones en las que se calcula o realiza algo, es decir son aquellas que implican hacer algo y no simplemente declarar características de variables o dar información.

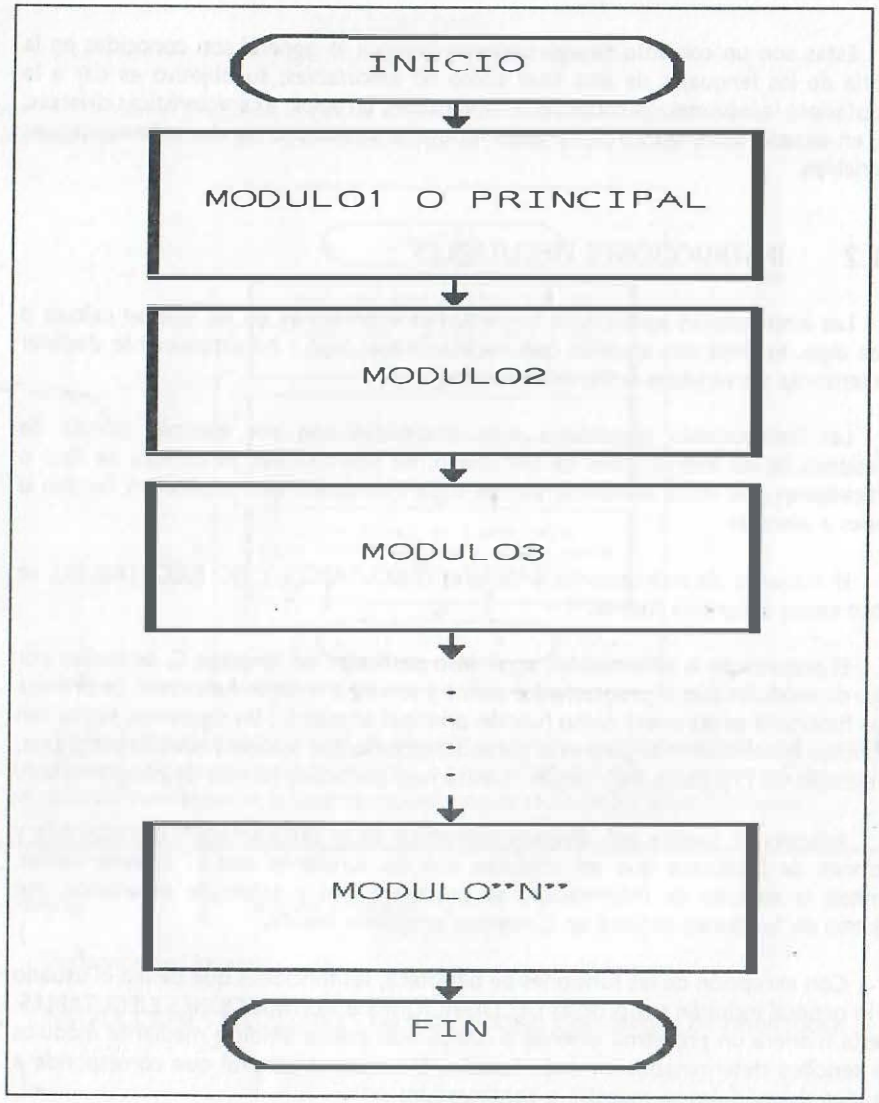
Las instrucciones ejecutables más empleadas son por ejemplo calculo de expresiones de las instrucciones de reemplazo, las instrucciones de control de flujo o ramificaciones, los ciclos iterativos, etc. Es toda instrucción que implica un cambio u órdenes a ejecutar.

El conjunto de instrucciones anteriores (EJECUTABLES Y NO EJECUTABLES), se conoce como programa fuente.

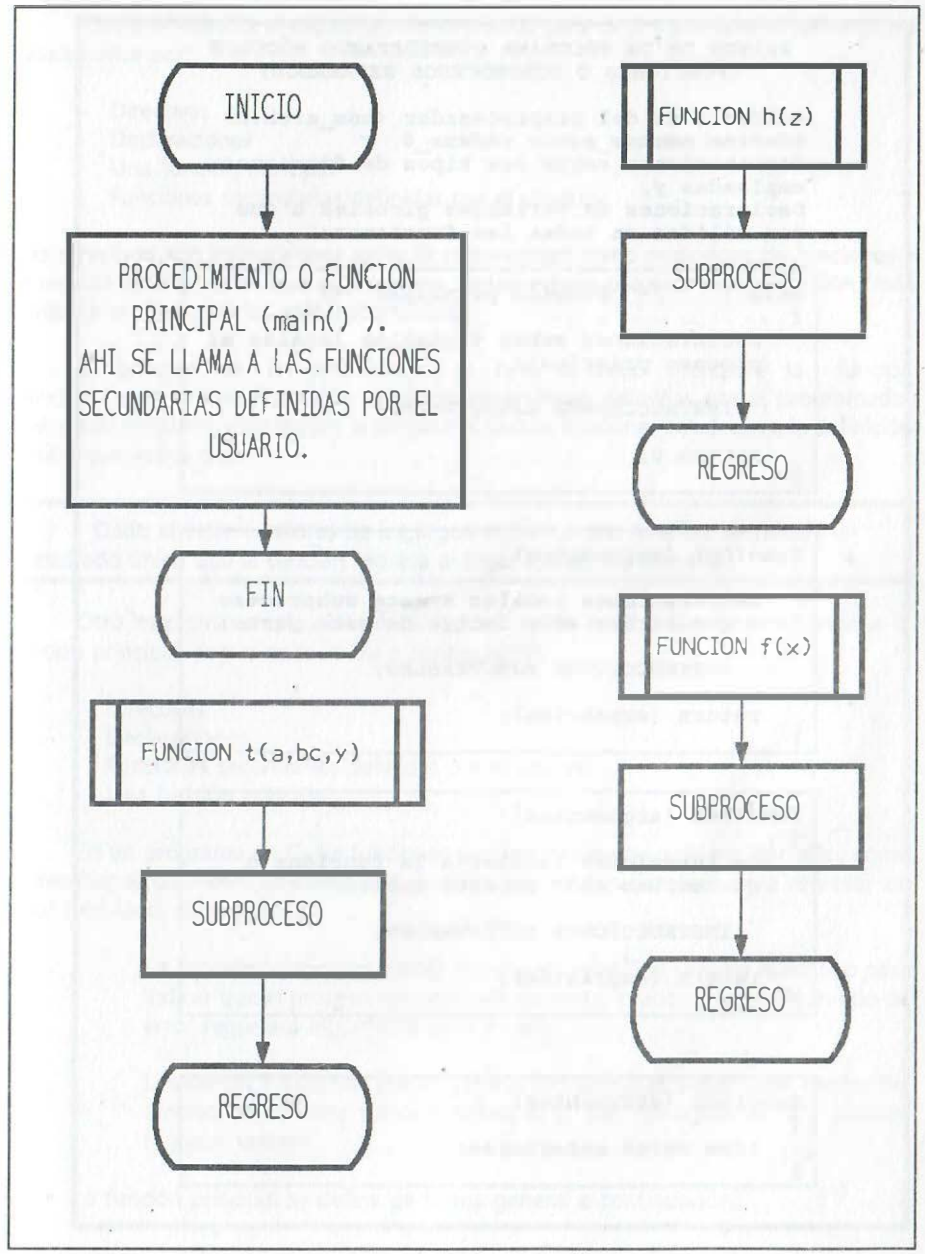
El proceso de la información, en el caso particular del lenguaje C, se realiza por medio de módulos que el programador define y son en sí mismos funciones. La primera de las funciones se conocerá como función principal o main() , las siguientes se podrán denominar arbitrariamente y agrupar parte del proceso que realizará nuestro programa, por ejemplo de f1() hasta fN(), según nuestra muy particular técnica de programación.

Además se cuenta con diversos elementos de la programación estructurada y funciones de biblioteca que en conjunto con las funciones que el usuario define, permiten la entrada de información, su procesamiento y salida de resultados, ese conjunto de funciones definirá en C nuestro programa fuente.

Con excepción de las funciones de biblioteca, las funciones que defina el usuario por lo general incluirán sus propias DECLARACIONES e INSTRUCCIONES EJECUTABLES, de esta manera un problema extenso o complicado puede dividirse mediante módulos más sencillos determinados en cada funclón. El esquema general que corresponde a esta idea de módulos se muestra a continuación:



Para el programador o usuario del lenguaje de programación C, el esquema de sus funciones se puede representar de acuerdo a lo siguiente:



PARTES DE UN PROGRAMA CONSIDERANDO MÓDULOS (FUNCIONES O SUBPROCESOS SEPARADOS)

```
#directivas del preprocesador <nom_arch.h>
#define nombre_equiv cadena_0
Declaraciones sobre los tipos de funciones
empleadas y,
Declaraciones de variables globales o que
son válidas en todas las funciones;

main()     /*  Función principal  */
{
   Declaraciones sobre variables locales al
   proceso principal;

     INSTRUCCIONES EJECUTABLES;

   return 0;
}

función1 (argumentos)
{
   Declaraciones locales a este subproceso
   o que actúan sólo dentro de esta parte;

     INSTRUCCIONES EJECUTABLES;

   return (expresión);
}

funciónX (argumentos)
{
   Declaraciones locales a la funciónx o
   que actúan sólo en este ámbito;

     INSTRUCCIONES EJECUTABLES;

   return (expresión);
}

funciónZ (argumentos)
{
   idem casos anteriores;
}
```


Como se observa el esquema más empleado para un programa en C se compone básicamente por:

- Directivas
- Declaraciones
- Una función principal
- Funciones secundarias definidas por el usuario

Las directivas son instrucciones específicas conocidas como prototipos de funciones y se indican sólo aquellas que el programa requiere para procesar la información (más adelante se describen las más importantes).

El proceso de la información se lleva a cabo mediante la función principal (main ()) auxiliada con funciones secundarias definidas por el programador que, en su conjunto, constituyen el programa. Dichas funciones obedecen a la definición clásica que indica que:

> Dado el valor o valores de los argumentos de una función, se calcula un resultado único que la función regresa al lugar donde fue llamada.

Otro esquema empleado coloca las funciones secundarias antecediendo a la función principal, según se muestra a continuación:

- Directivas
- Declaraciones
- Funciones secundarias definidas por el usuario
- Una función principal

En un programa en C, las funciones pueden emplearse para realizar algo, como el despliegue de mensajes como una portada y no necesariamente para regresar un valor calculado, sin embargo:

> La función principal o main() regresa un valor 0 al sistema operativo para indicar que el proceso terminó exitosamente, cuando ocurre algún tipo de error regresará algún otro valor entero.
>
> Las demás funciones, por lo general son llamadas a ejecución dentro del ámbito de nuestra función principal y, por consiguiente, ahí pueden regresar valores.

La función principal se define de forma general a continuación:

```
main()   /* Función principal */
{
   Declaraciones locales;
   INSTRUCCIONES EJECUTABLES;

   return 0;
}
```

Observe que toda función se escribe con letras minúsculas y que el ámbito de la función se inicia mediante una llave { que encierra el conjunto local de declaraciones e instrucciones ejecutables y que dicho ámbito está delimitado por la llave que cierra } , además toda declaración o instrucción ejecutable termina con un punto y coma ;

La sintaxis o escritura de las funciones que el programador defina es parecido a lo anterior. No obstante, debe tomarse en cuenta que en el lenguaje C, las funciones que el usuario genere pueden no regresar valor alguno. Cuando se requiere que la función regrese algún valor se usará la instrucción:

return (expresión);

Las funciones que únicamente realizan algo, pero que no necesitan regresar un resultado, indican esa condición mediante una declaración de tipo void y se suprime la instrucción return.

Toda función secundaria definida por el usuario deberá declararse, indicando su tipo (entera, real o de caracter), inmediatamente después de las directivas del preprocesador. Por ejemplo:

De tipo entero, por ejemplo:	int funcionz(argumentos)
De tipo real, por ejemplo:	float funciony(argumentos)
De tipo void (que no regresa nada):	void funcionx(argumentos)
De tipo void (sin argumentos):	void funcionw(void)

El esquema general para las funciones que el programador define es:


```
declaración_de_tipo_de_la_función;

tipo_y_nombre_de_la_función(argumentos)
 {
   Declaraciones locales;
   INSTRUCCIONES EJECUTABLES;

   return (expresión);
 }
```

Ejemplo de una función principal y dos funciones secundarias, observe que la última de ellas no regresa valor como resultado (no incluye return (exp)); :

PROGRAMA QUE CONSTA DE LA FUNCIÓN PRINCIPAL Y DOS FUNCIONES SECUNDARIAS

```
main()    /*  Función principal  */
{
   Declaraciones sobre variables locales al
   proceso principal;
     INSTRUCCIONES EJECUTABLES;
   return 0;
}
```

```
función1 (argumentos)
{
   Declaraciones locales a este subproceso
   o que actúan sólo dentro de esta parte;
     INSTRUCCIONES EJECUTABLES;
   return (expresión);
}
```

```
void funciónX (void)
{
   Declaraciones locales a la funciónx o
   que actúan sólo en este ámbito;
     INSTRUCCIONES EJECUTABLES;
}
```

También es necesario declarar el tipo de argumentos que emplea la función, por ejemplo:

```
int f1(int a);

f1(int a)
  {
    Declaraciones locales;
    INSTRUCCIONES EJECUTABLES;
    return (expresión);
  }
```

Comentario: Al iniciar el ámbito de la función, su nombre jamás incluye un punto y coma al final, sólo lo incluyen las declaraciones, funciones de biblioteca e instrucciones ejecutables.

9.2 DIRECTIVAS DE PREPROCESAMIENTO

Dado que el lenguaje C es de nivel intermedio, requiere de instrucciones que se conocen como prototipos de funciones y se utilizan con la finalidad de ampliar, complementar y enriquecer el entorno de programación de C; en la mayoría de los textos se le denomina como directivas de preprocesamiento, Dichas directivas, siempre acompañan a las funciones de la biblioteca standard que emplea el programa y empiezan con el símbolo #, de esa manera se complementa las instrucciones dirigidas al compilador en el código o programa fuente que se elabora, permitiendo el uso de funciones standard de la biblioteca de C.

Las directivas más importantes del preprocesador de C son las siguientes:

```
# define nombre_equiv cadena_0
# error mensaje_de_error
# include
```

EXPLICACIÓN ABREVIADA DE LAS DIRECTIVAS DE PREPROCESAMIENTO MAS EMPLEADAS

define permite que todo lo indicado en cadena_0 pueda utilizarse mediante su nombre alterno dado en nombre_equiv. Además permite definir constantes de cualquier tipo y funciones de una línea. Por ejemplo:

```
#define inicia_llave {   /* inicia_llave reemplaza a la llave {  */
```



```
#define cierra_llave }   /* cierra_llave equivale a la llave }   */
#define return regresa   /* return es igual a la palabra regresa */
#define c1 12            /* c1 = 12          (constante entera) */
#define pi 3.14159265    /* pi = 3.14159265  (constante real)   */
#define xx 'x'           /* xx = 'x'         (constante alfanumérica)*/
#define si "positivo"    /* si = "positivo" (constante alfanumérica)*/
#define cubo(x) x*x*x    /* cubo(x) = x*x*x (función de una línea)*/
```

error cuando el compilador encuentra un error, incluye además el mensaje_de_error que se indique.

include se usa para incluir los archivos de cabecera <nombre_de_archivo.h>, que requieren las funciones de biblioteca que emplea nuestro programa. Esta directiva es de las más importantes pues indica por ejemplo que se usará la entrada/salida standard <stdio.h>, o que se emplearán funciones matemáticas de biblioteca <math.h>, etc. La relación de archivos de cabecera más usuales son los siguientes:

<alloc.h>	para la asignación dinámica de memoria.
<conio.h>	para uso de funciones de entrada/salida por teclado o consola.
<graphics.h>	para uso de funciones relacionadas con gráficas.
<io.h>	para uso de funciones de entrada/salida de bajo nivel.
<math.h>	para uso de funciones matemáticas.
<mem.h>	para uso de funciones para manipulación de memoria.
<string.h>	para uso de funciones relacionadas con cadenas.
<time.h>	para uso de funciones de hora y fecha.

Para determinar que prototipos de funciones requiere nuestro programa, se recomienda consultar la ayuda que se activa colocando el cursor en nombre de la función o palabra y oprimiendo las teclas Ctrl F1.

9.3 TIPOS DE DECLARACIONES DE VARIABLES Y SU ÁMBITO DE ACCIÓN EN EL DISEÑO ESTRUCTURADO DE PROGRAMAS

El lenguaje C requiere de la declaración sobre los tipos de variables y el ámbito donde son válidas; los tipos de variables más empleados son los siguientes:

ENTERAS (presición simple y doble)
REALES (presición simple y doble)
ALFANUMÉRICAS O DE CADENA

Sin embargo en lenguaje C se definen también las variables sin signo. Estas variables pueden ser de tipo entero y alfanumérico; las variables sin signo aprovechan el bit que corresponde al signo y por consiguiente modifican los límites de los valores permitidos.

El ámbito de validez o de acción de las variables surge cuando un programa extenso se divide en varias partes, mediante varias funciones o módulos, de esa manera se tendrán variables:

GLOBALES
LOCALES
ESTÁTICAS

Las variables globales son las que son válidas en cualquiera de los módulos o funciones, es decir actúan en todo el ámbito del programa. Para definirlas basta indicarlas al principio del programa, antecediendo a todas las funciones (incluida la principal). De esta forma los valores de las variables globales se pueden emplear tanto en la función principal como en las demás funciones que el programador defina; dichas variables pueden modificar sus valores en cualquiera de las funciones y ser válidos en las restantes, sin requerir ser mencionadas como argumentos de función alguna.

Las variables locales son aquellas que son exclusivas de alguna función; para definirlas así, deberán ir inmediatamente al inicio de la función que les corresponda, es decir, al inicio del ámbito de la función principal o al inicio del ámbito de alguna de las otras funciones. El ámbito de acción de una variable local está limitado al lugar en el que fue declarada; su valor no es válido fuera del subproceso o proceso principal que implica. Una variable local toma su valor inicial cada vez que se llama al subproceso y su valor se borra al salir de él. Las declaraciones de variables locales permiten considerar a las variables como particulares de cada función (de esta forma los módulos o funciones que el programador define en lenguaje C pueden ser independientes entre si).

Las variables estáticas son aquellas que toman un valor inicial al ser utilizadas la primera vez pero, una vez cambiados, permanecen hasta la siguiente llamada y así sucesivamente durante toda la vida del programa.

OBSERVACIONES:

Tenga en cuenta que en el lenguaje C una función puede regresar como resultado a su llamada, un valor único mediante la instrucción return (expresión); a partir de los argumentos que se indican entre paréntesis y el subproceso que realiza o simplemente emplearse para escribir, leer y procesar algo, sin regresar valor alguno.

En el caso general del lenguaje C todas las funciones que el usuario genere, están al mismo nivel y no es posible definir una función de usuario dentro de otra función de usuario.

Las declaraciones de tipo más empleadas son las siguientes:

/* comentario */	para indicar que esa parte contiene un comentario. Su uso es importante pues permite documentar el programa en el lugar adecuado.
static variables;	para declarar variables estáticas
int variables;	para declarar variables enteras (almacenadas en 2 bytes)
long variables;	para declarar variables enteras largas (que ocupan 4 bytes)
float variables;	para declarar variables reales (que ocupan 4 bytes)
double	para declarar variables reales de doble precisión (8 bytes)
char	para declarar variables alfanuméricas o de cadena (1 byte por caracter)

TIPO		EJEMPLO DE DECLARACIÓN DE TIPO
ENTERAS	SIMPLES	`int a,b,c;`
	DOBLE PRECISIÓN	`long d,e,f;`
REALES O DE PUNTO FLOTANTE	SIMPLES	`float g,h,z;`
	DOBLE PRECISIÓN	`double x,y,v;`
DE CADENA O ALFANUMÉRICAS		`char w[5][5],t[18], m;`

9.4 ELEMENTOS DE PROGRAMACIÓN DEL LENGUAJE C

Los elementos de programación de todo lenguaje son:

CONSTANTES
VARIABLES
FUNCIONES DE BIBLIOTECA
OPERADORES
ELEMENTOS ESTRUCTURADOS DE LA PROGRAMACIÓN

En el lenguaje C muchas de las instrucciones ejecutables, como la de reemplazo, emplean los tres primeros elementos junto con los operadores aritméticos siguientes:

+	SUMA
-	RESTA O NEGACIÓN
*	MULTIPLICACIÓN
/	DIVISIÓN
%	RESIDUO DE UNA DIVISIÓN ENTERA
- -	DECREMENTO
+ +	INCREMENTO

los operadores relacionales:

<, <=, ==, !=, >, >=, (menor, menor o igual, igual, diferente, mayor, mayor o igual)

y los operadores lógicos:

! (negación),
&& (Y),
|| (O)

además de los paréntesis redondos que se utilizan como elementos de agrupamiento o de prioridad de evaluación de las expresiones aritméticas.

9.4.1 CONSTANTES DE C

9.4.1.1 CONSTANTES ENTERAS

Una constante entera es una cantidad sin punto decimal (con signo positivo opcional) cuyos valores mínimo y máximo en precisión sencilla (ENTERO CORTO que se almacena en 2 bytes) son:

-32768 y 32767

Una variante es la entera sin signo (que se almacena en 2 bytes), con los valores mínimo y máximo de:

0 y 65535

mientras que en doble precisión (ENTERO LARGO que se almacena en 4 bytes) :

-2 147 843 648 y 2 147 843 647

9.4.1.2 CONSTANTES REALES

Una constante real es una cantidad que incluye punto decimal (con signo positivo opcional) cuyos valores mínimo y máximo aproximados (en precisión sencilla 4 bytes) son:

$-3.4\ 10^{-38}$ Y $3.4\ 10^{38}$ (6 dígitos válidos)

mientras que sus valores aproximados en doble precisión (8 bytes) son:

$-1.79\ 10^{-308}$ Y $1.79\ 10^{308}$ (13 Dígitos válidos)

9.4.1.3 CONSTANTES ALFANUMÉRICAS

Una constante alfanumérica es un conjunto de caracteres delimitados por comillas dobles como por ejemplo:

"-32,768", "uno mas uno", "ABCFXY","SI","NO"

con una longitud máxima de 255 caracteres en C. Es importante comentar que si el valor alfanumérico es un solo caracter, se usa comilla simple, por ejemplo:

'a', 'u'

9.4.2 VARIABLES

DEFINICIÓN:

Una variable es un nombre que empieza con una letra y varios caracteres más (el total de caracteres depende de la versión del lenguaje y la computadora que se emplea); se recomienda usar sólo letras minúsculas, dígitos y el subguión.

Las variables del lenguaje C están clasificadas en:

enteras, reales, de cadena (alfanuméricas)

y se emplean para almacenar los valores definidos por constantes de tipo:

entero, real, alfanumérico

9.4.2.1 VARIABLES ENTERAS

Las variables enteras son aquellas cuyo nombre está mencionado en una declaración de tipo incluida en algún lugar del programa, por ejemplo:

```
int a,b,c;          unsigned int x, y, z;
```

también se puede definir la doble precisión para enteras. por ejemplo:

```
long int a,b,c;
```

9.4.2.2 VARIABLES REALES

Las variables reales son aquellas cuyo nombre está mencionado en una declaración de tipo incluida en algún lugar del programa, por ejemplo con precisión simple:

```
float a, b, c;
float d = 1.0, e = 0.5, f = 0.0;
```

y con doble precisión:

```
double i, j, k;
double g = 1.057894567, h = -0.45678901, f = -1.0;
```


NOTA: Observe que también se puede asignar valores iniciales de variables en la declaración de tipo.

9.4.2.3 VARIABLES ALFANUMÉRICAS O DE CADENA

Las variables alfanuméricas son aquellas cuyo nombre está mencionado en una declaración de tipo incluida en algún lugar del programa, por ejemplo:

```
char g[4] = "uno", h[10] = "siguiente", index = '0';
```

NOTA: observe que la cantidad indicada entre paréntesis rectangulares considera el número de caracteres de la cadena más un caracter nulo que indica el fin de la misma. La excepción son las variables con un solo elemento que se indican sin corchetes y con apóstrofos.

9.4.2.4 VARIABLES CON INDICE

Las variables con índice pueden ser de cualquiera de los tipos previamente vistos. Su uso en la ingeniería está relacionado con el manejo de arreglos en una y más dimensiones (vectores y matrices principalmente). La declaración de ellas se hace como se muestra a continuación:

```
tipo nombre_de_variable[ número_de_elementos ];
```

todos los arreglos inician con el elemento cero, por consiguiente la declaración que sigue:

```
int w[4], z[6][5];
```

indica que el vector de valores enteros w[4] cuenta con los elementos w[0] , w[1] , w[2] y w[3] , mientras que el siguiente arreglo es una matriz de 6 x 5, es decir de 30 elementos con valores de tipo entero, por ejemplo:

$$Z = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 6 & 5 & 4 & 1 \\ -1 & 3 & 2 & 0 & -2 \\ 5 & 0 & 1 & 1 & 12 \end{bmatrix} = \begin{bmatrix} Z_{0,0} & Z_{0,1} & Z_{0,2} & Z_{0,3} & Z_{0,4} \\ Z_{1,0} & Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,0} & Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,0} & Z_{3,1} & Z_{3,2} & Z_{3,3} & Z_{3,4} \\ Z_{4,0} & Z_{4,1} & Z_{4,2} & Z_{4,3} & Z_{4,4} \\ Z_{5,0} & Z_{5,1} & Z_{5,2} & Z_{5,3} & Z_{5,4} \end{bmatrix}$$

en donde, por ejemplo el elemento z[3][2] vale 5 y el elemento z[5][4] vale 12, debido a que los elementos inician con el valor z[0][0] que en este caso es un 1.

Observe que en el caso de las variables alfanuméricas se debe considerar él elemento nulo que se agrega automáticamente al final de la cadena.

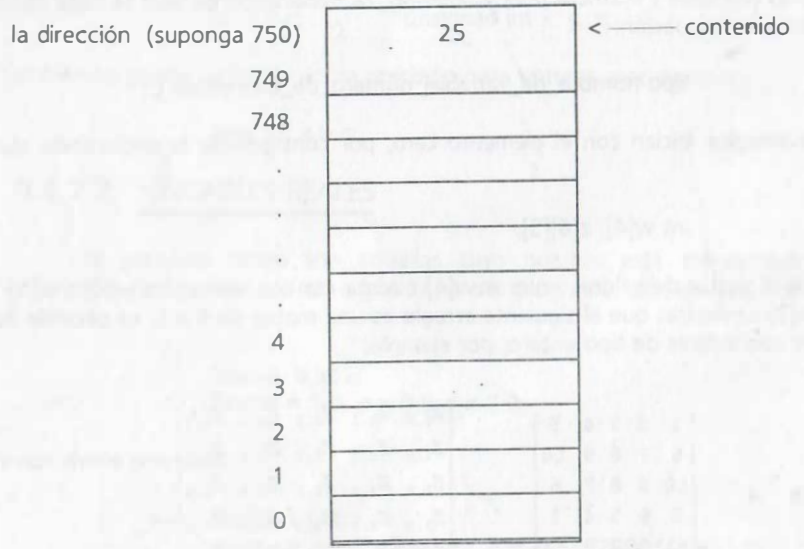
9.4.2.5 PARÁMETROS ASOCIADOS A LAS VARIABLES

Todas las variables tienen asociados tres parámetros muy importantes:

a) su nombre.
b) su dirección en memoria.
c) su contenido o valor asignado.

Recuerde que cada byte tiene asociada una dirección que permite localizar su contenido en memoria; de igual forma las variables se les asocian esos parámetros.

Físicamente la memoria puede imaginarse como un conjunto de cajones apilados y con la dirección mayor en la parte superior de la pila, considere para la variable entera a = 25; lo siguiente:



mientras que su nombre es simplemente a. De esa forma se puede hacer referencia al contenido de la variable a través de la dirección del primer byte (750).

9.5 FUNCIONES DE BIBLIOTECA

A) FUNCIONES PARA MANEJO DE CARACTERES MÁS COMUNES DEL LENGUAJE C:

strlen(v)	PROPORCIONA EL NÚMERO DE CARACTERES QUE COMPONEN EL VALOR ALFANUMÉRICO ALMACENADO EN LA VARIABLE v
strlwr(v)	CONVIERTE LOS CARACTERES ALFABÉTICOS DE LA CADENA EN MINÚSCULAS
strupr(v)	CONVIERTE LOS CARACTERES ALFABÉTICOS DE LA CADENA EN MAYÚSCULAS
strcmp(v,w)	COMPARA LA CADENA v CON LA CADENA w, REGRESA UN 0 SI LAS CADENAS SON IGUALES.
strcat(v,w)	AGREGA LA CADENA w AL FINAL DE LA CADENA v.
strcpy(v,w)	COPIA LA CADENA w EN LA CADENA v.

B) FUNCIONES ARITMÉTICAS DE BIBLIOTECA MÁS USUALES:

Las funciones aritméticas más comunes del lenguaje C son:

abs(x)	VALOR ABSOLUTO DE x (argumento entero)
atan(x)	ARCOTANGENTE DE x (con resultados con valores de -1.57 a 1.57 radianes)
exp(x)	FUNCIÓN e^x
log(x)	LOGARITMO NATURAL DE x
log10(x)	LOGARITMO BASE 10 DE x
pow(x,n)	ELEVA x AL EXPONENTE n
sin(x)	SENO DE x (en radianes)
cos(x)	COSENO DE x (en radianes)
sqrt(x)	RAÍZ CUADRADA DE x
randomize()	INDICA LA SEMILLA PARA GENERAR NUMEROS SEUDOALEATORIOS
random(x)	GENERA UN NUMERO SEUDOALEATORIO ENTERO ENTRE CERO Y (x-1).
tan(x)	TANGENTE DE x (x en radianes)

9.6 CODIFICACIÓN

Todos los elementos antes mencionados nos ayudan en la definición exacta de las fórmulas o instrucciones que permiten obtener la solución de algún problema ya que la sintaxis (correcta escritura) de todas las instrucciones de un programa no puede presentar errores para poder compilar el programa fuente.

La prioridad de ejecución de los operadores en el lenguaje C (de la más alta a la más baja) es la siguiente:

todos los tipos	
sentido de la prioridad ↓	!, ~, ++, --, - (tipo)
	*, /, %
	+, -
	<<, >>
	<, <=, >, >=
	==, !=
	&
	^
	|
	&&
	||
	?
	=, +=, -=, *=, /=

Sin embargo, cuando una expresión contiene funciones de biblioteca éstas se evalúan primero. Además, la presencia de paréntesis redondos o rectangulares obliga a evaluar primero el contenido que encierran. Empecemos pues con la codificación (escritura en C) de expresiones sencillas y su evaluación por la computadora:

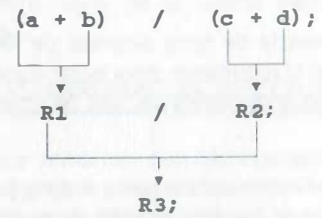
$$e1 = \frac{a + b}{c + d} \qquad \texttt{e1 = (a + b)/(c + d);}$$

$$e2 = a + \frac{b}{c + d} \qquad \texttt{e2 = a + b/(c + d);}$$

$$e3 = a + \frac{b}{c} + d \qquad \texttt{e3 = a + b/c + d ;}$$

$$e4 = \frac{a + b}{c} + d \qquad \texttt{e4 = (a + b)/c + d;}$$

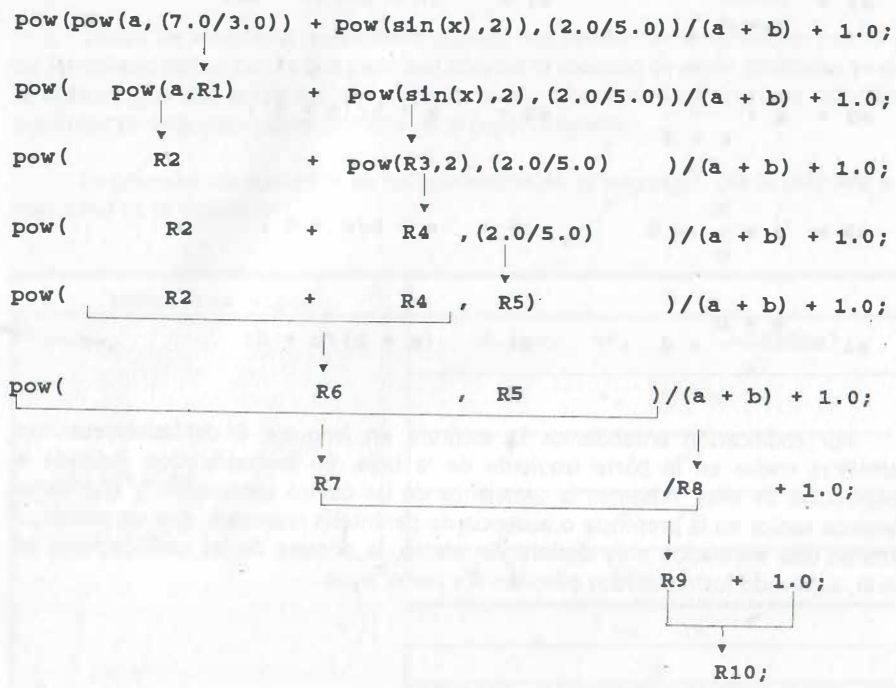
Por codificación entendemos la escritura en lenguaje C de las expresiones aritméticas dadas en la parte izquierda de la hoja. En la codificación indicada a continuación de ellas, notamos la semejanza de las cuatro expresiones y que dicha diferencia radica en la presencia o ausencia de paréntesis redondos, que sin embargo permiten una evaluación muy distinta. En efecto, la primera de las codificaciones se evalúa, siguiendo los resultados parciales R's como sigue:



Veamos otro ejemplo:

$$\frac{\left[a^{\frac{7}{3}} + \sin^2 x\right]^{\frac{2}{5}}}{a + b} + 1$$

La cual se codifica y evalúa apoyándose en resultados parciales (R's) como sigue:



Debe notarse que se evitó la mezcla de tipos distintos de variables incluyendo punto decimal en las constantes que así lo requirieron para evitar divisiones enteras, con excepción del exponente de la función trigonométrica. Las recomendaciones son las siguientes:

a) Todo exponente que pueda conservarse como entero debe expresarse así, ya que para su evaluación se usa el concepto de multiplicación mientras que los exponentes reales se evalúan por medio de series y puede perderse algo de precisión.

b) Usar punto decimal en exponentes que implican división de cantidades, para evitar que se consideren como enteras (en algunos lenguajes es importante expresar esas cantidades así para evitar la aritmética entera que sólo produce como resultado cantidades enteras).

Con las expresiones se obtiene lo que se conoce como instrucción de reemplazo o almacén:

V = expresión aritmética

que constituye el ejemplo más común de instrucción ejecutable.

Recuerde que la instrucción de reemplazo, es aquella que calcula el resultado numérico de la expresión aritmética indicada a la derecha del signo igual (signo conocido como orden de reemplazo) para, a continuación ***almacenar*** dicho valor en la variable indicada a la izquierda de ese igual. Del lado izquierdo únicamente se puede indicar el nombre de una sola variable.

Como aplicaciones de ese concepto, en programación frecuentemente se encuentran expresiones como:

C = C + constante

S = S + expresión aritmética

*F = F * contador*

que al emplearse dentro de los ciclos iterativos nos permiten: contar, obtener sumatorias y factoriales (vea estos conceptos en el tema de diagramas de flujo).

Al elaborar un algoritmo tenga en mente que la secuencia u orden de ejecución de las instrucciones de un programa estructurado siempre será descendente, desde la primera instrucción colocada en la parte alta de la página, hasta la última instrucción de su conjunto (parte baja de la última página), evitando ramificaciones hacia instrucciones previas.

9.7 OPERADORES ABREVIADOS TÍPICOS DEL LENGUAJE C

Es conveniente remarcar que los operadores incremento y decremento son característicos del lenguaje C y no se encuentran en otros lenguajes, con ellos se tienen formas abreviadas de contadores o variables que cuentan de uno en uno (el primero es un contador que incrementa de uno en uno y el segundo un contador que disminuye de uno en uno). De esta modalidad se tienen cuatro posibilidades, por ejemplo con las expresiones de reemplazo siguientes:

y = x++;	el valor de x primero se asigna y luego se incrementa, es decir primero y = x; seguido de x = x + 1;
y = ++x;	el valor de x primero se incrementa y luego se asigna es decir primero x = x + 1; seguido de y = x;

y = x- -; el valor de x primero se asigna y luego se decrementa es decir primero y = x; seguido de x = x - 1;

y = - -x; el valor de x primero se decrementa y luego se asigna es decir primero x = x - 1; seguido de y = x;

Además existe la posibilidad de expresar en forma abreviada a todos los operadores binarios (los que actúan sobre dos operandos como en el caso de contadores, sumadores y factoriales), de tal manera que son equivalentes los siguientes incisos:

a) variable = variable operador expresión;

b) variable operador = expresión;

Por consiguiente, las expresiones de los siguientes incisos son equivalentes:

1) x = x + c; puede expresarse: x + = c;

2) x = x - c; puede expresarse: x - = c;

3) x = x * c; puede expresarse: x * = c;

4) x = x / c; puede expresarse: x / = c;

Por último se menciona que también se acepta la asignación múltiple de un valor a varias variables, es decir, es válido lo siguiente:

a = b = c = d = 5; donde se indica que las cuatro variables valen cinco.

Como ejemplo de aplicación se sugiere correr el siguiente programa en el que se muestra como resultados la variación de incrementos:

CONVERSIÓN DE TIPOS PARA SEGMENTOS DE EXPRESIONES (CASTING)

En muchas expresiones que involucran diversas variables se recomienda no mezclar modos de variables (enteras con reales, etc.) con el fin de evitar posibles resultados erróneos. La acción que permite considerar momentáneamente una variable como si fuese de otro tipo se conoce como *casting*; cuando se le usa se convierte el resultado de un segmento de la expresión a otro tipo de valor (por ejemplo una variable entera se le considera real). Cuando se requiere afectar el tipo de un segmento de la expresión, se le encierra entre paréntesis anteponéniendole también entre paréntesis el tipo de valores al que se le convierte momentáneamente. El ámbito de esta operación conocida como *casting* siempre está circunscrita a la expresión entre paréntesis o variable que sigue a continuación, antes del siguiente operando. Considere los siguientes casos:

```
main()
{
  int a=1,b=2,c=3;
  int r1,r2;
  float d=4.5,e=5.5,f=6.5;
  int *g;          /* declaración de apuntador a variable entera */
  float *h;        /* declaración de apuntador a variable real */
  float r3,r4;

  r3 = (float)c/(2.0 + 5.0);

  /* la operación anterior considera como real a la variable entera c,
     de tal manera que el resultado de la división corresponda
     exactamente a:

     3.0/(2.0 + 5.0)    */

  r1 = ((int)(d + e) % 3;

  /* la división entera se efectúa considerando la expresión real (d + e)
     como entera, de tal manera que lo que se obtendrá será el resíduo
     de la división de 10 entre 3, dando como resultado 1 (entero). */

  g = (int *)&d;

  /* se asegura que se almacena la dirección como entero de la variable
     real d (consultar el capítulo sobre operaciones sobre apuntadores).
}
```

9.8 ENTRADA SALIDA

Prácticamente todos los programas requieren procesar información proveniente de fuentes externas para obtener la solución de un problema específico. Esa información constituye los datos de entrada del algoritmo que definió nuestro programa y es mediante ellos que se obtendrá la salida o solución correspondiente:

De ahí la importancia de proporcionar adecuadamente los valores o datos de aquellas variables del programa que deba leer la computadora; además de la importancia de incluir mensajes en las ordenes de salida que nos indiquen la naturaleza o nombre de los resultados obtenidos.

Definiremos entonces las formas generales de lectura y escritura en lenguaje C que nos permiten recibir resultados o enviar datos que solicitan las variables de nuestro programa cuando se ejecuta en la computadora.

ENTRADA DE DATOS

En lenguaje C se emplea una función para lectura de datos que nos permite incluir las facilidades necesarias para enviar datos a la computadora. Una función u orden de lectura menciona especificaciones de campo que indican como leer los valores que corresponden a las variables involucradas en la entrada de información.

LECTURA DE VALORES POR TECLADO (FUNCIÓN scanf();)

scanf("especificaciones de formato", APUNTADORES DE LAS VARIABLES);

Esta función direcciona automáticamente la lectura de valores por el teclado de la microcomputadora.

Las especificaciones de formato van precedidas de un signo % y a continuación alguna de las opciones siguientes:

a) opcionalmente para las cantidades, el tamaño mínimo del dato a leer, especificando el total de dígitos fraccionarios en caso de que los haya, a continuación un caracter que indica el tipo de dato (entero, flotante o de cadena). Las especificaciones de campo más empleadas son:

%c	para leer un caracter único
%d	para leer un entero en base diez
%e	para leer una cantidad exponencial de base 10 (punto flotante)
%f	para leer una cantidad real de base 10 (punto flotante)
%h	para leer un entero corto
%i	para leer un entero en base diez
%o	para leer una cantidad en base 8
%p	para leer un apuntador
%s	para leer una cadena
%x	para leer una cantidad hexadecimal

El lenguaje C emplea apuntadores en lectura de toda cantidad numérica debido a que asigna dichos valores a través de las direcciones de las variables, por tal motivo los nombres de ellas van precedidas del símbolo & (con lo que se indica que se usan apuntadores). En el caso de cadenas, los nombres de las variables son en si mismos apuntadores y no requieren dicho símbolo.

Por ejemplo la lectura de los valores de tres variables:

```
/* lectura de tres valores de variables enteras */
#include <stdio.h>
main(void)
{
   int e,m,c;

   scanf("%d%d%d",&e,&m,&c);

   instrucciones para
   proceso
   de la información
   e impresión de
   resultados
                              ;
 return 0;
}
```

Recuerde que en este caso los valores se teclean separados por un espacio en blanco o tabulador y al final un ENTER o también cada dato seguido de un ENTER.

En el ejemplo anterior debe observarse que no se incluyó caracter alguno entre las especificaciones de campo. Si se incluye algo, eso mismo deberá indicarse junto con

los valores a teclear, por ejemplo:

scanf("%d,%d,%d",&e,&m,&c);

está indicando a C que lea un entero, descarte una coma, lea otro entero, descarte una segunda coma y lea un último entero (en este caso se teclean los datos separados por comas y al final se da ENTER). Debe tenerse precaución en la lectura para evitar casos de confusión como pudiera ser el siguiente:

scanf("%s ",nombre_cadena);

pues ahí, además del valor de la cadena, scanf está esperando a que se indique el espacio en blanco que sigue a la especificación s para concluir la lectura; es decir que de esa forma se indica que se lea y descarte caracteres de espacios, tabulaciones y saltos de línea.

NOTA:

No es recomendable indicar las ordenes de lectura sin la presencia de ordenes de escritura que nos indiquen los valores que deberán proporcionarse por teclado (lectura indicada por el scanf();). En ocasiones se requerirá borrar la memoria temporal que usa scanf(); para evitar errores antes de volver a leer. Para ello usé la función fflush(stdin);

CONTROL DE PROCESO O FLUJO MEDIANTE getch(); o getche(); (INSTRUCCIONES DE LECTURA)

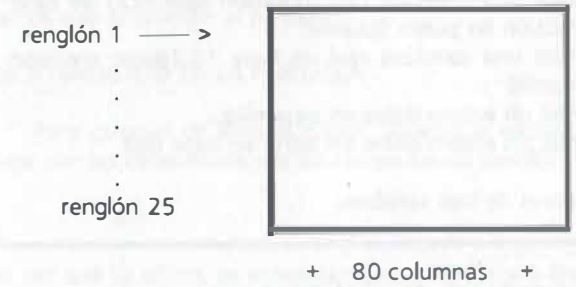
Estas funciones de lectura de un caracter se emplean cuando se requiere detener un proceso, o ejecutar otras tareas, pues el proceso se detiene en espera de que se oprima alguna tecla.

En el caso de getch(); el caracter correspondiente a la tecla no se despliega en pantalla (no hay eco). Cada vez que se ejecuta esta instrucción se detecta si se ha tocado alguna de las teclas y, con base en ello, se determina la acción a seguir. Su uso más común es colocarla inmediatamente después de una función de escritura para detener el proceso y visualizar el despliegue de un mensaje y valores en pantalla. Cuando se emplea se debe agregar el archivo de cabecera #include <conio.h>.

La función getche(); funciona igual que la anterior con la diferencia de que devuelve el caracter teclado.

ESCRITURA DE MENSAJES Y VALORES DE VARIABLES (FUNCIÓN printf())

La pantalla es el medio más utilizado para mostrar la salida de resultados, sin embargo, también se emplean las salidas impresas. El tamaño típico de la pantalla que se muestra a continuación:



se comenta que el tamaño típico de una hoja es de 54 renglones por 80 columnas.

La función de escritura nos permite incluir mensajes para visualizar o imprimir adecuadamente los resultados obtenidos por nuestro programa. Esta función utiliza también especificaciones de campo que corresponden al tipo de variables mencionadas, cuyo valor se envía como salida. La sintaxis de la función printf(); es la siguiente:

printf("mensajes y <especificaciones de campo>", <LISTA DE VARIABLES>);

Esta función despliega automáticamente los mensajes y los valores de las variables mencionadas en la lista. Es importante remarcar que el tipo de las especificaciones de campo deberán coincidir con el tipo de las variables a escribir y que los mensajes por lo general identifican a los valores que se escriben.

Las especificaciones de formato o campo van precedidas de un signo % y a continuación alguna de las opciones siguientes:

a) Un signo menos (-) para indicar alineación a la izquierda o un signo mas (+) para indicar que la cantidad se desplegará incluyendo un signo mas (+) y justificada a la derecha.

b) Un cero (0) para ordenar que se rellene con ceros los espacios a la izquierda de las cantidades.

Enseguida y opcionalmente para las cantidades, el ancho o tamaño mínimo del dato a imprimir, especificando el total de dígitos fraccionarios en caso de que los haya.

Por último un caracter que indica el tipo de dato (entero, flotante o de cadena). Los caracteres usados son los mismos que se emplean en la lectura, sin embargo, ahora si es importante agregar la posibilidad de valores en doble precisión con el modificador l. Las especificaciones de formato o campo en su forma más simple son:

%le	para escribir una cantidad real (notación científica) de base 10 (doble precisión en punto flotante)
%lf	para escribir una cantidad real de base 10 (doble precisión en punto flotante)
%li	para escribir un entero doble en base diez
%lu	para escribir un entero doble sin signo en base diez

EJEMPLO: Escritura de los valores de tres variables:

```
/* escritura de tres valores de variables enteras */
#include <stdio.h>
#include <conio.h>
main(void)
{
   int e,m,c;
   e = 5;
   m = 10;
   c = 15;

   printf(" e = ");
   printf("%d",e);
   printf("\n m = ");
   printf("%d",m);
   printf("\n c = ");
   printf("%d",c);
   printf("\n\n Para continuar oprima alguna tecla ");

/*  observe que a continuación se indica la función
    de lectura getche(); para detener el proceso */

    getche();
    return 0;
}
```

En algunos casos se requiere combinar mensajes y especificaciones de campo en una sola orden; los valores de variables se escriben ajustándose a la derecha del campo que les corresponde y no es necesario especificar el signo + de alineación a la derecha, pero se puede pedir que dichos valores se ajusten a la izquierda, si requiere que se coloque un signo - inmediatamente después del caracter %. Por ejemplo:


```
printf(" e = %-08d m = %08d", e, m);
```

que indica que el valor de e va precedido de un mensaje y que deberá escribirse en un campo de 8 espacios de ancho, empezando por la izquierda; el valor de m también incluye un mensaje y se escribirá en un campo de 8 espacios, pero empezando por la derecha de dicho campo; en caso de valores enteros sólo se rellena con ceros los espacios que anteceden al número.

POSICIONAMIENTO EN LA PANTALLA

Para cambiar de línea el cursor, durante la escritura de mensajes o valores, se incluye con las especificaciones de campo de un printf(); , la instrucción:

```
\n
```

cada vez que se utiliza, se indica cambiar el cursor a la línea siguiente; de igual forma cuando se incluye el tabulador:

```
\t
```

se indica desplazar el cursor 6 espacios horizontalmente.

CONVERSIÓN AUTOMÁTICA DE CANTIDADES DECIMALES A OCTAL Y HEXADECIMAL

Toda cantidad entera en base 10 que se ordena escribir mediante especificaciones de tipo octal y hexadecimal es convertida automáticamente a esas bases antes de imprimirse. Como ejemplo:

```
printf("\nValor decimal = %d  Valor octal = %o ",15,15);

printf("\nValor decimal = %d  Valor hexadecimal = %x ",15,15);
```

INDICACIONES ADICIONALES

Para retroceder un espacio horizontalmente:

```
\b
```

Para emitir un pitido:

```
\a
```

Como ejemplo de operadores reducidos y la función printf(); a continuación se muestra el siguiente programa en turboc:

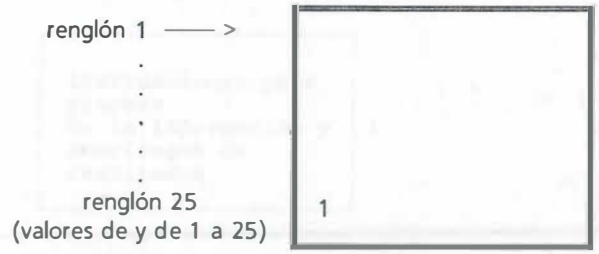
```
#include <stdio.h>
void main()              /* empieza la funcion principal */
{
 int a,b,c,d,e;
 int k;
 a=b=c=d=e=0;
 clrscr();
  printf("\n Valores iniciales de las variables:   a=b=c=d=e=0 \n");
  printf("\n Al escibir a+ +, a     resulta:\n");
  printf("\n a+ +  = %d (pues escribe el valor actual de  a  y continuación lo incrementa) ",a+ +);
  printf("\n a    = %d (aquí escribe el valor incrementado al final del paso anterior)\n", a);

  printf("\n Al escibir   + +b, b   resulta:\n");
  printf("\n + +b  = %d (pues primero incrementa el valor actual de  a  y luego lo escribe) ",+ +b);
  printf("\n b    = %d (se escribe el valor incrementado al inicio del paso anterior)\n", b);

  printf("\n Al escibir   c--, c   resulta:\n");
  printf("\n c--  = %d (pues escribe el valor actual de  b  y continuación lo disminuye) ",b--);
  printf("\n c    = %d (representa el valor disminuido al final del paso anterior)\n", b);

  printf("\n Al escibir   --d, d   resulta:\n");
  printf("\n --d  = %d (pues primero disminuye el valor actual de  d  y luego lo escribe) ",--d);
  printf("\n d    = %d (dado que escribe el valor disminuido al inicio del paso anterior)\n", d);

  printf("\n \t\t Para regresar al programa oprime cualquier tecla\n");
  getch();
}
```


ESCRITURA EN POSICIONES ESPECÍFICAS DE LA PANTALLA

En Turbo C, la función que permite ubicar el cursor en una posición específica de la pantalla es la siguiente:

gotoxy(coordenada_x, coordenada_y);

que coloca el cursor en las coordenadas (x,y) indicadas (sólo valores enteros y válidos, bajo el entendido de que se emplean 80 columnas y 25 renglones como máximo en la pantalla).



De esa manera, se elige el valor horizontal o de x, a continuación, a partir de la parte alta de la pantalla, se cuenta el renglón en el que se escribirá y que corresponde al valor de y. Por ejemplo, el valor unitario que se exhibe empleó gotoxy(2,24); seguida de la función printf("1"); (posición 2 del renglón 24).

Además para manipulación de lo desplegado en pantalla se tiene las funciones:

clrscr(); para borrado de la pantalla (prototipo en conio.h)

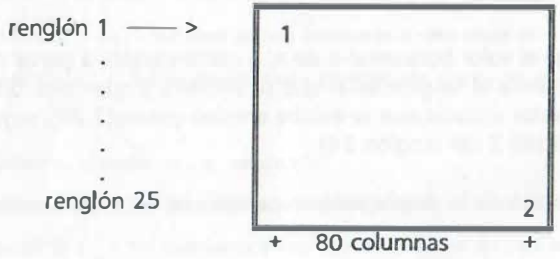
clreol(); para borrado de parte de una línea (prototipo en conio.h)

delline(); para borrado de una linea en una ventana (prototipo en conio.h)

Como ejemplo, a continuación se codifica un programa que despliega mensajes por medio de la función printf(); que muestra sus valores o mensajes a partir del renglón y columna especificados por gotoxy(x, y);

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

main(void)
{

   int e,m;
   clrscr();

   e = 1;
   m = 2;

   gotoxy(1, 1);
   printf("%d", e);

   gotoxy(79, 24);
   printf("%d", m);

   return 0;
}
```

El resultado del ejemplo anterior produce en la pantalla la salida siguiente:



EJEMPLO Combinación de mensaje de salida con printf(); y entrada de valores con scanf();:

```
#include <stdio.h>
#include <conio.h>
main(void)
{
   int e,m,c;
   clrscr();

   printf("dame los valores de e,m,c ";
   scanf("%d,%d,%d",&e,&m,&c);


   instrucciones para
   proceso
   de la información y  ;
   despliegue de
   resultados


   getche();
}
```

El suministro de los valores se realiza dando los valores separados por comas y oprimiendo al final la tecla ENTER, para los ejemplos anteriores aparece en pantalla:

dame los valores de e,m,c (a lo que, por ejemplo se responde lo siguiente:)

```
10, -10, 100
```

La lectura se realiza asociando cada uno de los valores indicados con los apuntadores de las variables de la lista del scanf(); (en estricto orden de aparición, uno a uno). Lo único necesario es que haya concordancia con los modos de las variables y su especificación de tipo (no es válido especificar valores alfanuméricos para variables reales o enteras, o valores de tipo flotante en lugar de valores enteros).

Como se observa, la presencia de una orden de lectura implica que el proceso se detiene hasta que se suministran datos. Por consiguiente, también existe la posibilidad de usar este tipo de órdenes para detener el proceso hasta que se teclee algún caracter y de esta manera controlar la ejecución, por ejemplo, para visualizar

resultados. Dentro de esa clase de instrucciones se encuentran: getch(); getche();

Por el momento se ejemplifica una aplicación del getche(); mediante diagrama de flujo.

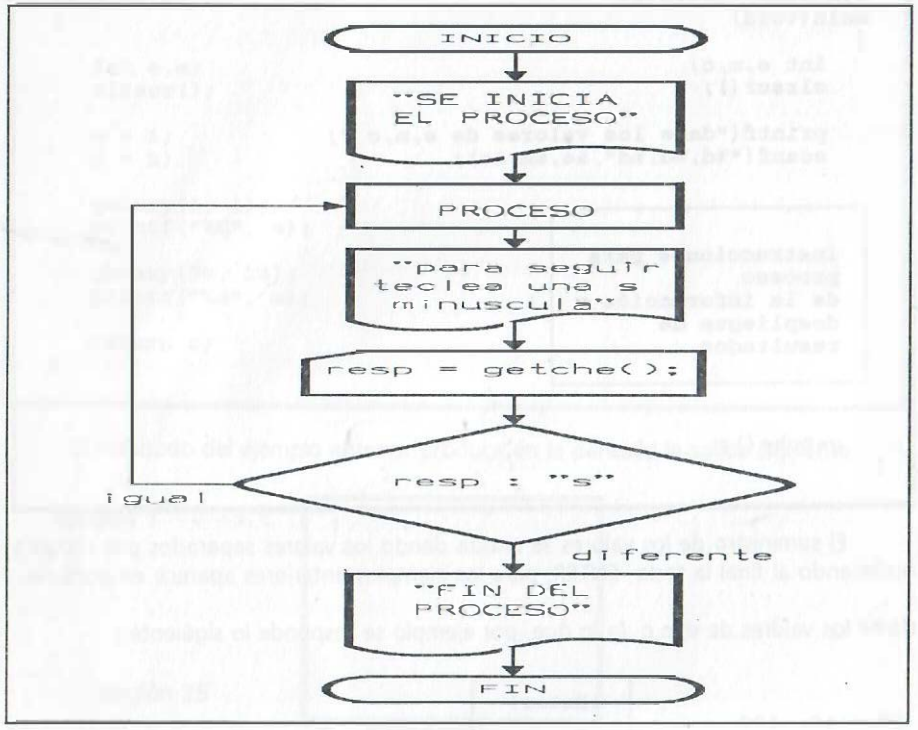


DIAGRAMA NO ESTRUCTURADO O TRADICIONAL

ESCRITURA CON FORMATO ESPECÍFICO

Cuando se requieren escribir cantidades con un número específico de decimales o parte entera o con algunos caracteres especiales, se emplea la opción de ancho de campo en la especificación de escritura, por ejemplo considere la cantidad -3525.78, se observa que incluyendo el signo, el tamaño total para escribirla es de 8 columnas y de ellas, las dos últimas son para los decimales, por consiguiente la instrucción queda:

```
printf("%8.2f",-3525.78);
```


en la que se entiende que el tamaño mínimo del campo es de 8 columnas y la cantidad se escribirá con dos dígitos decimales.

I) FUNCIONES ADICIONALES

Las funciones básicas que complementan la entrada - salida son:

getchar(); regresa un caracter o fin de archivo (EOF)
putchar(); despliega un caracter

getc(); regresa un caracter o fin de archivo (EOF)
putc(); despliega un caracter o fin de archivo (EOF)

gets(); regresa una cadena y un apuntador de la cadena
puts(); despliega una cadena

cgets(); lee una cadena
cputs(); despliega una cadena

J) ESCRITURA DIRIGIDA A LA IMPRESORA

La impresión de todo lo que aparece en la pantalla se logra oprimiendo la tecla:

Impr Pant (PRTSC)

NOTA COMPLEMENTARIA:

Además de la importancia de documentar el programa mediante mensajes que identifiquen la naturaleza de variables, tanto de entrada como de salida, es importante incluir comentarios (que únicamente brinden información sobre lo que se hace en el proceso) en el cuerpo del programa. Recuerde que en el lenguaje C los comentarios se indican : /* comentario */.

Como ejemplo de todo lo anterior, considere el siguiente programa que define dos funciones además del main():

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

/* Se declaran los tipos de funciones que se emplean en el programa */

void portada(void);
void aritmetica(void);

/* inicia la función principal que llama a dos funciones creadas */

void main(void)
{
   clrscr();
   /* borra la pantalla */
   portada();
   aritmetica();
   getche();
   /* getche se utiliza para leer un caracter del teclado */
}
void portada(void)
{
   gotoxy(18,2);
   printf("═══════════════════════════════════════════");
   gotoxy(18,3);
   printf("                                           ");
   gotoxy(18,4);
   printf("    PROGRAMA ELABORADO POR: XXXXXXXXXXX YYYYY   ");
   gotoxy(18,5);
   printf("                                           ");
   gotoxy(18,6);
   printf("    GRUPO 03 DE COMPUTADORAS Y PROGRAMACION     ");
   gotoxy(18,7);
   printf("                                           ");
   gotoxy(18,8);
   printf("    8:30 - 10:00 HRS                       ");
   gotoxy(18,9);
   printf("                                           ");
   gotoxy(18,10);
   printf("                                           ");
   gotoxy(18,11);
   printf("    OBJETIVO: MOSTRAR LA ARITMETICA REAL       ");
   gotoxy(18,12);
   printf("                                           ");
   gotoxy(18,13);
   printf("              Y LA ARITMETICA ENTERA.      ");
   gotoxy(18,14);
   printf("                                           ");
   gotoxy(18,15);
   printf("                                           ");
   gotoxy(18,16);
   printf("    Para continuar oprima cualquier tecla...   ");
   gotoxy(18,17);
   printf("                                           ");
   gotoxy(18,18);
   printf("═══════════════════════════════════════════");
   getche();
   }
void aritmetica(void)
{
   int a=1,b=2,c;
   float d=1.0,e=2.0,f;
   c= a/b;
   f= d/e;
   printf("\n\n Aritmetica entera %d/%d = %d",a,b,c);
   printf("\n Aritmetica real %f/%f = %f",d,e,f);
   printf("\n\n Para regresar al listado del programa oprima cualquier tecla");
}
```


DESPLIEGUE DE ESCRITURA Y FONDOS CON COLORES DIVERSOS

Para las microcomputadoras con monitor a color como es el VGA y otros se tiene la posibilidad de desplegar textos en pantalla empleando colores, tanto para el primer plano (escritura) como para el fondo. Las funciones son:

textcolor(color);
textbackground(color);

La primera función permite fijar el color del texto en alguna de 16 tonalidades distintas (color es un valor entero del 0 al 15). La segunda fija el fondo en alguno de 8 colores (color en esta caso es un valor entero del 0 al 7). Una vez especificado lo anterior el despliegue de textos en color se realiza con:

cprintf(" texto y especificaciones de campo ",variables);
putch(" texto ");

En las directivas se debe agregar el prototipo #conio.h ; Los valores y colores asociados se indican a continuación:

0 produce fondo negro
1 produce fondo azul marino claro
2 produce fondo verde perico
3 produce fondo verde claro
4 produce fondo rojo
5 produce fondo morado o violeta
6 produce fondo café claro
7 produce fondo gris claro

En el caso del color de las letras (primer plano) puede fijarse dentro de diez y seis tonos según se indica a continuación:

0 a 7 los colores anteriores
8 produce fondo gris ligeramente más fuerte
9 produce fondo azul marino claro
10 produce fondo verde claro
11 produce fondo azul claro
12 produce fondo rojo claro
13 produce fondo rosa mexicano
14 produce fondo amarillo
15 produce fondo blanco

Para el texto se tiene además tres posibilidades:

1. La posibilidad de agregar la característica de parpadeo agregando la instrucción BLINK (con letras mayúsculas) en la función de escritura:

```
textcolor(color + BLINK);
```

2. Intensidad baja para las letras con la función:

```
lowvideo();
```

3. Intensidad alta para las letras con la función:

```
highvideo();
```

Cada vez que se requiera reestablecer el color normal del texto se debe emplear la función:

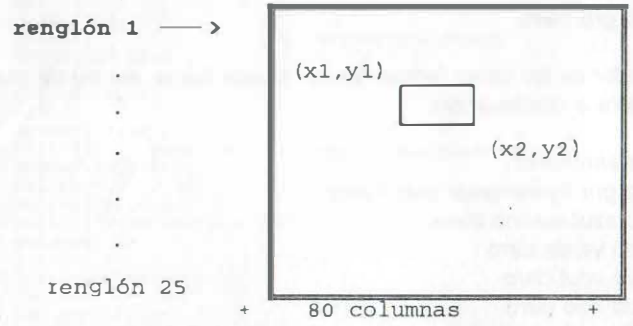
```
normvideo();
```

ESPECIFICACIÓN DE VENTANAS PARA ESCRITURA DE TEXTO

Para especificar una porción de la pantalla para escribir información, tal vez en colores llamativos o con letras de doble tamaño (ventana), se emplea la función:

```
window(x1, y1, x2, y2);
```

en la cual se especifican las coordenadas mediante valores enteros dentro del rango máximo de la pantalla:



Como muestra se incluyen programas para despliegue de tonos y ventanas después de estudiar el ciclo iterativo for();.

USO DE PROGRAMAS EJECUTABLES COMO PARTE DE OTRO PROGRAMA

Una vez que el programa ha sido ejecutado y se ha obtenido un archivo con extensión .EXE, como por ejemplo una portada de presentación de una tarea que se desea incluir en todos los programas subsiguientes que se elaboran, no es necesario teclear todas las instrucciones de nuevo, basta con incluir las instrucciones siguientes:

```
#include <process.h>
system("c:nombre.exe");
```

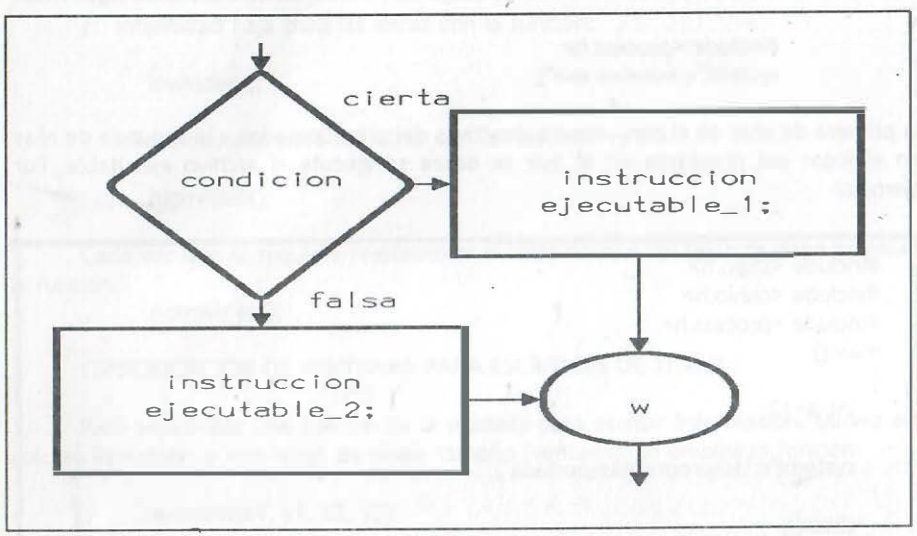
la primera de ellas en el conjunto de directivas del preprocesador y la segunda de ellas en el lugar del programa en el que se desea se ejecute el archivo ejecutable. Por ejemplo:

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
main()
{
  int a=12;

  system("c:\tc\programas\portada");

  clrscr();

  printf("Dame un valor entero  ");

  scanf("%d",&a);

  printf("El valor de a = %d",a);

  getche();
}
```

La instrucción *system* indica la trayectoria (directorio y subdirectorios en su caso) en donde se encuentra almacenado el archivo ejecutable. En el ejemplo se indica que a partir del directorio raiz *c:*, debe buscar en el directorio denominado *tc* y luego en el subdirectorio *programas*. Observe que no es necesario indicar la extensión EXE en el nombre del archivo *portada*. La instrucción *system* permite también ejecutar archivos con extensión *.bat* o *.com*,además de desplegar el directorio con *DIR*.

9.9 PREGUNTA LÓGICA if - else SENCILLA

Permite calcular una de dos expresiones aritméticas. Si la condición es cierta evalúa la primera expresión, cuando la condición es falsa calcula la instrucción que sigue al else. Su forma general es la siguiente:



SINTAXIS:

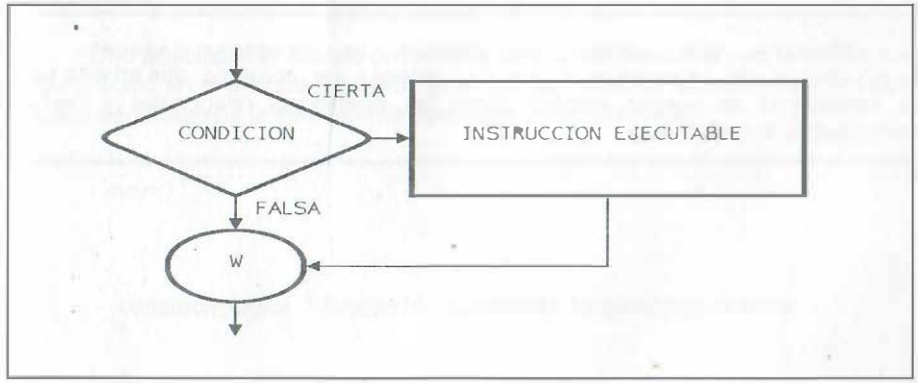
if (condición) instrucción ejecutable_1;
else instrucción ejecutable_2;

en ella la *instrucción ejecutable_1;* corresponde al caso de SI se cumple la condición y la *instrucción ejecutable_2;* al caso de *NO* se cumple la condición. Una vez que se ha evaluado alguna de ellas, el control de ejecución pasa a la línea siguiente del programa. Por ejemplo:


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

main()
{
     float a=1.0,b,c=2.0,d=3.0;
     int y;
     clrscr();       /* borra la pantalla */
     randomize();    /* para generar un número aleatorio
                        con <time.h> incluida en
                        <stlib.h> */

     y = random(2);  /* permite generar un cero o un 1 */
     b = y;
     .
     /*   instrucciones del proceso;   */
     .
     if( (a < b) && (c > d) )  e = b + d;
     else e = b - d;
     r = z;  /* esta es la línea siguiente del prog. */
     .
     .
}
```

que se interpreta: si *a* es menor que *b* y *c* mayor que *d*, entonces calcula y almacena en e la suma de ***b más d,*** *en caso contrario calcula y almacena en* e la resta de *b menos d.*

De esta pregunta se deriva el caso particular sin la opción ***else****, es decir:*



en el que únicamente se considera la opción de calcular la expresión dada si se cumple la condición y en caso negativo, continuar con el resto del programa. Considere por ejemplo la codificación siguiente:

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

main()
{
   float a=1.0,b,c=2.0,d=3.0;
   int y;
   clrscr();       /* borra la pantalla */
   randomize();    /* para generar un número aleatorio
                      con <time.h> incluida en
                <stlib.h> */

   y = random(2);  /* genera un cero o un 1 */
   b = y;
       .
   /*   instrucciones del proceso;    */

   printf("dame el valor de b "); /* despliega mensaje */
   scanf("%f",&b);                /* lectura de dato   */


   if( (a < b) && (c > d) )  e = b + d;
       .
       .
}
```

Observe que cuando se cumple la pregunta ejecuta la instrucción de la suma e = b + d;. Además se comenta que en este tipo de condiciones se pueden emplear todos los operadores relacionales y lógicos mencionados anteriormente.

9.10 EL OPERADOR ? COMO PREGUNTA if- else SIMPLE

La forma abreviada que ofrece el lenguaje C para la pregunta sencilla tiene la siguiente:

SINTAXIS:

condición ? expresión_1 : expresión-2;

en la cual se evalúa la *condición* y si el resultado es verdadero, el valor de la *expresión_1* se convierte en el resultado de esa instrucción, en caso contrario, *expresión_2* será el resultado que se obtiene para la pregunta simple abreviada.

Por ejemplo para el caso de una instrucción de reemplazo:

```
main()
{
   .
   .
   e = (a < b) && (c > d) ) ? b + d: b - d;
   .
   .
}
```

Que se interpreta exactamente igual que el primer ejemplo mostrado, es decir, si la condición se cumple, se almacena en e la suma y en caso contrario la resta.

Otra posibilidad es cuando únicamente se ejecutan funciones y no se asigna valor alguno como en el ejemplo siguiente en el que se considera la evaluación de valores lógicos de variables y la ejecución de funciones como alternativas:

```
main()
{
   .
   .
   condición_logica ? funcion1(argumentos): funcion2(argumentos);
   .
   .
}
```

En el cual si el resultado de **condición_lógica** es un 1 (CIERTO), se llama a **funcion1()** y si el resultado es un 0 (FALSO), se llama a **funcion2()**.

Considere ahora otro ejemplo en el que se despliegan mensajes de saludo o despedida dependiendo de un número aleatorio:

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

/* inicia la función principal que llama a funciones
   de biblioteca */

main()
{
   int y;
   clrscr();       /* borra la pantalla */
   randomize();    /* permite generar un número aleatorio con
                      <time.h> */

   y = random(2);    /* permite generar un cero o un 1 */

   y ? printf(" Hola "): printf(" Adios ");

   getche();
   /* getche es una función que espera hasta que se presione
      una tecla */
}
```

Considere el mismo ejemplo llamando a dos funciones que el programador define:


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

/* Se declaran los tipos de funciones que se emplean en el
   programa */

void saludo(void);
void despedida(void);

/* inicia la función principal que llama a dos funciones
   creadas */

main()
{
   int y;
   clrscr();       /* borra la pantalla */
   randomize();    /* permite generar un número aleatorio con
                      <time.h> */

   y = random(2);    /* permite generar un cero o un 1 */

   y ? saludo(): despedida();

   getche();
   /* getche es una función que espera hasta que se presione
      una tecla */
}
void saludo(void)
{

   printf("\n\n Bienvenido a la sesión de trabajo");
   getche();

}
void despedida(void)
{
   printf("\n\n Hasta la próxima");
}
```

9.11 ELEMENTO ESTRUCTURADO for

Por lo general se considera a este elemento como un ciclo controlado por contador o que permite repetir procesos dependiendo de una condición. Su forma general es la siguiente:

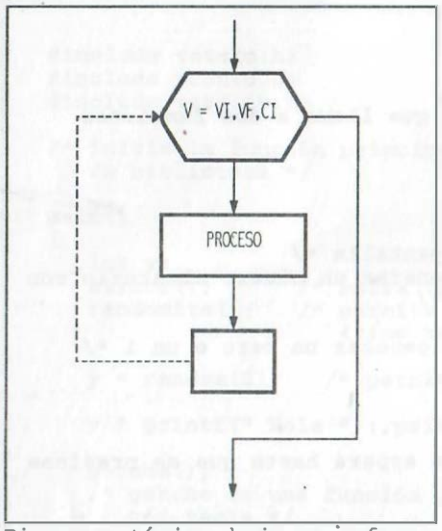


Diagrama típico bajo un enfoque general a todos los lenguajes

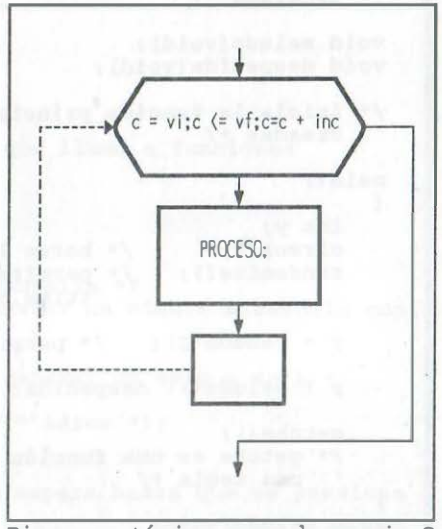


Diagrama típico para lenguaje C

SINTAXIS:

```
for(contador=valor_inicial; condición; contador=contador+incremento)
{
  instrucciones ejecutables;
}
```

Este elemento de la programación estructurada involucra una condición que controla, mediante un contador, el número de veces que se repite un proceso. Como es una estructura simplificada, facilita la repetición de procesos controlados por contador. En nuestro enfoque se considerará cualquiera de los dos diagramas anteriores y la sintaxis general siguiente:

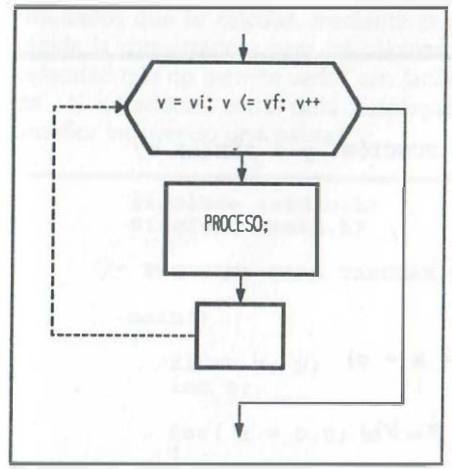


DIAGRAMA ESTRUCTURADO

```
for(v = vi; v <= vf; v++)
{
    INSTRUCCIONES
    DEL PROCESO      ;

}
```

CODIFICACIÓN

Cuando el control de ejecución pasa a este elemento estructurado, la variable *v* toma el valor inicial almacenado en *vi* y el proceso dentro del ciclo se ejecuta mientras se cumpla la condición, al final del proceso el control regresa automáticamente a la primera instrucción del ciclo para incrementar el valor de *v en forma unitaria, según indica el operador incremento v + +*; si el nuevo valor de *v* es menor que el valor final indicado en *vf*, nuevamente se repite el proceso y así sucesivamente hasta que el valor de *v* no sea menor que el valor de *vf*, en cuyo momento el control sale del ciclo, para continuar con la ejecución de la instrucción que sigue a la llave que delimita el ámbito del ciclo.

Los valores *vi* , *vf* , *v + + (incremento unitario)*, pueden ser asignados mediante expresiones que involucren diversas variables o, inclusive por cantidades específicas en lugar de variables. Cuando el incremento no sea unitario, deberá indicarse explícitamente mediante. la variable correspondiente a un contador, por ejemplo: *v = v + incremento*.

NOTA:

Es importante remarcar que debe haber congruencia entre el valor inicial, el final y el incremento para que se obtenga la variación deseada, es decir, si se desea variar de 30 a -2 de uno en uno, el incremento deberá ser negativo, etc.

Considere como ejemplo la tabulación de la función trigonométrica *SIN(X)* en el intervalo de 0 a 1.6 radianes:

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

/* PROGRAMA PARA TABULAR LA FUNCIÓN  y = SIN(x) */

main()
{
   float a = 0;
   float b = 1.6;
   float c = 0.1;
   float x, y;

   for( x = a; x <= b; x = x + c)
   {
       y = sin(x);
       printf("%f    %f", x, y);

   }
   getch();
}
```

Considere el mismo programa que indica cantidades y no variables en el ciclo:

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

/* PROGRAMA PARA TABULAR LA FUNCIÓN  y = SIN(x) */

main()
{
   float x, y;

   for( x = 0.0; x <= 1.6; x = x + 0.1)
   {
       y = sin(x);
       printf("%f    %f", x, y);

   }
   getche();
}
```


Otra aplicación del ciclo iterativo es para producir una pausa que permite ver resultados que se calculan mediante procesos iterativos como el anterior. Al ser tan rápida la computadora para los cálculos, todos los resultados son desplegados a una velocidad que no permite verlos con facilidad, debido a ello se emplean ciclos iterativos de N iteraciones entre cada desplegado. Considere el mismo programa del caso anterior incluyendo una pausa:

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

/* PROGRAMA PARA TABULAR LA FUNCIÓN  y = SIN(x) */

main()
{
   float x, y;
   int z;

   for( x = 0.0; x <= 1.6; x = x + 0.1)
   {
       y = sin(x);
       printf("%f    %f", x, y);

       for( z = 0; z <= 10000; z++);

   }

   /* Al final es conveniente detener el proceso con
      la función getche();  */

   getche();
}
```

Observe que el ciclo de la variable z varia de 1 a 10000 y que dentro de su ámbito no hay instrucción alguna. La pausa la produce el tiempo que le lleva el contar de 1 a 10000; de esa manera es posible visualizar los resultados que, de no ser así, pasarían rápidamente ante nuestros ojos, prácticamente sin la menor posibilidad de ver los primeros valores. Recuerde que día a día las computadoras son más rápidas.

OBSERVACIÓN: Cuando el ciclo no involucra más de una instrucción dentro de su ámbito, pueden suprimirse las llaves.

Substituya el for de la pausa con la función delay(valor_miliseg); en la que el argumento expresa un valor en milisegundos (se sugiere 300); la alternativa también es la función sleep(valor_segundos); en la que la pausa estará dada en segundos.

Considere ahora el uso del for para observar los valores que se obtienen con operadores abreviados típicos del lenguaje C:

```
/*  operadores reducidos  */

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

 main()
 {
  int i, k, l, m, n;
  int j1, j2, j3, j4;
  k = l = m = n =  1;
  clrscr();

  printf("Inicia la prueba a partir de k = l = m = n = 1\n ");

   for(i=1; i<=6; i+ = 1) /*     observe el incremento        */
   {
   j1 = ++k;
   j2 = l++;
   j3 = - - m;
   j4 = n - -;
   printf("\n\n Valores en la aplicación No. %d ,  + + k - %d   l + + =  %d   - - m = %d  n - - = %d",i, j1,j2,j3,j4);

   getch();

   }
   printf("\n\n\n Recuerde que: \t\t  + + v incrementa y luego asigna\n\t\t\t  v + +  asigna y luego incrementa");
   printf("\n\n                \t\t  - - v decrementa y luego asigna\n\t\t\t  v - -  asigna y luego decrementa");
   getch();
}
```

Observe que en los resultados se comprueba que ++v primero incrementa y luego asigna, mientras que v + + primero asigna y luego incrementa; de forma similar, - - v primero disminuye y luego asigna y v - - primero asigna y luego disminuye.

Como otro ejemplo de uso del ciclo for se calcula un polinomio de grado "n", mediante la función:

$$P(X) = a_0 X^n + a_1 X^{n-1} + a_2 X^{n-2} + \ldots + a_{n-1} X + a_n$$

y se tabula en el intervalo de -10 a 10, con un incremento unitario, desplegando los valores tabulados.

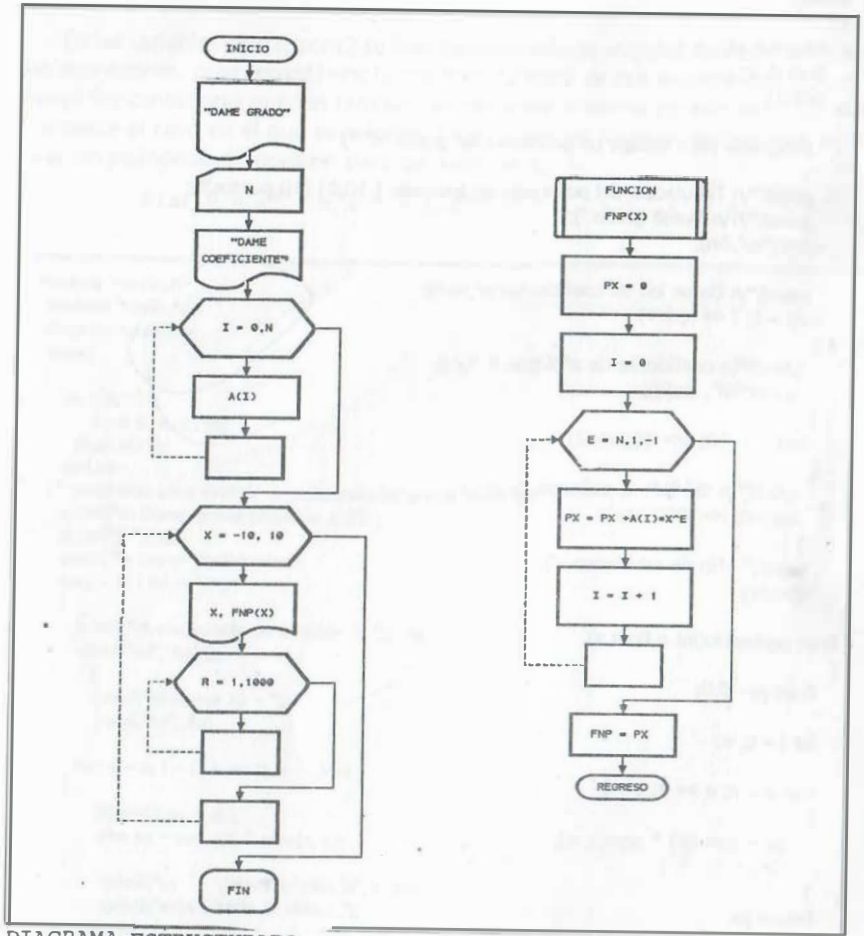


DIAGRAMA ESTRUCTURADO

A continuación se muestra una codificación en lenguaje C, en la cual se han

agregado mensajes que ayudan a aclarar la naturaleza de los datos pedidos.

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float a[100];
float polinomio(int n, float x);
main()
{
  int i, n,r, r1;
  float d, x;
  clrscr();

 /* programa para tabular un polinomio de grado "n" */

   printf("\n Tabulacion del polinomio en intervalo {-10,0} (10 puntos)");
   printf("\n\n Dame grado ");
  scanf("%i",&n);

   printf("\n Dame los %i coeficientes\n",n+1);
  for(i = 0; i <= n; i++)
  {
    printf("\n coeficiente de x^%2d= ?  ",n-i);
    scanf("%f", &a[i]);
  }
   for( x = -10; x<= 10; x+=2)
  {
    printf("\n %f %f", x, polinomio(n,x));
    for(i=0; i<=1000; i++);
   }
   printf("   Fin de tabulacion  ");
   getch();
}
float polinomio(int n,float x)
{
  float px= 0.0;

  int i = 0, e;

  for( e = n; e >= 1; e - -)
  {
     px = px+ a[i] * pow(x, e);
     i = i + 1;
  }
  return px;
}
```


CICLO for CON DOBLE CONTADOR

La sintaxis agrega una coma como separador en los contadores como sigue:

for(cont1=val1,cont2=val2; condición; cont1=cont1+inc1,cont2=cont2+inc2)
{
instrucciones ejecutables;
}

En las variables cont1, cont2 se han definido valores iniciales de esos contadores; en las expresiones *cont1=cont1+inc1,cont2=cont2+inc2* se dan incrementos típicos, sin embargo los contadores pueden también decrementar o usarse combinados. Considere nuevamente el caso en el que se calculan los exponentes de la sumatoria que permite evaluar un polinomio de grado n para un valor de x_0:

$$P(x) = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \ldots + a_{n-1} x^1 + a_n$$

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
main()
{
  int i, n,r, r1;
      float d, e, x , px;
    float a[21];
  clrscr();
 /* programa para evaluar un polinomio de grado "n" en x0 */
  printf("\n Dame grado (maximo 20)");
  scanf("%i",&n);
  printf("\n Dame coeficientes");
  for(i = 0; i <= n; i++)
  {
    printf("\n coeficiente de x^%2d= ?  ",n - i);
    scanf("%f", &a[i]);
     }
      printf("\n Dame x0 = ");
      scanf("%f", &x);

    for( e = n, i = 0; e >= 0; e - - ,i++)
   {
       if(e==0) px = a[i];
       else px = px+ a[i] * pow(x, e);
   }
       printf("\n      \t%6.3f\t\t%6.3f", x, px);
       printf("\n\n   hasta la vista ...");
      getch();
}
```

9.11.1 CICLOS ITERATIVOS for ANIDADOS

Algunas aplicaciones requieren de procesos en los que se necesita que un contador varíe en un rango por cada valor de otro contador, como en el caso de los años, meses y días de un siglo (considere idealmente meses de 30 días):

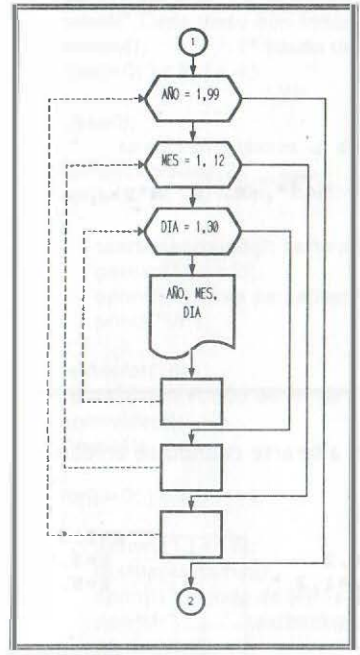
inicio del siglo:

año 1
  mes 1
    los días varían de 1 a 30
  mes 2
    los días varían de 1 a 30
  mes 3
    los días varían de 1 a 30
  .
  .
  .
  mes 12
    los días varían de 1 a 30
año 2
  mes 1
    los días varían de 1 a 30
  mes 2
    los días varían de 1 a 30
  mes 3
    los días varían de 1 a 30
  .
  .
  .
  mes 12
    los días varían de 1 a 30
  .
  .
  .
año 99
  mes 1
    los días varían de 1 a 30
  mes 2
    los días varían de 1 a 30
  mes 3
    los días varían de 1 a 30
  .
  .
  .
  mes 12

Observamos que los años varían de 1 a 99, mientras que los meses varían de 1 a 12 y por cada mes los días en promedio varían de 1 a 30.

Otro ejemplo de este tipo de variación de contadores es la variación del reloj (cada día cuenta 24 horas y en cada hora cuenta 60 minutos y cada minuto cuenta 60 seg.), y un último ejemplo es la variación de los factores que se generan de las tablas de multiplicar.

Ese tipo de variaciones (cíclicas) se obtienen cuando un ciclo for contiene a otro ciclo for y este a su vez a un tercero, y se dice que están anidados.



DIAGRAMA

```
for( anio = 1; anio <= 99; anio++)
{
  for( mes = 1; mes <= 12; mes++)
  {
    for( dia = 1; dia <= 24; dia++)
    {
      printf("%d%d%d", anio, mes, día)
    }
  }
}
```

CODIFICACIÓN

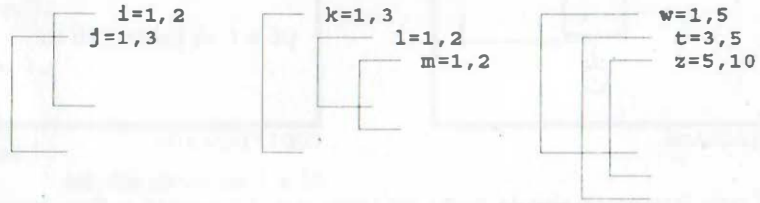
El ciclo interno se ejecuta todas las veces que determinan sus parámetros (*los días varían de 1 a 30*, por cada valor del ciclo intermedio que proporciona *los meses que van de 1 a 12* y éstos, a su vez, realizan su variación por cada valor del ciclo externo que da *los años que se generan de 1 a 99*. Cuando el incremento es unitario, es conveniente utilizar los operadores incremento que proporciona C.

También como aplicación inmediata se encuentra su uso en el manejo de arreglos

de 2 o más dimensiones o los procesos que involucran varios tipos de variaciones en algunas variables. Por ejemplo considere el procedimiento para obtener las tablas de multiplicar del 1 al 10 (para cada tabla el multiplicando genera el dígito que define a la tabla, por ejemplo la del 6, etc., mientras que el multiplicador varia de 1 a 10):

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

/* PROGRAMA PARA OBTENER LAS TABLAS DE MULTIPLICAR */

main()
{
    float x, y;

    for( x = 1; x <= 10; x++)
    {
        for( y = 1; y <= 10; y++)
        {
            printf("%3d por %3d = %4d", x, y, x*y);

        }

    }

    return 0;
}
```

Debido a que la sintaxis de cada ciclo no puede alterarse cuando se anidan, no es válido ningún tipo de traslape como los siguientes:



Tampoco es válido el intentar reinventar el ciclo for(), colocándole una pregunta al final, con la supuesta finalidad de que de esa manera regrese a completar el ciclo (error de concepto).

A continuación se muestran programas con ciclos for(); para despliegue de texto en color:


```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <stdlib.h>
main()                  /* textos parpadeantes */
{
 int i,j;
 clrscr();
 gotoxy(18,12);
 textcolor(4);
 printf(" Se muestra texto parpadeante sobre diversos fondos");
 gotoxy(18,14);
 printf(" Cada texto con fondo se muestra solo cuatro segundos...");
 sleep(4);          /* pausa de 4 segundos */
 for(j=0; j<8; j++)
 {
 clrscr();
 /*    se escriben textos en diferentes colores y diferentes fondos  */
 textbackground(7);
 for(i=0; i<16; i++)
 {
    textbackground(j); textcolor(i + BLINK);
    gotoxy(18,i+2);
    cprintf(" Letras parpadeantes en color con texcolor( %2d ) ",i);
    printf("\n");
 }
 textcolor(14);
 cprintf("\n\n Fondo de letras especificado con textbackground( %d ) ",j);
 normvideo();
 sleep(4);
 }
 for(j=0; j<7; j++)
 {
    gotoxy(1,j+18);
    textbackground(j);
    cprintf("  Fondo de letras especificado con .....");
    cprintf("..... textbackground( %d )      ",j);
    normvideo();
 }
 gotoxy(1,j+18);
 textbackground(7);
 textcolor(14);
 cprintf("  Fondo de letras especificado con ..... ");
 cprintf("..... textbackground( %d )      ",j);
 normvideo();
 getche();
 /* getche es una función que espera hasta que se presione
 una tecla */
}
```

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

main()          /*  ventanas aleatorias    */
{
  int x1,y1,x2,y2,i,j;
  randomize();
  window(1,1,80,25);     /* ventana para toda la pantalla */
  textbackground(1);
  clrscr();       /* el borrar la ventana activa el color de fondo */
  gotoxy(5,21);
  textcolor(14);
  cprintf(" Se muestran 5 ventanas. La posición y los colores son aleatorios");
  gotoxy(5,22);
  cprintf(" El texto se ajusta a la ventana y se muestra solo unos segundos...");
  sleep(4);            /* pausa de 4 segundos */
  for(i=1; i<6; i++)
  {
      x1 = random(13) + 1;
      y1 = random(17) + 1;
      x2 = x1 + 33;
      y2 = y1 + 4;
      window(x1,y1,x2,y2);
      textbackground(random(8));
      textcolor(random(16));
      clrscr();
      gotoxy(x1,y1);
      cprintf("                              ");
      gotoxy(x1,y1+1);
      cprintf(" Ventana num. %d posicion al azar ",i);
      gotoxy(x1,y1+2);
      cprintf("\n  Adivine posicion de la siguiente");
      cprintf("                              ");
      sleep(5);
      normvideo();
      clrscr();
  }
  lowvideo();
  textcolor(14 + BLINK);
  cprintf("\n Para salir oprima cualquier tecla ");
  normvideo();
  highvideo();
  textcolor(14 + BLINK);
  cprintf("\nPara salir oprima cualquier tecla");
  /* getch es una función que espera hasta que se presione
  una tecla */
  getch();
}
```


9.12 CICLOS ITERATIVOS CONTROLADOS POR CONDICIÓN

9.12.1 CICLO CON CONDICIÓN AL INICIO DEL CICLO: ELEMENTO ESTRUCTURADO while

Este elemento se emplea cuando se requiere repetir un proceso mientras que se cumpla una condición, su diagrama y sintaxis general se muestran a continuación:

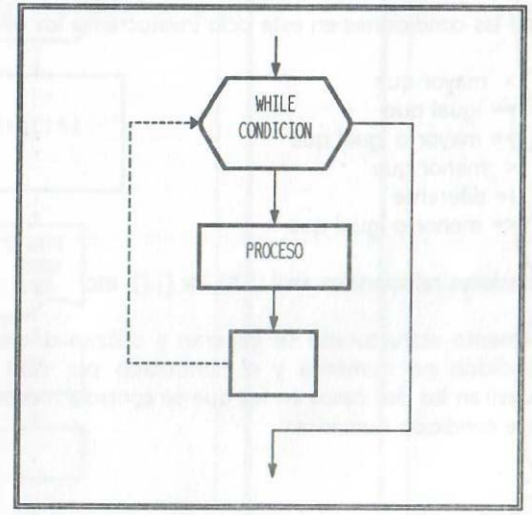


DIAGRAMA ESTRUCTURADO

SINTAXIS:

```
while(condición)
{
   instrucciones
   ejecutables   ;

}
```

CODIFICACIÓN

La figura anterior indica que el proceso enmarcado en el rectángulo, se ejecutará una y otra vez mientras se cumpla la condición indicada al principio del ciclo iterativo y que el ámbito del ciclo está delimitado por las llaves que engloban las instrucciones ejecutables.

En el caso anterior el proceso se repetirá N veces, siempre y cuando se cumpla la condición que se indica a un lado de la primera instrucción o while. Cuando la condición no se cumple no se ejecuta el proceso y el control de ejecución pasa a la línea que se indica inmediatamente después de la llave que cierra el ámbito del ciclo.

Por lo general las condiciones en este ciclo involucran a los operadores:

> mayor que
== igual que
>= mayor o igual que
< menor que
!= diferente
<= menor o igual que

además de los operadores relacionales and (&&), or (||), etc.

Con este elemento estructurado se generan y utilizan dos tipos de ciclos: el controlado por condición no numérica y el controlado por contador explícito. A continuación se muestran los dos casos en los que se controla mediante condición no numérica y mediante condición numérica:

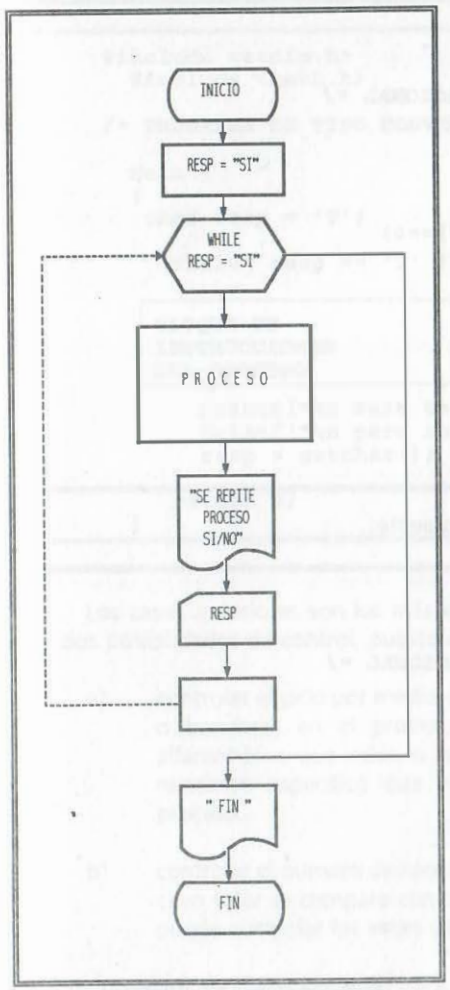


DIAGRAMA ESTRUCTURADO

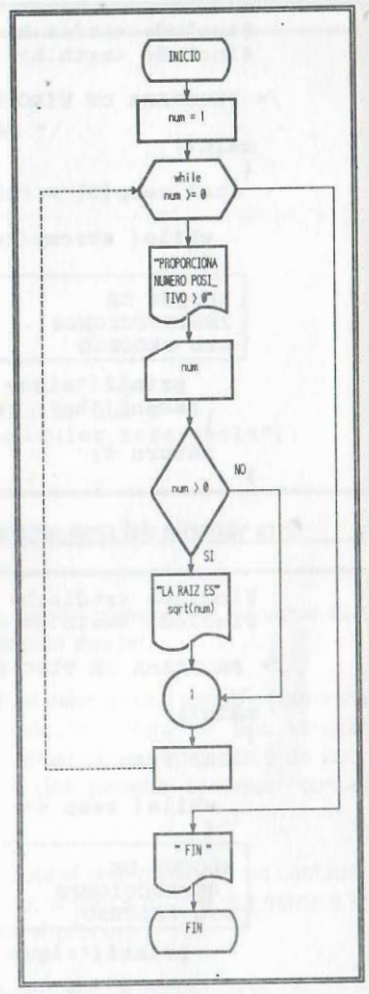


DIAGRAMA ESTRUCTURADO

En ambos diagramas se indica un ciclo iterativo que permite repetir un proceso enmarcado en un rectángulo. Dicho ciclo se repetirá mientras se cumpla la condición indicada al principio del ciclo y su ámbito está delimitado por la instrucción dada en el rectángulo que indica con línea punteada la secuencia de ejecución (llave que cierra o fin del while). Por ejemplo considere la siguientes codificaciones:

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

/* PROGRAMA DE TIPO CONVERSACIONAL */

main()
{
 char resp[3] = "si";

   while( strcmp(resp,"si")==0)
   {
   BLOQUE DE
   INSTRUCCIONES          ;
   DEL PROCESO

     printf("sigue si/no");
     scanf("%s", resp);
   }
   return 0;
}
```

Otra variante del caso anterior es la siguiente:

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

/* PROGRAMA DE TIPO CONVERSACIONAL */

main()
{
 char resp = 's';

   while( resp == 's')
   {
   BLOQUE DE
   INSTRUCCIONES          ;
   DEL PROCESO

    printf("sigue si/no");

/*  la función siguiente borra la memoria de teclado
    evitando errores de lectura */

     fflush(stdin);

     scanf("%c", &resp);
   }
   return 0;
}
```


Una variante del caso anterior es:

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

/* PROGRAMA DE TIPO CONVERSACIONAL */

main()
{
 char resp = 'T';

   while( resp == 'T' )
   {
   BLOQUE DE
   INSTRUCCIONES          ;
   DEL PROCESO

     printf("\n Para terminar teclea una T ");
     printf("\n para repetir cualquier otra tecla");
     resp = getchar();
   }
   return 0;
}
```

Los casos anteriores son los más empleados, siendo conveniente remarcar que hay dos posibilidades de control, puesto que la condición puede:

a) controlar el ciclo por medio del cambio del valor de una variable (centinela o bandera) en el proceso. Por ejemplo la lectura de una variable alfanumérica que indica si se desea continuar con la repetición o un valor numérico específico que cuando se teclea permite continuar con el proceso.

b) controlar el número de veces que se ejecuta el ciclo mediante un contador cuyo valor se compara con un valor dado: while i < n . De esa manera se puede controlar las veces que se realiza el proceso.

Considere ahora las opciones para control del número de veces que se realiza el ciclo, cunado se emplea condición de tipo numérico:

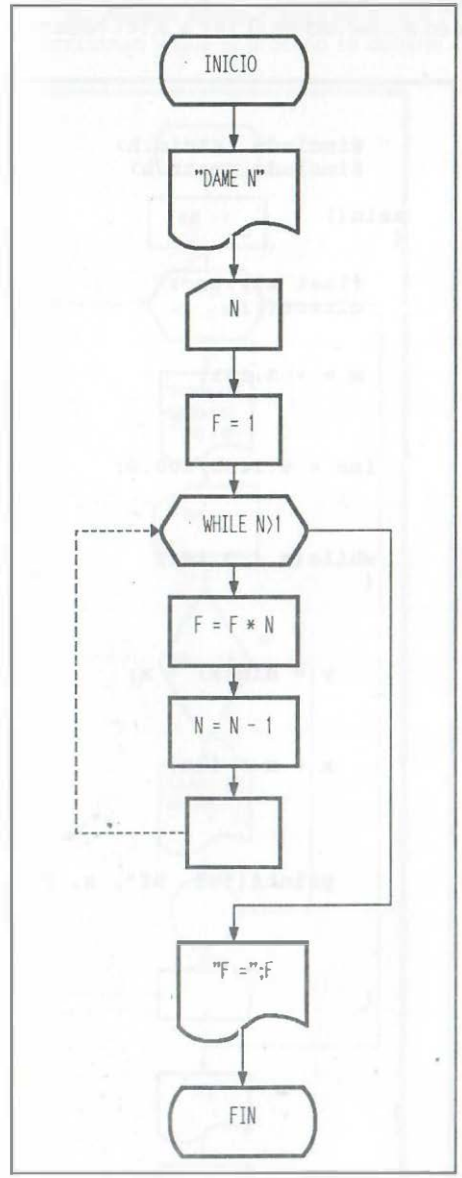
```
#include <stdio.h>
#include <math.h>

/* PROGRAMA DE TIPO CONVERSACIONAL */

main()
{
 int resp = 1;

   while( resp == 1 )
   {
   BLOQUE DE
   INSTRUCCIONES
   DEL PROCESO                ;

      printf("\n Para continuar teclea un número 1 ");

      scanf("%i", &resp);
   }
   return 0;
}
```

Considere la siguiente variante del proceso anterior:

```
#include <conio.h>
#include <stdio.h>
main()
{
   while(a > b)
   {
      PROCESO                 ;
   }
   INSTRUCCIÓN EJECUTABLE;
            .
            .
            .
   }
```

Ejemplo de aplicación:

Considere la codificación del programa para calcular el factorial descendente de un número positivo no nulo.



D I A G R A M A

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

main()
{
   int n;
   float f;
   clrscr();
   printf("\nDame el valor n");

   scanf("%d",&n);

   f = 1;

   while(n > 1)
   {
      f = f * n;

      n = n - 1;

   }

   printf("\nFactorial = %f",f);

}
```

C O D I F I C A C I Ó N

Como otro ejemplo, se muestra el diagrama de flujo y codificación del programa que permite tabular la función: y = sin(x) + x en el intervalo de -3.141 a 3.141 radianes.

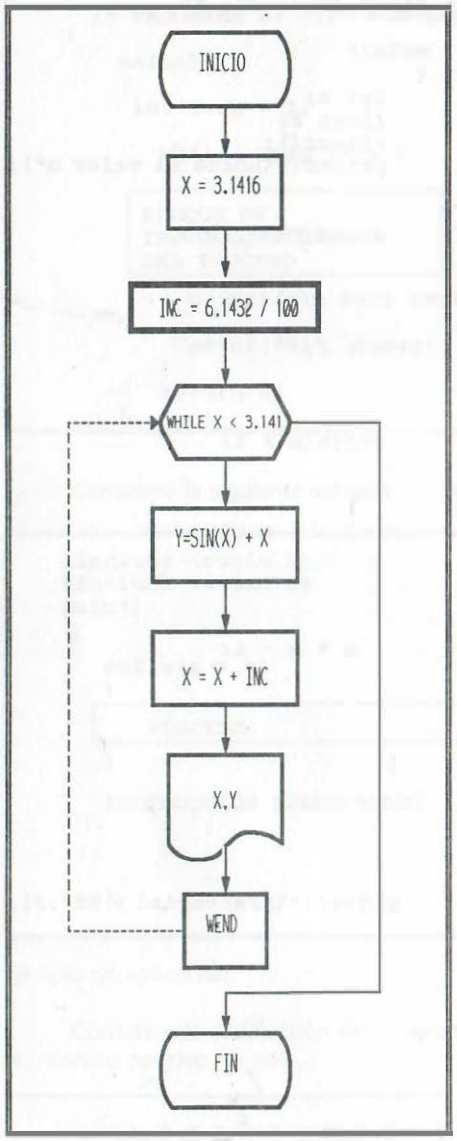


DIAGRAMA ESTRUCTURADO

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

main()
{
   float x,y,inc;
   clrscr();

   x = - 3.141;

   inc = 6.1432/100.0;

   while(x < 3.141)
   {

      y = sin(x) + x;

      x = x + inc;

      printf("%f  %f", x, y);

   }

}
```

CODIFICACIÓN

Considere ahora el caso en el que se calcula la raíz de números positivos que se proporcionan y que el proceso se detiene cuando se alimenta un número negativo:

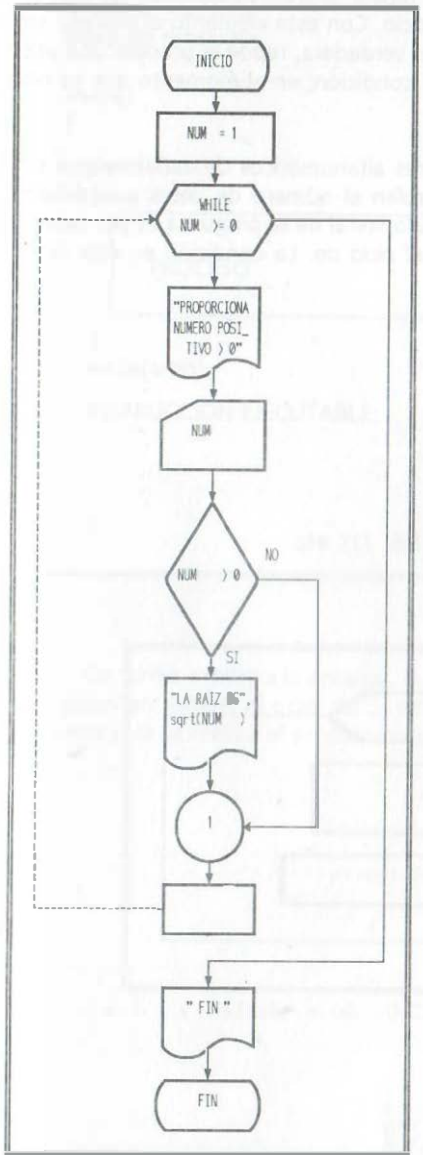


DIAGRAMA ESTRUCTURADO

```
#include <stdio.h>
#include <math.h>
main()
{
   float num;
   clrscr();
num = 1;

while(num > = 0)
{

   printf("Da numero > 0");


   scanf("%f", &num);


   if(num > 0)
    printf("raiz = %f", sqrt(num));









}



printf("fin");


}
```

CODIFICACIÓN

9.12.2 CICLO do CON OPCIÓN while

Otro de los elementos estructurados para repetir procesos es el ciclo do con la opción while(condición);, al final del ámbito del ciclo. Con este elemento el proceso se ejecuta al menos una vez. Cuando la condición es verdadera, repite el proceso una vez más y así sucesivamente mientras no se altere la condición; en el momento que ya no se cumple, se abandona el ciclo.

La condición puede involucrar tanto valores alfanuméricos de variables que se lean en el proceso, como contadores que controlen el número de veces que deban repetirse el conjunto de instrucciones. El significado literal de la pregunta es por tanto: *si no se cumple la condición, entonces finaliza el ciclo do*. La condición en este ciclo puede involucrar a los operadores:

> mayor que
== igual que
>= mayor o igual que
< menor que
!= diferente
<= menor o igual que

además de los operadores relacionales and, or (&&, ||), etc.

El diagrama de este ciclo es el siguiente:

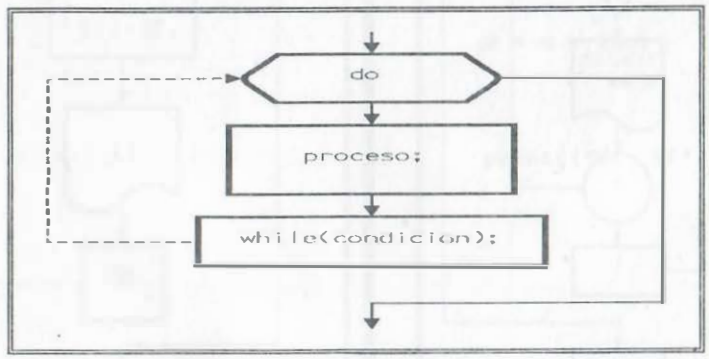


DIAGRAMA ESTRUCTURADO DEL CICLO do - while();

Considere como ejemplo de sintaxis:

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

main()
{
  do
  {
     PROCESO ;
  }
  while(a >b);
  INSTRUCCIÓN EJECUTABLE;
        .
        .
        .
  }
```

De forma similar a la anterior, la condición por lo general involucra dos variables que permiten romper el ciclo: do ... while(a > b) . De esa manera se puede controlar las veces que se realiza el proceso que, por lo menos lo realiza una vez.

A continuación se muestra el diagrama del factorial descendente, resuelto con do while:

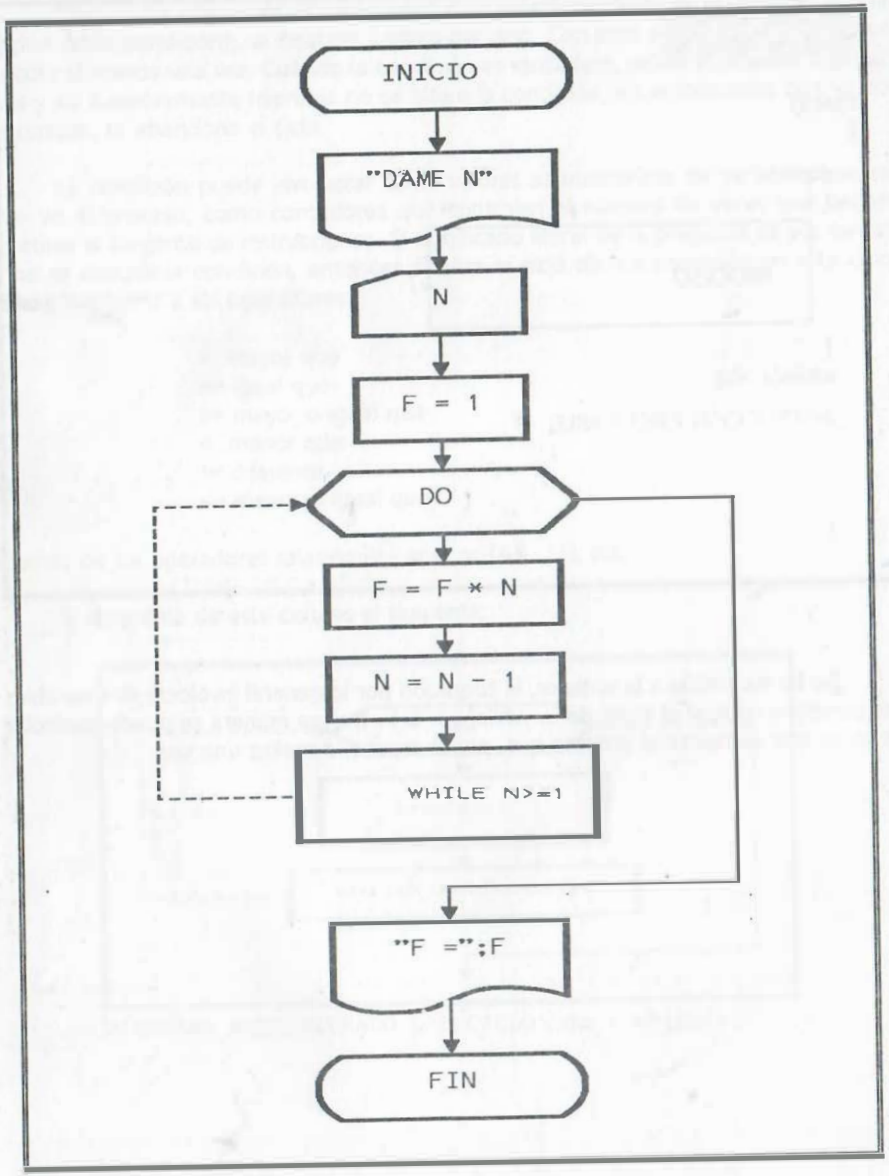


DIAGRAMA ESTRUCTURADO

La codificación del ejemplo anterior se muestra a continuación:

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

main()
{
   int n;
   float f;
   clrscr();
   printf("\nDame el valor n");


   scanf("%d",&n);

   f = 1;

   do
   {

     f = f * n;

     n = n - 1;


   }
  while(n > 1);

  printf("\nFactorial = %f",f);


}
```

Este tipo de ciclo tiene la característica de que al menos se ejecuta una vez

Como último ejemplo de aplicación, considere el siguiente ejemplo en el cual se obtiene la resultante de un sistema de fuerzas en el plano a partir de la suma de proyecciones sobre los ejes coordenados en el plano xy.

En este ejemplo se hace uso de variables estáticas para acumular las proyecciones en x y las proyecciones en y ; de esa manera, cada vez que se llama a la función que las declara, los valores que se asignan se conservan hasta la siguiente llamada y se puede continuar acumulando pues los últimos valores no se destruyen.

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

float m, ang, sx, sy;        /* Variables globales o externas */

void proyeacu( void );       /* Declaracion de la funcion que proyecta */

main()
{
   int i, n, r1;
   char r, resp;
   float d, x, rm;
   clrscr();
   do  /*..... inicia la acumulación de componentes */
   {
   printf("\n\n dame la magnitud y ángulo de la fuerza en grados ");
        scanf("%f,%f", &m, &ang);
        proyeacu();

   /* al llamar a la funcion el control de proceso se transfiere
        a ella para, al final de su subproceso devolver el control
        y continuar con el printf(); que sigue:
   */
        printf("\n\n\ otra fuerza del sistema s/n ");
        r = getch();
    }
   while (r == 's');   /*.....termina proceso de acumulación */

   rm = sqrt(pow(sx,2) + pow(sy,2));
   printf("\n\n La magnitud de la  resultante del sistema es %f ", rm);
   getch();
}

   void proyeacu()    /* Definicion de la funcion que proyecta  */
   {
        static float sfx=0, sfy=0;  /*    Variables estaticas    */
        sfx = sfx + m * cos(ang*3.1415926/180.0);
        sfy = sfy + m * sin(ang*3.1415926/180.0);
        printf("\n Las proyecciones en los ejes son: %f %f ",sx=sfx,sy=sfy);
   }
```


9.12.2.1 CICLO INFINITO: ELEMENTO ESTRUCTURADO do - while(1)

Este es un caso particular del ciclo anterior que, teóricamente puede repetir un proceso de manera infinita. Su ámbito de acción inicia con la instrucción do que significa *hacer*, seguida de un conjunto de instrucciones ejecutables que se repetirán indefinidamente, dicho conjunto está delimitado por la llave que cierra el ámbito del ciclo. La sintaxis de este ciclo se muestra a continuación:

```
do
{
instrucciones ejecutables ;
}
while(1);
```

sin embargo, no es adecuado generar ciclos infinitos y por lo general se le usa acompañada de la pregunta:

if(*condición*) break;

la cual normalmente forma parte del PROCESO y de esa forma, cuando se cumple la condición indicada en ella, se rompe el ciclo y a continuación se transfiere el control de ejecución a la instrucción que sigue al while(1).

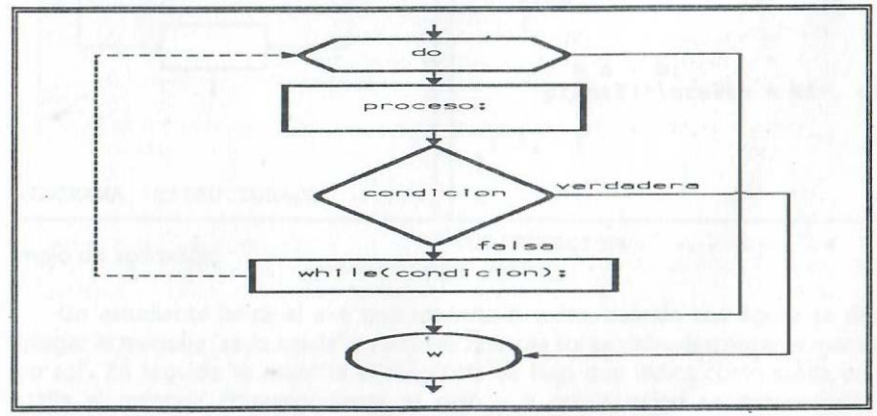


DIAGRAMA ESTRUCTURADO

Sintaxis:

```
do
{
  instrucciones ejecutables;
  if( condición ) break;
  instrucciones ejecutables;
}
while(1);
instrucción ejecutable;
        .
        .
        .
```


9.13 ELEMENTO ESTRUCTURADO if else COMPUESTO

(RAMIFICACIONES O TRANSFERENCIAS)

Cuando en un algoritmo es necesario realizar una pregunta para tomar decisión sobre procesar uno de dos bloques de instrucciones, el elemento estructurado que le corresponde es if else.

En dicho elemento si el resultado de la condición es verdadera, entonces ejecuta el conjunto de instrucciones específicas que corresponden al bloque_1, delimitadas por llaves, caso contrario (else) ejecuta el bloque_2 de instrucciones, según se muestra:

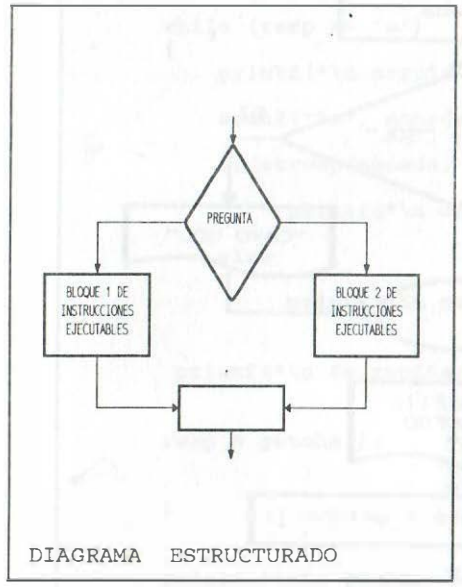


DIAGRAMA ESTRUCTURADO

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

main()
{
  float a, c, b;
  clrscr();

   if ( condición )
   {

     c = a + b;
     printf("\n suma = %f", c);

   }
   else
   {

     c = a - b;
     printf("\nresta = %f", c);

   }
}
```

CODIFICACIÓN

Ejemplo de aplicación:

Un estudiante lanza al aire una moneda n veces, cuando cae águila se debe desplegar el mensaje "cayo aguila" y cada vez que cae sol se debe desplegar el mensaje "cayo sol". En seguida se muestra el diagrama de flujo que indica como salida en la pantalla el mensaje correspondiente al caso y a continuación se proporciona la codificación.

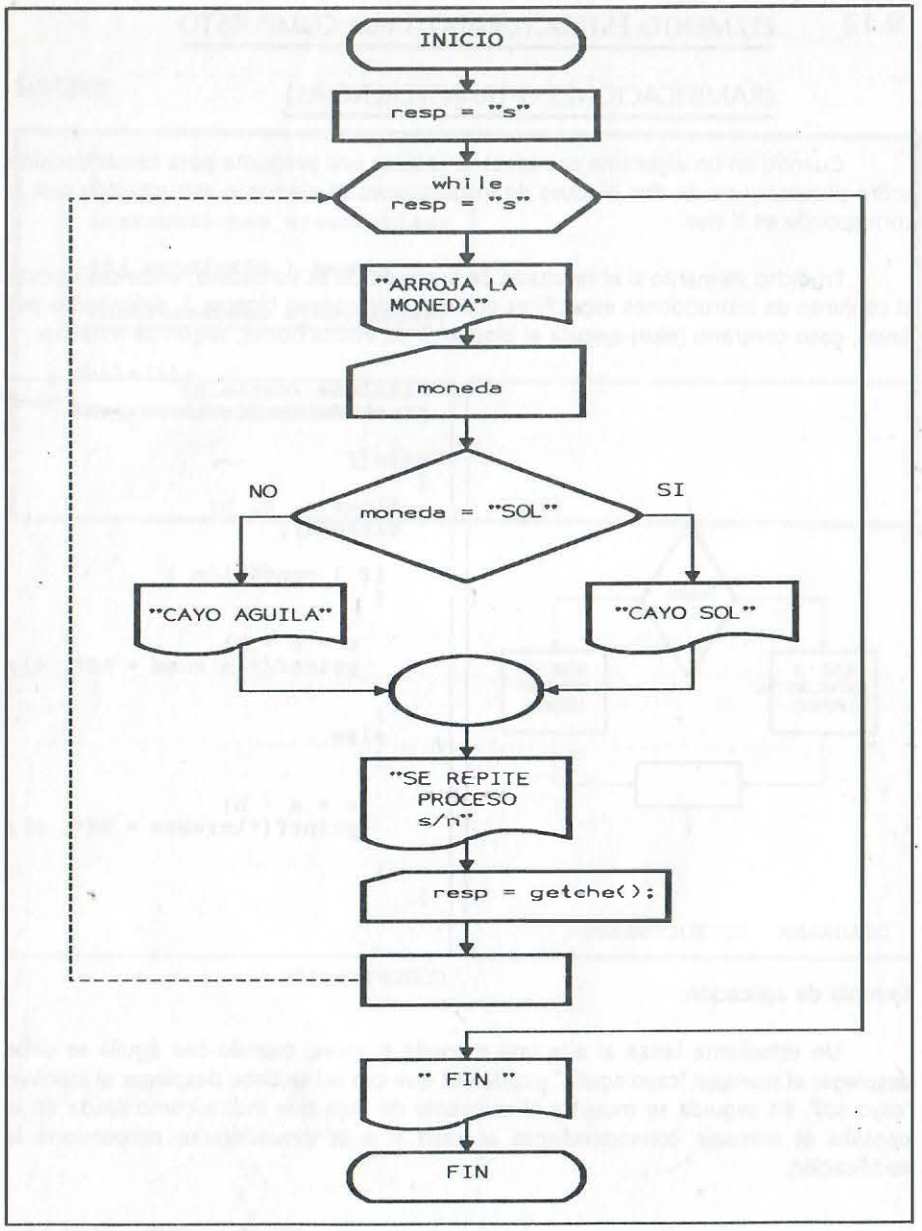


DIAGRAMA ESTRUCTURADO

La codificación del ejemplo anterior se muestra a continuación:

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

main()
{
    char resp, moneda;
    clrscr();

    resp = 's'

    while (resp == 's')
    {
        printf("\n arroja la moneda");

        scanf("%s", moneda);

        if(strcmp(moneda, "sol") == 0 )

            printf("\n cayo sol");

        else

            printf("\n cayo águila");

     printf("\n Se repite el proceso s/n");

    resp = getche();

    }

    printf("\n\n fin");

}
```

Observe que la lectura de un caracter se realiza con la función getche();, mientras que la comparación de la cadena "sol", se realiza por medio de la función strcmp(,); cuyo resultado es de tipo lógico es decir un cero para falso y un uno para verdadero.

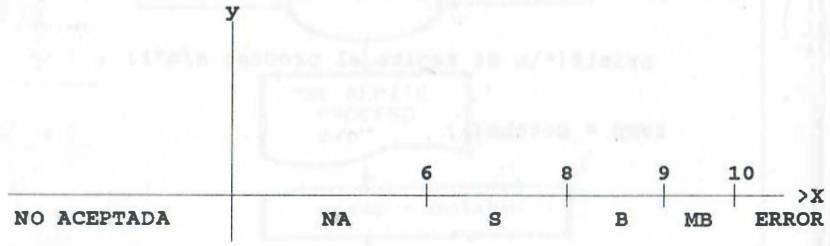
9.14 ELEMENTO ESTRUCTURADO DE RAMIFICACIÓN ENCADENADO

O ANIDADO if- else if - else if..... else

Cuando se presentan preguntas en cascada sobre una expresión, se pueden establecer categorías, intervalos o clases (de mayor a menor o viceversa, a manera de un filtro que separa en distintas categorías los valores por la posición que ocupan en un rango de comparación de una expresión dada: precios, intervalos, clases, etc.). La primera pregunta captura los datos de la primera categoría, la segunda establece el segundo intervalo con valores acotados por la segunda condición establecida y así sucesivamente. Por esta razón se dice que se tiene una estructura de preguntas encadenadas o de selección de clases. Veamos en diagramación tradicional el siguiente ejemplo:

Considere el problema de asignar la calificación del examen de un alumno, en escala de 0 a 10 puntos, bajo el siguiente criterio:

```
No son válidas calificaciones negativas
NA si la calificación es menor de 6
S si se obtiene de seis a menos de ocho
B si la calificación es menor de 9 y mayor o igual a 8
MB si obtiene de 9 a 10
Las calificaciones mayores a 10 se consideran como errores
```



Obviamente si se desea filtrar las calificaciones, realizando preguntas en cadena, deberá iniciarse por uno de los extremos del rango de calificaciones, es decir de menor a mayor o viceversa. Preguntar en cadena puede ejemplificarse entonces diciendo que si la calificación no fue menor de 6, hay que ver si fue menor de 8 y, si no fue ninguna de esas dos, hay que ver si fue menor de 9, y cuando no fue ninguna de las anteriores, verificar si corresponde a 10, por último, en caso contrario ubicarla como error. Consideremos entonces el filtro a partir de los valores no aceptados:

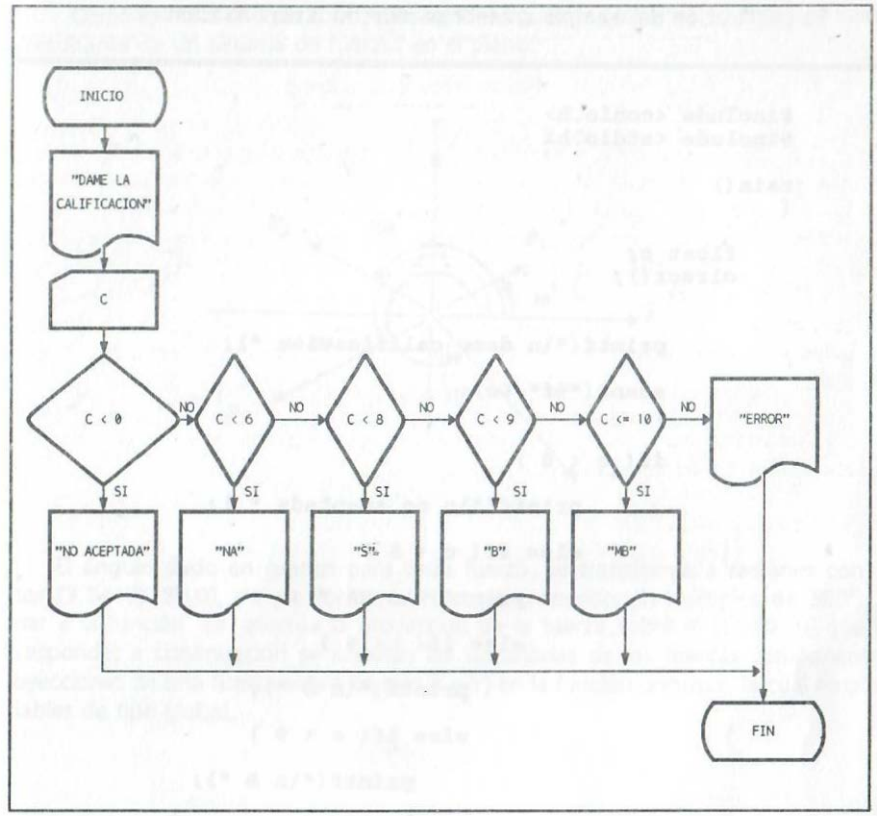
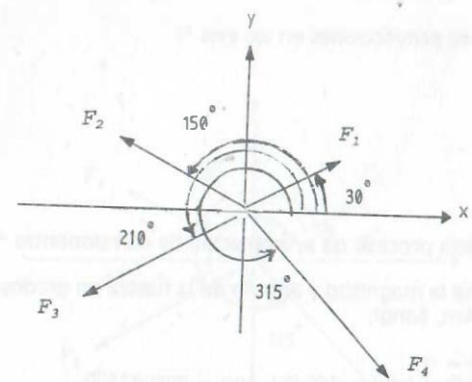


DIAGRAMA ESTRUCTURADO

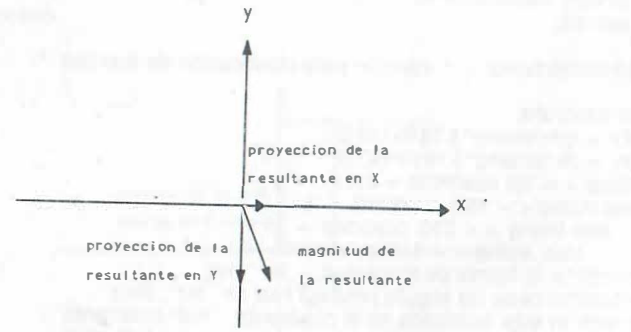
La codificación del ejemplo anterior se muestra a continuación:

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

main()
{
    float c;
    clrscr();

        printf("\n dame calificación ");

        scanf("%f",&c);

        if( c < 0 )

              printf("\n no aceptada " );

              else if( c < 6 )

                  printf("\n NA ");

                  else if( c< 8 )

                      printf("\n S ");

                      else if( c < 9 )

                          printf("\n B ");

                          else if( c <= 10 )

                              printf("\n MB ");
        else

              printf("\n error ");

        printf("\n fin ");

}
```


Como ejemplo de aplicación, considere el siguiente problema en el cual se calcula la resultante de un sistema de fuerzas en el plano:

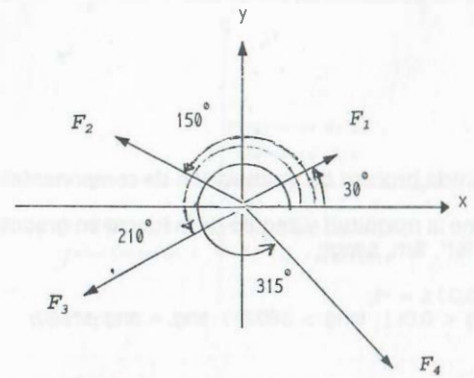


El ángulo dado en grados para cada fuerza, se transforma a radianes con el factor (3.1416/180.0), de esa forma, aunque se proporcionen múltiplos de 360^0, al llamar a la función se efectúa la proyección de la fuerza sobre el cuadrante que le corresponde; a continuación se calculan las sumatorias de las fuerzas componentes (proyecciones de una fuerza sobre los ejes X y Y) en la función principal, la cual emplea variables de tipo global.

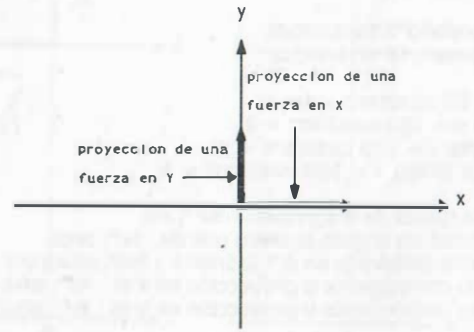


```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float m, pfx=0, pfy=0, sfx, sfy;
float ang;
 /* valores iniciales de proyecciones en los ejes */
void clasific(void);
main0
{
   int i, n, r1, s;
   float d, x, rm;
   char r;
   clrscr0;
   r = 's';
   do  /*'..... inicia proceso de acumulación de componentes */
   {
      printf("\n dame la magnitud y ángulo de la fuerza en grados ");
    scanf("%f,%f", &m, &ang);
     s = 1;
    if( ang < 0.0 ) s = -1;
    while( (ang < 0.0) || (ang > 360.0) )  ang = ang -s*360;

    clasific0;
  /* al llamar a la funcion el control de proceso se transfiere
    a ella para al final de su subproceso devolver el control
       y en este caso, continuar con el printf0; que sigue: */
    printf("\n\n Otra fuerza del sistema s/n ");
      r = getche0;
      sfx + = pfx;
      sfy + = pfy;
      } while (r = = 's');  /*'...fin del ambito del ciclo de acumulación */
      rm = sqrt(sfx*sfx + sfy*sfy);
      printf(" Resultante %f ",rm);
      getch0;
}
   void clasific(void)   /*  función para clasificación de fuerzas)  */
   {
    int cuadrant;
    pfx = m*cos(ang*3.1416/180.0);
    pfy = m*sin(ang*3.1416/180.0);
    if(ang < = 90) cuadrant = 1;
    else if(ang < = 180) cuadrant = 2;
        else if(ang <= 270) cuadrant = 3;
            else  if(ang < = 360) cuadrant = 4;
    printf("\n la fuerza de magnitud = %f ", m);
    printf("\n tiene un ángulo positivo real de : %f", ang);
    printf("\n esta localizada en el cuadrante  : %d",cuadrant);
    printf("\n su componente o proyección en x es : %f", pfx);
    printf("\n su componente o proyección en y es : %f", pfy);
    /*  la función regresa el control del proceso al  procedimiento
       principal al encontrar la llave siguiente:   */
   }
```


Como otro ejemplo de aplicación, considere el problema en el que se procesan fuerzas por separado para, a partir de la magnitud de cada fuerza en el plano y su ángulo en grados (positivo o negativo), determinar el cuadrante que le corresponde y sus fuerzas componentes en los ejes X, Y (proyecciones en los ejes de la magnitud de cada fuerza):

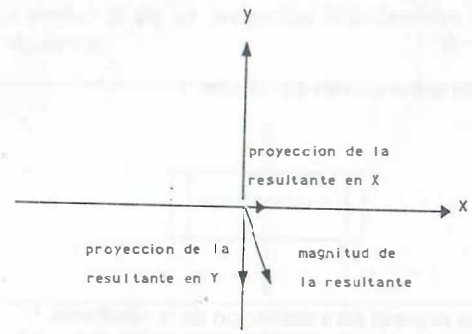


Dadas la magnitud y el ángulo para cada fuerza, se reduce el ángulo al intervalo ce cero a trescientos sesenta grados, agregando 360 cuando es negativo o restando 360 cuando es mayor de 360; a continuación se llama a una función que establece un filtro que determina el cuadrante que le corresponde y las magnitudes de las fuerzas componentes sobre los ejes (proyecciones de una fuerza sobre los ejes X y Y); al final se regresa el control a la función principal con el fin de continuar clasificando fuerzas o terminar el proceso.

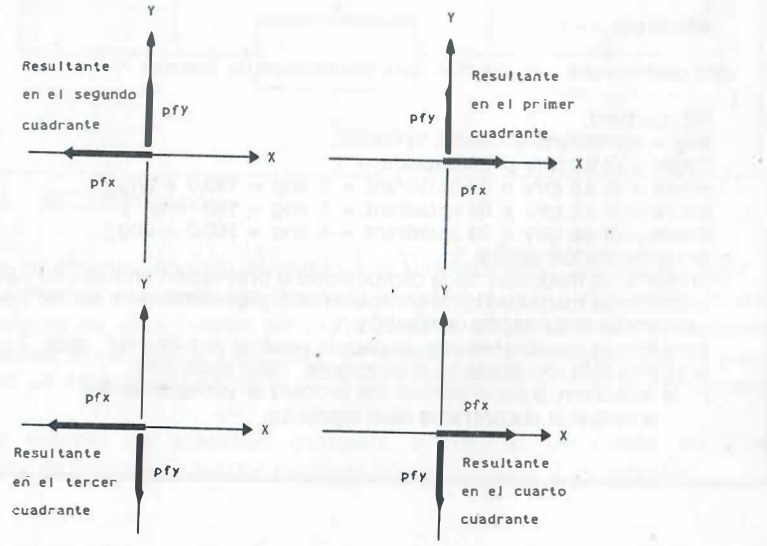


```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float m, pfx, pfy;
float ang;
 /* valores iniciales de proyecciones en los ejes */
void clasific(void);
main()
{
   int i, n, r, rn, s;
   float d, x, rm;
   clrscr();
   r = 's';
   do /*'..... inicia proceso de acumulación de componentes */
   {
     printf("\n dame la magnitud y ángulo de la fuerza en grados ");
        scanf("%f,%f", &m, &ang);
     s = 1;
        if( ang < 0.0 ) s = -1;
        while( (ang < 0.0) || (ang > 360.0) ) ang = ang -s*360;

        clasific();
   /* al llamar a la función el control de proceso se transfiere
      a ella y al final del subproceso se devuelve el control
     para, en este caso, continuar con el printf(); que sigue:
   */
        printf("\n\n Otra fuerza del sistema s/n ");
        r = getche();
      } while (r == 's');  /*'...fin del ambito del ciclo de acumulación */

}
   void clasific(void)   /*  función para clasificación de fuerzas)  */
   {
        int cuadrant;

        pfx = m*cos(ang*3.1416/180.0);
        pfy = m*sin(ang*3.1416/180.0);

        if(ang <= 90) cuadrant = 1;
        else if(ang <= 180) cuadrant = 2;
           else if(ang <= 270) cuadrant = 3;
                else  if(ang <= 360) cuadrant = 4;

        printf("\n la fuerza de magnitud = %f ", m);
        printf("\n tiene un ángulo positivo real de : %f", ang);
        printf("\n esta localizada en el cuadrante  : %d", cuadrant);
        printf("\n su componente o proyección en x es : %f", pfx);
        printf("\n su componente o proyección en y es : %f", pfy);

       /*  la función regresa el control del proceso al procedimiento
          principal al encontrar la llave siguiente:    */

   }
```


Como último ejemplo de aplicación, considere el problema de determinar la magitud de la resultante, su ángulo y cuadrante cuando son dadas las componentes de la resultante de un sistema de fuerzas en el plano.



En este algoritmo se consideran cada uno de los casos que permiten determinar el cuadrante y el ángulo que le corresponde a la resultante:



A continuación se muestra la codificación del problema anterior:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float m, pfx=0, pfy=0;
float ang;
 /* valores iniciales de proyecciones en los ejes */
void clasific(void);
main()
{
   int i, n, r1, s;
   float d, x, rm;
   char resp;
   do
   {
        clrscr();
       /*'..... se inicia proceso para obtencion de la resultante */
        printf("\n dame las fuerzas componentes en los ejes x, y ");
        scanf("%f,%f", &pfx, &pfy);
        clasific();
       /* al llamar a la funcion el control de proceso se transfiere
       a ella para realizar su cometido y al final devolver el control
       para, en este caso, continuar con el printf(); que sigue: */
        rm = sqrt(pfx*pfx + pfy*pfy);
        printf("\n La magnitud de la resultante es %f ",rm);
        printf("\n Otro calculo S/n ");
        resp = tolower(getche());
   }
   while(resp== 's');
}
void clasific(void)   /*  función para clasificación de fuerzas)  */
{
   int cuadrant;
   ang = atan(pfy/pfx)*180.0/3.14159265;
   if((pfx > 0) && (pfy > 0))cuadrant = 1;
   if((pfx < 0) && (pfy > 0)) {cuadrant = 2; ang = 180.0 + ang;}
   if((pfx < 0) && (pfy < 0)) {cuadrant = 3; ang = 180 - ang; }
   if((pfx > 0) && (pfy < 0)) {cuadrant = 4; ang = 360.0 + ang;}
   printf("\n DATOS LEIDOS: ");
   printf("\n La magnitud de la componente o proyección en x es : %f", pfx);
   printf("\n La magnitud de la componente o proyección en y es : %f", pfy);
   printf("\n\n RESULTADOS OBTENIDOS: ");
   printf("\n La resultante tiene un ángulo positivo real de : %f", ang);
   printf("\n Está localizada en el cuadrante  : %d",cuadrant);
   /*  la función regresa el control del proceso al  procedimiento
       principal al encontrar la llave siguiente:   */
}
```


9.15 RAMIFICACIÓN MÚLTIPLE O SELECCIÓN DE CASOS ESPECÍFICOS

Este elemento de la programación estructurada para selección específica de un caso ante opciones múltiples en lenguaje C se denomina switch; su funcionamiento puede considerarse similar al de las preguntas encadenadas. En efecto considere el diagrama de flujo siguiente:

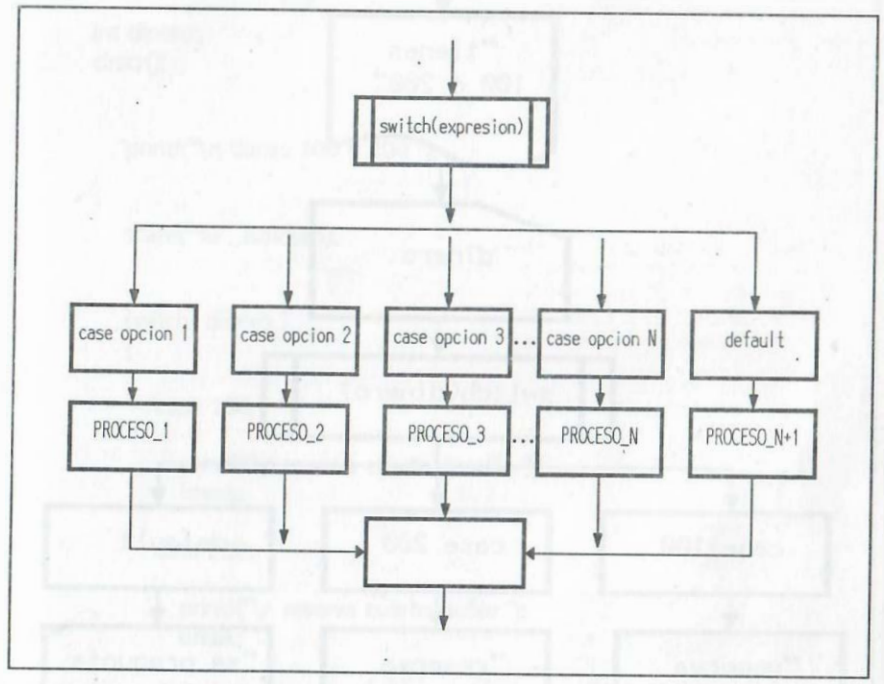
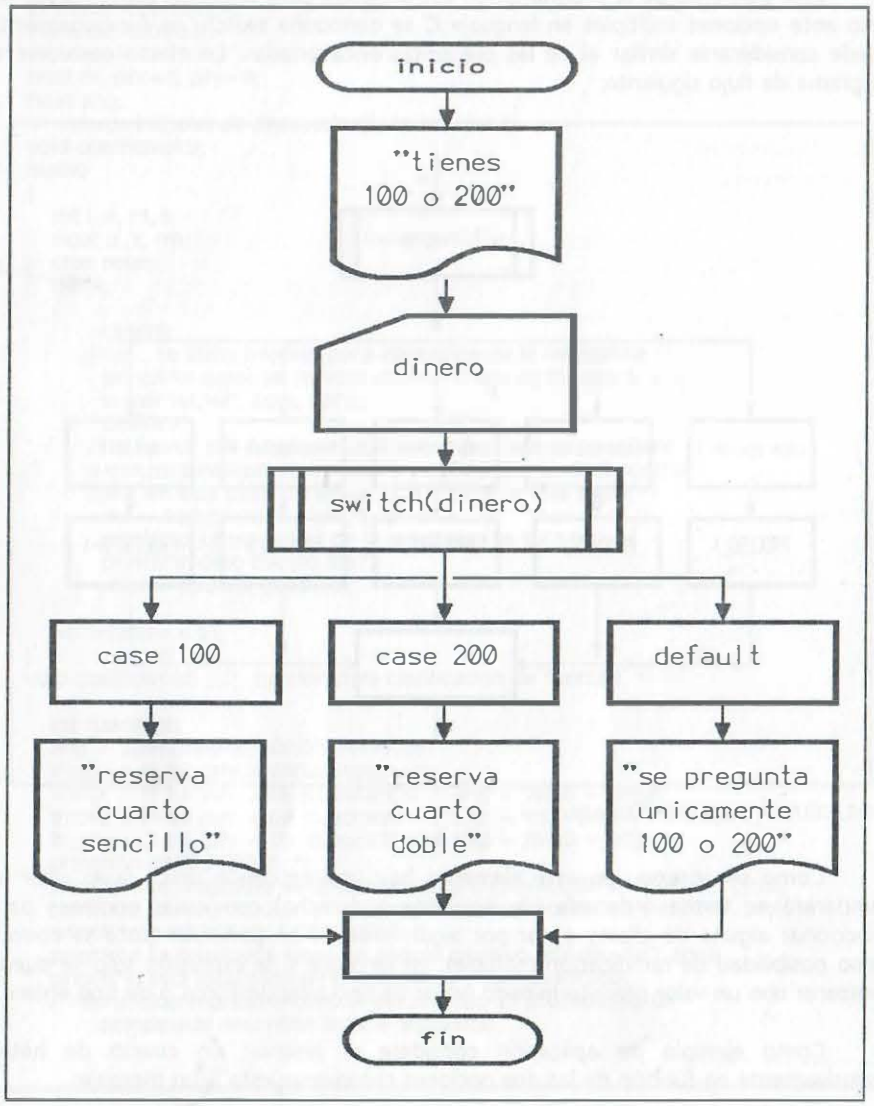


DIAGRAMA ESTRUCTURADO

Como se observa, en este elemento hay una expresión única cuyo valor se comparará en forma ordenada (de izquierda a derecha) con varias opciones para seleccionar alguna de ellas y optar por algún proceso en particular (esto se conoce como posibilidad de ramificación múltiple); en lenguaje C la expresión sólo se puede comparar con un valor predeterminado (valor de tipo alfanuméricos o de tipo entero):

Como ejemplo de aplicación considere el reservar un cuarto de hotel, exclusivamente en función de las dos opciones como respuesta a un mensaje:

En el diagrama siguiente la comparación se realiza de izquiera a derecha hasta encontrar la opción que le corresponde.



Y la codificación de ese algoritmo es:


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

main()
{

   int dinero;
   clrscr();

      printf("\n tienes 100 o 200 ");

      scanf("%i", &dinero);

      switch( dinero )
      {

         case 100:

            printf("\n reserva cuarto sencillo ");
            break;

         case 200:

            printf("\n reserva cuarto doble ");
            break;

        .default:

            printf("\n se pregunta unicamente 100 o 200");
            break;

      }

}
```

Considere como otro ejemplo la codificación del proceso que resuelve la ecuación de segundo grado utilizando el elemento switch, según se muestra a continuación:

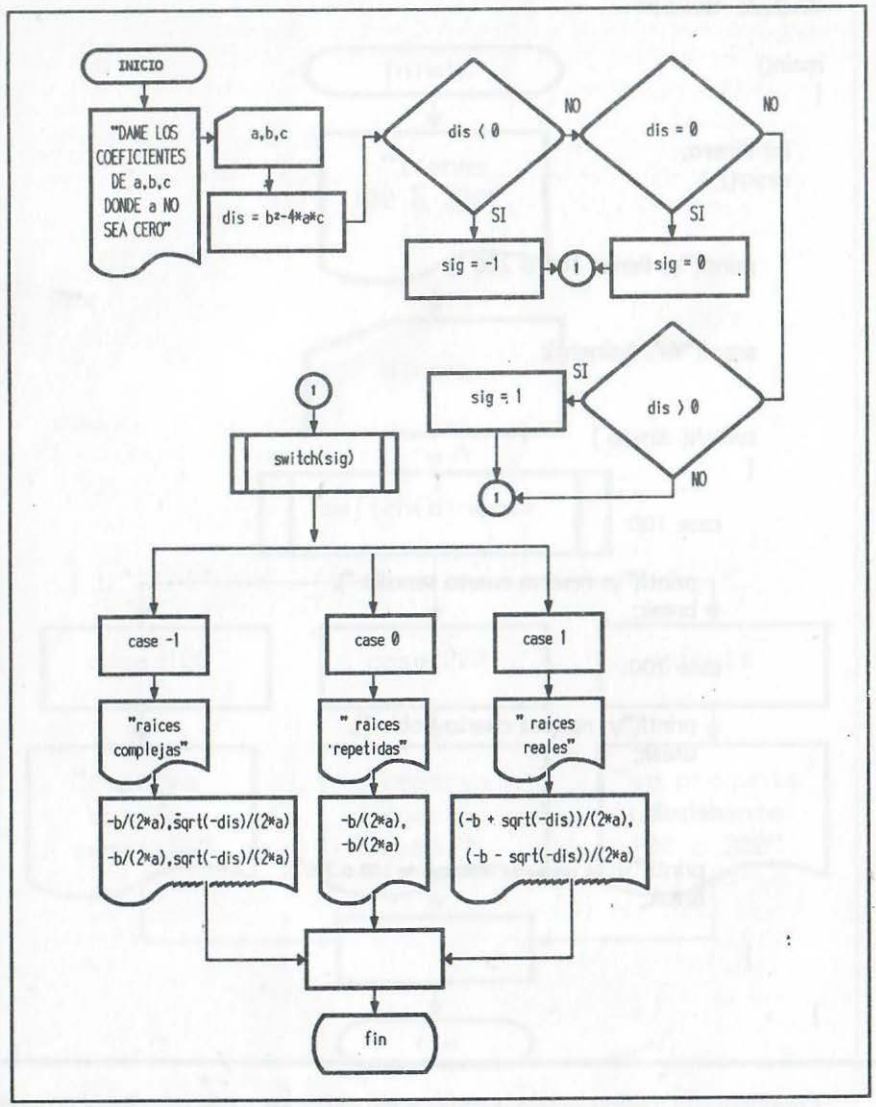


DIAGRAMA ESTRUCTURADO


```
#include <conio.h>
#include <stdio.h>
#include <math.h>
main()
{
      float a, b, c, dis;
      int sig;
      clrscr();
      printf("dame los coeficientes a, b, c, donde a no sea cero ");
      scanf("%f,%f,%f", &a, &b, &c);

      dis = pow(b,2) - 4.0 * a * c;

      if( dis < 0 ) sig = -1;
      if( dis == 0 ) sig = 0;
      if( dis > 0 ) sig = 1;

      switch( sig )
      {
         case -1:

            printf("\n raíces complejas ");
            printf("x1r = %f  x1i = %f", -b/(2.0*a), -sqrt(-dis)/(2.0*a));
            printf("x2r = %f  x2i = %f", -b/(2.0*a), sqrt(-dis)/(2.0*a));
            break;

         case 0:

            printf("\n raíces repetidas ");
            printf("x1 = %f", -b/(2.0*a));
            printf("x2 = %f", -b/(2.0*a));
            break;

         case 1:

            printf("\n raíces reales ");
            printf("x1 = %f", (-b + sqrt(dis))/(2.0*a));
            printf("x2 = %f", (-b - sqrt(dis))/(2.0*a));
            break;
      }
}
```

Observe que no se requiere el default.

Como otro ejemplo de aplicación considere la asignación de premios en un sorteo en el que se extrae un dígito:

- si se extrae alguno de los dígitos 2,5,9, se asigna como premio un reloj.
- si se extrae alguno de los dígitos 1,3,4,8, se asigna como premio un radio portátil.
- si se extrae cualquier otro dígito, únicamente se le exhorta a seguir participando.

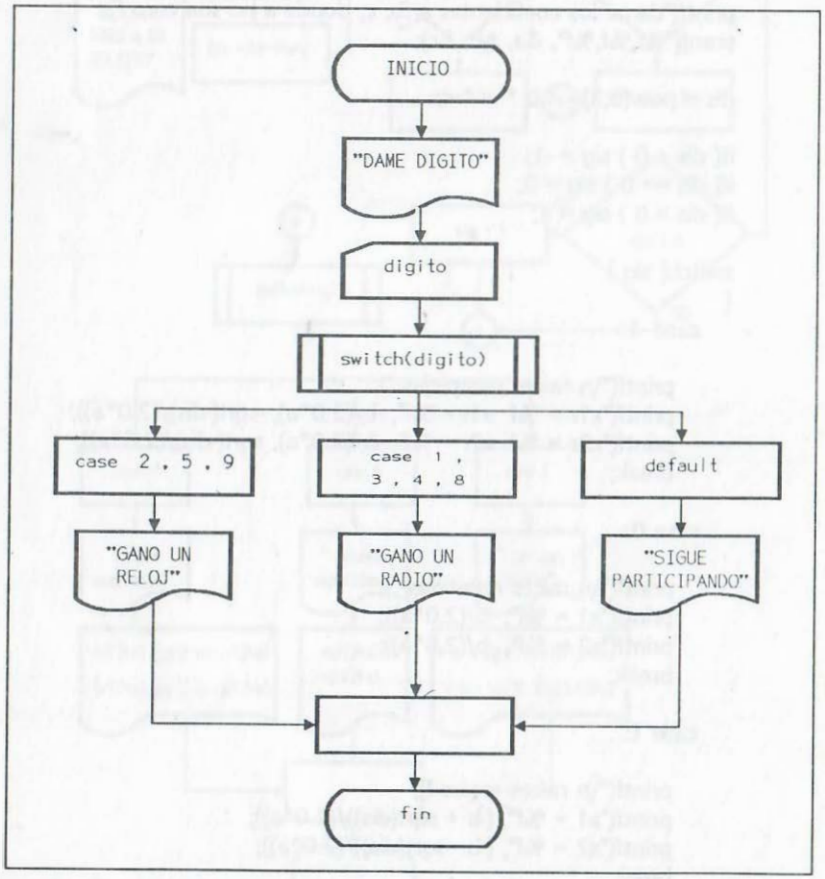


DIAGRAMA -ESTRUCTURADO

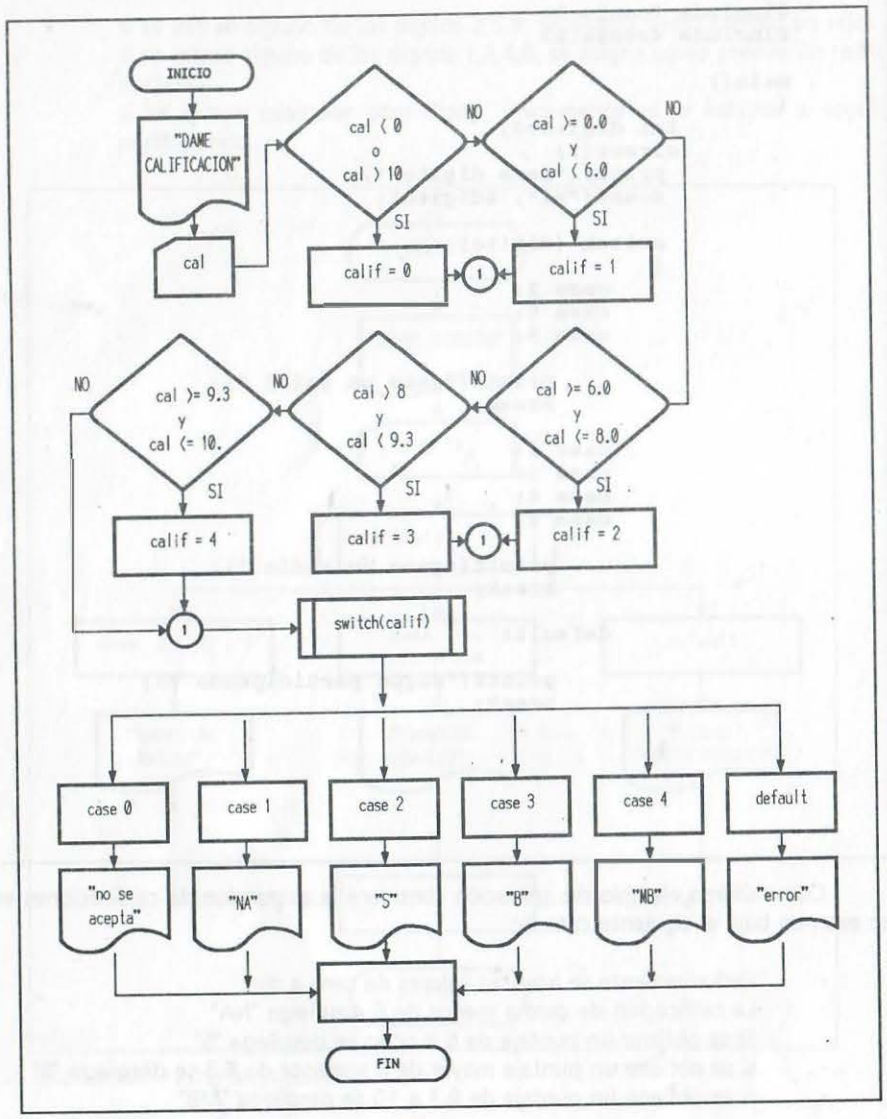
La codificación del ejemplo anterior se muestra a continuación:

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

main()
{
        int digito=0;
        clrscr();
         printf("dame digito ");
         scanf("%i", &digito);

         switch (digito)
         {
             case 2:
             case 5:
             case 9:

                 printf("gano un reloj ");
                 break;

             case 1:
             case 3:
             case 4:
             case 8:

                 printf("gano un radio ");
                 break;

             default:

                 printf("sigue participando ");
                 break;

         }
}
```

Como último ejemplo de aplicación considere la asignación de calificaciones en un examen bajo el siguiente criterio:

- Exclusivamente se aceptan valores de cero a diez
- La calificación de cero a menos de 6 despliega "NA"
- Si se obtiene un puntaje de 6 a ocho se despliega "S"
- si se obtiene un puntaje mayor de 8 y menor de 9.3 se despliega "B"
- si se obtiene un puntaje de 9.3 a 10 se despliega "MB"

Para la asignación de la calificación se asigna un dígito por cada rango de calificaciones que se mencionó previamente:



La codificación de este diagrama se muestra a continuación:


```
#include <conio.h>
#include <stdio.h>
main()
{
      int calif;
      float cal;
      clrscr();
      printf("dame calificacion ");
      scanf("%f", &cal);
      if((cal <  0   ) || (cal >10.0  )) calif = 0;
      if((cal >= 0.0 ) && (cal <  6.0 )) calif = 1;
      if((cal >= 6.0 ) && (cal <= 8.0 )) calif = 2;
      if((cal >  8.0 ) && (cal <  9.3 )) calif = 3;
      if((cal >= 9.3 ) && (cal <= 10.0)) calif = 4;

      switch (calif)
      {
          case 0:

              printf("no se acepta ");
              break;

          case 1:

              printf("\n NA");
              break;

          case 2:

              printf("\n S");
              break;

          case 3:

              printf("\n B");
              break;

          case 4:

              printf("\n MB");
              break;

          default:

              printf("\n error");
          }
}
```

Observe que no es necesario el default y que para acotar los rangos de calificación hubo necesidad de usar el elemento if().

9.16 APUNTADORES

Un apuntador es una variable que almacena la dirección o posición en memoria de otra variable apuntando o señalando la posición del primer byte. Permite:

a) Utilizar el contenido de la variable apuntada si le antecede el operador indirección (*).

b) Enviar/recibir información desde/hacia las funciones adicionales que emplea un programa a través de direcciones (envío por referencia o dirección).

c) Enviar funciones del usuario como argumentos de otra función adicional que emplea un programa (estableciendo una función variable).

d) Emplear únicamente los elementos de arreglos que tienen asignado valor, obteniendo un arreglo de apuntadores de menor dimensión.

Un apuntador debe declarar su tipo (mismo que debe coincidir con el de tipo de la variable que apuntará, es decir real o entero), seguido del operador * (operador indirección) y por último indicando el nombre que lo identifica (arbitrario y definido por el programador); las variables alfanuméricas son en sí mismas apuntadores.

Una vez que se declara un apuntador, su valor (dirección de una variable) se asigna con el operador unario & (operador dirección) aplicado a la variable a la que se apunta, según los siguientes esquema de parámetros en la memoria y declaraciones:

Para una variable alfanumérica de dos caracteres:

dirección (2 byte)	nombre	contenido
12922	y	"12"
12923		

```
char y[3]="12";   /* se declara el tipo, nombre y se asigna contenido */
```

Observe que en el caso de un arreglo alfanumérico el nombre de la variable sin índices es por definición un apuntador y no requiere del operador (*). Cuando un arreglo alfanumérico se menciona sin índice, se menciona al apuntador que hace referencia a la dirección del primer byte (y = 12922). Recuerde que al final del arreglo debe agregarse un elemento más para el caracter nulo que indica fin del arreglo ('\0').

Para una variable entera:

dirección (2 bytes)	nombre	contenido
12535	a	25
12536		

```
int a=25;   /* se declara y da valor para una variable entera   */
int *aa;    /* se declara el tipo y nombre de un apuntador      */
.
.
.
aa = &a;  /* se asigna la dirección:  aa = 12535;  al apuntador */
```

En este caso la dirección de la variable a la que apunta se obtiene por medio del operador dirección (&). En este caso se hace referencia por medio del primer byte. Por consiguiente se tiene que el valor de la variable se puede accesar con *aa o con la variable a.

Para una variable real:

dirección (4 bytes)	nombre	contenido
3527	b	47.35
3528		
3529		
3520		

```
float b=47.35;   /* se declara y da valor para una variable real   */
float *ab;    /* se declara el tipo y nombre de un apuntador real  */
.
.
.
ab = &b;  /* se asigna la dirección:  ab = 3527;  al apuntador */
```

En este caso la dirección de la variable a la que apunta se obtiene por medio del operador dirección (&). Nuevamente observe que siempre se hace referencia por medio del primer byte. El valor de la variable se puede accesar con *aa o con la variable a.

NOTA:

Las direcciones de las variables son siempre números positivos y por lo tanto su valor será en todos los casos un entero sin signo.

APUNTADORES REEMPLAZANDO A VARIABLES

Como ejemplo de acceso al contenido de las variables empleando apuntadores, considere el programa en el que se emplearán cuatro variables reales y una entera para realizar operaciones con los correspondientes apuntadores, en lugar de las variables mismas:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
main()
{
      float a, b, c, dis;          /* declaración de variables reales */
      float *aa, *ab, *ac, *adis;  /* declaración de sus apuntadores */

      int sig;    /* declaración de variable entera */
      int *asig;  /* declaración de su apuntador  */

      clrscr();
      printf("dame los coeficientes a, b, c, donde a no sea cero ");
      scanf("%f,%f,%f", &a, &b, &c);
      dis = pow(b,2) - 4.0 * a * c;

      adis = &dis;  /* se asigna la dirección del resultado al apuntador */
      asig = &sig;

      if(*adis < 0) sig = -1; /* el operador indirección refiere contenido */
      if( dis == 0 ) sig = 0;
      if(*adis > 0) sig = 1; /* usa contenido a través de su apuntador */

      switch( *asig )  /* se conmuta mediante apuntador */
      {
         case -1:
            printf("\n raíces complejas ");
            printf("x1r = %f  x1i = %f", -b/(2.0*a), -sqrt(-dis)/(2.0*a));
            printf("x2r = %f  x2i = %f", -b/(2.0*a), sqrt(-dis)/(2.0*a));
            break;
         case 0:
            printf("\n raíces repetidas ");
            printf("x1 = %f", -b/(2.0*a));
            printf("x2 = %f", -b/(2.0*a));
            break;
         case 1:
            printf("\n raíces reales ");
            printf("x1 = %f", (-b + sqrt(dis))/(2.0*a));
            printf("x2 = %f", (-b - sqrt(dis))/(2.0*a));
            break;
      }
}
```


Como otro ejemplo de manipulación de información mediante apuntadores, considere el ejemplo siguiente:

```
#include <conio.h>
#include <stdio.h>
main()
{
  float f1, f2;
  int v11, v22;
  float v33;
  int v1 = 1, v2 = 2;      /* se declaran variables y sus valores iniciales     */
  float v3 = 1.2;
  int *apunt1, *apunt2;    /* declara apuntadores para variables enteras */
  float *apunt3;           /* declara un apuntador para una variable real */
                           /* se asignan direcciones a los apuntadores */
  apunt1 = &v1;
  apunt2 = &v2;
  apunt3 = &v3;
                           /* se asignan valores a través de apuntadores */
  v11 = *apunt1;
  v22 = *apunt2;
  v33 = *apunt3;
            /* de esa forma las expresiones siguientes son equivalentes */
            /* pues se MENCIONAN valores a través de direcciones */

  f1 = (*apunt1 + *apunt2 + *apunt3)/3.2;
  f2 = (v11 + v22 +v33)/3.2;
  printf("\n f1 = %f \t\t f2 = %f ",f1,f2);
  printf("\n\n Observe que los resultados son los mismos ");
  printf("\n\n Para continuar oprime una tecla ");
  getch();
            /* de esa forma también es posible asignar valores a traves
               de los apuntadores, es decir a traves de las direcciones */
  *apunt1 = 10;
  *apunt2 = 20;
  *apunt3 = 30;
  f1 = (*apunt1 + *apunt2 + *apunt3)/3.2;
  f2 = (v11 + v22 +v33)/3.2;
  printf("\n f1 = %f \t\t f2 = %f ",f1,f2);
  printf("\n\n Observe nuevamente que los resultados son los mismos ");
  getch();
}
```

LOS APUNTADORES COMO ARGUMENTOS DE FUNCIONES

La ventaja de emplear apuntadores como argumentos radica en que los valores que contienen las variables involucradas se alteran de forma global y los nuevos valores pueden ser empleados tanto en el main() como en otras funciones. Cuando los argumentos de una función son variables comunes, sus valores son copiados para su uso local (envío de valores por valor) y, por consiguiente no válidos fuera de esa función. Considere el siguiente ejemplo:

```
#include <conio.h>
#include <stdio.h>
main()
{
  void f1(int v1, int v2);
  void f2(int *ap1, int *ap2);
  int w1 = 1, w2 = 1;      /* se declaran variables y sus valores iniciales     */


                           /* se asignan valores a través de apuntadores */
  printf("\n Valores antes de llamar a la función w1 = w2 = %d",w1);
  f1(w1,w2);
  printf("\n Valores despues de llamar a la función w1 = w2 = %d",w1);

  printf("\n\n Observe que los resultados son los mismos ");
  printf("\n cuando se pasa los argumentos por valor ");
  printf("\n\n Para continuar oprime una tecla ");
  getch();

  printf("\n Valores antes de llamar a la función w1 = w2 = %d",w1);
  f2(&w1,&w2);
  printf("\n Valores despues de llamar a la función w1 = w2 = %d",w1);

  printf("\n\n Observe que los resultados han cambiado ");
  printf("\n pues ahora se paso los argumentos por dirección ");
  printf("\n\n Para continuar oprime una tecla ");
  getch();

}
void f1(int v1, int v2)
{
 v1 = v2 = -1;
 printf("\n Nuevos valores en la función  v1 = v2 = %d",v1);
}
void f2(int *ap1, int *ap2)
{
 *ap1 = *ap2 = -1;
 printf("\n Nuevos valores en la función  *ap1 = *ap2 = %d",*ap1);
}
```


FUNCIONES COMO ARGUMENTOS DE OTRAS FUNCIONES

El manejo de direcciones en lugar de valores, permite que el nombre dado a una función genérica, sea utilizado para reemplazarlo con otras funciones que se requieran, dando lugar a lo que podría llamarse *función de reemplazo o función variable*. Para emplearlas:

a) Se declaran las funciones del usuario que reemplazarán a la función genérica (no es necesario en el caso de las funciones de biblioteca).

   Por ejemplo:

   int suma(int a, int b);

   int resta(int a, int b);

   int multiplica(int a, int b);

b) Se declaran los nombres de las funciones genéricas, según la sintaxis siguiente:

   tipo_de_función (*nombre_funcion1)();

   tipo_de_función (*nombre_funcion2)();

   tipo_de_función (*nombre_funcion"n")();

Por ejemplo: void clasifica(int a, int b, int (*opera)());

En ese caso *opera es el nombre del apuntador de la función genérica que permitirá reemplazar ese argumento con el nombre de alguna función. Observe que va siempre va encerrado entre paréntesis y seguido de otro par de paréntesis para los argumentos. La declaración int se relaciona con la dirección y por consiguiente siempre es de tipo entero.

c) En la función principal se menciona la función con el nombre de la función que le corresponda (sin argumentos), según el caso por ejemplo:

```
main(void)
{
   char r;
   printf("\n Indica operación + - * ");
   r = getche();
   switch(r)
   {
case '+':
        printf("\n se llama a clasifica con la función argumento suma ");
        clasifica(a, b, suma);        break;
case '-':
        printf("\n se llama a clasifica con la función argumento resta ");
        clasifica(a, b, resta);       break;
case '*':
        printf("\n se llama a clasifica con la función argumento multiplica");
        clasifica(a, b, multiplica); break;
default:
    printf("\n imposible creo que es error");  break;
}
```

c) Al definir la función que empleará otras funciones se menciona como argumento el nombre genérico, según el ejemplo siguiente:

```
void clasifica(int a, int b, int(*opera)())
   {
        int y;
        y = (*opera(a,b));   /* se apunta a alguna función */
        return (y);
   }
int suma(int a, int b)
   {
        return (a + b);
   }
int resta(int a, int b)
   {
        return (a - b);
   }
int multiplica(int a, int b)
   {
        return (a * b);
   }
```


El programa completo se muestra a continuación:

```
#include <stdio.h>
#include <conio.h>
int suma(int a, int b);
int resta(int a, int b);
int multiplica(int a, int b);
int clasifica(int a, int b, int (*opera)());
main(void)
{
   char r;
   int (*opera)();
   int a=1,b=1;
   printf("\n Indica operación + - * ");
   r = getche();
   switch(r)
   {
        case '+':
           printf("\n se llama a clasifica con la función argumento suma ");
           printf("\n %d  %c  %d = %d",a,r,b,clasifica(a,b,suma));
           getch(); break;
        case '-':
           printf("\n se llama a clasifica con la función argumento resta ");
           printf("\n %i  %c  %i = %d",a,r,b,clasifica(a,b,resta));
           getch(); break;
        case '*':
           printf("\n se llama a clasifica con la función argumento multiplica");
           printf("\n %d  %c  %d = %d",a,r,b,clasifica(a,b,multiplica));
           getch(); break;
        default:
           printf("\n imposible creo que es error");
           getch(); break;
       }
}
int clasifica(int a, int b, int (*opera)())
{
   int y;
   y = (*opera)(a,b);
   return y;
}
int suma(int a, int b)
{
   return (a + b);
}
int resta(int a, int b)
{
   return (a - b);
}
int multiplica(int a, int b)
{
   return (a * b);
}
```

USO DE APUNTADORES EN LOS ARREGLOS

Los arreglos y los apuntadores en C se emplean practicamente de forma indistinta, de tal manera que muchos consideran que un arreglo es un apuntador constante. Cuando se emplean arreglos en la programación (cadenas alfanuméricas, vectores y matrices), es necesario reservar memoria suficiente para todos los elementos mediante las declaraciones correspondientes, como se muestra para un vector unidimensional y tres matrices:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float vector[100];
main()
{
  double a[3][3] =
            { 1,2,1,
              2,1,2,
              1,2,1};
  double b[][3] =
            { 1,1,1,
              1,1,1,
              1,1,1};
  double c[100][100];

    PROCESO

  }
  printf("\n Hasta la próxima");
  getche();
}
```

Como se observa, en algunos casos se exagera en las dimensiones (c[100][100]) y se reserva memoria de más; el problema de desperdicio de la memoria se duplica si se hace necesario enviarlos a proceso mediante funciones adicionales o secundarias, ya que de esa forma se requiere incluir declaraciones similares en cada función adicional que se use. La forma más práctica de manipular arreglos por teclado, con las dimensiones que el caso requiera, es haciendoles referencia como apuntadores para sólo emplear aquellos elementos de los arreglos a los que se les vaya a asignar valor, junto con una asignación dinámica de memoria (empleando la función malloc y la función free() para liberar la memoria reservada).

ASIGNACIÓN DE VALORES A LOS ELEMENTOS DE ARREGLOS

Los valores de los arreglos pueden asignarse mediante lectura comno se muestra a continuación para el caso de dos matrices bidimensionales:

```
for(i=1; i<=n; i++)
  {
  for(j=1; j<=m; j++)
    {
      printf(" Dame a(%d,%d)=",i,j);
      scanf("%d",&a[i][j]);
    }
  }
printf("\n");
for(i=1; i<=n; i++)
  {
  for(j=1; j<=m; j++)
    {
      printf(" Dame b(%d,%d) =  ",i,j);
      scanf("%d",&b[i][j]);
    }
  }
```

Sin embargo, la opción más práctica para asignar valores de todo tipo de arreglos es mediante asignación de valores iniciales con declaraciones tanto de tipo estático como normales; en ellas se asigna directamente los valores de sus miembros, por ejemplo para los vectores unidimensionales y matrices siguientes:

```
int Z[]={1,2,3,4,5}, W[]={1,2,3}, T[10];

int A[][cols]={{1,2,3,4,5},
               {2,3,4,5,4},
               {3,4,5,4,3},
               {4,5,4,3,2},
               {5,4,3,2,1}},

static char *dia_semesp[8] =
                  {"Lunes", "Martes", "Miércoles", "Jueves",
                   "Viernes", "Sábado","Domingo"};
```

A continuación se muestra un programa completo para el segundo de estos casos.

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

main()
{
 int ndia,i=0;
 static char *dia_seming[7] =
                        {"Monday", "Tuesday",
                         "Wendsday", "Thursday",
                         "Friday", "Saturday",
                         "Sunday"};
 static char *dia_semesp[8] =
                        {"Lunes", "Martes",
                         "Miércoles", "Jueves",
                         "Viernes", "Sábado",
                         "Domingo"};
 clrscr();
 gotoxy(10,4);   textcolor(4);
 cprintf("  Programa que indica los nombres de los dias de la semana  ");
 gotoxy(10,6);
 printf(" tanto en español como en inglés a partir de su número (1 a 7)");
 normvideo();
 do
 {
     gotoxy(10,8+i);
     printf(" Te indico que el lunes el No 1, dame el numero del dia ");
     scanf("%d",&ndia); i++;
 }
 while((ndia>7) || (ndia<1));
 gotoxy(10,10+i);   textcolor(14);
 cprintf("  El numero %d corresponde a: ", ndia);
 gotoxy(10,12+i);   textcolor(14 +BLINK);
 cprintf("  %s  ", dia_semesp[ndia-1]);
 normvideo();
 gotoxy(22,12+i);   textcolor(9);
 cprintf("  que se traduce al inglés como ");
 normvideo();
 gotoxy(55,12+i);    textcolor(14 +BLINK);
 cprintf("  %s  ", dia_seming[ndia-1]);
 normvideo();
 gotoxy(15,15+i);    textcolor(10);
 cprintf("Para regresar al programa oprime cualquier tecla ");
 getch();
}
```


Los apuntadores permiten señalar la dirección del primer elemento. En los arreglos bidimensionales se puede no especificar el primer índice (considere un arreglo denominado *b* de 5 x 5 elementos con valor entero). Lo anterior puede expresarse de la siguiente manera:

```
int (*b)[5];   /* puede equivaler a int b[5][5]; */
```

Para n dimensiones el apuntador del arreglo se expresa:

```
tipo_apuntador (*nombre_apuntador)[indice_1][indice_2] ... [indice_n];
```

Observe que a partir de los arreglos bidimensionales el nombre del arreglo junto con el asterisco que le precede debe escribirse entre paréntesis redondos para evitar confundirlo con un arreglo multidimensional de apuntadores.

Para determinar la memoria de nuestro arreglo, se asigna arbitrariamente el número de columnas (se recomienda usar #define) y se declara el apuntador del arreglo; a continuación se proporciona por teclado el tamaño de real de cada dimensión (renglones y columnas por ejemplo), mismos que se multiplican por el número de bytes asociados a su tipo (determinados con la función *sizeof(tipo_dato)*), con lo que se obtiene el tamaño en bytes para todos los elementos del arreglo. Lo anterior queda por tanto:

```
#define cols 5
main()
{
  int i,j,k,r,t;
  int renglones, columnas;
  int (*b)[cols];  \* se declara el apuntador del arreglo  *\
  printf("Cuantos renglones de la matriz B = ");
  scanf("%d",&renglones);
  printf("Cuantas columnas de la matriz B = ");
  scanf("%d",&columnas);

  b = malloc(renglones*columnas * sizeof(int));

  PROCESO

}
```

Observe los programas siguientes para la suma y producto de dos matrices:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define cols 5
main()
{
     int i,j,k,n,m;
     int renglones, columnas;
     int (*a)[cols];  /* se declara el apuntador del arreglo  */
     int (*b)[cols];  /* se declara el apuntador del arreglo  */
     int (*c)[cols];  /* se declara el apuntador del arreglo  */
 printf("Cuantos renglones y columnas de las matrices A y B = ");
 scanf("%d,%d",&renglones,&columnas);
 a = malloc((renglones+1) * (columnas+1) * sizeof(int));
 b = malloc((renglones+1) * (columnas+1) * sizeof(int));
 c = malloc((renglones+1) * (columnas+1) * sizeof(int));
 n = renglones;       m = columnas;   printf("\n");
 for(i=1; i<=n; i++)
    {
    for(j=1; j<=m; j++)
      {
         printf(" Dame a(%d,%d)=",i,j);
         scanf("%d",&a[i][j]);
      }
    }
 printf("\n");
 for(i=1; i<=n; i++)
    {
    for(j=1; j<=m; j++)
      {
         printf(" Dame b(%d,%d) = ",i,j);
         scanf("%d",&b[i][j]);
      }
    }
 for(i=1; i<=n; i++)
    {
    for(j=1; j<=m; j++)
      {
              c[i][j] = a[i][i] + b[i][i];
      }
    }
 printf("\n La matriz suma es: \n\n");
 for(i=1; i<=n; i++)
    {
    for(j=1; j<=m; j++)
      {
         printf("%d ", c[i][j]);
      }
         printf("\n");
    }
}
```


En el primero de ellos se hace uso de apuntadores locales al main() para el manejo de los arreglos:

```
#include <stdio.h>
#include <stdlib.h>
#define cols 5
void suma(void);
main()
{
   int i, j, n, m, renglones, columnas;
   int (*a)[cols];  /* se declara el apuntador del arreglo  */
   int (*b)[cols];  /* se declara el apuntador del arreglo  */
   int (*c)[cols];  /* se declara el apuntador del arreglo  */
  printf("Cuantos renglones y columnas de las matrices A y B = ");
  scanf("%d,%d",&renglones,&columnas);
  a = malloc((renglones+1) * (columnas+1) * sizeof(int));
  b = malloc((renglones+1) * (columnas+1) * sizeof(int));
  c = malloc((renglones+1) * (columnas+1) * sizeof(int));
```

En el segundo de ellos se declaran globalmente los apuntadores y la memoria dinámica se asigna en el main():

```
#include <stdio.h>
#include <stdlib.h>
#define cols 5
int i,j,n,m;
void suma(void);
  int (*a)[cols];  /* se declara el apuntador del arreglo  */
  int (*b)[cols];  /* se declara el apuntador del arreglo  */
  int (*c)[cols];  /* se declara el apuntador del arreglo  */
void main(void)
{
  int renglones, columnas;
  printf("Cuantos renglones y columnas de las matrices A y B=");
  scanf("%d,%d",&renglones,&columnas);
  a = malloc((renglones+1) * (columnas+1) *sizeof(int));
  b = malloc((renglones+1) * (columnas+1) *sizeof(int));
  c = malloc((renglones+1) * (columnas+1) *sizeof(int));
```

de esa forma no es necesario nombrar argumentos en la función secundaria y se aplica la asignación dinámica de memoria. Cuando la memoria que se reserva no es suficiente manda el mensaje de asignación a NULL. Para liberar memoria se usa free().

A continuación se muestra un programa que realiza el producto de dos matrices, enviando arreglos como argumentos de funciones:

```
#include <stdio.h>
#define rens 5
#define cols 5
int i,j, k;
void producto(int A[rens][cols],int B[rens][cols],int C[rens][cols])
  {
    for (i=0;i<5;i++)
      for (j=0;j<5;j++)
        {
          C[i][j]=0;
          for(k=0;k<5;k++) C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
  }

void despliega(int Z[rens][cols])
  {
     for(i=0;i<rens;i++)
       {
         for(j=0;j<cols;j++) printf("%d ",Z[i][j]);
         printf("\n");
       }
  }

  void main()
  {
    int A[][cols]={{1,2,3,4,5},
                   {2,3,4,5,4},
                   {3,4,5,4,3},
                   {4,5,4,3,2},
                   {5,4,3,2,1}},
      B[][cols]={{5,4,3,2,1},
                 {4,3,2,1,2},
                 {3,2,1,2,3},
                 {2,1,2,3,4},
                 {1,2,3,4,5}},
      C[rens][cols];
      clrscr();
      printf("\n MATRIZ A\n\n"); despliega(A);
      printf("\n MATRIZ B\n\n"); despliega(B);
      puts("\n EL PRODUCTO\n");
      producto(A,B,C);
      despliega(C);
      getch();
  }
```


9.17 ENUMERACIONES

Una declaración de tipo enumerado especifica el nombre de una variable que a su vez recibirá una lista de nombres que equivalen a constantes enteras. Por ejemplo:

```
enum calificaciones
{
  c1, c2, c3, c4, c5
};
```

los nombres mencionados entre las llaves reciben un valor por la posición que ocupan, de esa manera c1 tiene el valor 0, c2 el valor 1 y asi sucesivamente hasta llegar a c5 que tiene el valor 4; sin embargo se puede asignar valor inicial diferente a cada nombre, por ejemplo:

```
enum calificaciones
{
  c1 = 10, c2 = 5, c3, = 2, c4 = 2, c5 =1
};
```

además se pueden enlistar variables adicionales que adoptan también los valores especificados en la declaración de la enumeración, por ejemplo:

```
enum calif_aprob
{
  c1 = 6, c2 = 8, c3, = 10, c4 = 12, c5 =10
}examen1;

enum calif_aprob examen2;
```

en donde examen1 y examen2 pueden recibir cualquiera de los valores de c1 a c5. Si sólo se altera el valor de uno de los elementos, se asigna a los que restan el valor entero siguiente, por ejemplo:

```
enum repite
{
  w = 6, x , y = 6, z
};
```

en la enumeración w vale 6 y por consiguiente x recibe un 7, como y vale 6 corresponde a z un 7. Los valores de las enumeraciones no pueden leerse o escribirse y normalmente se emplean cuando son asignados a otras variables.

9.18 ESTRUCTURAS

Una estructura es una variable que reune varios elementos o miembros, consta de un nombre opcional, seguido de un conjunto de valores (datos) de tipos diversos que en conjunto permiten describir un concepto, como en el caso de los registros que se tiene para préstamo de libros en una biblioteca. Los elementos de cada estructura indican su tipo cuando se encierran entre llaves (son conocidos como miembros de la estructura y generalmente están relacionados entre sí, tal como ocurre con los campos de un registro). Los miembros pueden ser de tipo entero, real, alfanumérico e inclusive otras estructuras, sin embargo el ámbito de la existencia de cada miembro está limitado a la estructura que lo define; de esa manera, pueden existir otras estructuras con elementos que tengan el mismo nombre y diferentes tipos de datos (sin estar relacionados de estructura a estructura); por último opcionalmente se puede indicar una lista de variables que tienen también asociados a los miembros de la estructura. Considere la declaración o definición de la estructura siguiente:

```
struct fecha_prestamo =  {int dia;
                          int mes;
                          char anio};
struct alumno ={char nombre[33];
                int num_cta;
                char carrera[33];
                int telefono;
               struct fecha_prestamo} generacion_95[200],generacion_96[200];
```

La estructura alumno cuenta con cuatro miembros que la describen el concepto alumno, dentro de ella se incluye otra estructura para la fecha; además generación_95[200], generación_96[200]; son arreglos que especifican 200 registros cada uno con los campos o miembros de las estructuras. Observe que la estructura fecha debe anteceder a la estructura alumno debido a que esta última le hace referencia.

Considere ahora el siguiente ejemplo:

```
struct nomb_tel
      {
        *nombre;
        *telefono;
      };
```



```
static struct nom_tel agenda[] =
                        {{"Laura",  "523-1210"},
                         {"Brenda", "645-1012"},
                         {"Alicia", "628-1010"},
                         {"Sharon", "567-8722"},
                         {"Marta",  "555-666"}};
```

en este caso se tiene un arreglo de estructuras desde la estructura 0 hasta la estructura 4, cada una con dos miembros o elementos que además definen sus valores iniciales y por consiguiente su tipo y tamaño.

Ahí se muestra una de las formas para asignar valores a los miembros de una estructura; primero se define el tipo de miembros de la estructura mediante la estructura nom_tel (que define y almacena nombre y teléfono), a continuación con la estructura estática se establece que agenda[] es del tipo nom_tel y por consiguiente se asigna los dos campos o miembros denominados *nombre, *teléfono a cada miembro de ella con los datos corresponden a cada uno de ellos.

Cuando los miembro de una estructura son a su vez estructuras, los miembros se accesan indicando el nombre de la estructura, subestructura (en caso de que haya), un punto y el nombre del miembro:

```
nombre_estructura.miembro

nombre_estructura.nombre_subestructura.miembro
```

Los valores de las estructuras preferentemente se asignan tal y como se indicó anteriormente para asegurar que el programa corre en todas las versiones de C. También se pueden asignar valores mediante lectura.

Cuando se desea definir los elementos o miembros de una estructura sin asociarle un nombre específico y al final asociarle la variable se utiliza typedef:

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

main()
{
  int y=1,i,j;
     struct nomb_tel
            {
              char *nombre;
              char *teléfono;
            };

  static struct nomb_tel agenda[] =
                                    {{"Laura ", "523-1210"},
                                     {"Brenda", "645-1012"},
                                     {"Alicia", "628-1010"},
                                     {"Sharon", "567-8722"},
                                     {"Marta ", "555-6666"}};
  clrscr();
  gotoxy(24,4);
  textcolor(4 + BLINK);
  cprintf("   Agenda de amistades internacionales  ");
  gotoxy(17,6);
  printf(" La lista corresponde exclusivamente a nuevos amigos.");
  sleep(4);            /* pausa de 4 segundos */
  normvideo();
  gotoxy(17,7);
  for(i=0; i<=4; i++)
  {
      printf("\n          %s     %s ",agenda[i].nombre,agenda[i].teléfono);
  }
  textcolor(14+BLINK);
  gotoxy(20,15);
  cprintf("Para regresar al programa oprima cualquier tecla ");
  getch();
}
```

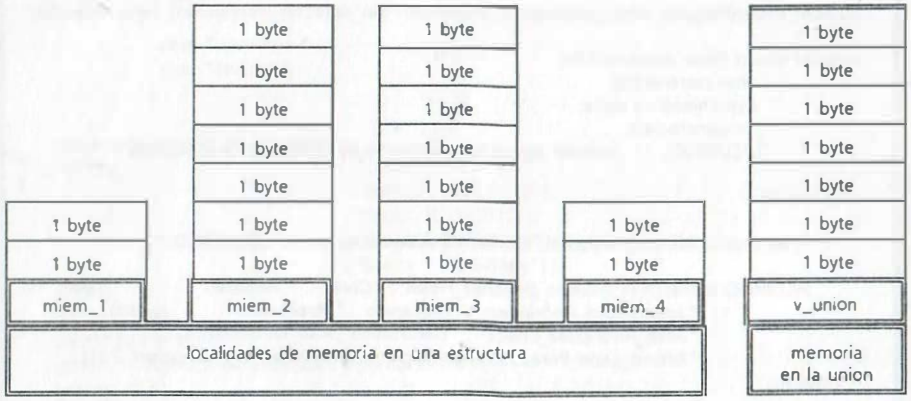
En el programa anterior se muestra una de las formas para asignar valores a los miembros de una estructura; primero se definen los miembros de la estructura mediante la estructura nom_tel (que almacena nombre y teléfono), a continuación con la estructura estática se establece que agenda[] es del tipo nom_tel y por consiguiente asigna los dos campos o miembros denominados *nombre, *teléfono a cada miembro de ella con los datos les corresponden a cada uno de ellos. Al final se mandan a despliegue en pantalla todos los miembros de agenda[].

A continuación se muestra otro ejemplo de estructuras emplendo typedef, asocoando los miembros de la estructura a ALUMNO; la información se asocia a la estructura datos[] que resulta ser de tipo ALUMNO:

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

typedef enum{Regular, Alto_rendimiento, Repetidor, Sin_derecho_inscripcion} denominacion;

typedef struct {char nombre[33];
                char carrera[33];
                denominacion clase;
                int generacion;
               }ALUMNO;  /*  typedef asocia los miembros de la estructura a ALUMNO  */

void main()
{
    /*   Se crea la estructura datos[] con las caractersticas dadas a ALUMNO  */

      ALUMNO datos[]={{"Alberto_Sanchez_Rojas", "Civil      ",Regular,              94},
                      {"Jose_Pedro_Rodriguez ", "Mecanico   ",Repetidor,            90},
                      {"Jose_Rodriguez_Lopez ", "Electronico",Alto_rendimiento,     92},
                      {"Arturo_Lepe_Perez    ", "Geologo    ",Sin_derecho_inscripcion, 75}};
  clrscr();

  printf("\t\t\t\t Datos de los alumnos\a\n");

  printf("\n%s  Carrera %s  Generacion %2i  Tipo  %2i\n",datos[0].nombre,
          datos[0].carrera, datos[0].generacion, datos[0].clase);
  printf("%s  Carrera %s  Generacion %2i  Tipo  %2i\n",datos[1].nombre,
          datos[1].carrera, datos[1].generacion, datos[1].clase);
  printf("%s  Carrera %s  Generacion %2i  Tipo  %2i\n",datos[2].nombre,
          datos[2].carrera, datos[2].generacion, datos[2].clase);
  printf("%s  Carrera %s  Generacion %2i  Tipo  %2i",datos[3].nombre,
          detos[3].carrera, datos[3].generacion, datos[3].clase);

  printf("\n\nTipos:\n 0  = regular\n 1  = Alto rendimiento\n 2  = Repetidor\n");
  printf(" 3  = Sin derecho a inscripcion");
  getch();
}
```

9.19 UNIONES

Las uniones son variables que se definen exactamente como las estructuras pero a diferencia de ellas, sus elementos se almacenan en un solo registro variable que puede recibir miembros de tipos diversos (entero, real o alfanumérico) en la misma localidad de memoria. El tamaño de la localidad de memoria deberá permitir almacenar al más grande de los miembros de la unión.



Según el esquema anterior el tamaño máximo de la union debe alcanzar 8 bytes para una localidad de memoria única y no cuatro como en el caso de la estructura. De esa forma cuando se emplea un miembro su valor reemplaza al anterior y así cada vez que se emplea un miembro diferente. Por ejemplo:

```
union generacion =
        {
         int anio;
         char nombre[8];
         char carrera[8];
         int teléfono;
         };

union generacion gen_95, gen_96;
```

en el que se declara las variables denominadas gen_95 y gen_96 que almacenarán a cualquiera de los cuatro miembros cuando se utilicen en el programa según la secuencia que indique el algoritmo que se resuelva, es decir se deberá tener conocimiento sobre el miembro que está en la union en un momento dado, de tal manera que se obtengan los resultados deseados.

A continuación se muestra la codificación de un programa que hace uso de enumeraciones, uniones y estructuras para crear una base de datos con información de alumnos:

NOTA:

Observe que en la función principal la estructura denominada persona, hace mención a las estructuras fecha y dirección, motivo por el cual se declaran antecediéndole.

```
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#define ff fflush(stdin)
#define x 4
char menu(int numreg);
void lee_base(int numreg);
int crea_base(int numreg);
enum sit_actual {soltero, casado, viudo, divorciado};
enum estudiante {con_repreg, sin_repreg, oyente, prog_PARA};
enum beca {tiene, notiene};
struct fecha
        {
          unsigned int dia;
          unsigned int mee;
          unsigned int anio;
        };
struct direccion
            {
               char n[6];
               char calle[20];
               char pobl[20];
               char cod_post[6];
            };
struct persona
            {
               char nombre[25];
               char apellido[25];
               enum sit_actual est_civil;
               struct fecha nacimiento;
               struct direccion dir;
               enum estudiante tipo_est;
               union seleccion
               {
                  char nombre[25];
                  enum beca ayuda;
               } opcion;
            };
struct persona tab[x];
void main(void)
{
   int numreg = 1;
   char resp1;
   do
   {
      resp1 = menu(numreg);
      if(resp1 == 'P') numreg = crea_base(numreg);
      if(resp1 == 'L') lee_base(numreg);
   }
   while(resp1 != 'T');
   gotoxy(40,24);  printf("Hasta la próxima ");
   getch();
   clrscr();
}
```

NOTA:

La pantalla para captura de datos, se diseño con ayuda de una tabla o arreglo bidimensional de 80 columnas por 25 renglones que permitió ubicar las posiciones para despliegue de mensajes y para lectura de datos.

```
/*====================== FUNCIÓN PARA DESPLIEGUE DE MENU ====================*/
char menu(int numreg)
{
   int bandera = 0;
   char resp;
   do
   {
      clrscr();
      gotoxy(5,2); printf("Menu para creación de registros ");
      gotoxy(5,3); printf("============================== ");
      if(numreg != (x+1))
      {
           gotoxy(5,8);
           printf("Proporcionar datos (máximo 4 registros)... 'P'");
      }
      if(numreg != 1)
              {
                    gotoxy(5,10);
                    printf("Leer datos.............................. 'L'");
              }
      gotoxy(5,12); printf("Para terminar la sesión.................. 'T'");
      gotoxy(5,16); printf("Seleccione su opción "); ff;
      resp = toupper(getche());
      if((resp!='T' && resp!='L'&& resp!='P') || (numreg == (x+1) && resp == 'P') ||
      ((numreg == 1) && resp == 'L'))
      {
         gotoxy(26,16); clreol(); printf("SELECCIÓN ERRÓNEA"); sleep(3);
         ff;
      }
      else bandera = 1;
    }
    while(bandera == 0);
    return resp;
}
/*=========================== FUNCIÓN PARA CREAR PANTALLA DE CAPTURA DE DATOS ============*/
void pantalla_datos(int n, int numreg)
{
   if(n == 1)
   {
      printf("\t\t Captura de registros desde teclado");
      printf("\t Registro núm %d",numreg);
   }
   if(n == 2)
   {
      printf("\t\t Lectura de registros de la base");
      printf("\t Registro núm %d",numreg);
   }
   gotoxy(1,3);printf("Tipo de estudiante: (A, B, C, D) ");
   gotoxy(1,5);printf("Nombre: ");
   gotoxy(1,7);printf("Apellidos: ");
   gotoxy(1,9);printf("Estado civil: (S, C, V, D)  ");
   if(n == 1)
   {
      gotoxy(1,11);printf("Fecha de nacimiento (dd, mm, aaaa): ");
      gotoxy(6,12);printf("Dia (1 a 31).....: ");
      gotoxy(6,13);printf("Mes (1 A 12).....: ");
      gotoxy(6,14);printf("Año (mayor de 1582): ");
   }
   if(n == 2)
   {
      gotoxy(1,11);printf("Fecha de nacimiento  ");
      gotoxy(6,13);printf("Nació el ");
      gotoxy(18,13);printf("de ");
      gotoxy(34,13);printf("del año de ");
   }
   gotoxy(1,16);printf("Beca: (S,N) ");
   gotoxy(1,18);printf("Dirección: ");
   gotoxy(10,19);printf(" Calle.....: ");
   gotoxy(10,20);printf(" Numero....: ");
   gotoxy(10,21);printf(" Población.....: ");
   gotoxy(10,22);printf(" Código postal: ");
}
```

NOTA:

A continuación se muestra la función que crea la base de datos (4 registros) para lo cual emplea filtros que evitan se proporcione información falsa en cuanto al tipo de estudiante, su estado civil, fecha de nacimiento y beca.


```
/*====================== FUNCIÓN PARA CREAR LA BASE DE DATOS ================*/
int crea_base(int numreg)
{
   char resp;
   int n;
   int bandera=0;
   int a, la, m, lm, d, ld;
   do
   {
      clrscr();
      n = 1;
      pantalla_datos(n, numreg);
      if(numreg==x)
      {
           textcolor(6+BLINK);
           gotoxy(50,2); cprintf("¡último registro! ");
           normvideo();
      }
      do
      {
         bandera=0;
         gotoxy(34,3); ff; resp = toupper(getche());
         switch(resp)
         {
            case 'A': tab[numreg-1].tipo_est = con_repreg;break;
            case 'B': tab[numreg-1].tipo_est = sin_repreg;break;
            case 'C': tab[numreg-1].tipo_est = oyente;break;
            case 'D': tab[numreg-1].tipo_est = prog_PARA;break;
              default : bandera=1;gotoxy(34,3);printf("error"); sleep(2);
              gotoxy(34,3);clreol();
         }
      }
      while(bandera==1);
      {
      gotoxy(10,5); ff; gets(tab[numreg-1].nombre);
      strupr(tab[numreg-1].nombre);
      gotoxy(13,7); ff; gets(tab[numreg-1].apellido);
      strupr(tab[numreg-1].apellido);
      do
      {
         gotoxy(29,9); ff;
         resp=toupper(getche());
         if((resp !='C') && (resp != 'S') && resp != 'V' && resp != 'D')
         {
            gotoxy(28,9); printf("error "); sleep(3);
            gotoxy(28,9);clreol();
         }
       }
       while ((resp != 'C') && (resp != 'S' ) && resp != 'V' && resp != 'D');
       switch(resp)
       {
          case 'S': tab[numreg-1].est_civil = soltero; break;
          case 'C': tab[numreg-1].est_civil = casado; break;
          case 'V': tab[numreg-1].est_civil = viudo; break;
          case 'D': tab[numreg-1].est_civil = divorciado; break;
       }
       do
       {
          bandera=0;
          gotoxy(29,12); ff;
          scanf("%d",&d);
          gotoxy(29,13); ff;
          scanf("%d",&m);
          lm = (m >= 1) && (m <= 12);
          gotoxy(29,14); ff;
          scanf("%d",&a);
          la = (a >= 1582);
          switch(m)
          {
             case 2:
              if((a % 4 == 0) && (a % 100 != 0) || (a % 400 == 0))
                 ld = (d >= 1) && (d <= 29);
              else
                 ld = (d >= 1) && (d <= 28);
                 break;
              case 4: case 6: case 9: case 11:
                 ld = (d >= 1) && (d <= 30);
                 break;
              default:
                 ld = (d >= 1) && (d <= 31);
          }
```

```
        if(!(ld && lm && la))
        {
           gotoxy(29,12); ff;
           printf("DATOS NO VALIDOS");sleep(2);
           gotoxy(29,12); clreol();
           gotoxy(29,13);
           printf("DATOS NO VALIDOS");sleep(2);
           gotoxy(29,13); clreol();
           gotoxy(29,14);
           printf("DATOS NO VALIDOS");sleep(2);
           gotoxy(29,14); clreol();
           bandera = 1;
        }
        else
        {
           bandera = 0;
           tab[numreg-1].nacimiento.dia = d;
           tab[numreg-1].nacimiento.mes = m;
           tab[numreg-1].nacimiento.anio = a;
        }
     }
     while(bandera==1);
  }
  while(bandera==1);
  do
  {
     bandera=0;
     gotoxy(15,16); ff; resp = toupper(getche());
     if(resp!='S' && resp!='N')
     {
        bandera=1;
          gotoxy(15,16); printf("error"); sleep(2);
        gotoxy(15,16); clreol();
     }
  }
  while(bandera==1);
  tab[numreg-1].opcion.ayuda=((resp=='S') ? tiene:notiene);
  gotoxy(26,19); ff;gets(tab[numreg-1].dir.calle); strupr(tab[numreg-1].dir.calle);
  gotoxy(26,20); ff;gets(tab[numreg-1].dir.n); strupr(tab[numreg-1].dir.n);
  gotoxy(26,21); ff;gets(tab[numreg-1].dir.pobl); strupr(tab[numreg-1].dir.pobl);
  gotoxy(26,22); ff;gets(tab[numreg-1].dir.cod_post); strupr(tab[numreg-1].dir.cod_post);
  numreg++;
  gotoxy(1,25); printf("Para continuar oprima cualquier tecla, Para terminar teclee $ ");
  resp = getche();
  }
  while(resp != '$' && numreg <= x);
  return (numreg);
}
```

NOTA:

Por último se muestra la función que lee los registros de la base que se ha creado, la cual permite ampliarla cuando no cuenta con el máximo de cuatro registros; posteriormente se puede navegar entre ellos o decidir salir de la sesión.


```
/*=============== FUNCIÓN PARA LEER REGISTROS DE LA BASE ===============*/
void lee_base(int numreg)
{
   int n;
   typedef char tabla[25];
   int ficha=1, bandera=0;
   char resp;
   static char *tabestud[35] = {" regular con asignaturas reprobadas",
                                " sin ninguna asignatura reprobada ",
                                " sin derecho a inscripción",
                                " de alto rendimiento académico"};
   tabla tabayuda[] = {" tiene beca asignada", " no tiene beca"};
   tabla tabedo_civil[] = {" soltero(a)", " casado(a)",
                           " viudo(a)"," divorciado(a)"};
   tabla tabmes[] = {"  enero   "," febrero  "," marzo   "," abril   ",
                     "  mayo    "," junio   "," julio   "," agosto  ",
                     "septiembre"," octubre  "," noviembre ","diciembre"};
   textcolor(4);
   textbackground(7);
   do
   {
      clrscr();
      n = 2;
      pantalla_datos(n, ficha);
      gotoxy(20,3); clreol();
      printf("%s",tabestud[tab[ficha-1].tipo_est]);
      gotoxy(13,5); printf("%s",tab[ficha-1].nombre);
      gotoxy(13,7); printf("%s",tab[ficha-1].apellido);
      gotoxy(14,9); clreol();
      printf(" %s",tabedo_civil[tab[ficha-1].est_civil]);
      gotoxy(15,13);printf("%d",tab[ficha-1].nacimiento.dia);
      gotoxy(22,13);printf("%s",tabmes[tab[ficha-1].nacimiento.mes-1]);
      gotoxy(45,13);printf("%d",tab[ficha-1].nacimiento.anio);
      gotoxy(7,16); clreol();
      printf("%s",tabayuda[tab[ficha-1].opcion.ayuda]);
      gotoxy(27,19); printf("%s",tab[ficha-1].dir.calle);
      gotoxy(27,20); printf("%s",tab[ficha-1].dir.n);
      gotoxy(27,21); printf("%s",tab[ficha-1].dir.pobl);
      gotoxy(27,22); printf("%s",tab[ficha-1].dir.cod_post);
      do
      {
         bandera=0;
         if(ficha != x && ficha != (numreg-1))
         {
              gotoxy(40,22); printf("'+' para ver el registro siguiente");
         }
         if(ficha != 1)
         {
              gotoxy(40,23); printf("'-' para ver el registro anterior");
         }
         gotoxy(10,24);
         printf("'$' para salir de la lectura de registros de la base ");
         resp = getche();
         if(numreg>2)
         switch(resp)
         {
            case '+': if (ficha != (numreg-1)) ficha++; break;
            case '-': if (ficha != 1) ficha--; break;
            case '$': break;
            default : gotoxy(62,24); printf(" error");sleep(2);
                      gotoxy(62,24); clreol();
                      bandera = 1;
         }
      }
      while(bandera==1);
   }
   while(resp != '$');
   textcolor(5); textbackground(0);
}
```

APÉNDICE 1

ALGORITMOS NUMÉRICOS

ESCRITOS EN LENGUAJE C

El siguiente programa muestra la lectura y escritura de valores enteros, reales, alfanuméricos y uso de funciones de biblioteca.

Observe que la única diferencia entre valores ENTEROS y valores REALES es la presencia del punto decimal y que la aritmética entera siempre da como resultado ENTERO (12/24 es cero). Para visualizar cada resultado oprima una tecla.

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

main()
{
  char a[5] = "hola";  /*    declaraciones de variables    */
  char w = 'x'
  int a = 12, b = 24, c, d;
  float aa = 1.2345;
  float x;

  clrscr();          /*    lectura y escritura de un valor entero  */
  printf("Dame un valor entero  ");
  scanf("%d",&d);
  printf("El valor de d = %d",d);
  printf("La división entera 12/24 da %d",a/b);
  getche();

  clrscr();          /*    lectura y escritura de un valor real  */
  printf("Dame un valor real  ");
  scanf("%f",&a);
  printf("El valor de a = %f",a);
  getche();

  clrscr();          /*    escritura de una cadena y un caracter (alfanuméricos)  */
  printf("El valor de a = %s",a);
  printf("El valor de w = %c",w);
  getche();

  clrscr();          /*  empleo de funciones de biblioteca (utilizan prototipo math.h)  */
  printf("Dame un valor real ?  ");
  scanf("%f",&x);
  printf("\n\nEl seno(%f) = >%f<",x,sin(x));
  printf("\n\nEl coseno(%f) = >%f<",x,cos(x));
  getche();

}
```


El siguiente programa muestra la conversión automática de valores de decimal a octal y hexadecimal, así como ejemplo de operaciones en octal y hexadecimal.

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

main()
{
  int a=12;      /*  declaraciones de variables    */
  int b;
  int c;

  clrscr();
  printf("\nDame un valor octal (solo digitos del 0 al 7 )");
  scanf("%o",&b);
  printf("\nDame un valor hexadecimal (dígitos del 0 al 9, A,B,C,D,E,F )");
  scanf("%x",&c);
  clrscr();

  printf("\nEl valor decimal %d se escribe en octal %o y en hexadecimal %x ",a,a,a);
  printf("\n\nLa suma octal de %o + %o es = %o",b*b,b,b*b+b);
  printf("\n\nLa suma hexadecimal de %x x %x es = %x",c*c,c,c*c+c);

  printf("\n\nLa resta octal de %o + %o es = %o",b*b,b,b*b-b);
  printf("\n\nLa resta hexadecimal de %x x %x es = %x",c*c,c,c*c-c);

  printf("\n\nEl producto octal de %o x %o es = %o",b,b,b*b);
  printf("\n\nEl producto hexadecimal de %x x %x es = %x",c,c,c*c);

  printf("\n\nLa división octal de %o + %o es = %o",b*b,b,b*b/b);
  printf("\n\nLa suma hexadecimal de %x x %x es = %x",c*c,c,c*c/c);

  getche();
}
```

El siguiente programa compara valores de tipo ENTERO y valores de tipo REAL.

```
main()
{
  int a,b;
  clrscr();
/*              USO DE LA INSTRUCCIÓN if(condición) instrucción_ejecutable;

 Recordemos que la forma mas simple de la instrucción if() nos permite
 ejecutar o saltar una instrucción en función de una pregunta que normalmente
 utiliza los operadores relacionales siguientes:

            Igual          (==)
            Diferente      (!=)
            Mayor          (>)
            Menor          (<)
            Mayor o Igual  (>=)
            Mayor o Igual  (<=)
*/
  printf("Introduce un valor ENTERO para 'a' ? ");
  scanf("%d",&a);
  printf("Introduce un valor ENTERO para 'b' ? ");
  scanf("%d",&b);
  if(a != b)
  {
     printf("\n a es Diferente de b (a != b)");
  }
  if(a == b)
  {
     printf("\n a es Igual que b (a == b)");
  }
/* Observe que cuando el if()...; únicamente tiene en su ámbito a una sola instrucción,
    se pueden eliminar las llaves y escribir como a continuación se indica:
*/
  if(a > b)printf("\n a es Mayor que b (a > b)");
  if(a < b)printf("\n a es Menor que b (a < b)");
  if(a >= b)printf("\n a es Mayor o Igual que b (a >= b)");
  if(a <= b)printf("\n a es Menor o Igual que b (a <= b)");
  printf("\n\n a=%d ,  b=%d",a,b);
  getche();
  printf("Introduce un valor REAL para 'a' ? ");
  scanf("%f",&a);
  if(a >= 3.5)
  {
     printf("\nEl valor de la variable 'a'= %fes mayor o igual a 3.5 ",a);
  }
  getch();
}
```


El siguiente programa compara valores de tipo real estableciendo intervalos según el resultado de la pregunta lógica.

```
mai.ı()
{
  float s;

  clrscr();

  /*        USO DE if() CON OPERADORES LÓGICOS Y RELACIONALES


  En este caso solo se mencionan los operadores lógicos siguientes:

            AND (y)          OR (o)
          F    F  F        F    F  F
          F    V  F        F    V  V
          V    F  F        V    F  V
          V    V  V        V    V  V

  Recuerde que para que el resultado de una pregunta que involucra varias
  condiciones con el operador AND es verdadera cuando se cumplen todas las
  condiciones y es falsa si alguna no se cumple (ver tabla anterior).

  En forma similar, para que el resultado de una pregunta que involucra varias
  condiciones con el operador OR es verdadera cuando al menos se cumple una de
  las condiciones y es falsa cuando ninguna se cumple (ver tabla anterior).

 */

  printf("Introduce un valor REAL para 's' ? ");
  scanf("%f",&s);

 */  '     a continuación se usa el AND que en lenguaje C se indica con &&    */

  if((s >= 0) && (s <= 8))
  {
    printf("\n\nEl valor de la variable 's' se encuentra entre 0 y 8");
  }

 */       a continuación se usa el OR que en lenguaje C se indica con ||    */

  if((s < 0) || (s > 8))
  {
    printf("\n\nEl valor de la variable 's' NO se encuentra entre 0 y 8");
  }

  getche();
}
```

El siguiente programa compara valores de tipo alfanumérico.

```
main()
{
  char a[20].b[20];
  char y;
  clrscr();
/*              USO DE LA INSTRUCCIÓN if(condición)

 En el caso de una variable tipo cadena utilizaremos una función que
 se encuentra en la librería <string.h> que se llama strcmp(); Esta función
 regresa un CERO si las cadenas a comparar son iguales, y un numero
 DIFERENTE DE CERO si las cadenas son diferentes.

*/
  printf("Introduce un valor ALFANUMÉRICO para 'a' ? ");
  scanf("%s",a);

  printf("Introduce un valor ALFANUMÉRICO para 'b' ? ");
  scanf("%s",b);

  printf("\n\nLa función strcmp regresa %d",strcmp(a,b));

  if( strcmp(a,b) == 0)
  {
    printf("\n a es Igual que b (a == b)");
  }

  if( strcmp(a,b) != 0)
  {
    printf("\n a es Diferente de b (a != b)");
  }

  printf("\n\n a='%s' ,  b='%s'",a,b);
  printf("\n Oprima una s para continuar u otra letra para salir ");
  y = getche();
/*

 En el caso de una variable tipo caracter es más práctico leerla con la función
 getche();. Esta función captura el caracter tecleado, el cual puede compararse
 directamente con algún otro caracter.
*/

  if( y != 's') printf("\n Lo que deseas es terminar. Hasta la vista");
  if( y == 's') printf("\n Lo que deseas es continuar...");
  getche();
}
```

A continuación se muestra un programa para el if - else que permite ejecutar uno de dos conjuntos de instrucciones (ramificación).


```
main()
{
  int a,b;
  clrscr();
/*          USO DE LA INSTRUCCIÓN if - else

 La instrucción de ramificación if - else se emplea cuando se requiere ejecutar
 uno de dos bloques de instrucciones en función de una pregunta que, al igual que el
 if sencillo, normalmente utiliza los operadores relacionales siguientes:

          Igual          (==)
          Diferente      (!=)
          Mayor          (>)
          Menor          (<)
          Mayor o Igual  (>=)
          Menor o Igual  (<=)

 Si la pregunta es verdadera se ejecuta un bloque, y si la pregunta es falsa
 se ejecuta otro bloque de instrucciones diferente. Obviamente un bloque es dos o más
 de dos instrucciones
*/

  printf("Introduce un valor ENTERO para 'a' ? ");
  scanf("%d",&a);

  printf("Introduce un valor ENTERO para 'b' ? ");
  scanf("%d",&b);

  if (a == b)
  {
      printf("\n a es Igual que b (a == b)");
  }
  else
  {
      if (a > b)
      {
          printf("\n a es Mayor que b (a > b)");
          printf("\n en este caso se realiza la suma (a + b) %d ", a+b);
      }
      else
      {
          printf("\n a es Menor que b (a < b)");
          printf("\n en este caso se realiza la resta (a - b) %d ", a-b);
      }
  }

  printf("\n\n Los valores leidos son a=%d ,  b=%d",a,b);

  getch();
}
```

El siguiente programa calcula una sumatoria involucrando contadores y factoriales:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float a[100];
float serie1(int n, float x);
main()
{
   int i, n,r, r1;
   float d, x;
   clrscr();
 /* programa para evaluar la serie:

                serie1 = 1 + x + x^2/2! + x^3/3! + x^4/4! + ... + x^n/n!


 en el punto x0 */

   printf("\n Cálculo de la serie en el punto x0 ");
   do
   {
      printf("\n\n Dame numero de términos (n >= 1) ");
      scanf("%i",&n);
      if(n<1) printf("\nDato no válido ");
   }
   while(n < 1);
   n = n + 1;
   printf("\n Dame el valor x0 ",n+1);
   scanf("%f",&x);
   printf("\n La serie para x = %f  vale %f", x, serie1(n,x));
   printf("\n\n Fin de proceso  ");
   getch();
}
float serie1(int n,float x)
{
   float s1= 1.0, fact = 1;
   int e;
   for( e = 1; e <= n-2; e++)
   {
      fact = fact*(e);
      s1 = s1 +  pow(x,e)/fact;
   }
   return s1;
}
```


El siguiente programa calcula otra sumatoria involucrando contadores y factoriales:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float a[100];
float serie2(int n, float x);
main()
{
   int i, n,r, r1;
   float d, x;
   clrscr();
 /* programa para evaluar la serie:

                serie2 = 1 + x^2/2! + x^4/4! + x^6/6! + ... + x^2n/2n!


 en el punto x0 */
   printf("\n Cálculo de la serie en el punto x0 ");
   do
   {
      printf("\n\n Dame numero de términos (n >= 1) ");
      scanf("%i",&n);
      if(n<1) printf("\nDato no válido ");
   }
   while(n < 1);
   n = n + 1;
   printf("\n Dame el valor x0 ",n+1);
   scanf("%f",&x);
   printf("\n La serie para x = %f  vale %f", x, serie2(n,x));
   printf("\n\n Fin de proceso  ");
   getch();
}
float serie2(int n,float x)
{
   float s2= 1.0, fact = 1;
   int e;
   for( e = 1; e <= n-2; e++)
   {
      fact = fact*(2.0*e-1)*(2.0*e);
      s2 = s2 +  pow(x,2.0*e)/fact;
   }
   return s2;
}
```

El siguiente programa calcula la división sintética de un polinomio para un valor x0, de tal manera que se obtiene el residuo del polinomio en el punto x0.

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float a[100], b[100];
float polinomio(int n, float x);
main()
{
     int i, n,r, r1;
     float d, x, x0;
     clrscr();
   /* programa para obtener el residuo de un polinomio de grado "n" */

     printf("\n Se obtiene el residuo por división sintética y con la función directamente ");
     printf("\n\n Dame grado ");
     scanf("%i",&n);
     printf("\n Dame los %i coeficientes\n",n+1);

     for(i = 0; i <= n; i++)
     {
          printf("\n coeficiente de x^%2d= ?  ",n-i);
          scanf("%f", &a[i]);
     }
     printf("\n Dame el valor x0 para obtener el residuo ");
     scanf("%f", &x0);
     b[0] = a[0];
     for(i=1; i<= n; i++)
     {
         b[i] = x0*b[i-1] + a[i];
     }
     printf("\n Con la división sintética: F( %f ) = %f  ", x0, b[n]);
     printf("\n\n Directamente con la función F( %f ) = %f", x0, polinomio(n,x0));
     printf("\n\n Hasta la próxima ");
     getch();
}
float polinomio(int n,float x)
{
     float px= 0.0;

     int i = 0, e;

     for( e = n; e >= 0; e--)
     {
          px = px+ a[i] * pow(x, e);
          i = i + 1;
     }
     return px;
}
```


El siguiente programa calcula la media aritmetica de un conjunto de datos:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float x[100];
main()
{
    int i, n;
    float media;
    clrscr();
 /*                    MEDIDAS DE TENDENCIA CENTRAL                    */

 /* programa para obtener la media aritmética de un conjunto de datos (la muestra); medida
    numérica también conocida como valor promedio de las observaciones de la muestra    */
    printf("\n MEDIDAS DE TENDENCIA CENTRAL: LA MEDIA ARITMETICA  ");
    printf("\n\n Dame el número de datos ");
    scanf("%i",&n);

    printf("\n Proporciona la muestra, dato tras dato\n");

    for(i = 1; i <= n-1; i++)
    {
        printf("\n Elemento X(%2d) = ?  ",i);
        scanf("%f", &x[i]);
    }
    printf("\n\a Dame el último elemento X(%2d) = ?  ",i);
    scanf("%f", &x[i]);
    media = 0.0;
    for(i=1; i<= n; i++)
    {
        media += x[i];
    }
    media = media/n;
    printf("\n La media (o promedio de la muestra) es: %f ", media);
    printf("\n\n Hasta la próxima ");
    getch();
    return 0;
}
```

El siguiente programa calcula la varianza y la desviación estandard de un conjunto de datos:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
float x[100];
main()
{
    int i, n;
    float media, varianza;
    clrscr();
/*                          MEDIDAS DE DISPERSIÓN                    */

/* La varianza es un valor que nos indica que tan dispersos o separados están los
   datos, si los datos coinciden en un sólo valor la varianza es cero (no hay dispersión).
   La varianza se emplea para determinar cual de dos o más conjuntos de datos está más
   disperso (MAYOR VALOR EN LA VARIANZA SIGNIFICA MAYOR DISPERSIÓN)  */

/* La desviación estandar es la raíz cuadrada de la varianza y nos da una medida de
   dispersión en las mismas unidades que la variable original (los datos)        */

    printf("\n MEDIDAS DE DISPERSIÓN: LA VARIANZA Y LA DESVIACIÓN ESTANDAR ");
    printf("\n\n Dame el número de datos ");
    scanf("%i",&n);
    printf("\n Proporciona la muestra, dato tras dato\n");
    for(i = 1; i <= n-1; i++)
    {
        printf("\n Elemento X(%2d) = ?  ",i);
        scanf("%f", &x[i]);
    }
    printf("\n\a Dame el último elemento X(%2d) = ?  ",i);
    scanf("%f", &x[i]);
    media = 0.0;
    for(i=1; i<= n; i++) media += x[i];
    media /= n;
    varianza = 0;
    for(i=1; i<= n; i++) varianza += (x[i] - media)*(x[i] - media);
    varianza /= (n - 1);
    printf("\n La varianza de la muestra) es: %f ", varianza);
    printf("\n La desviación estandar de la muestra) es: %f ", sqrt(varianza));
    printf("\n\n Hasta la próxima ");
    getch();
    return 0;
}
```


El siguiente programa calcula el número de permutaciones y combinaciones de 'r' elemntos seleccionados de un total de 'n' posibles:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
long factorial(int x);
main()
{
    int n, r;
    long permutaciones, combinaciones;
    clrscr();
/*                          PERMUTACIONES Y COMBINACIONES                */

 /* Dado un conjunto de 'n' objetos distinto, las permutaciones son los distintos arreglos de 'r'
    objetos distintos (el orden de aparición de los objetos genera una permutación)
    (el numero demaneras de seleccionar 'r' objetos de los 'n' y de permutar los 'r' objetos) */

 /* Dado un conjunto de 'n' objetos distinto, las combinaciones son los distintos arreglos de 'r'
    objetos distintos en los cuales no importa el orden de aparición de ellos. De esa forma
    es menor que el número de permutaciones.                              */

    printf("\n PERMUTACIONES Y COMBINACIONES ");
    do
    {
        printf("\n\n Dame total de objetos que tienes ");
        scanf("%d",&n);
        printf("\n Cuantos objetos seleccionas (r <= n) ");
        scanf("%d",&r);
        if(r>n) printf("\n Datos erróneos ");
    }
    while(r>n);
    permutaciones = factorial(n)/factorial(n-r);
    combinaciones = permutaciones/factorial(r);

    printf("\n El total de permutacion(es) al seleccionar %i elemento(s) de %i es ", r, n);
    textbackground(3);textcolor(4  + BLINK); cprintf("%-10li",permutaciones);
    printf("\n\n El total de combinacion(es) al seleccionar %i elemento(s) de %i es ", r, n);
    textbackground(7); textcolor(1 + BLINK); cprintf("%10ld",combinaciones);
    printf("\n\n Hasta la próxima ");
    getch();
    return 0;
}
long factorial(int z)
{
    int i;
    long fact = 1.0;
    for(i=1; i<=z; i++)
    {
        fact = fact*i;
    }
    return (fact);
}
```

El siguiente programa busca una raíz real mediante el método de cambio de signo en las ordenadas, al encontrarlo reconsidera el entorno del cambio con un incremento menor; el proceso se repite hasta alcanzar la precisión deseada para el valor de la raíz.

```
#include <math.h>
#include <stdio.h>
#include <conio.h>
double f(double y);
double h=0,tol=0.00001, yy, aux;
int vuelta=0;
main()                  /* empieza la función principal  */
{
   double x, nmi, inc, x1, x2, y=0;
   do
   {
      printf("\n Dame intervalo para tabular e incremento");
      printf("\n Dame valor inicial x1 ");
      scanf("%lf",&x1);
      printf("\n Dame valor final x2 ");
      scanf("%lf",&x2);
      printf("\n Dame número máximo de vueltas ");
      scanf("%lf",&nmi);
      printf("\n  Dame tolerancia ");
      scanf("%lf",&tol);
      if(x1 > x2 || nmi == 0 || tol == 0) printf("\n\n Algún dato no es válido \n ");
   }
   while(x1 > x2 || nmi == 0 || tol == 0);
   inc = (x2 - x1)/nmi;
   y = f(x1);
   x1 = x1 + inc;
   for(x = x1; x <=x2; x = x + inc)
   {
      h=f(x);
      if(y*h < 0)
      {
         printf("\n Se detectó cambio de signo en la vuelta %d ",vuelta);
         printf("\n F( %lf ) = %lf ", x, yy);
         if(h <=tol)
         {
            printf("\n La raiz aproximada es %lf ",x);
            getch();
            return;
         }
         x = x - inc;
         inc = inc/10;
         y = f(x);
      }
   }
   printf("\n No converge en %lf iteraciones ",nmi);
   getch();
}
double f(double x)    /*  se define la función    */
{
   vuelta = vuelta +1;
   yy= pow(x,3) - x - 1;
   return(yy);
}
```


El siguiente programa busca una raíz real mediante el método de bisección; el proceso de bisectar el intervalo se repite hasta acorralar la raíz en un entorno e intentar alcanzar tolerancia prefijada para el valor de la raíz en un número dado de iteraciones.

```
#include <math.h>
#include <stdio.h>
#include <conio.h>
double f(double y);
double yy, aux;
double h=0,tol=0.00001;
int vuelta=0;

/* empieza la función principal  */

main()
{
   double x, xm, xn, nmi, inc, x1, x2, y=0;
   int i;
   do
   {
      printf("\n Dame intervalo para tabular e incremento");
      printf("\n Dame valor inicial x1 ");
      scanf("%lf",&x1);
      printf("\n Dame valor final x2 ");
      scanf("%lf",&x2);
      printf("\n Dame nmi ");
      scanf("%lf",&nmi);
      printf("\n Dame tol ");
      scanf("%lf",&tol);
      if(x1 > x2 || nmi == 0 || tol ==0) printf("\n\n Algún dato no es válido \n ");
   }
   while(x1 > x2 || nmi == 0 || tol == 0);
   xm = (x2 - x1)/2.0;
   for(i=1; i<=nmi; i++)
   {
      y = f(xm);
      h = f(x1);
      if(y*h > 0) x1 = xm;
      else x2 = xm;
      xn = (x1 + x2)/2.0;
      if(fabs(xn - xm)/xn <= tol)
      {
         printf("\n La raiz aproximada es %lf ",xn);
         printf("\n Se obtuvo en %d iteraciones ",vuelta);
         getch();
         return;
      }
      xm = xn;
   }
   printf("\n No converge en %lf iteraciones ",nmi);
   getch();
}
double f(double x)
{
   vuelta = vuelta +1;
   yy= pow(x,3) - x - 1;
   return(yy);
}
```

El siguiente programa busca una raíz real mediante el método de punto fijo; el proceso es sumamente inestable, sin embargo en ocasiones se obtiene la raíz real que se busca.

```
#include <math.h>
#include <stdio.h>
#include <conio.h>
double f(double y);
double yy, yd, aux;
double h=0,tol=0.00001;
int vuelta=0;

/* empieza la función principal */

main()
{
    double x, xm, xn, nmi, inc, x1, x2;
    int i;
    do
    {
        printf("\n Dame valor inicial de la raiz (distinto de cero) ");
        scanf("%lf",&x1);
        printf("\n Dame número máximo de vueltas ");
        scanf("%lf",&nmi);
        printf("\n Dame tolerancia ");
        scanf("%lf",&tol);
        if(x1 ==0 || nmi == 0 || tol == 0) printf("\n\n Algún dato no es válido \n ");
    }
    while(x1 == 0 || nmi == 0 || tol == 0);
    for(i=1; i<=nmi; i++)
    {
        x2 = f(x1) + x1;
        if(fabs((x2 - x1)/x2) <= tol)
        {
            printf("\n La raiz aproximada es %lf ",x2);
            printf("\n Se obtuvo en %d iteraciones ",vuelta);
            getch();
            return;
        }
        x1 = x2;
    }
    printf("\n No converge en %lf iteraciones ",nmi);
    getch();
}
double f(double x)
{
    vuelta = vuelta +1;
    yy= 5.0*exp(x)*sin(x) - 0.75*x;
    return(yy);
}
```


El siguiente programa busca una raiz real mediante el método de Newton Raphson, viajando por las tangentes en la búsqueda de una mejor raíz real de la ecuación; dicho procedimiento se repite hasta acorralar la raíz en un entorno e intentar alcanzar tolerancia prefijada para el valor de la raíz con un número dado de iteraciones.

```
#include <math.h>
#include <stdio.h>
#include <conio.h>
double f(double y);
double fd(double y);
double yy, yd, aux;
double h=0,tol=0.00001;
int vuelta=0;

/* empieza la función principal */

main()
{
    double x, xm, xn, nmi, inc, x1, x2;
    int i;
    do
    {
        printf("\n Dame valor inicial de la raiz (distinto de cero) ");
        scanf("%lf",&x1);
        printf("\n Dame número máximo de vueltas ");
        scanf("%lf",&nmi);
        printf("\n Dame tolerancia ");
        scanf("%lf",&tol);
        if(x1 ==0 || nmi ==0 || tol == 0) printf("\n\n Algún dato no es válido \n ");
    }
    while(x1 == 0 || nmi ==0|| tol == 0);
    for(i=1; i<=nmi; i++)
    {
        x2 = x1 - f(x1)/fd(x1);
        if(fabs((x2 - x1)/x2) <= tol)
        {
            printf("\n La raiz aproximada es %lf ",x1);
            printf("\n Se obtuvo en %d iteraciones ",vuelta);
            getch();
            return;
        }
        x1 = x2;
    }
    printf("\n No converge en %lf iteraciones ",nmi);
    getch();
}
double f(double x)
{
    vuelta = vuelta +1;
    yy= pow(x,3) - x - 1;
    return(yy);
}
double fd(double x)
{
    yd= 3.0*pow(x,2) - 1;
    return(yd);
}
```

El siguiente programa resuelve un sistema de ecuaciones lineales por el método de eliminación Gaussiana:

```
#include <math.h>
#include <stdio.h>
main()
{
    int i,id,im1,j,jm1,k,km1,n;
    double may1, may2, sum, aux, elempiv, a[25][26], x[25];
    clrscr();
    printf("\nProporciona el número de incógnitas "); scanf("%d",&n);
    printf("\nProporciona los coeficientes de la matriz ampliada del sistema por renglones\n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n+1; j++)
        {
            printf(" Dame a(%d,%d)=",i,j); scanf("%lf",&a[i][j]);
        }
    }
    for(i=1; i<=n-1; i++)
    {
        id = i;  may1 = fabs(a[i][i]);  km1 = i + 1;
        for(j=km1; j<=n; j++)
        {
            may2 = fabs(a[j][i]);
            if((may1 < may2)) { may1 = may2;  id = j; }
        }
        if(id != i)
        {
            for(j=i; j <= n+1; j++)
            {
                aux = a[id][j];  a[id][j] = a[i][j];  a[i][j] = aux;
            }
    }
    for(j=km1; j<= n; j++)
    {
        elempiv = a[j][i]/a[i][i];
        for(k=km1; k<=n+1; k++)
        {
            a[j][k] = a[j][k] - elempiv * a[i][k];
        }
    }
    for(j=km1; j <= n; j++) a[j][i] = 0;
    }
    x[n] = a[n][n+1]/a[n][n];
    for(i=1; i <= n-1; i++)
    {
    sum = 0.0;   j = n - i;   jm1 = j +1;
    for(k=jm1; k<=n; k++) sum = sum + a[j][k]*x[k];
    x[j] = (a[j][n+1] - sum )/a[j][j];
}
printf("\n Vector solución ");
for(i=1; i<=n; i++)
{
 printf("\n x[%d] %lf", i,x[i]);
}
printf("\n Hasta la próxima");
getch();
return;
}
```


El siguiente programa resuelve un sistema de ecuaciones lineales por el método de Gauss - Jordan

```
#include <math.h>
#include <stdio.h>
main()          /* solución de un sistema de ecuaciones con el método de Gauss - Jordan */
{
    int i,id,im1,j,jm1,k,km1,n,m;
    double may1, may2, sum, aux, elempiv, a[25][26], x[25], b[25][26];
    clrscr();
    printf("\nProporciona el número de incógnitas "); scanf("%d",&n);
    printf("\nProporciona los coeficientes de la matriz ampliada del sistema por renglones\n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n+1; j++)
        {
            printf(" Dame a(%d,%d)=",i,j); scanf("%lf",&a[i][j]);
        }
    }
    m = n+1;
    do
    {
        if(a[1][1] == 0)
        {
            k = m-1;
            for(i=2; i<= k; i++)
            {
                if(a[i][1] != 0)
                {
                    for(j=1; j<=m; j++)
                    {
                        aux = a[i][j];
                        a[i][j] = a[1][j];
                        a[1][j] = aux;
                    }
                }
                goto sigue;
            }
            printf("\n No hay solución de tipo única "); getch(); return;
        }
        else
        {
            sigue:
            for(i=2; j<=m; j++)
            {
                for(i=2; i<= n; i++) b[i-1][j-1] = a[i][j] - a[1][j]*a[i][1]/a[1][1];
            }
            for(j=2; j<=m; j++)
            {
                b[n][j-1] = a[1][j]/a[1][1];
            }
            m = m -1;
            for(j=1; j<=m; j++)
            {
                for(i=1; i<=n; i++)    a[i][j] = b[i][j];
            }
        }
    }
    while(m>1);
    printf("\n Vector solución del sistema \n");
    for(i=1; i<=n; i++) printf("\n x(%d) = %f ",i, a[i][1]);
    getch(); return;
}
```

```
#include <math.h>
#include <stdio.h>
double f(double x); /* declaración de la función para tabular */
main()              /* inicia la función principal que integra con el método de Simpson 3/8 */
{
   double z, sim3, sro, integral=0, x, inicio, fin, incremento, y[1000];
   int franjas, k;
   do
   {
      printf("\n Dame intervalo para tabular y número de franjas múltiplo de tres");
      printf("\n Dame inicio ");
      scanf("%lf",&inicio);
      printf("\n Dame final ");
      scanf("%lf",&fin);
      printf("\n Dame número de franjas (sug. mayor de tres) ");
      scanf("%d",&franjas);
      if((fin <= inicio) || (franjas < 3) || (franjas % 3 != 0)) printf("\n\n Algún dato es erróneo ");
   }
   while((fin <= inicio) || (franjas < 3) || (franjas % 3 != 0));
   incremento = fabs(fin - inicio)/franjas;
   k = 0;
   for(x = inicio; x <= fin; x = x + incremento)
   {
      y[k] = f( x );
      printf("\n punto num  %3d    x = %6.2lf,    f(x) = %lf ",k,x,y[k]);
      k++;
   }
   sim3 = 0;
   for(x=3; x <= k-2; x+=3)
   {
      sim3 = sim3 + y[x];
    }
   sro = 0;
   for(x=1; x <= k-2; x+=3)
   {
      sro = sro + y[x] + y[x+1];
   }
/* se integra la función con la fórmula

   integral = (3*incremento_de_x/8)( y[0] + y[n] + 2( Σ ords indice mult. tres) + 3( Σ resto ords ) }
*/
   integral = 3.0*incremento/8.0*(y[0] + y[k-1] + 2*sim3 + 3*sro);
   printf("\n integral = %lf ", integral);
   getch();
}
double f(double x)    /* se define la función que se integrará */
{
   double y;
  /* en este ejemplo se usó un polinomio de grado 3 */
   y= pow(x,3) - x - 1.0;
   return y;
}
```


Los tres siguientes programas definen una ecuación cúbica que se tabula en un intervalo dado y se integra por los métodos trapecial, Simpson del 1/3 y Simpson 3/8.

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

double f(double x); /* declaración de la función para tabular */

main()              /* empieza la función principal para integrar con el método trapecial */
{
   double z, sum, integral=0;
   double x, inicio, fin, incremento;
   double y[1000];
   int franjas;
   int k;
   do
   {
      printf("\n Dame intervalo para tabular y número de franjas ");
      printf("\n Dame inicio ");
      scanf("%lf",&inicio);
      printf("\n Dame final ");
      scanf("%lf",&fin);
      printf("\n Dame número de franjas ");
      scanf("%d",&franjas);
      if((fin <= inicio) || (franjas < 2)) printf("\n\n Algún dato es erroneo ");
   }
   while((fin <= inicio) || (franjas < 2));
   incremento = fabs(fin - inicio)/franjas;
   k = 0;
   for(x = inicio; x <= fin; x = x + incremento)
   {
      y[k] = f( x );
      printf("\n punto num  %3d    x = %6.2lf,    f(x) = %lf ",k,x,y[k]);
      k++;
   }
   sum = 0;
   for(x=1; x <= k-2; x++)
   {
      sum = sum + y[x];
   }
/* se integra la función con la fórmula

   integral = incremento_de_x/2( y[0] + y[n] + 2( Σ resto de ordenadas))
*/
   integral = incremento/2.0*(y[0] + y[k-1] + 2*sum);
   printf("\n integral = %lf ", integral);
   getch();
}

double f(double x)    /* se define la función que se integrará */
{
   double y;
  /* en este ejemplo se usó un polinomio de grado 3 */
   y= pow(x,3) - x - 1.0;
   return y;
}
```

```
#include <math.h>
#include <stdio.h>
double f(double x);  /*  declaración de  la función para tabular   */
main()               /* inicia la función principal que integra con el método de Simpson del 1/3  */
{
    double z, sii, sip, integral=0, x, inicio, fin, incremento, y[1000];
    int franjas, k;
    do
    {
        printf("\n Dame intervalo para tabular y número par de franjas ");
        printf("\n Dame inicio ");
        scanf("%lf",&inicio);
        printf("\n Dame final ");
        scanf("%lf",&fin);
        printf("\n Dame número de franjas ");
        scanf("%d",&franjas);
        if((fin <= inicio) || (franjas < 2) || (franjas % 2 != 0)) printf("\n\n  Algún dato es erróneo ");
    }
    while((fin  <= inicio) || (franjas < 2) || (franjas % 2 != 0));
    incremento = fabs(fin - inicio)/franjas;
    k = 0;
    for(x = inicio; x <= fin; x = x + incremento)
    {
        y[k] = f( x );
        printf("\n  punto num  %3d    x = %6.2lf,     f(x) = %lf ",k,x,y[k]);
        k++;
    }
    sii = 0;
    for(x=1; x <= k-2; x+=2)
    {
        sii = sii + y[x];
    }
    sip = 0;
    for(x=2; x <= k-2; x+=2)
    {
        sip = sip + y[x];
    }
/* se integra la función con la fórmula

    integral = incremento_de_x/3{ y[0] + y[n] + 2( Σ ords índice par) + 4( Σ ords indice impar) }
*/
    integral = incremento/3.0*(y[0]  + y[k-1] + 2*sip + 4*sii);
    printf("\n integral = %lf ", integral);
    getch();
}
double f(double x)    /* se define la función que se integrará  */
{
    double y;
  /* en este ejemplo se usó un polinomio de grado 3 */
    y=  pow(x,3) - x - 1.0;
    return y;
}
```


El siguiente programa permite interpolar por el método de Lagrange:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
main()
{
    int i, j, n;
    float d, x[101], y[101], xinterpol, yinterpolada=0;
    float prodsnum, prodsden;
    clrscr();

 /* programa para interpolar por el método de lagrange "n coordenadas" */

    printf("\n Interpolación por el método de Lagrange ");
    printf("\n\n  Cuantos puntos (n <= 100) ");
    scanf("%i",&n);

    printf("\n Dame los valores (x,y) separados por una coma\n");

    for(i = 1; i <= n; i++)
    {
        printf("\n\t  punto x(%3d), y(%3d) = ?  ",i,i);
        scanf("%f,%f",  &x[i], &y[i]);
    }
    do
    {
        printf("\n Para cual valor de x interpolas ");
        scanf("%f",&xinterpol);
    }
    while(xinterpol<  x[1] || xinterpol > x[n]);
    for(i=1;  i<=n; i++)
    {
        prodsnum = 1.0;
        prodsden = 1.0;
        for(j=1;  j<=n; j++)
        {
            if(i != j)
            {
                prodsnum = prodsnum*(xinterpol  - x[j]);
                prodsden = prodsden*(x[i]  - x[j]);
            }
        }
        yinterpolada = yinterpolada + (prodsnum/prodsden)*y[i];
    }

    printf("\n La ' y ' interpolada = %f ",yinterpolada);

    printf("\n\n\t  Fin de proceso. Hasta la vista.... ");
    getch();
}
```

El siguiente programa permite ajustar una recta por el método de mínimos cuadrados:

```
#include <conio.h>
#include <stdio.h>
main()
{
    int i, j, n;
    float d, x[101], y[101], a0,a1, sx=0, sy=0, sxy=0, sx2=0;
    clrscr();
/* programa para ajustar una recta a un conjunto de puntos dados 'n' coordenadas"
```

$$y = a0 + a1*X$$

$$a0 = \frac{\sum_{i=1}^{n} X_i^2 \sum_{i=1}^{n} Y_i - \sum_{i=1}^{n} X_i \sum_{i=1}^{n} X_i Y_i}{n\sum_{i=1}^{n} X_i^2 - (\sum_{i=1}^{n} X_i)^2}$$

$$a1 = \frac{n\sum_{i=1}^{n} X_i Y_i - \sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i}{n\sum_{i=1}^{n} X_i^2 - (\sum_{i=1}^{n} X_i)^2}$$

```
*/
    printf("\n Ajuste por el método de mínimos cuadrados ");
    printf("\n\n Cuantos puntos (n <= 100) ");
    scanf("%i",&n);
    printf("\n Dame los valores (x,y) separados por una coma\n");
    for(i = 1; i <= n; i++)
    {
        printf("\n\t punto x(%3d), y(%3d) = ? ",i,i);
        scanf("%f,%f", &x[i], &y[i]);
    }
    printf("\n La mejor recta para el conjunto de puntos es\n");
    for(i=1; i<=n; i++)
    {
        sx = sx + x[i];
        sy = sy + y[i];
        sx2 = sx2 + x[i]*x[i];
        sxy = sxy + x[i]*y[i];
    }
    a0 = (sx2*sy - sx*sxy)/(n*sx2 - sx*sx);
    a1 = (n*sxy - sx*sy)/(n*sx2 - sx*sx);

    printf("\n          y = %f + %f(x)  ", a0, a1);

    printf("\n\n\t Fin de proceso. Hasta la vista...  ");
    getch();
}
```


APÉNDICE 2

INTRODUCCIÓN AL AMBIENTE GRÁFICO

DEL LENGUAJE C

INTRODUCCIÓN AL AMBIENTE GRÁFICO DEL LENGUAJE C

La modalidad gráfica del lenguaje C permite una gran variedad de posibilidades que van de figuras geométricas a diagramas de barras de dos y tres dimensiones, gráficas de pastel, tipos de letras y tamaños, etc.; de entre todas las posibilidades se hará una introducción a:

- Gráficas de figuras geométricas como líneas, rectángulos, círculos, elipses, diagramas de barras tridimensionales, diagramas de pastel circular y elíptico.
- Gráficas cartesianas de funciones con un tamaño predeterminado.

PARÁMETROS BÁSICOS

Debido a que la pantalla tiene ciertos parámetros físicos, para graficar en ella es necesario conocer los parámetros que la configuran y, cómo se pueden utilizar dichos parámetros para que cumplan nuestros requerimientos (principalmente para ajustar las dimensiones o tamaño necesario de las gráficas para poder visualizarlas en la pantalla).

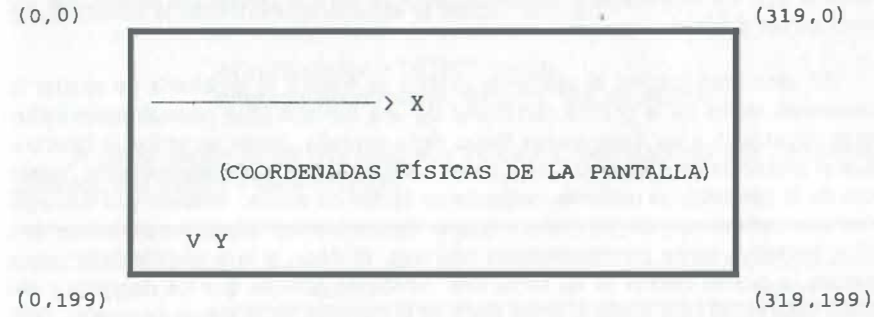
De esta manera, para utilizar el monitor a color de una computadora dentro del ambiente gráfico de C se activa:

- Su controlador, el cual corresponde al el tipo de monitor: CGA, EGA, VGA, etc.
- Su resolución o MODO que normalmente se identifica tanto por nombre como por su equivalente numérico, por ejemplo: CGAC0, EGALO, VGALO, VGAMED, VGAHI, etc.

DESCRIPCIÓN FÍSICA DE LA PANTALLA

El parámetro más importante de todo monitor es la resolución de la pantalla, que se refiere al número máximo de puntos (pixeles) que pueden activarse (encenderse o apagarse); de acuerdo con el número de pixeles se manejan una gran variedad de tipos de resolución como por ejemplo:

1. Para gráficas en resolución media y en color (320 x 200 puntos para monitores CGA, EGA, y VGA con sus variantes):

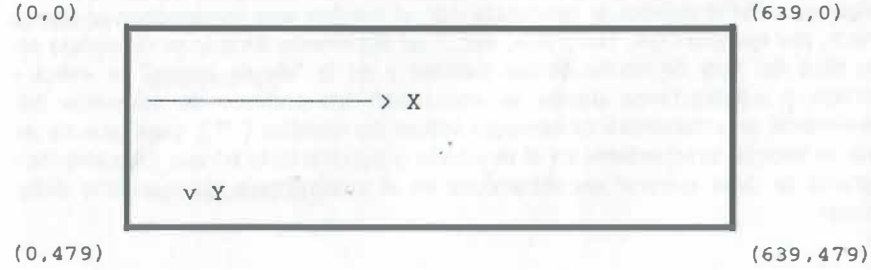


Horizontalmente a partir del extremo superior izquierdo, tenemos desde la posición 0 (cero) hasta la posición 319, y verticalmente desde el extremo superior izquierdo hasta la parte baja de la pantalla, se tiene desde la posición 0 hasta la 199, es decir, los puntos físicamente se definen considerando el eje vertical en forma invertida (no de forma cartesiana).

De esa manera la pantalla está cubierta por una malla fina de puntos o retícula en la que es posible identificar cada uno de ellos por la posición dada por sus coordenadas (X,Y).

Cuando se ilumina o prende un conjunto ordenado de puntos, se obtiene una gráfica, por ejemplo una línea, un círculo, un rectángulo, etc.

2. Para gráficas en alta resolución a color (640 x 480 puntos para monitores EGA, VGA y sus variantes):



De forma similar al caso anterior, horizontalmente a partir del extremo superior izquierdo, se tiene desde la posición 0 hasta la posición 639, y verticalmente desde el extremo superior izquierdo hasta la parte baja de la pantalla se tiene desde la posición 0 hasta la 479. De esta manera horizontalmente se tiene la posibilidad de emplear más puntos en las gráficas.

En esta introducción al ambiente gráfico se tratará el problema de ajustar las dimensiones reales de la gráfica cartesiana de una función (que normalmente incluye valores negativos), a las dimensiones físicas de la pantalla, desde un enfoque básico en el cual el problema de utilizar exclusivamente valores positivos predeterminados (valores físicos de la pantalla), se resuelve mediante un factor de escala, rotación y la traslación de los ejes cartesianos, de tal manera que se determina los valores equivalentes de la gráfica, exclusivamente con magnitudes positivas, es decir, si la gráfica incluye valores negativos, a dichos valores se les suma una constante positiva que los desplaza y solo permite valores del cero hasta el límite físico de la pantalla en términos de pixeles, tanto horizontalmente como verticalmente. Todo ello ajustado mediante un factor de escala que permite establecer de forma biunívoca los valores reales con el número de puntos de la pantalla.

ACTIVACIÓN DEL CONTROLADOR GRÁFICO

Para emplear las funciones gráficas se debe preferentemente utilizar computadora con monitor a color; indicar en los prototipos de funciones la directiva #include <graphics.h>, y dentro del programa fuente agregar lo siguiente:

```
int controlador, modo;

initgraph(&controlador, &modo, "vía_de_acceso");
```

Para que la función initgraph(); cargue el controlador de gráficos en memoria, el argumento &controlador se reemplaza con el nombre que corresponde al tipo de monitor, por ejemplo CGA, EGA, VGA, etc, ..., el argumento &modo se reemplaza con el nombre del tipo de resolución del monitor y en la "vía_de_acceso" se indica el directorio y subdirectorios donde se encuentren los archivos de extensión BGI; normalmente se acostumbra únicamente indicar las comillas (""), para que de esa forma se busque directamente en el directorio o subdirectorio en uso (obviamente el programa se debe ejecutar encontrándose en el subdirectorio que contiene dichos archivos).

La alternativa que simplifica lo anterior consiste en indicar para el controlador la macro DETECT que automáticamente identifica o detecta el controlador que corresponde a nuestro equipo y selecciona la mayor resolución posible en el video, estableciendo el valor correspondiente al modo;, esto es:

```
int controlador = DETECT, modo;
initgraph(&controlador, &modo, "vía_de_acceso");
```

A continuación se da una tabla que indica los controladores más comúnmente utilizados, sus modos y valores típicos:

controlador	Modo	Valor equivalente	resolución
CGA	CGAC0	0	320 X 200
	CGAC1	1	320 X 200
	CGAC2	2	320 X 200
	CGAC3	3	320 X 200
	CGAHI	4	640 X 200
EGA	EGALO	0	640 X 200
	EGAHI	1	640 X 350
VGA	VGALO	0	640 X 200
	VGAMED	1	640 X 350
	GVGAHI	2	640 X 480

En algunas versiones de C, como la 2.0, es necesario activar la modalidad gráfica oprimiendo la tecla ALT y sin soltar la letra O; del submenú que aparece, la opción Linker que al seleccionarla nos permite colocar en ON la opción Graphics library.

Al final del programa para restablecer el modo texto debe emplearse alguna de las funciones:

```
closegraph();

o

restorecrtmode();
```

ASIGNACIÓN DE COLOR PARA EL FONDO, GRÁFICAS Y TEXTOS

El tipo de adaptador gráfico determina la cantidad de colores que pueden utilizarse. Consideraremos en estas notas los monitores más comunes para nuestro medio, es decir, los monitores CGA/VGA y/o EGA/VGA, que manejan 16 colores.

En la modalidad gráfica se puede asignar color de fondo y color a las gráficas. Las funciones más comunes que permiten fijar los colores son:

```
setbkcolor(COLOR);      /* se asigna color de fondo   */

setcolor(COLOR);        /* se asigna color de líneas  y figuras geométricas */
```

A continuación se muestra la tabla de los colores que se emplean en CGA/VGA:

Color aproximado	Nombre_de_macro	Valor
Negro	BLACK	0
Azul marino claro	BLUE	1
Verde perico	GREEN	2
Verde claro	CYAN	3
Rojo	RED	4
Morado o violeta	MAGENTA	5
Café ligero	BROWN	6
Gris claro	LIGHTGRAY	7
Gris semiobscuro	DARKGRAY	8
Azul marino claro	LIGHTBLUE	9
Verde claro	LIGHTGREEN	10
Azul claro	LIGHTCYAN	11
Rojo claro	LIGHTRED	12
Rosa mexicano	LIGHTMAGENTA	13
Amarillo	YELLOW	14
Blanco	WHITE	15

La alternativa para dar color por planos es con la función:

```
setpalette(capa_seleccionada, macro_color);
```

La tabla de colores para los monitores EGA/VGA es la siguiente:

color	nombre_de_macro	valor
Negro	EGA_BLACK	0
Azul marino claro	EGA_BLUE	1
Verde perico	EGA_GREEN	2
Verde claro	EGA_CCYAN	3
Rojo	EGA_RED	4
Morado o violeta	EGA_MAGENTA	5
Gris claro	EGA_LIGHTGRAY	7
Café ligero	EGA_BROWN	20
Gris semiobscuro	EGA_DARKGRAY	56
Azul marino claro	EGA_LIGHTBLUE	57
Verde claro	EGA_LIGHTGREEN	58
Azul claro	EGA_LIGHTCYAN	59
Rojo claro	EGA_LIGHTRED	60
Rosa mexicano	EGA_LIGHTMAGENTA	61
Amarillo	EGA_YELLOW	62
Blanco	EGA_WHITE	63

En VGA se definen cuatro capas pero normalmente se usan las capas 0 para el fondo y 1 para el color de los contornos de las figuras gráficas.

DESPLIEGUE DE MENSAJES EN MODO GRÁFICO

Los textos que acompañan a las gráficas se escriben en forma estándar, utilizando 25 renglones por 80 columnas. Aunque a veces es posible usar la función printf(); para desplegar texto, se recomienda sólo emplear las funciones específicas para escribir dentro del modo gráfico, mismas que se indican a continuación:

```
outtextxy(corrdenada_x, coordenada_y, "texto_a_desplegar");
```

outtext("texto a desplegar");

La primera de las funciones despliega el mensaje en las coordenadas físicas de la pantalla que se indican; la segunda de ellas despliega los mensajes a partir de el extremo superior izquierdo. Además, los mensajes también pueden ubicarse en posiciones específicas mediante las funciones:

moveto(corrdenada_x, coordenada_y);

settextjustify(justificación_horizontal, justificación_vertical);

La justificación, tanto horizontal como vertical se realiza mediante:

LEFT_TEXT,
CENTER_TEXT,
RIGHT_TEXT,
BOTTOM_TEXT,
TOP_TEXT.

Por último el tipo de letra, sentido y tamaño de la letra se puede modificar con:

settextstyle(tipo_de_letra, dirección_del_despliegue, tamaño);

Los mensajes pueden escribirse tanto horizontalmente (de izquierda a derecha) como verticalmente (de la parte inferior de la pantalla hacia arriba o de abajo hacia arriba), mediante las funciones:

HORIZ_DIR
VERT_DIR

Los tipos de letras se definen mediante:

DEFAULT_FONT,
TRIPLEX_FONT,
SMALL_FONT,
SANS_SERIF_FONT,
GOTHIC_FONT.

Como ejemplo de despliegue de texto en modo gráfico, considere el siguiente programa cuya salida son mensajes en modo gráfico:


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

void letra(void)
{
    int x;
    settextjustify( CENTER_TEXT, TOP_TEXT );
    moveto( 2, 3 );
    setbkcolor(WHITE);
    setcolor(RED);
    settextstyle( GOTHIC_FONT, VERT_DIR, 7 );
    moveto(20,5);
    outtext( "Letra Gótica");
    settextstyle( SMALL_FONT, VERT_DIR, 7 );
    outtextxy(500,400," Prueba numero 2");
    getch();
}
void main(void)
{
    int controlador, modo,color;
    controlador = DETECT;
    initgraph(&controlador, &modo, "");
    letra();
}
```

INSTRUCCIONES BÁSICAS PARA DIBUJAR EN LA PANTALLA

Algunas de las funciones más empleadas en el ambiente gráfico son las siguientes:

putpixel(x, y, color); que ilumina o activa el punto que corresponde a la coordenada (x,y) con el color indicado.

line(x1 ,y1, x2, y2); que traza una línea a partir del punto (x1, y1) al punto (x2, y2).

rectangle(x1 ,y1, x2, y2); construye un rectángulo o recuadro a partir del punto (x1, y1) y con extremo opuesto en el punto (x2, y2).

bar(x1 ,y1, x2, y2); define un área rectangular a partir del punto (x1, y1) y con extremo opuesto en el punto (x2, y2). Dicha área puede achurarse de diversas formas.

circle(x ,y, radio); dibuja un círculo con centro en (x, y) y radio indicado.

ellipse(x ,y, ángulo_inicial, ángulo_final, radio_x, radio_y);

que traza un sector elíptico con centro en las coordenadas (x,y), a partir de los valores dados en grados para ángulo_inicial y ángulo_final, con los radios horizontal y vertical indicados.

fillellipse(x ,y, radio_x, radio_y);

que dibuja y rellena con algún achurado en uso, una elipse con centro en las coordenadas (x,y) y los radios horizontal y vertical indicados.

bar3d(xi, yi, xd, yd, profundidad, bandera_recorte);

que traza una barra tridimensional cuya área frontal está definida por el rectángulo determinado del punto izquierdo (xi,yi) al punto derecho (xd,yd), mismo que se proyecta con la profundidad asignada; la bandera recorte evita que la barra sobrepase el puerto gráfico.

pieslice(x ,y, ángulo_inicial, ángulo_final, radio);

que traza y rellena con la trama en uso un pedazo de pastel o sector circular con centro en las coordenadas (x,y) y el radio indicado.

sector(x ,y, ángulo_inicial, ángulo_final, radio_x, radio_y);

que dibuja y rellena con la trama en uso un sector elíptico con centro en las coordenadas (x,y) y los radios horizontal y vertical indicados.

El dibujo de la línea, barra de tres dimensiones, pedazo de pastel circular y sector elíptico puede realizarse con puntos o tres tipos de rayas (estilo); el grueso de la línea para los círculos y rectángulos puede indicarse normal o grueso (1 o 3) cuando antes se ha indicado alguna de esas características mediante la función:

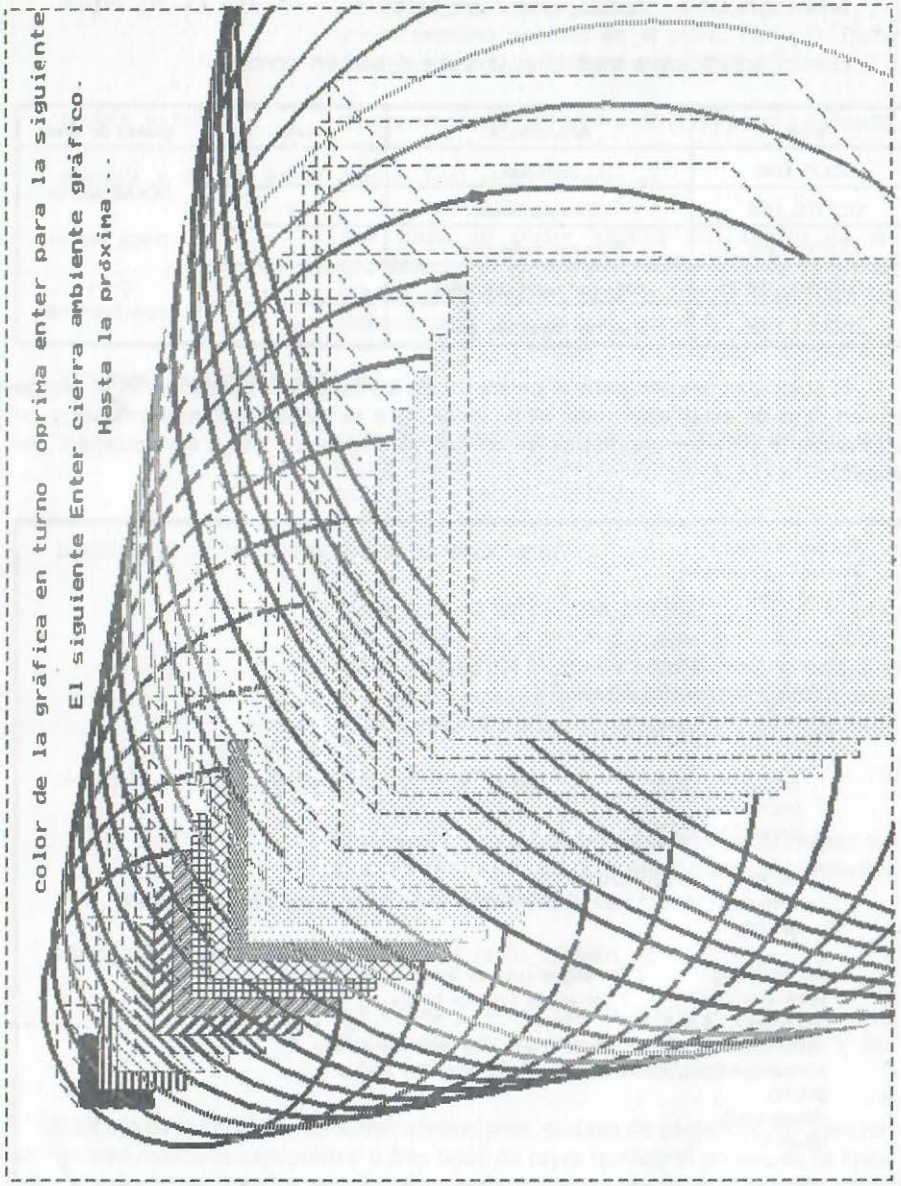
setlinestyle(estilo , modelo, grueso_de_línea);

El valor adjudicado a modelo es un valor entero sin signo.

estilo	APARIENCIA	valor	grueso de línea
SOLID_LINE	contínua	0	NORM_WIDTH O THICK_WIDTH 0 y 1 respectivamente
DOTTED_LINE	línea punteada	1	
CENTER_LINE	línea con rayas finas	2	
DASHED_LINE	línea con rayas pequeñas	3	
USERBIT_LINE	línea con rayas tipo guión	4	

El color por omisión para el perímetro de las funciones anteriores es el blanco pero en caso de establecer antes algún color, éste se asigna automáticamente y así sucesivamente para los perímetros de las figuras que se dibujen a continuación. Por ejemplo:

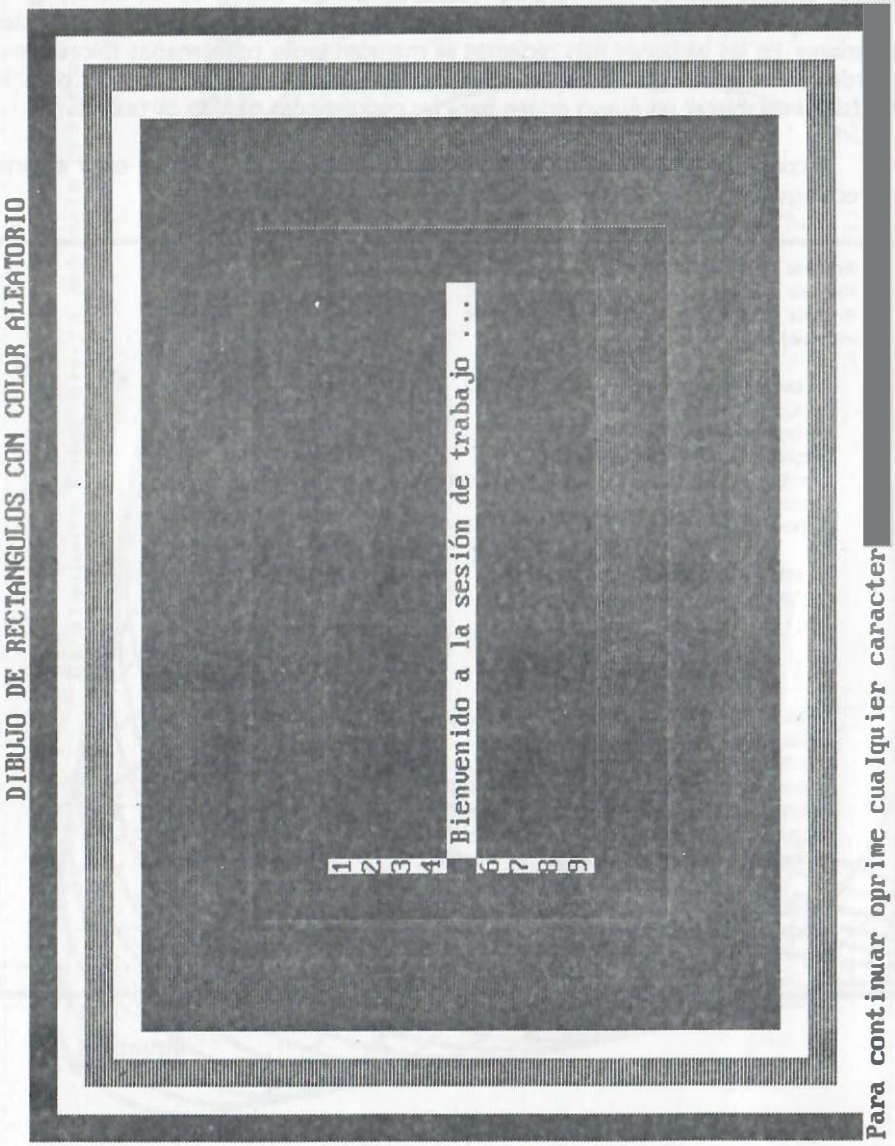
```
#include <stdio.h>
#include <graphics.h>
int main(void)
{
   int controlador, modo, i, ii;
   controlador = DETECT;
   initgraph(&controlador, &modo, "");
   setcolor(GREEN);
   for(i=4, ii=0; i<=25; i++, ii++)
   {
      setlinestyle(3,0,3);
      setcolor(ii);
      circle(25*i,25*i,20*i);

      bar3d(10*i,10*i,20*i,20*i,4*i,1);
      outtextxy(150,15," color de la gráfica en turno  oprima enter para la siguiente");
      getch();
   }
   setcolor(RED);          /* se asigna color de líneas  y rectángulo */
   setbkcolor(WHITE);      /* se asigna color de fondo   */
   rectangle(1,1,638,478);
   outtextxy(250,35," El siguiente Enter cierra ambiente gráfico...");
   outtextxy(400,50," Hasta la próxima...");
   getch();
   closegraph();
   return 0;
}
```



En la versión 2.00 de C, todas las coordenadas deben especificarse mediante valores enteros que permitan graficar mediante puntos dentro de los límites de la resolución de la pantalla, asimismo el valor del color será según los valores de las tablas anteriores. En las versiones más recientes se manejan tanto coordenadas físicas como coordenadas lógicas que permiten manipular un tamaño predeterminado para la pantalla y establecer un nuevo origen para las coordenadas de tipo cartesiano.

A continuación se muestra un programa que genera recuadros en color a partir de rectángulos; dentro de ellos se genera una ventana de texto.

```
#include <graphics.h>
#include <stdlib.h>
#include <time>
int main(void)
{
   int controlador, modo;
   int x, xx, y, yy, i;
   controlador = DETECT;
   initgraph(&controlador, &modo, "");
   printf("\t\t\t\t DIBUJO DE RECTÁNGULOS CON COLOR ALEATORIO");
   randomize();
   for(xx=15, yy=479; xx<=135; xx+=15,yy-=15)
   {
      setcolor(random(7));
      for(x=xx,y=yy; x<=350; x+=1, y-=1)
      {
          rectangle(x,xx,y+150,yy);
      }
   }
   window(21,12,61,20);  /*  se abre ventana para texto (9 renglones) */
   textbackground(0);    clrscr();
   printf("1");          /*  se posiciona texto en la ventana  */
   gotoxy(1,2); printf("2");     gotoxy(1,3); printf("3");
   gotoxy(1,4); printf("4");     gotoxy(2,5); printf(" Bienvenido a la sesión de trabajo ... ");
   gotoxy(1,6); printf("6");     gotoxy(1,7); printf("7");
   gotoxy(1,8); printf("8");     gotoxy(1,9); printf("9");
         /*  texto fuera de la ventana       */
   printf("\n\n\n\n\n\n\n\n\n\n\n  Para continuar oprime cualquier caracter");
   getch();     restorecrtmode();
   return 0;
}
```



COLOR Y ACHURADO EN ÁREAS CERRADAS

Para colorear un área cerrada como primer paso se establece que se empleará línea contínua (SOLID_LINE o 1) en los trazos de figuras siguientes mediante la función:

```
setlinestyle(SOLID_LINE, 1, 1);
```

como segundo paso seleccione y establezca el color y tipo de achurado con la función:

```
setfillstyle(achurado, color);
```

como tercer paso se define el área o la figura geométrica cerrada (sin discontinuidades), por ejemplo un círculo:

```
circle(x, y, radio);
```

y por último se achura la figura mediante la función:

```
floodfill(x, y, color_perímetro);
```

en la que las coordenadas (x,y) deben corresponder a algún punto interno de la figura; el valor de color_perímetro puede ser 15 (WHITE) para algunas figuras como el círculo pues si el perímetro presenta discontinuidades, el color abarcará tanto dentro como fuera de la figura (se achura el área delimitada por líneas continuas del mismo color)..

TIPOS DE ACHURADOS

El área de las áreas cerradas y de las funciones bar(); fillelipse(); bar3d(); pieslice(); y sector(); puede achurarse de 12 formas básicas y 16 colores mediante la función:

```
setfillstyle(tipo_de_achurado, color);
```

La tabla de tipos de achurado es la siguiente:

tipo_de_achurado	APARIENCIA	valor
EMPTY_FILL	color de fondo	0
SOLID_FILL	color sólido	1
LINE_FILL	color en líneas	2
LTSLASH_FILL	línea inclinada tipo: (/)	3
SLASH_FILL	línea inclinada tipo: (/)	4

tipo_de_achurado	APARIENCIA	valor
BKSLASH_FILL	línea inclinada tipo: (\)	5
LTBKSLASH_FILL	línea inclinada tipo: (\)	6
HATCH_FILL	cruzamiento de líneas horizontales con verticales	7
XHATCH_FILL	líneas con cruzamiento en X	8
INTERLEAVE_FILL	líneas cruzadas tipo fino	9
WIDE_DOT_FILL	malla de puntos	10
CLOSE_DOT_FILL	malla de puntos densa y fina	11

Además se tiene la posibilidad para definir achurados propios del usuario y emplearlos mediante la función:

setfillpattern(achurado_definido_por_el_usuario, color);

El achurado definido por el usuario se define mediante un arreglo alfanumérico con valores en un total de 8 x 8 bits, mismos que se refieren apuntando a alguno de los achurados, según se ejemplifica en el programa de la página siguiente.

Los achurados se emplean frecuentemente para resaltar la apariencia de figuras trazadas con las funciones bar(); fillelipse(); bar3d(); pieslice(); y sector(); para las cuales sólo se requiere anteceder la función setfillstyle(achurado, color);

Cuando se requiere transformar valores enteros a valores alfanuméricos para desplegarlo en modo gráfico (valores de variables enteras), se emplea la función:

itoa(entero, representación_ASCCI,base);

en dicha función se puede especificar para base los valores 10 o 16 para que la representación de las cantidades enteras se visualice en base 10 o en base 16 respectivamente.

Las funciones que determinan el total de pixeles horizontales y verticalmente son:

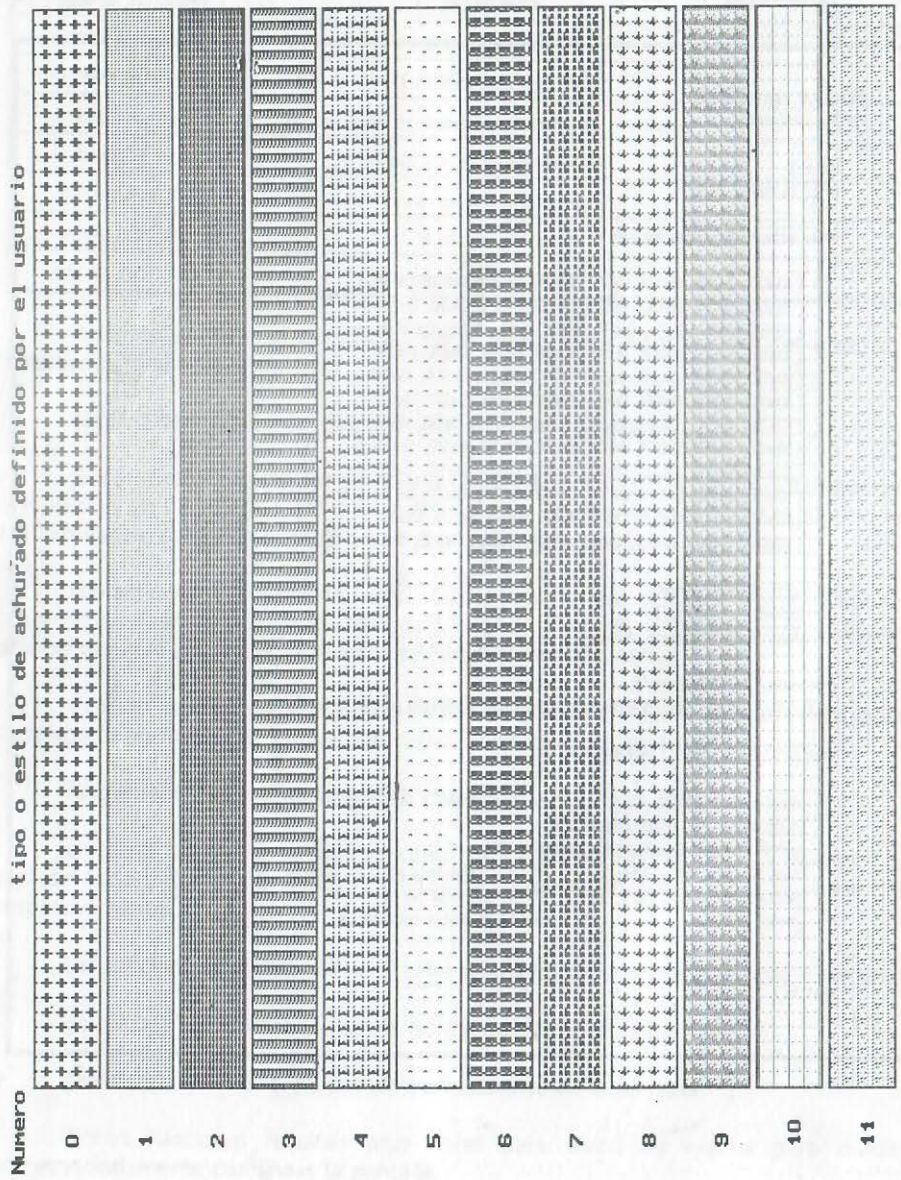
getmaxx(); /* total de puntos horizontalmente en la pantalla */

getmaxy(); /* total de puntos verticalmente en la pantalla */.

Dichas funciones resultan muy útiles para trazo de ejes o para dividir equiespaciadamente con líneas la pantalla.

Por ejemplo:

```
#include <graphics.h>
void main(void)
{
   int i, j;
   char numASCII[3];

   char achurado_usuario[][8] =
   {
       { 0x00, 0x10, 0x10, 0x7C, 0x7C, 0x10, 0x10, 0x00 },
       { 0x00, 0x55, 0x00, 0x55, 0x00, 0x55, 0x00, 0x55 },
       { 0xEE, 0xFF, 0xEE, 0xFF, 0xEE, 0xFF, 0xEE, 0xFF },
       { 0x88, 0x78, 0x88, 0x78, 0x88, 0x78, 0x88, 0x78 },
       { 0x00, 0x7A, 0x20, 0x30, 0x30, 0x20, 0x7A, 0x00 },
       { 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10 },
       { 0x20, 0x5C, 0x5C, 0x5C, 0x6C, 0x5C, 0x5C, 0x20 },
       { 0x00, 0x77, 0x77, 0x00, 0x7C, 0x77, 0x77, 0x00 },
       { 0x00, 0x10, 0x20, 0x30, 0x7C, 0x20, 0x10, 0x00 },
       { 0xFF, 0x7E, 0x7D, 0x7C, 0x78, 0x7A, 0x70, 0x79 },
       { 0x00, 0x10, 0x10, 0xFF, 0xFF, 0x10, 0x10, 0x00 },
       { 0x00, 0xAA, 0x21, 0x05, 0x05, 0x21, 0xAA, 0x00 }
   };
   int controlador, modo;
   controlador = DETECT;
   initgraph(&controlador, &modo, "");

   outtextxy(1,1,"Numero");
   outtextxy(160,1,"tipo o estilo de achurado definido por el usuario ");

   for( j=10,i=0; i<=11 ; j=j+33,i+=1 )
   {
       itoa( i, numASCII, 10 );     /* Se convierte i a valor ASCII  */
       outtextxy( 16, j+20, numASCII );
       setfillpattern( &achurado_usuario[i][0], i+1 );
       bar(50, 3+j, 635, 38+j );       /* Se dibuja recuadro achurado  */
       rectangle(50,3+j, 635, 38+j ); /* Se delimita su contorno con un rectángulo */
       j+=6;                          /*  Separación entre recuadros   */
   }

   getch();
}
```



Como ejèmplo completo de la teoría anterior sobre el manejo de figuras gráficas considere el siguiente programa:

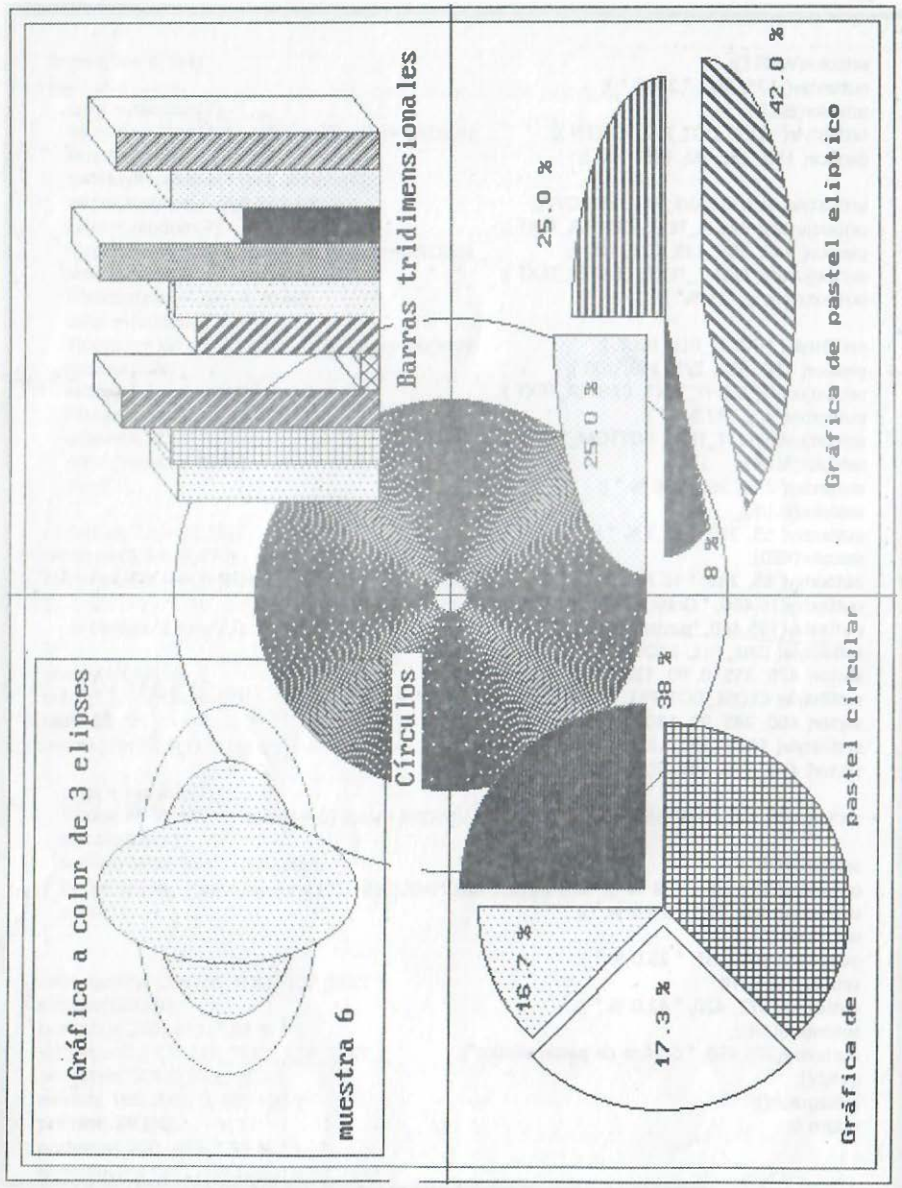
```
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
#include <time.h>

float x, y;
int main(void)
{
   int controlador, modo,color;
   int i, ejex,ejey,de,base;
   randomize();
   controlador = DETECT;
   initgraph(&controlador, &modo, "");

   setcolor(RED);          /* se asigna color de líneas  y rectángulo */
   setbkcolor(WHITE);      /* se asigna color de fondo   */

   rectangle(1,1,638,478);  /*  recuadro para la pantalla  */
   ejex = getmaxx();        /*  valores máximos de la pantalla  */
   ejey = getmaxy();
   circle(ejex/2,ejey/2,150);
   ejex = getmaxx();        /*  valo es máximos de la pantalla  */
   ejey = getmaxy();
   line(ejex/2,0,ejex/2,ejey);     /* trazo de ejes  */
   line(0,ejey/2,ejex,ejey/2);

   rectangle(15,15,ejex/2-35,ejey/2-35);     /* trazo de rectángulo1  */
   gotoxy(8,3);
   printf("Gráfica a color de 3 elipses");
   printf("\n\n\n\n\n\n\n\n\n\n\n\n");
   printf("                    ");
   printf("                    ");
   printf("          ");
   printf("Barras tridimensionales ");

for(i=1; i<= 6; i++)
{
  color = random(7);
  if(color == WHITE || color == 0) color= BROWN;
  setcolor(color);

 /*  continúa el programa en la siguiente página  */
```

función (circle) (1, 2, 3)

```
for(i=1; i<= 6; i++)
{
   color = random(7);
   if(color == WHITE || color == 0) color= BROWN;
   setcolor(color);
   setfillstyle(random(11)+1,random(6));
   ellipse(ejex/4,ejey/4,0,360,100,50);
   color = random(7);
   if(color == WHITE || color == 0) color= BROWN;
   setcolor(color);
   fillellipse(ejex/4,ejey/4,70,40);
   color = random(7);
   if(color == WHITE || color == 0) color= BROWN;
   setcolor(color);
   setfillstyle(random(11)+1,random(7));
   fillellipse(ejex/4,ejey/4,40,70);
   gotoxy(4,12);
   printf("muestra %d",i);
   sleep(1);
}
circle(ejex/2,ejey/2,50);
rectangle(9,9,630,470);
for(i=ejex/64; i <= (ejex/6); i+=1)
{
   circle(ejex/2,ejey/2,i);
}
gotoxy(19,14);
printf("          Circulos ");
base=20;
for(i=1; i<=10; i++)
{
   color = random(7);
   if(color == WHITE || color == 0) color= BROWN;
   setcolor(color);
   setfillstyle(random(11)+1,color);
   bar3d(370+(i-1)*base,random(175)+25,370+i*base,200,20,1);
   sleep(1);
}

settextjustify( CENTER_TEXT, TOP_TEXT );
setcolor(GREEN);
outtextxy( 200, 415, " 38 % " );
settextjustify( CENTER_TEXT, TOP_TEXT );
setfillstyle( SOLID_FILL, RED );
pieslice( 160, 345, 0, 90, 100 );
setcolor(GREEN);
outtextxy( 200, 415, " 38 % " );
settextjustify( LEFT_TEXT, BOTTOM_TEXT );
```



```
   setcolor(WHITE);
   outtextxy( 175, 300, " 25 % " );
   setcolor(BLUE);
   setfillstyle( WIDE_DOT_FILL, GREEN );
   pieslice( 150, 355, 90, 135, 100 );

   setfillstyle( INTERLEAVE_FILL, YELLOW );
   settextjustify( RIGHT_TEXT, CENTER_TEXT );
   pieslice( 140, 355, 135, 225, 100 );
   settextjustify( RIGHT_TEXT, CENTER_TEXT );
   outtextxy( x, y, "25 %" );

   setfillstyle( HATCH_FILL, BLUE );
   pieslice( 150, 355, 225, 360, 100 );
   settextjustify( RIGHT_TEXT, CENTER_TEXT );
   outtextxy( x, y, "37.5 %" );
   settextjustify(LEFT_TEXT, BOTTOM_TEXT);
   setcolor(BLUE);
   outtextxy( 250, 360, " 38 % " );
   setcolor(BLUE);
   outtextxy( 55, 360, " 17.3 % " );
   setcolor(RED);
   outtextxy( 85, 285, " 16.7 % " );
   outtextxy(15,460, " Gráfica de");
   outtextxy(195,460, "pastel circular");
   setfillstyle( LINE_FILL, BROWN );
   sector( 470, 355, 0, 90, 130,50 );
   setfillstyle( CLOSE_DOT_FILL, CYAN );
   sector( 460, 345, 90, 180, 130,50 );
   setfillstyle( SOLID_FILL, LIGHTBLUE );
   sector( 470, 355, 180, 208, 130,50 );

   sector( 480, 375, 208, 360, 130,50 );

   setcolor(BLUE);
   outtextxy( 312, 385, "  8 %" );  setcolor(BROWN);
   outtextxy( 380, 320, " 25.0 % " );
   setcolor(RED);
   outtextxy( 500, 295, " 25.0 % " );
   setcolor(GREEN);
   outtextxy( 572, 420, " 42.0 % " );
   setcolor(BLUE);
   outtextxy(370,450, " Gráfica de pastel elíptico");
   getch();
   closegraph();
   return 0;
}
```



GRÁFICAS DE FUNCIONES

En el estudio de las ciencias físico matemáticas, se requiere analizar diversas funciones, considerando como parte de ello sus gráficas cartesianas en la pantalla; obviamente, los valores máximo y mínimo de muchas funciones exceden los valores físicos de la pantalla o son muy pequeños para observarse con los valores de la resolución de la pantalla. La manera de evitar este problema en las versiones más recientes del lenguaje C, es asignar nuevas dimensiones o escala a la pantalla (mediante coordenadas lógicas que asignan coordenadas cartesianas y un nuevo tamaño), sin embargo en esta introducción al manejo de gráficas, exclusivamente se bosquejará el enfoque básico del factor de escala y el desplazamiento de ejes cartesianos.

Para dibujar los ejes coordenados con el origen de coordenadas en el centro de la pantalla, es conveniente usar las funciones:

```
getmaxx();

getmaxy();
```

que automáticamente determinan los valores máximos de resolución tanto horizontalmente (valores de x) como verticalmente (valores de y); obtenidos esos valores, el centro de la pantalla está a la mitad de ellos.

Se debe ajustar los valores de la función a una escala tal que los valores máximos de la gráficas no sobrepasen del 80 al 99 % de los valores físicos de la pantalla para asegurar que la gráfica se aprecia perfectamente en la pantalla. Un método para graficar consiste en tabular primero la función en un intervalo predeterminado para obtener sus valores máximos (positivo y negativo) y con ellos determinar el factor de escala y el desplazamiento de los ejes coordenados que se requiere.

Debe tenerse en cuenta que el número máximo de puntos de la función que se podrán iluminar corresponde al valor obtenido con getmaxx(); de tal manera que el incremento mínimo para tabular la función puede obtenerse con:

```
incremento = (valor_final_de_x  -  valor_inicial_de_x)/getmaxx();
```

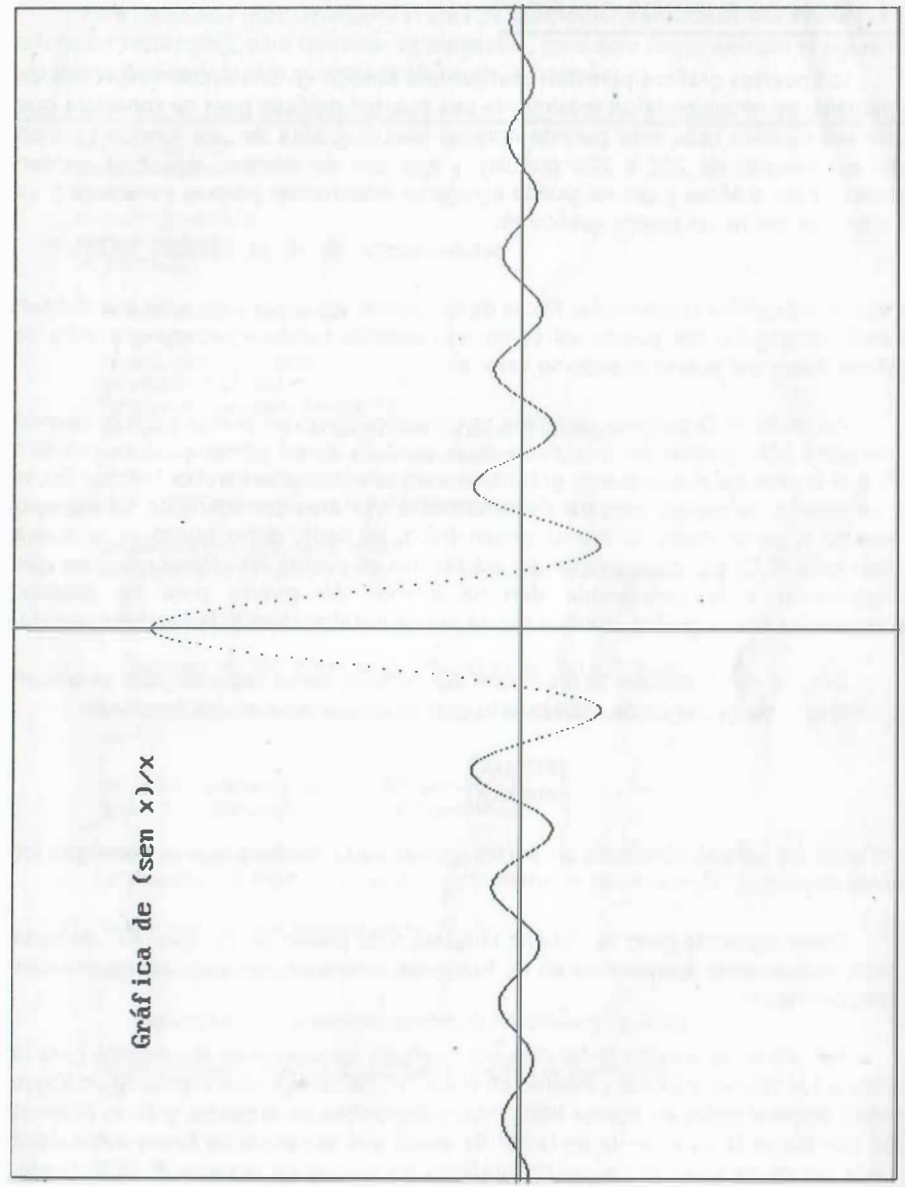
con esa expresión se puede obtener para cualquier monitor el máximo de puntos a iluminar (por ejemplo en VGA un máximo de 640 puntos). Debe verificarse la congruencia del incremento que se emplea para tabular para que realmente se obtengan puntos representativos de la gráfica de la función y no valores aislados o muy separados que al unirse no corresponden al contorno continuo de la gráfica de la función.

Por ejemplo considere el obtener la gráfica de la función:

$$y = \frac{sen(x)}{x}$$

cuyos valores máximo y mínimo no sobrepasan la unidad, es decir, valores que no es posible graficar directamente en la pantalla pues se requiere un factor de escala, por ejemplo un valor de 200 que amplifique la gráfica para representarla mediante los pixeles; además se requiere desplazar el origen de coordenadas e invertir la gráfica:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
int main(void)
{
    int controlador, modo;
    int i;
    controlador = DETECT;
    initgraph(&controlador, &modo, "");
    setcolor(RED);          /* se asigna color de líneas  y rectángulo */
    setbkcolor(WHITE);      /* se asigna color de fondo   */
    line(0,275,639,275);
    line(300,0,300,479);
    rectangle(1,1,638,478);

    gotoxy(10,5);
    printf("Gráfica de (sen x)/x ");

/* La siguiente gráfica se realiza iluminando puntos sobre la pantalla;
   para ello se ha amplificado la gráfica de la función y desplazado los
   ejes para ajustarlos a los puntos físicos de la pantalla.
*/
    for(i=0; i<=639; i+=1)
    {
        putpixel(i,275-200.0*sin((i-300.0)/10.0)/((i-300.000061)/10),8LUE);
    }
    getch();
    closegraph();
    return 0;
}
```

CREACIÓN DE PUERTOS GRÁFICOS

Los puertos gráficos permiten graficar una función en una sección específica de la pantalla; se recomienda un máximo de seis puertos gráficos pues se considera que hasta ese número cada área permite apreciar bien la gráfica de una función (con un total aproximado de 200 x 200 puntos), y que con un número mayor se pierden detalles de las gráficas y casi no puede agregarse información (valores y mensajes). La función que define un puerto gráfico es:

setviewport(xi, yi, xd, yd, bandera_recorte);

en ella se indican las coordenadas físicas de los puntos izquierdo y derecho que definen el área rectangular del puerto así como una variable bandera_recorte que evita se grafique fuera del puerto cuando su valor es 1.

Por omisión, la pantalla completa se considera como un puerto gráfico; cuando se requiere sólo graficar en una parte de la pantalla como primer paso se requiere definir el áreade ese nuevo puerto gráfico; una vez seleccionadas las coordenadas físicas de un puerto, la gráfica utilizará exclusivamente esa área considerando su extremo izquierdo superior como su nuevo origen físico, es decir, dicho punto es la nueva coordenada (0,0), por consiguiente deberá tenerse en cuenta los valores máximos que corresponden a la coordenada derecha inferior del puerto para no intentar sobrepasarlos con la gráfica, aunque ello se pueda evitar activando la bandera_recorte.

En caso de no conocer la resolución del monitor, como segundo paso se ubican los centros y los extremos de cada puerto gráfico apoyándose en las funciones:

getmaxx();
getmaxy();

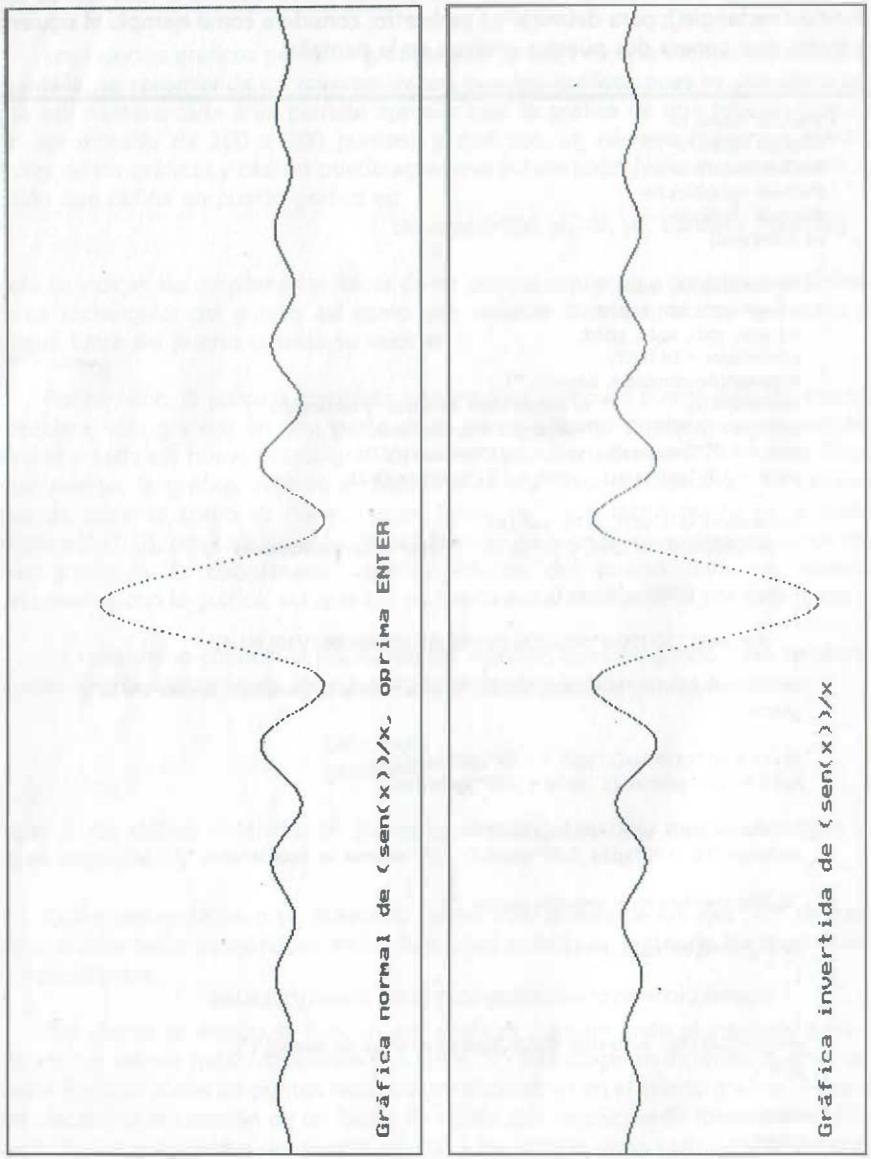
dividiendo los valores obtenidos en partes iguales de tal manera que se obtengan los valores deseados.

Como siguiente paso se obtiene el número de pixeles de los ejes "XY" de cada puerto, nuevamente apoyándose en las funciones anteriores, restando las magnitudes correspondientes.

Por último se analiza la función por graficar, considerando el intervalo para la gráfica y sus valores máximo y mínimo en el eje "Y". Se escoge un incremento en X que permita emplear todos los puntos horizontales disponibles en el puerto gráfico. Además debe plantearse la creación de un factor de escala que se calcule de forma automática a partir de los parámetros del puerto gráfico y los valores de la variable dependiente. En esta introducción se emplearán factores de escala estimados manualmente.

Para reconocer más fácilmente el área de cada puerto gráfico se recomienda usar la función rectangle(); para delimitar su perímetro; considere como ejemplo el siguiente programa que genera dos puertos gráficos en la pantalla:

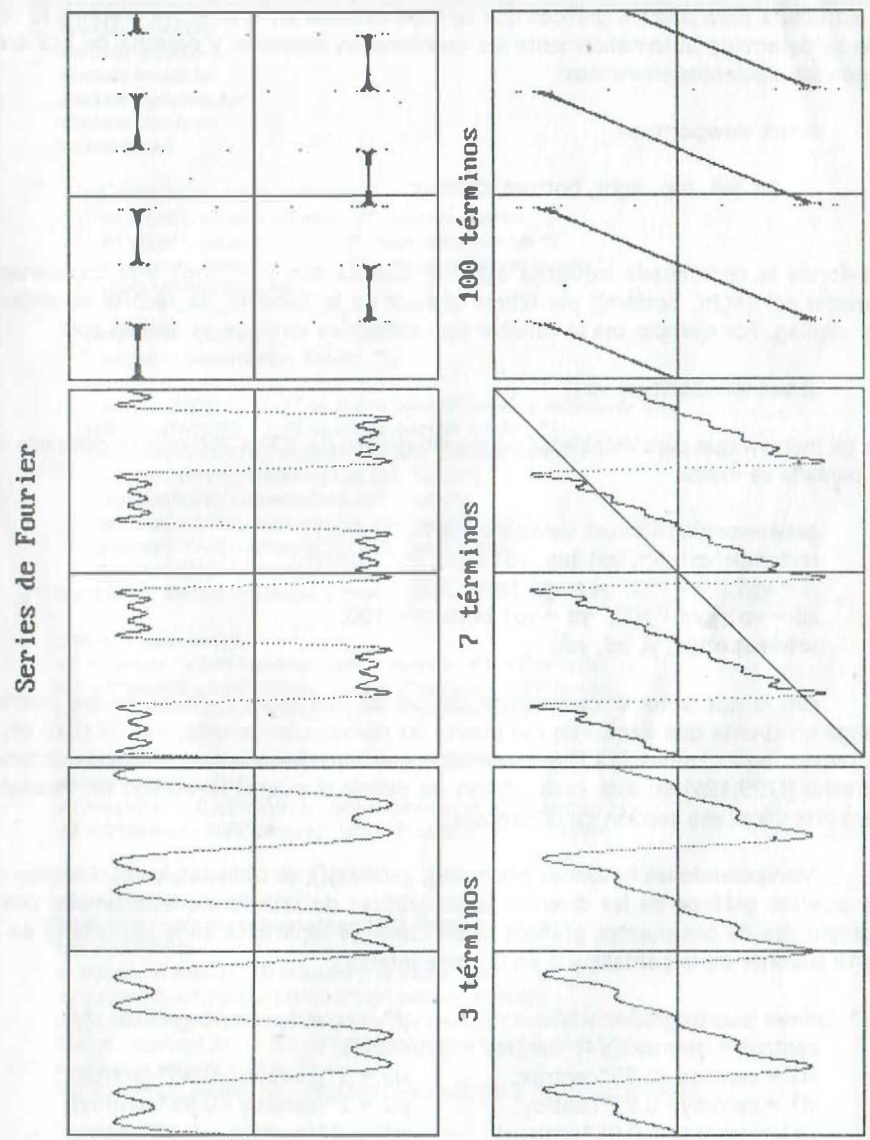
```
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
int main(void)
{
   int controlador, modo, i;
   int xp1i,yp1i, xp1d,yp1d;
   int xp2i, yp2i, xp2d, yp2d;
   controlador = DETECT;
   initgraph(&controlador, &modo, "");
   setcolor(RED);          /* se asigna color de líneas  y rectángulo */
   setbkcolor(WHITE);      /* se asigna color de fondo   */
   xp1i = 0.01*getmaxx();  yp1i = 0.03*(getmaxy()/2);
   xp1d = 0.99*getmaxx();  yp1d = 0.97*(getmaxy()/2);

   setviewport(xp1i, yp1i, xp1d, yp1d,0);
   rectangle(0,0, 0.99*xp1d, 0.97*yp1d);  /* observe las coordenadas  */

   for(i=0; i <= 0.99*xp1d; i+=1)
   {
      putpixel(i,150-100.0*sin((i-300.0)/10.0)/((i-300.000061)/10),BLUE);
   }
   setcolor(MAGENTA); outtextxy(10,200, "Gráfica normal de (sen(x))/x, oprima ENTER");
   getch();

   xp2i = 0.01*getmaxx();  yp2i = 1.03*(getmaxy()/2);
   xp2d = 0.99*getmaxx();  yp2d = 0.97*(getmaxy());

   setviewport(xp2i, yp2i, xp2d, yp2d,0);
   rectangle(0,0, 0.99*xp2d, 0.97*yp2d/2);  /*  observe las coordenadas */

/*  Gráfica invertida en el segundo puerto  */

   for(i=0; i<=0.99*xp2d; i+=1)
   {
      putpixel(i,100+100.0*sin((i-300.0)/10.0)/((i-300.000061)/10),BLUE);
   }
   setcolor(GREEN);  outtextxy(10,200,"Gráfica invertida de (sen(x))/x");
   getch();

   closegraph();
   return 0;
}
```



Como ejemplo de un programa con seis puertos gráficos a continuación se grafican dos series de Fourier mediante 3 puertos que muestran gráficas con 3,7 y 100 términos:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
float f1(int n, float x)
{
    float f=0; int i;
    for(i=1; i<=n; i+=2)
    {
        f += sin(i*x)/i;
    }
    return (10.0/3.14159*f);
}
float f2(int n, float x)
{
    float f=0; int i;
    for(i=1; i<=n; i+=2)
    {
        f += sin(i*x)/i - sin((i+1)*x)/(i+1.0);
    }
    return (6.0/3.14159*f);
}
int main(void)
{
  int controlador, modo, errorcode;
  float x;
  window(25,1,60,1); textbackground(RED); clrscr();
  textcolor(BLUE + BLINK); gotoxy(21,1);
  cprintf("                    Gráficas de funciones ");
  sleep(1);
  controlador = DETECT;
  initgraph(&controlador, &modo, "");
  setpalette(0,EGA_CYAN);
  printf("\t\t\t\t\t      Series de Fourier  ");

  setviewport(10,30,210,230,1);
  rectangle(0,0,200,200);
  line(0,100,200,100);
  line(100,0,100,200);
  for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f1(3,x/10.0),BROWN);

  /* continua el programa en la siguiente página   */
```

```
/* continuación del programa de las gráficas de Fourier    */

setviewport(220,30,420,230,1);
rectangle(0,0,200,200);
line(0,100,200,100);
line(100,0,100,200);
for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f1(7,x/10.0),RED);

setviewport(430,30,630,230,1);
rectangle(0,0,630-430,200);
line(0,100,200,100);
line(100,0,100,200);
for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f1(100,x/10.0),BLUE);

printf("\n\n\n\n\n\n\n\n\n\n\n\n\n          3 términos ");
printf("              7 términos                100 términos");

setviewport(10,260,210,460,1);
for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f2(3,x/10.0),BROWN);
rectangle(0,0,200,200);
line(0,100,200,100);
line(100,0,100,200);

setviewport(220,260,420,460,1);
rectangle(0,0,200,200);
line(0,100,200,100);
line(100,0,100,200);
line(0,200,200,0);
for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f2(7,x/10.0),RED);

setviewport(430,260,630,460,1);
rectangle(0,0,630-430,200);
line(0,100,200,100);
line(100,0,100,200);
for(x=0; x<=200; x+=.1) putpixel(x, 100 - 25.0*f2(100,x/10.0),BLUE);
getch();
closegraph();
return 0;
}
```

Para determinar automáticamente las coordenadas de un puerto gráfico, se declara una variable del tipo estructura que el programa asociará automáticamente a la estructura para puertos gráficos que se haya definida en la directiva graphics.h; con ello se determina automáticamente las coordenadas izquierda y derecha de esa área según los siguientes elementos:

```
struct viewporttype
{
    int left, top, right, bottom, clipflag;
}
```

en donde la coordenada izquierda esta identificada con (left, top); y la coordenada derecha por (right, bottom); por último el valor de la_bandera_de_recorte se obtiene en clipflag. Por ejemplo sea la variable tipo estructura vp1 que se declara con:

```
struct viewporttype vp1;
```

de tal manera que para establecer un puerto gráfico de 300 x 200 pixeles centrado en la pantalla se indica:

```
getviewsettings((struct viewporttype far *) &vp1);
rectangle(vp1.left, vp1.top, vp1.right, vp1.bottom);
xi = vp1.left - 150;  yi = vp1.top + 150;
xd = vp1right - 100;  yd = vp1.bottom +.100;
setviewport(xi, yi, xd, yd);
```

con lo que automáticamente se obtuvo las coordenadas físicas de ese puerto. Tenga en cuenta que dentro de ese puerto las nuevas coordenadas van de (0,0) en el extremo superior izquierdo a las coordenadas máximas del mismo en el extremo inferior derecho [(299,199) en este caso]. Antes de definir el puerto se agregó un recuadro para identificar esa sección de la pantalla.

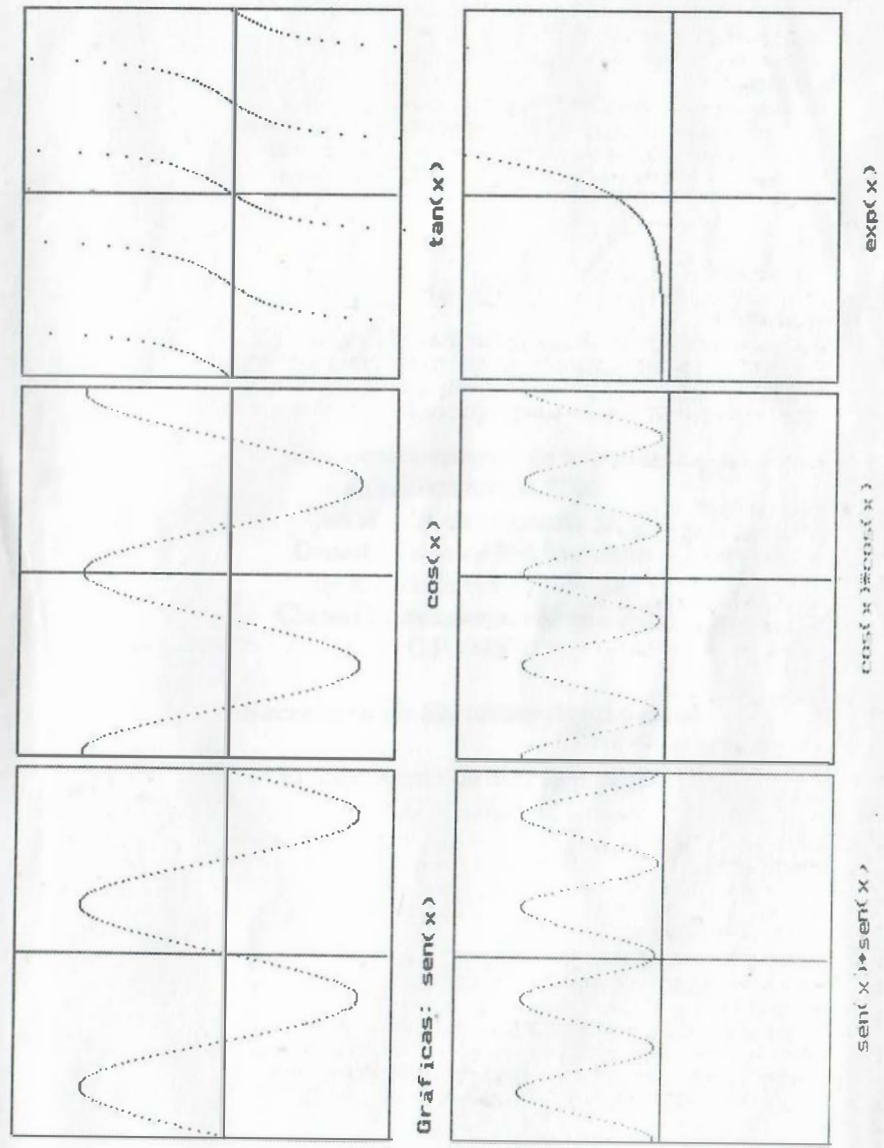
Manipulando las funciones getmaxx(); getmaxy(); se obtienen los parámetros de los puertos gráficos de las diversas áreas gráficas de una forma más simple; como ejemplo dos de seis puertos gráficos simétricamente repartidos en la pantalla (3 en la parte superior de la pantalla y 3 en la parte inferior):

```
/*  primer puerto gráfico  */              /*  segundo puerto gráfico  */
    centrox = getmaxx()/4;  centroy = getmaxy/3;
    xi1 = centrox - 0.95*centrox;          xi2 = 2*(centrox - 0.95*centrox);
    yi1 = centroy - 0.95*centroy;          yi2 = 2*(centroy - 0.95*centroy);
    xd1 = centrox + 0.95*centrox;          xd2 = 2*(centrox + 0.95*centrox);
    yd1 = centroy + 0.95*centroy;          yd2 = 2*(centroy + 0.95*centroy);
```


A continuación se muestra el programa completo para los seis puertos:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
int main(void)
{
   int controlador, modo, i; float x;
     int xi1,xd1, xi2,xd2, xi3,xd3;  /*  abscisas puertos  */
     int yi1,yd1, yi2,yd2;           /*  ordenadas puertos */
     int centrox, centroy;           /*  centro del primer puerto */
     struct viewporttype vp1;

   controlador = DETECT;
     initgraph(&controlador, &modo, "");

     setcolor(RED);          /* se asigna color de líneas  y rectángulo */
   setbkcolor(WHITE);        /* se asigna color de fondo  */
     setcolor(MAGENTA); outtextxy(10,225, "Gráficas: sen(x)");
     setcolor(GREEN); outtextxy(300,225, "cos(x)");
     setcolor(BLUE); outtextxy(500,225, "tan(x)");
     setcolor(BROWN); outtextxy(55,460, "sen(x)*sen(x)");
     setcolor(CYAN); outtextxy(267,460, "cos(x)*cos(x)");
     setcolor(RED); outtextxy(500,460, "exp(x)");
/*  se obtienen abscisas izquierdas y derechas de los puertos gráficos  */

   centrox = getmaxx()/6;
   xi1 = centrox  - 0.95*centrox;   xd1 = centrox  + 0.95*centrox;
   xi2 = 3*centrox - 0.95*centrox;  xd2 = 3*centrox + 0.95*centrox;
   xi3 = 5*centrox - 0.95*centrox;  xd3 = 5*centrox + 0.95*centrox;

/*  se obtienen ordenadas superiores e inferiores de los puertos  */
   centroy = getmaxy()/4;
   yi1 = centroy  - 0.95*centroy;   yd1 = centroy  + 0.95*centroy;
   yi2 = 3*centroy - 0.95*centroy;  yd2 = 3*centroy + 0.95*centroy;

/*  primer puerto gráfico  */
   setviewport(xi1,yi1,xd1,yd1,1);
   getviewsettings((struct viewporttype far*) &vp1);
   setcolor(GREEN);
/*  se delimita el área con un recuadro y se trazan ejes */
   rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
   line(0,(vp1.bottom-vp1.top)/2.0,vp1.right-vp1.left,(vp1.bottom-vp1.top)/2.0);
   line((vp1.right-vp1.left)/2.0,0,(vp1.right-vp1.left)/2.0,vp1.right-vp1.left);
/*  se dibuja la primera gráfica  */
   for(i=0,x=-6.42; i<=200; i++,x+=0.0633) putpixel(i,110.0-75.0*sin(x),BROWN);

/*  continúa el programa en la siguiente página  */
```

```
/* segundo puerto gráfico */

    setviewport(xi2,yi1,xd2,yd1,1);
    getviewsettings((struct viewporttype far*) &vp1);
    setcolor(BLUE);
    rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
    line(0,(vp1.bottom-vp1.top)/2,vp1.right-vp1.left,(vp1.bottom-vp1.top)/2);
    line((vp1.right-vp1.left)/2,0,(vp1.right-vp1.left)/2,vp1.right-vp1.left);
    for(i=0,x=-6.42; i<=200; i++,x+=0.064) putpixel(i,110.0-75.0*cos(x),GREEN);

/* tercer puerto gráfico */

    setviewport(xi3,yi1,xd3,yd1,1);
    getviewsettings((struct viewporttype far*) &vp1);
    setcolor(RED);
    rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
    line(0,(vp1.bottom-vp1.top)/2,vp1.right-vp1.left,(vp1.bottom-vp1.top)/2);
    line((vp1.right-vp1.left)/2,0,(vp1.right-vp1.left)/2,vp1.right-vp1.left);
    for(i=0,x=-6.42; i<=200; i++,x+=0.063) putpixel(i,110.0-25.0*tan(x),BLUE);

/* cuarto puerto gráfico */

    setviewport(xi1,yi2,xd1,yd2,1);
    getviewsettings((struct viewporttype far*) &vp1);
    setcolor(GREEN);
    rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
    line(0,(vp1.bottom-vp1.top)/2.0,vp1.right-vp1.left,(vp1.bottom-vp1,top)/2.0);
    line((vp1.right-vp1.left)/2.0,0,(vp1.right-vp1.left)/2.0,vp1.right-vp1.left);
    for(i=0,x=-6.42; i<=200; i++,x+=0.063) putpixel(i,110.0-75.0*sin(x)*sin(x),BROWN);

/* quinto puerto gráfico */

    setviewport(xi2,yi2,xd2,yd2,1);
    getviewsettings((struct viewporttype far*) &vp1);
    setcolor(BLUE);
    rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
    line(0,(vp1.bottom-vp1.top)/2,vp1.right-vp1.left,(vp1.bottom-vp1.top)/2);
    line((vp1.right-vp1.left)/2,0,(vp1.right-vp1.left)/2,vp1.right-vp1.left);
    for(i=0,x=-6.42; i<=200; i++,x+=0.063) putpixel(i,110.0-75.0*cos(x)*cos(x),CYAN);

/* sexto puerto gráfico */
    setviewport(xi3,yi2,xd3,yd2,1);
    getviewsettings((struct viewporttype far*) &vp1);
    setcolor(RED);
    rectangle(0,0,vp1.right-vp1.left,0.9*(vp1.bottom-vp1.top));
    line(0,(vp1.bottom-vp1.top)/2,vp1.right-vp1.left,(vp1.bottom-vp1.top)/2);
    line((vp1.right-vp1.left)/2,0,(vp1.right-vp1.left)/2,vp1.right-vp1.left);
    for(i=0,x=-3; i<=200; i++,x+=0.063) putpixel(i,110.- exp(x),RED);
    getch();
}
```

Esta obra se terminó de imprimir
en septiembre de 2005
en el taller de imprenta del
Departamento de Publicaciones
de la Facultad de Ingeniería,
Ciudad Universitaria, México, D.F.
C.P. 04510

Secretaría de Servicios Académicos

El tiraje consta de 300 ejemplares