78
2ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES "
"A R A G O N"

METODOLOGIA DE LA PROGRAMACION ESTRUCTURADA Y LENGUAJE C

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A:

SERGIO ZARAGOZA LIÑAN

SAN JUAN DE ARAGON, ESTADO DE MEXICO 1997

LA METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA Y LENGUAJE C

METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA Y LENGUAJE C

Dedicatorias

A mi Esposa :

A ti Irma Lorena, por el inmenso Amor que siempre me has demostrado en todo momento, y que sin él y sin tu apoyo, habría sido imposible la realización de esta Tesis. Te Amo...

A mi hija Itzel Alejandra :

Por la enorme satisfacción que me diste de ser padre y por los alientos que me das con tu bella presencia.

A mis Padres :

A quienes les debo la vida, no tengo otra cosa sino agradecer todo el Amor desinteresado que siempre he recibido, es por eso que hago ésta la ocasión propicia para expresarles mi Amor y respeto perpetuo.

A Sylvia :

Dedico el presente trabajo de tesis a tu memoria, por tantas cosas bellas que por ti conocimos : el Amor, la hermandad, la unidad y sobre todo el tener fuerza de voluntad para seguir adelante, aunque sea en la circunstancia más adversa. Donde quiera que te encuentres, recibe mi Amor eterno y recuerda que sigues presente en nuestros corazones.

A Jaime, Ma. Esther y Carlos Alberto :

A Ustedes dedico la realización del presente trabajo, como símbolo de su gran apoyo y comprensión. Esperando que crezca el vínculo de Amor que nos une y que nunca olvidemos lo que somos.

A la UNAM :

Siempre viviré agradecido a la Universidad Nacional Autónoma de México por haberme albergado durante varios años, en los cuales conocí a personas que marcaron positivamente mi formación como profesionista; a mis Profesores y Compañeros de Clase, Gracias.

A mi Asesor de Tesis :

Agradezco la acertada asesoría en el desarrollo de este trabajo a mi Director de Tesis Ing. Davi J. González Maxinez, con admiración y respeto.

Al IMP :

Asimismo, agradezco al Área de Ingeniería Informática del Instituto Mexicano del Petróleo, las facilidades otorgadas en la elaboración de este trabajo, en especial al Ing. Zenón Pérez Matus.

A todos aquellos que de alguna manera intervinieron en el desarrollo y terminación de este trabajo de Tesis, Gracias...

Abreviaturas

PE	Programación Estructurada
TD	Tabla de Decisión
POO	Programación Orientada a Objetos
TMV	Tabla de Método Virtual

Introducción

El siguiente trabajo de Tesis esta conformado por tres capítulos, de los cuales el primero describe la Metodología de la Programación Estructurada, que fue utilizada como una herramienta completa en algún momento de la programación.

En sus inicios, la programación de los sistemas de cómputo estuvo determinada por el trabajo individual de los programadores; es decir, no había un grupo que se coordinara en la tarea del desarrollo de los sistemas, ni una forma sistemática de estudiar la complejidad de los mismos.

Como consecuencia de ello, el usuario al hacer uso del sistema tenía una serie de dificultades que en gran medida, se podrían atribuir a que la programación del sistema no cumplía con los requerimientos.

Por otra parte, cuando un programador codificaba individualmente sin atenerse a un método específico, ocasionaba que otros programadores tuvieran dificultad para implementar cambios sustanciales al sistema, tales como reprogramar módulos incorrectos, integrar otras funciones al sistema y cualquier otra especificación para que éste fuera actualizado.

Esto, naturalmente incrementaba los costos de mantenimiento, además de que un sistema de estas características ocasionaba grandes retrasos. Asimismo, los programadores no aplicaban una metodología en el desarrollo de la programación de los sistemas que permitiera obtener un producto que cumpliera con ciertas normas de calidad. Esto se debía a que el programador aplicaba la mayor parte de su esfuerzo a la programación y prueba del sistema y desatendía por completo las fases de Análisis y Diseño, además de que olvidaba que un programa también es un documento para comunicar la solución de un problema.

Resultaba así que mientras el equipo de cómputo progresaba grandemente y su costo disminuía, la eficiencia en la programación se incrementaba con la complejidad del sistema y, como consecuencia, sus costos se incrementaban. Esta situación llegó a agravarse tanto que dió origen a lo que se llamó Crisis del Software o de la programación; lo cual motivó la búsqueda de una solución a los problemas con objeto de:

- Abatir Costos de Mantenimiento
- Obtener Alta Confiabilidad
- Entregar a Tiempo
- Lograr una Administración Objetiva
- Otros

Es por eso que la Programación Estructurada surge como solución inmediata a los problemas de programación ya que logró satisfacer en aquel entonces la gravedad de la situación.

Se podría pensar que la PE actualmente esta atrasada o no, pero en realidad cualquier método que cumpla con el objetivo de minimizar los esfuerzos y recursos y aumentar la productividad será aceptado siempre.

Por esa razón en el segundo capítulo se ha incluido el Lenguaje de Programación C, como ejemplo de que en la actualidad muchos programadores siguen ocupando la PE como herramienta de diseño y elaboración de sistemas, además se ha seleccionado este lenguaje, porque en el capítulo tres y último se presenta la nueva tendencia de programación de sistemas que es la Programación Orientada a Objetos, y en la cual el lenguaje C tiene su mayor expresión.

El lenguaje C fue inventado e implementado por primera vez por Dennis Ritchie en un DEC PDP-11 usando UNIX como sistema operativo. C es el resultado de un proceso de desarrollo comenzado con un lenguaje anterior denominado BCPL, que todavía se sigue usando principalmente en Europa. BCPL fue desarrollado por Martin Richards e influenció otro lenguaje denominado B, inventado por Ken Thompson. En los anos 70's el lenguaje B llevó al desarrollo del C.

A menudo se denomina al lenguaje C como lenguaje de nivel medio. Esto no significa que sea menos potente, más difícil de usar o menos evolucionado que lenguajes de alto nivel, ni implica que C sea similar al lenguaje ensamblador y por tanto presente al usuario sus problemas asociados. C se presenta como lenguaje de nivel medio porque combina elementos de alto nivel con el funcionalismo del lenguaje ensamblador.

Cabe aclarar que en este trabajo no se describe la forma de compilar ningún programa, ya que sólo se mencionan las funciones del lenguaje básicas, y si se desea compilar se puede utilizar el compilador incluido en cualquier versión del lenguaje.

Es obvio, que no todas las librerías del C están incluidas ni todas las funciones, para ello se necesitaría realizar otro trabajo que se concentre básicamente en el lenguaje. Pero con las presentadas aquí bien se podría aprender lo más elemental de tan potente lenguaje.

Por último en el capítulo tres, se hace mención de la Programación Orientada a Objetos que actualmente es la herramienta más utilizada, por así decirlo, en la programación.

Aunque la POO, nació hace más de veinte años, es hasta fechas recientes que esta técnica de diseño ha recibido la atención de los ingenieros de software. La POO basa sus métodos en la creación de tipos de datos abstractos; el mundo real se modela utilizando entidades llamadas objetos que se describen de acuerdo a sus atributos (datos) y métodos (operaciones con esos datos). De esta forma, la POO hace enfásis en el diseño de los tipos para expresar los objetos, no en los algoritmos para manipularlos. El diseño de un sistema orientado a objetos es modular, además de que su código fuente puede ser utilizado con facilidad, mejorando los procesos de mantenimiento y extensión del sistema. El primer lenguaje orientado a objetos fue Simula (1967), posteriormente aparecieron otros lenguajes como Smalltalk (1970), Eiffel (1986), y algunas extensiones del Lisp (1980,1982).

Aunque C no es lenguaje totalmente orientado a objetos. su diseño se hizo bajo fuertes restricciones de eficiencia por lo que no causa la misma sobrecarga que lenguajes como Simula o Smalltalk. El lenguaje C se aplica actualmente en áreas como CAD, compiladores, manejadores de bases de datos, procesamiento de imágenes, simulación, robótica, redes y síntesis de música.

La POO es en la actualidad una realidad en el ambiente de la computación. El acelerado desarrollo que presenta es muy reciente. Sin embargo, no se trata de ideas nuevas, las raíces de las POO datan de algunas décadas atrás.

Durante varios años, la POO permaneció encerrada en una cúpula de cristal, era considerada una curiosidad académica, sin beneficio práctico (o muy poco) y tema de una élite de personas relacionadas con las ciencias de la computación, por lo cual fue prácticamente desconocida por numerosos programadores. No era posible entonces percibir la potencialidad y beneficios que podría acarrear la utilización de esta técnica. Sólo hasta hace pocos años, se ha observado una gran proliferación de las técnicas de programación orientada a objetos.

CAPÍTULO I

METODOLOGÍA DE LA PROGRAMACIÓN ESTRUCTURADA

I.1 LA PROGRAMACIÓN ESTRUCTURADA

La Programación Estructurada (PE) o Sistemática, fue la primera metodología que surgió como respuesta para solucionar problemas de programación en los sistemas de cómputo.

Se considera a Edsger W. Digkstra como el padre de la PE, ya que fue quien propuso un método que los resolviera. Digkstra hace mención de conceptos que son clásicos de esta metodología como son :

- El diseño de arriba hacia abajo " Top - Down "
- La programación con la exclusión de " Goto's "
- Otros

Sin embargo, la PE no resolvía todos los problemas, por ejemplo, no daba una visión general del sistema en las primeras fases del desarrollo del mismo.

Se propusieron métodos para solucionar este problema. Uno de ellos que ha sido aceptado en la actualidad es el Diseño Estructurado, pero este aún no resolvía el problema de la especificación de los requerimientos que el usuario pide del sistema, lo cual ha llevado a la búsqueda de conceptos fundamentales, incluso matemáticos para resolver este problema.

Un método que en la actualidad es ampliamente aceptado para la especificación de los requerimientos del usuario, es el Análisis Estructurado.

El Análisis, el Diseño y la Programación Estructurada, en realidad no se excluyen si no que se complementan, y actualmente es común usarlos en las fases de Análisis, Diseño y Programación de Sistemas respectivamente.

La PE es una serie de normas a seguir en el diseño y programación de los algoritmos de solución de problemas.

Siempre es deseable que nuestros programas cumplan con las siguientes características :

- Altamente confiables
- Fáciles de entender

- Fáciles de probar
- Fáciles de corregir
- Fáciles de mantener
- Eficientes en cuanto a memoria, tiempo y usuario

Las normas de la PE van desde el uso de lenguajes para el diseño de programas, hasta el hecho de correr las últimas pruebas y correcciones, cubriendo estas normas el hecho de cuales unidades lógicas son las que la PE permite y la forma de expresar nuestros programas mediante el uso de diagramas de bloques, diagramas de estructuras, etc. También nos propone una serie de estándares de programación :

- Estilo
- Propiedad
- Estructura
- Claridad
- Sangrado
- Unidades lógicas permitidas
- Documentación de programas, etc.

Mediante los cuales aseguramos que los programas tendrán como características principales claridad y facilidad de comprensión, lo que permitirá que satisfagan las necesidades del usuario.

Para visualizar los problemas que atacaremos a través de la programación, se diseñan algoritmos de solución, que deberán ser resueltos mediante el diseño de programas eficientes, midiendo la eficiencia en cuanto a memoria utilizada y tiempo de ejecución.

Además, como objetivo nuestros programas deben ser fáciles de entender, es decir, que nuestros compañeros programadores sepan de qué se trata nuestro programa y entiendan la función de cada una de las variables que estamos utilizando.

Por último, es el hecho de que los programas una vez elaborados deben ser sencillos de mantener y modificar de acuerdo a las necesidades de los usuarios y del sistema que lo contiene.

Con la aparición del Diseño Estructurado y de la PE, surgen una serie de técnicas en cuanto a la mejor forma de enfocar los problemas y de elaborar la solución.

I.1.1 Definición del Problema

La definición del problema, es una parte esencial dentro de la programación, ya que no se puede resolver alguna inquietud, si no se define correctamente, es decir, concentrar dentro de un Universo lo que nos interesa resolver, debemos saber las necesidades del usuario y tener el máximo de argumentos u opciones para poder satisfacer tales necesidades, poder enunciar un método para resolverlo.

Este paso, a menudo es obviado, incluso por programadores profesionales, debido a las prisas por crear el programa y satisfacer necesidades urgentes. El problema puede ser tan simple como el de querer automatizar una sencilla tarea, pero en el proceso de definirlo, habría que preguntar a la gente que vaya a usar el programa para ver lo que espera de él.

Todo elemento desarrollado por el hombre primero es una idea en su mente. Los sistemas computacionales, como otros productos de la tecnología, se desarrollan en respuesta a requerimientos detectados. Las fuentes que producen las ideas de productos de programación incluyen las necesidades del cliente generadas externamente, las necesidades internas de la organización, planes de mercadotecnia, y los planes o misiones organizacionales. La mayor parte de las organizaciones que desarrollan productos de programación son muy selectivas al decidir qué productos desarrollarán; no explotan todas las oportunidades. La decisión de llevar a cabo el proyecto se basa, generalmente, en el resultado de un estudio de factibilidad.

El primer paso en la planeación de un proyecto de programación es preparar, en la terminología del cliente, un enunciado breve del problema que se solucionará y de las restricciones que existen en su resolución. El enunciado definitivo del problema debe incluir una descripción de la situación actual y de las metas que debe lograr el nuevo sistema.

La definición del problema requiere de un entendimiento cabal del dominio del problema y del entorno de éste. Las técnicas para obtener este conocimiento, por parte del planeador, son entrevistas con el cliente, observación de las tareas problemáticas, y desarrollo de las reales.

Algunas veces los sistemas computacionales se construyen para aliviar un síntoma y no la causa primaria del problema. Esto ocurre cuando el problema real se entiende, pero no puede resolverse debido a circunstancias económicas, políticas o sociales, cuando el cliente no es capaz de comunicar el problema real o cuando el planeador no entiende la explicación del cliente sobre el problema.

El segundo paso en la planeación de un proyecto de programación es determinar lo apropiado de una solución computacional. Además de ser eficaz en términos de costo, un sistema debe aceptarse social y políticamente. Para ser eficiente en costo, un nuevo producto de programación debe proporcionar los mismos servicios e información que el sistema antiguo, usando menos tiempo y personal, o proporcionar servicios e información que antes eran innaccesibles.

Un sistema computacional está formado por los subsistemas de personal, equipo y de productos de programación, más las interconexiones entre ellos. El primer subsistema incluye operadores, personal de mantenimiento y usuarios finales. El segundo comprende el equipo de cómputo y los dispositivos periféricos, y puede tener otros dispositivos como sensores y accionadores para control de proceso. El tercer subsistema contiene programas que deben desarrollarse, más programas que ya existen y que pueden emplearse como están o modificándolos.

Las funciones que debe realizar cada subsistema principal se deben identificar, se deben establecer las interacciones entre subsistemas y determinar las restricciones en el desarrollo y operación para cada subsistema principal.

Las limitaciones especifican número y tipo de equipos, cantidad y habilidades de personal, y características del producto de programación como funcionamiento, precisión y nivel de confiabilidad. La asignación precisa de funciones entre equipo, programación y personal

puede dificultarse durante la planeación preliminar, tal vez sea necesario desarrollar primero un análisis detallado. No obstante, debe intentarse la definición preliminar de las funciones de los subsistemas principales.

Dado el enunciado preciso del problema y la indicación de las restricciones que existen para su solución, se pueden formular metas y requisitos preliminares.

Las metas son logros por alcanzar, sirven para establecer el marco de referencia para el proyecto de desarrollo del producto de programación. Estas se aplican tanto para el proceso de desarrollo como para los productos finales.

I.1.2 Identificación de los Módulos

Un módulo, está constituido por una o varias instrucciones físicamente contiguas y lógicamente encadenadas, las cuales se pueden referenciar mediante un nombre y pueden ser llamadas desde diferentes partes del programa. Un módulo puede ser un programa, una función, un subprograma, etc.

Características de un módulo

La salida del módulo debe ser función de la entrada, pero no de ningún estado interno. En esencia el módulo ha de ser una caja negra que facilite unos valores de entrada y suministre valores de salida que sean exclusivamente función de las entradas.

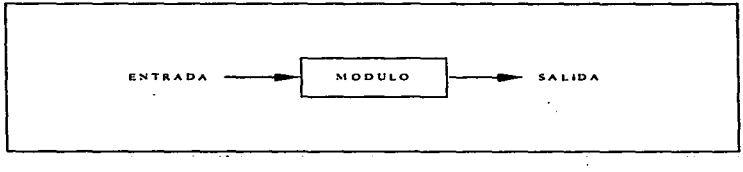


FIGURA 1

En la creación de los módulos deben cumplirse tres aspectos básicos: descripción, rendimiento y diseño.

En la descripción se definen las funciones y objetivos del programa. Para obtener el máximo rendimiento se ha de comprobar que el programa realice el proceso aprovechando al máximo todos los recursos de los que dispone. En cuanto al diseño se debe comprobar la estructura que sigue el módulo así como la estructura de datos y la forma de comunicación entre los diversos y diferentes módulos.

El organigrama modular se realiza mediante bloques, en el que cada bloque corresponde a un módulo y muestra gráficamente la comunicación entre el módulo principal y los secundarios.

El módulo principal debe ser claro y conciso, reflejando los puntos fundamentales del programa.

Los módulos básicos deben resolver partes bien definidas del problema. Sólo pueden tener un punto de entrada y un punto de salida. Si un módulo es complejo de resolver conviene subdividirlo en otros módulos. Ningún módulo puede ser llamado desde distintos puntos del módulo principal.

Las normas de programación dependerán del análisis de cada problema y de las normas generales o particulares que haya recibido el programador.

Clasificación de los módulos

Según las funciones que pueden desarrollar, cada módulo se clasifica en :

- Módulos tipo raíz, director o principal
- Módulos tipo subraíz
- Módulos de entrada (captura de datos)
- Módulos de variación de entradas
- Módulos de proceso
- Módulos de creación y formatos de salida

El hecho de dividir un programa en módulos, de ninguna manera se refiere a fragmentar un programa en pedazos sin ninguna jerarquía entre ellos, si no que cada módulo, además de guardar cierta jerarquía con respecto a los demás, debe de cumplir con lo siguiente :

Realizar una función específica

Cada módulo debe de realizar una función clara y el nombre del módulo reflejará esto mismo.

El diagrama de estructura como representación jerárquica entre módulos, es semejante al organigrama de una empresa y, al igual que la empresa, los módulos con mayor jerarquía sólo tienen la función de controlar y coordinar a los módulos subordinados.

Presentar bajo acoplamiento

Podemos decir, que el acoplamiento es una forma de medir la independencia entre módulos, debemos procurar que éstos sean lo más independientes posible, esto es, que la función de un módulo no dependa de lo que se haya hecho en otro, pues esto obviamente restringe la independencia entre ambos, por lo cual nunca debemos pasar banderas de un módulo de mayor jerarquía a otro de menor jerarquía.

Presentar alta cohesión

Cohesión es una medida de solidez interna del módulo. Se dice que el módulo es sólido cuando tiene una función que realizar y todas sus variables y actividades están encaminadas a dicho fin.

Entre más sólido sea un módulo, más independiente será de los demás. Por esto mismo, en términos generales se cumple que, a mayor cohesión menor acoplamiento y a menor cohesión mayor acoplamiento.

I.1.3 Refinamiento Sucesivo de los Módulos

El refinamiento por pasos es una técnica para la descomposición de sus especificaciones de alto nivel hasta sus niveles más elementales, esta técnica también se denomina "desarrollo a pasos de un programa" y "refinamiento sucesivo". Esta técnica requiere de las siguientes actividades :

1. Decisiones de diseño para la descomposición en niveles elementales.
2. Aislamiento de los aspectos de diseño que no sean totalmente interdependientes.
3. Posposición al máximo de las decisiones que conciernen a los detalles de representación.

4. Demostración cuidadosa de que en cada paso sucesivo, el refinamiento es sólo una expresión fiel de los pasos anteriores.

El aumento constante de detalle en cada paso en el proceso de refinamiento, permite la posposición al máximo de decisiones dando oportunidad al constructor de argumentar en forma definitiva que el producto desarrollado es consistente con las especificaciones de diseño.

El refinamiento por pasos empieza con las especificaciones derivadas del análisis de requerimientos y el diseño externo; el problema queda inicialmente descompuesto en un grupo de pasos fundamentales de trabajo que resuelven el problema; después se repite el proceso para cada una de estas partes hasta que se descompone en detalle suficiente para que la instrumentación en un lenguaje de programación sea sencilla.

No se requiere de una representación explícita para esta técnica, sin embargo, el uso de cartas de estructura, las especificaciones de procedimientos y el pseudocódigo son consistentes con esta técnica. Los pasos iniciales del refinamiento quedan establecidos por medio de un pseudocódigo informal que se vuelve más detallado conforme avanza el refinamiento. El diseño resultante es similar e inclusive puede incorporar a los estatutos del lenguaje utilizados en la instrumentación.

El refinamiento sucesivo puede ser utilizado para desarrollar el diseño detallado de los módulos particulares de un producto de programación.

Los principales beneficios del refinamiento por pasos de una técnica de diseño son :

1. Descomposición hacia abajo
2. Adición incremental del detalle
3. Aplazamiento de decisiones de diseño
4. Verificación continua de la consistencia

Por medio de un refinamiento por pasos, un problema se puede dividir en piezas pequeñas y manejables, y así la cantidad de detalle con en el que debe tratar en cierto tiempo particular se minimiza. De esta forma, el esfuerzo intelectual del diseñador se puede encauzar a los objetivos apropiados en el tiempo adecuado. Sin embargo, se debe observar que el refinamiento sucesivo no es tanto una técnica de diseñar sino una forma general para la

solución de problemas. El éxito que se obtenga con este método dependerá en gran medida de tener un concepto claro de la solución deseada y de la capacidad del diseñador.

I.1.3.1 Pseudocódigo y Diagramas Estructurados

Pseudocódigo : Descripción con palabras o fórmulas de un algoritmo, resumen general de las funciones de un programa que se utiliza con un diagrama de flujo para diseñar o documentar un programa.

Pseudo o seudo, significa falso, imitación y ***código*** se refiere a las instrucciones escritas en un lenguaje de programación; pseudocódigo no es realmente un código sino una imitación y una versión abreviada de instrucciones reales para las computadoras.

No existe una versión universal de pseudocódigo, lo único que se necesita es que su versión sea capaz de representar las tres estructuras lógicas fundamentales: secuencial, de selección e iterativa. La entrada, salida y procesamiento de datos se describen en frases condensadas y se utiliza el desplazamiento de las líneas para clarificar las relaciones entre las instrucciones en el pseudocódigo.

Con la PE, el pseudocódigo sigue siendo un excelente medio para expresar la lógica de un programa. Se utiliza además de las sentencias tradicionales la asignación, selección, E/S de datos, transferencia de control, ejecute, etc., las siguientes :

- ejecute (una función) mientras que (una condición) y fin-hmq (para la estructura HACER-MIENTRAS-QUE)
- ejecute (una función) hasta que (una condición) y fin-hhq (para la estructura HACER-HASTA-QUE)
- fin-si (para delimitar una estructura de selección SI-CIERTO-FALSO)

Por lo tanto, el pseudocódigo sigue siendo una herramienta muy útil en el análisis de programación. Las tres herramientas que utilizan los programadores son: diagramas de flujo estructurados, tablas de decisión y pseudocódigo.

Diagramas Estructurados

El Diagrama Estructurado es una representación gráfica del sistema que permite visualizar la instrumentación, documentación y mantenimiento de los sistemas. Este modelo se genera independientemente del tiempo y de las relaciones jerárquicas entre módulos de un programa o sistema, por eso no podemos inferir a partir de un diagrama estructurado cuál es el origen de ejecución de los módulos.

El diagrama estructurado esta formado por los siguientes elementos :

- Módulos
- Conexiones (Flechas)
- Interfases (Nombre de los datos que entran y salen de cada módulo)

El Diagrama Estructurado muestra lo siguiente :

- La división del sistema en módulos
- Jerarquía y organización de los módulos
- Interfases de comunicación entre módulos
- Nombres (funciones) de los módulos

La lógica del programa es el orden en el que la computadora ejecuta las instrucciones del mismo. Un diagrama de flujo estructurado es un gráfico que describe el orden de operaciones en el algoritmo. Puede por lo tanto, utilizarse para diseñar la lógica de un problema, antes de escribir el programa. Si se va actualizando el diagrama de flujo con todos los cambios en el programa, sirve también como documentación.

En general, un diagrama de flujo estructurado contará con las siguientes fases :

1. Inicio
2. Entrada de datos
3. Proceso
4. Salida de datos
5. Fin

Los símbolos que suelen utilizarse en los diagramas de flujo estructurados se muestran a continuación :

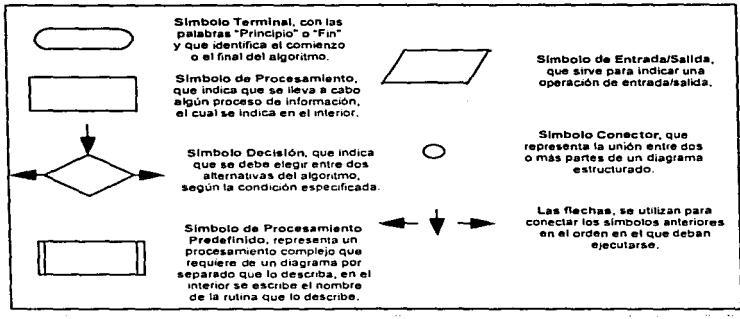


FIGURA 2

Las estructuras lógicas básicas necesarias para confeccionar un programa se reducen a tres : secuenciales, selectivas y repetitivas.

Estructuras Secuenciales

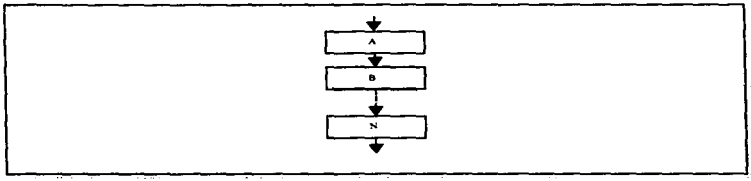


FIGURA 3

Constan esencialmente de pasos sucesivos, uno detrás de otro.

Estructuras Selectivas

Requieren una prueba o comparación de ciertas condiciones seguidas de rutas alternativas en el programa.

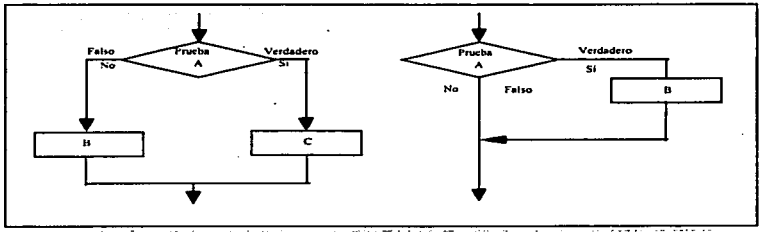


FIGURA 4

Estructuras Repetitivas

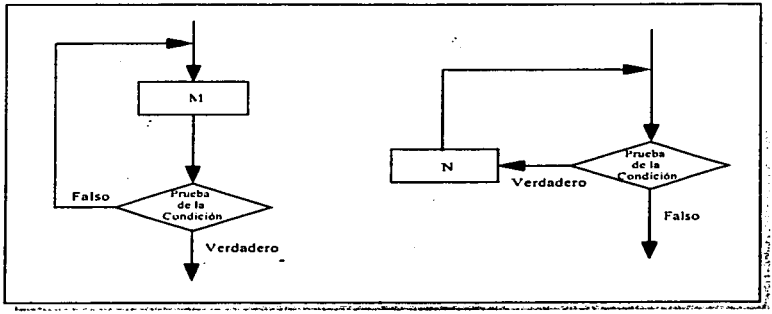


FIGURA 5

Cada una de las estructuras tiene una sola flecha de entrada y una sola flecha de salida y cada una de ellas es legible de arriba hacia abajo (diseño "top-down"), por consiguiente estas estructuras llamadas básicas o fundamentales pueden descomponerse sustituyendo cada bloque básico por una de dichas estructuras.

I.1.3.2 El Árbol y la Tabla de Decisión

El Árbol, es una estructura no lineal en la que cada elemento está relacionado con dos o más elementos. Un árbol consta de un conjunto de nodos (unidades de información) en las que además de la propia información contiene dirección de otros nodos de menor importancia o jerarquía, y cumple las siguientes condiciones :

- Existe un nodo raíz
- El resto de los nodos se distribuye en un número *n* de subconjuntos distintos.
- Cada uno de estos subconjuntos es un subárbol del nodo raíz.

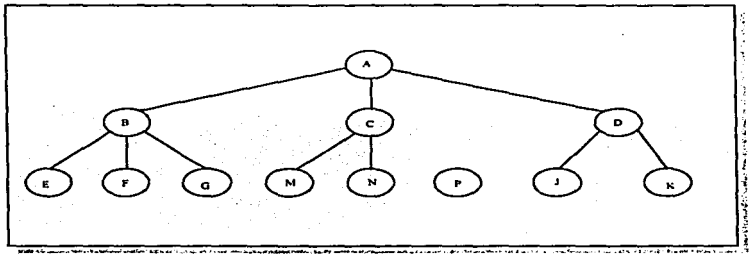


FIGURA 6

Grado de un nodo

Es el número de subárboles de ese nodo. El grado del nodo A es 3 y el del nodo D es 2.

Los nodos de grado son los terminales también llamados hojas. Las líneas que unen dos nodos se llaman lados, aristas o ramas.

Camino de un nodo

Es el conjunto de aristas a través de las cuales se pasa desde el nodo raíz a ese nodo. A cada nodo se le asocian uno o varios subárboles llamados descendientes o hijos. Los nodos B, C y D son hijos del nodo A. Los nodos hermanos son los sucesores directos de un mismo nodo (hijos de un mismo padre). E, F y G son todos hermanos. F y P no son nodos hermanos.

Si el nivel del nodo raíz es 1, el nivel de un nodo es el de su padre más 1. Se dice que un árbol es *n-ario* (*n*, constante entera) cuando el número máximo de hijos de un mismo nodo es *n*. Un árbol binario es un árbol en el cual cada nodo tiene, como máximo, dos hijos.

Todo árbol *n-ario* se puede transformar en un árbol binario equivalente. Un árbol no puede estar vacío, contiene como mínimo un elemento, la raíz. Un ejemplo gráfico completo de un árbol sería :

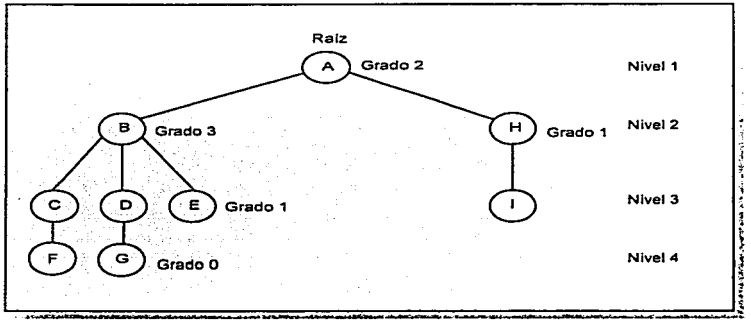


FIGURA 7

Otras formas de representar un árbol son :

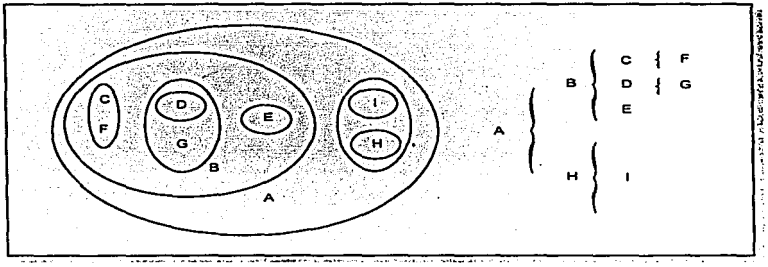


FIGURA 8

Árboles Binarios

Una de las estructuras más comunes de los árboles, es aquella en que cada nodo tiene como máximo dos subárboles (grado 2), que se conocen como subárbol izquierdo y subárbol derecho. Dos árboles binarios contiguos son distintos aunque estas representaciones sean idénticas.

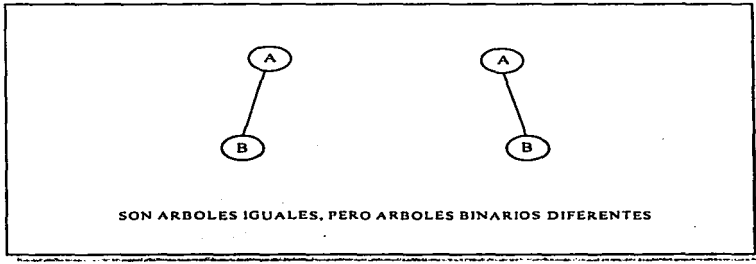


FIGURA 9

La altura o profundidad de un árbol binario es el número de nodos que constituyen el camino más largo desde la raíz a una hoja.

Un problema típico consiste en la conversión de un árbol ordinario en un árbol binario equivalente.

Un árbol binario se dice que está completo o lleno cuando tiene todos los nodos de grado 2 excepto los nodos terminales.

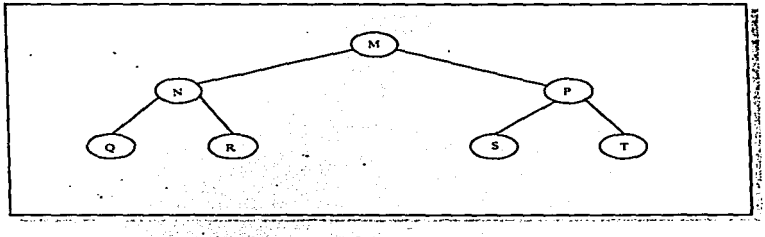


FIGURA 10

El problema de convertir un árbol cualquiera en un árbol binario se presenta a continuación.

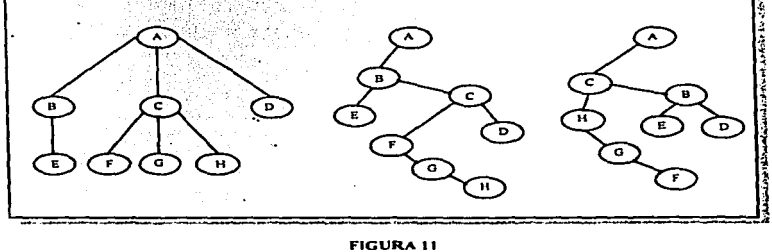


FIGURA 11

Recorrido de un árbol binario

Existen diferentes maneras de recorrer un árbol para tratar los diferentes nodos que los constituyen. Supongamos un árbol binario.

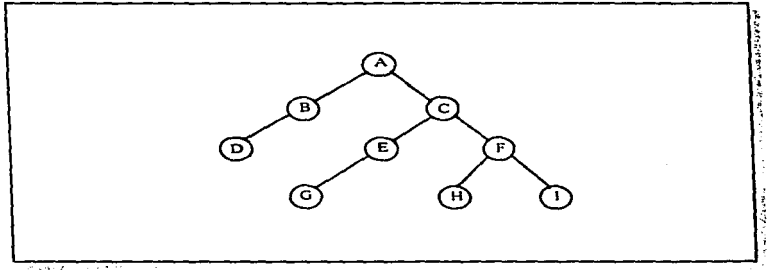


FIGURA 12

Las acciones a considerar son tres y según el orden en que se efectúen se tendrá el recorrido en un sistema o en otro :

- Posicionarse en la raíz
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho

Los tres tipos de recorridos más frecuentes y su aplicación concreta al árbol anterior son :

Recorrido prefijo o preorden

- Posicionarse en la raíz
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- A B D C E G F H I

Recorrido postfijo o postorden

- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- Posicionarse en la raíz
- D B G E H I F C A

Recorrido en orden o simétrico

- Recorrer el subárbol izquierdo
- Posicionarse en la raíz
- Recorrer el subárbol derecho
- D B A G E C H F I

Tablas de Decisión

Las Tablas de Decisión (TD) constituyen una herramienta poderosa para definir la lógica de un programa complejo. Constituyen un medio de representación de la información en forma tabular. cuyo objetivo primordial es el de aportar información en un formato que sea fácil de leer y comprender.

El uso de las tablas de decisión aunque no tan extendido como los diagramas estructurados, suele constituir en algunas ocasiones una técnica para capturar datos y la primera operación de análisis del problema. Una vez planteada la TD se suele realizar la construcción del algoritmo correspondiente. En ocasiones la TD sustituye a los diagramas de flujo.

La secuencia de un programa puede ser descrita por una estructura de decisión, la Tabla de Decisión.

Una TD es un documento de comunicación entre usuarios, analistas, programadores, etc., y un instrumento de análisis y programación, que se puede aplicar a numerosos y diversos problemas permitiendo la representación de las diferentes situaciones de un modo fácil y lógico.

Una TD es una herramienta que permite presentar de forma concisa las reglas lógicas que hay que utilizar para decidir acciones a ejecutar en función de las condiciones y la lógica de decisión de un problema específico.

Una TD es un tipo de tabla que muestra lo que debe realizar el programa cuando se cumplen ciertas condiciones. Son especialmente útiles para describir procesos que incluyan muchas decisiones múltiples.

En esencia las TD constituyen una técnica fácil de aprender y emplear que no requiere grandes esfuerzos de imaginación para su comprensión e interpretación.

Una TD se representa en un cuadro de cuatro bloques :

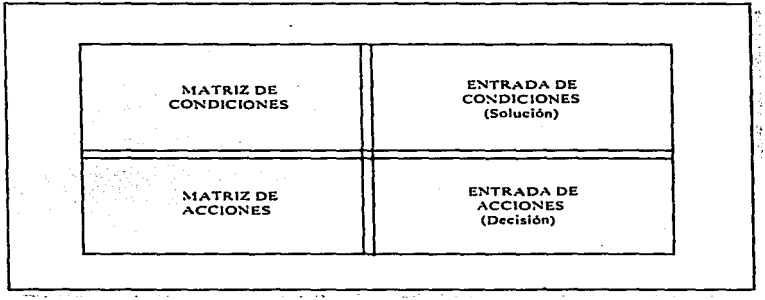


FIGURA 13

A la izquierda se tiene la matriz y a la derecha las entradas. La matriz de condiciones recoge las condiciones de todo o parte del problema, dicho en otras palabras, las preguntas que deben probarse para alcanzar una decisión. La matriz de acciones refleja todas o parte de las acciones del problema a realizar o las acciones que han de emprenderse cuando se presenta un conjunto dado de condiciones.

La entrada de las condiciones (combinación de las posibles condiciones), permiten reflejar en la TD si se cumplen o no tal condición o si es indiferente, es decir, sea cualquiera su condición de entrada, no tiene influencia sobre la acción que deberá efectuarse.

La entrada de las acciones (decisión) indica efectuar la acción correspondiente a un conjunto de condiciones que se han cumplido.

Reglas de Decisión

Cada combinación de entrada de condiciones y su correspondiente entrada de acciones constituyen una relación denominada regla de decisión. Existen tantas reglas como pares distintos de entradas condiciones/acciones. El número de reglas de decisión debe cubrir todos los casos posibles, sin repeticiones ni omisiones.

Notas de construcción

- El número de reglas de decisión es de 2 a la n el número de condiciones posibles.
- A una condición de entrada sólo le corresponde una decisión o condición de salida.
- A una condición de salida le pueden corresponder varias condiciones de entrada.
- Para que una TD esté bien construida es necesario que en cada momento sea cierta una y sólo una de las situaciones.
- El resultado de una regla no varía si se permuta el orden de las lineas de condición, ni la lógica de decisión cambia si se permuta el orden de las columnas (reglas).
- Cada regla de decisión (columna) es una estructura Si entonces (si se cumple la condición de entradas entonces realizar tal o tales acciones), del tipo booleano de condiciones y se le pueden aplicar las leyes del Álgebra de Boole.
- La entrada de condiciones se representa con los símbolos lógicos S, N o X (S, afirmativa; N, negativa; X, indiferente).
- La entrada de acciones se representan con una X si se realiza y en blanco o un guión (-), si no se realiza.
- La lista de condiciones puede ponerse en cualquier orden.

- La lista de acciones debe ponerse en el orden que se tengan que ejecutar.
- Cada columna (regla de decisión) de una TD equivale a una ruta o camino de un diagrama de flujo.

Modos de presentación de las tablas de decisión

Presentación HORIZONTAL :

- Las condiciones y las acciones se escriben horizontalmente.
- Las reglas se leen verticalmente.

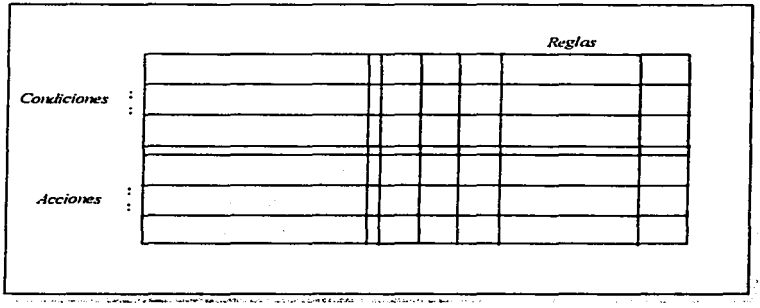


FIGURA 14

Presentación VERTICAL :

- Las condiciones y las acciones se escriben verticalmente.
- Las reglas se leen horizontalmente

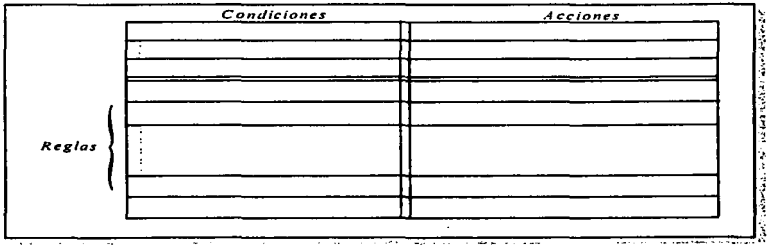


FIGURA 15

La representación de una TD completa se representa en la siguiente figura.

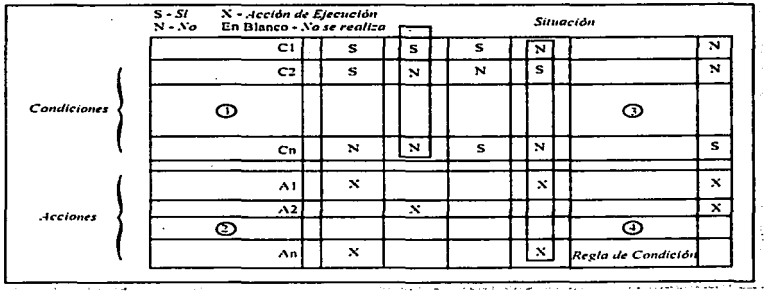


FIGURA 16

Las TD se leen siempre de izquierda a derecha y las reglas de arriba a abajo.

Las líneas horizontales y verticales dobles sirven como limite o frontera : las condiciones aparecen encima de la doble línea horizontal y los tratamientos debajo de dicha línea.

I.1.4 Instrumentación de los Módulos

La fase de instrumentación del desarrollo de la programación tiene que ver con la traducción de las especificaciones de diseño a código fuente. El objetivo principal de la instrumentación es el escribir código fuente y la documentación interna de modo que la concordancia del código con sus especificaciones sea fácil de verificar, y que se facilite la depuración, pruebas y modificaciones.

Este objetivo puede alcanzarse haciendo el código fuente tan claro y sencillo como sea posible. Sencillez, claridad, y elegancia son los sellos de los buenos programas; oscuridad, ingeniosidad, y complejidad son indicaciones de un diseño inadecuado y un pensamiento mal orientado.

La claridad del código fuente se mejora mediante técnicas de codificación estructurada, buen estilo de codificación, documentos adecuados de apoyo, buenos comentarios internos, y por las características que proporcionan los lenguajes de programación modernos.

La producción de una programación de alta calidad requiere que el equipo de programación tenga un líder designado, una estructura organizacional bien definida, y responsabilidades de cada miembro del equipo.

El equipo de instrumentación debe contar con un conjunto de requisitos de la programación bien definidos, una especificación del diseño estructural, y una descripción del diseño detallado. También, cada miembro del equipo debe comprender los objetivos de la instrumentación.

I.2 EL CICLO DE VIDA DE LOS SISTEMAS

El ciclo de vida de un sistema se ha centrado primeramente en el uso de técnicas estructuradas para desarrollar software, es decir, Análisis Estructurado, Diseño, Programación y Prueba del Sistema. Sin embargo, durante la vida de un sistema, el mantenimiento y mejoramientos pueden consumir más mano de obra y dinero, que el esfuerzo de desarrollo original.

El éxito del mantenimiento del sistema estructurado requiere documentación estructurada con administración apropiada, control y procedimientos.

El ciclo de vida del sistema esta firmemente involucrado con el proceso de desarrollo.

En el mundo real un proyecto de software a gran escala abarca un considerable periodo de tiempo. El ciclo de vida de un sistema es la secuencia de fases a desarrollar en el proceso de programación. Las fases típicas son :

1. Especificación o Definición de Requisitos (Andlisis).

Se establecen y especifican los requerimientos del software. Se refiere al periodo durante el cual se especifican las características funcionales y detalles operacionales. La entrada a esta fase son las necesidades declaradas para el software y la salida de la misma es un "documento de requerimientos". Precede al diseño y especifica lo que debe hacer sin especificar cómo hacerlo.

El proceso de diseño comienza tan pronto como la especificación de requerimientos esta terminada. Una parte importante del proceso de diseño del software es la descomposición del sistema software completo en un conjunto de subsistemas. Cada subsistema se descompone posteriormente en un conjunto de programas y procedimientos más pequeños. Es importante determinar si cualquiera de los procedimientos de los sistemas existentes se puede reutilizar en el nuevo sistema. Otra parte del proceso de diseño es determinar la representación interna de los datos.

2. Diseño.

El proceso de diseño comienza tan pronto como se especifican los requisitos.

El diseño es una fase predominantemente creativa. La entrada a esta fase es un documento de requerimientos (depurado y validado); la salida es un diseño expresado en alguna herramienta adecuada (por ejemplo, pseudocódigo).

3. Implementación (Codificación y Depuración).

Una vez que el diseño se ha terminado, este se codifica, es decir, se implementa (realiza) como un programa en un lenguaje de programación específico. En esta fase pueden intervenir los mismos programadores de fases anteriores o nuevos. Por esta causa, es importante que el diseño del software sea cuidadosamente documentado en un conjunto de diagramas de estructuras y pseudocódigo esencialmente.

Una vez que el programa ha sido codificado debe ejecutarse y depurarse. Cuando no quedan errores se habrá terminado la fase de depuración y codificación, y es momento de probar el producto desarrollado para el sistema.

4. Prueba.

La prueba de un programa debe ser un proceso riguroso, que normalmente se ejecutará por un grupo distinto de los programadores originales; en esta fase deben estar implicados los usuarios del software. Es importante identificar todos los errores temporalmente, ya que el software que controla un proceso delicado debe estar libre de errores antes de su primera utilización.

5. Funcionamiento y Mantenimiento.

El software debe poder correrse eficazmente durante el periodo de vida útil del mismo, incluso en entornos cambiantes. El proceso de actualizar o mantener un programa consiste en corregir los nuevos errores que se produzcan o incorporar los cambios necesarios.

El costo total de un proyecto es una función del tiempo implicado y del número de personas que trabajan en el proyecto a lo largo de su ciclo de vida completo. La división del ciclo de vida de los sistemas en fases constituyen una primera estimación o análisis de costo.

Se ha observado que no todas las fases contribuyen de igual modo al costo total del proyecto. Así, por ejemplo, la fase de mantenimiento puede contribuir tanto al costo como todas las fases de desarrollo combinadas. El ingeniero de software debe de tratar de reducir,

en la medida posible, el costo total; para ello debe distribuir juiciosamente el tiempo empleado entre todas las fases.

6. *Los Equipos de Programación.*

En la actualidad es difícil y raro que un gran proyecto de software sea implementado por un solo programador. Normalmente, un proyecto grande se asigna a un equipo de programadores, que por anticipado deben coordinar toda la organización global del proyecto.

Cada miembro del equipo es responsable de un conjunto de procedimientos, algunos de los cuales pueden ser utilizados por otros miembros del equipo. Cada uno de estos miembros deberá proporcionar a los otros las especificaciones de cada procedimiento. condiciones "pretest o postest" y su lista de parámetros formales; es decir, la información que un potencial usuario del procedimiento necesita conocer para poder ser llamado.

Normalmente, un miembro del equipo actúa como bibliotecario, de modo que a medida que un nuevo procedimiento se termina y comprueba, su versión actualizada sustituye la versión actualmente existente en la librería. Una de las tareas del bibliotecario es controlar la fecha en que cada nueva versión de un procedimiento se ha incorporado a la librería, así como asegurarse de que todos los programadores utilizan la versión ultima de cualquier procedimiento.

Es misión del equipo de programadores crear librerías de procedimientos, que posteriormente pueden ser utilizadas en otras aplicaciones. Una condición importante deben cumplir los procedimientos: estar comprobados y ahorro de tiempo/memoria.

CAPÍTULO II

LENGUAJE DE PROGRAMACIÓN C

II.1 ELEMENTOS BÁSICOS DE C

II.1.1 La Función main()

La función *main()* es especial, ya que es la primera función que se llama cuando el programa se ejecuta. Indica el comienzo del programa. Al contrario que en un programa BASIC, que comienza en el número de línea más bajo, o en la "cabecera" de programa; un programa C comienza con una llamada a la función *main()*.

La función *main()* es otra función C cualquiera, excepto que la llave que cierra la función *main()* indica el final del programa. Cuando esta llave se alcanza el programa cede control al sistema operativo.

Sólo puede haber un *main()* en un programa; si hubiera más de uno, el programa no sabría donde empezar la ejecución. La mayoría de los compiladores detectan los errores de este tipo antes de llegar a la fase de ejecución.

II.1.2 Identificadores

El lenguaje C llama identificadores a los nombres que se utilizan para aludir a las variables, funciones, etiquetas y otros varios objetos definidos por el usuario. Los identificadores de C pueden tener un caracter o varios. El primer caracter debe ser una letra o un guión de subrayado, siendo los caracteres siguientes letras, números o guiones de subrayado. En seguida se muestran algunos ejemplos de identificadores correctos e incorrectos :

Correctos	*Incorrectos*
contador	1contador
prueba23	hola!titi
res_total	res..total

La longitud de un identificador puede variar entre uno y varios caracteres. El estándar ANSI establece que los identificadores pueden ser de cualquier longitud, pero que al menos los seis primeros caracteres del nombre deben ser significativos si es que el identificador va estar envuelto en el proceso de enlazado. En esos identificadores, denominados nombres externos, se incluyen los nombres de las funciones y de las variables globales que se comparten entre archivos. Si el identificador no va a ser usado en un proceso de enlazado externo, los primeros 31 caracteres han de ser significativos. Este tipo de identificador se denomina nombre interno.

Un nombre de identificador puede ser más largo que el número de caracteres significativos reconocibles por el compilador. Sin embargo, los caracteres que se pasen del límite serán ignorados. Por ejemplo, si un compilador reconoce 31 caracteres significativos, los dos identificadores siguientes serán considerados iguales :

```
los_identificadores_demasiado-largos-son_latosos
los_identificadores_demasiado-largos-son_latosos_de_usar
```

En C las minúsculas y las mayúsculas se tratan como distintas. Así, cuenta, Cuenta, CUENTA son tres identificadores distintos. En algunos entornos puede que ignore la caja en los nombres de las funciones y de las variables globales si el enlazador no es sensible a mayúsculas y minúsculas (aunque la mayoría de los entornos actuales soportan un enlazado sensible).

Un identificador no puede ser igual que una palabra clave o reservada de C y no debe tener el mismo nombre que alguna función ya sea escrita por el usuario o que se encuentre en la biblioteca de C.

II.1.3 Entrada y Salida de Datos

Este apartado tratará de la forma de leer y escribir sobre teclado y pantalla. Nos mostrará las funciones de la librería estándar de C que han sido diseñadas.

Es conveniente señalar que en C no hay funciones construidas e incorporadas para realizar operaciones de E/S. En su lugar estas funciones se encuentran en la librería estándar de C.

En C toda la E/S es orientada a caracter. Esto no sólo es para la lectura y escritura por teclado y pantalla, sino también para las funciones de archivos en disco, que se verá más adelante. En C uno puede leer y escribir bytes. Las funciones *gets()* y *getnum()* permiten escribir funciones que leen cadenas y número, pero estas funciones todavía usan llamadas a funciones de E/S orientadas a caracter.

E/S por Teclado y Pantalla

Las funciones de E/S más sencillas son *getchar()*, que lee un caracter desde la entrada estándar (normalmente el teclado), y *putchar()*, que imprime un caracter por la salida estándar (normalmente la pantalla). La función *getchar()* espera a que se pulse una tecla y después devuelve su valor. Normalmente, *getchar()* también enviará el "eco" a la pantalla, del correspondiente caracter pulsado. Esto significa que el caracter que se pulsa en el teclado quedará escrito en la pantalla sin necesidad de indicarlo específicamente.

La función *putchar()* escribirá el argumento -un caracter- en la pantalla, pero sólo si ese argumento es parte del conjunto de caracteres que se pueda visualizar.

El siguiente programa tomará caracteres desde el teclado y los imprimirá en la pantalla, cambiando las mayúsculas por minúsculas y viceversa. El programa se detiene cuando se pulsa un punto.

La función *islower()* devuelve cierto si *ch* es un caracter en minúscula. La función *toupper()* convertirá una letra minúscula en mayúscula, y la función *tolower()* convertirá una letra mayúscula en minúscula. Estas funciones no afectarán a los caracteres no alfabéticos como + o ?. Estas funciones se encuentran en la librería estándar.

```
main() /* cambia tipo de letra */
        {
                char ch;
                do {
                        ch=getchar();
                        if (islower(ch)) putchar(toupper(ch));
                        else putchar(tolower(ch));
                } while (ch='.'); /* detenerse /*
        }
```

Las funciones gets() y puts()

El siguiente paso en cuanto a complejidad y potencia son las funciones *gets()* y *puts()*. Permiten leer cadenas de caracteres por teclado.

La función *gets()* devuelve una cadena terminada con un nulo en el arreglo de caracteres que recibe como argumento. Esto significa que cuando se utiliza *gets()* se pueden teclear caracteres en el teclado hasta que se pulse un retorno de carro. Al pulsar un retorno de carro, se coloca un nulo como terminador al final de la cadena y *gets()* devuelve el valor. El retorno de carro en sí no va en la cadena, y es imposible usar *gets()* para devolver un retorno de carro; sin embargo *getchar()* sí lo podrá hacer. *gets()* le permite corregir errores usando la tecla del espaciador -BACKSPACE- antes de pulsar RETURN.
La función *puts()* escribe su argumento de cadena, sobre la pantalla. *puts()* reconoce los mismos códigos de barra invertida que *printf()*, como \n para cambio de línea. Una llamada a *puts()* produce menos sobrecarga que la misma llamada a printf(), porque *puts()* sólo saca por pantalla una cadena de caracteres. No puede sacar números o hacer conversiones de

formatos. Por lo tanto, *puts()* ocupa menos espacio y funciona más rápido que *printf()* cuando muestra cadenas en la pantalla. Aunque algunos compiladores han omitido esta función, *puts()* está en la mayoría de las librerías estándar.

La función *puts()* es importante cuando el tamaño del código es importante. Si un programa no necesita de todas las funciones de *printf()* es ventajoso el no utilizar una función grande como *printf()* cuando se pueda utilizar la función más sencilla *puts()*.

Las funciones más sencillas que realizan operaciones de E/S de datos, se concentran a continuación :

Función	*Operación*
getchar()	*Lee un caracter desde el teclado*
putchar()	*Escribe un caracter en la pantalla*
gets()	*Lee una cadena desde el teclado*
puts()	*Escribe una cadena en la pantalla*

E/S Formateada por Teclado y Pantalla

Además de las funciones sencillas de E/S por consola descritas anteriormente, la librería estándar de C contiene dos funciones que realizan salida y entrada formateada: *printf()* y *scanf()*. La E/S formateada se refiere al hecho de que con estas funciones se puede formatear la información. Debemos recordar que las rutinas descritas anteriormente sólo introducirán y mostrarán en pantalla los datos en forma de filas.

Tanto *printf()* como *scanf()* permiten la mezcla y utilización de formatos de datos; por ejemplo, especificadores de campo y puntos decimales.

La Función printf()

El formato general de *printf()* es :

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

La cadena de control consta de dos tipos de elementos. El primer tipo está formado por caracteres que se imprimirán en la pantalla. El segundo tipo contiene comandos de formato que definen la forma en que se mostrarán en pantalla los argumentos siguientes. Debe de haber exactamente el mismo número de comandos de formato que de argumentos, y los comandos de formatos y los argumentos se aparean por orden.

A continuación se muestran los formatos permitidos para los datos en pantalla :

Código de printf()	*Formato*
%c	*Un único caracter*
%d	*Decimal*
%e	*Notación Científica*
%f	*Coma flotante decimal*
%g	*Utiliza %e o %f, la más corta*
%o	*Octal*
%s	*Cadena de caracteres*
%u	*Decimal sin signo*
%x	*Hexadecimal*

Los códigos de control de formato pueden tener modificadores que especifiquen la anchura de campo, el número de decimales, y un indicador de ajuste a la izquierda. Un entero situado entre signo *%* y el comando de formateo actúa como un *especificador de anchura mínima de campo*. Este rellena la salida con espacios en blanco o ceros para asegurarse de que al menos tiene una cierta longitud mínima. Si la cadena o el número son mayores que el mínimo, se imprimirá entero aunque sobrepase el mínimo. El relleno se hace con espacios en blanco, salvo que se indique lo contrario. Si se requiere rellenar con ceros se coloca un *0*

delante del especificador de anchura de campo. Por ejemplo *%05d* rellenará con 0's los números de menos de 5 dígitos.

Para especificar el número de decimales a imprimir en un número en coma flotante, se coloca la coma decimal seguida por el número de decimales que se desean mostrar en pantalla después del especificador de anchura de campo. Por ejemplo, *%10.4f* mostrará en pantalla un número de al menos 10 caracteres con cuatro decimales. Este método también funciona cuando se requiere especificar la longitud de campo máxima para cadenas y valores enteros. Por ejemplo, *%5.7s* mostrará en pantalla una cadena que tendrá al menos cinco caracteres pero que no excederá de siete. Si la cadena es más larga que la anchura de campo máxima, los caracteres del final se truncarán.

Por defecto, todas las salidas se ajustan a la derecha: si la anchura de campo es más larga que la del dato a imprimir, el dato se ajustará a la derecha del campo. Se puede obligar a que la información se muestre en pantalla a la izquierda colocando un signo menos justo después del *%*. Por ejemplo, *%-10.2f* ajustará a la izquierda un número en coma flotante con dos decimales colocándolos en una anchura de campo de 10. El modificador *l* le indica a *printf()* que lo que sigue es un dato del tipo *long int*. Con *printf()*, prácticamente se pueden sacar los datos con cualquier formato de dato, como se puede ver en los ejemplos siguientes :

Sentencia printf()	***Salida***
("%-5.2f",123.234)	*|123.23 |*
(5.2f",3.234)	*| 3.23|*
("10s","hola")	*| hola|*
("-10s","hola")	*|hola |*
("%5.7s,"123456789")	*| 1234567|*

La Función scanf()

La función *scanf()*, es una rutina de entrada de propósito general, que permite leer datos formateados y convertir automáticamente la información numérica a enteros o a flotantes por ejemplo. Es muy parecida a la inversa de *printf()*. El formato general de *scanf()* es :

scanf(cadena de control, lista de argumentos);

La cadena de control está formada por códigos de formatos de entrada, que están precedidas por un signo %. Estos códigos se listan a continuación :

Código	*Significado*
c	*Lee un único caracter*
d	*Lee un entero decimal*
e	*Lee un número en notación científica*
f	*Lee un número en coma flotante*
h	*Lee un entero corto*
o	*Lee un entero octal*
s	*Lee una cadena*
x	*Lee un número hexadecimal*

Los comandos de formato pueden utilizar modificadores de la longitud de campo y son números enteros colocados entre % y el código de comando de formato. Un * colocado después del % suprimirá la asignación y avanzará a la siguiente entrada del campo. Cualquier otro caracter en la cadena de control será apareado y desechado.

Los datos de entrada tienen que estar separados por espacios en blanco, tabuladores o cambios de línea. Los signos de puntuación como las comas o puntos y comas no sirven como separadores. Al igual que en *printf()* los códigos de formato de *scanf()* se aparean por orden con las variables que reciben la entrada.

Todas las variables utilizadas para recibir valores a través de *scanf()* se pasan por sus direcciones. Esto significa que todos los argumentos, distintos de los de la cadena de control, tienen que apuntar a las variables que recibirán la entrada. Esta es la forma en que C crea una "llamada por referencia". Por ejemplo, si se quiere leer un entero dentro de la variable *cuenta*, se utilizará esta llamada a *scanf()* :

scanf("%d",&cuenta);

Las cadenas se leerán sobre arreglos de caracteres, y el nombre del arreglo, sin ningún índice, es la dirección del primer elemento del arreglo. En consecuencia, para leer una cadena dentro del arreglo de caracteres *direccion*, se utilizaría :

scanf("%s",direccion);

En este caso *direccion* ya es un puntero y no es necesario que vaya precedido por el operador *&*.

El *modificador de longitud máxima de campo* se puede aplicar a los códigos de formato. Si no se desean leer más de 20 caracteres en *direccion*, se escribirá :

scanf("%20s",direccion);

Si el flujo de entrada fuese mayor de 20 caracteres, habría que hacer una segunda llamada de entrada desde donde se dejó la anterior. Por ejemplo, si :

Avenida Parque del Retiro 28

se hubiese introducido como respuesta a la llamada de *scanf()*, sólo se habrían colocado en *direccion* los 20 primeros caracteres, o hasta *R* de *Retiro*, debido al espaciador de máximo

tamaño. Esto significa que los 8 caracteres restantes, *etiro 28*, todavía no se han utilizado. Si se hace otra llamada a *scanf()*, tal como

scanf("%s",cdn);

entonces *etiro 28*, se colocaría en *cdn*. Sin embrago, muchos sistemas operativos de microcomputadoras se limitarían a perder los caracteres que se hubiesen tecleado y no se hubiesen asignado. Sólo en el caso de que el sistema soporte E/S intermedia, esos caracteres permanecerán para su procesamiento.

Aunque los espacios en blanco, tabuladores y los cambios de línea se utilizan como separadores de campo, cuando se va a leer un único caracter, se leen como otro caracter cualquiera. Por ejemplo, con un flujo de entrada x y.

scanf("%c%c%c",&a,&b,&c);

se devolverá el caracter *x* en *a*, un espacio en blanco en *b*, y el caracter *y* en *c*. Si se tiene cualquier otro caracter en la cadena de control incluyendo espacios en blanco, tabuladores y cambios de línea, estos caracteres se utilizarán para comprobarlos con los caracteres del flujo de entrada. Cualquier otro caracter que se aparee con ellos, se descartará. Por ejemplo, el flujo de entrada *abcdttttttefg*,

scanf("%st%s",nombre1,nombre2);

colocará los caracteres *abcd* en *nombre1* y los caracteres *efg* en *nombre2*. Las *tes* se han descartado debido a la *t* de la cadena de control. En otro ejemplo,

scanf("%s ",nombre);

no volverá hasta que se teclee un caracter después de teclear un terminador: el espacio después de *%s* indica a *scanf()* que lea y descarte los espacios en blanco, tabuladores y caracteres de cambio de línea.

La cadena de control no se puede utilizar para la salida de caracteres como en *printf()*. Por lo tanto, todas las peticiones de entrada tienen que hacerse explícitamente antes de la llamada a *scanf()*. La posibilidad de *scanf()* para procesar varios tipos de datos se utiliza frecuentemente en programas de bases de datos.

II.1.4 Caracteres y Cadenas

Caracteres

Otro tipo de datos muy importante en C es el *char*, que representa los caracteres. Un caracter es un valor de un byte que se puede utilizar para almacenar caracteres imprimibles o enteros pertenecientes al intervalo de 0 a 255. Las constantes de tipo caracter se encierran entre comillas sencillas. Por ejemplo, este programa imprime la cadena "ABC", en la pantalla.

```
/* Programa sencillo que utiliza caracteres */
main()
        {
        char car;
        car = 'A';
        printf("%c",car);
        car = 'B';
        printf("%c",car);
        car = 'C';
        printf("%c",car);
        }
```

Aunque es posible utilizar *scanf()* para leer un sólo caracter del teclado, una manera más usual es utilizar la función *getchar()*.

Cadenas

El uso más común de los arreglos unidimensionales, con mucho, es como cadenas de caracteres. En C una cadena se define como un arreglo de caracteres que termina en un caracter nulo. Un caracter nulo se especifica como '\0', y generalmente es un cero. Por esta razón, para declarar arreglos de caracteres es necesario que sean de un caracter más que la cadena más larga que pueda contener. Por ejemplo, para declarar un arreglo *s*, que contenga una cadena de 10 caracteres, se escribiría :

char s(11);

Esto dejará sitio para el caracter nulo del final de la cadena. Aunque C no define un tipo de datos de cadenas, permite disponer de constantes de cadena. Una constante de cadena es una lista de caracteres encerrada entre dobles comillas. Por ejemplo,

"hola, que tal"

No es necesario añadir explícitamente el caracter nulo al final de las constantes de cadena, el compilador de C lo hace automáticamente.

C soporta una gran variedad de funciones de manejo de cadenas. Las más comunes son :

Nombre	*Función*
strcpy(c1,c2)	*Copia c2 en c1*
strcat(c1,c2)	*Concatena c2 al final de c1*
strlen(c1)	*Devuelve la longitud de c1*
strcmp(c1,c2)	*Devuelve 0 si c1 y c2 son iguales; menor que 0 si c1<c2; mayor que 0 si c1>c2*
strchr(c1,car)	*Devuelve un puntero a la primera ocurrencia de car en c1*
strstr(c1,c2)	*Devuelve un puntero a la primera ocurrencia de c2 en c1*

Estas funciones usan el archivo de cabecera estándar STRING.H. El siguiente programa ilustra el uso de las funciones de cadenas :

```
#include "stdio.h"
#include "string.h"
void main(void)
        {
        char c1(80),c2(80);
        gets(c1); gets(c2);
        printf("longitudes: %d %d\n",strlen(c1),strlen(c2);
        if(!strcmp(c1,c2)) printf("Las cadenas son iguales\n");
                strcat(c1,c1);
                printf("%s\n",c1");
                strcpy(c1, "Esta es una prueba.\n");
                printf(c1);
        if(strchr("hola", 'o')) printf("0 está en hola\n");
        if(strstr("hola que tal", "hola"));
                printf("hola encontrado");
        }
```

Si se ejecuta este programa y se introducen las cadenas "hola" y "hola que tal", la salida es :

```
longitudes : 5 5
Las cadenas son iguales
holahola que tal
Esta es una prueba
o está en hola
hola encontrado
```

II.1.5 Constantes y Comentarios

Constantes

Las constantes en C se refieren a valores fijos que no pueden estar alterados por el programa. Pueden ser cualquier tipo de datos básicos. La forma en que se represente cada

constante depende de su tipo. Las constantes de caracter van encerradas entre comillas simples. Por ejemplo, *'a' y '%'* son ambas constantes de caracter.

Las constantes enteras se especifican como números sin parte fraccional. Por ejemplo, *10* y *-100* son constantes enteras. Las constantes en coma flotante requieren el punto decimal seguido de los números de la componente fraccional. Por ejemplo, *11.123* es una constante en coma flotante. C también permite el uso de la notación científica para los números en coma flotante.

Existen dos tipos en coma flotante: *float* y *double*. Asimismo, existen distintas variaciones de los tipos básicos que se pueden generar usando los modificadores de tipo. Por omisión, el compilador de C escoge el tipo de dato compatible más pequeño que pueda albergar a una constante numérica. Por lo tanto, *10* es de tipo *int* por omisión, pero *60.000* es un *unsigned* y *100.000* es *long*. Incluso aunque el valor *10* podría caber en una tipo caracter, el compilador no sobrepasa los límites entre tipos. La única excepción a la regla de selección del tipo más pequeño la constituyen las constantes en coma flotante, que se asumen de tipo *double*.

Para la mayoría de los programas que se escriben, los tipos implícitos que selecciona el compilador son adecuados. Sin embargo, se puede especificar de forma precisa el tipo de una constante numérica usando un sufijo. Para los tipos en coma flotante si se sigue el número con una *F*, se trata el número como *float*. Si se sigue con una *L*, el número será *long double*. Para los tipos enteros, el sufijo *U*, significa *unsigned* y la *L long*. Algunos ejemplos se muestran a continuación :

Tipo de Dato	*Ejemplos de Constantes*
int	*1 123 21000 -234*
long int	*35000L -34L*
short int	*10 -12 90*
unsigned int	*10000U 987U 40000*

Tipo de Dato	*Ejemplos de Constantes*
float	*123.23F -1.34e-3F*
double	*123.23 12312333 -0.9876324*
long double	*1001.2L*

Constantes Hexadecimales y Octales

A veces es más cómodo usar un número en base 8 o 16 que en base 10. El sistema de numeración en base 8 usa los dígitos del 0 al 7. En octal, el número 10 equivale al 8 en decimal. El sistema de numeración en base 16 se denomina hexadecimal y usa los dígitos del 0 al 9 y las letras de la A a la F, que representan respectivamente, los valores 10,11,12,13,14 y 15. Por ejemplo, el número hexadecimal 10 equivale al 16 decimal. C permite especificar constantes enteras en hexadecimal o en octal en lugar de en decimal. Una constante hexadecimal consiste en Ox seguido de la constante en forma hexadecimal. Una constante octal comienza por O. Ejemplos :

int hex = Ox80; / 128 en decimal */ int oct = O12; /* 10 en decimal */*

Constantes de Cadena

C soporta otro tipo de constantes: la cadena. Una cadena es una secuencia de caracteres encerrados entre dobles comillas. Por ejemplo, "esto es una prueba" es una cadena. Aunque C permite definir consiantes de cadena, no tiene formalmente un tipo de datos cadena.

Constantes de Caracter con Barra Invertida

El incluir entre comillas simples las constantes de caracter es suficiente para la mayoría de los caracteres imprimibles. Pero unos pocos, como el retorno de carro, son imposibles de introducir desde el teclado. Por esta razón, C incluye las constantes especiales de caracter con barra invertida.

C soporta varios códigos especiales de barra invertida como se muestra a continuación, para que se puedan introducir fácilmente estos caracteres especiales como constantes. Se deben usar los códigos de barra invertida en lugar de sus equivalentes ASCII con el fin de asegurar la portabilidad del código.

Código	*Significado*
\b	*Espacio atrás*
\f	*Salto de página*
\n	*Salto de linea*
\r	*Salto de carro*
\t	*Tabulación horizontal*
\"	*Comillas dobles*
\'	*Comilla simple*
\0	*Nulo*
**	*Barra invertida*
\v	*Tabulador vertical*
\a	*Alerta*
\N	*Constante octal (donde N es una constante octal)*
\xN	*Constante hexadecimal (donde N es una constante hexadecimal)*

Comentarios

Los comentarios en C se pueden colocar en cualquier parte en un programa, y deben de ir entre dos marcas. La marca de comienzo es /* y la de fin de comentario es */.

Toda oración, caracter o signo raro que se encuentre entre dichas marcas no será tomado en cuenta para la realización del programa y sólo ayudará a la comprensión del mismo.

II.2 ESTRUCTURAS DE CONTROL DE PROGRAMA

En este apartado se presenta el conjunto de sentencias de control de programa que C incorpora. El estándar ANSI clasifica las sentencias de C en los siguientes grupos : Arreglos, Relación y Lógicos, Iteración, Condición y Selección.

II.2.1 Arreglos (Arrays)

Un arreglo es una colección de variables del mismo tipo que se referencían por un nombre común. A un elemento específico de un arreglo se accede mediante un índice. En C todos los arreglos constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento. Los arreglos pueden tener de una a varias dimensiones. El arreglo más común en C es la cadena, que simplemente es un arreglo de caracteres terminado por un nulo. Este enfoque de las cadenas de C, le da una mayor potencia y una mayor eficiencia que otros lenguajes.

En C, los arreglos y los apuntadores están íntimamente relacionados; una explicación de uno de ellos siempre se refiere a los otros.

Arreglos Uni-dimensionales

Los arreglos uni-dimensionales son simplemente listas con información del mismo tipo. El ejemplo más común de arreglos uni-dimensionales es la cadena de caracteres. Cuando se declara un arreglo de caracteres como para contener una cadena debe de preverse un byte extra para el caracter nulo de terminador, que tienen todas las cadenas. Por ejemplo, si se quiere declarar un arreglo *s* para una cadena de *10* caracteres se escribiría :

```
char s[11];
```

Esto deja espacio para el terminador nulo al final de la cadena.

Cuando se pasan a funciones de arreglos uni-dimensionales, se llama a la función con el nombre del arreglo. Por ejemplo, para pasar el arreglo *s* a la función *gets()*, se escribiría :

```
gets(s);
```

Esto haría que la dirección del primer elemento de *s* pasase a *gets()*. Realmente se esta pasando un apuntador a caracter.

Si una función va a recibir un arreglo uni-dimensional, hay que declararlo como parámetro formal en una de estas dos formas : como un apuntador o como un arreglo.

Por ejemplo, para recibir *s* dentro de una función denominada *func1()*, se declararía *func1()* como :

```
func1(cad)
char*cad;
{
        .
        .
        .
```

o como

```
func1(cad)
char cad[];
{
        .
        .
        .
```

Ambos métodos de declaración son idénticos porque ambos le indican al compilador que se va a recibir un apuntador a caracter.

Realmente, por lo que respecta al compilador,

```
func1(cad)
char cad[11];
{
        .
        .
        .
```

también valdría porque C generará un código que indica a *func10* que va a recibir un apuntador a caracter. Como no hay comprobación de límites, al llamar a la rutina se puede pasar un arreglo de cualquier tamaño, incluso aunque sólo se hubiese declarado un tamaño de *11*.

Arreglos Bi-dimensionales

C permite arreglos multidimensionales. La forma más simple del arreglo multidimensional es el arreglo bi-dimensional. En esencia un arreglo bi-dimensional es un arreglo uni-dimensional. Para declarar un arreglo entero bi-dimensional *d* de tamaño *10,20* se escribiría: :

int d[10][20];

Hay que poner especial cuidado en la declaración (diferencia de la mayoría de los demás lenguajes que utilizan comas para separar las dimensiones de arreglo), C coloca cada dimensión entre paréntesis rectos.

Igualmente, para acceder al elemento *3,5* del arreglo *d*, se utilizaría :

d[3][5]

Haciendo una analogía con el programa en lenguaje BASIC del siguiente programa . En ambas versiones BASIC y C un arreglo bi-dimensional se cargará con los números del *1* al *12*.

En la versión en C, *num[0][0]* tendrá el valor de *1*, *num[0][1]* el valor de *2*, *num[0][2]* el valor de *3*, y así sucesivamente. El valor de *num[2][3]* será *12*.

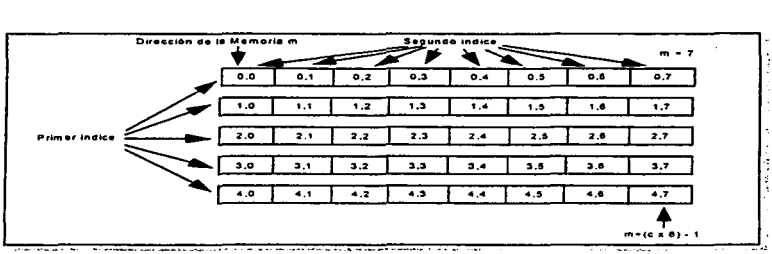
Los arreglos bi-dimensionales se almacenan en una matriz de filas y columnas, en donde el primer índice indica la fila y el segundo la columna. Esto significa que el índice más a la derecha cambia más rápidamente que el que está más a la izquierda cuando se recorre el arreglo en secuencia.

En el esquema se muestra la representación en memoria de un arreglo bi-dimensional. En esencia, el primer índice es como si fuera un puntero apuntando a la fila correcta.

```
main()
{
        int t,i,num[3][4];
        for(t=0;t<3;++t);
                for(i=0;i<4;++i);
                        num[t][i]=(t*4)+i+1;
}
```

```
10 DIM N(3,4)
20 FOR T=1 TO 3
30      FOR I=1 TO 4
40              N(T,I)=((T*4)+I+1)
50      NEXT
60 NEXT
```

Las versiones en C y en Basic de un programa que carga un arreglo bi-dimensional



Hay que recordar que el almacenamiento para todos los elementos está asignado permanentemente. En el caso de un arreglo bi-dimensional, la siguiente fórmula determinará el número de bytes en la memoria :

*bytes = fila * columna * número de bytes según el tipo de dato*

Por lo tanto, un arreglo entero con dimensiones *10,5* tendrá *10x5x2* ó 100 bytes asignados. También debemos recordar que los arreglos grandes utilizan mucha memoria; por lo tanto hay que procurar al declararlo que sea lo suficientemente largo para las necesidades, pero sin excederse demasiado.

Cuando se pasan arreglos bi-dimensionales a las funciones, sólo se pasa un apuntador al primer elemento. Esto se puede hacer utilizando el nombre del arreglo sin índices. Sin embargo, una función que reciba un arreglo bi-dimensional como parámetro tiene que definir la longitud de la segunda dimensión. Por ejemplo, una función que recibe un arreglo entero bi-dimensional con las dimensiones 10,10 se declararía como :

```
func1(x)
int x[][10];
    {
    .
    .
    .
    }
```

También sería posible dar la primera dimensión, pero no es necesario.

En C se necesita conocer la segunda dimensión para trabajar en sentencias tales como *x[2][4]* dentro de la función. Si no se conocen la longitud de las filas, sería imposible saber dónde empieza la tercera fila.

Arreglos Multidimensionales

C permite arreglos de más de dos dimensiones. El límite exacto, si hay, vendrá determinado por el compilador. El formato general de una declaración de un arreglo multidimensional es:

type nombre [a][b][c]...[z];

Los arreglos de tres o más dimensiones a menudo no se utilizan debido a la cantidad de memoria que se necesita para mantenerlos. Como se indicó anteriormente, el almacenamiento de todos los elementos del arreglo está asignado permanentemente durante la ejecución del programa. Por ejemplo, un arreglo de caracteres 4-dimensional de las dimensiones 10,6,9,4 necesitaría 10x6x9x4, ó 2,160 bytes. Si el arreglo fuese de enteros, estarían asignados 4,320 bytes. Si el arreglo fuese *double float*, entonces se necesitarían 34,560 bytes. El almacenamiento necesario aumenta exponencialmente con el número de dimensiones. La computadora tarda tiempo en generar cada índice y esto puede hacer que los arreglos multidimensionales accedan más despacio que los arreglos uni-dimesionales

con el mismo número de elementos. Por estas y otras razones, cuando son necesarios arreglos multidimensionales grandes, se deben asignar dinámicamente los bits y las partes del arreglo que sean necesarios, y utilizar apuntadores. Sin embargo, este procedimiento denominado procesamiento de arreglos esparcidos, no se tratará en este Trabajo de Tesis.

Cuando se pasan a funciones arreglos multidimensionales, hay que declarar todas las dimensiones excepto la primera. Por ejemplo, si se declarará el arreglo *m* como :

int m[4][3][6][5];

entonces una función, *func1()*, que recibiera *m*, empezaría :

```
func1(d)
int d[][3][6][5];
        {
        .
        .
        .
        }
```

Ubicación de los Arreglos

En C se puede asignar y liberar memoria dinámicamente utilizando rutinas de la librería estándar *malloc()* y *free()*. Si la memoria es limitada y se necesita un arreglo por poco tiempo, se puede asignar utilizando *malloc()* y devolverlo como memoria libre utilizando *free()* cuando se ha acabado.

Este fragmento asigna 1000 bytes de memoria :

```
charp *p;
p = malloc(1000); /* toma 1000 bytes */
```

La *p* apunta al primero de los 1000 bytes de memoria libre. Si se necesita utilizar esa memoria como un arreglo bi-dimensional *10,10* para realizar el procesamiento del arreglo, se podría declarar la función similar a esta :

```
proceso(s)
char s[][10];
{       /* proceso del arreglo */
}
```

Esto complica al compilador C al simular un arreglo de caracteres de *10* por *10*. Realmente, se tiene un arreglo de caracteres de *10* por *10* dentro de la función; la diferencia es que la asignación se ejecuta manualmente utilizando la sentencia *malloc()*, en vez de automáticamente utilizando la declaración de sentencia del arreglo normal.

La secuencia esencial de asignación, procesamiento y liberación se muestra en el siguiente programa abstracto. Este asignará memoria para un arreglo de caracteres de *10* por *10*, el apuntador a esa memoria se pasa a la función denominada *proceso()*, y libera la memoria cuando vuelve el proceso :

```
main()
{
        char *p;
        .
        p = malloc(1000);
        proceso(p)
        free(p)
        .
}
proceso(ax)
char ax[10][10];
{
.
/* ax se puede usar como arreglo
normal bi-dimensional de caracteres */
.
}
```

Debido a que una codificación de esta naturaleza puede inducir a error y confundir a alguien que lea el programa, esto sólo se debe hacer cuando la memoria es escasa. Cuando sea posible, es mejor el declarar los arreglos explícitamente. Sin embargo, la utilización de *malloc(), free()*, y de la asignación dinámica en general es realmente aceptable.

II.2.2 Operadores Aritméticos, de Relación y Lógicos

El lenguaje de programación C es rico en operadores. Un operador es un símbolo que le dice al compilador que realice manipulaciones matemáticas o lógicas específicas. C tiene tres clases generales de operadores : aritméticos, relacionales y lógicos y sobre bits, este último será tratado al final de este capítulo.

Operadores Aritméticos

La tabla siguiente lista los *operadores aritméticos*. Los operadores +, -, * y / funcionan de la misma manera en C que en la mayoría de los lenguajes de computadora. Cuando se aplica / a un entero o caracter, la computadora truncará cualquier resto : por ejemplo. 10/3 será igual a 3 en la división entera.

El operador módulo de la división % recoge el resto de una división entera. Sin embargo, no se puede usar % sobre los tipos *float* o *double*.

Operador	*Acción*
-	*Resta y menos unitario*
+	*Suma*
*	*Multiplicación*
/	*División*
%	*Módulo División*
--	*Decremento*
++	*Incremento*

El programa siguiente ilustra los operadores *división* / y *módulo* % :

```
main()
{
        int x,y;
        x = 10;
        y = 3;
        printf("%d", x/y); /* visualizará 3 */
        printf("%d", x%y); /* visualizará 1, el resto
        de la división entera */
        x = 1;
        y = 2;
        printf("%d %d", x/y, x%y); */ visualizará 0 y 1 */
}
```

La razón de que la última línea imprima 0 y 1 es que la división 1/2 es 0 con un resto de 1.

El menos unitario multiplica su operando por -1. Así, cualquier número precedido por el signo menos cambia el signo del número.

Incremento y Decremento

C permite dos operadores utilísimos que no se encuentran en otros lenguajes de programación. Estos son los operadores incremento y decremento ++ y --. El operador ++ añade 1 a su operando y el -- le resta uno. Por tanto, los siguientes son operaciones equivalentes :

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

es lo mismo que

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

Los operadores incremento y decremento pueden preceder o seguir al operando. Por ejemplo, se puede escribir :

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

o como `++x;` *o* `x++;`

Sin embargo, hay diferencia cuando se usa en una expresión. Cuando un operador incremento o decremento precede a su operando, C realiza la operación de incremento o decremento antes de usar el valor del operando. Cuando el operador sigue al operando, C usa el valor del operando antes de incrementar o decrementar. Por ejemplo :

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

En este caso, C pone y a *11* porque C incrementa primero x y después lo asigna a y. Sin embargo, si el código ha sido escrito como :

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

C establecería y a *10* y después incrementaría x. En ambos casos x estaría a *11*; la diferencia es cuando se hace.

Esta es la precedencia de los operadores aritméticos :

Más alta	++ --
	* / %
Más baja	+ -

La computadora evalúa de izquierda a derecha los operadores con el mismo nivel de precedencia. Por supuesto, se pueden usar paréntesis para alterar el orden de la evaluación.

El lenguaje C trata los paréntesis de la misma manera que prácticamente todos los lenguajes: fuerzan una operación, o un conjunto de ellas a tener un nivel de precedencia mayor.

Operadores Relacionales y Lógicos

En los términos operador relacional y lógico, relacional se refiere a las relaciones que los valores pueden tener con otros, y lógico se refiere a la manera en que tienen lugar estas relaciones. La clave de los conceptos de operadores relacional y lógico es la idea de verdadero y falso. En C, verdadero es cualquier valor distinto de cero, mientras que falso es cero. Las expresiones que usan operadores relacionales y lógicos devolverán 0 para falso y 1 para verdadero. El resumen siguiente muestra los operadores relacionales y lógicos :

OPERADORES RELACIONALES

Operador	*Acción*
>	*Mayor que*
>=	*Mayor o igual que*
<	*Menor que*
<=	*Menor o igual que*
==	*Igual*
!=	*No igual*

OPERADORES LÓGICOS

Operador	*Acción*
&&	*AND*
||	*OR*
!	*NOT*

Se usan los operadores relacionales para determinar las relaciones de una cantidad con otra. Siempre devuelven 1 ó 0, dependiendo del resultado de la prueba. El programa siguiente ilustra el resultado de cada operación y visualiza el resultado de cada operación como 0 ó 1:

```
/* Este programa ilustra los operadores relacionales */
main()
        {
                int i, j;
                printf("Introducir dos numeros : ");
                scanf("%d%d", &i, &j);
                printf("%d == %d es %d\n", i,j, i==j);
                printf("%d != %d es %d\n", i,j, i!=j);
                printf("%d <= %d es %d\n", i,j, i<=j);
                printf("%d >= %d es %d\n", i,j, i>=j);
                printf("%d < %d es %d\n", i,j, i<j);
                printf("%d > %d es %d\n", i,j, i>j);
        }
```

Se puede aplicar los operadores relacionales a cualquier tipo básico de datos. Por ejemplo, lo siguiente visualiza el mensaje *"mayor que"* porque una *B* es mayor que una *A*.

```
ch1 = 'A';
ch2 = 'B';
if (ch2>ch1) printf("mayor que");
```

Los operadores lógicos se usan para soportar las operaciones básicas lógicas de AND, OR, NOT de acuerdo con la siguiente tabla que usa 1 para verdad y 0 para falso.

p	q	pANDq	pORq	NOTp
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

TABLA DE VERDAD

Este programa ilustra el funcionamiento de los operadores lógicos :

```
/* Ilustra los operadores lógicos */
main()
        {
                int i,j;
                printf("introducir dos números 0 ó 1 : ");
                scanf("%d%d, &i, &j);
                printf("%d AND %d es %d\n", i,j, i&&j);
                printf("%d OR %d es %d\n", i,j, i \\j);
                printf("%d NOT %d es %d\n", i, i, !j);
        }
```

Los operadores lógicos tienen menor precedencia que los aritméticos. Esto significa que C evalúa una expresión como 10>1+12 como si se escribiera 10>(1+12). El resultado de la expresión es falso.

Lo siguiente muestra la precedencia relativa de los operadores relacionales y lógicos :

Más alto	*!* *> >= < <=* *== !=* *&&* *||*
Más bajo	

Se usan los operadores relacionales y lógicos para soportar las sentencias de control del programa, que incluyen los bucles y sentencias *if*.

II.2.3 La Estructura FOR

La estructura *for* se utiliza cuando se quieren ejecutar las sentencias más de una vez. Las siguientes instrucciones comparan un programa C utilizando la estructura *for* y un programa BASIC utilizando *FOR-NEXT* :

```
10 FOR X=1 TO 100 STEP 1
20 PRINT "HOLA", X
30 NEXT
```

```
main()
{
        int x;
        for (x=1;x<=100;++x);
        printf("HOLA" %d",x);
}
```

Ambos programas imprimirán en la pantalla 100 veces la palabra *HOLA* y el valor de x. La estructura *FOR-NEXT* en BASIC, se ejecutará 100 veces las líneas de la 10 a la 30, aumentando cada vez *x* en 1. En el código C, *x* se inicializa a 1. Como *x* es menor que 100, se llama a *printf()*, *x* se aumenta en 1, y se comprueba si todavía *x* es menor o igual que 100. Este proceso se repite hasta que *x* sea mayor que 100, entonces, deja de llamar a *printf()* y el programa termina. En este ejemplo, *x* es la variable de control de la estructura, que se cambia y comprueba cada vez que se repite este.

El formato general de *for* para repetir una sentencia única es :

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

Para repetir un bloque, el formato general es

```
for(inicialización; condición; incremento)
{
        sentencia 1;
        .
        .
        .
        sentencia n;
}
```

La *inicialización* normalmente es una sentencia de asignación que se utiliza para establecer la variable de control de la estructura. La *condición* es una expresión relacional que determina cuando saldrá la estructura. El *incremento* define como cambiará la variable de

control de la estructura cada vez que se repita éste. Estas tres partes principales tienen que ir separadas por puntos y comas. La estructura *for* se continuará ejecutando mientras que las pruebas de ejecución sean ciertas. Cuando la *condición* sea falsa, la ejecución del programa se reanudará en la sentencia siguiente al bloque *for*.

II.2.4 La Estructura WHILE

Otra forma de construir un bucle es con la estructura *while*. El formato general de la sentencia es :

```
while(condición) sentencia;
```

en donde *sentencia* puede ser una sentencia simple o un bloque de sentencias que hay que repetir. La *condición* puede ser cualquier expresión, siendo cierto cualquier valor distinto de cero. La *sentencia* se ejecuta mientras que la *condición* sea cierta. Cuando la *condición* sea falsa, el control en el programa pasa a la línea siguiente después del código del bucle.

El siguiente ejemplo muestra una rutina que toma un caracter desde el teclado, y se limita a dar vueltas hasta que se pulse la letra *A*.

```
espera_ca()
{
        char ca;
        ca=0; /* inicializa ca */
        while(ca!='A') ca=getchar();
}
```

Se puede ver que primero *ca* se inicializó a 0. Igual que una variable local, su valor no es conocido cuando se ejecuta *espera_ca()*. Entonces el bucle *while* empieza comprobando si *ca* es distinto de *A*. Debido a que *ca* de antemano se inicializó a 0, la prueba es cierta y el bucle comienza. Cada vez que se pulsa una tecla se hace la prueba. Una vez pulsada la letra *A*, la condición se convierte en falsa porque *ca* es igual a *A*, y el bucle termina.

Igual que el bucle *for*, el bucle *while* comprueban la condición en la parte superior del bucle, lo que significa que el código del bucle puede que no se ejecute. Esto elimina el tener que ejecutar una prueba antes del bucle.

Al tener varias sentencias diferentes dentro de *while*, el que cada una pueda terminar el bucle es una práctica muy común. Un bucle *while* también puede tener sólo una variable como condición y se puede utilizar simplemente para repetir el conjunto de instrucciones hasta que se termine el procedimiento.

```
func1()
{
        int trabajo;
        trabajo=1
        while(trabajo)
                {
                        trabajo=proceso1();
                        if(trabajo)
                                trabajo=proceso2();
```

```
            if(trabajo)
                trabajo=proceso3();
        }
    }
```

Cualquiera de las tres rutinas puede devolver falso (0) y originar la salida del bucle. No hay ninguna necesidad de que haya sentencias en el cuerpo *while*. Por ejemplo,

```
while((ca=getchar()) != 'A')
```

simplemente se quedará en el bucle hasta que se teclee la letra *A*. Si no se encuentra cómodo con la asignación dentro de la condición while, debemos recordar que el signo = es realmente un operador en el sentido de que devuelve un valor : el valor del operador de asignación es el valor de la expresión situada a la derecha.

II.2.5 La Estructura DO-WHILE

A diferencia de los bucles *for* y *while* que prueban la condición de bucle al principio, el bucle *do while* comprueba su condición al final del bucle. Esto significa que un bucle *do-while* siempre se ejecutará una vez por lo menos. El formato general del bucle *do-while* es :

```
do {
        sentencias;
    } while(condición);
```

Aunque las llaves no son necesarias cuando sólo hay una sentencia, normalmente se utilizan para mejorar la legibilidad de la construcción *do-while*.
A continuación un ejemplo sencillo:

```
do {
        num=getnum();
    } while(num > 100);
```

Los números se leerán desde el teclado hasta que se encuentre un número menor que 100.

Quizá la utilización más común de *do-while* sea una rutina de selección por menú en la que el usuario hace una selección de un menú que se muestra en la pantalla. Cuando se teclea una respuesta válida, se devuelve como el valor de la función. Las respuestas no válidas serán la causa de que el programa presente otra vez una petición de entrada.

Lo que sigue es un menú para ejemplificar lo citado anteriormente.

```
menu()
{
        char ca;
        printf("1. Altas\n");
        printf("2. Bajas\n");
        printf("3. Modificaciones\n");
        printf("Pulsar cualquier tecla para salir\n");
        printf("Introduzca su selección: ");
        do {
        ca=getchar(); /* lee selección */
        switch(ca)      {
                        case '1':
                        Altas();
                        break;
                        case '2':
                        Bajas();
                        break;
                        case '3':
                        Modificaciones();
                        break;
                        }
          }while(ca!='1' && ca!='2' && ca!='3');
}
```

En el caso de un menú de funciones, siempre se querrá que por lo menos se ejecute una vez. Después de que las opciones se han mostrado en pantalla, el programa entrará en el bucle hasta que se seleccione una opción válida.

II.2.6 La Estructura IF-ELSE

El formato general de la sentencia *if* es :

```
if(prueba de condición)sentencia1
    else sentencia2
```

en donde los objetos de *if* y *else* son sentencias simples. La sentencia *else* es opcional. Los objetos tanto de *if* como de *else* pueden ser bloques de sentencias.

El formato general de *if* con bloques de sentencias como objeto es :

```
if(prueba de condición)
{
        sentencias /* bloque 1 */
}
else
        sentencias /* bloque 2 */
}
```

Si la condición es cierta (es decir, cualquier valor distinto de 0), *sentencia 1* o *bloque 1* se ejecutarán, si no, *sentencia 2* o *bloque 2*, si es que existen. Debemos recordar que sólo se ejecutará una sentencia o un bloque, nunca ambos.

A continuación se muestra un programa en el que hay que adivinar un número, si se adivina imprimirá el mensaje de ** *correcto* ** y si no se adivina el programa proporcionará una pista para saber su aproximación.

```
main()          /* Programa del número mágico */
{
        int magico = 123; /* número mágico */
        int adivinado;
        adivinado = getnum(); /* lee un número entero */
```

```
        if (adivinado = magico)
        {
                printf("** correcto **");
                printf("%d es el número mágico", magico);
        }
        else
        {
                printf(".. Equivocado ..");
                if(adivinado > magico) printf("Demasiado Alto");
                printf("Demasido Bajo");
        }
}
```

La versión del programa anterior utiliza como objetos de las sentencias *if* y *else* bloques de sentencias. Uno de los aspectos más confusos de las sentencias *if* en cualquier lenguaje de programación son los *if's anidados*. Un *if anidado* es una sentencia *if* que es el objeto o bien de un *if* o bien de un *else*. Debido a que el *if* interno está contenido en un bloque, no hay confusión sobre su función o ejecución. Sin embargo, en lo que sigue :

```
if(x)
        if(y)sentencia 1;
        else sentencia 2;
```

en C el *else* se asocia al *if* más inmediato que no tenga una sentencia *else*. En este caso, el *else* está asociado a la *sentencia if(y)*. Si se quiere que el *else* se asocie a *if(x)*, hay que utilizar llaves para forzar una evaluación diferente; por ejemplo :

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

Ahora el *else* se asocia a *if(x)*. En esencia, el *if* y el *else* están en el mismo nivel. Una construcción común de programación es la escalera *if-else-if*. Su esquema es :

```
if(condición)
        sentencia;
else if(condición)
        sentencia;
else if(condición)
        sentencia;
.
.
.
else
        sentencia;
```

Las condiciones se evalúan de arriba a abajo. Tan pronto como se encuentre una condición cierta, se ejecuta la sentencia asociada, y se salta el resto de la escalera. Si ninguna de las condiciones son ciertas, se ejecutará el último *else*. El último *else* a menudo actúa como una condición por defecto; es decir, si todas las demás pruebas de condición fallan, se ejecuta la última sentencia *else*. Si el último *else* no está presente, no tendrá lugar ninguna acción si todas las otras condiciones son falsas.

II.2.7 La Estructura SWITCH

Aunque la escalera *if-else-if* puede ejecutar una secuencia de pruebas, es poco elegante. El código se hace difícil de seguir e incluso puede confundir al programador en una fecha posterior. Por estas razones, el C incluye una sentencia de decisión con bifurcación múltiple llamada *switch*. La sentencia *switch* actúa comparando sucesivamente una variable con una lista de enteros o de caracteres constantes. Cuando se obtiene una igualdad, se ejecutan una sentencia o bloque de sentencias. El formato general de la sentencia *switch* es :

```
switch(variable)
        {
        case constante1;
        sentencia;
        case constante2;
        sentencia;
```

```
case constante3;
sentencia;
.
.
.
default:
        sentencia;
}
```

en donde *default* se ejecuta si no se encuentran igualdades. *default* es opcional, y si no está presente no tiene lugar ninguna acción cuando todas las comprobaciones fallan.

switch difiere de *if* en que *switch* sólo puede probar una igualdad, mientras que *if* puede evaluar una expresión relacional o lógica.

La sentencia *switch* a menudo se utiliza para procesar comandos introducidos por teclado como es el caso de selección por menús. Como se muestra en el programa siguiente, la función *menu()* mostrará en pantalla un menú para un programa de comprobación sintáctica y llamará a los procedimientos adecuados.

```
menu()
        {
        char ca;
        printf("1. Comprobación Sintáctica\n");
        printf("2. Corrige Errores Sintácticos\n");
        printf("3. Muestra Errores Sintácticos\n");
        printf("Pulsar cualquier tecla para saltar\n");
        printf("Introduzca su elección : ");
        ca=getchar();/* lee selección del teclado */
        switch(ca)      {
                        case '1':
                                comp_sint();
                                break;
                        case '2':
                                cor_error();
                                break;
                        case '3':
                                mues_error();
```

```
                    break;
            default :
                    print("No selecciona opción");
            }
}
```

La sentencia *break* utilizada dentro de cada *case* de un *switch* origina que el control del programa salga de la sentencia completa *switch* y continúe en la sentencia siguiente al *switch*. Si no se incluyen las sentencias *break*, todas las sentencias antes y después de la igualdad se ejecutarán. Se puede pensar en *case* como en una etiqueta para indicar donde debe seguir la ejecución después de leer una opción desde el teclado. A diferencia de *break*, *case* no para la ejecución.

Cabe hacer mención, que una segunda forma de terminar un bucle desde dentro, es utilizando la función *exit()*, que se encuentra en la librería estándar. Debido a que la función *exit()* originará una terminación inmediata del programa y una vuelta al sistema operativo.

II.3 FUNCIONES

El lenguaje C se basa en el concepto de construcción con módulos. Estos módulos se llaman *funciones*. Un programa C es una colección de una o más funciones. Para escribir un programa, primero hay que crear las funciones luego relacionarlas entre sí.

Una función es una subrutina que contiene una o más sentencias C, y que lleva a cabo una o más tareas. En un código C bien escrito, cada función lleva a cabo sólo una tarea. Cada función tiene un nombre y una lista de argumentos que recibirá la función. En general se le puede dar a cada función el nombre que se quiera, excepto el de *main()*, que está reservado para la función que inicia la ejecución del programa.

Para indicar funciones, se ha seguido el convenio de que cada función lleve un paréntesis después del nombre de la función, para distinguir las funciones de las variables.

II.3.1 Estructura de una Función

La forma general de una función es :

```
especificador_de_tipo nombre_de_la_función(lista de parámetros)
{
        cuerpo de la función
}
```

El *especificador_de_tipo* especifica el tipo de valor que devuelve la sentencia *return* de la función. El valor puede ser cualquier tipo. Si no se especifica ningún tipo, el compilador asume que la función devuelve como resultado un entero. La *lista de parámetros*, es la lista de nombres de variables separados por comas con sus tipos asociados que reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, en cuyo caso la lista de parámetros esta vacía. Sin embargo, incluso cuando no hay parámetros se requieren paréntesis.

En las declaraciones de variables, se puede declarar múltiples variables del mismo tipo, mediante una lista con los nombres de las variables separados por comas. Pero en las funciones todos los parámetros deben incluir tanto el tipo como el nombre de la variable. Es decir, la lista de declaración de parámetros de una función tiene la siguiente forma general :

f(tipo var1, tipo var2,...,tipo varN)

II.3.2 Funciones Prototipo

Los prototipos de funciones ayudan a detectar errores antes de que se den. Además, ayudan a verificar que el programa funciona correctamente al no permitir que se llame a funciones con argumentos que no sean válidos. Cuando se usan prototipos, C puede encontrar e

informar sobre conversiones de tipo ilegales entre el tipo de los argumentos usados en la llamada a la función y las definiciones de tipos de sus parámetros. C también detectará diferencias entre el número de argumentos usados en la llamada a la función y el número de parámetros de la misma.

La forma general de un prototipo de función es :

tipo nombre_de_la_función(tipo parámetro1, tipo parámetro2,... tipo de parámetroN);

El uso de los nombres de los parámetros es opcional. Sin embargo, permiten que el compilador identifique cualquier discordancia de tipo por su nombre, cuando se de un error, por lo que es una buena idea incluirlos.

Debido a la necesidad de compatibilidad con la versión original de C, a los prototipos de funciones se les aplican ciertas reglas especiales. En primer lugar, cuando se declara el tipo devuelto por la función sin información de prototipo, el compilador asume sencillamente que no se da información sobre parámetros. Por lo que al compilador concierne, la función puede tener varios parámetros o ninguno. Cuando una función no tiene parámetros, su prototipo usa *void* entre los paréntesis. Por ejemplo, si una función llamada *f()* devuelve un *float* y no tiene parámetros, su prototipo es :

float f(void);

Esto indica al compilador que la función no tiene parámetros y que cualquier llamada a la función que utilice argumentos será errónea. Los prototipos afectan a la promoción de tipo automática de C. Cuando se llama a una función que no tiene prototipo, todos los caracteres se convierten a enteros y todos los *float* a *double*. Estas, ligeramente extrañas promociones de tipo, tienen que ver con las características del entorno original en que se desarrolló C. Sin embargo, si se incluye el prototipo de la función, se mantienen los tipos especificados en el mismo y no se lleva a cabo ninguna promoción de tipo.

Es recomendable el uso de prototipos de funciones, incluso aunque no sea un error el que no se incluya el prototipo de una función. Esto es necesario para soportar código en C que incluye prototipos. De cualquier forma, en general, su código debe incluir una información de prototipos completa.

II.3.3 Variables

Una variable es una posición de memoria con nombre, que se usa para mantener un valor que puede ser modificado por el programa. Todas las variables en C han de ser declaradas antes de poder ser usadas. La forma general de declaración es :

tipo lista de variables;

Aquí, *tipo* debe ser un tipo de datos válido en C con cualquier modificador y la *lista de variables* puede consistir en uno o más nombres de identificadores separados por comas. A continuación se muestran algunas declaraciones :

```
int i,j,l;
short int si;
unsigned int ui;
double balance, beneficio, perdida;
```

Los nombres de las variables pueden llevar uno o varios caracteres, debiendo ser el primero una letra y los siguientes pueden ser letras, números o el caracter de subrayado. Una variable no puede tener el mismo nombre que el de una palabra clave de C, y no conviene que tenga el mismo nombre que el de una función, bien sea escrita por uno mismo, o perteneciente a la librería de C.

Tipos de Datos

Hay siete tipos de variables predefinidos. El tamaño y el rango de esos tipos dependen del procesador y de la realización concreta de compilador de C. La información sobre el tamaño y el rango se da a continuación.

Tipo	*Número de Bits*	*Rango*
char	*8*	*0 a 255*
int	*16*	*-32768 a 32767*
short int	*8*	*-128 a 127*
unsigned int	*16*	*0 a 65535*
long int	*32*	*-4294967296 a 4294967295*
float	*32*	*aprox. 6 dígitos de prec.*
double	*64*	*aprox.12 dígitos de prec.*

Los rangos de los tipos *float* y *double* normalmente vienen dados por los dígitos de precisión. Las magnitudes representables con *float* y *double* dependen del método utilizado para representar los números en coma flotante.

Declaración de Variables

En C, todas las variables deben haber sido declaradas antes de usarlas. El nombre de una variable no tiene nada que ver con su tipo. La sintaxis para la declaración de cada tipo, se muestra en los siguientes ejemplos :

```
int i;
short int si;
unsigned int ui;
long int li;
float f;
double d;
```

Básicamente son tres los sitios en un programa en C, donde las variables deben ser declaradas; dentro de las funciones, en la definición de los parámetros de la función, o fuera de todas las funciones. Estas variables se llaman respectivamente, *variables locales*, *parámetros formales* o *variables globales*.

Variables Locales

Las *variables locales* se declaran dentro de una función. Sólo las pueden referenciar las sentencias que hay dentro de la función en la que las variables han sido declaradas. Las *variables locales* no son conocidas por las funciones externas a aquella que las contienen; por ejemplo

```
func1()
{
        int x;
        x = 10;
}
func2()
{
        int x;
        x = -199;
}
```

La variable entera *x* ha sido declarada dos veces, en *func1()* y en *func2()*. La *x* en *func1()* no tiene nada que ver ni está relacionada con la *x* en *func2()*.

En C, *las variables locales* son creadas cuando la función es llamada, y destruidas cuando sale de la función. De forma análoga, la memoria necesaria para estas variables locales es creada y destruida dinámicamente. También son llamadas *variables dinámicas* o *variables automáticas*. Como las *variables locales* son creadas y destruidas en cada llamada a la función, sus contenidos se pierden cada vez que se retorna de la función.

Parámetros Formales

Si una función tiene argumentos, estos deben de haber sido declarados. Son los llamados *parámetros formales* de la función. Son como otras variables locales de la función; su declaración se hace después del nombre de la función y antes de la llave abierta.

Por ejemplo :

```
func1(primero, ultimo, ca)
int primero, ultimo;
char ca;
{
        int cont;
        cont = primero*ultimo;
        ca = 'a';
                .
                .
                .
}
```

En este ejemplo *func1()* tiene tres argumentos llamados *primero*, *ultimo* y *ca*. Hay que especificarle a C qué tipo de variables son las que se declaran. Una vez hecho, se usan dentro de la función como variables locales normales.

Se debe asegurar que los *parámetros formales* declarados son del mismo tipo que el de los argumentos que se utilizarán al llamar a la función.

Al igual que con las variables locales, se pueden hacer asignaciones a los parámetros formales de una función o utilizarlos en cualquier expresión válida de C. Aun cuando estas variables lleven a cabo la específica tarea de recibir los valores de los argumentos pasados a la función, pueden utilizarse como cualquier otra variable local.

Variables Globales y Extern

Al contrario que las variables locales, las *variables globales* mantienen sus valores en todo el programa, y mientras dura la ejecución. Las *variables globales* se crean declarándolas fuera de toda función. Pueden ser accedidas por cualquier expresión, independientemente de en qué función se encuentre.

Consideremos este ejemplo:

```
int cont; /* cont es global */
main()
{
        cont = mul(10,123);
        .
        .
        .
}
func1()
{
        int temp;
        temp = cont;
        .
        .
        .
}
func2()
{
        int cont;
        cont = 10;
        .
        .
        .
}
```

Como se puede ver, la variable *cont* ha sido declarada fuera de todas las funciones. Sin embargo, puede colocarse en cualquier sitio aunque no haya sido declarada dentro de una función. Es mejor declarar las *variables globales* al comienzo del programa. También

muestra que ni *main()* ni *func1()* tienen declarada la variable *cont*, aunque ambas la usen. Sin embargo, la función *func2()* tiene declarada *cont* como *variable local*. Cuando *func2()* haga referencia a *cont*, se referirá a la *variable local*, no a la *global*.

Se puede declarar una *variable global* sólo una vez, si se intenta declarar dos variables globales con el mismo nombre, el compilador de C imprimirá el mensaje ***duplicate variable name***, que significa que el compilador no sabe cuál de las dos quiere que utilice. El mismo problema aperece si se declara las *variables locales* en todos los archivos; porque uno puede intentar crear dos copias de cada variable, pero al encadenar los módulos se obtendrá el mensaje ***duplicate label***, etiqueta repetida, pues el encadenador no sabrá qué variable debe utilizar. La solución es declarar todas sus *variables globales* en un archivo y utilizar *extern* (externas) en los otros, como se muestra en el ejemplo siguiente:

Archivo Uno

```
int x,y;
char ca;
main()
{
        .
        .
        .
}
func1()
{
        x=123;
}
```

Archivo Dos

```
extern int x,y;
extern char ca;
func2()
{
        x=y/10;
}
func3()
{
        y=10;
}
```

En el archivo dos, la lista de variables se ha copiado del archivo uno, añadiendo el modificador *extern* a las declaraciones. Este modificador le indica al compilador que los nombres y tipos de variables que siguen, ya han sido declaradas en algún sitio. En otras palabras *extern* le indica al compilador que ya conoce los tipos y nombres de estas variables, por lo que no necesita crearlas de nuevo. Cuando el encadenador enlace los módulos, todas las referencias a las *variables externas* serán eliminadas.

Las Variables Estáticas (static)

Las *variables estáticas static* son variables permanentes en su propia función o archivo. Se diferencian de las variables globales, en que aunque no son conocidas en el exterior de su función o archivo, mantienen sus valores entre llamadas sucesivas. Esta característica las hace muy útiles cuando uno escribe funciones generalizadas y librerías de funciones, para ser usadas por otros programadores.

Un ejemplo de función a la que viene bien esta clase de variable es un generador de una serie numérica que produce un número nuevo que es función del que produjo la vez anterior. Sería posible declarar una variable global para ese valor, pero se debe recordar su nombre cada vez, lo que es un inconveniente. El uso de una variable global haría difícil el colocarla en una librería de funciones. La solución es declarar la variable que mantendrá el número generado, como *static*, tal como se indica :

```
serie()
{
        static int num_serie;
        num_serie=num_serie+23;
        return(num_serie);
}
```

En este ejemplo, la variable *num_serie* permanenece entre llamadas sucesivas a la función en vez de andar enviando y recibiendo valores cada vez, como ocurriría con una variable local normal. Esto permite producir un número nuevo cada vez que se llama a *serie()*, y que se basará en el último número generado, sin la necesidad de declarar la globalidad de la variable.

Las Variables Registro (register)

C tiene un último modificador, de declaración de variable, aplicado casi exclusivamente a los tipos *int* y *char*. El modificador *register* (registro), obliga al compilador de C a mantener el valor de las variables declaradas con este modificador, en un registro del CPU, en vez de en memoria, que es donde normalmente se almacenan las variables.

Esto hace que las operaciones sobre las *variables register* sean mucho más rápidas, ya que el valor de las *variables register* se mantiene en el CPU, y no se requiere un acceso a memoria. Esta característica las hace ideales para el control de un bucle. El modificador *register* sólo puede aplicarse a las variables locales, y a los parámetros formales en una definición de función. No se permiten *variables globales register*. Lo que sigue es un ejemplo de cómo se pueden declarar *variables register* de *int* y *char* :

```
func1(s,u)
register int s;
register char u;
{
        float temp;
        register int contador;
        .
        .
        .
}
```

El número exacto de *variables register* dentro de una misma función, queda determinado tanto por el tipo de procesador empleado, como por la realización concreta del C que se esté utilizando. Para la mayoría de los sistemas de 8 bits, sólo se permite una *variable register*, mientras que en los de 16 bits normalmente se permiten dos o más. Esto sólo preocupará cuando la velocidad sea importante, ya que si uno declara muchas *variables register*, el compilador de C las colocará automáticamente en variables que no son de registro.

II.3.4 Clases de Almacenamiento

C admite cuatro especificadores de clase de almacenamiento :

auto
extern
static
y register

Se utilizan para indicar al compilador la forma en que debe ser almacenada la variable que va detrás. El especificador de almacenamiento precede al resto de la declaración de la variable.

Su forma general es :

especificador-clase-almacenamiento especificador-tipo lista-variables:

Auto

El especificador *auto* se utiliza para declarar variables locales. Sin embargo, se usa rara vez, porque las variables locales son *auto* por omisión. Es sumamente raro ver esta palabra en algún programa.

Los especificadores *extern*, *static* y *register*, fueron descritos dentro del punto anterior de Variables, ya que se encuentran dentro de la clasificación de variables y se utilizan como especificadores de almacenamiento.

III.3.5 Macros

Una Macro-Instrucción es simplemente la sustitución de una parte del texto por otra al compilar el programa. En el caso de cadenas, a menudo es más fácil, y más corto trabajar con macros en el programa. En el caso de números, el utilizar la macroinstrucción permite cambiar fácilmente una constante utilizada en el programa.

Por otra parte el estándar ANSI especifica 5 nombres de macros predefinidos. Los cuales son : *_LINE_*, *_FILE_*, *_DATE_*, *_TIME_* y *_STDC_*.

Si el compilador no es estándar pueden faltar alguno o todos. El compilador también puede proporcionar otras macros predefinidas.

Las macros *_LINE_* y *_FILE_* se comentan a continuación :

La directiva *#line* cambia los contenidos de *_LINE_* y *_FILE_*. que son identificadores de compilador predefinidos. El identificador *_LINE_* contiene el número de línea que se está compilando actualmente. El identificador *_FILE_* es una cadena que contiene el nombre del archivo fuente que se está compilando.

La forma general de *#line* es :

#line número "nombre_de_archivo"

donde *número* es cualquier entero positivo que se convierte en el nuevo valor de *_LINE_* y *nombre_de_archivo* es opcional, y es cualquier identificador válido de archivo que se convierte en el nuevo valor de *_FILE_*. La directiva *#line* se usa principalmente para depuración y aplicaciones especiales.

La macro *_DATE_* contiene una cadena de la forma mes/día/año. Esta cadena representa la fecha de traducción del código fuente a código objeto.

La hora de la traducción del código fuente a código objeto está contenida como una cadena en *_TIME_*. La forma de esta cadena es horas:minutos:segundos.

La macro *_STDC_* contiene la constante decimal 1. Esto significa que la implementación se ajusta al estándar. Si la macro contiene otro número es que la implementación varia de la estándar.

11.3.6 El Preprocesador C

Se pueden incluir varias instrucciones dirigidas al compilador en el código fuente de un programa en C. Se llaman *directivas de preprocesamiento* y, aunque no son realmente parte del lenguaje C, amplian el ámbito del entorno de programación en C.

Tal como esta definido por el estándar ANSI, el preprocesador de C contiene las siguientes directivas : *#if, #ifdef, #ifndef, #else, #elif, #include, #define, #undef, #line, #error,* y *#pragma*. Como se puede ver, todas las directivas del preprocesador empiezan con el símbolo #. Además, cada directiva de preprocesamiento debe estar en su propia línea.

#define

La directiva *#define*, define un identificador y una cadena que será sustituida por el identificador cada vez que se encuentre en el archivo fuente. El estándar ANSI denomina al identificador nombre de macro y al proceso de reemplazamiento sustitución de macro. La forma general de la directiva es :

#define nombre_de_macro cadena

Observamos que no hay punto y coma en esta sentencia. Puede haber cualquier número de espacios en blanco entre el identificador y la cadena, pero una vez que empieza la cadena, sólo acaba con un salto de línea.

Por ejemplo, si se desea usar CIERTO para el valor 1 y FALSO para el valor 0, se pueden declarar dos macros *#define*.

```
#define CIERTO 1
#define FALSO 0
```

Esto hace que el compilador sustituya por un 1 o un 0 cada vez que encuentre CIERTO o FALSO en el archivo fuente.

#error

La directiva *#error* fuerza al compilador a parar la compilación. Se usa principalmente en la depuración. La forma general de la directiva *#error* es :

```
#error mensaje_de_error
```

El *mensaje_de_error* no está entre comillas. Cuando se encuentra la directiva *#error*, se muestra el mensaje de error, posiblemente junto con otra información definida por el compilador.

#include

La directiva *#include* hace que el compilador incluya otro archivo fuente en el que tiene la directiva *#include*. El nombre del archivo fuente a leer debe estar entre dobles comillas o entre ángulos. Por ejemplo :

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

ambas líneas de código hacen que el compilador de C lea y compile la cabecera de las rutinas de la biblioteca de archivos en disco. Los archivos incluidos pueden contener directivas *#include*. Este tipo de situación se llama anidamiento de inclusiones. El número de niveles de anidamiento permitido depende del compilador. Sin embargo, el estándar ANSI estipula que al menos se deben permitir ocho niveles de anidamiento.

#if, #else, #elif, #endif

Hay directivas que permiten compilar selectivamente partes del código fuente de un programa. Este proceso se llama *compilación condicional* y se usa mucho en casas comerciales de software que suministran y mantienen versiones a medida de un programa.

Si la expresión constante que sigue a *#if* es cierta, se compila el código que hay entre el *#if* y el *#endif*. En otro caso, se ignora ese código. La directiva *#endif* marca el final de un bloque *#if*. La forma general de *#if* es :

```
#if expresión-constante
        secuencia de sentencias
#endif
```

Si la expresión constante es cierta, se compila el bloque de código; si no, se salta.

La directiva *#else* funciona de una manera muy parecida al *else* que forma parte del lenguaje C: establece una alternativa para el caso de que *#if* falle.

La directiva *#elif* quiere decir *"else if"* y establece una escalera de tipo *if-else-if* para opciones de compilación múltiples. La directiva *#elif* va seguida de una expresión

constante. Si la expresión es cierta, ese bloque de código se compila y no se comprueba ninguna expresión *#elif* más. En cualquier otro caso, se comprueba el siguiente bloque de la serie. La forma general de *#elif* es :

```
#if expresión
        secuencia de sentencias
#elif expresión 1
        secuencia de sentencias
#elif expresión N
        secuencia de sentencias
#endif
```

#ifdef* e *#ifndef

Otro método de *compilación condicional* utiliza las directivas *#ifdef* e *#ifndef* que quieren decir *"si definido"* y *"si no definido"* respectivamente. La forma general de *#ifdef* es :

```
#ifdef nombre_de_macro
        secuencia de sentencias
#endif
```

Si se ha definido previamente el *nombre_de_macro* en una sentencia *#define*, se compila el bloque de código que sigue a la sentencia. La forma general de *#ifndef* es :

```
#ifndef nombre_de_macro
        secuencia de sentencias
#endif
```

Si no se ha definido previamente el *nombre_de_macro* mediante una sentencia *#define*, se compila el bloque de código.

Ambas, *#ifdef* e *#ifndef* pueden usar una sentencia *#else*, pero no una sentencia *#elif*.

#undef

La directiva *#undef* elimina una definición anterior del *nombre de macro* que le sigue. La forma general es :

#undef nombre_de_macro

El propósito principal de *#undef* es asignar los nombres de macro sólo a aquellas secciones que las necesiten.

#line

La directiva *#line* se definió en el apartado de Macros.

#pragma

La directiva *#pragma* es una directiva definida por la implementación que permite que se den varias instrucciones al compilador. Por ejemplo, un compilador puede tener una opción que permita el seguimiento paso a paso de un programa en ejecución. Podría especificarse una opción tal de traza mediante una sentencia *#pragma*.

II.4 ESTRUCTURAS, UNIONES Y APUNTADORES

El C permite crear tipos de datos nuevos de dos formas: primera, combinando muchas variables en una variable conglomerado denominada estructura; y segunda utilizando una unión para permitir que muchas variables compartan la misma memoria. Estas características se combinan dando al C un conjunto muy rico de tipos de variables.

II.4.1 Estructuras y Uniones

Una estructura es una colección de variables que están referenciadas bajo un nombre. C utiliza estructuras para proporcionar medios convenientes de mantener en un sitio la información que está relacionada.

Una definición de estructura forma una plantilla que se puede utilizar para crear estructuras de variables. Cada estructura está formada por una o más variables que están relacionadas lógicamente. Estas variables se denominan *elementos de la estructura*.

Las estructuras, como grupo de variables conectadas lógicamente, se pueden pasar fácilmente a funciones. La utilización de estructuras también puede hacer que sea mucho más fácil de leer el código fuente debido a que la conexión lógica entre los elementos de la estructura es obvia.

Por ejemplo, un nombre y una dirección agrupadas en una lista de correo es un conjunto común de información relacionada. El siguiente fragmento de código declara una estructura para mantener los campos del nombre y de la dirección; la palabra clave *struct* le indica al compilador que se va a definir una plantilla de una estructura :

```
struct dir    {
              char nombre[30];
              char calle[40];
              char ciudad[20];
              char prov[20];
              unsigned long int DP;
              };
```

Acerca de esta definición hay dos aspectos. Primero, se termina con un punto y coma porque una definición de estructura es una sentencia. Segundo, que la etiqueta de la estructura *dir* identifica esta estructura concreta de dato y es su nombre.

Hasta este momento no se ha declarado una variable. Sólo se ha definido el formato del dato. Para declarar una variable real con esta estructura, se debería escribir :

```
struct dir ainfo;
```

Esto declarará una variable del tipo *dir* denominada *ainfo*. Cuando se define una estructura, en esencia se esta definiendo una variable de tipo complejo formada por elementos de la estructura. También se pueden declarar una o más variables al mismo tiempo que se define una estructura. Por ejemplo :

```
struct dir      {
                char nombre[30];
                char calle[40];
                char ciudad[20];
                char prov[20];
                unsigned long int DP;
                } ainfo, binfo, cinfo;
```

definirá una estructura denominada *dir* y declarará las variables *ainfo*, *binfo* y *cinfo* del tipo *dir*.

Si sólo se necesita una variable estructura, no es necesario el nombre de la estructura. Esto significa que :

```
struct  {
                char nombre[30];
                char calle[40];
                char ciudad[20];
                char prov[20];
                unsigned long int DP;
        } ainfo;
```

declara una variable estructura, denominada *ainfo* con la estructura que le precede.

El formato general de una definición de estructura es :

```
struct nombre_estructura    {
                            type nombre_variable;
                            type nombre_variable;
                            type nombre_variable;
                            .
                            .
                            .
                            } variables_estructura;
```

en donde se puede omitir o *nombre_estructura* o *variables_estructura*.

Referencia a los elementos de la estructura

El siguiente código asignará *12345* a la variable estructura *ainfo* declarada anteriormente.

```
ainfo.DP = 12345;
```

Como se puede ver, el nombre de la estructura *ainfo* seguido por un punto y el nombre del elemento referenciará a ese elemento individual de la estructura. Al punto se le denomina *operador punto*, significa que sigue a un elemento de la estructura. A todos los elementos de la estructura se accede de la misma forma. El formato general es :

```
nombre_estrcutura.nombre_elemento
```

Por lo tanto, para imprimir en la pantalla el código *DP*, se escribiría :

```
printf("%d",ainfo.DP);
```

Esto imprimirá el código contenido en la variable *DP* de la variable estructura *ainfo*. Por ejemplo, consideremos *ainfo.nombre*. Este elemento es un arreglo de caracteres. Al utilizar *gets()* para introducir un nombre, se escribiría :

```
gets(ainfo.nombre);
```

Esto pasará un puntero a caracter al principio de nombre. Si se quisiera acceder a los elementos individuales de *ainfo.nombre*, se podría indexar *nombre*. Por ejemplo, se imprimiría el contenido de *ainfo.nombre* utilizando :

```
register int t;
for(t=0; ainfo.nombre[t]; ++t) putchar(ainfo.nombre[t]);
```

Uniones

En C una *unión* es un lugar de la memoria que se utiliza por algunas variables diferentes potencialmente de diferentes tipos. A continuación se da la definición de una unión, denominada *u*, de un caracter y de un entero :

```
union uf
        {
            int i;
            char ca;
        }
```

Como con las estructuras, esta definición no declara ninguna variable. Una variable se puede declarar o colocando su nombre al final de la definición o utilizando una sentencia de declaración separada. Para declarar una variable *union. cnvt.* sea del tipo *u* utilizando la definición que se acaba de dar, se escribiría :

```
union u cnvt;
```

En *union cnvt.* tanto el entero *i* como el caracter *ca* comparten el mismo lugar en memoria, es decir *i* y *ca* comparten la misma dirección.

Cuando se declara una unión el compilador creará automáticamente una variable lo suficientemente larga como para contener la longitud de tipo de variable en la unión.

Para acceder a un elemento de la unión, se utiliza la misma sintaxis que se utilizó para las estructuras. Si la variable *union* es global, se utiliza el operador punto; si la variable *union* se pasa a una función, se utiliza el operador flecha. Por ejemplo si *union cnvt* es global, para asignar entero 10 a su elemento i, se escribiría :

```
cnvt.i=10
```

Sin embargo, si *cnvt* se pasase a una función, hay que utilizar el operador -> :

```
func1(un)
union u un;
{
        un->i=10; /* asigna 10 a cnvt usando función */
}
```

La utilización de una unión puede ayudar a la producción de código independiente de la máquina, o transportable. Debido a que el compilador sigue la pista de todos los tamaños, no se producen dependencias de la máquina. No hay que preocuparse por el tamaño de un entero, de un caracter, o de un *float*. Por ejemplo, se puede utilizar una unión en conversiones de tipo. La función *putw()* escribirá la representación binaria de un entero a un archivo en disco. Primero hay que crear una unión con un entero y un arreglo de caracteres de dos bytes :

```
union pw    {
                int i;
                char ca[2];
            }
```

Ahora, *putw()* se puede escribir utilizando esta unión :

```
putw(palabra,fp) /* putw con unión */
union pw palabra;
```

```
FILE *fp;
{
        putc(palabra->ca[0];
        putc(palabra->ca[1];
}
```

Para escribir una palabra en un archivo en disco, se llamaría a *putw()* con el valor entero que se quisiese escribir.

II.4.2 Apuntadores como Direcciones

Un *apuntador* es una variable que contiene una dirección, esta dirección es la posición de otra variable en memoria. El valor de un apuntador *"apunta"* a una variable, a la que se puede acceder indirectamente con los operadores especiales sobre apuntadores * y &.

El operador * accede al contenido de una variable cuya dirección es el valor del apuntador. El * se puede recordar como *"en la dirección"*. El operador & devuelve la dirección de una variable y se puede recordar como *"la dirección de"*.

Por ejemplo, si *xyz* y *k* son variables esteras y *h* es un apuntador a entero, entonces

```
h = &xyz;
k = *h;
```

asigna el valor de *xyz* a *k*.

Los apuntadores tienen que haber sido declarados. Para declarar *x* como un apuntador a entero, se debe utilizar :

```
int *x;
```

Para un apuntador a *float*, *x*, se indica :

*float *x;*

Hay que asegurase de que las variables apuntador siempre apuntan al tipo de dato correcto. Cuando se declara que un apuntador es del tipo *int*, entonces el compilador supone que cualquier dirección que tenga el apuntador apuntará a una variable entera.

II.4.3 Operaciones sobre Apuntadores

Sólo hay dos operadores aritméticos que se pueden utilizar con los apuntadores, + y -. Para comprender la aritmética, supongamos que *p1* es un apuntador a entero con un valor de 2000. Después de la expresión *p1++;* .

El contenido de *p1* será 2002, no 2001. Cada vez que *p1* se incrementa, apuntará al entero siguiente, que en la mayoría de las computadoras ocupan dos bytes. Esto se mantiene en los decrementos. Por ejemplo, *p1--;* hará que *p1* tenga el valor de 1998, suponiendo que inicialmente era 2000.

Cada vez que un apuntador se incrementa, apuntará a la posición de memoria del siguiente elemento de su tipo. Cada vez que se decremente, apuntará a la posición del elemento anterior de su tipo.

En los apuntadores a caracter normalmente coincide con la aritmética normal. Sin embrago, todos los demás apuntadores se incrementarán o decrementarán según la longitud del tipo de dato al que apuntan. Para los enteros, esa longitud normalmente es de dos bytes; para los flotantes es de ocho.

Sin embargo, no se está limitado a incrementar y decrementar apuntadores. También se pueden sumar o restar a los apuntadores. La expresión *p1 = p1 + 9*; hará que *p1* apunte al noveno elemento del tipo de *p1* a partir del que actualmente apunta.

Además de la suma y resta de un apuntador y un entero, no se pueden ejecutar otras operaciones aritméticas con apuntadores, no se pueden multiplicar o dividir apuntadores, no se pueden sumar o restar dos apuntadores, no se les pueden aplicar los desplazamientos de bits ni los operadores lógicos; y no se pueden sumar o restar tipos *float* o *double* a los apuntadores.

11.4.4. Apuntadores a Funciones

Una utilización especialmente confusa en el uso de los apuntadores es con los apuntadores a funciones. Incluso aunque una función no sea una variable, sigue teniendo una posición en memoria que se puede asignar a un apuntador. A continuación este apuntador se puede utilizar para manejar las llamadas a funciones.

Para comprender el concepto de apuntador a función, consideremos el programa siguiente. La función *strcmp()* es la función estándar de comparación de cadenas que se encuentra en la librería.

Se ha declarado en *main()* por lo que el programa sabrá qué tipo de valor se ha devuelto (en este caso un entero) y que es una función, no una variable. Cuando se llama a la función *comp()*, se pasan como parámetros dos caracteres y un apuntador a función. Dentro de la función *comp()*, los argumentos se han declarado como apuntadores a caracter y como apuntador a función.

Hay que utilizar exactamente el mismo método que se muestra cuando se declara un apuntador a función. Los paréntesis son necesarios para que el compilador pueda interpretar

esta sentencia correctamente. Sin el paréntesis alrededor de **cmp*, el compilador supondría que simplemente se está declarando una función, que no es lo que deseamos.

Una vez dentro de *comp()*, se puede ver cómo se llama a la función *strcmp()*. La sentencia *(*cmp) (a,b)* realizará una llamada a la función (en este caso *strcmp()*), a la que apunta *cmp* con los argumentos *a* y *b*.

```
main()
{
        int strcmp(); /* declara una función */
        char s1[80], s2[80];

        gets(s1),gets(s2);
        comp(s1,s2,strcmp); }
comp(a,b,cmp)
char *a, *b;
int (*cmp) (); {
printf("comprueba la igualdad\n");
if(!(*cmp) (a,b)) printf("igual");
else printf("distinto");}
```

II.4.5 Apuntadores y Arreglos

Como se puede imaginar, existe una estrecha relación entre apuntadores y arreglos. Por ejemplo en :

```
char cdn[80];
char *p1;
p1 = str;
```

p1 apunta a la dirección del primer elemento del arreglo en *cdn*. Si se quisiera ir al quinto elemento de *cdn* se podría escribir *cdn[4]* o **(p1+4)*.

Ambas sentencias devolverán el quinto elemento. Debemos recordar que los arreglos empiezan en cero. También se debe añadir un cuatro al apuntador *p1* para obtener el quinto elemento porque *p1* actualmente apunta al primer elemento de *cdn*.

En esencia, C permite dos métodos de acceso a los elementos de un arreglo. Esto es importante porque la aritmética de los apuntadores puede ser más rápida que el indexar un arreglo. Ya que la velocidad es a menudo una consideración importante en programación, la utilización de los apuntadores para acceder a los elementos de un arreglo es muy corriente en los programas en C.

Por ejemplo, a continuación hay dos versiones de *puts()*. una indexando un arreglo,

```
puts(s) /* con arreglos */
char *s;
{
        register int t;
        for(t=0;s[t];++t);
        putchar (s[t]);
}
```

y otra con apuntadores

```
puts(s) /* con apuntadores */
char *s;
{
        while(*s) putchar (*s++);
}
```

No debemos pensar que el indexar un arreglo es una equivocación porque tiene su sitio. Recordemos que si se va a acceder al arreglo en orden estrictamente ascendente o descendente, los apuntadores son más rápidos y más fáciles de utilizar.

Apuntadores a Arreglos de Caracteres

La mayoría de las operaciones sobre cadenas en C se realizan generalmente con apuntadores a arreglos y apuntadores aritméticos porque los apuntadores son más rápidos y más fáciles de utilizar.

Lo que sigue son dos formas de escribir la función *strcmp()*, que se encuentra en la librería de C.

```
strcmp(s1,s2) /* con arreglos */
char *s1, *s2;
{
        register int i;
        for(i=0;s1[i];++i)

                if(s1[i]!=s2[i]) return s1[i];
        if(s2[i]) return (s2[i]);
        return '\0'
}

strcmp(s1,s2) /* con apuntadores */
char *s1, *s2;
{ while(*s1)
                if(*s1++!=*s2++) return *(s1--);
        if(*s2) return (*s2);
        return '\0';
}
```

Todas las cadenas en C se terminan con un caracter nulo, que es un valor falso. Por lo tanto, una sentencia como *while(*s1)* es cierta hasta que se llegue al final de la cadena.

En el siguiente ejemplo, cada vez que el bucle se itera, *p1* se restaura al principio de la cadena. Recordemos que hay que saber adonde apuntan los apuntadores cada vez que se usan.

```
main()
{
        char s[80];
        char *p1;
        do      {
                p1=s;
                gets(s);  /* lee una cadena */
                while(*p1) printf("%d",*p++);
                /*imprime el equivalente decimal de cada caracter*/
                } while(!strcmp(s,"hecho");
}
```

Arreglos de Apuntadores

Los apuntadores pueden estar en arreglos como cualquier otro tipo de dato. La declaración para un arreglo de apuntadores *int* de tamaño 10 es *int *x[10]*.

Para asignar la dirección de una variable entera denominada *var* al tercer elemento del arreglo de apuntadores, se escribiría :

```
int var;
x[2]=&var;
```

Recordemos que se está trabajando con un arreglo de apuntadores. Los únicos valores que los elementos del arreglo pueden mantener son las direcciones de variables enteras. Para buscar el valor de *var*, se escribiría **x[2]*.

Apuntadores a arreglos en general

Los apuntadores de cualquier tipo de arreglo operan como una forma alternativa de la indexación. Existen unos cuantos conceptos que hay que tener en cuenta al utilizar apuntadores a arreglos. Las comparaciones entre apuntadores que no acceden al mismo arreglo no son válidas y originarán errores. Puede que nunca se sepa en que lugar de la

memoria está colocado el dato, si estará allí otra vez de la misma forma, o si cada compilador lo tratará de la misma forma. Por lo tanto, el hacer cualquier comparación entre apuntadores a dos arreglos distintos producirá resultados inesperados. Por ejemplo :

```
char s[80];
char y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1<p2)...
```

Es un concepto intrínsecamente inválido. No hay que hacer este tipo de programación a menos que la aplicación sea poco corriente y requiera el conocimiento del lugar en memoria de algunas variables.

Un error de esta clase es el suponer que dos arreglos juntos se pueden indexar como uno, simplemente incrementando un apuntador apuntando a un límite del arreglo. Por ejemplo :

```
int prim[10];
int segun[10];
int *p,i;
p = prim;
for(i=0;i<20;++i) *p++=i;
```

No se puede utilizar para inicializar los arreglos *prim* y *segun* con los números del 0 al 19. Incluso aunque pueda funcionar en algunos compiladores bajo ciertas circunstancias, presupone que ambos arreglos se colocarán inicialmente juntos en memoria. Puede que ese no sea siempre el caso, y normalmente puede que sólo induzca a problemas.

II.5 ARCHIVOS

En C hay básicamente dos formas distintas para leer y escribir archivos en disco. La primera es *E/S de alto nivel* : todas las lecturas y escrituras se hacen caracter a caracter. A veces esto se denomina E/S intermedia porque no hay que preocuparse por los tamaños del sector, longitudes del buffer, ni de cualquier otra consideración dependiente del sistema operativo. En otras palabras, estas funciones proporcionan su propia memoria intermedia. La segunda forma es *E/S de bajo nivel*, lo que a veces se denomina tipo-UNIX. En este método hay que ejecutar cada lectura y escritura manualmente, indicando las memorias intermedias, contadores y apuntadores propios.

Archivos de Cabecera

C soporta dos sentencias que simplifican enormemente algunos aspectos de la programación. La primera es *#define*, que se utiliza para definir una cadena de caracteres como una constante. Por ejemplo, esta sentencia hace que la cadena *MAX_NUM* signifique *100* :

#define MAX_NUM 100

La segunda sentencia que se necesita conocer es *#include*. Se utiliza durante la compilación para leer otro archivo fuente, normalmente indicado en información de la cabecera del programa. Por ejemplo, si se tienen varios archivos para un programa denominado *hoja de cálculo electrónica*, se puede crear un programa de cabecera estándar que contenga todas las variables globales necesarias para los distintos archivos. Si a este archivo, se le denomina *hoja de cálculo electrónica.h*, la primera línea del programa sería la siguiente sentencia :

#include "hoja de cálculo electrónica.h"

La razón por la que se han determinado aquí *#include* y *#define* es que se necesitarán para utilizar las funciones sobre archivos en disco que se suministran con el compilador. Cada archivo que utilice E/S por disco requerirá que se lea en un archivo cabecera denominado *stadio.h*. Tiene que estar en el mismo disco en el que estén los archivos programa. Se debe colocar la línea cerca de la primera línea de cada programa :

#include "stadio.h"

El archivo no solamente definirá algunas estructuras de datos dependientes del sistema, sino que también definirá algunas macros, tales como *EOF* para señalar el final del archivo.

11.5.1 Archivos de Alto Nivel

En el sistema C de E/S de alto nivel hay cinco funciones esenciales. Son :

fopen()	*que abre un archivo para utilizarlo.*
putc()	*que escribe un caracter en un archivo.*
getc()	*que lee un caracter de un archivo.*
fclose()	*que cierra un archivo.*
fseek()	*que se utiliza para ejecutar, operaciones de disco aleatorias.*

La Función fopen()

La función *fopen()* sirve para dos funciones: la primera es abrir un archivo en disco para utilizarlo, y la segunda para devolver un apuntador al archivo. El formato general de *fopen()* es :

```
FILE *fp;
fp=fopen(nombre de archivo,modo);
```

donde *nombre de archivo* es un puntero a una cadena de caracteres que representan un nombre válido del archivo. La cadena a la que apunta *modo* determina cómo se abre el archivo. El siguiente resumen muestra los valores permitidos para *modo*.

Modo	*Significado*
r	*Abre un archivo de texto para lectura*
w	*Crea un archivo de texto para escritura*
a	*Abre un archivo de texto para añadir*
rb	*Abre un archivo binario para lectura*
wb	*Crea un archivo binario para escritura*
ab	*Abre un archivo binario para añadir*
r+	*Abre un archivo de texto para lectura/escritura*
w+	*Crea un archivo de texto para lectura/escritura*
a+	*Añade o crea un archivo de texto para lectura/escritura*
r+b	*Abre un archivo binario para lectura/escritura*
w+b	*Crea un archivo binario para lectura/escritura*
a+b	*Añadir en un archivo binario en modo de lectura/escritura*

La variable *fp* es del tipo *FILE* y es el apuntador a archivo. Todos los apuntadores a archivos deben de declararse como que son de tipo *FILE*.

Si se quisiera abrir un archivo para escritura con el nombre *prueba*, se deberia escribir :

```
fp = fopen("prueba","w");
```

Sin embargo, normalmente se escribirá como :

```
if ((fp=fopen(prueba","w"))==NULL)
{
        puts("no se puede abrir el archivo\n";
        exit();
}
```

Este método detecta cualquier error al abrir un archivo, tal como una protección a escritura o un disco completo, antes de intentar escribir en él. Un nulo, que normalmente es 0, se utiliza porque ningún archivo a apuntador irá a tener valor 0.

Si se utiliza *fopen()* para abrir un archivo en escritura, cualquier archivo preexistente con ese nombre se borrará y se abrirá un archivo nuevo. Si se quiere añadir algo al final del archivo hay que utilizar el *modo a*.

La Función putc()

La función *putc()* se utiliza para escribir caracteres en un archivo en disco que se haya abierto utilizando la función *fopen()* con el *modo w*. El formato general de la función es :

```
putc(c,fp);
```

en donde *fp* es el apuntador a archivo devuelto por *fopen()* y *c* es el caracter a sacar. El apuntador a archivo le dice a *putc()* en qué archivo en disco debe escribir.

La Función getc()

La función *getc()* se utiliza para leer caracteres de un archivo abierto en modo lectura por *fopen()*. El formato general de la función es :

```
char ca;
ca = getc(fp);
```

en donde *fp* es un apuntador a archivo del tipo *FILE* devuelto por *fopen()*. El apuntador a archivo le dice a *getc()* de qué archivo debe leer.

La función *getc()* devolverá una marca de final de archivo cuando se encuentre el final del archivo. El archivo de cabecera *stdio.h* utilizará *#define* para crear la macro *EOF* que será la marca final del archivo. Por lo tanto, para leer hasta la marca final del archivo, se podría utilizar el siguiente código :

```
ca = getc(fp);
while(ca!=EOF)
{
        .
        .
        .
        ca = getc(fp)
}
```

Ya que en el marcador *EOF* no se puede imprimir un caracter, no debemos tratar de imprimirlo.

La Función fclose()

La función *fclose()* se utiliza para cerrar un archivo que se haya abierto mediante una llamada a *fopen()*. Hay que cerrar todos los archivos antes de terminar el programa. La función *fclose()* hace más que liberar el apuntador a archivo; escribe cualquier dato que todavía no se haya escrito en el disco y hace un cierre formal al nivel del sistema operativo. Un fallo al cerrar un archivo invita a todo tipo de problemas, incluyendo la pérdida de datos, la destrucción de archivos y posibles errores intermitentes en el programa.

El formato general para llamar a la función *fclose()* es :

```
fclose(fp);
```

en donde *fp* es el apuntador a archivo devuelto por la llamada a *fopen()*.

La Función fseek()

Se pueden ejecutar operaciones de lectura y escritura de acceso aleatorio utilizando el sistema de E/S de alto nivel con la ayuda de *fseek()*. La función *fseek()* se utiliza para establecer la posición actual (byte específico) en un archivo. El formato general es :

fseek(fp,desplazamiento,origen);

en donde *fp* es un apuntador a archivo devuelto por una llamada a *fopen()*, el *desplazamiento* es el número de bytes desde el origen para determinar la posición actual, y *origen* es o bien un 0 para el principio de archivo, o un 1 para la posición actual, o un 2 para el final del archivo.

Por ejemplo, si se quisiera leer el byte 234 de un archivo denominado *prueba*, se podría utilizar esto :

```
func1()
{
        FILE *fp;
        if((fp=fopen("prueba","r"))==NULL)
                {
                printf("no puede abrirse el archivo\n");
                exit(1);
                }
fseek(fp,234,0);
return getc(fp); /* lee caracter en 234 */
}
```

Las Funciones getw() y putw()

Además de *getc()* y *putc()*, la mayoría de los compiladores C soportan dos funciones adicionales de E/S de alto nivel : *getw()* y *putw()*. Se utilizan para leer y escribir enteros de dos bytes en o desde un archivo en disco.

Las funciones *getc()* y *putc()* sólo pueden operar sobre caracteres, entonces para leer o escribir enteros de dos bytes de un archivo en disco se utiliza las funciones *getw()* y *putw()*. Estas funciones trabajan exactamente igual que *getc()* y *putc()* con la excepción de que en vez de escribir un único caracter escribe un entero de dos bytes.

La idea general que hay detrás de *getw()* y de *putw()* es que los enteros realmente son de dos bytes de ancho. Por lo tanto, el entero se puede dividir en dos bytes, como un *putw()*, asignando su dirección a un apuntador a caracter y escribiendo un byte cada vez utilizando *putc()*. Lo inverso también es cierto: un entero se puede reconstruir byte a byte asignando su dirección a un apuntador a caracter y ejecutando dos llamadas sucesivas a *getc()*.

Lectura y Escritura de Otros Tipos de Datos

La mayoria de las librerias C no incluyen funciones para leer y escribir cualquier tipo de datos distintos de los de caracter y entero. Sin embargo, se puede escribir estos otros tipos de datos construyendo funciones que operan en forma parecida a *getw()* y *putw()*.

Por ejemplo, si el dato de tipo *float* tuviera ocho bytes de longitud, esta función, *putfloat()*, se podría utilizar para escribir un numero en coma flotante en un archivo en disco :

```
putfloat(num, fp)
float num;
FILE *fp;
{
        char *t;
```

```
    int cuenta;
    r = &num;
    for(cuenta=0; cuenta<8; +=cuenta)
        putc(r[cuenta],fp);
}
```

Se podrían construir funciones *put_x()* y *get_x()* específicas en donde *x* es cualquier estructura de datos o unidad arbitraria. No es necesario que se limite solamente a tipos de datos predefinidos.

Los Archivos stdin, stdout y stderr

Cuando empieza la ejecución de cada programa C, se abren automáticamente tres archivos: entrada estándar (standard input) o *stdin*; salida estándar (standard output) o *stdout*; y error estándar (standard error) o *stderr*. Normalmente, se refiere a la pantalla. Sin embargo, son apuntadores a archivos y se pueden utilizar por el sistema de E/S de alto nivel para ejecutar operaciones de E/S por teclado y pantalla.

En general, *stdin* se utiliza para leer desde el teclado, y *stdout* y *stderr* se utilizan para escribir en la pantalla. Se puede utilizar *stdin*, *stdout* y *stderr* como apuntadores a archivos en cualquier función que utiliza una variable del tipo *FILE.

Debemos recordar que *stdin*, *stdout* y *stderr* no son variables sino constantes, y como tales no pueden ser asignadas o alteradas. Al igual que estos apuntadores a archivos se crean automáticamente al principio del programa, se cierran automáticamente al final; no se debe tratar de utilizar *fclose()* para cerrarlos.

Las Funciones fprintf() y fscanf()

Las funciones *fprintf()* y *fscanf()* se pueden utilizar para escribir varios formatos de datos en un archivo abierto por *fopen()*. El formato general de *fprintf()* es :

fprintf(fp,cadena de control,lista de argumentos);

y el formato general de *fscanf()* es :

fscanf(fp,cadena de control,lista de argumentos);

en donde *fp* es un apuntador a archivo devuelto al llamar a *fopen()*. Excepto en el caso de dirigir su salida al archivo definido mediante *fp*, *fprintf()* y *fscanf()* operan exactamente igual que *printf()* y *scanf()*, respectivamente.

Para ilustrar cómo pueden ser útiles estas funciones, el programa siguiente leerá la información desde el teclado, la escribirá en un archivo en disco, y luego leerá y mostrará la información devuelta en la pantalla.

Debemos tener precaución de que aunque *fprintf()* y *fscanf()* son a menudo la forma más fácil para escribir y leer archivos en disco, no son las más eficaces, ya que los datos ASCII se van a escribir justo como aparecen en la pantalla en vez de en binario. Si nos preocupa la velocidad o el tamaño del archivo, deberemos escribir rutinas de archivo adaptadas similares a *putw()* y *getw()*.

```
main()
{
        FILE *fp;
        char s[80];
        int t;
        if((fp=fopen("prueba","w")) == NULL)
        {
                printf("no se puede abrir el archivo\n");
                exit(0);
        }
        fscanf(stdin,"%s%d",s,&t); /* lee desde el teclado */
        fprintf(fp,"%s %d", s,t);  /* escribe en el archivo */
```

```
fclose(fp);
if((fp=fopen("prueba","r")) == NULL)
{
        printf("no se puede abrir el archivo\n");
        exit(0);
}
fscanf(fp,"%s%d",s,&t);  /* lee desde el archivo */
fprintf(stdout,"%s %d",s,t); /* imprime en pantalla */
```

II.5.2 Archivos de Bajo Nivel

Debido a que originalmente el C se desarrolló bajo el sistema operativo UNIX y a que algunas aplicaciones requieren la posibilidad de dirigir las operaciones de disco a nivel del sistema operativo, se creó un segundo subsistema de E/S en disco.

Las funciones de E/S en disco de bajo nivel para archivos son : *read(), write(), open(), close(), creat(), unlink()* y *lseek()*.

La razón por la que el subsistema de E/S en disco hace que estas funciones se denominen de "bajo nivel" es la de que, como programador, hay que suministrar y mantener todas las memorias intermedias de disco. A diferencia de las funciones *getc()* y *putc()*, que escribían y leían caracteres desde o a un conjunto de datos que automáticamente se escribían o leían desde un archivo en disco, las funciones *read()* y *write()* leerán o escribirán en cada llamada una memoria intermedia completa de información. Hay que colocar la información que se va a utilizar en esa memoria intermedia, y saber cuando se ha llenado. El programador define para su archivo las longitudes de la memoria y del registro.

Los principiantes en C normalmente encontrarán el sistema de E/S de alto nivel más fácil de utilizar y menos propenso a error. Sin embargo, según se avanza, el sistema de E/S de bajo nivel puede ofrecer una mayor flexibilidad y velocidad para algunas aplicaciones.

Las Funciones open(), close() y creat()

A diferencia del sistema de E/S de alto nivel, el sistema de bajo nivel no utiliza apuntadores a archivos del tipo *FILE*, sino algunos descriptores de archivos del tipo *int*.

El formato general para llamar a *open()* es :

```
int fd;
fd = open(nombre archivo, modo);
```

en donde el *nombre del archivo* es cualquier nombre válido y el *modo* es uno de los enteros siguientes :

Modo	*Efecto*
0	*lectura*
1	*escritura*
2	*lectura/escritura*

La función *open()* devuelve un -1 si el archivo no se puede abrir. Sin embargo, la llamada a *open()* será como esta :

```
int fd;
if((fd=open(nombre archivo, modo)) == -1)
{       printf("no puede abrirse el archivo\n");
        exit(0);
}
```

Dependiendo de la realización concreta del compilador C, se podrá utilizar *open()* para crear un archivo que actualmente no existe.

El formato general de *close()* es :

```
int fd;
close(fd);
```

La función *close()* devuelve un -1 si no es posible cerrar el archivo. Esto podría ocurrir, por ejemplo, si el disco se cambio de unidad.

La función *close()* libera el descriptor del archivo, por lo que se puede volver a utilizar para otro archivo. Siempre existe algún límite en el numero de archivos que pueden estar simultáneamente abiertos; por lo tanto se debe utilizar *close()* en un archivo cuando ya no se le necesite.

Si el compilador no permite crear un archivo nuevo utilizando *open()*, o si se quiere asegurar la transportabilidad, se tendrá que utilizar *creat()*. Esencialmente la función *creat()* abre un nuevo archivo para escribir operaciones. El formato general de *creat()* es :

```
int fd;
fd = creat(nombre_archivo, pmodo);
```

en donde el *nombre_archivo* es cualquiera válido. El *pmodo* no se utiliza con la mayoría de los compiladores de C basados en microcomputadoras, aunque seguirá siendo parte de la llamada *creat()*.

Las Funciones write() y read()

Una vez que se ha abierto un archivo para escritura y se ha declarado un arreglo para que actúe como una memoria intermedia, se puede acceder al archivo mediante *write()*. Con cada operación de escritura, la memoria intermedia se escribirá en el disco. También será necesario especificar el tamaño de la memoria intermedia, es decir, el número de bytes que realmente se escriben en el archivo en disco. Generalmente, el número debería ser el mismo que el del tamaño de la memoria intermedia. En algunos sistemas, puede ser necesariamente un múltiplo de 128. En otros puede ser cualquier número.

El formato general de la función *write()* es :

```
#define BUF_SIZE 128
int fd;
char buf[BUF_SIZE];
write(fd,buf,BUF_SIZE);
```

Cada vez que se ejecuta una llamada a *write()*, los caracteres designados por *BUF_SIZE* se escriben en el archivo en disco especificado por *fd* desde el arreglo *buf*. El caracter *buf* no necesita estar terminado por un nulo, porque es una cadena.

La razón por la que no se escribe automáticamente en el disco el contenido completo de la memoria intermedia, es que puede que esa memoria intermedia no esté terminada por el caracter nulo. No existe forma de que la función *write()* conozca la longitud de la memoria intermedia si no se le ha dicho explícitamente. También es posible de que no se quiera escribir toda la memoria intermedia.

La función *write()* devolverá el tamaño de la memoria intermedia después de una operación de escritura correcta. Después de un error, la mayoría de las realizaciones devolverán un -1.

La función *read()* es la complementaria de *write()*. El formato general de *read()* es :

```
read(fd,buf,BUF_SIZE);
```

en donde *fd*, *buf* y *BUF_SIZE* son iguales que para *write()*. Si *read()* es correcta, devuelve el número de caracteres realmente leídos. Devuelve un 0 despues del final físico de archivo, y un -1 si ocurren errores.

El programa siguiente ilustra algunos aspectos de la E/S de bajo nivel. Leerá líneas de texto desde el teclado y las escribirá en un archivo en disco. Después de escritas, el programa las volverá a leer.

```
#include "stdio.h"
define BUF_SIZE 128
main()
{
        char buf[BUF_SIZE];
        int fd1,fd2,i;
        if((fd1=open("prueba",1))==-1) /*abierto para escribir*/
        {
                printf("no puede abrirse el archivo/n");
                exit(0);
        }
        input(buf,fd1);
        /* ahora cierra el archivo y lee hacia atrás */
        close(fd1);
        if((fd2=open("prueba",0))==-1) /*abierto para leer*/
        {
                printf("no puede abrirse el archivo/n");
                exit(0);
        }
        visualiz(buf,fd2);
        close(fd2);
}
input(buf,fd1);
char *buf;
int fd1;
{
        do {
        gets(buf); /* toma caracteres desde el teclado */
        if(write(fd1,buf,BUF_SIZE)!=BUF_SIZE)
                printf("error en escritura\n");
                exit(0);
        }
        while (!strcmp(buf,"quit"));
}
visualiz(buf,fd2);
char *buf;
int fd2;
{
        do {
        gets(buf); /* toma caracteres desde el teclado */
        if(read(fd2,buf,BUF_SIZE) <0){
                printf("error en escritura\n");
                exit(0);
        }
        printf(buf);
        while (!strcmp(buf,"quit"));
}
```

La Función unlink()

Si se quiere quitar un archivo del directorio, se debe utilizar *unlink()*. Aunque *unlink()* se considera parte del sistema de E/S de bajo nivel, quitará cualquier archivo del directorio. El formato general de *unlink()* es :

unlink(nombre del archivo);

en donde el *nombre del archivo* es un apuntador a caracter de cualquier nombre de archivo válido. La función *unlink()* devolverá un error (normalmente -1) si no fuese posible borrar el archivo. Esto puede suceder si el archivo no estuviese presente en el disco al empezar o si el disco estuviera protegido contra escritura.

Archivos de Acceso Aleatorio lseek()

El C soporta archivos de E/S de acceso aleatorio bajo el sistema de E/S de bajo nivel mediante llamadas a *lseek()*. El formato general de *lseek()* es :

```
int fd,origen;
long desplazamiento;
lseek(fd,desplazamiento,origen);
```

en donde *fd* es un descriptor de archivo devuelto al llamar a *creat()* o a *open()*. La forma en que trabaja *lseek()* depende de los valores del origen y del desplazamiento. El origen puede ser 0 ó 1 ó 2. A continuación se explica como se interpreta el desplazamiento para cada valor de origen.

Origen	*Efecto al llamar lseek()*
0	*Cuenta el desplazamiento desde el principio del archivo*
1	*Cuenta el desplazamiento desde la posición actual*
2	*Cuenta el desplazamiento desde el final del archivo*

Un ejemplo sencillo de la utilización de *lseek()* es el programa siguiente. Para que funcione hay que especificar la memoria intermedia específica sobre la que se quiere leer. La introducción de un número negativo permitirá salirse. Se puede cambiar el tamaño de la memoria intermedia para que se iguale al tamaño del sector del sistema, aunque no es necesario.

```
#include "stdio.h"
#define BUF_SIZE 128
main(argc,argv) /* lee y escribe */
int argc;
char *argv[];
{
        char buf[BUF_SIZE+1],s[10];
        int fd, sector;
        buf[BUF_SIZE+1]='\0'; /* terminador nulo del buffer */
        if((fd=open(argv[1],0))==-1) /*abierto p/escribir*/
                printf("no puede abrirse el archivo\n");
                exit(0);
}
        do
                {
                gets(s);
                sector=atoi(s); /*toma sector p/leer*/
                if(lseek(fd,sector*BUF_SIZE,0)==-1)
                        printf("error de busqueda\n");
                if(read(fd,buf,BUF_SIZE)==0)
                {
                        printf("sector fuera de rango\n");
                }
                else
                        printf(buf);
                }
                while(sector>0);
        close(fd);
}
```

II.6 PROGRAMACIÓN DE BAJO NIVEL

La programación de Bajo Nivel se explicó detenidamente en el apartado II.5.2. Para completar este concepto es necesario incluir dos puntos importantes como lo son Operaciones y Campos de Bits.

II.6.1 Operaciones de Bits

Al contrario que en otros lenguajes, C soporta un completo juego de operadores de bits. Ya que el C se ha diseñado para sustituir al lenguaje ensamblador en muchas tareas de programación.

Las operaciones de bits se refieren a la comprobación, colocación, o desplazamiento de los bits actuales de una variable entera o de caracter. Estas operaciones no pueden utilizarse sobre los tipos *float* ni *double*. A continuación se presentan estas operaciones.

Operador	*Acción*
&	*AND*
|	*OR*
^	*OR exclusivo*
~	*Complemento a uno*
>>	*desplazamiento a la derecha*
<<	*desplazamiento a la izquierda*

Las operaciones sobre bits son frecuentes en las aplicaciones de controladores de dispositivos (programas de modem, rutinas de archivos en disco y rutinas de la impresora) ya que permiten enmascarar ciertos bits, como el de paridad (el bit de paridad se utiliza para confirmar que el resto de los bits del byte no se han cambiado. Siempre es el bit de mayor peso de cada byte).

Se puede imaginar que la operación AND es una forma de poner a cero los bits. Con AND, cualquier bit que esté a 0 en el operando, hará que el correspondiente bit de la variable se ponga a 0. Por ejemplo, la siguiente función leerá un caracter en el puerto del modem, utilizando la función *leer_modem()* y pondrá el bit de paridad a cero :

```
char ca;
tomar_ca_del_modem()
{       ca=leer_modem(); /* toma un caracter del puerto del modem */
        return(ca & 127);
}
```

La paridad la indica el octavo bit, que se pone a cero, al hacer un AND con un byte que tiene a 1 los bits del 1 al 7, y el bit 8 a 0. La expresión *ca & 127* significa hacer el AND entre los bits de *ca* y los bits que forman el número 127. El resultado es que el octavo bit de *ca* se pone a 0. En el ejemplo siguiente, se supone que *ca* ha recibido el caracter 'A' y que tiene el bit de paridad a 1.

```
        bit de paridad
        1 1 0 0 0 0 0 1      ca conteniendo 'A' y a 1 el
&       0 1 1 1 1 1 1 1      de paridad 127 en binario
        ---------------      se hace el AND bit a bit
        0 1 0 0 0 0 0 1      'A' con el bit de paridad a 0
```

El OR bit a bit puede utilizarse para poner los bits a 1. Cualquier bit del operando que este puesto a 1 hará que su correspondiente bit en la variable se ponga a 1. Por ejemplo, si hacemos 128 | 3 :

```
        1 0 0 0 0 0 0 0      128 en binario
        0 0 0 0 0 0 1 1      3 en binario
|       ---------------      OR bit a bit
        1 0 0 0 0 0 1 1      resultado
```

Un OR exclusivo normalmente abreviado como XOR, pondrá a uno sólo los bits que al compararlos sean distintos. Por ejemplo, 127^120 es :

```
  01111111     127 en binario
  01111000     120 en binario
^ --------     XOR bit a bit
  00000111     resultado
```

En general, las operaciones AND, OR y XOR se aplican de forma individual entre cada pareja de bits. Por estas y otras razones, las operaciones entre bits no se utilizan en las sentencias condicionales, en el mismo sentido, que se utilizan los operadores lógicos.

Por ejemplo, si x=7, entonces x && 8 se evalúa como cierto, ó 1, mientras que x & 8 se evalúa como falso, ó 0. Los operadores relacionales y lógicos siempre producen un resultado que es o cero o uno; análogamente las operaciones entre bits modifican la variable de acuerdo con la operación especifica. En otras palabras las operaciones entre bits se utilizan para cambiar el valor de las variables, no para evaluar si las condiciones son ciertas o falsas.

Los *operadores de desplazamiento*, >> y <<, mueven todos los bits de una variable a la derecha o a la izquierda, según se especifique. La forma general de una sentencia de desplazamiento a la derecha es :

variable >> número de posiciones en bits

y de una sentencia de desplazamiento a la izquierda es :

variable << número de posiciones en bits

A medida que se desplazan los bits hacia el extremo izquierdo, se va rellenando con ceros por el extremo derecho. Un desplazamiento no es una rotación; o sea que los bits que salen

por la izquierda, no se introducen por la derecha. Los bits que salen por la izquierda son perdidos, introduciéndose ceros por la derecha. Los operadores de desplazamiento también pueden utilizarse para llevar a cabo operaciones muy rápidas de multiplicación y división de enteros. Un desplazamiento a la izquierda equivale a una multiplicación por 2, y un desplazamiento a la derecha a una división por 2, como se muestra a continuación. Se supone que los huecos se rellenan con ceros y que los unos que salen son perdidos.

char x;	*x después de c/ejecución*	*Valor de x*
x=7;	*00000111*	*7*
x<<1;	*00001110*	*14*
x<<3;	*01110000*	*112*
x<<2;	*11000000*	*192*
x>>1;	*01100000*	*96*
x>>2	*00011000*	*24*

El operador de complemento a uno, ~, cambiará el estado de cada bit en la variable especificada; o sea los unos serán ceros y los ceros unos.

Los operadores sobre bits se usan a menudo en las rutinas cifradas. Si queremos que un archivo aparezca ilegible, basta que lleve a cabo con él algunas manipulaciones a nivel de bits. Uno de los métodos más sencillos es el de complementar a uno, invirtiendo cada bit de un byte así :

byte original	*00101100*
después del primer complemento	*11010011*
después del segundo complemento	*00101100*

Observemos que una secuencia de dos complementos sobre una fila, produce el número original. Por ello el primer complemento puede representar la versión codificada; y el segundo la decodificación con el valor inicial.

Para codificar un archivo utilizando el operador de complemento a uno, puede usarse la función *codif()* :

```
codif(ca)  /* una sencilla función de cifras */
char ca;
{
        ca=~ca;  /* lo complementa */
        return(ca);
}
```

11.6.2 Campos de Bits

A diferencia de la mayoría de los demás lenguajes, C incluye un método para acceder a un único bit dentro de un byte. Esto resulta útil por varias razones :

- *primera*, si la capacidad de almacenar está limitada, se pueden almacenar algunas variables boolenas (cierto/falso) en un byte.
- *segunda*, algunas interfases de dispositivos transmiten información codificada en bits dentro de un byte.
- *tercera*, algunas rutinas criptográficas necesitan acceder a los bits dentro de un byte.

Aunque todas estas funciones se pueden realizar utilizando bytes y los operadores de bit, el campo de un bit puede añadir una mayor estructura al código y hacerlo más portátil.

El método que C utiliza para acceder a los bits está basado en la estructura. Esta estructura define tres variables de un bit de cada una :

```
struct dispositivo
{
        unsigned activo : 1;
        unsigned listo  : 1;
        unsigned xmt_error : 1;
}
codi_dispositivo;
```

Las tres variables se han declarado como *unsigned* porque un bit no puede tener un signo. Los únicos valores que puede tener un bit son 0 y 1. La variable estructura *codi_dispositivo* se podrá utilizar para decodificar información desde el puerto de una cinta magnética, por ejemplo. El siguiente fragmento de código escribirá un byte de información en la cinta y comprobará los errores utilizando *codi_dispositivo* :

```
wr_cinta(c)
char c;
{
        while(!codi_dispositivo-activo)rd(&codi_dispositivo); /*espera*/
        wr_to_tape(c); /* escribe un byte */
        while(codi_dispositivo-activo)rd(&codi_dispositivo); /*espera
                                hasta que la información se escriba */
if(dev_code.xmt_error) printf("error de escritura");
}
```

Aquí; *rd()* devolverá la situación de la cinta y *wr_cinta()* escribe realmente el dato. Se muestra que la variable campo_bit *codi_dispositivo* aparece como en memoria. Como se puede ver, a cada campo de bit se accede utilizando la estructura del operador punto. Sin embargo, si la estructura se pasa a una función hay que utilizar el operador ->.

No hay que nombrar cada campo de bit. Esto facilita el camino hasta el bit que se quiere. Por ejemplo, si la unidad de cinta también devuelve una señal de final de cinta en el bit 5, para acomodar esto se podría alterar la estructura dispositivo utilizando :

```
struct dispositivo
                {
                        unsigned activo : 1;
                        unsigned list : 1;
                        unsigned xmt_error : 1;
                        unsigned : 1;
                        unsigned : 1;
                        unsigned EOT : 1;
                } codi_dispositivo;
```

Las variables de campo de bit tienen algunas restricciones. No se puede tomar la dirección de esa variable. No pueden ser arreglos. No se pueden sobrepasar sus límites enteros. De una computadora a otra, no se puede saber si los campos funcionarían de derecha a izquierda o viceversa; esto implica que cualquier código que utilice campos de bit puede tener algunas dependencias de máquina.

CAPÍTULO III

PROGRAMACIÓN ORIENTADA A OBJETOS

III.1 PROGRAMACIÓN ORIENTADA A OBJETOS

La Programación Orientada a Objetos (POO), es una forma nueva de aproximarse a las tareas de programación. Las aproximaciones de programación han ido cambiando desde la invención de la computadora. La razón principal de los cambios han sido adaptarse a la creciente complejidad de los programas.

Los años 60's vieron nacer la Programación Estructurada. Este es es el método que propician los lenguajes como C y Pascal. Al utilizar lenguajes estructurados fue posible por primera vez, escribir programas moderadamente complejos con bastante facilidad. Sin embargo, incluso con métodos de programación estructurada, cuando un proyecto alcanza unas ciertas dimensiones se vuelve incontrolable, ya que su complejidad sobrepasa lo que puede gestionar un programador que utiliza las técnicas de programación estructurada. Para resolver este problema, se ha inventado la Programación Orientada a Objetos .

La POO ha tomado las mejores ideas de la programación estructurada y las ha combinado con varios conceptos nuevos y potentes que incitan a contemplar las tareas de programación desde un nuevo punto de vista. La POO permite descomponer más fácilmente un problema en subgrupos de partes relacionales de problema. Entonces, utilizando el lenguaje, se pueden traducir estos subgrupos a unidades autocontenidas, llamadas objetos.

Todos lo lenguajes de la POO tienen tres cosas en común: objetos, polimorfismo y herencia.

III.1.1 Definición del Problema

Esta es la primera etapa de la fase de planeación, dentro del ciclo de vida del sistema, en la cual se trata de describir en forma general los requerimientos del usuario y plantear un conjunto de objetivos, que de lograrse implicarán la satisfacción de las necesidades del cliente. El definir claramente los objetivos es condición necesaria para el éxito de todo proyecto, de no hacerlo así, puede propiciar una mala interpretación que después se traduzca en productos muy complejos de lo requerido.

La definición requiere un entendimiento cabal del dominio del problema y del entorno de éste. Este trabajo consiste en tener una serie de entrevistas con el cliente, observaciones de las tareas problemáticas y desarrollo de las reales. El analista debe ser hábil en la técnica de definición del problema, ya que distintos representantes del cliente tendrán diferentes puntos de vista, sesgos y perjuicios que influirán en su percepción del área del problema. También el analista debe exponer el problema en forma clara a su equipo de programación, de no hacerlo así existirán problemas de comunicación que a la larga repercutirá en la solución correcta.

En este momento, después de analizar y comprender de manera general el problema y presentar una posible solución, se orientan las concepciones del entorno de problema hacia lo que se llama POO.

III.1.2 Identificación de Objetos y Clases

Al igual que la identificación de los módulos (subproblemas) es para la programación estructurada, la identificación de los Objetos y Clases es para la POO. Aquí es donde se analiza la característica individual más importante de un lenguaje orientado a objetos, que son los objetos mismos, que representan en analogía con la PE los módulos.

Objetos

En pocas palabras, un objeto es una entidad lógica que contiene datos y un código que manipula esos datos. Dentro de un objeto, parte del código y/o de los datos puede ser privado para ese objeto, y no será accesible directamente para nada de lo que haya fuera de ese objeto. De esta manera, un objeto proporciona un nivel significativo de protección contra alguna otra parte del programa que no esté relacionada con él, y que pudiera modificar o utilizar incorrectamente las partes privadas de objeto. El enlazado de código y de datos de esta manera es a lo que se llama *encapsulación*.

A todos los efectos, un objeto es una variable de un tipo definido por el usuario. Quizá parezca extraño al principio pensar que un objeto, que enlaza un código y unos datos, es una variable. Sin embargo, esto es precisamente lo que sucede en la programación orientada a objetos. Cuando se define un objeto, se está creando implícitamente un nuevo tipo de datos.

Un objeto en el mundo real es algo con masa, un peso asociado y un volumen (una computadora, una impresora, etc.); algo que se puede ver o tocar, que permite su descripción fiel, y en algunos casos puede existir algún procedimiento o acción que describe su función; es decir, un objeto es algo con un conjunto de atributos y características que describen su naturaleza y funcionalidad.

Nuestro mundo esta lleno de objetos, de modo que parece natural su descripción y la resolución de problemas en términos de objetos. Esta idea es la base de la POO, aunque se necesitan otros atributos además de los objetos para que un lenguaje pueda ser considerado orientado a objetos.

Los objetos pertenecen a categorías o clases de objetos. Estas clases, a su vez, pueden ser subclases de clases más externas. Cada subclase está conectada a su antecesor (padre) mediante un enlace.

Clases

Por su parte, una clase es un objeto encargado de crear objetos basados en reglas preestablecidas, es decir, crea código y datos correlacionados en función a un modelo. Esto implica, que pueden existir objetos de la misma clase. Una clase es una fábrica de objetos.

Es muy clara la diferencia que en los lenguajes modernos de programación se hace entre la definición de un tipo y la declaración de una variable. Por ejemplo, con la declaración *typedef* del lenguaje C.

```
typedef struct { char*nombre;
                 int ficha;
LIST_EMP empleado
```

En este ejemplo se define el nuevo tipo *LIST_EMP*, que es una estructura con dos campos, y en la tercera línea se hace la declaración de una variable del tipo *LIST_EMP* llamada ***empleado***.

En lenguajes orientados a objetos la definición de un tipo se conoce como una clase; en este aspecto se trata del mismo concepto. Al definir tipo, como se hizo en el ejemplo anterior con la declaración *typedef*, se pueden crear nuevas entidades (variables) en el programa, las cuales tendrán las características definidas en la declaración *typedef*. Esto es, el tipo *LIST_EMP* funciona como un molde o plantilla a partir del cual se crean nuevas variables en el programa (como es el caso de "*empleado*"). Similarmente, en un lenguaje orientado a objetos, una clase funciona como una plantilla a partir de la cual se crean objetos en el programa. En la definición de una clase están incluidas las variables muestra (estado o representación), y los métodos (operaciones) para los objetos de tal clase. La diferencia de la definición de una clase en el ejemplo anterior sería la inclusión de algunas funciones dentro de la estructura, de manera que las funciones quedaran asociadas con los tipos de datos dentro de la estructura.

Una clase puede considerarse como un patrón o plantilla a partir de la cual se pueden crear objetos. Los diferentes objetos derivados de una misma clase tienen operaciones comunes y, por lo tanto, presentan un comportamiento uniforme. La clase provee un mecanismo para que los diferentes objetos derivados de ella puedan compartir las mismas funciones.

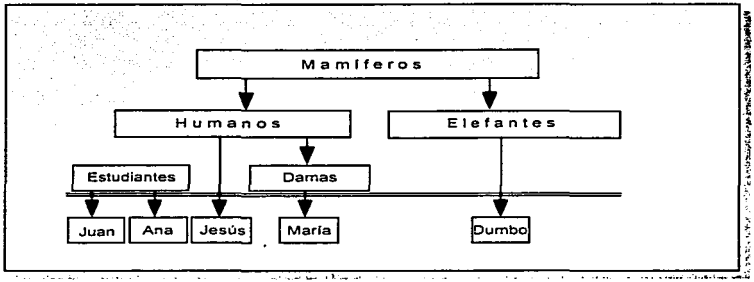
III.1.3 Herencia

La herencia es el proceso mediante el cual un objeto puede adquirir las propiedades de otro objeto. Esto es importante porque sirve de base para el concepto de clasificación. Si se piensa en esto, la mayor parte del conocimiento resulta de una clasificación jerárquica. Por ejemplo, una *manzana golden* es parte de la clase *manzanas*, que a su vez forma parte de la clase *frutas*, que pertenece a una clase más grande *comida*. Si se utilizan las clasificaciones, cada objeto tendría que definir explícitamente todas sus características. Sin embargo, cuando se usan las clasificaciones, los objetos sólo necesitan definir aquellas características que los hacen únicos dentro de su clase. El mecanismo de herencia es el que hace posible que un objeto sea un caso concreto de un caso mas general.

La herencia es una propiedad (mecanismo) que permite a un objeto heredar propiedades de otra clase de objetos. La herencia permite a un objeto contener sus propios procedimientos o funciones y heredar los mismos de otros objetos. Sólo tipos de objetos pueden heredar características de otros tipos de objetos. La herencia no es posible para otras estructuras de datos (tales como registros).

Uno de los mecanismos más poderosos para manejar algo complejo es una estructura jerárquica. Un ejemplo clásico de esto es una clasificación taxonómica de insectos, o de mamíferos como la que se muestra a continuación. Se pueden apreciar dos subclases de los *mamíferos* (*humanos y elefantes*), y dos subclases de los *humanos* (*estudiantes y damas*). Las muestras (abajo de la doble línea) se agrupan en clases, mientras que las clases (arriba de la doble línea) están categorizadas por sus superclases. En este tipo de clasificación, un

objeto sólo puede pertenecer a una clase base y, además, las muestras heredan todas las características de sus clases base (las de los *mamíferos*, en este ejemplo).



En ingeniería de software, la herencia se utiliza no sólo para propósitos de clasificación, sino también para conceptuar la evolución de los sistemas y la modificación de los mismos conforme se modifican y crecen. La característica de la herencia, de ser flexible en la especificación de un cambio, es una herramienta valiosa en la computación, pero requiere de un mecanismo que es fundamentalmente más poderoso que la clasificación.

En la POO, la herencia es el mecanismo por el cual, al ser derivado un objeto de una clase, el primero tendrá los mismos atributos (datos) y el mismo comportamiento (funciones) que el segundo. Además, la herencia en la POO permite que un objeto pueda modificar o prescindir de las funciones heredadas de su ancestro. El objeto es una entidad modificada con respecto a su ancestro.

Cuando un objeto puede tener únicamente un ancestro o, en otras palabras, un objeto sólo puede derivarse de una clase, se dice que se tiene herencia lineal o simple. Este es el tipo de

herencia que permite un gran número de lenguajes orientados a objetos. Sin embargo, otros lenguajes (C++, Smaltalk, CLOS) permiten que un objeto sea derivado de más de una clase; en este caso se dice que se tiene herencia múltiple o no lineal.

III.2 LOS MÉTODOS

Una característica importante en la POO es la capacidad para especificar métodos cuando se define un objeto. Un método es una rutina asociada con un tipo objeto específico. Esta rutina está disponible a cualquier objeto y objetos descendientes. En esencia es una parte de una definición del tipo objetos.

Un método es un procedimiento o función que está asociado a un tipo de objeto. No se pueden asociar métodos con otros tipos de datos. Los procedimientos y funciones declarados dentro de un objeto se conocen como métodos.

Los métodos se definen en dos partes en el programa: las cabeceras de los procedimientos y funciones se especifican en la declaración del tipo objeto, y la implementación de estos subprogramas aparece fuera de la definición tipo.

III.2.1 Métodos Estáticos

Los Métodos Estáticos son similares en su planteamiento a las variables estáticas. El compilador los asigna y resuelve todas las referencias a ellas en *tiempo de compilación.*

Los métodos estáticos son menos complicados: requieren menos memoria y se ejecutan mas rápidamente; presentan el inconveniente de que no son los mas óptimos para la POO.

Los métodos se llaman estáticos si no pueden cambiar después de la compilación.

III.2.2 Métodos Virtuales

Los métodos examinados hasta ahora son estáticos. En muchas ocasiones es necesario que todas las referencias se resuelvan en tiempo de ejecución. En este caso se necesita una ligadura tardía que significa que el camino exacto de ejecución se determina en tiempo de ejecución y no en tiempo de compilación. Esta operación se realiza utilizando una tabla de método virtual (TMV). Una TMV es una tabla de direcciones que apunta a procedimientos y funciones. Manteniendo una tabla de direcciones para cada tipo de objeto, el lenguaje orientado a objetos puede determinar un camino de ejecución que será imposible determinar en tiempo de compilación.

III.3 POLIMORFISMO

El Polimorfismo, corresponde al hecho de que bajo un mismo mensaje, los objetos se comporten de forma diferente. El polimorfismo es equivalente a "muchos comportamientos". Los lenguajes de POO admiten el Polimorfismo, que en esencia significa que un nombre se puede utilizar para especificar una clase genérica de acciones. Sin embargo, y dependiendo del tipo de datos con que esté trabajando, se ejecuta una variante concreta del caso general.

Por ejemplo, se puede tener un programa que defina tres tipos de pilas. Una pila se utiliza para valores enteros, una para valores de coma flotante y otra para valores largos *(long)*. Como consecuencia del polimorfismo, se pueden crear tres grupos de funciones para estas pilas denominadas poner() y quitar(), y el compilador seleccionará la rutina correcta dependiendo del tipo de dato con el cual se invoque. En este ejemplo, el concepto general es el de poner y quitar datos de una pila. Las funciones definen el modo concreto en que se hace esto para cada tipo de datos.

Los primeros lenguajes de la POO eran intérpretes. así que en el polimorfismo se permitía en el momento de la ejecución. Sin embargo, C++ es un lenguaje compilado. Por lo tanto, se admiten tanto el polimorfismo en ejecución como en compilación.

III.4 CONSTRUCTORES Y DESTRUCTORES

Los constructores y destructores son rutinas, o métodos, que se asocian con un objeto particular. Desde el punto de vista sintáctico, un método tal como constructor es similar a un procedimiento (ejecuta alguna tarea relativa a un objeto particular).

Un constructor puede inicializar los campos de una variable objeto especifico. Este constructor puede ser utilizado por otros objetos que se definan como casos especiales del objeto con el cual se asoció el constructor.

El constructor advierte al compilador que un método particular se utilizará para establecer una estructura de datos manipulada por un método virtual.

El destructor, denominado así porque limpia y dispone de objetos asignados dinámicamente. Como con cualquier método, se pueden definir destructores múltiples para un sólo tipo de objeto.

Un destructor se define con todos los restantes métodos de objetos en la definición de tipos objetos.

Los destructores pueden ser heredados y estáticos o virtuales, aunque es recomendable que sean virtuales de modo que en cada caso el destructor correcto se ejecutará para su tipo objeto.

III.5 SOBRECARGA DE FUNCIONES

Una de la maneras que tiene un lenguaje de POO de llegar al polimorfismo, es a través de la sobrecarga de funciones. Dos o más funciones pueden compartir un nombre siempre y cuando las declaraciones de sus parámetros sean diferentes. En esta situación, se dice que las funciones que comparten el mismo nombre están sobrecargadas, y el proceso se denomina sobrecarga de funciones.

Se puede utilizar el mismo nombre para sobrecargar funciones que no están relacionadas, pero no se debe hacer. En la práctica, sólo se deben sobrecargar operaciones que están íntimamente relacionadas.

III.6 RESOLUCIÓN DE PROBLEMAS CON POO

La POO, está basada en una filosofía distinta a la PE hasta este momento. Por esta razón, aunque las ideas fundamentales de programación se mantienen, es necesario que a la hora de resolver problemas utilizando la POO se piense de una manera diferente. La secuencia de acciones o tareas a realizar por un programador orientado a objetos para resolver un problema son :

- Definir el problema
- Identificar los objetos y sus clases
- Determinar los métodos requeridos para cada clase
- Escribir el programa principal
- Determinar los elementos que integran cada clase
- Realizar los métodos de cada clase

Conclusiones

Con la Programación Estructurada elaborar programas sigue siendo una labor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este estilo podemos obtener las siguientes ventajas :

1. Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están más ligadas o relacionadas entre lo que hace cada función.

2. Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas se facilita, debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.

3. Reducción de los costos de mantenimiento.

4. Programas más sencillos y más rápidos.

5. Aumento de la productividad del programador.

6. Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación.

7. Los programas quedan mejor documentados internamente.

C a menudo es conocido como un lenguaje de medio nivel, que proporciona a los programadores un conjunto mínimo de sentencias de control y de manipulación de datos, que pueden utilizarse para definir construcciones de alto nivel. Los lenguajes de medio nivel se ven a veces como lenguajes basados en bloques predefinidos, ya que el programador crea previamente las rutinas necesarias para realizar las funciones que necesita, y luego las junta entre sí.

El código C es muy portable. *Portabilidad* significa que el software escrito para un determinado tipo de computadora, puede adaptarse a otro tipo.

Hay varias razones por las que se utiliza el lenguaje C para Programación de Sistemas. A menudo los programas de sistemas deben ejecutarse muy rápidamente. Los programas generados por los compiladores de C, suelen ejecutarse tan rápidamente como los escritos en lenguaje ensamblador. El escribir en lenguaje ensamblador es un trabajo duro y tedioso, y como el código C escribe más rápidamente que en ensamblador, el C reduce tremendamente los costos.

El lenguaje C es muy legible, una vez familiarizado con él, se puede seguir el flujo de control y la lógica de un programa, así como una fácil verificación de las operaciones de las subrutinas.

C es un lenguaje estructurado y utilizado actualmente en la Programación Orientada a Objetos en sus diferentes versiones. La característica básica de un lenguaje estructurado es la utilización de bloques. Un bloque es un conjunto de sentencias que están relacionadas lógicamente.

Un lenguaje estructurado proporciona diversas posibilidades de programación; soporta el concepto de subrutinas con variables locales. Una variable local no es más que una variable que sólo es conocida por la subrutina en la que ha sido definida. Un lenguaje estructurado

también soporta varias construcciones de bucles, como el *while*, *do while* y el *for*, permite el compilar por separado a las subrutinas, sin necesidad de que formen parte de un programa propiamente dicho. Esto permite el crear una librería de subrutinas con funciones útiles y ya probadas, a las que se puede acceder desde cualquier programa que se escriba.

Los lenguajes estructurados se consideran más modernos, llegando a ser considerada como característica de un lenguaje de computadora antigüo el que no sea estructurado. Debido a su claridad, los estructurados no sólo hacen más fácil el programar, sino que incluso resulta más fácil su mantenimiento ya que se basan en la separación de funciones y datos, llevando a que cada tarea se reduzca a su propia subrutina o bloque de código.

La Programación Orientada a Objetos se presenta como una técnica que brinda grandes ventajas respecto de la Programación Estructurada. No obstante que durante mucho tiempo sólo existió como una curiosidad académica, en la actualidad la Programación Orientada a Objetos se esta incorporando rápidamente a las ciencias de la computación. No es posible desconocer esta técnica, ya que es el futuro inmediato en varias áreas de la computación. Cualquier persona que tenga que ver con la computación esta seriamente comprometido con ésta técnica.

Dentro de las principales ventajas que se obtienen con la Programación Orientada a Objetos se pueden citar:

1. *Mantenimiento* : Los programas hechos con POO se leen y se escriben más fácilmente que los que se crean con técnicas estructuradas, lo cual facilita su mantenimiento.
2. ***Reutilización*** : Es posible escribir código para objetos que puedan ser considerados como cajas negras, estas a su vez, pueden ser usadas como partes estándar de software en futuras aplicaciones, aún sin cambio alguno. Es posible crear bibliotecas de objetos, que permiten

ensamblar rápidamente aplicaciones completas a partir de partes prefabricadas.

3. *Extensibilidad* : Un beneficio del concepto de herencia de la POO es que los objetos se pueden extender fácilmente con nuevas características sin necesidad de duplicar el código.

Bibliografía

1. JOYANES AGUILAR, LUIS. **"La Metodología de la Programación : Diagramas de Flujo, Algoritmos y Programación Estructurada".**
   Ed. McGraw Hill. 248 pp.

2. NEWCOMER, LAWRENCE R. **"Programación en Cobol Estructurado".**
   Ed. McGraw Hill. 377 pp.

3. PARDO ORTIZ, EFRAIN. **"Metodología para el Desarrollo de Sistemas de Información".**
   Facultad de Ingeniería 1991. División de Educación Continua, UNAM.

4. PRESSMAN, ROGER S. **"Ingeniería del Software".**
   Ed. McGraw Hill. Segunda Edición. 628 pp.

5. FAIRLEY, RICHARD. **"Ingeniería del Software".**
   Ed. McGraw Hill.

6. **"Programación Estructurada".**
   PUC (Programa Universitario de Cómputo). 243 pp.

7. COLEMAN, D. **"Organización de Datos y Programación Estructurada".**
   Ed. Gustavo Gili. S.A. 1990.

8. NELL, DALE. **"Pascal y Estructuras de Datos"**

9. **"El enfoque de Sistemas a la Solución de Problemas".**
   Facultad de Ingeniería 1991. División de Educación Continua. UNAM.

10. KERNIGHAN, BRIAN W. Y R. DENNIS M. **"El Lenguaje de Programación C".**
    Ed. Prentice-Hall Hispanoamericana. 235 pp.

11. SCHILDT, HERBERT. **"Aplique Turbo C++".**
    Ed. McGraw Hill. 579pp.

12. SCHILDT, HERBERT. **"Programación en Turbo C".**
    Ed. McGraw Hill. 382 pp.

13. SCHILDT, HERBERT. **"Programación en Lenguaje C".**
    Ed. McGraw Hill. 284 pp.

14. SCHILDT, HERBERT. **"C Manual de Referencia".**
    Ed. McGraw Hill. 749 pp.

15.JOYANES AGUILAR, LUIS. **"Programación en Turbo Pascal".**
Ed. McGraw Hill. 822 pp.

16.**"Programación Orientada a Objetos".**
Facultad de Ingeniería 1990. División de Educación Continua, UNAM.

17.RESKALA RUIZ. ALEJANDRO. **"PC/Tips, Julio 1991".**
Sección Herramientas de Desarrollo. Art. Programación Orientada a Objetos en Clipper.

18.CARDENAS LAILSON, DANIEL. **"Programación Orientada a Objetos"**
Centro de Cálculo, División de Investigación y Desarrollo.
IPN.42 pp.