UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

"ARAGON"

Lenguaje C como Herramienta a la Ingeniería en el Desarrollo de Sistemas

TESIS PROFESIONAL

Que para obtener el Título de:

INGENIERO EN COMPUTACION

Presenta:

RICARDO RAMIREZ SALVADOR

San Juan de Aragón, Méx. 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

C A P I T U L O V I I

TIPOS DE DATOS CONGLOMERADOS

C A P I T U L O V I I I

PREPROCESADOR DE C

C A P I T U L O I X

APLICACION DEL LENGUAJE C

PROLOGO

PROLOGO

El lenguaje C puede ser considerado como una herramienta a la Ingeniería para el desarrollo de sistemas.

Este trabajo pretende mostrar las principales características que tiene C, para explotar al máximo los recursos con que cuenta una computadora.

El objetivo básico de la tésis se presenta en un apartado, en donde se explica el papel que desarrolla C en la elaboración de sistemas útiles para el control presupuestal de la Subdirección de Transformación Industrial (S.T.I.) de Petróleos Mexicanos.

El capítulo I llamado Introducción (al lenguaje C) pretende presentar y ubicar al Lenguaje C. Se habla acerca de sus orígenes, sus características y usos que lo hacen sobresalir ante otros lenguajes, se mencionan los principales compiladores existentes, y la estructura misma de un programa en C.

El capítulo II, Conceptos Básicos de C, habla acerca de los diferentes tipos de datos tantos básicos como avanzados que maneja C, además presenta la clasificación que se tiene de variables, constantes y operadores, se habla de conceptos básicos como son identificadores, secuencias de escape, palabras claves, la precedencia y asociatividad de operadores, conversiones de tipo y construcciones de expresiones.

El capítulo III, Sentencias, hace referencia a los diferentes tipos de sentencias que puede poseer C como son, las sentencias de expresión, compuestas, de control, y de selección. Se hace una clasificación de las mismas y se presenta su sintaxis, su uso y su aplicación ante diferentes situaciones.

El capítulo IV, Array y Punteros, trata aspectos importantes y básicos en C. Se explica la forma en que un array está representado en la memoria, se habla además de los punteros, como lo es su definición, declaración, inicialización, aritmética y expresiones con punteros. Se abarcan conceptos tan importantes como son los array de punteros, punteros a funciones y paso de punteros a una función.

El capítulo V, Funciones, presenta las características de una función en C, tales como su definición, declaración, reglas de ámbito, argumentos y vuelta de una función. Se habla además de la diferencia que existe entre la declaración de una función en forma tradicional ante la forma moderna también llamada prototipo de funciones.

El capítulo VI, Sistema de E/S, habla principalmente de funciones de biblioteca que ofrecen una forma simple y sencilla para transferir datos entre dispositivos. Se hace una clasificación de las mismas, E/S por consola y E/S por archivos. La primera agrupa generalmente a todas aquellas funciones que

controlan la entrada por teclado y la salida por pantalla aunque se puede redireccionar a otro dispositivos. Y la segunda básicamente agrupa a todas aquellas funciones que crean y procesan archivos de datos. También en este capítulo se habla de lo que es un puntero a un archivo (tipo FILE), en donde se definen cosas importantes como son el nombre, estado y posición actual del archivo.

El capítulo VII, Tipos de Datos Conglomerados, habla básicamente sobre las estructuras en C, su definición, asignaciones y elementos, así como los array de estructuras, paso de estructuras a funciones, punteros a estructuras y una modalidad de este tipo como son los campos de bits. Otro tipo de dato avanzado que se define son las uniones, las cuales tiene un valor importante en C ya que permiten el ahorro de memoria. Y por último se mencionan las enumeraciones y los tipos definidos por el usuario.

El capítulo VIII, Preprocesador de C, presenta las principales directivas con que cuenta el preprocesador, como son la sustitución de macros, inclusión de archivos, compilación condicional, etc..

El capítulo IX, Aplicación del Lenguaje C, presenta el Sistema de Control del Ejercicio Presupuestal (S.C.E.P.) y el Módulo de Control de Información Financiera (C.I.F.). Se definen sus objetivos, la estructura de sus bases de datos, sus campos de aplicación y los beneficios que ofrecen que son básicamente el tener la información actualizada y el llevar un control eficiente y oportuno del ejercicio presupuestal de la S.T.I.

En este capítulo se mecionan las características principales que tiene el manejador de base de datos Informix 3.30 en el cual fueron hechos los sistemas, de ALL-II C que es un conjunto de librerías que ayudan en un momento dado a la manipulación de los datos de archivos propios de informix, y de archivos de cabecera, útiles para definir y establecer el ambiente de trabajo.

Por último se presentan las rutinas en C que ayudan a mejorar los procesos especiales de los sistemas. En cada rutina se especifica una breve descripción de su aplicación, se detalla paso a paso el contenido de la misma y su relación con los archivos propios de Informix. La pantalla principal de módulo C.I.F. es un claro ejemplo de como C puede ayudar a tener un mejor control de la información.

OBJETIVO

O B J E T I V O

El objetivo de éste trabajo es el de resaltar las principales características del lenguaje C, así como su aplicación en el desarrollo de sistemas y en particular, a sistemas que hemos desarrollado (utilizando conjuntamente el manejador de bases de datos Informix 3.30) para el manejo presupuestal de la Subdirección de Transformación Industrial de PEMEX.

El lenguaje C no está orientado a ninguna área en especial, por lo que es considerado un lenguaje de programación de propósito general, que puede servir tanto para el desarrollo de editores de texto, controladores de red, bases de datos, compiladores, sistemas operativos, juegos de video, sistemas hechos a medida, etc..

Además, se puede llevar acabo una codificación tanto en alto y bajo nivel, esto ayuda en gran medida a la manipulación de datos a nivel de bits, bytes y direcciones de memoria, logrando con ello un control directo de la E/S de información en la computadora así como para el área de memoria.

El lenguaje presenta una gran variedad de operadores que conjuntamente con sentencias de control, permiten la construcción de expresiones tanto simples como complejas. Esto ayuda en un momento dado a minimizar el código de programas. Otros aspectos importantes y que se hablan de ellos en este trabajo son los tipos de datos, tanto los básicos como los definidos por el usuario, esto puede garantizar la portabilidad de los programas, las clases de almacenamiento tanto de tipo global como local , arrays, punteros, funciones, estructuras, uniones, el sistema de E/S, etc.

Es importante mencionar que para poder realizar éste trabajo se realizó una investigación precisa de cada una de las características importantes que tiene el lenguaje, se reunió material de diversas fuentes (libros, revistas, manuales, cursos) y se hicieron comparaciones de diversas implementaciones sobre todo de Microsoft C, Turbo C y Laticce, se realizaron pruebas y se hicieron conclusiones objetivas cuyo fin es el de mostrar lo más sobresaliente y reflejar las ventajas que se tienen al manejar C.

En la aplicación que se incluye se presenta el **Sistema del Ejercicio Presupuestal**, cuyo propósito principal es el de llevar acabo un control eficiente y oportuno del ejercicio presupuestal que tiene la Subdirección de Transformación Industrial de Petróleos Mexicanos; y como complemento de mismo se presenta el Módulo de Control de Información Financiera CIF.

Estos sistemas fueron desarrollados con el manejador de base de datos relacionales Informix 3.30 pero ambos, presentan rutinas hechas en C (utilizando la interface ALL-II C) que permiten una mejor manipulación de la información. La pantalla principal del

Módulo CIF esta muy ligada al código de C. En ella se permite entre otras cosas, realizar validaciones deacuerdo a archivos de catálogos, conversiones de tipo de cambio, agregar la información en otro archivos, deacuerdo a diferentes niveles de agrupación, etc..

Relational Database System, Inc. creador de Informix, realizó también una interface con el lenguaje C llamada ALL-II C la cual permite establecer un acceso a los archivos de datos propios de Informix, utilizando las funciones de biblioteca que ahí son definidas y usando como compilador Lattice ver 3.0.

CAPITULO I

INTRODUCCION

INTRODUCCION

1.1 Origenes del Lenguaje C

C es un lenguaje de programación desarrollado en los Laboratorios Bell de AT&T hacia 1972. Fué diseñado y escrito por Dennis Ritchie, quién estaba trabajando conjuntamente con Ken Thompson en el sistema operativo UNIX.

UNIX se concibió como una especie de taller lleno de herramientas para el especialista en programación y C se convirtió en la herramienta más básica de todas. Casi todas las herramientas de programación suministradas con UNIX, incluyendo el compilador C, y casi todo el sistema operativo, se escriben ahora en C.

A mediados de los años setenta UNIX se extendió fuera de los Laboratorios Bell, y se utilizó ampliamente en las universidades.

Sin ningún alboroto, C comenzó a sustituir a los lenguajes más familiares disponibles en UNIX. Nadie empujó a C. No se le hizo el lenguaje "oficial" de los Laboratorios Bell. Aparentemente autopropulsado, sin ninguna publicidad, la reputación de C se extendió y el número de sus usuarios creció.

Ante la sorpresa de su creador, los programadores empezaron a preferir C, ante lenguajes como Fortran o PL/I, o a los más modernos como Pascal y APL.

En la actualidad existen docenas de compiladores de C, muchos de ellos funcionando en sistemas no UNIX.

Está dentro del carácter de C el haber realizado un debut modesto. Pertenece a una familia bien establecida de lenguajes cuya tradición enfatiza virtudes claves como: **fiabilidad, regularidad, simplicidad y facilidad de uso.**

Los miembros de esta familia se llaman a menudo "lenguajes estructurados", puesto que están bien adecuados para la programación estructurada, una disciplina cuya finalidad es hacer que los programas sean más fáciles de leer y de escribir.

La programación estructurada se hizo algo así como una ideología en los años setentas, y otros lenguajes golpearon en la línea de flotación más estrechamente que C. El precio de la pureza se concede a menudo al Pascal, la hermana bonita de C. El lenguaje C no fué pensado para ganar premios; su objetivo fue ser amigable, capaz y fiable. Virtudes, éstas sencillas, pero que hacen que bastantes programadores que comienzan enamorados de Pascal finalicen felizmente casados con C.

En la figura 1.1 se muestran los antecesores del lenguaje C.

ANTECESORES DE C

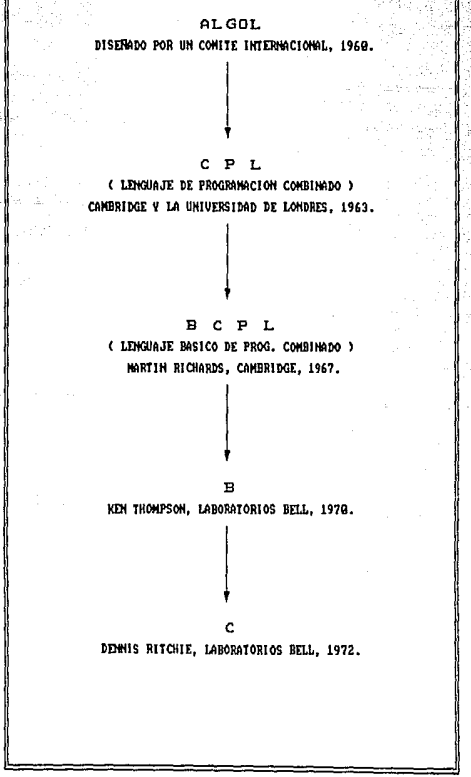


Figura 1.1

Aunque **Algol** apareció solamente unos pocos años después que Fortran, es un lenguaje mucho más sofisticado, y por esa razón ha tenido enorme influencia sobre el diseño de lenguajes de programación. Sus autores presentaron una especial atención a la regularidad de la sintaxis, estructura modular y otras características que tendemos a pensar como "modernas". Desgraciadamente, Algol realmente nunca se comprendió en los Estados Unidos, probablemente porque parecía demasiado abstracto, demasiado general.

CPL fué un intento de mejorar Algol, pero como él, era grande, con una multitud de características que le hacían difícil de aprender y difícil de realizar.

BCPL intentó resolver el problema reduciendo CPL a sus buenas características básicas.

El lenguaje B, escrito por Ken Thompson para una de las primeras versiones de UNIX, es todavía más una simplificación de CPL, aunque apropiado para utilizarse con la arquitectura de la máquina que disponía en aquel entonces Thompson.

Tanto BCPL como B eran lenguajes útiles solamente cuando se trataban con cierta clases de problemas. El logro de Ritchie con el lenguaje C, fué recuperar de alguna manera esta generalidad perdida, principalmente mediante el uso hábil de los **tipos de datos**. Consiguió hacer esto sin sacrificar la simplicidad o "contacto con la computadora" que fueron los objetivos de diseño de CPL.

1.2 C como Lenguaje de Sistemas

En el lenguaje C, se tiene la característica de que mediante unas simples y pocas reglas, se puedan poner juntas determinadas piezas de C para construir elementos más complejos, que pueden a su vez generar incluso construcciones más elaboradas.

Esta capacidad de construir programas complejos a partir de elementos simples es la principal fuerza de C. Si C tuviese un escudo de armas, su lema podría ser "un mucho a partir de un poco".

Los lenguajes escritos por una única persona suelen reflejar su campo de experiencia. El campo de Dennis Ritchie es la programación de sistemas (lenguajes de computadoras, sistemas operativos, generadores de programas, etc.) y C es de lo mejor cuando se utiliza para realizar herramientas de esta clase.

Se puede utilizar C para escribir cualquier cosa, desde programas de contabilidad hasta juegos de video. Los programas escritos en C corren rápido y necesitan menos espacio de memoria. Así pues, el dominio especial de C es la programación de sistemas. Su uso en este campo se debe a dos importantes razones.

En primer lugar, es un lenguaje relativamente de bajo nivel que permite especificar cada detalle en la lógica de un programa para conseguir un máximo rendimiento de la computadora.

En segundo, lugar es un lenguaje relativamente de alto nivel que oculta los detalles de la arquitectura de la computadora, logrando así un mayor rendimiento en la programación.

Para saber cual es lugar de C en el mundo de los lenguajes de programación nos podemos referir a la figura 1.2 que muestra la jerarquía de los lenguajes de programación:

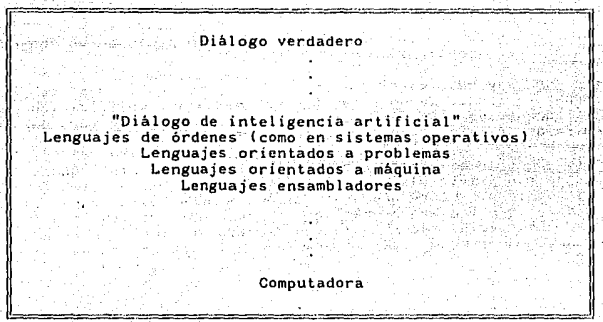


Figura 1.2

Leyéndolo desde abajo hacia arriba, éstas categorías van de lo concreto a lo abstracto, de lo orientado a la máquina a lo orientado a la persona y más o menos, de lo pasado a lo futuro.

Los puntos representan grandes saltos con muchos pasos omitidos.

Los primitivos ascendientes de la computadora, como el telar Jacquard (1805) o la "máquina analítica" de Charles Babbage (1834), se programaban a nivel máquina, y puede llegar el día que se programe una computadora teniendo simplemente una charla con ella.

El lenguaje de ensamblaje, o ensamblador como se le llama normalmente, es simplemente una representación simbólica del código máquina que realmente puede leer la computadora.

Este tipo de lenguajes nos obligaban a pesar en términos de una máquina y a especificar cada operación (como por ejemplo,

transferir estos bits dentro de este registro y sumarlos a los bits de aquel otro registro, y acontinuación colocar el resultado en memoria en tal posición), son muy tediosos de utilizar y los errores son usuales.

Los primeros lenguajes de alto nivel, como Fortran o Algol, fueron creados como alternativas a los lenguajes ensambladores. Eran mucho más generales, más abstractos, permitiendo a los programadores pensar en términos del problema que tenían, en lugar de en función de la máquina. La estructura lógica se podía imponer visiblemente sobre el programa.

Pero el inconveniente de Algol y Fortran es que son demasiado abstractos para trabajar a nivel de sistema; son lenguajes principalmente orientados al problema, la clase de lenguajes que utilizamos para resolver problemas en ingeniería, ciencia o aplicaciones comerciales.

Los programadores que necesitaban escribir programas de sistemas todavía tenían que realizarlos con el ensamblador de la máquina.

Pasado unos pocos años, algunas personas tomaron un paso hacia atrás, o en términos de nuestra jerarquía hacia abajo, y crearon la categoría de lenguajes orientados a la máquina.

El lenguaje C, está estrechamente unido a la computadora y es excelente para la programación a nivel de máquina, aunque es importante mencionar que no nada más sirva para eso. Es un lenguaje tanto de alto como de bajo nivel.

C se encuentra en un lugar muy cómodo en la jerarquía de los lenguajes de programación, justamente aquel en el que se sienten muy bien muchos programadores de sistemas. Está lo suficientemente próximo a la computadora para dar al programador un gran control sobre los detalles de la realización de su programa y lo suficientemente lejos para ignorar los detalles de la máquina.

1.3 C como Lenguaje de Nivel Medio

A menudo se denomina al lenguaje C como lenguaje de computadora de "nivel medio". Esto no significa que sea menos potente, más difícil de usar o menos evolucionado que un lenguaje de "alto nivel", ni que sea tan difícil de usar como un lenguaje de "bajo nivel" como el lenguaje ensamblador.

C se considera un **lenguaje de nivel medio** porque combina elementos de lenguajes de alto nivel, como Pascal o Modula-2, con la funcionalidad del lenguaje ensamblador.

La figura 1.3 muestra la relación (en nivel) del lenguaje C con algunos otros lenguajes.

NIVEL	LENGUAJE
ALTO	ADA FORTRAN PASCAL COBOL MODULA-2
MEDIO	C
BAJO	ENSAMBLADOR LENGUAJE MAQUINA

Figura 1.3

Teóricamente, un lenguaje de alto nivel intenta proporcionar al programador todo lo que pueda necesitar, teniéndolo ya incorporado en el lenguaje.

Un **lenguaje de nivel medio** proporciona el acceso a las instrucciones reales de la máquina, ofrece un conciso conjunto de herramientas y permite que el programador desarrolle construcciones de mayor nivel. Por tanto, un lenguaje de nivel medio ofrece al programador una gran potencia junto con una gran flexibilidad.

Viéndolo de otra forma, a los lenguajes de nivel medio a veces se les denomina lenguajes de "bloques constituyentes", debido a que el programador primero crea las rutinas que lleven acabo casi todas las funciones necesarias para el programa y luego las pone todas juntas.

C, como lenguaje de nivel medio, permite la manipulación directa de bits, bytes, direcciones y puertos que componen la computadora. Es decir, C no separa significativamente el hardware de la máquina respecto del programa. Por ejemplo, a diferencia de muchos lenguajes de alto nivel que pueden operar directamente sobre cadenas de caracteres para llevar a cabo multitud de

manipulaciones de cadenas, C trabaja con caracteres.

La mayoría de los lenguajes de alto nivel incluyen sentencias para leer y escribir en archivos de disco. En C, esos procedimientos se llevan acabo mediante llamadas a funciones que no son técnicamente parte del lenguaje C. Aunque todos los compiladores del estándar ANSI proporcionan funciones para realizar la E/S en disco, sus programas pueden pasar por alto esas funciones si así lo quiere. C ofrece una mayor flexibilidad que la mayoría de los otros lenguajes.

C tiene muy pocas sentencias que haya que recordar, sólo 32 palabras claves definidads por el ANSI, como comparación, el lenguaje Basic del IBM-PC tiene aproximadamente 159 palabras clave. Esto significa que los compiladores de C son relativamente fáciles de escribir y que, por tanto, generalmente hay uno disponible para cualquiera que sea la máquina que se use. Por esta razón, el primer compilador disponible para una nueva computadora suele ser un compilador de C.

Como C trabaja con los mismos tipos de datos que la computadora, el código que genera un compilador de C suele ser muy eficiente y rápido. De hecho, se puede usar C en lugar de ensamblador para muchas tareas.

El código de C es muy portable. La **portabilidad** significa que es posible adaptar el software escrito para un tipo de computadora en otra. Esto es importante si alguna vez se necesita llevar una aplicación a una nueva computadora que use un procesador o un sistema operativo diferente. La mayoría de los programas sólo necesitarán ser recompilados usando un compilador de C escrito para la nueva máquina. Esto ahorra una cantidad incalculable de tiempo y de dinero.

1.4 C como Lenguaje Estructurado

C es un lenguaje estructurado, que se distingue por su uso de bloques. Un bloque es un conjunto de sentencias que están relacionadas de forma lógica. Por ejemplo, una sentencia IF, si tiene éxito, ejecutará cinco sentencias individuales. Si se pueden agrupar esas sentencias y referenciarlas como una unidad indivisible, entonces se forma un bloque.

Un lenguaje estructurado permite muchas posibilidades en programación. Una de ellas es el concepto de subrutinas con variables locales, las cuales sólo son conocidas dentro de la subrutina en la que están declaradas. Además soporta varias construcciones de bucles, tales como While, do/While y for. (El uso de la sentencia goto, o bien está prohibido, o bien desaprobado, no siendo la forma normal de control, de la forma en lo que es en Basic y Fortran). Otra característica importante es que se permite sangrar las sentencias y no requiere de un estricto concepto de campos (como en versiones iniciales de Fortran).

En la figura 1.4 se muestran algunos ejemplos de lenguajes estructurados y no estructurados:

LENGUAJES NO ESTRUCTURADOS	LENGUAJE ESTRUCTURADOS
F O R T R A N	P A S C A L
B A S I C	A D A
C O B O L	C

Figura 1.4

Los lenguajes estructurados generalmente son más recientes que los no estructurados. De hecho, una característica de un lenguaje de computadora antiguo es que no sea estructurado. Hoy en día la mayoría de los programadores no sólo consideran que se programa más fácilmente en un lenguaje estructurado, sino que además el mantenimiento es mucho más fácil.

1.5 Características Principales del Lenguaje C

C es un lenguaje de programación de propósito general, por lo que cada día hay más programadores de sistemas interesados en él. Las características principales con que cuenta el lenguaje C son las siguientes:

- Programación estructurada.
- Economía en la expresiones.
- Abundancia en operadores y tipos de datos.
- Codificación en alto y bajo nivel simultáneamente.
- Reemplaza ventajosamente la programación en ensamblador.
- Utilización natural de las funciones primitivas del sistema.
- No está orientado a ninguna área en especial.
- Producción de código objeto altamente optimizado.
- Facilidad de aprendizaje.

1.6 Usos Principales del Lenguaje C

Inicialmente, C fué usado para la programación de sistemas, la cual se refiere a una clase de programas que, o bien son parte del sistema operativo de la computadora, o bién cooperan con él. Los programas del sistema son aquellos que hacen posible que la computadora realice adecuadamente el trabajo.

Algunos ejemplos de lo que se puede hacer con el lenguaje C para la programación de sistemas se muestra en la figura 1.5.

El lenguaje C se usa para la programación de sistemas por dos poderosas razones. En primer lugar, la mayoría de los programas compilados con C pueden ejecutarse casi tán rápido como aquellos escritos en ensamblador. Antiguamente, la mayoría de los programas de sistemas tenían que escribirse en lenguaje ensamblador porque los lenguajes de computadora no podían crear programas lo suficientemente rápidos. El escribir programas en ensamblador es una tarea larga, difícil y tediosa, por lo que se puede escribir código en C más rápidamente que en ensamblador, **reduciendo tremendamente los costos de desarrollo**.

En segundo lugar, C es un "lenguaje para programadores". Los programadores profesionales encuentran el lenguaje C muy atractivo por su ausencia de restricciones y su facilidad de manipulación de bits, bytes y direcciones. Además, el programador de sistemas necesita las funciones de C de control directo de la E/S y de la memoria.

En los últimos años, C también ha sido usado como lenguaje de programación de propósito general debido a la popularidad que tiene entre los programadores. Una vez que se está familiarizado con el lenguaje C, se puede seguir el flujo y la lógica de un programa y verificar el funcionamiento general de las subrutinas, todo ello en forma bastante fácil. Los listados de programas en C tienden a parecer claros a la vez que intrigantes.

Quizá la principal razón por la que C se ha convertido en un lenguaje de propósito general es porque a los programadores se les ha hecho divertido el usarlo, y sobre todo el poder generar sus propias bibliotecas de funciones.

1.7 Compiladores e Intérpretes

Los términos compilador e intérprete se refieren a la forma en que se ejecuta un programa. En teoría, cualquier lenguaje de programación puede ser compilado o interpretado. Aunque normalmente algunos lenguajes se ejecutan de una forma o de la otra, la forma en que ejecuta un programa no viene definida por el lenguaje en que haya sido escrito.

Los intérpretes y los compiladores son simplemente programas sofisticados que trabajan sobre el código fuente del programa.

USOS PRINCIPALES DEL LENGUAJE C

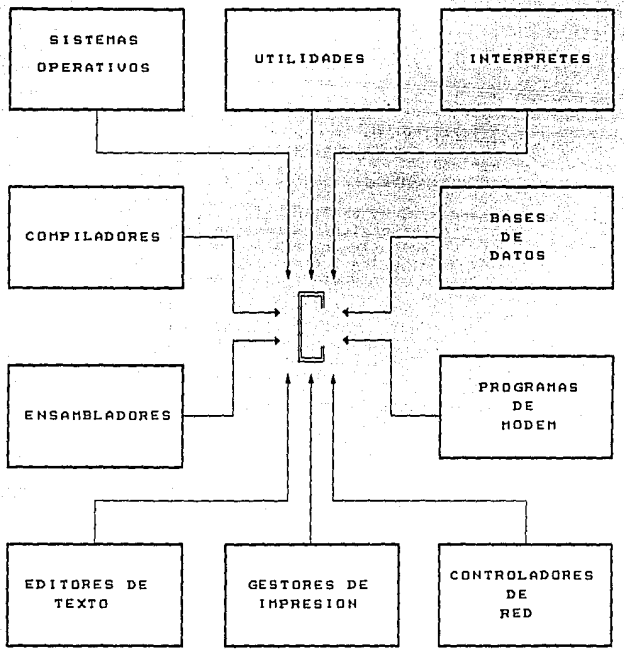


Figura 1.5

Un intérprete lee el código fuente de un programa línea a línea y realiza las instrucciones específicas contenidas en esa línea.

Un compilador lee el programa entero y lo convierte a código objeto, el cual puede ser enlazado con librerías y dar origen a un archivo ejecutable. El código objeto también se suele denominar binario o código máquina.

Una vez que el programa está compilado, las líneas de código fuente dejan de tener sentido en la ejecución del programa.

Por ejemplo, Basic normalmente es interpretado y C casi siempre se compila.

El intérprete debe estar presente cada vez que se ejecute el programa. En Basic se debe ejecutar primero el intérprete de Basic, cargar el programa y escribir el comando RUN cada vez que se quiera usar. Por contra, un compilador convierte el programa a un archivo ejecutable que puede ser directamente ejecutado por la computadora con el simple proceso de escribir su nombre.

Los programas compilados se ejecutan mucho más rápido que los interpretados. El propio proceso de compilación lleva un tiempo adicional, pero se compensa con el tiempo que se gana cuando se usa el programa.

Además de las ventajas de velocidad, los compiladores protegen el código fuente contra la manipulación y el robo. El código compilado no refleja el código fuente. Por esta razón, las casas comerciales de software usan compiladores casi exclusivamente.

Hay dos términos que se usarán con mucha frecuencia en este trabajo, son el **tiempo de compilación** y el **tiempo de ejecución.**

El tiempo de compilación se refiere al proceso de compilación y el tiempo de ejecución se refiere a la ejecución del programa.

1.8 Principales Compiladores de C

Entre los compiladores de C de uso más generalizado, de mayor éxito, y que de alguna forma se analizarán aquí, se encuentran:

- TURBO C
  Borland/Analytica International, Inc.
- MICROSOFT C 5.0
  Microsoft Corporation
- MICROSOFT QUICK C 1.0
  Microsoft Corporation
- LATTICE C 3.1
  Lattice Inc.

Al interesarnos en el lenguaje C, hemos tenido la fortuna de poder manejar estos compiladores, y en su momento encontrar algunas diferencias entre ellos. Todas éstas diferencias las mencionaremos poco a poco conforme avance este trabajo.

Durante muchos años el estándar de C fué realmente la versión proporcionada con la versión 5 del sistema operativo UNIX.

Con la popularidad de las microcomputadoras se crearon muchas implementaciones de C. Por lo que podría ser un milagro que un código fuente fuera aceptado por cada una de estas implementaciones. Para remediar esta situación, el Instituto de Estándares Americano (ANSI) estableció un comité a principios de 1983 para crear un estándar que definierá de una vez por todas al lenguaje C, dando origen al **ANSI C.**

Este trabajo esta orientado al ANSI C. Trabajar con el estándar del lenguaje C es útil, primero porque al crear código fuente se asegura que sea aceptado por las diferentes implemetaciones de C. Y segundo, al consultar los manuales de los principales compiladores, se pueden detectar las características individuales de cada compilador en determinados casos.

En la actualidad existen productos tales como Quick C para Windows ver. 2.0, Quick C Compiler ver. 2.50, y de Borland Turbo C++ ver 3.0,

1.9 Creación y Compilación de un Programa en C

La forma de crear y compilar un programa está determinada en gran parte por el compilador que se use y el sistema operativo sobre el que funcione.

Si se usa una PC, existe una amplia variedad de excelentes compiladores que contienen entornos de desarrollo de programas integrados, tales como Turbo C y Quick C.

Si se utiliza uno de esos entornos, se puede editar, compilar y ejecutar los programas desde dentro de él. (Para los principiantes esta es una opción excelente.) Simplemente hay que seguir las instrucciones proporcionadas con el compilador.

Si se usa un compilador tradicional de línea de órdenes, tal como el compilador de C de UNIX, entonces hay que seguir los siguientes pasos para crear y compilar un programa.

1.- Creación del programa usando un editor.

2.- Compilación del programa.

3.- Enlazado del programa con las bibliotecas necesarias.

4.- Ejecución del programa.

El método exacto que se use para hacer esto siempre vendrá explicado en el manual de usuario del compilador.

Un ejemplo que muestra los pasos a seguir para la realización de un programa en C, viene mostrado en la figura 1.6.

En ese ejemplo se indica que una vez editados los ficheros fuentes **A.C** y **B.C** se compilan obteniéndose los archivos objetos **A.OBJ** y **B.OBJ** los cuales son enlazados con el archivo **C.OBJ**, con la librería **D.LIB** y con las librerías del sistema **.LIB** dando lugar a un único archivo ejecutable **A.EXE**.

Para ejecutar el archivo **A.EXE** simplemente se escribe el nombre del mismo en el entorno en donde se esté trabajando.

En la mayoría de los compiladores se proporcionan al menos un programa de ejemplo, por lo que siempre es una buena idea compilar, enlazar y ejecutar el programa de muestra, de acuerdo a las intrucciones del manual del compilador.

1.10 Estructura de un Programa en C

Todo programa en C consta de una o más funciones, la principal de ellas es la función main. El programa siempre comenzará por la ejecución de la función main.

Acontinuación se presenta un sencillo programa en C que calcula el área de un círculo dando un determinado radio; el fin de ello es el de mostrar la estructura de un programa en C.

```
#include <stdio.h>

/* Programa para Calcular el Area de un Circulo */

main()
{

   float radio, area;

   printf("Radio : ?");
   scanf("%f", &radio);
   area = 3.14159 * radio *radio;
   printf("\n Area = %f",area);
}
```

Por lo general los programas en C están escritos en minúsculas y sin acentuar, excepto cuando se trata de comentarios.

Los comentarios pueden aparecer en cualquier parte del programa, mientras estén situados entre los delimitadores **/*** y ***/** (por ejemplo: /* esto es un comentario */).

REALIZACION DE UN PROGRAMA EN C

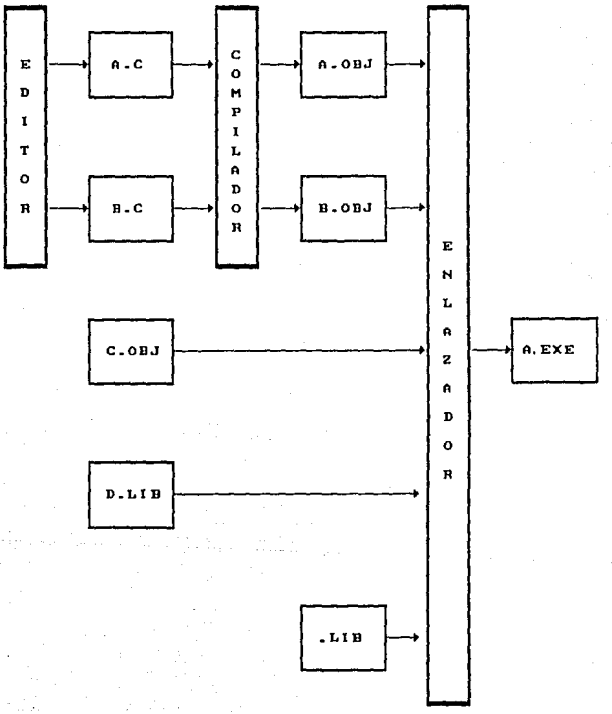


Figura 1.6

Los comentarios son útiles para identificar los elementos principales de un programa o para la explicación de la lógica de éstos.

En programas escritos en C es muy común escribir los identificadores de constantes en mayúsculas.

La primera línea de este programa contiene una referencia a un archivo especial (stdio.h, archivo de biblioteca estándar), que contiene información que se debe incluir en el programa cuando se compila.

La segunda línea es un comentario que describe el propósito del programa.

La tercera línea es la cabecera de la función **main()**. Los paréntesis vacíos que siguen al nombre de la función indican que ésta no incluye argumentos.

Las cinco líneas restantes encerradas por un par de llaves integran la sentencia compuesta dentro de **main()**. La primera línea de la sentencia compuesta es una **declaración de variables**. Aquí se establece que los nombres simbólicos de **radio** y **area** son variables en coma flotante.

Las otras cuatro líneas son un tipo particular de sentencias de expresión. La primera (**printf()**) genera una solicitud de información (el valor del radio). Este valor se introduce en la computadora al ejecutar la siguiente línea (**scanf()**). La tercer línea representa un tipo particular de sentencia de expresión llamada **sentencia de asignación**. Esta sentencia hace el cálculo del área a partir del valor del radio. Los asteriscos representan signos de multiplicación. Y la cuarta línea presenta el valor calculado del área. El valor numérico será precedido por un breve mesaje ("Area = ").

Cada sentencia de expresión dentro de la sentencia compuesta acaba con un punto y coma. Esto es necesario en toda sentencia de expresión.

Las líneas en blanco son muy útiles porque separan partes diferentes del programa en componentes lógicamente identificables.

Las identaciones indican relaciones de subordinación entre instrucciones.

Esto dos últimos aspectos no son requisitos indispensables para que pueda ejecutarse un programa, pero su presencia es fundamental en la práctica de una excelente programación, ya que puede ayudar a un programador a identificar partes en donde se presenten errores o que en un momento dado se pueda identificar una parte determinada para proceder a hacer cambios.

Un programa en C además de la función main(), puede contener otras funciones y para ello, se deben de tener en cuenta los siguientes aspectos :

1.- Una cabecera de la función, que conste del nombre de la función, seguido de una lista opcional de argumentos encerrados con paréntesis.

2.- Una lista de declaración de argumentos, si se incluyen éstos en la cabecera.

3.- Una sentencia compuesta, que contiene el resto de la función.

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa. (También se les llaman parámetros a los argumentos.)

Cada sentencia compuesta se encierra con un par de llaves, { y }. Las llaves pueden contener combinaciones de sentencias elementales (denominadas sentencias de expresión) y otras sentencias compuestas. Así las sentencias compuestas pueden estar anidadas, una dentro de otra.

Esta es una visión general de la estructura básica que caracteriza a la mayoría de los programas en C, conforme se avance en este trabajo, se hablará más ampliamente sobre los aspectos fundamentales que tiene el lenguaje C.

CAPITULO II

CONCEPTOS BASICOS DE C

NO
EXISTE
PAGINA

NO
EXISTE
PAGINA

CONCEPTOS BASICOS DE C

En este capítulo se presentan los elementos básicos para la construcción de sentencias simples de C, como por ejemplo, el conjunto de caracteres, identificadores, palabras claves de C, tipos de datos, constantes, variables, arrays, declaraciones, operadores, expresiones y sentencias. Veremos como combinar estos elementos para formar componentes más grandes de programas.

2.1 Conjunto de Caracteres de C

En la redacción de programas en C se pueden utilizar para formar los elementos básicos (como constantes, variables, operadores y expresiones) las letras mayúsculas de la **A** a la **Z**, las minúsculas de la **a** a la **z** (el compilador de C trata las letras mayúsculas y minúsculas como caracteres diferentes), los dígitos del 0 al 9 y ciertos caracteres especiales.

En la figura 2.1 se presenta el conjunto de caracteres ascii que normalmente soportan los compiladores de C.

En algunas implementaciones de C (ya que en algunas no se contempla) se permite la inclusión de los caracteres **@** y **$**, para formar parte en cadenas de caracteres y comentarios.

2.2 Secuencias de Escape

C utiliza ciertas combinaciones de caracteres, como \b, \n y \t, para representar elementos como retroceso de un espacio, nueva línea y un tabulador respectivamente. Estas combinaciones de caracteres se conocen como **secuencias de escape.**

Una secuencia de escape está formada por el carácter \ seguido de una letra o una combinación de dígitos. Son utilizadas para acciones como nueva línea, tabulaciones y para representar caracteres no imprimibles.

Una secuencia de escape siempre representa un sólo carácter, aún cuando se escriba con dos o más caracteres.

En la figura 2.2 se presenta una lista con las secuencias de escape más utilizadas:

De particular interés es la secuencia de escape \0. Representa el carácter nulo (ASCII 000), que se utiliza para indicar el final de una cadena de caracteres.

Una secuencia de escape también se puede expresar en términos de uno, dos o tres dígitos octales que representan patrones de bits correspondientes a un carácter. La forma general de una secuencia de escape tal es \ooo, donde cada **o** representa un dígito octal (de 0 al 7).

CONJUNTO DE CARACTERES ASCII

VALOR ASCII	CARACTER	VALOR ASCII	CARACTER	VALOR ASCII	CARACTER	VALOR ASCII	CARACTER
000	NUL	032	blanco	064	@	096	'
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	'	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	-	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	\	124	|
029	GS	061	=	093	]	125	}
030	RS	062	>	094	^	126	~
031	US	063	?	095	_	127	DEL

NOTA. LOS 32 PRIMEROS CARACTERES Y EL ULTIMO SON CARACTERES DE CONTROL; NO SE PUEDEN IMPRIMIR.

Figura 2.1

SECUENCIA DE ESCAPE	NOMBRE	VALOR ASCII
\a	Bell (Sonido)	007
\b	Backspace (Retroceso)	008
\t	Tabulador Horizontal	009
\v	Tabulador Vertical	011
\n	Nueva Línea	010
\f	Alimentación de Página (sólo para impresoras)	012
\r	Retorno de Carro	013
\"	Comillas	034
\'	Apóstrofe	039
\?	Signo de Interrogación	063
\\	Backslash (barra hacia atrás)	092
\0	Nulo	000
\ooo	Carácter ASCII Representación Octal	
\xhh	Carácter ASCII Representación Hexadecimal	

Figura 2.2

Algunas versiones de C, también permiten expresar una secuencia de escape en términos de uno o más dígitos hexadecimales, precedidos por la letra **x**. La forma general de una secuencia de escape en hexadecimal es \xhh, donde cada **h** representa un dígito hexadecimal (de 0 a 9 y de A a F).

2.3 Identificadores

Los identificadores son nombres que se les dá a varios elementos de un programa, como variables, constantes, funciones

y arreglos.

Un identificador está formado por letras y dígitos, en cualquier orden, excepto el primer carácter, que debe ser una letra. Se pueden utilizar mayúsculas y minúsculas (no acentuadas en la mayoría de los compiladores), aunque es costumbre utilizar minúsculas para la mayoría de los identificadores. No se pueden intercambiar mayúsculas y minúsculas, esto es, una letra mayúscula no es equivalente a la correspondiente minúscula.

El carácter de subrayado (_) se puede incluir también, y es considerado como una letra. Se suele utilizar este carácter a la mitad de los identificadores. También se puede comenzar con un carácter de subrayado un identificador, aunque en la práctica no se suele hacer.

No se limita la longitud de los identificadores. Algunas implementaciones de C reconocen sólo los ocho primeros caracteres, aunque la mayoría de ellas reconocen más. El estándar ANSI reconoce 31 caracteres.

La sintaxis para formar un identificador es:

letra/_[letra/dígito/_]...

Ejemplos:

nombre	y12	_temperatura
suma_1	x	calculo_abc

Como norma general, un identificador debe tener los suficientes caracteres como para que su significado se reconozca fácilmente, por otra parte se debe evitar un excesivo número de caracteres.

2.4 Palabras Claves

Las palabras claves son identificadores predefinidos que tienen un significado especial para el compilador C. Un identificador definido por el usuario, no puede tener el mismo nombre que una palabra clave.

El C del estándar ANSI tiene 32 palabras clave que no se pueden usar como nombres de variables o de funciones.

Las palabras clave combinadas con la sintaxis formal de C, forman el lenguaje de programación C.

En la página siguiente se presenta una lista de las palabras clave definidas por la norma ANSI de C y que son utilizadas por todos los compiladores de C.

Palabras Clave Definidas por la norma ANSI C.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Además, muchos compiladores de C tienen varias palabras clave adicionales que se usan para sacar partido de la organización de memoria de la familia de procesadores 8086/8088. Esas palabras clave tambien dan soporte a la programación con múltiples lenguajes y a las interrupciones.

Palabras claves adicionales:

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

Turbo C cuenta también con palabras clave correspondiente a los registros del IBM-PC, las cuales aparecen en letras mayúsculas precedidas por un simbolo de subrayado (por ejemplo, _AH). Todas las palabras claves anteriores están en Turbo C; Microsoft C utiliza cdecl, far, fortran, huge, near y pascal. LATTICE C se vale de las mismas palabras clave que Microsoft, con excepción de fortran.

Algunos compiladores de C pueden reconocer otras palabras clave. Siempre es recomendable consultar el manual de referencia del compilador de C que se este utilizando, para tener una lista completa de las palabras claves que podemos reconocer.

Hay que hacer notar que todas las palabras clave están en minúsculas. Ya que los caracteres de las minúsculas y los de las mayúsculas no son equivalentes, se puede utilizar una palabra clave escrita en mayúscula como un identificador. Esto no se suele hacer normalmente, y se considera propio de un estilo de programación pobre.

2.5 Tipos de Datos

Existen en C distintos tipos de datos, cada uno de los cuales se puede encontrar representado de forma diferente en la memoria de la computadora.

En el estándar ANSI C existen cinco tipos de datos básicos:

a) **char** (Carácter)

b) **int** (Entero)

c) **float** (Coma Flotante)

d) **double** (Coma Flotante de Doble Precisión)

e) **void** (Sin valor)

En la figura 2.3 se muestra el tamaño y rango de estos tipos de datos:

Tipo	Tamaño en Bits	Rango
char	8	-128 a 127
int	16	-32768 a 32767
float	32	3.4E-38 a 3.4E+38
double	64	1.7e-308 a 1.7E+308
void	0	Sin valor

Figura 2.3

El requerimiento de memoria para cada tipo de datos numérico determinará el rango permisible de valores para ese tipo de datos. Hay que señalar que los requerimientos de memoria para cada tipo de datos puede variar de un compilador de C a otro.

Las variables de tipo **char** se usan para guardar caracteres ASCII de ocho bits (1 byte) tales como "A", "B", "C" o cualquier otra cantidad de ocho bits. Para especificar un carácter, se debe encerrar entre comillas (").

Las variables tipo **int** pueden guardar cantidades que no requieran parte fraccionaria. Las variables de este tipo se usan a menudo para controlar bucles y sentencias condicionales.

Las variables de tipo **float** y **double** se utilizan cuando se requiere parte fraccionaria o cuando la aplicación utiliza números muy grandes. La diferencia entre las variables **float** y **double** está en el tamaño de los números mayor y menor que pueden contener. Una variable **double** puede guardar en C un número

aproximadamente diez veces mayor que una de tipo **float**.

El tipo **void** se utiliza para declarar funciones que no retornan un valor o para declarar un puntero a un tipo no especificado. Si **void** aparece entre paréntesis a continuación del nombre de una función, no es interpretado como un tipo. En este caso indica que la función no acepta argumentos.

A excepción del tipo void, los tipos de datos básicos pueden tener distintos **modificadores** precediéndolos.

Un **modificador** se usa para alterar el significado del tipo base para que se ajuste más precisamente a las necesidades de cada momento. Dentro de los principales modificadores de tipos se encuentran:

a) **signed**

b) **unsigned**

c) **long**

d) **short**

Los modificadores **signed**, **short**, **long**, y **unsigned** se puede aplicar a los tipos base entero y de carácter. Sin embargo, **long** también se puede aplicar a **double**.

En la figura 2.4 se muestran todas las combinaciones posibles de tipos básicos y modificadores, así como los tamaños más comunes de cada tipo y sus rangos mínimos más usuales.

Aunque se permite el uso de **signed** con enteros es redundante, ya que la declaración implícita de enteros asume números con signo.

La diferencia entre enteros con y sin signo está en cómo se interpreta el bit más significativo del entero. Si se especifica un entero con signo, el compilador de C genera código que asume que el bit más significativo va a ser usado como un indicador de signo. Si es 0, el número es positivo; si es un 1, el número es negativo. Los números negativos se representan utilizando el método de complemento a dos. En este método, se invierten todos los bits del número (excepto el indicador de signo) y luego se suma 1 al número. Finalmente, se pone el indicador de signo a 1.

Como ya sabemos, el tipo char es utilizado para almacenar un valor entero en el rango -128 a 127, correspondiente a un carácter del código ASCII. Solamente los valores 0 a 127 son equivalentes a un carácter; de forma similar el tipo **unsigned char** puede almacenar valores en el rango de 0 a 255, valores correspondientes a los números ordinales de los 256 caracteres ASCII.

Tipo	Tamaño en Bits	Rango
unsigned char	8	0 a 255
signed char	8	-128 a 127
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-128 a 127
unsigned short int	8	0 a 255
signed short int	8	-128 a 127
long int	32	-2147483648 a 2147483649
signed long int	32	-2147483648 a 2147483649
unsigned long int	32	0 a 4294967296
long double	80	3.4E-4932 a 1.1E+4932

Figura 2.4

En la figura 2.5 se muestra la forma en la que es almacenada la información en bytes, de los tipos de datos.

2.6 Tipos de Datos Avanzados

Existen otros **tipos de datos más avanzados** que en su momento los veremos con mayor detenimiento. Por el momento sólo nos limitaremos a listarlos y a definirlos:

a) enum

b) pointers (punteros)

c) struct

d) union

e) arrays

f) const

g) typedef

FORMAS DE ALMACENAJE DE LOS TIPOS DE DATOS DE C

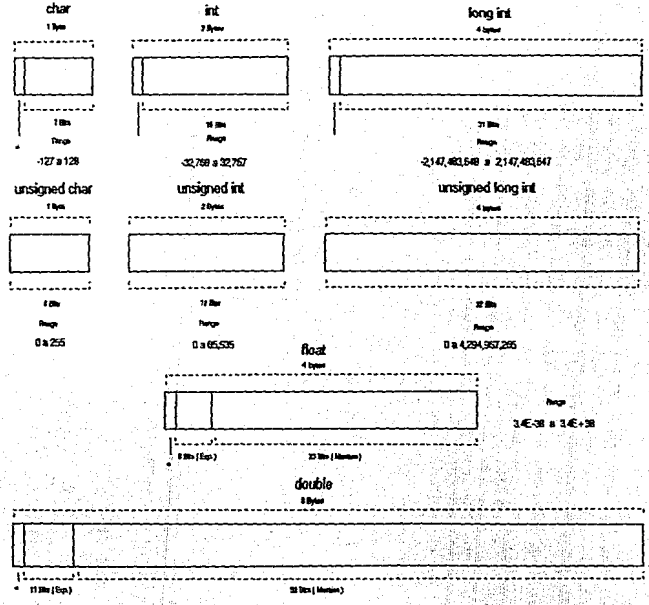


Figura 2.5

2.7 enum

Define un tipo enumerado. Un **tipo enumerado** no es más que una lista de valores, representados por identificadores, que pueden ser tomados por una variable de ese tipo.
Ejemplo:

```
enum dia_semana
        {
          lunes,
          martes,
          miercoles,
          jueves,
          viernes,
          sabado,
          domingo
        } hoy;

enum dia_semana ayer;
```

En este ejemplo se declaran las variables hoy y ayer del tipo enumerado dia_semana. Estas variables pueden tomar cualquier valor de los especificados, lunes a domingo. El valor ordinal de lunes es 0. Los elementos que aparecen enumerados en la lista son considerados como constantes enteras.

2.8 pointers (Punteros)

Un **puntero** es una dirección de memoria que indica donde se localiza un objeto de un tipo especificado. Para definir un tipo **puntero** se utiliza el modificador "*".
Ejemplos:

```
int *p;
char *plineas[40];
```

En este ejemplo se declara un puntero **p** a un valor entero y un array de punteros **plineas** a valores de tipo char.

2.9 struct

El tipo **struct** se utiliza para declarar variables que representan registros, formados por uno o más campos de igual o diferentes tipos.
Ejemplos:

```
struct
     {
       float a,b;
     } complejo
```

```
struct persona
    {
      char nombre[20];
      char apellidos[40];
      long dni;
    }
struct persona reg;
```

En estos ejemplos se declaran tanto la variable **complejo** y **reg**, como variables de estructura o registros.

La variable de estructura **complejo** comprende los campos **a** y **b** de tipo **real** y la variable de estructura **reg** de tipo **persona**, comprende los campos **nombre** y **apellidos** que son cadenas de caracteres y el campo **dni** de tipo **long**.

2.10 union

Este tipo de dato sirve para definir uniones, las cuales tienen la misma forma de definición que las estructuras. Las **uniones** a diferencia de las estructuras representan registros variables. Esto quiere decir que una variable de este tipo, puede alternar entre varios tipos.

2.11 arrays

Un **array** es un conjunto de objetos, todos del mismo tipo que ocupan posiciones sucesivas en memoria. Para definir una array se utiliza el modificador "[]".
Ejemplo:

```
int lista[40];
```

Este ejemplo declara un array **lista** de 40 elementos (lista[0] a lista[39]) para almacenar valores enteros.

2.12 const

El tipo **const** es utilizado como un modificador de un tipo fundamental. Indica que el valor de un objeto o de un puntero no puede ser modificado.

2.13 typedef

C permite definir explícitamente un nuevo nombre de tipo de datos usando la palabra clave **typedef**.

La forma general de la sentencia **typedef** es:

```
typedef tipo nombre;
```

Donde **tipo** es cualquier posible tipo de datos y **nombre** es el nuevo nombre para ese tipo.

El uso de **typedef** puede hacer que el código sea más fácil de leer y más fácil de transportar a una nueva máquina. Pero siempre hay que tener presente que no se crean nuevos tipos de datos, sino que se define un nuevo nombre para un tipo ya existente.

2.14 Constantes

Una constante es un valor que, una vez fijado por el compilador, no cambia durante la ejecución del programa.

C tiene tres tipos básicos de constantes:

a) Constantes Numéricas.
   - Enteras.
   - de Coma Flotante.

b) Constantes de Cáracter.

c) Constantes de Cadenas de Caracteres.

2.15 Constantes Numéricas

Las constantes enteras y de coma flotante representan números. Se les denomina, en general, **constantes de tipo numérico**.

Las siguientes reglas se pueden aplicar a todas las constantes numéricas:

1.- No se pueden incluir comas ni espacios en blanco en la constante.

2.- Si se desea, la constante puede ir precedida de un signo menos (-). Realmente, el signo menos es un operador que cambia el signo de una constante positiva, aunque se puede ver como parte de la constante misma.

3.- El valor de una constante no puede exceder un límite máximo y un mínimo especificados. Para cada tipo de constante, estos límites varían de un compilador de C a otro.

2.16 Constantes Enteras

Una **constante entera** es un número con un valor entero, consistente en una secuencia de dígitos. Las constantes enteras se pueden escribir en tres sistemas numéricos diferentes:

Decimal (base 10)

Octal (base 8)

Hexadecimal (base 16)

Una **constante entera decimal** puede ser cualquier combinación de dígitos tomados del conjunto de 0 a 9. Si la constante tiene dos o más dígitos, el primero de ellos debe ser distinto de 0.

Una **constante entera octal** puede estar formada por cualquier combinación de dígitos tomados del conjunto de 0 a 7. El primer dígito debe ser obligatoriamente la letra 0, con el fin de identificar la constante como un número octal.

Una **constante entera hexadecimal** debe comenzar por **0x** o **0X**. Puede aparecer después cualquier combinación de dígitos tomados del conjunto de 0 a 9 y de la letra a a la f, tanto minúsculas como mayúsculas. Las letras representan las cantidades 10 a 15 respectivamente.

El valor de una constante entera se ha de encontrar entre 0 y algún valor máximo que varía de una computadora a otra (y de un compilador a otro en una misma computadora).

Un valor máximo típico en la mayoría de las computadoras personales y muchas minicomputadoras es **32767** en decimal (equivalente a **77777** en octal o a **7fff** en hexadecimal), o lo que es igual, $2^{15}-1$.

Las grandes computadoras suelen permitir valores más grandes, tales como **2 147 483 647**, esto es, $2^{31}-1$.

Las **constantes enteras sin signo** pueden tener un valor máximo de aproximadamente el doble del máximo de las constantes enteras ordinarias, pero su valor como es natural no puede ser negativo.

Una **constante entera sin signo** se identifica añadiéndole la letra **U**, mayúscula o minúscula (**U** del inglés unsigned), al final de la constante.

Las **constantes enteras largas** pueden tomar valores máximos mayores que las constantes enteras ordinarias, pero ocupan más memoria en la computadora. En algunas computadoras (y/o algunos compiladores) se generará una constante entera larga cuando simplemente se especifique una cantidad que exceda el valor máximo. En cualquier caso, siempre es posible especificar una constante entera larga añadiendo el letra **L** (mayúscula o minúscula) al final de ésta.

Una **constante entera larga sin signo** se puede especificar añadiendo las letras **UL** al final de la constante. Las letras pueden estar en mayúsculas o minúsculas. Sin embargo, la **U** debe ir delante de la **L**.

Ejemplos de Constantes:

```
50000U      Decimal (sin signo)
123456789L  Decimal (larga)
123456789UL Decimal (larga sin signo)
0123456L    Octal (larga)
0777777U    Octal (sin signo)
0X50000U    Hexadecimal (sin signo)
0XFFFFFUL   Hexadecimal (larga sin signo)
```

Los valores máximos permitidos de las constantes enteras largas y sin signo varían de una computadora (y un compilador) a otra. En algunas computadoras, el valor máximo permitido de una constante entera larga puede ser el mismo que el de una constante entera ordinaria; otras computadoras permiten que el valor de una constante entera larga sea mucho mayor que el de una ordinaria.

2.17 Constantes de Coma Flotante

Una **constante de coma flotante** es un número de base 10 que contiene un punto decimal o un exponente (o ambos).

Si existe un exponente, su efecto es el de desplazar la posición del punto decimal a la derecha si el exponente es positivo, o a la izquierda si es negativo. Si no se incluye punto decimal en el número, se supone que se encuentra a la derecha del último dígito.

La interpretación de una constante de coma flotante con exponente es justamente la misma que en notación científica, excepto que se sustituye la base 10 por la letra **E** ó **e**.

De esta forma, el número 1.2×10^{-3} se debería escribir como 1.2E-3 ó 1.2e-3. Esto es equivalente a 0.12e-2 o 12e-4.

Los valores que pueden tener las constantes de coma flotante se encuentran dentro de un rango mucho mayor que el de las constantes enteras. Típicamente, la magnitud de una **constante de coma flotante** puede variar entre un valor mínimo de aproximadamente 3.4E-38 y un máximo de 3.4E+38. Algunas versiones del lenguaje permiten constantes de coma flotante que cubren un rango mucho mayor, como de 1.7E-308 a 1.7E+308. También es una constante de coma flotante válida el valor 0.0.

Las constantes de coma flotante se representan en C normalmente como cantidades de **doble precisión**. Por tanto, cada constante de coma flotante ocupará, típicamente, dos palabras (8 bytes) de memoria.

Algunas versiones de C permiten la especificación de constante de coma flotante de "simple precisión", añadiendo la letra F (mayúscula o minúscula) al final de la constante.

De forma análoga, algunas versiones de C permite la especificación de una constante de coma flotante "larga" añadiendo la letra L (minúscula o mayúscula) al final de la constante (por ejemplo 0.123456789E-33L).

La precisión de constantes de coma flotante (el número de cifras significativas) puede variar de una versión de C a otra. De hecho, todas las versiones del lenguaje permiten al menos seis cifras significativas, y algunas versiones proporcionan dieciocho cifras significativas.

Debe quedar claro que las constantes enteras son cantidades exactas, mientras que las constantes de coma flotante son aproximaciones. Al trabajar con C, se debe tener presente que la constante de coma flotante 1.0 puede ser representada en la memoria de la computadora como 0.99999999..., aún, cuando aparezca como 1.0 cuando se presente por pantalla, debido al redondeo automático que se efectúa. Por esta razón no se pueden utilizar los valores de coma flotante para ciertas funciones, tales como indexación o conteo (no pueden ser índices ni contadores), en las que son necesarios valores exactos.

Ejemplos de constantes de coma flotante:

0.	827.602	.3e6	2E-8
0.006E-3	1.6667E+8	300e3	19.3

2.18 Constantes de Carácter

Una **constante de carácter** es un solo carácter, con comillas simples.

Las constantes de carácter tienen valores enteros determinados por el conjunto de caracteres particular de la computadora.

La mayoría de las computadoras, y prácticamente todas las computadoras personales, utilizan el Código Estándar Americano para el Intercambio de Información (**ASCII**) como conjunto de caracteres, en el cual cada carácter individual se codifica numéricamente con su propia combinación única de 7 bits (Existen pues un total de $2^7 = 128$ caracteres diferentes.

Si consultamos la tabla que contiene el conjunto de caracteres ASCII, nos daremos cuenta que aparece junto con cada carácter, el número decimal correspondiente. Hay que hacer notar que además de codificados, los caracteres están ordenados. En particular, los dígitos están ordenados consecutivamente en su propia secuencia numérica (0 a 9), y las letras están dispuestas en orden alfabético, precediendo las mayúsculas a las minúsculas. Esto permite que las unidades de datos de tipo carácter se puedan comparar entre sí, basándose en su orden relativo dentro del conjunto de caracteres.

Los valores que tienen asignados cada uno de los caracteres serán los mismos para todas las computadoras que utilicen el conjunto de caracteres **ASCII**. Sin embargo, los valores serán diferentes en computadoras que utilicen un conjunto de caracteres distinto.

Las grandes computadoras IBM, por ejemplo, utilizan el conjunto de caracteres **EBCDIC** (Extended Binary Coded Decimal Information Code), en el cual cada carácter se codifica numéricamente en una combinación única de 8 bits. El conjunto de caracteres **EBCDIC** es diferentes en gran medida del **ASCII**.

Ejemplo de constantes de caracter:

```
'A'     'X'     '3'     '$'     ' '
```

2.19 Constantes de Cadenas de Caracteres

Una constante de cadena de caracteres consta de cualquier número de caracteres consecutivos (o ninguno) encerrados entre comillas dobles (").

A veces es necesario incluir una barra (\) o unas comillas dobles (") en una constante de cadena de caracteres. Estos caracteres se deben representar en términos de sus secuencias de escape. De igual forma, ciertos caracteres no imprimibles (por ejemplo, el tabulador y nueva línea) se pueden incluir en cadenas de caracteres si se representan en términos de sus correspondientes secuencias de escape. La mayoría de los compiladores permiten que los restantes caracteres de escape imprimibles (por ejemplo ' ?) aparezcan dentro de una cadena de caracteres tanto como un carácter ordinario, como una secuencia de escape.

Los caracteres de una cadena de caracteres son almacenados en localizaciones sucesivas de memoria.

El compilador **inserta** automáticamente un carácter nulo (\0) al final de toda constante de cadena de caracteres, como el último carácter de ésta (antes de finalizar con las comillas dobles).

Este carácter no aparece cuando se visualiza la cadena. Sin embargo, podemos examinar individualmente los caracteres de una cadena de forma fácil y comprobar si cada uno de ellos es o no es un carácter nulo. De esta forma se puede indentificar rápidamente el final de una cadena. Esto es una gran ayuda si la cadena es examinada carácter a carácter, como se requiere en muchas aplicaciones. Además en muchas situaciones, esta designación del final de cadena elimina la necesidad de especificar una longitud máxima de las cadenas de caracteres.

Hay que recordar que una constante de carácter (por ejemplo 'A') y su correspondiente constante de cadena de caracteres de

uno sólo ("A") no son equivalentes. Una constante de carácter tiene un valor entero correspondiente, mientras que una constante de cadena de un solo carácter no tiene un valor entero equivalente y de hecho constan de dos caracteres (el carácter especificado seguido de un carácter nulo, \0).

En el siguiente ejemplo, la constante de cadena de caracteres incluye tres caracteres especiales representados por sus correspondientes secuencias de escape:

```
"\tPara continuar, apretar la tecla \"RETURN\"\n"
```

Los caracteres especiales son \t (tabulador horizontal), \" (comillas que aparecen dos veces) y \n (nueva línea).

2.20 Variables

Una variable es un identificador que se utiliza para representar cierto tipo de información dentro de una determinada parte del programa.

En su forma más sencilla, una variable contendrá un dato simple, esto es, una cantidad numérica o una constante de carácter. En algún punto del programa se le asignará a la variable un determinado valor. Este, se puede recuperar después en el programa simplemente haciendo referencia al nombre de la variable.

A una variable se le pueden asignar diferentes valores en distintas partes del programa, de esta forma la información representada por la variable puede cambiar durante la ejecución del programa. Sin embargo, el tipo de datos asociado a la variable no puede cambiar.

2.21 Declaración de las Variables

Una declaración asocia **clase** y **tipo de datos** determinado a un grupo de variables. Se deben declarar todas las variables antes de que aparezcan en sentencias ejecutables.

La sintaxis correspondiente a la declaración de una variable es la siguiente:

```
[clase] tipo lista_de_variables;
```

donde la **clase** de almacenamiento de una variable, puede estar representada por: **auto**, **register**, **static**, o **extern**.

Además la clase determina si una variable tiene carácter **global** (**static** o **extern**) o **local** (**auto** o **register**).

Las diferentes clases de almacenamiento se verán con mayor detalle más adelante.

tipo determina el tipo de la variable (**char**, **int**, **float**, **double**, etc.).

Ejemplos de declaración de variables:

```
int   i, j, k;
char  c, car;
float f, balance;
double d;
```

En C el nombre de una variable no tiene nada que ver con su tipo.

El estándar **ANSI** garantiza que al menos los primeros 31 caracteres de un nombre de identificador (incluyendo los nombres de las variables) han de ser significativos.

2.22 Ambito de las Variables

Se denomina ámbito (o scope) a la parte de un programa donde una variable puede ser referenciada por su nombre. Una variable puede ser limitada a un archivo, a una función, a un bloque o a una declaración prototipo.

Una variable declarada fuera de todo bloque (conjunto de sentencias encerradas entre llaves {}) es, por defecto, **variable global** y es accesible en el resto del archivo fuente en el que está declarada. Por el contrario, una variable declarada dentro de algún bloque, es por defecto **variable local** y es accesible solamente dentro de éste. En la figura 2.6 se muestra el ámbito de una **variable global** y el de una **variable local**.

2.23 Clases de Almacenamiento

Las variables no solamente tiene un tipo de dato sino también una clase de almacenamiento.

La **clase de almacenamiento** se refiere a la permanencia de la variable y a su ámbito, que como ya sabemos, es la porción de un programa en el cual se reconoce la variable.

Desde el punto de vista de un compilador de C, un nombre de una variable identifica alguna posición física dentro de la computadora donde la cadena de bits que representan el valor de la variable se almacenan.

Existen básicamente dos clases de lugares en una computadora en donde se pueden almacenar tales valores: los **registros de memoria** o los **registros de la CPU**.

AMBITO DE VARIABLES

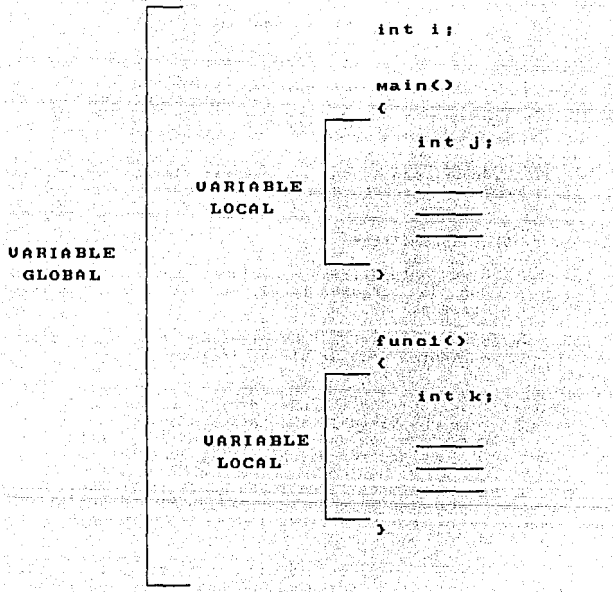


Figura 2.6

Es la clase de almacenamiento de la variable lo que determina si se va a almacenar en memoria o en un registro. Aún más, una clase de almacenamiento de variable también dice cuándo está activa, esto es, especifica el alcance de la variable.

Existen cuatro clases de almacenamiento en C:

a) Automático

b) Externo

c) Estático

d) Registro

Y están identificadas por las palabras claves: **auto**, **extern**, **static** y **register** respectivamente.

2.24 Variables Automáticas

Las variables automáticas se declaran siempre dentro de una función y son de caracter **local**; es decir, su ámbito está confinado a esa función.

Las variables automáticas definidas en funciones diferentes serán independientes unas de otras, incluso si tienen el mismo nombre.

Cualquier variable declarada dentro de una función se interpreta como una variable automática a menos que se incluya dentro de la declaración una clase distinta de almacenamiento. Esto incluye la declaración de argumentos formales.

Como la localización de la declaración de la variable dentro de una función determina la clase de almacenamiento automático, no se necesita la palabra clave **auto** al principio de cada declaración de variable.

Se pueden asignar valores iniciales a las variables automáticas incluyendo expresiones adecuadas dentro de la declaración de variables, o por asignación explícita de expresiones en cualquier parte de la función. Tales valores se reasignarán cada vez que se entre en la función. Si una variable automática no es inicializada, de alguna manera su valor inicial será impredecible y probablemente incomprensible.

Una variable automática no mantiene su valor cuando el control es transferido fuera de la función en que está definida. Por lo tanto, cualquier valor que fuera asignado a una variable automática dentro de una función, se perderá una vez que se salga de ella. Si la lógica de un programa requiere que un valor particular sea asignado a una variable automática cada vez que se

ejecuta la función, ese valor tendrá que reasignarse cada vez que se entre en la función (siempre que la función sea accedida).

La declaración de variables automáticas puede parecerse a éstas:

```
{
auto int a;
auto int b = 12345;
auto char c;
auto float d = 123.45;
.
.
.
}
```

Es importante remarcar que el alcance o ámbito de una variable automática se limita al bloque en el cual aparece. La variable existe dentro de ese bloque y en cualquier otro que se encuentre dentro de el. Tan pronto como el programa deja ese bloque, la variable deja de existir.

Cuando una variable existe, ocupa memoria dentro de la computadora, las variables automáticas tienen una cualidad obvia: **no gastan sitio cuando no se necesitan realmente.**

2.25 Variables Externas

Las **variables externas**, en contraste con las variables automáticas, no están confinadas a funciones simples. Su ámbito se extiende desde el punto de definición hasta el resto del programa. Por tanto, se expanden normalmente, por dos o más funciones y frecuentemente por todo el programa.

Como las variables externas se reconocen globalmente, pueden accederse desde cualquier función que caiga dentro de su ámbito. Y además, mantienen los valores que fueran asignados dentro del mismo. Por tanto una variable externa se le puede asignar un valor dentro de una función, y este valor puede usarse (accediendo a la variable externa) dentro de otra función.

El uso de variables externas proporciona un mecanismo adecuado de transferencia de información entre funciones. En particular, podemos transferir información a una función sin usar argumentos. Esto es esencialmente conveniente cuando una función requiere numerosos datos de entrada.

Al trabajar con variables externas, hay que distinguir entre lo que son **definiciones** y **declaraciones** de variables externas.

Una **definición de variable externa** se escribe de la misma manera que una declaración de una variable ordinaria. Tiene que aparecer fuera, y normalmente antes, de las funciones que acceden variables externas.

En la **definición** automáticamente se reserva el espacio de almacenamiento requerido (en la memoria de la computadora) para las variables externas. La asignación de valores iniciales puede incluirse, si se desea, dentro de una definición.

No se requiere el especificador de tipo **extern** en una definición de variable externa, ya que éstas se identifican por la localización de su definición dentro del programa. De hecho, muchos compiladores de C prohiben la aparición del especificador de tipo o clase de almacenamiento **extern** dentro de una definición de variable externa.

Si una función requiere una variable que ha sido definida antes en el programa, entonces la función puede acceder libremente a la variable externa, sin ninguna declaración especial dentro de la función.

Por otra parte, si la definición de función precede a la definición de variable externa, entonces la función tiene que incluir una **declaración** para la variable externa. Las definiciones de funciones en un programa grande frecuentemente incluyen declaraciones de variables externas.

Una **declaración de variable externa** tiene que empezar con el especificador de la clase de almacenamiento **extern**, el nombre de la variable y su tipo, que además tendrán que coincidir con la correspondiente definición de variable externa.

El espacio de almacenamiento para variables externas no se reservará como consecuencia de una declaración de variable externa. Es más, una declaración de este tipo no puede incluir una asignación de valores iniciales. Estas son las diferencias importantes entre lo que es una declaración y una definición de variable externa. En la figura 2.7 se muestra un ejemplo de ello.

A medida de que el archivo crece, el tiempo de compilación se hace tan largo que puede ser desesperante. Cuando ocurre esto, se debe partir el programa en dos o más archivos separados. De esta forma, al hacer pequeños cambios en un archivo no es necesario recompilar todo el programa (lo que ahorra un tiempo sustancial en grandes proyectos). C contiene la palabra clave **extern** para ayudar en este enfoque multiarchivo.

Cuando el programa consiste en dos o más archivos, debe haber alguna forma de hacer conocer a todos ellos las variables globales requeridas por el programa. En el programa sólo puede haber una copia de cada variable global. Si se declaran dos variables globales con el mismo nombre en el mismo archivo, el compilador de C imprimirá un mensaje de error o eligirá arbitrariamente una de ellas. Análogamente, aparecerán problemas si se declaran las variables globales necesitadas por el programa en cada archivo de un programa multi-archivo. Aunque el compilador no daría ningún mensaje de error en tiempo de compilación, realmente se intenta crear dos o más copias de cada variable. El problema aparecerá cuando se intente enlazar los

DIFERENCIA ENTRE DEFINICION Y DECLARACION DE UNA VARIABLE EXTERNA

DEFINICION

```
DEFINICION DE VARIABLE
      EXTERNA

func( )
{


}
```

DECLARACION

```
func( )
{
DECLARACION DE VARIABLE
      EXTERNA

 extern int x;

}
DEFINICION DE VARIABLE
      EXTERNA
```

FIGURA 2.7

módulos. El enlazador mostrará el mensaje de error porque no sabrá que variable usar.

La solución está en declarar todas las variables globales en un archivo y usar declaraciones **extern** en los otros, tal y como lo muestra la figura 2.8.

En el archivo dos la lista de las variables globales se copió del archivo uno añadiendo a las declaraciones el especificador **extern**. El especificador **extern** indica al compilador que los **nombres y tipos de variables que siguen ya han sido declarados en alguna otra parte**. En otras palabras, **extern** permite que el compilador conozca los tipos y nombres de esas variables globales sin crear realmente de nuevo almacenamiento para ellas. Cuando se enlazan los dos módulos, se resuelven todas las referencias a variables externas.

Cuando se usa una variable global dentro de una función que está en el mismo archivo que la declaración de la variable global, se puede usar **extern** (aunque raramente se hace). El siguiente fragmento de programa muestra cómo se usa esta opción:

```
int primero, ultimo;          /* Definición global de las
                                 variables primero y último */
main()
{
  extern int primero;         /* Uso opcional de la declaración
                                 extern */

}
```

Aunque las declaraciones de variables **extern** se pueden dar dentro del mismo archivo que contiene la declaración global, no son necesarias. Si el compilador de C encuentra una variable que no ha sido declarada, el compilador mira si cuadra con alguna de las variables globales. Si es así, el compilador asume que es la variable referenciada.

2.26 Variables Estáticas

Una declaración que incluye variables estáticas puede parecerse a ésta:

```
{
static int a;
static int b=1;
      .
      .
      .
}
```

Al igual que las variables automáticas, las variables estáticas son locales a la función en la que se declaran. La diferencia es

USO DE VARIABLES GLOBALES EN MODULOS COMPILADOS POR SEPARADO

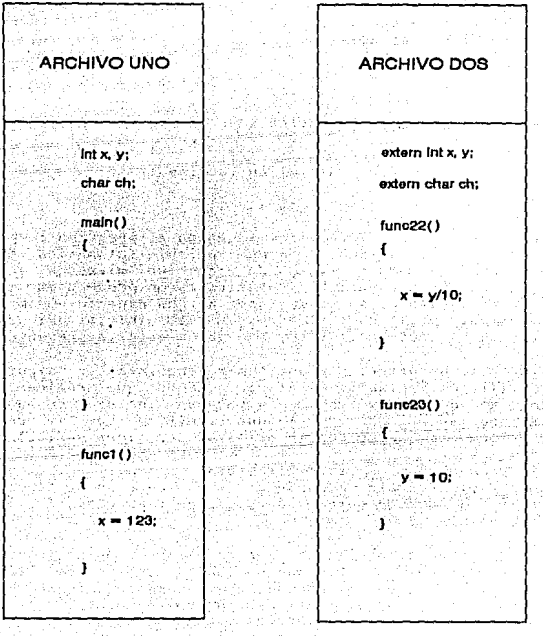


FIGURA 2.8

que las variables estáticas no desaparecen cuando la función no está activa. Sus valores persisten; si el programa regresa a la misma función otra vez encontrará que las variables estáticas tiene los mismos valores que tenían la última vez que la dejaron. Así pues, las variables estáticas dan memoria a una función. Pueden contar el número de veces que ocurre un suceso, llenar elementos consecutivos de un array, etc..

Cuando se especifica un valor inicial a una variable automática, ese valor se asigna a esa variable cada vez que el bloque en el cual la variable está activa se ejecuta. Esto no es verdad para las variables estáticas. Por ejemplo, el valor inicial de 1 en la línea:

```
static int x = 1;
```

se asigna a la variable estática **x** en el momento en el que se compila el programa.

Por otro lado, el valor inicial en la línea:

```
auto int x = 1;
```

se asigna a la variable automática **x** cuando el programa compilado se ejecuta. La inicialización de una variable automática es lo que se llama **fenómeno en tiempo de ejecución**, mientras que la inicialización de una variable estática es un proceso en **tiempo de compilación**; esto es, el compilador asigna espacio en memoria para la variable estática y almacena su valor inicial ahí. Cuando el programa se ejecuta realmente, ese valor está ya presente.

Por todo lo anterior, hay que tener encuenta lo siguiente: evitar utilizar variables estáticas a menos que realmente se necesite, ya que sus valores se almacenan incluso cuando las variables no están activas, lo que significa que se consume espacio en memoria que se podría utilizar por otras variables.

Si el programador no inicializa una variable estática, el compilador fija su valor a cero (si la variable es de tipo int, su valor inicial por defecto es 0; si es de tipo char, '\0'; y si es float o double, 0.0).

Como ya lo hemos mencionado, dentro de su propia función o archivo, las variables **static** son variables permanentes. Difieren de las variables globales en que no son conocidas fuera de su función o archivo, aunque mantienen sus valores entre llamadas. Esta posibilidad las hace muy útiles cuando se escriben funciones generalizadas y biblioteca de funciones, para que sean usadas por otros programadores. Las variables **static** tienen efectos diferentes siendo locales o siendo globales, por lo que se examinan por separado.

2.27 Variables Locales Static

Cuando se aplica el modificador **static** a una variable local, el compilador crea un almacenamiento permanente para ella de forma muy parecida a cuando crea almacenamiento para una variable global. La diferencia clave entre una variable local **static** y una variable global es que la primera sólo es conocida en el bloque en el que está declarada. Sencillamente, una variable local **static** es una variable local que retiene su valor entre llamadas de funciones.

2.28 Variables Globales Static

Cuando se aplica el modificador **static** a una variable global, se indica al compilador que cree una variable global conocida únicamente en el archivo en el que se declara la variable global **static**. Esto significa que, aunque la variable es global, las rutinas de otros archivos no la reconocerán ni alterarán su contenido directamente. Consecuentemente, estará libre de efectos secundarios.

En resumen los nombres de las variables locales **static** sólo se conocen en la función o el bloque de código en el que están declaradas y los nombres de las variables globales **static** sólo se conocen en el archivo en el que residen.

Las variables de tipo **static** permiten al programador ocultar partes de un programa a otras partes del mismo. Esto constituye una tremenda ventaja cuando se intenta gestionar un programa grande y complejo.

2.29 Variables Registro

Los registros son áreas especiales de almacenamiento dentro de la unidad central de proceso. Las operaciones aritméticas y lógicas reales se realizan dentro de estos registros. Normalmente estas operaciones se realizan transfiriendo información desde la memoria de la computadora hasta esos registros, realizando las operaciones indicadas, y transfiriendo de nuevo los resultados a la memoria de la computadora. Este procedimiento general se repite muchas veces durante el curso de la ejecución de un programa.

Para algunos programas el tiempo de ejecución se puede reducir considerablemente si ciertos valores pueden almacenarse dentro de los registros en vez de en la memoria de la computadora. Además tales programas pueden ser algo menores en tamaño (pueden requerir menos instrucciones), ya que se necesitan menos transferencias. Sin embargo, normalmente no se reducirá el tamaño espectacularmente y será menos significativo que el ahorro en tiempo de ejecución.

En C los valores de las variables **registro** se almacenan dentro de los registros de la unidad central de proceso. A una variable se le puede asignar esta clase de almacenamiento simplemente precediendo la declaración de tipo con la palabra **register**. Pero sólo puede haber unas pocas variables registro (típicamente, dos o tres) dentro de cualquier función. El número exacto depende de la computadora particular y del compilador específico. Normalmente sólo las variables enteras tiene asignado el tipo o clase de almacenamiento **register**.

Las clases de almacenamiento **automatic** y **register** están muy relacionadas. En particular su ámbito es el mismo. Por tanto, las variables registro son locales a la función en la que han sido declaradas. Además, las reglas que gobiernan el uso de las variables **register** son las mismas que las de las variables automatic, excepto que el operador dirección (&) no se puede aplicar a las variables registro.

Estas semejanzas entre variables **register** y **automatic** no son casuales, porque el tipo de almacenamiento **register** sólo puede asignarse a variables que de otro modo tendrían el tipo de almacenamiento **automatic**. Además, el declarar ciertas variables como **register** no garantiza que sean realmente tratadas como variables de tipo registro. La declaración será válida sólo si el espacio requerido de registro está disponible. Si una declaración **register** no se puede tener en cuenta, las variables se tratarán como si tuvieran el tipo de almacenamiento automatic.

Aunque la clase de almacenamiento **register** se asocia normalmente con variables de tipo entero, algunos compiladores permiten que sea asociado con otros tipos de variables que tengan el mismo tamaño de palabra (por ejemplo **short** o **unsigned**). Además se pueden permitir punteros a tales variables.

La especificación de la clase de almacenamiento **register** puede incluirse como una parte de la declaración formal de argumento dentro de una función, o como una parte de la especificación de tipo de argumento dentro de un prototipo de función.

2.30 Operadores

C es un lenguaje muy rico en operadores. Un **operador** es un símbolo que indica al compilador que lleve a cabo manipulaciones matemáticas o lógicas específicas.

Los operadores se pueden clasificar en los siguientes grupos:

a) Operadores Aritméticos

b) Operadores Relacionales, de Igualdad y Lógicos

c) Operadores de Asignación

d) Operadores a Nivel Bits

e) Operador Ternario

f) Operador Coma

g) Operador Sizeof

h) Operadores de Puntero

i) Operador Punto y Operador Flecha

j) Los Paréntesis y los Corchetes como Operadores

Los datos sobre los que actúan los operadores se denominan operandos. Algunos operadores requieren dos operandos, mientras que otros sólo actúan sobre uno (Operador Monario).

La mayoría de los operadores permiten que los operandos puedan ser expresiones. Mientras que unos pocos sólo permiten variables como operandos.

2.31 Conversión de Tipo

Los operandos que difieren en el tipo pueden sufrir una **conversión de tipo** antes de que la expresión alcance su valor final. En general, el resultado siempre se expresará con la **mayor precisión posible**, de forma consistente con los tipos de datos de los operandos.

Se pueden aplicar las siguientes reglas sencillas cuando ninguno de los operandos es **unsigned**.

1.- Si los dos operandos son tipos de coma flotante con precisión distinta (por ejemplo un float y un double), el operando de menor precisión se transformará al de mayor precisión y así se mostrará el resultado. Por tanto, una operación entre un float y un double dará como resultado un double; un float y un long double dará lugar a un long double, y un double con un long double producirán un long double.

2.- Si un operando es de tipo de coma flotante (por ejemplo float, double o long double) y el otro es un char o un int (incluyendo short int y long int), el char/int se convertirá al tipo de coma flotante del otro operando, y el resultado se expresará de igual forma. Por tanto, una operación entre un int y un double tendrá como resultado un double.

3.- Si ninguno de los dos operandos es del tipo de coma flotante, pero uno es un long int, el otro se transformará en long int y el resultado será long int.

NO

EXISTE

PAGINA

GRUPOS DE PRECEDENCIA Y ASOCIATIVIDAD DE OPERADORES

FUNCION ARRAY MIEMBRO ESTRUCTURA PUNTERO A MIEMBRO DE ESTRUCTURA	() [] . ->	I -> D
OPERADORES MONARIOS	- ++ -- ! ~ * & sizeof	D -> I
MULTIPLICACION DIVISION MODULO	* / %	I -> D
SUMA RESTA	+ -	I -> D
OPERADORES DESPLAZAMIENTO	<< >>	I -> D
OPERADORES RELACIONALES	< <= > >=	I -> D
OPERADORES IGUALDAD	== !=	I -> D
AND A NIVEL DE BITS	&	I -> D
OR EXCLUSIVA A NIVEL DE BITS	^	I -> D
OR A NIVEL DE BITS	|	I -> D
AND LOGICO	&&	I -> D
OR LOGICO	||	I -> D
OPERADOR CONDICIONAL	?	D -> I
OPERADORES ASIGNACION	= += -= *= /= %= &= ^= |= <<= >>=	D -> I
OPERADOR COMA	,	I -> D

Figura 2.9

Operador	Acción
+	Suma o Adición. Los operandos pueden ser enteros o reales.
-	Resta o Sustracción. Los operandos pueden ser enteros o reales.
*	Multiplicación. Los operandos pueden ser enteros o reales.
/	División. Los operandos pueden ser enteros o reales, si ambos operadores son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	Módulo o resto de una división entera Los operandos tiene que ser enteros.

Figura 2.10

Si uno o los dos operandos representan valores negativos, las operaciones de adición, sustracción, multiplicación y división tendrán como resultado valores cuyos signos están determinados por las reglas del álgebra.

La interpretación de la operación de resto no está clara cuando uno de los operandos es negativo. La mayoría de las versiones de C asignan al resto el mismo signo del primer operando. Por lo general hay que tener mucho cuidado con el uso de la operación de resto cuando uno de los operandos es negativo ya que se puede tener resultados no esperados.

Los operadores aritméticos, *, / y % se encuentran dentro de un mismo grupo de precedencia, y + y - se encuentran en otro. El primero tienen mayor precedencia que el segundo. Por lo que las operaciones de multiplicación, división y resto se realizarán antes que las de adición y sustracción.

Dentro de estos grupos de precedencia, la asociatividad es de izquierda a derecha. En decir, operaciones consecutivas de adición y sustración se realizarán de izquierda a derecha, así como operaciones consecutivas de multiplicación, división y resto.

Es importante mencionar que la precedencia natural de las operaciones se puede modificar mediante el uso de paréntesis, éstos permiten que se puedan realizar las operaciones de una expresión en el orden en que se desee. De hecho se pueden anidar los paréntesis, es decir, un par dentro de otro. En estos casos se realizan primero las operaciones más internas. A veces es muy

buena idea utilizar paréntesis para hacer más clara una expresión aunque no sean necesarios.

2.34 Operadores Relacionales, de Igualdad y Lógicos

En C existen cuatro operadores relacionales que se muestran en la figura 2.11

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

Figura 2.11

Estos operadores se encuentran dentro del mismo grupo de precedencia, que es menor que la de los operadores aritméticos. La asociatividad de estos operadores es de izquierda a derecha.

Muy asociados a los operadores relacionales, existen los operadores de igualdad quer se muestran en la figura 2.12 .

Operador	Significado
==	Igual que
!=	No igual que (Diferente)

Figura 2.12

Los operadores de igualdad se encuentran en otro grupo de precedencia, por debajo de los operadores relacionales. La asociatividad de estos operadores es también de izquierda a derecha.

Los operadores de igualdad y relacionales se utilizan para formar expresiones lógicas que representan condiciones que pueden ser ciertas o falsas. La expresión resultante será de tipo

entero, ya que cierto es representado por el valor entero 1 y falso por el valor 0.

Además de los operadores relacionales y de igualdad, existe otro grupo de operadores muy relacionados con éstos y son los operadores lógicos que se muestran en la figura 2.13 .

Operador	Significado
&&	**AND**. Da como resultado el valor lógico 1 si ambos operandos son distintos de cero. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado.
||	**OR**. El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de cero, el resultado es 1. Si el primer operando es distinto de cero, el segundo operando no es evaluado.
!	**NOT**. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario.

Figura 2.13

Los operadores lógicos actúan sobre operandos que son a su vez expresiones lógicas. Permiten combinar expresiones lógicas individuales, formando otras condiciones lógicas más complicadas que pueden ser ciertas o falsas.

Los operandos de los operadores puede ser de tipo entero, real o puntero. Los tipos del operando pueden ser diferentes. El resultado es de tipo **int**.

Cada uno de los operadores lógicos pertenece a su propio grupo de precedencia. El operador **AND** tiene mayor precedencia que el operador **OR**. Los dos grupos de precedencia se encuentran por debajo del grupo que contiene los operadores de igualdad. La asociatividad es de izquierda a derecha.

El operador monario (**!**) niega el valor de una expresión lógica, es decir, hace que una expresión que era originalmente cierta se haga falsa y viceversa. Este operador se denomina operador de **negación lógica**.

Las expresiones lógicas compuestas que constan de expresiones lógicas individuales unidas por los operadores lógicos (**&&**) y

(||) se evalúan de izquierda a derecha, pero sólo hasta que se ha establecido el valor cierto/falso del conjunto.

2.35 Operadores de Asignación

Existen varios operadores de asignación en C. Todos se utilizan para formar expresiones de asignación, en las que se asigna el valor de una expresión a un identificador.

El operador de asignación más usado es el operador (=). Las expresiones de asignación que utilizan este operador se escriben de la siguiente forma.

identificador = expresión

en donde **identificador** representa generalmente una variable y **expresión** una constante, una variable, o una expresión más compleja.

Es muy importante remarcar que el operador de asignación (=) y el operador de igualdad (==) son muy distintos. El operador de asignación se utiliza para asignar un valor a un identificador, mientras que el operador de igualdad se usa para determinar si dos expresiones tienen el mismo valor.

Las expresiones de asignación se suelen llamar **sentencias de asignación**, ya que se suelen escribir como sentencias completas. Sin embargo, también se pueden escribir expresiones de asignación como expresiones que estén incluidas dentro de otras sentencias.

Si los dos operandos de una sentencia de asignación son de diferente tipo de datos, el valor de la expresión a la derecha se convertirá automáticamente al tipo de identificador de la izquierda. De esta forma, toda la expresión de asignación será del mismo tipo de datos.

En determinados casos, esta conversión de tipo automática puede conllevar una alteración del dato que se está asignando. Por ejemplo:

- Un valor de coma flotante puede ser truncado si se asigna a un identificador entero.

- Un valor de doble precisión puede ser redondeado si se asigna a un identificador de coma flotante (de simple precisión).

- Una cantidad entera puede ser alterada si es asignada a un identificador de un entero más corto o a un identificador de carácter (se pueden perder algunos de los bits más significativos). Además, cuando se asigne a un identificador de tipo numérico el valor de una constante de carácter, el efecto que pudiera resultar dependerá del conjunto de caracteres que se esté utilizando. De esto pueden resultar

inconsistencias entre distintas versiones de C.

En C están permitidas asignaciones múltiples de la siguiente forma:

```
identificador1 = identificador2 = ... = expresión
```

En estos casos, las asignaciones se afectúan de izquierda y derecha. Por tanto, la asignación múltiple:

```
identificador1 = identificador2 = expresión
```

es equivalente a:

```
identificador1 = (identificador2 = expresión)
```

C posee además, los siguientes cinco operadores de asignación:

```
+=          -=
*=          /=
          %=
```

Que de una forma o de otra realizan una función semejante. Para ver cómo se utilizan, consideremos el primer operando, +=. La expresión de asignación:

```
expresión1 += expresión2
```

es equivalente a

```
expresión1 = expresión1 + expresión2
```

de forma análoga, la expresión de asignación

```
expresión1 -= expresión2
```

es equivalente a

```
expresión1 = expresión1 - expresión2
```

y esto es de igual forma para los restantes tres operadores.

Normalmente, **expresión1** es un identificador tal como una variable o un elemento de un array.

Los operadores de asignación tienen menor precedencia que los operadores aritméticos, relacionales, de igualdad y lógicos. Las operaciones de asignación tienen asociatividad de derecha a izquierda.

2.36 Operaciones a Nivel Bits

Algunas aplicaciones requieren la manipulación de los bits individuales de un byte o de una palabra de memoria.

Las operaciones sobre bits son frecuentes en aplicaciones de control de dispositivos, programas de modem, rutinas de archivos de disco, rutinas de impresora, etc., debido a 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 han cambiado. Normalmente es el bit más significativo de cada byte.)

Usualmente se requiere el lenguaje ensamblador o máquina para este tipo de operaciones. Sin embargo, C contiene varios operadores especiales que realizan fácil y eficientemente estas operaciones a nivel de bits.

Los operadores a nivel de bits se pueden dividir en cuatro categorías generales:

a) El operador de complemento a uno.

b) Los operadores lógicos a nivel de bits.

c) Los operadores de desplazamiento.

d) Los operadores de Asignación a nivel de bits.

2.37 El Operador de Complemento a Uno

El operador de complemento a uno (~) es un operador monario que invierte los bits de su operando, por tanto los unos se convierten en ceros y los ceros en unos. Este operador precede siempre a su operando. El operando tiene que ser un entero (incluido integer long, short, unsigned o char). Generalmente el operando será un octal o una cantidad hexadecimal sin signo, pero no es obligatorio.

El operador de complemento a uno se refiere como operador de complementación. Es un miembro del mismo grupo de precedencia que los otros operadores monarios; por tanto, su asociatividad es de derecha a izquierda.

2.38 Operadores Lógicos a Nivel de Bits

En la figura 2.14 se presentan los operadores lógicos a nivel de bits:

Cada uno de estos operadores requiere dos operandos enteros. Las operaciones se realizan de forma independiente en cada par de

Operador	Acción
&	Operación **AND** a nivel de bits. (Y)
|	Operación **OR** a nivel de bits. (O)
^	Operación **XOR** a nivel de bits. (O exclusiva)

Figura 2.14

bits que corresponden a cada operando. Así se compararán los bits menos significativos (los bits más a la derecha) de las dos palabras, después los siguientes bits menos significativos, y así sucesivamente, hasta que se comparen todos los bits. Los resultados de estas comparaciones son:

- Una expresión Y (AND) a nivel de bits retornará un 1 si ambos bits tienen el valor 1 y 0 en caso contrario.
- Una expresión O exclusiva (XOR) a nivel de bits retornará un 1 si uno de los bits tiene un valor de 1 y el otro tiene un valor 0 (un bit es cierto, el otro falso). En otro caso retornará un valor de 0.
- Una expresión O (OR) a nivel de bits retornará un 1 si uno o más de los bits tiene el valor de 1 (uno o ambos bits son ciertos). En otro caso retornará un valor de 0.

Estos resultados están resumidos en la figura 2.15 .

Operandos		Operador AND	Operador XOR	Operador OR
b1	b2	b1 & b2	b1 ^ b2	b1 | b2
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Figura 2.15

Cada uno de los operadores lógicos a nivel de bits tienen su propia precedencia. El operador Y (AND) a nivel de bits tiene la mayor precedencia, seguido por la O exclusiva (XOR) a nivel de bits, y después la O (OR) a nivel de bits.

La asociatividad para cada operador a nivel de bits es de izquierda a derecha.

2.39 Operadores de Desplazamiento

En la figura 2.16 se muestran los operadores de desplazamiento a nivel de bits:

Operador	Acción
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

Figura 2.16

Cada operador requiere dos operandos. El primero es un operando de tipo entero que representa el patrón de bits a desplazar. El segundo es un entero sin signo que indica el número de desplazamientos. Este valor no puede exceder del número de bits asociado con el tamaño de palabra del primer operando.

La forma general de una sentencia de desplazamiento a la derecha es la siguiente:

variable >> número de posiciones a desplazar

y de la sentencia de desplazamiento a la izquierda es:

variable << número de posiciones a desplazar

El operador de **desplazamiento a la izquierda** produce que los bits en el primer operando sean desplazados a la izquierda el número de posiciones indicado por el segundo operando, los bits de más a la izquierda (los bits de desbordamiento) en el patrón original de bits se perderán. las posiciones de la derecha que quedan vacantes se rellenan con ceros.

El operador de desplazamiento a la derecha produce que los bits en el primer operando sean desplazados a la derecha el número de

posiciones indicadas por el segundo operando. Los bits de más a la derecha (los bits de desbordamiento por abajo) en el patrón original de bits se perderán. Si el patrón de bits que se desplaza representa un entero sin signo, las posiciones de la izquierda que quedan vacantes se rellenan con ceros. Por tanto el comportamiento del operador de desplazamiento a la derecha es similar al del desplazamiento a la izquierda cuando el primer operando es un entero sin signo.

Si se desplaza hacia la derecha el patrón de bits correspondiente a un entero con signo, el resultado puede depender del valor del bit más a la izquierda (el bit de signo). La mayoría de los compiladores llenan las posiciones vacantes con el contenido de este bit. (Los valores enteros negativos tendrán este bit a 1, mientras que los positivos lo tendrán a 0.) Sin embargo, otros compiladores rellenarán las posiciones vacantes con ceros independientemente del bit de signo del entero original.

Las operaciones de desplazamiento de bits pueden ser muy útiles cuando se **decodifica** la entrada a través de dispositivos externos, como los **convertidores D/A**, y en la lectura de información de estado. Los operadores de desplazamiento también pueden utilizarse para llevar a cabo **operaciones muy rápidas de multiplicación y de división de enteros**. Un desplazamiento a la izquierda equivale a una multiplicación por dos y un desplazamiento a la derecha a una división por 2.

2.40 Operadores de Asignación a Nivel Bits.

El lenguaje C contiene también los operadores de asignación a nivel de bits, que se muestran en la figura 2.17 .

Operador	Acción
&=	Operación **AND** sobre bits más asignación.
|=	Operación **OR** sobre bits más asignación.
^=	Operación **XOR** sobre bits más asignación.
>>=	Desplazamiento a derecha más asignación.
<<=	Desplazamiento a izquierda más asignación.

Figura 2.17

Estos operadores combinan las operaciones precedentes a nivel de bits con la asignación. El operando de la izquierda debe ser

un identificador de tipo entero asignable (por ejemplo una variable entera), el operando de la derecha debe ser una expresión a nivel de bits. El operando de la izquierda se interpreta como el primer operando en la expresión a nivel de bits. El valor de la expresión a nivel de bits se asigna al operando de la izquierda. Por ejemplo, la expresión **a &= 0x7f** es equivalente a **a = a & 0x7f**.

Los operadores de asignación a nivel de bits son miembros del mismo grupo de precedencia que el resto de los operadores de asignación. Su asociatividad es de derecha a izquierda.

2.41 Operador Ternario (?)

En C el operador ternario (?) (se le denomina así porque requiere de tres operandos) se utiliza en expresiones condicionales, las cuales tienen la forma:

```
operando1 ? operando2 : operando3
```

La expresión **operando1** debe de ser de tipo entero, real o puntero. La evaluación se realiza de la siguiente forma:

- Si el resultado de la evaluación de **operando1** es distinta de 0, el resultado de la expresión condicional es **operando2**.
- Si el resultado de la evaluación de **operando1** es 0, el resultado de la expresión condicional es **operando3**.

Ejemplo:

```
mayor = (a > b) ? a : b
```

Si **a > b** entonces **mayor = a**, en caso contrario, **mayor = b**.

2.42 Operador Coma

De todos los operadores de C, el de menor prioridad es el operador coma. Se utiliza principalmente en los bucles **for**. Sentencia que veremos con mayor detalle en el siguiente capítulo.

Este operador permite que aparezcan dos expresiones en situaciones en donde sólo se utilizaría una expresión. Por ejemplo es posible escribir:

```
for(expresión1a, expresión1b; expresión2; expresión3) sentencia
```

en donde **expresión1a y expresión1b** son dos expresiones distintas, separadas por el operador coma (,). Normalmente sólo aparecería una expresión (**expresión1**). Estas operaciones se ocuparán de inicializar dos índices, que se utilizarán simultáneamente dentro del bucle **for**.

Análogamente se puede utilizar en una sentencia **for** el operador coma de la siguiente forma:

```
for(expresión1; expresión2; expresión3a, expresión3b) sentencia
```

Aquí **expresión3a y expresión3b**, separadas por el operador coma, aparecen en lugar de la expresión única que lo suele hacer. La forma frecuente de utilizar las dos expresiones es para alterar (incrementar o decrementar) los dos índices que se están utilizando simultáneamente dentro del bucle. Por ejemplo, un índice podría irse incrementando mientras el otro decrece.

En resumen el operador coma se utiliza para permitir sentencias de inicialización o de incremento/decremento múltiples.

También éste operador puede encadenar varias expresiones. El valor de la lista de expresiones separadas por comas es el valor de la expresión de más a la derecha. Se ignora el valor de las otras expresiones. Esto significa que la expresión de la parte derecha se convierte en el valor de la expresión total separada por comas.

Ejemplo:

```
var = (cuenta = 19, incr = 10, cuenta + 1);
```

Primero se asigna el valor 19 a **cuenta**, después a **incr** se le asigna el valor de 10 y finalmente se le asigna a **var** el valor de 20 (la suma del valor que tiene **cuenta** + 1).

El operador coma se encuentra sólo dentro de su grupo de precedencia, por debajo del grupo de precedencia de los distintos operadores de asignación. Su asociatividad es de izquierda a derecha.

2.43 Operador sizeof

En C, un carácter es equivalente a 1 byte. Sin embargo, el resto de tipos de datos incorporados, incluyendo int, float y double, no tiene longitudes definidas. Por ejemplo, en algunas implementaciones un float puede ocupar 4 bytes y en otras 6 bytes. A menudo es necesario conocer la longitud exacta del tipo o tipos de datos básicos con los que se está trabajando. Para realizar esto, C suministra el **operador en tiempo de compilación sizeof**.

El operador **sizeof** devuelve el tamaño en bytes de la variable o tipo de su operando. Este es un valor sin signo. La forma general de **sizeof** es la siguiente:

```
sizeof(expresión)
```

donde **expresión** es un identificador o un tipo básico. Los paréntesis son opcionales excepto cuando el identificador es un tipo básico.

Si el identificador es un array, el resultado es el tamaño total del array.

El operador **sizeof** ayuda a construir código portable y se usa con rutinas de asignación dinámica de C.

Ejemplo:

```
int b = sizeof(int)
```

El resultado que se obtiene en b es 2.

2.44 Los Operadores de Puntero & y *

Un **puntero** es la dirección de memoria de una variable. Una **variable puntero** es una variable especificamente declarada para contener un puntero a su tipo específico. El conocer la dirección de una variable puede ser una gran ayuda en ciertos tipos de rutinas. Sin embargo, en C los punteros tiene tres funciones básicas:

a) Proporcionan una rápida forma de referenciar los elementos de un array.

b) Permiten a las funciones de C modificar los parámetros de llamada.

c) Dan soporte a las listas enlazadas y a otras estructuras de datos dinámicas.

De ésto se hablará con mayor profundidad en el capítulo relacionado a los punteros. Por el momento sólo hablaremos de los dos operadores que se usan para manipular punteros.

El primer operador de punteros es &, un operador monario que devuelve la dirección de memoria del operando.

Por ejemplo:

```
m = &cont;
```

coloca en m la dirección de memoria de la variable **cont**. Esta es

la dirección de la posición interna en la computadora de la variable. No tiene nada que ver con el valor de **cont**. Se puede pensar en & como "la dirección de". Por tanto, la sentencia anterior de asignación significa "m recibe la dirección de **cont**".

Para comprenderlo mejor, supongamos que la variable **cont** utiliza la posición de memoria 2000 para guardar su valor. El valor de **cont** es 100. Después de la asignación, **m** tendrá el valor de 2000.

El segundo operador de punteros es *, que es el complemento de &. Es un operador monario que devuelve el valor de la variable ubicada en la dirección que se especifica. Por ejemplo, si **m** contiene la dirección de memoria de la variable **cont**, entonces:

```
q = *m;
```

colocará el valor de **cont** en **q**. Por lo tanto, **q** tendrá el valor 100, que es lo guardado en la posición 2000. Hay que pensar en * como "en la dirección". En este caso la sentencia de asignación significa "q recibe el valor en la dirección m."

Por desgracia, el signo de la operación Y (AND) y el de "la dirección de" son el mismo y el de multiplicación y el de "en la dirección" también lo es. Estos operadores no tiene relación entre sí. Tanto & como * tienen una precedencia mayor que cualquier operador aritmético, excepto el menos monario, respecto al cual la tienen igual.

Las variables que vayan a contener punteros se han de declarar como tales. Las variables que vayan a contener direcciones de memoria, o punteros, como se llaman en C, deben declararse colocando un * delante del nombre de la variable. Esto indica al compilador que va a contener un puntero a ese tipo de variable. Por ejemplo, para declarar c como puntero a carácter podemos escribir:

```
char *c;
```

Aquí, **c** no es un carácter, sino un puntero a un carácter. (hay una gran diferencia)

El tipo de dato al que apunta un puntero, en este caso **char**, se denomina **tipo base** del puntero. Sin embargo, la propia variable puntero es una variable que mantiene la dirección de un objeto del tipo base. Así, un puntero a carácter (o cualquier puntero, generalizando) tiene un tamaño suficiente para guardar una dirección tal como esté definida por la arquitectura de la computadora que se utilice. Sin embargo, un puntero sólo debe ser usado para apuntar a datos que sean del tipo base del puntero.

Se pueden mezclar directivas de puntero y normales en la misma sentencia de declaración. Por ejemplo;

```
int x, *y, cont;
```

En este caso se declara **x** y **cont** como de tipo entero, **e** y como puntero a un tipo entero.

El operador **&** no se puede aplicar a un campo de bits pertenecientes a una estructura o a un identificador declarado con el especificador **register**.

2.45 Los Operadores Punto (.) y Flecha (->)

Los operadores punto (.) y flecha (->) hacen referencia a elementos individuales de las estructuras y de las uniones.

Como ya sabemos las estructuras y las uniones son **tipos de datos compuestos** que se pueden referenciar bajo un sólo nombre.

El operador punto se usa cuando se trabaja realmente con la estructura o la unión. El operador flecha se usa cuando se utiliza un puntero a una estructura o a una unión.

Tanto el operador punto como el operador flecha pertenecen al grupo de mayor precedencia. Su asociatividad es de izquierda a derecha.

2.46 Los Paréntesis y los Corchetes como Operadores

En C, los paréntesis son operadores que aumenta la precedencia de las operaciones que contienen.

Los corchetes llevan a cabo el indexamiento de arrays. Dado un array, la expresión entre corchetes proporciona un índice para el array.

Tanto los paréntesis como los corchetes se encuentran en el grupo de mayor precedencia. Su asociatividad es de izquierda a derecha.

2.47 Expresiones

Los operadores, las constantes y las variables son los constituyentes de las expresiones. Una **expresión** en C es cualquier combinación válida de estos elementos.

El estándar ANSI estipula que las subexpresiones de una expresión pueden ser evaluadas en cualquier orden. Esto deja libre al compilador de C para reorganizar una expresión de forma que se obtenga el código óptimo. Sin embargo, esto también significa que nunca se debe basar el código en el orden de evaluación de una expresión. Por ejemplo, la expresión:

```
x = f1() + f2();
```

no asegura que se llame a f1() antes que a f2().

2.48 Reglas de Conversión de Tipos en las Expresiones

La conversión de tipos en las expresiones se rigen por las siguientes reglas:

1.- Si uno de los operandos es **long double**, el otro será convertido a **long double** y el resultado será un **long double**.

2.- Si uno de los operandos es **double**, el otro se converirá a **double** y el resultado será **double**.

3.- Si uno de los operandos es **float**, el otro será convertido a **float** y el resultado será **float**.

4.- Si uno de los operandos es **unsigned long int**, el otro será convertido a **unsigned long int** y el resultado será **unsigned long int**.

5.- Si uno de los operandos es **long int** y el otro es **unsigned int**, entonces:

   a) Si **unsigned int** puede convertirse a **long int**, el operando **unsigned int** será convertido y el resultado será **long int**.

   b) En otro caso, ambos operandos será convertidos a **unsigned long int** y el resultado será **unsigned long int**.

6.- En otro caso, si uno de los operandos es **long int**, el otro será convertido a **long int** y el resultado será **long int**.

7.- En otro caso, si uno de los operandos es **unsigned int**, el otro será convertido a **unsigned int** y el resultado será **unsigned int**.

8.- Si no se puede aplicar ninguna de las condiciones anteriores, entonces ambos operandos serán convertidos a **int** (si es necesario) y el resultado será **int**, tal como ocurre en los siguientes casos:

   a) Cualquier operando de tipo **char** o **short** es convertido a **int**.

   b) Cualquier operando de tipo **unsigned char** o **unsigned short** es convertido a tipo **unsigned int**.

Es importante mencionar que en algunas versiones de C, todos los operandos de tipo **float** se convierten automáticamente en **double**.

En resumen, una vez aplicadas las reglas de conversión de tipos, cada par de operandos será de un mismo tipo y el resultado

de cada operación será del mismo tipo que el de los dos operandos.

2.49 Moldes en las Expresiones (Cast)

Es posible forzar a que una expresión sea de un tipo determinado utilizando una construcción denominada **molde** o **cast**. La forma general de un molde es:

```
(tipo)expresión
```

donde **tipo** es un tipo válido de C.

Por ejemplo, si se quiere asegurar que la expresión **x/2** se evalúe como de tipo **float**, se puede escribir lo siguiente:

```
(float) x/2
```

A menudo los moldes son considerados como operadores. Como tal, es monario y tiene la misma precedencia que cualquier otro operador monario.

Los **moldes o cats** se utilizan a menudo para garantizar que el código será transportable de una computadora a otra. Por ejemplo, algunas veces las conversiones de C por defecto producen códigos que justamente trabajan sobre la máquina "A" pero no funcionan sobre la máquina "B".

CAPITULO III

SENTENCIAS

NO

EXISTE

PAGINA

NO
EXISTE
PAGINA

SENTENCIAS

Una sentencia hace que la computadora lleve a cabo alguna acción. Hay tres tipos diferentes de sentencias en C:

a) Sentencias de Expresión.

b) Sentencias Compuestas.

c) Sentencias de Control.

3.1 Sentencias de Expresión

Una sentencia de expresión consiste en una expresión seguida de un punto y coma (;). La ejecución de esta sentencia hace que se evalúe la expresión.

Ejemplos:

```
a = 3;
c = a + b;
++i;
;
```

Las dos primeras sentencias de expresión son **sentencias de tipo asignación**. Cada una hace que el valor de la expresión a la derecha del signo (=) le sea asignado a la variable de la izquierda.

La tercera sentencia de expresión es una **sentencia de tipo incremento**, que hace que el valor de i sea incrementado en 1.

La cuarta sentencia no hace nada, ya que consta sólo de un punto y coma. Es un sencillo mecanismo de conseguir una sentencia de expresión vacía en lugares en donde se requiera. Comunmente se le denomina **sentencia nula**.

3.2 Sentencias Compuestas

Una **sentencia compuesta** está formada por varias sentencias individuales encerradas con un par de llaves ({ y }). Las sentencias individuales pueden ser a su vez sentencias de expresión, compuestas o de control.

A diferencia de una sentencia de expresión, una sentencia compuesta no acaba con un punto y coma.

En la siguiente página se muestra un claro ejemplo de lo que es una sentencia compuesta.

Ejemplo:

```
{
   pi = 3.141593;
   circunferencia = 2. * pi * radio;
   area = pi * radio * radio;
}
```

Esta sentencia compuesta consta de tres sentencias de expresión de tipo asignación.

3.3 Sentencias de Control

Las **sentencias de control** se utilizan para conseguir ciertas acciones especiales en los programas, tales como comprobaciones lógicas, bucles y ramificaciones.

C dispone de un rico y muy variado conjunto de sentencias de control. El estándar ANSI clasifica este tipo de sentencias en los siguientes grupos:

a) Sentencias de Selección.

b) Sentencias de Iteración.

c) Sentencias de Salto.

d) Sentencias de Etiquetado.

La mayoría de las sentencias de control contienen sentencias de expresión o sentencias compuestas (o en algunos casos ambas).

Las sentencias de control comunmente definen el carácter de un lenguaje de programación; ya que permiten especificar qué operaciones se van a realizar y en qué orden. Por lo que determinan el flujo de control en un programa.

Muchas sentencias de C se basan en una prueba condicional que determina la acción que se va a llevar a cabo. Una expresión condicional tiene como resultado un valor cierto o falso. A diferencia de muchos otros lenguajes de computadora, en C, cualquier valor distinto de cero es cierto, incluyendo los números negativos. El cero es el único valor falso.

3.4 Sentencias de Selección

Las sentencias de selección son aquellas que dependiendo del valor de una expresión, variable o constante se realiza una determinada tarea.

En el lenguaje C existen las siguientes sentencias de selección:

a) if

b) switch

3.5 Sentencia if

La forma general de la sentencia if es:

```
if(expresión) sentencia;
else sentencia;
```

donde sentencia puede ser una sentencia simple, compuesta, o nada (en el caso de sentencias vacías). La cláusula **else** es opcional.

Si la expresión del **if** es cierta (cualquier valor que no sea 0), se ejecuta la sentencia o el bloque de sentencias que contituye el objetivo del if; en cualquier otro caso se ejecuta la sentencia o el bloque de sentencias que contituye el objetivo del **else**, en caso de que exista. Es importante remarcar que sólo se ejecuta el código asociado al **if** o al **else**, nunca ambos.

La sentencia condicional que controla el **if** debe producir un resultado escalar. Un **escalar** es cualquiera de los tipos enteros, de carácter o de coma flotante. Sin embargo, es raro usar un número en coma flotante para controlar una sentencia condicional, ya que disminuye la velocidad de ejecución considerablemente. (A la C.P.U. le lleva varias instrucciones el realizar una operación en coma flotante. Necesita relativamente pocas instrucciones para llevar a cabo una operación entera o de carácter.)

En el lenguaje C existen los **if's anidados**. Un **if anidado** es un **if** que es el objeto de otro **if** o **else**, son muy comunes en programación. Una sentencia **else** siempre se refiere al **if** más próximo que esté en un mismo bloque, por ejemplo:

```
if(i) {

   if(j) sentencia1;
   if(k) sentencia2;        /* este if está */
   else  sentencia3;        /* asociado con este else */

}
else sentencia4;
```

El **else** final no está asociado a **if(j)**, ya que no están en el mismo bloque. En cambio, si esta en el bloque del **if(i)**.

El else interno está asociado con if(k), que es el if más cercano.

El estándar ANSI especifica que al menos se deben permitir 15 niveles de anidamiento. En la práctica, la mayoría de los compiladores permiten más.

Una construcción común en programación es la escala if-else-if, a veces denominada **escalera if-else-if**. Su forma general es:

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

Aquí las expresiones se evalúan de arriba hacia abajo. Tan pronto como se encuentre una condición cierta, se ejecuta la sentencia asociada con ella y se pasa por alto el resto de la escala. Si ninguna de las condiciones es cierta, se ejecuta el else final en caso de que exista.

Aunque el sangrado de la escala if-else-if anterior es técnicamente correcto, puede llevar a un exceso de sangrado. Por esta razón, la escala if-else-if se escribe generalmente como sigue:

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

Es importante recordar que también se puede usar el operador ternario ? para reemplazar las sentencias if-else.

3.6 Sentencia switch

C incorpora una sentencia de selección múltiple, que es la denominada **switch**, la cual compara sucesivamente el valor de una

expresión con una lista de constantes enteras o de carácter. Cuando se encuentra una correspondencia, se ejecutarán las sentencias asociadas con la constante.

La forma general de la sentencia **switch** se muestra acontinuación:

```
switch (expresión) {
  [declaraciones]
  case contante_1:
    secuencia de sentencias
    break;
  case contante_2:
    secuencia de sentencias
    break;
  case contante_3:
    secuencia de sentencias
    break
    .
    .
    .
  default:
    secuencia de sentencias
}
```

Se comprueba el valor de la expresión por orden, con los valores de las constantes especificadas en las sentencias **case**. Cuando se encuentra una correspondencia, se ejecuta la secuencia de sentencias asociada con ese **case**, hasta que se encuentra la sentencia **break** o el final de la sentencia **switch**.

Al principio del cuerpo de la sentencia **switch**, pueden aparecer declaraciones. Las inicializaciones, si las hay, son ignoradas.

La sentencia **default** se ejecuta si no se ha encontrado ninguna correspondencia. Esta sentecia es opcional y si no está presente, no se ejecuta ninguna acción al fallar todas las comprobaciones. **default** puede colocarse en cualquier parte de la forma general del **switch** y no necesariamente el final.

El estándar ANSI especifica que un **switch** ha de poder contener al menos 257 sentencias **case**. Por razones de eficiencia, lo mejor es limitar el número de sentencias **case** a un número menor.

La sentencia **case** no tiene entidad por sí misma fuera de un **switch**.

La sentencia **break** es una de las sentencias de salto del lenguaje C. Se puede usar en bucles además de la sentencia **switch**. Técnicamente, ésta sentencia es opcional dentro de un **switch**. Cuando está presente, la ejecución del programa "salta" a la línea de código que sigue a la sentencia **switch**.

Existen cuatro caracteristicas importantes que se deben saber sobre la sentencia **switch**:

a) Esta sentencia es diferente de **if** ya que sólo puede comprobar la igualdad, mientras que en un **if** se pueden evaluar expresiones relacionales o lógicas.

b) No puede haber dos constantes **case** en el mismo **switch** que tengan los mismos valores. Por supuesto, una sentencia **switch** contenida en otra sentencia **switch** puede tener constantes **case** que sean iguales.

c) Si se utilizan constantes de tipo carácter en la sentencia **switch**, esto se convierten automáticamente a sus valores enteros.

d) Una sentencia **switch** es más eficiente que los **if** anidados.

Las sentencias **switch** a menudo se utilizan para procesar órdenes introducidas por teclado, como la orden de selección de un menú.

Hay que tener en cuenta que las sentencias asociadas con cada **case**, no son bloques de código, sino secuencia de sentencias.

Se puede tener un **switch** formando parte de la secuencia de sentencias de otro **switch**. No aparecerán conflictos si las constantes **case** del **switch** interior y del exterior contienen valores comunes. Cuando existen este tipo de estructuras se les conoce como sentencias **switch anidadas**. El estándar ANSI especifica que se han de permitir al menos 15 niveles de anidamiento

3.7 Sentencias de Iteración.

En C y en todos los lenguajes de programación modernos, las sentencias de iteración (también denominadas de bucles) permiten que un conjunto de instrucciones sean ejecutadas hasta que se alcance una cierta condición. Esta condición puede estar predefinida (como en el bucle **for**) o no haber final determinado (como los bucles **while** y **do-while**).

Las sentencias de iteración que maneja el lenguaje C son:

a) for

b) while

c) do-while

3.8 Sentencia for.

Cuando se desea ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido, la construcción adecuada es la sentencia **for**.

La sentencia **for** proporciona una potencia y flexibilidad sorprendentes.

La forma general para la sentencia **for** es:

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

La **inicialización** normalmente es una sentencia de asignación que se utiliza para iniciar la variable de control del bucle.

La **variable de control** del bucle es la variable que actúa como contador que controla el bucle.

La **condición** es una expresión relacional que determina cuando finaliza el bucle.

El **incremento** define cómo cambia la variable de control cada vez que se repite el bucle. Se puede incrementar o inclusive decrementar en cualquier cantidad.

Estas tres secciones principales deben estar separadas por un punto y coma (;).

Un aspecto importante de los bucles **for** es que la prueba de la condición se hace siempre al principio del bucle. Esto supone que el código dentro del bucle se ejecuta mientras que la condición sea cierta. Una vez que la condición sea falsa, la ejecución del programa sigue con la sentencia siguiente al **for**.

La sentencia **for** tiene muchas variantes. Una de las variaciones más comunes es que se utiliza el operador coma para permitir dos o más inicializaciones de variables de control del bucle, así como dos o más expresiones de incremento.

Es importante recordar que el operador coma se utiliza para encadenar un número de expresiones de la forma "hacer esto y lo otro".

Se puede tener cualquier número de sentencias de inicialización y de incremento, pero en la práctica, más de dos o tres hacen que el bucle **for** esté sobrecargado y sea difícil de leer.

La **condición** del bucle puede ser cualquier sentencia de C, relacional o lógica. No se limita sólo a la comparación de la variable de control del bucle con algún otro valor.

Otro aspecto interesante de **for** y que es diferente en otros

lenguajes de programación es que puede no tener todas las secciones de definición del bucle. Puede no tener incremento y sentencias de inicialización, pero si es requisito que estén presentes los separadores (;) de cada una de las secciones. Por ejemplo se puede crear un bucle infinito (un bucle que nunca termina) mediante esta construcción:

```
for(;;)
{
  .
  .
  .
}
```

En los programas se usan con frecuencia los "bucles de retardo de tiempo". Estos bucles dejar pasar un tiempo considerable antes de que se ejecute la siguiente instrucción dentro de un programa.

Un ejemplo de esto sería:

```
for(x=0; x<1000; x++) ;
```

Aquí, lo único que pasa es que se incrementa **x** de uno en uno hasta 1000. El punto y coma que termina la línea es necesario porque **for** espera una sentencia, la cual puede estar vacía.

Un bucle **for** puede colocarse dentro de otro bucle **for** y entonces se dice que están anidados. En este caso el bucle interno se ejecutará totalmente, por cada valor del bucle externo que lo contiene.

3.9 Sentencia while

El segundo bucle disponible en C es el bucle **while**. Su forma general es:

```
while(condición)sentencia;
```

donde **sentencia** puede ser una sentencia vacía, simple o un bloque de sentencias que se repiten. Además en ella se deben incluir algún elemento que altere el valor de la expresión, que forma la condición, proporcionando así la salida del bucle.

La **condición** puede ser cualquier expresión. Frecuentemente es una expresión lógica que es cierta o falsa.

El bucle itera mientras que la condición es cierta, cuando la condición se hace falsa, el control del programa pasa a la línea siguiente, al código del bucle.

Al igual que el bucle **for**, el bucle **while** comprueba la condición al principio, lo que supone que el código del bucle

puede que no se ejecute. Esto elimina la necesidad de hacer una evaluación aparte antes del bucle.

3.10 Sentencia do-while

A diferencia de los bucles **for** y **while**, que analizan la condición del bucle al principio del mismo, el bucle **do-while** analiza la condición al final. Esto significa que el bucle siempre se ejecuta al menos una vez.

La forma general del bucle **do-while** es:

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

Aunque las llaves no son necesarias cuando sólo hay una sentencia, se utilizan normalmente para evitar confusiones con el **while**. El bucle **do-while** itera hasta que la condición se hace falsa.

El uso más común de este tipo de bucle, es en rutinas de selección por menú. Ya que al usarlo se pueden mostrar las opciones que contiene el menú en pantalla e iterar hasta que se selecciona una opción válida.

La **sentencia** podrá ser simple o compuesta aunque en la mayoría de las veces será compuesta.

La **condición** es frecuentemente una expresión lógica que puede ser cierta (con valor no nulo) o falsa (con valor nulo). La sentencia incluida se repetirá si la expresión lógica es cierta.

Es importante mencionar que para la mayoría de las aplicaciones es más natural comprobar la condición al comienzo del bucle que al final. Por esta razón, la sentencia **do-while** se utiliza con menor frecuencia que la sentencia **while**.

3.11 Sentencias de Salto

C tiene cuatro sentencias que llevan a cabo un salto incondicional y son :

a) **return**

b) **goto**

c) **break**

d) **continue**

De ellas se puede usar **return** y **goto** en cualquier parte del programa. Las sentencias **break** y **continue** se deben usar junto con una sentencia de bucle. **break** también se puede usar con la sentencia **switch**.

En las siguientes secciones se presenta una descripción de cada una de estas sentencias.

3.12 Sentencia return

La sentencia **return** se usa para volver de una función. Se trata de una sentencia de salto porque hace que la ejecución vuelva (o salte hacia atrás) al punto en que se hizo la llamada a la función. Si existe algún valor asociado con **return**, se trata del valor de vuelta de la función. Si no se especifica un valor de vuelta, se asume que se devuelve un valor sin sentido. (Algunos compiladores de C devuelven automáticamente el valor 0)

La forma general de la sentencia **return** es:

```
return expresión;
```

Donde la **expresión** es opcional.

Se pueden usar tantas sentencias **return** como se quiera en una función. Sin embargo, la función termina tan pronto como encuentra al primer **return**. Es importante mencionar que la llave (}) que termina el código de la función también hace que se vuelva de ella. Equivale a un **return** sin valor específico.

Una función declarada como **void** puede no contener una sentencia **return** que especifique un valor.

3.13 Sentencia goto y Sentencias de Etiqueta

La sentencia **goto** se utiliza para alterar la secuencia de ejecución normal del programa, transfiriéndose el control a otra parte de él. En su forma general, esta sentencia se escribe:

```
goto etiqueta;
```

en donde **etiqueta** es un identificador que se utiliza para señalar la sentencia a la que se transferirá el control.

Se puede transferir el control a cualquier otra sentencia del programa. (Para ser más precisos, se puede transferir el control a cualquier lugar dentro de la función actual, no se permite transferir el control fuera de la misma.)

La sentencia por la que se continuará la ejecución debe encontrarse etiquetada, y la etiqueta debe encontrarse seguida de dos puntos (:). Por tanto, la sentencia etiquetada aparecerá de la siguiente forma:

```
etiqueta: sentencia
```

Cada sentencia etiquetada dentro de un programa (más concretamente, dentro de la función actual) debe tener una única etiqueta.

Todos los lenguajes de propósito general populares poseen la sentencia **goto**, aunque actualmente todas las técnicas de programación instan a EVITAR su utilización.

En algunos de los lenguajes más viejos, como FORTRAN y BASIC, la sentencia **goto** se utiliza con gran profusión. Las aplicaciones más comunes son las siguientes:

1.- Bifurcar a sentencias o grupos de sentencias bajo determinadas condiciones.

2.- Saltar al fin de un bucle en determinadas condiciones, no ejecutándose por tanto el resto del bucle en la pasada actual.

3.- Finalizar totalmente la ejecución de un bucle bajo determinadas condiciones.

Los elementos estructurados del C permiten realizar todas estas operaciones sin recurrir a la sentencia **goto**.

Por ejemplo, la bifurcación se puede realizar mediante la sentencia **if-else**; saltar al final de un bucle se puede hacer mediante la sentencia **continue**; y terminar la ejecución de un bucle mediante la sentencia **break**.

Es preferible la utilización de estos elementos estructurados al de la sentencia **goto** porque el uso de éste tiende a aumentar (o al menos, no a disminuir) el que se disperse la lógica del programa.

Los elementos estructurados de C requieren que todo el programa se escriba de una forma ordenada y secuencial. Por esta razón, la utilización de esta sentencia se debe evitar, como norma general, dentro de los programas en C.

A veces se presentan situaciones de forma esporádica en las que la sentencia **goto** puede ser útil. Por ejemplo, en una situación en la que se necesita salir de un bucle doblemente anidado al detectar determinada condición. Esto se puede hacer mediante dos sentencias **if-break**, una dentro de cada bucle, aunque esto es un

tanto problemático. Una solución mejor podría ser el utilizar una sentencia **goto** para transferir el control fuera de ambos bucles de una vez.

Si se va a usar el lenguaje C como sustituto del ensamblador, se debe conocer bien la sentencia **goto**, ya que, bajo ciertas condiciones, permite crear código muy pequeño y rápido. Sin embargo, siempre hay que tener cuidado en el manejo de esta sentencia.

Por último, se remarca que sólo se puede usar la sentencia **goto** para saltar a una etiqueta que este en la misma función. Para los saltos no locales se utiliza la función de biblioteca estándar **longimp()**.

3.14 Sentencia break

La sentencia **break** se utiliza para terminar la ejecución de bucles o salir de una sentencia **switch**.

Es posible crear un bucle infinito en C mediante las sentencias **for**, **while** y **do-while**, y para salir de ese tipo de bucle así como para los bucles no infinitos se utiliza comunmente la sentencia **break**.

La sentencia **break** se puede escribir sencillamente de la siguiente forma:

```
break;
```

sin contener ninguna otra expresión o sentencia.

Cuando se encuentra la sentencia **break** dentro de un bucle, el bucle finaliza inmediatamente y el control pasa a la sentencia que sigue al bucle.

Esto proporciona una forma conveniente de terminar un bucle cuando se detecta un error o alguna otra condición irregular.

Es importante mencionar que en caso de que se tengan bucles anidados la sentencia **break** dá lugar a la salida sólo del bucle donde este incluida.

Un **break** utilizado en una sentencia **switch** afecta sólo a ese **switch**. No afecta a cualquier bucle que haya dentro del switch.

3.15 Sentencia continue

La sentencia **continue** se utiliza para saltarse el resto de la pasada actual a través de un bucle. El bucle no termina cuando se encuentra una sentencia continue, sencillamente no se ejecutan las sentencias que se encuentren acontinuación en él y se salta

directamente a la siguiente pasada a través del bucle.

La sentencia **continue** se puede incluir dentro de una sentencia **while, do-while, o for**, simplemente se escribe así:

```
continue;
```

sin incluir otras sentencias o expresiones.

CAPITULO IV

ARRAY Y PUNTEROS

CAPITULO IV

ARRAY Y PUNTEROS

NO

EXISTE

PAGINA

4.1 Array

Un array es una colección de variables de un mismo tipo que responden a un nombre común.

En C todos los arrays constan de posiciones de memoria contiguas, con la dirección más baja correspondiendo al primer elemento y la dirección más alta al último elemento.

Los arrays pueden tener de una a varias dimensiones (como son los arreglos unidimensionales, bidimensionales y multidimensionales).

El número máximo de dimensiones o el número máximo de elementos para un array depende de la memoria disponible.

A un elemento específico del array se accede mediante un **índice**, el cual puede ser representado por una constante, una variable o una expresión cualquiera.

El lenguaje C no incorpora un tipo de datos de cadena, en su lugar, se utilizan los arrays de caracteres, obteniéndose con ello una mayor flexibilidad y poder que la que disponen ciertos lenguajes que utilizan el tipo especial de cadena.

4.2 Arrays Unidimensionales

La forma general de declaración de un array unidimensional es:

```
tipo nombre_variable[tamaño];
```

En donde **tipo** declara el **tipo base de array**. Es decir, se determina el tipo de dato de cada elemento del array. **nombre_variable** es el nombre propiamente del array y el valor de **tamaño** indica cuantos elementos mantendrá el array.

En la figura 4.1 se muestra la representación del array unidimensional X de n elementos:

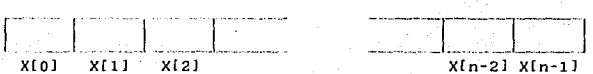


Figura 4.1

Como en otros lenguajes de programación, los arrays tienen que declararse explícitamente para que así el compilador pueda reservar espacio en memoria para ellos.

Por ejemplo, la siguiente sentencia declara un array entero de 100 elementos:

```
int muestra[100];
```

En C todos los arrays tienen el cero como índice de su primer elemento.

Los arrays unidimensionales son en esencia listas de información del mismo tipo.

La cantidad de memoria requerida para guardar un array está directamente relacionada con su tipo y su tamaño. Para un array unidimensional, el tamaño total en bytes se calcula de la manera siguiente:

```
total de bytes = número de bytes de tipo * número de elementos
```

Los arrays son muy comunes en programación puesto que permiten tratar fácilmente un gran número de variables relacionadas.

Es importante mencionar que no se puede asignar un array a otro. Para transferir el contenido de un array a otro, se tiene que asignar cada valor individualmente.

El lenguaje C no comprueba los límites de los arrays, ya que nada le impedirá transpasar cualquier extremo de un array. Si esto llegara a ocurrir durante una operación de asignación, puede que se asignen valores a los datos de alguna otra variable o incluso a una parte del código del programa.

En otras palabras, un array de tamaño n se puede referenciar mediante un índice con valores mayores que n sin que se produzca ningún mensaje de error de compilación o de ejecución.

Básicamente es tarea del programador proporcionar una comprobación de los límites de un array cuando sea necesario.

El lenguaje C no porporciona una comprobación de los límites de los arrays simplemente porque se diseño para reemplazar a la codificación en lenguaje ensamblador, y por ello se incluyó poca comprobación de errores, ya que ésto hacía más lenta la ejecución de los programas.

4.3 Arrays Bidimensionales

El lenguaje C soporta arrays multidimensionales, y la forma más simple de un array multidimensional es el array bidimensional (un array de arrays unidimensionales).

Los arrays bidimensionales se almacenan en matrices fila-columna, en las que el primer índice indica la fila y el segundo indica la columna. Esto significa que el índice más a la derecha cambia más rápido que el de más a la izquierda cuando accedemos a elementos del array en el orden en que realmente se han almacenado en memoria.

En la figura 4.2 se muestra la representación de un array bidimensional X de m filas y n columnas

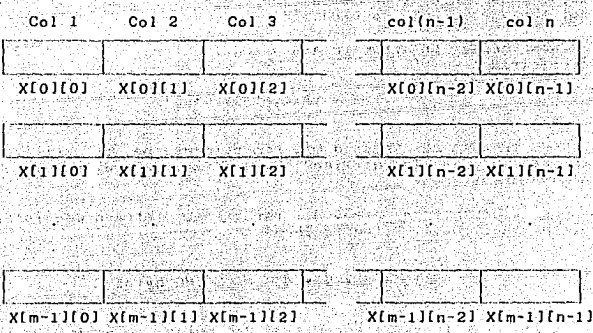


Figura 4.2

Por ejemplo, para declarar un array d de enteros bidimensional de tamaño 10, 20, se escribiría la siguiente sentencia:

```
int d[10][20];
```

Es importante hacer notar, que a diferencia de la gran mayoría de los lenguajes de computadora, que utilizan comas para separar las dimensiones del array, C coloca cada dimensión en su propio conjunto de corchetes.

En el caso de un array bidimensional, la siguiente fórmula dá el número de bytes de memoria necesarios para guardarlo:

```
total de bytes = filas * columnas * número de bytes del tipo
```

4.4 Arrays Multidimensionales

Los arrays multidimensionales son definidos prácticamente de la misma manera que los arrays unidimensionales, excepto que se requiere un par separado de corchetes para cada índice. Así, un array bidimensional requerirá dos pares de corchetes, un array tridimensional requerirá tres, y así sucesivamente.

En términos generales, la declaración de un array multidimensional puede escribirse como:

```
tipo array[expresión_1][expresión_2]...[espresión_N];
```

donde **tipo** es un tipo base de datos, **array** es el nombre del arreglo y **expresión_1**, **expresión_2**, . . . , **expresión_N** son expresiones enteras positivas que indican el número de elementos del array asociado con cada índice.

Para crear un array tridimensional entero de 4 x 10 x 3 se escribiría:

```
int multidim[4][10][3];
```

Los arrays de tres o más dimensiones no se utilizan con frecuencia por la cantidad de memoria que se requiere para almacenarlos.

Por ejemplo, un array de caracteres tetra-dimensional con dimensiones 10, 6, 9, 4 requiere:

```
10 * 6 * 9 * 4   =  2 160 bytes
```

Si el array fuera de enteros de dos bytes, se necesitarían 4320 bytes. El almacenamiento requerido aumenta exponencialmente con el número de dimensiones. Es importante tener encuenta que un programa con arrays de más de tres o cuatro dimensiones puede agotar la memoria rápidamente.

Si la definición de un array multidimensional incluye la asignación de valores iniciales, se debe tener cuidado en el orden en que los valores iniciales son asignados a los elementos del array. La regla es que el último índice (el de más a la derecha) es el que se incrementa más rápidamente, y el primer índice (el de más a la izquierda) es el que se incrementa más lentamente. Así, los elementos de un array bidimensional deben ser asignados por filas, esto es, primero serán asignados los elementos de la primera fila, luego los elementos de la segunda, y así sucesivamente.

4.3 Arrays Bidimensionales

El lenguaje C soporta arrays multidimensionales, y la forma más simple de un array multidimensional es el array bidimensional (un array de arrays unidimensionales).

Los arrays bidimensionales se almacenan en matrices fila-columna, en las que el primer índice indica la fila y el segundo indica la columna. Esto significa que el índice más a la derecha cambia más rápido que el de más a la izquierda cuando accedemos a elementos del array en el orden en que realmente se han almacenado en memoria.

En la figura 4.2 se muestra la representación de un array bidimensional X de m filas y n columnas.

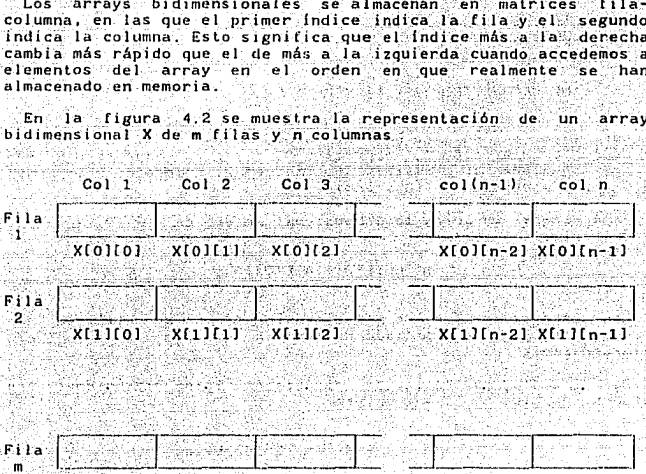


Figura 4.2

Por ejemplo, para declarar un array d de enteros bidimensional de tamaño 10, 20, se escribiría la siguiente sentencia:

```
int d[10][20];
```

Es importante hacer notar, que a diferencia de la gran mayoría de los lenguajes de computadora, que utilizan comas para separar las dimensiones del array, C coloca cada dimensión en su propio conjunto de corchetes.

En el caso de un array bidimensional, la siguiente fórmula dá el número de bytes de memoria necesarios para guardarlo:

total de bytes = filas * columnas * número de bytes del tipo

4.4 Arrays Multidimensionales

Los arrays multidimensionales son definidos prácticamente de la misma manera que los arrays unidimensionales, excepto que se requiere un par separado de corchetes para cada índice. Así un array bidimensional requerirá dos pares de corchetes, un array tridimensional requerirá tres, y así sucesivamente.

En términos generales, la declaración de un array multidimensional puede escribirse como:

```
tipo array[expresión_1][expresión_2]...[espresión_N];
```

donde **tipo** es un tipo base de datos, **array** es el nombre del arreglo y **expresión_1, expresión_2, . . . , expresión_N** son expresiones enteras positivas que indican el número de elementos del array asociado con cada índice.

Para crear un array tridimensional entero de 4 x 10 x 3 se escribiría:

```
int multidim[4][10][3];
```

Los arrays de tres o más dimensiones no se utilizan con frecuencia por la cantidad de memoria que se requiere para almacenarlos.

Por ejemplo, un array de caracteres tetra-dimensional con dimensiones 10, 6, 9, 4 requiere:

```
10 * 6 * 9 * 4   =  2 160 bytes
```

Si el array fuera de enteros de dos bytes, se necesitarían 4320 bytes. El almacenamiento requerido aumenta exponencialmente con el número de dimensiones. Es importante tener encuenta que un programa con arrays de más de tres o cuatro dimensiones puede agotar la memoria rápidamente.

Si la definición de un array multidimensional incluye la asignación de valores iniciales, se debe tener cuidado en el orden en que los valores iniciales son asignados a los elementos del array. La regla es que el último índice (el de más a la derecha) es el que se incrementa más rápidamente, y el primer índice (el de más a la izquierda) es el que se incrementa más lentamente. Así, los elementos de un array bidimensional deben ser asignados por filas, esto es, primero serán asignados los elementos de la primera fila, luego los elementos de la segunda, y así sucesivamente.

4.5 Inicialización de Arrays

El lenguaje C permite la inicialización de arrays. La forma general de ello se presenta acontinuación:

```
tipo array[tamaño1]...[tamaño N] = { lista_valores };
```

donde **lista_valores** es una lista de constantes separadas por comas cuyo tipo es compatible con el tipo base del array. La primera constante se coloca en la primera posición del array, la segunda constante en la segunda posición y así sucesivamente.

Un ejemplo de la inicialización de un array unidimensional sería el siguiente:

```
int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10 };
```

Significa que i[0] tendrá el valor de 1 e i[9] el valor de 10.

Los arrays multidimensionales se inicializan del mismo modo que los unidimensionales. Por ejemplo:

```
int cuads[10][2] = {
    1 ,   1,
    2 ,   4,
    3 ,   9,
    4 ,  16,
    5 ,  25,
    6 ,  36,
    7 ,  49,
    8 ,  64,
    9 ,  81,
   10, 100,
};
```

El array consta de 10 filas y 2 columnas. Elementos de este array pueden ser:

```
cuad[6][1] = 49      y
cuad[9][1] = 100
```

Los arrays de tipo carácter que contienen cadenas permiten una inicalización abreviada de la forma:

```
char array[tamaño] = "cadena";
```

Se puede inicializar un array de tipo carácter de dos formas:

```
char cad[5] = "hola";

char cad[5] = { 'h', 'o', 'l', 'a', '\0' };
```

Los dos casos producen el mismo resultado. Todas las cadenas en C deben terminar con un caracter nulo (\0), por lo que al dimensionar un array de caracteres se debe contemplar este aspecto. Cuando se utiliza una constante de cadena (como en el primer caso del ejemplo), el compilador automáticamente proporciona la terminación nula.

4.6 Array de Tamaño indeterminado

C puede calcular automáticamente las dimensiones de los arrays utilizando **arrays de tamaño indeterminado**. Si en una sentencia de inicialización de un array no se especifica el tamaño del array, el compilador de C automáticamente crea un array lo suficientemente grande para mantener todos los inicializadores presentes.

Por ejemplo:

```
char e[] = "Este es un ejemplo\n";
```

el compilador de C cuenta los caracteres del mensaje para determinar la dimensión del array. Esto podría llegar hacer útil cuando el programador piense en cambiar un determinado mensaje que esta asignado a un array.

El uso de inicializadores de arrays de tamaño indeterminado no está restringido a los arrays unidimensionales.

Cuando se utilizan inicializaciones de arrays multidimensionales indeterminados, se deben especificar todas las dimensiones excepto la de más a la izquierda, para permitir al compilador de C indexar el array adecuadamente. De este modo se pueden contruir tablas de longitudes variables y el compilador asignará automáticamente suficiente espacio para ellas.

Un ejemplo de arrays multidimensionales de tamaño indeterminado se presenta acontinuación:

```
int cuads[][2] = {
     1  ,   1,
     2  ,   4,
     3  ,   9,
     4  ,  16,
     5  ,  25,
     6  ,  36,
     7  ,  49,
     8  ,  64,
     9  ,  81,
     10 , 100
};
```

La ventaja de esta declaración sobre la versión de tamaño determinado, es que la tabla puede alargarse o acortarse sin cambiar las dimensiones del array.

4.7 Cadenas

El uso más común de los arrays unidimensionales es para representar cadenas de caracteres.

En C, una cadena se define como un array de caracteres que termina en un carácter nulo. El carácter nulo se especifica como '\0' y generalmente en todos los compiladores de C es un cero.

Por esta razón, para declarar un array de caracteres es necesario que sean de un carácter más que la cadena más larga que pueda contener. Por ejemplo si se declara un array que contenga diez caracteres se tendría que escribir:

```
char cadena[11];
```

Al declarar así el arreglo, se dejará sitio para el carácter nulo del final de la cadena.

Como ya lo mencionamos, C no define un tipo de datos de cadena, pero permite disponer de constantes de cadena. Una **Constante de Cadena** es una lista de caracteres encerrada entre dobles comillas, como es el caso de:

```
"hola"
"contante de cadena"
```

Aquí no es necesario añadir explícitamente el carácter nulo, ya que el compilador de C lo hace automáticamente.

Si se declara un array de caracteres de un tamaño determinado y la cadena que es asignada a este array es más larga, los caracteres en exceso se ignoran. En caso contrario de que la cadena asignada al array sea más corta que el tamaño del array de caracteres, el resto de los elementos del array son inicializados a valor nulo (\0).

4.8 Array de Cadenas de Caracteres

Un array de cadenas de caracteres en un array donde cada elemento es a su vez un array de caracteres. Dicho de otra forma, es un array de dos dimensiones de tipo char.

Por ejemplo:

```
char lista[10][60];
```

En el ejemplo se declara un array de 10 filas de 60 caracteres cada una.

El tamaño del índice izquierdo determina el número de cadenas y el tamaño del índice derecho especifica la longitud máxima de cada cadena.

4.9 Punteros

Un puntero es una variable que contiene una **dirección de memoria**. Esa dirección es la posición que tiene una determinada variable en la memoria de la computadora. Si una variable contiene la dirección de otra variable, entonces se dice que la primera apunta a la segunda.

Dicho en otras palabras, un puntero es una variable que representa la posición (más que el valor) de otro dato, tal como una variable o un elemento de un array.

Los punteros son usandos frecuentemente en el lenguaje C y tiene gran cantidad de aplicaciones, como el proporcionar los medios por los cuales las funciones pueden modificar sus argumentos de llamada, soportar rutinas de asignación dinámica de C, y mejorar la eficiencia de ciertas rutinas.

Los punteros están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales del array. Es más, proporcionan una forma conveniente para representar arrays multidimensionales, permitiendo que éstos sean reemplazados por un array de punteros de **menor dimensión**. Esta característica permite que una colección de cadenas de caracteres sean representadas con un solo array, incluso cuando las cadenas pueden tener distinta longitud.

Dentro de la memoria de la computadora cada dato almacenado ocupa una o más celdas contiguas de memoria (bytes adyacentes). El número de celdas de memoria requerida para almacenar un dato depende de su tipo. Por ejemplo, un **carácter** será normalmente almacenado en 1 byte (8 bits) de memoria; un **entero** usualmente necesita 2 bytes contiguos. Un número en **coma flotante** puede necesitar 4 bytes contiguos, y una cantidad en **doble precisión** puede requerir 8 bytes contiguos.

Las celdas adyacentes de memoria dentro de la computadora están enumeradas consecutivamente, desde el principio hasta el fin de la área de memoria. El número asociado con cada celda de memoria es conocido como la **dirección de la celda**. La mayoría de las computadoras usan el sistema de numeración hexadecimal para designar las direcciones de celdas de memoria consecutivas, aunque algunas computadoras usan el sistema octal de numeración.

En la figura 4.3 se muestra la forma en que las celdas adyacentes de memoria están enumeradas, así como el que una variable puede apuntar a otra.

Si se declara que "v" es una variable que representa un determinado dato, el compilador le asignará celdas de memoria de acuerdo a su tipo y se podrá acceder a ella, conociendo su localización (o dirección) de la primera celda de memoria que le fué asignada.

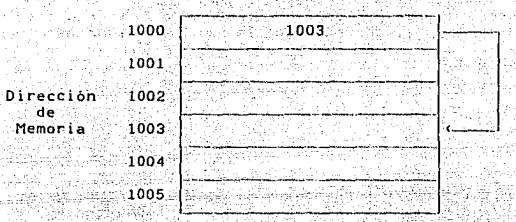


Figura 4.3

La dirección de memoria de la variable v puede ser determinada mediante la expresión &v, donde & es un operador monario, llamado el operador dirección, el cual proporciona la dirección del operando.

Para poder asignar la dirección de la variable v a otra variable, como por ejemplo pv, se tendría que escribir la siguiente sentencia de asignación:

```
pv = &v
```

Esta nueva variable (pv) es un puntero a v, desde su "posición" hasta el lugar donde v está almacenada en memoria.

Es importante remarcar que la variable pv representa la dirección de v y no su valor. De esta forma pv es referida como variable apuntadora.

En la figura 4.4 se muestra la forma en que se almacena la dirección de la variable v en la variable apuntadora pv.

Figura 4.4

El dato almacenado en las celdas de memoria de v puede ser accedido mediante la expresión *pv, donde el * es también un

operador monario, llamado operador indirección, que opera sólo sobre una variable puntero. Por tanto *pv y v representan el mismo dato que es el contenido de las mismas celdas de memoria.

Si se escribe:

```
pv = &v
```

y

```
u = *pv
```

La variables u y v representan el mismo valor. El valor de v puede ser indirectamente asignado a u. (Se asume que u y v están declaradas con el mismo tipo de dato.)

4.10 Declaración de un Puntero

Los punteros, como cualquier otra variable, deben ser declarados antes de ser usados dentro de un programa de C. Sin embargo, la interpretación de una declaración de puntero es un poco diferente de la declaración de otras variables.

Una declaración de puntero consiste en un tipo base, un * y el nombre de la variable. La forma general para declarar una variable puntero es:

```
tipo *nombre;
```

donde tipo es cualquier tipo base de C y nombre es el nombre de la variable puntero.

El tipo base del puntero define el tipo de variables a las que puede apuntar el puntero. Técnicamente, cualquier tipo de puntero puede apuntar a cualquier lugar de la memoria. Sin embargo, toda la aritmética de punteros está hecha en relación a su tipo base, por lo que es importante declarar correctamente el puntero.

4.11 Operadores de Punteros

Existen dos operadores especiales de punteros:

a) & Operador de dirección.

b) * Operador de indirección.

El & es un operador monario que devuelve la dirección de memoria de su operando, y el *, es el complemento de &. Es un operador monario que devuelve el valor de la variable localizada en la dirección que sigue.

El acto de utilizar un puntero se llame a menudo indirección,

puesto que se accede indirectamente a una posición de memoria mediante otra variable.

Las variables puntero deben apuntar siempre al tipo de datos correcto. Por ejemplo, cuando se declara un puntero a un tipo int, el compilador asume que cualquier dirección que mantenga apunta a una variable entera.

Debido a que C permite asignar cualquier dirección a una variable puntero, el fragmento de código que se presenta acontinuación, se puede compilar sin obtener mensajes de error (o sólo advertencias, dependiendo del tipo de compilador), pero no produce el resultado deseado

```
main()
{
 float x, y;
 int *p;

 p = &x;
 y = *p;
}
```

En el ejemplo, no se asignará el valor de x a y. Debido a que p se declara como un puntero a entero, por ello, sólo se transfieren 2 bytes de información a y, y no los 4 bytes que normalmente forman un número en coma flotante.

4.12 Expresiones de Punteros

En general las expresiones que involucran punteros se ajustan a las mismas reglas que cualquier otra expresión de C. Dentro de las expresiones de punteros se verán:

a) Las asignaciones de punteros.

b) Aritmética de punteros.

c) Comparación de punteros.

4.13 Asignaciones de Punteros

Como cualquier otra variable, un puntero puede utilizarse a la derecha de una declaración de asignación para asignar su valor a otro puntero. Por ejemplo:

```
main()
{
  int x;
  int *p1, *p2;
  p1 = &x;
  p2 = p1;
}
```

Tanto p1 como p2 apuntan a x.

También se puede asignar un valor a la dirección apuntada por un puntero. Por ejemplo, si suponemos que p es un puntero a un entero, lo siguiente asigna el valor de 101 a la dirección apuntada por p:

```
*p = 101;
```

Se puede incrementar o decrementar el valor de la dirección apuntada por un puntero mediante una sentencia como ésta:

```
(*p)++;
```

Aquí, los paréntesis son necesarios por que el operador * tiene menor nivel de precedencia que el operador ++.

4.14 Aritmética de Punteros

Existen sólo dos operaciones aritméticas que se pueden usar con punteros: la suma y la resta.

Para poder entender lo que pasa con la aritmética de punteros nos vamos a basar en un ejemplo. Si suponemos que p1 es un puntero a un entero con un valor actual de 2000 y que los enteros son representados por 2 bytes de longitud. Después de la expresión:

```
p1++;
```

p1 contendrá 2002, y no 2001, ya que al incrementarse p1 se apunta al siguiente entero. Lo mismo ocurre si se decrementara p1:

```
p1--;
```

esto hace que p1 tenga el valor 1998, si anteriormente tuviera el valor de 2000.

El porque de no tener los valores esperados es que cada vez que se incrementa un puntero, se apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa, se apunta a la posición del elemento anterior.

Con punteros a caracteres, esto frecuentemente hace que aparezca una aritmética "normal", porque en general los caracteres son de 1 byte de longitud. Sin embargo, el resto de los punteros aumentan o decrecen en la longitud del tipo de datos a los que apuntan.

Toda la aritmética de punteros se lleva a cabo de acuerdo con el tipo base del puntero, de forma que siempre se apunte a un elemento adecuado del tipo base.

La siguiente expresión:

```
p1 = p1 + 9;
```

hace que **p1** (siendo un puntero) apunte al noveno elemento del tipo **p1** que está más allá del elemento al que apunta actualmente.

Es importante mencionar, que no pueden realizarse otras operaciones aritméticas sobre los punteros, más allá de la suma y la resta de un puntero y un entero. En particular, no se pueden multiplicar, dividir, sumar o restar punteros entre si mismos; no se les puede aplicar el desplazamiento a nivel de bits y los operadores de máscara; y no se puede sumar o restar el tipo **float** o el tipo **double** a los punteros.

Ejemplos de la aritmética de punteros:

```
int *p;
p++;          /* Hace que p apunte al siguiente entero */
p--;          /* Hace que p apunte al entero anterior */
p = p + 3;    /* Se avanza tres enteros */
p = p - 3;    /* Se retorceden tres enteros */
```

4.15 Comparación de Punteros

En el lenguaje C se pueden comparar dos punteros en una expresión relacional. Comunmente la comparación de punteros se utiliza cuando dos o más punteros apunta a un objeto común, por ejemplo, si los punteros p1 y p2 apuntan a dos variables separadas y no relacionadas, entonces cualquier comparación entre ambos carece de sentido. Sin embargo, si los dos punteros apuntan a variables relacionadas, como por ejemplo los elementos de un mismo array, entonces p1 y p2 se pueden comparar con sentido.

La comparación de punteros se utiliza principalmente en rutinas de manejo de pilas, las cuales sirven para guardar y recuperar valores. Una pila es una lista que utiliza un acceso de "primero en entrar", "último en salir", además se usan frecuentemente en compiladores, intérpretes, hojas de cálculo y otros software de sistemas.

4.16 Punteros y Arrays

Es importante hacer notar que los array y los punteros sostienen una estrecha relación.

El nombre de un array es realmente un **puntero al primer elemento** de ese array. Por ejemplo, si **x** es un array unidimensional, entonces la dirección del primer elemento del array puede ser expresada tanto como **&x[0]** o simplemente como **x**, la dirección del segundo elemento se puede escribir tanto como **&x[1]** o como **(x+1)**, y así sucesivamente.

En general, la dirección del elemento (i+1) del array se puede expresar o bien como **&x[i]** o como **(x+i)**. Por tanto, tenemos dos modos diferentes de escribir la dirección de cualquier elemento de un array:

a) Escribiendo el elemento real del array precedido por un &.

b) Escribiendo una expresión en la cual el índice se añade al nombre del array.

Es importante entender que se está trabajando con un tipo muy especial e inusual de expresión.

Por ejemplo en la expresión:

(x+i)

Se esta especificando una dirección que está un cierto número de posiciones de memoria (i), más allá de la dirección del primer elemento del array **x**.

Por lo tanto la expresión **(x+i)** es una representación simbólica de una dirección en vez de una expresión aritmética.

Hay que recordar que el número de celdas de memoria asociadas con un elemento del array depende del tipo de dato del array, así como de la propia arquitectura de la computadora.

Sin embargo, cuando se escribe la dirección de un elemento del array de la forma **(x+i)**, el programador no tiene que preocuparse por el número de celdas de memoria asociadas con cada tipo de datos del array; el compilador lo ajusta automáticamente. El programador sólo debe indicar la dirección del primer elemento (con el nombre del array) y el número de elementos más allá del primero (valor del índice). El valor de i se denomina a veces **desplazamiento** (offset) cuando se usa de esta manera.

Si **&x[i]** y **(x+i)** representan la dirección del i-ésimo elemento de **x**, es razonable llegar a la conclusión que **x[i]** y ***(x+i)** representan el contenido de esa dirección, el **valor** del i-ésimo elemento de **x**.

Cuando se asigna un **valor** a un elemento de un array, la parte izquierda de la asignación puede ser escrita como **x[i]** o como ***(x+i)**. De esta forma, un valor puede ser asignado directamente a un elemento del array o al área de memoria cuya dirección es la del elemento del array.

En ciertas situaciones una variable puntero puede aparecer en la parte izquierda de una asignación para poder recibir la dirección de una variable. Pero no es posible asignar una dirección arbitraria a un nombre de array o a un elemento del mismo. Por lo que expresiones como **x**, **(x+i)** y **&x[i]** no pueden

aparecer en la parte izquierda de una sentencia de asignación.

Para poder entender un poco más el concepto de asignación, se muestra el siguiente ejemplo:

```
main()
{
  int linea[80];
  int *p1;

  /* Asignación de valores */

  linea[2] = linea[1];
  linea[2] = *(linea + 1);
  *(linea + 2) = linea[1];
  *(linea + 2) = *(linea + 1);

  /* Asignación de direcciones */

  p1 = &linea[1];
  p1 = linea + 1;
}
```

En cada una de las cuatro primeras sentencias se le asigna el valor del segundo elemento del array (**linea[1]**) al tercer elemento del array (**linea[2]**). Por lo tanto, las cuatro sentencias son equivalentes.

En una programación con mayor profesionalismo se elegiría la primera o la cuarta sentencia.

En las dos últimas sentencias, se asigna la **dirección** del segundo elemento del array al puntero **p1**.

Es importante hacer notar que la dirección de un elemento de un array no puede ser asignada a otro elemento del array. Como por ejemplo:

```
&linea[1] = &linea[2];
```

En caso de que se quisiera asignar el valor de un elemento de un array a otro mediante un puntero se tendría que escribir:

```
p1 = &linea[1];
linea[2] = *p1;
```

ó

```
p1 = linea + 1;
*(linea + 2) = *p1;
```

Como el nombre de un array es en realidad un puntero al primer elemento, debe ser posible definir un array como una variable puntero en vez de un array convencional.

Sintácticamente, las dos definiciones son equivalentes. Sin embargo, la definición convencional de un array produce la reserva de un bloque fijo de memoria al principio de la ejecución del programa, mientras que esto no ocurre si el array se representa en términos de una variable puntero.

Para solucionar este problema se necesitará de algún tipo de asignación inicial de memoria, antes de que los elementos del array sean procesados. Generalmente, tales tipos de reserva de memoria se realizan usando la función de biblioteca **malloc()**.

Si pensamos que un array unidimensional puede ser representado en términos de puntero (el nombre del array) y de un desplazamiento (el índice), es razonable llegar a la conclusión que los array multidimensionales también pueden ser representados con una notación equivalente de punteros.

Como un array bidimensional es en realidad una colección de array unidimensionales, podemos definirlo como un puntero a un grupo contiguo de array unidimensionales.

Una declaración de un array bidimensional puede ser escrita como:

```
tipo (*ptvar)[expresión_2];
```

en vez de

```
tipo array[expresión_1][expresión_2];
```

y este mismo concepto puede generalizase para un array multidimensional:

```
tipo (*ptvar)[expresión_2][expresión_3]...[expresión_N];
```

que reemplaza a

```
tipo array[expresión_1][expresión_2]...[expresión_N];
```

En todas estas declaraciones **tipo** es el tipo de datos del array, **ptvar** es el nombre de la variable puntero, **array** es el nombre del array y **expresión_1**, **expresión_2**, ..., **expresión_N** son expresiones enteras positivas que indican el máximo número de elementos del array con cada índice.

Hay que hacer notar que los paréntesis que rodean el nombre del array, y el asterisco que lo precede en la versión de puntero de

cada declaración deben estar presentes. Ya que sin ellos se estaría definiendo un array de punteros en vez de un puntero a un grupo de arrays.

Si suponemos que x es un array bidimensional de enteros de 10 filas y 20 columnas, se podría declarar a x de la siguiente forma:

```
int (*x)[20];
```

en vez de

```
int x[10][20];
```

En la primera declaración **x** se define como un puntero a un grupo contiguo de array unidimensionales de 20 elementos enteros.

Así **x** apunta al primero de los array de 20 elementos, que es en realidad la primera fila (fila 0) del array bidimensional. Similarmente (x+1) apunta al segundo array de 20 elementos, que es la segunda fila del array bidimensional, y así sucesivamente, tal y como se ilustra en la figura 4.5.

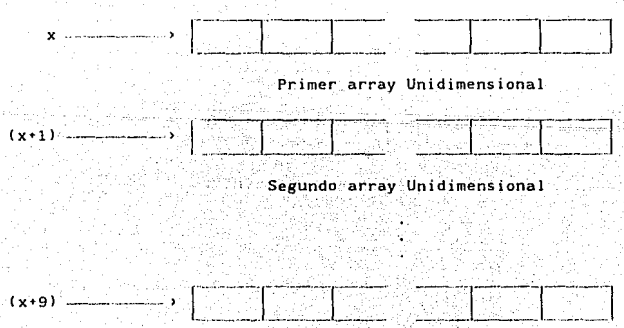


Figura 4.5

Ahora si se tuviera un array tridimensional t de números en coma flotante de 10 x 20 x 30, se podría definir como sigue:

```
float (*t)[20][30];
```

en vez de

```
float t[10][20][30];
```

En la primera declaración t se define como un puntero a un grupo contiguo de array bidimensionales de coma flotante de 20x30, y además se apunta al primer array, la expresión (t+1) apunta al segundo, y así sucesivamente.

Un elemento individual del un array puede ser accedido mediante el uso repetido del operador indirección. Sin embargo, normalmente este método es más difícil que el convencional para acceder a un elemento del array.

Por ejemplo, si se tiene un array bidimensional x de 10 filas y 20 columnas, el elemento de la fila 2, columna 5, puede ser accedido de las dos siguientes formas:

```
x[2][5]
```

```
*(*(x + 2) + 5)
```

En la segunda forma, (x + 2) es un puntero a la fila 2. Por tanto, el objeto de este puntero, *(x + 2), referirá a toda la fila. Como la fila 2 es un array unidimensional, *(x + 2), es realmente un puntero al primer elemento de la fila 2, y si se suma 5 a este puntero de la forma (*(x + 2) + 5) se indicará que es un puntero al elemento 5 (sexto elemento) de la fila 2. Por último, el objeto de este puntero, *(*(x + 2) + 5), se referirá al elemento de la columna 5 de la fila 2, (x[2][5]). En la figura 4.6 se presenta la forma en la cual se accesa a este elemento.

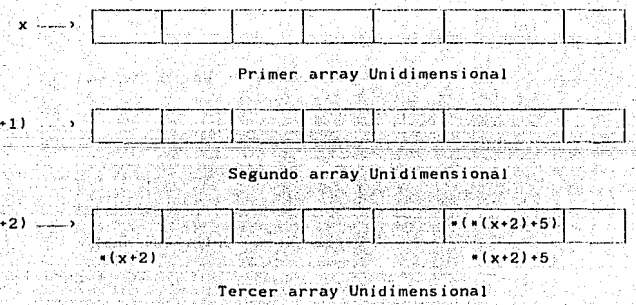


Figura 4.6

4.17 Array de Punteros

Un array muldimensional puede ser expresado como un array de punteros en vez de un puntero a un grupo contiguo de arrays. En estos casos el nuevo array será de una dimensión menor que el array multidimensional. Cada puntero indicará el principio de un array de dimensión (n - 1).

En términos generales, un array bidimensional puede ser definido como un array unidimensional de punteros escribiendo;

```
tipo *array[expresión_1];
```

en vez de la definición convencional del array:

```
tipo array[expresión_1][expresión_2];
```

En una forma general, un array n dimensional puede ser definido como un array de punteros de dimensión (n - 1) de la siguiente forma:

```
tipo *array[expresión_1][expresión_2] . . . [expresión_N-1];
```

en vez de:

```
tipo array[expresión_1][expresión_2] . . . [expresión_N];
```

En esta declaración **tipo** se refiere al tipo de datos del array **n** dimensional original, **array** es el nombre del array y **expresión_1**, **expresión_2**, ..., **expresión_N** son expresiones enteras positivas que indican el máximo número de elementos asociados con cada Indice.

Es importante hacer notar que el nombre del **array** precedido por un asterisco **no está encerrado entre paréntesis** en este tipo de declaración. Así la regla de precedencia de derecha a izquierda asocia primero el par de paréntesis cuadrados con **array**, definiéndolo como un array. El asterisco que lo precede establece que el arreglo contendrá punteros.

La **última expresión** (la de más a la derecha) se omite cuando se define un array de punteros , mientras que la primera **expresión** (la más a la izquierda) se omite cuando se define un puntero a un grupo de arrays.

Si se tiene un array bidimensional **x**, de 10 filas y 20 columnas, lo podemos definir como un array unidimensional de

punteros, como se muestra a continuación:

```
int *x[10];
```

Aquí x[0] apunta al primer elemento de la primera fila, x[1] al primer elemento de la segunda fila, y así sucesivamente. Es importante hacer notar que el número de elementos dentro de cada fila no está especificado explícitamente.

Un elemento individual del array, tal como x[2][5], puede ser accedido de la siguiente forma:

```
*(x[2] + 5);
```

En esta expresión x[2] es un puntero al primer elemento en la fila 2, de modo que (x[2] + 5) apunta al elemento 5 (que es en realidad el sexto elemento) de la fila 2. El objeto de este puntero, *(x[2] + 5), refiere por tanto a x[2][5]. En la figura 4.7 se muestra este ejemplo.

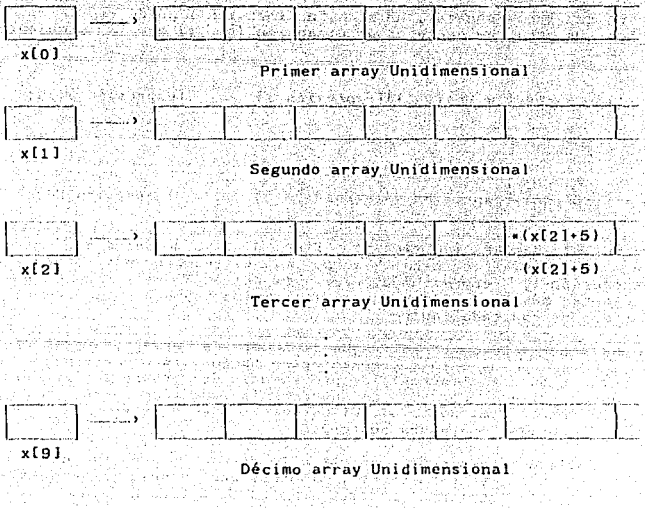


Figura 4.7

Ahora, si se tiene un array tridimensional t de coma flotante de dimensión 10 x 20 x 30, se podría expresar como un array bidimensional de punteros, tal como:

```
float *t[10][20];
```

Por tanto, se tendrán 200 punteros (10 filas, 20 columnas), cada uno apuntando a un array unidimensional.

Si quisiéramos acceder al elemento del array t[2][3][5], se tendría que escribiría lo siguiente:

```
*(t[2][3] + 5)
```

En esta expresión t[2][3] es un puntero al primer elemento en el array representado por t[2][3]. De aquí (t[2][3] + 5) apunta al elemento 5 (el sexto elemento) dentro de este array. Este puntero, *(t[2][3] + 5), por tanto representa t[2][3][5].

Los array de punteros ofrecen un método particularmente conveniente para almacenar cadenas. En esta situación cada elemento del array es un puntero que indica donde empieza cada cadena. Así, un array de n elementos puede apuntar a n cadenas diferentes y cada cadena puede ser accedida refiriendo su puntero correspondiente.

Por ejemplo las siguientes cadenas:

Pacifico	Negro
Atlantico	Rojo
Indico	Norte
Caribe	Baltico
Bering	Caspio

se podrían guardar en el siguiente array bidimensional de caracteres:

```
char mar[10][12];
```

El array mar tiene 10 filas para las 10 cadenas. Cada fila debe ser lo suficientemente grande para almacenar por lo menos 10 caracteres, ya que por ejemplo la cadena Atlantico tiene nueve letras y además el carácter nulo (\0) al final. De cualquier modo se da la opción de tener 12 caracteres, para tener la posibilidad de cadenas más largas.

Una forma mejor de hacer esto es definir un array de 10 punteros:

```
char *mar[10];
```

De esta manera, mar[0] apuntará a Pacifico, mar[1] a Atlantico, y así sucesivamente. No es necesario incluir el máximo tamaño de cadena dentro de la declaración del array. Sin embargo, se puede reservar una cantidad específica de memoria para cada cadena,

mediante la función de biblioteca **malloc**:

```
mar[i] = malloc(12 * sizeof(char));
```

Una cadena puede ser accedida refiriendo el correspondiente puntero, así un elemento de la cadena puede ser accedido mediante el uso del operador indirección. Por ejemplo, *(*(mar+2)+3) es el cuarto carácter (carácter número 3) en la tercera cadena (fila número 2) del array mar.

Si los elementos de un array son punteros a cadenas, se le puede asignar un conjunto de valores iniciales como parte de la declaración del array. En tales casos, los valores iniciales serán cadenas, donde cada una corresponde a un elemento distinto del array. Es importante recordar que un array debe ser especificado como **static** si es inicializado dentro de una función.

Una ventaja de este método es que no tenemos que reservar un bloque fijo de memoria por adelantado, como se hace cuando se inicializa un array convencional. Así si la declaración inicial tiene muchas cadenas y algunas de ellas son relativamente cortas, se puede ahorrar sustancialmente el uso de memoria. Además, si algunas de las cadenas son especialmente largas, no hay que preocuparse de exceder la longitud máxima de cadena (el máximo número de caracteres por fila). Los array de este tipo se refieren a menudo como array irregulares (**ragged array**).

La siguiente declaración de array aparece dentro de una función:

```
static char *mar[10] = {
                        "Pacifico",
                        "Atlantico",
                        "Indico",
                        "Caribe",
                        "Bering",
                        "Negro",
                        "Rojo",
                        "Norte",
                        "Baltico",
                        "Caspio",
                        };
```

En este ejemplo **mar** es un array de 10 punteros. El primer elemento del array (el primer puntero) apuntará a Pacifico, el segundo a Atlantico, y así sucesivamente.

El array se declara como **static** para poder ser inicalizado dentro de la función. Si la declaración del array fuera externa a todas las funciones, el indicador de tipo de almacenamiento **static** no sería necesario.

Como la declaración del array incluye valores iniciales, no es necesario incluir una designación explícita de tamaño dentro de

la declaración. El tamaño del array será automáticamente igual al número de cadenas presentes. Como resultado, la declaración anterior puede ser escrita como:

```
static char *mar[] = {
                        "Pacifico",
                        "Atlantico",
                        "Indico",
                        "Caribe",
                        "Bering",
                        "Negro",
                        "Rojo",
                        "Norte",
                        "Baltico",
                        "Caspio",
                      };
```

Es importante tener presente que el concepto de **arrays irregulares** sólo se refiere a la **inicialización** de arrays de cadenas y no a la asignación de cadenas que se lean mediante la función **scanf**.

4.18 Paso de Punteros a una Función

A menudo los punteros son pasados a las funciones como argumentos. Esto permite que datos de la porción del programa en la que se llama a la función sean accedidos por la función, alterados dentro de ella y luego devueltos al programa de forma alterada. Nos referimos a este uso de los punteros, como el de pasar argumentos por referencia (o por dirección o por posición), en contraste con pasar argumentos por **valor**.

Cuando un argumento es pasado por valor, el dato es **copiado** a la función. Así, cualquier alteración hecha al dato dentro de la función, no es devuelta a la rutina llamante. Sin embargo cuando un argumento es pasado por referencia (cuando un puntero es pasado a una función), la dirección del dato es pasada a la función.

El contenido de esta dirección puede ser accedido libremente, tanto dentro de la función como dentro de la rutina de llamada. Además cualquier cambio que se realice al dato (al contenido de la dirección) será reconocido tanto en la función como en la rutina de llamada. Así el uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente dentro de la función.

Cuando los punteros son usados como argumentos de una función, es necesario tener cuidado con la declaración de los argumentos formales dentro de la función. Los argumentos formales que sean punteros, deben ir precedidos por un asterisco. También si una declaración de función está incluida en la parte llamante del programa, el tipo de dato de cada argumento que corresponda a un puntero debe ir seguido por un asterisco.

Acontinuación se presenta un ejemplo que muestra la diferencia que existe entre **argumentos ordinarios**, que son pasados por **valor** y **argumentos puntero** que son pasados por **referencia**.

```
main()
{
   int u = 1;
   int v = 3;
   void func1(int u, int v);      /* Declaración de Función */
   void func2(int *pu, int *pv);  /* Declaración de Función */

   func1(u,v);
   func2(&u,&v);
}


void func1(int u, int v)
{
   u = 0;
   v = 0;
   return;
}


void func2(int *pu, int *pv)
{
   *pu = 0;
   *pv = 0;
   return;
}
```

Este programa contiene dos funciones llamadas **func1** y **func2**. La primera recibe dos variables enteras como argumentos. Estas variables tiene originalmente asignados los valores 1 y 3 respectivamente. En la función los valores son puestos a cero, sin embargo, estos valores no son reconocidos en la función **main**, porque los argumentos fueron pasados por valor y cualquier cambio sobre ellos es estrictamente local a la función en la cual se han producido los cambios.

En cambio en la función **func2** se reciben dos punteros a variables enteras como argumentos. Los argumentos son identificados como punteros por los operadores de indirección (*) que aparecen en la declaración de los argumentos. Además, la declaración de argumentos indica que los punteros representan direcciones de cantidades enteras.

Dentro de la función **func2**, los contenidos de las direcciones apuntadas, son reasignados con el valor 0. Como las direcciones son reconocidas tanto en **main** como en **func2**, los valores reasignados serán reconocidos dentro de **main** después de la llamada a **func2**. Por tanto, las variables enteras **u** y **v** habrán cambiado su valor de 1, 3 a 0, 0.

El ejemplo anterior contiene características adicionales que deben ser remarcadas, como es la forma con la cual **func2** es declarada dentro del **main**.

```
void func2(int *pu, int *pv);
```

En esta declaración, los elementos entre paréntesis identifican los argumentos como **punteros a cantidades enteras**. Las variables puntero **pu** y **pv** no han sido declaradas en ninguna parte dentro de **main**. Esto está permitido ya que son argumentos ficticios en vez de argumentos actuales.

Otra característica es la propia declaración de los argumentos formales dentro de la primera línea de **func2**:

```
void func2(int *pu, int *pv)
```

Los argumentos formales **pu** y **pv** son consistentes con los argumentos ficticios del prototipo de la función. En este ejemplo, los nombres de la variables correspondientes son los mismos, pero en general esto no es necesario.

Y finalmente, es importante hacer notar la forma en que **u** y **v** son accedidos dentro de la función **func2**:

```
*pu = 0;
*pv = 0;
```

Así, **u** y **v** son accedidos indirectamente, referenciando el contenido de las direcciones contenidas en los punteros **pu** y **pv**. Esto es necesario ya que las variables **u** y **v** no son conocidas como tales dentro de **func2**.

Como ya lo hemos mencionado el **nombre de un array** es un puntero al array que contiene la dirección del primer elemento. Por tanto, **el nombre de un array se trata como un puntero cuando se pasa a una función**.

Un array que aparece como argumento formal en una definición de una función puede ser declarado como puntero o como array de tamaño no especificado. La elección es cuestión de preferencia personal, pero a menudo vendrá determinada por la forma en la cual los elementos individuales del array sean accedidos dentro de la función.

Se puede pasar una porción de un array en vez de un array completo a un función. Para hacerlo, la dirección del primer elemento a ser pasado ha de ser especificada como argumento. El resto del array, empezando por el elemento especificado, será

entonces pasado a la función. Como ejemplo se presenta el siguiente programa:

```
main()
{
  float z[100];
  void proceso(float z[]);

    . . .

  /* Bloque donde se introducen valores para
          los elementos de z */

    . . .

   proceso(&z[50]);

    . . .
}

void proceso(float f[])
{
    . . .

    /* Proceso de los elementos de f */

    . . .

    return;
}
```

Hay que hacer notar que **z** se declara dentro de **main** como un array de 100 elementos de coma flotante. Después de que se introducen los elementos de **z** en la computadora, se pasa la dirección de **z[50]** (**&z[50]**) a la función **proceso**. Así los últimos 50 elementos de **z** (los elementos de z[50] a z[99]) estarán disponibles para la función **proceso**.

La dirección de **z[50]** se puede escribir también de la forma **z+50**. Por tanto, la llamada a la función **proceso** puede hacerse de la siguiente forma:

```
proceso(z+50);
```

Dentro de la función **proceso**, el array correspondiente es referido como **f**. Este array se declara como de coma flotante pero de tamaño no especificado. De esta manera, el hecho de que la función reciba sólo una parte de **z** es indiferente; si todos los elementos de **z** son modificados dentro de **proceso**, sólo los 50

últimos serán afectados dentro del main.

Si se deseara declarar el argumento formal f como un puntero en vez de un array dentro de la función proceso, se podría escribir lo siguiente:

```
void proceso(float *f)
{
   . . .

   /* Proceso de los elementos de f */

   . . .

   return;
}
```

De cualquier modo ambas declaraciones son válidas.

4.19 Vuelta de Punteros de una Función

Una función también puede devolver un puntero a la parte llamante del programa. Para hacer esto, la definición de la función y cualquier declaración de la misma debe indicar que la función devolverá un puntero. Esto se realiza precediendo la función con un asterisco.

A continuación se presenta el esqueleto de la estructura de un programa que transfiere un array de elementos de doble precisión a una función y devuelve un puntero a uno de los elementos del array:

```
main()
{
  double z[100];              /* Declaración de un Array */
  double *pz;                 /* Declaración de un Puntero */
  double *scan(double z[]);   /* Declaración de una Función */

  . . .

  /* Introducir valores para elementos de z */

  . . .

  pz = scan(z);

  . . .

}
```

```
double  *scan(double f[])
{
   double *pf;                  /*  Declaración de Puntero */

   . . . .

   /* Proceso de elementos de f */

   pf . . .;

   return(pf);
}
```

Dentro de main, z está declarada como un array de 100 elementos de doble precisión y pz es un puntero a una cantidad en doble precisión. También se declara la función scan; la cual acepta un array de elementos de doble precisión como argumento y devolverá un puntero (la dirección) a una cantidad en doble precisión. El asterisco precediendo al nombre de la función indica que la función devuelve un puntero.

Dentro de la definición de la función, la primera línea indica que scan acepta un parámetro formal (f[]) y devuelve un puntero a una cantidad en doble precisión. El parámetro formal será un array unidimensional de elementos en doble precisión. El boceto sugiere que la dirección de uno de los elementos del array se asigna a pf durante o después del proceso de los elementos del array. Esta dirección es devuelta a main donde se asigna a la variable puntero pz.

4.20 Punteros a Funciones

Una característica muy importante en el lenguaje C son los punteros a una función.

Una función no es una variable, pero tiene una posición física en memoria que puede asignarse a un puntero. La dirección de la función es el punto de entrada de la misma. Por ello, se puede usar un puntero a una función para llamarla posteriormente.

Cuando se compila una función, el código fuente se transforma en código objeto y se establece un punto de entrada. Cuando se llama a la función mientras se ejecuta el programa, se hace una llamada en lenguaje máquina a ese punto de entrada. Por tanto, si un puntero contiene la dirección del punto de entrada a la función, puede ser utilizado posteriormente para llamar a la función.

La dirección de la función se obtiene utilizando el nombre de la función sin paréntesis ni argumentos.

4.21 Paso de Funciones a otras Funciones

Cuando una declaración de función aparece dentro de otra, el nombre de la función declarada convierte en un puntero a esa función. Por ejemplo, si la función proceso se declara dentro de main, entonces proceso se interpretará como una variable puntero dentro de main. Tales punteros pueden pasarse a otras funciones como argumentos, y ésto tendrá el efecto de pasarle una función a otra, como si la primera función fuera una variable.

Cuando una función acepta el nombre de otra como argumento, la declaración formal debe identificar este argumento como un puntero a otra función. En su forma más simple, un argumento formal que es un puntero a función puede declararse como sigue:

```
tipo_dato (*nombre_función)();
```

donde **tipo_dato** es el tipo de la cantidad devuelta por la función. Esta función puede accederse mediante el operador indirección. Para ello, el operador indirección debe preceder al nombre de la función (el argumento formal), y además deben estar encerrados entre paréntesis, por ejemplo:

```
(*nombre_función)(arg_1, arg_2, ..., arg_n);
```

donde **arg_1, arg_2, ..., arg_n** se refiere a los argumentos requeridos en la llamada a la función.

Para aclarar más estos conceptos se presenta el siguiente **programa ejemplo** que consta de cuatro funciones: main, proceso, func1 y func2. Cada una de ellas retorna un valor entero:

```
/* Programa Ejemplo */

main()
{
  int i, j;
  int proceso();          /* Declaración de Funciónes */
  int func1();
  int func2();

  . . . .

  i = proceso(func1);     /* Se pasa func1 a proceso; retorna un
                             valor para i  */
  . . . .

  j = proceso(func2);     /* Se pasa func2 a proceso; retorna un
                             valor para j  */
  . . . .
}
```

```
proceso(pf)                    /* Definición de Función */
int *pf();                     /* Declaración de Argumento Formal
{                                 (El argumento formal es un puntero a
   int a, b, c;                   una función) */

   . . . .

   c = (*pf)(a,b);             /* Acceso a la función pasada a esta
                                  función; retorna una valor apara c */

   . . . .

   return(c);
}


func1(a,b)                     /* Definición de Función */
int a,b;
{
  int c;

  c = . . . ;                  /* Se usa a y b para evaluar c */

  return(c);
}


func2(x,y)                     /* Definición de Función */
int x,y;
{
  int z;

  z = . . . ;                  /* Se usa x e y para evaluar z */

  return(z);
}
```

La función principal **main** llama a la función **proceso** dos veces. En la primera llamada pasa **func1** a proceso, miestras que en la segunda pasa a **func2**. En cada llamada retorna un valor entero que es asignado a las variables i y j.

En la definición de **proceso**, la función tiene un parámetro formal, **pf**, que es declarado como un puntero a función. La función a la que apunta **pf** devuelve un valor entero.

Ya dentro de la función **proceso** se llama a la función a la que apunta **pf**, se le pasan dos cantidades enteras (a y b) como argumentos y se devuelve un entero que es asignado a la variable c.

El resto de las definiciones de funciones son normales (**func1** y **func2**). Estas son las funciones pasadas desde **main** hasta **proceso**. Los valores resultantes se asume que se obtienen de los

argumentos, aunque los detalles de los cálculos no estén presentes en el programa.

En la mayoría de los compiladores se permite, por lo menos incluir, el tipo de datos de los argumentos. En tal caso una función que acepta un puntero a otra función como argumento puede declararse como sigue:

```
tipo_func nombre_func(tipo_arg (*)(tipo_1, tipo_2,...,tipo_n);
                      └──────────────────────────────────┘
                      Puntero a una función pasada como argumento
```

donde **tipo_func** se refiere al tipo de dato devuelto por la función, **nombre_func** es el nombre de la función, **tipo_arg** es el tipo de dato devuelto por la función argumento y **tipo_1**, **tipo_2**, función.

El operador indirección aparece entre paréntesis para indicar que es un puntero a la función argumento, los argumentos de esta función aparecen encerrados entre paréntesis.

Cuando se usa el prototipo completo, la declaración se expande como sigue:

```
tipo_func nombre_func(tipo_arg (*ptr)(tipo_1 arg_1,
                      tipo_2 arg_2,...,tipo_n arg_n);
```

La notación es la misma que la primera declaración, excepto que **ptr** se refiere a la variable que apunta a la **función argumento** y **tipo_1 arg_1**, **tipo_2 arg_2**, ...**tipo_n arg_n** se refiere a los tipos y nombres de los argumentos de esa función.

Se presenta ahora una nueva versión del programa **ejemplo** utilizando el prototipo completo de la función:

```
/* Programa Ejemplo */

main()
{
  int i, j;
  int proceso(int (*pf)(int a, int b));
  int func1(int a, int b);
  int func2(int x, int y);

  . . .

  i = proceso(func1);
```

```
  . . . .
  j = proceso(func2);
  . . . .
}

proceso(int (*pf)(int a,int b))
{
   int a, b, c;
   . . . .
   c  = (*pf)(a,b);
   . . . .
   return(c);
}

func1(int a,int b)
{
  int c;
  c = . . . . ;
  return(c);
}

func2(int x,int y)
{
  int z;
  z = . . . . ;
  return(z);
}
```

Las declaraciones de funciones dentro de **main** incluyen los nombres de los argumentos y sus tipos. Además, la declaración de **proceso** incluye el nombre de la variable pf que apunta a la función pasada a proceso.

En cada función subordinada, la declaración de argumentos formales se combina con la primera línea de la definición de la función. Es importante hacer notar que la declaración del argumento formal pf dentro de **proceso** es consistente con la declaración de proceso que aparece en **main**.

Algunas aplicaciones pueden programarse fácilmente pasando una función a otra. Por ejemplo, una función puede representar una

ecuación matemática, y otra puede contener la estrategia de resolución. Esto es particularmente útil si el programa contiene diferentes ecuaciones matemáticas, de las cuales el usuario selecciona una cada vez que se ejecuta el programa.

4.22 Aspectos Importantes en el Manejo de Punteros

El manejo de los punteros puede ser muy complicado, por lo que es necesario tener mucho cuidado con ellos, sobre todo en su interpretación. Esto es muy común con declaraciones que involucren funciones y arrays.

Un problema puede ser el uso de los paréntesis, ya que comunmente se usan para indicar funciones y anidaciones (para establecer precedencias) dentro de declaraciones más complejas. Así la declaración:

```
int *p(int a);
```

indica a una función que acepta un argumento entero y devuelve un puntero a entero. Pero si la declaración fuera de esta forma:

```
int (*p)(int a);
```

se indicaría un puntero a una función que acepta un argumento entero y retorna un entero. En esta declaración, los primeros paréntesis se usan para anidar y los segundos para indicar una función.

La interpretación de declaraciones puede hacerse muy compleja, por ejemplo:

```
int * (*p)(int (*a)[]);
```

En esta declaración (*p)(...) indica un **puntero a función**, y int (*a)[] indica un **puntero a un array de enteros**, uniendo estas dos partes se tiene un puntero a una función cuyo argumento es un puntero a un array de enteros. Pero falta agregar **int ***, lo que dá como resultado que la declaración completa se interprete como un puntero a una función que acepta un puntero a un array de enteros como argumento y devuelve un puntero a entero.

Siempre hay que recordar que un paréntesis izquierdo siguiendo inmediatamente a un identificador representa una función. De igual forma un paréntesis cuadrado izquierdo siguiendo inmediatamente a un identificador representa un array.

Los paréntesis que indican función y los paréntesis cuadrados que indican array tiene mayor precedencia que el operador

indirección (*), por lo que serán necesario paréntesis adicionales cuando se declare un puntero a una función o a un array.

Las siguientes declaraciones muestran las diferentes formas en las que pueden ser interpretadas:

```
int *p;                 /* p es un puntero a un entero. */

int *p[10];             /* p es un array de 10 punteros a
                           enteros. */

int (*p)[10];           /* p es un puntero a un array de 10
                           enteros. */

int *p(void);           /* p es una función que devuelve un
                           puntero a entero. */

int p(char *a);         /* p es una función que acepta un
                           argumento que es un puntero a carácter,
                           y devuelve un entero. */

int *p(char *a);        /* p es una función que acepta un
                           argumento que es un puntero a carácter,
                           y devuelve un puntero a entero. */

int (*p(char *a))[10];  /* p es una función que acepta un
                           argumento que es un puntero a carácter, y
                           devuelve un puntero a un array de 10
                           enteros. */

int *p(char a[]);       /* p es una función que acepta un
                           argumento que es un array de caracteres,
                           y devuelve un puntero a entero. */

int *p(char (*a)[]);    /* p es una función que acepta un
                           argumento que es un puntero a un array de
                           caracteres, y devuelve un puntero a
                           entero. */

int *p(char *a[]);      /* p es una función que acepta un
                           argumento que es un puntero a un array de
                           punteros a caracteres, y devuelve un
                           puntero a entero.

int *(*p)(char (*a)[]); /* p es un puntero a una función que
                           acepta un argumento que es un puntero a
                           un array de punteros a caracteres, y
                           devuelve un puntero a entero. */

int *(*p)(char *a[]);   /* p es un puntero a una función que
                           acepta un argumento que es un array de
                           punteros a caracteres, y devuelve un
                           puntero a entero. */
```

```
int (*p[10])(void);        /* p es un array de 10 punteros a
                           función; cada función devuelve un
                           entero. */

int *(*p[10])(char a);     /* p es un array de 10 punteros a
                           función; cada función acepta un argumento
                           que es un carácter, y devuelve un puntero
                           a entero. */

int *(*p[10])(char *a);    /* p es un array de 10 punteros a
                           función; cada función acepta un argumento
                           que es un puntero a carácter, y devuelve
                           un puntero a entero. */
```

CAPITULO V

FUNCIONES

NO

EXISTE

PAGINA

F U N C I O N E S

Las funciones son los bloques constructores de C y el lugar donde se dá toda la actividad del programa. Son una de las características más importantes del lenguaje C.

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo program de C consta al menos de una función, la denominada **main**. Además de ésta, puede haber otras funciones cuya finalidad es fundamentalmente, descomponer el problema general en subproblemas más fáciles de resolver y de mantener. La ejecución de un programa siempre comienza por la función **main**.

Cuando se llama a una función, el control se pasa a la misma para su ejecución y cuando ésta finaliza, el control es devuelto de nuevo al módulo que la llamó para continuar con la ejecución del mismo y a partir de la sentencia que efectúo la llamada. En la figura 5.1 se muestra un ejemplo de ello.

5.1 Definición de una Función

La definición de una función consta de la cabecera de la función y el cuerpo de la función.

La forma general de una función es:

```
clase tipo nombre_función(parámetros_formales)
  {
     cuerpo de la función
  }
```

donde **clase** define el ámbito de la función, el **tipo** indica el tipo del valor devuelto por la función, en caso de que no se especifique cualquier tipo, el compilador asume que la función devuelve como resultado un entero, y los **parámetros formales** es una secuencia de declaraciones de parámetros separados por comas y encerrados entre paréntesis.

Los **parámetros formales** son variables con sus tipos asociados que reciben los valores pasados en la llamada a la función.

Una función puede no tener parámetros, en cuyo caso la lista de parámetros estará vacía. Sin embargo, siempre se requerirá la presencia de los paréntesis. Para algunos compiladores es necesario que la palabra clave **void** aparezca entre ellos.

En las declaraciones de variables se pueden declarar múltiples variables del mismo tipo mediante una lista con los nombres de las variables separadas por comas. Pero en las funciones todos los parámetros deben incluir tanto el tipo como el nombre de la

FUNCIONES

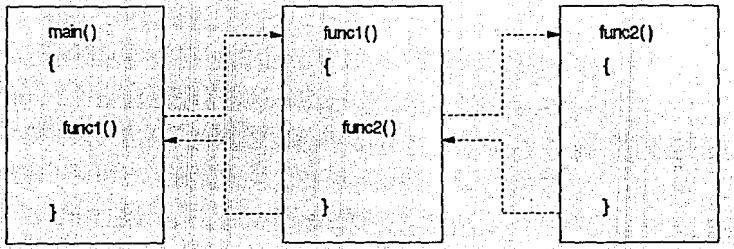


Figura 5.1

variable. Es decir, la lista de declaración de parámetros de una función tiene la siguiente forma general:

```
func(tipo var1, tipo var2, ... tipo var n)
```

El cuerpo de la función está formado por una sentencia compuesta que a su vez contiene sentencias individuales que definen lo que hace la función, y además puede haber declaraciones de variables que son usadas dentro de la función.

5.2 Reglas de Ambito de las Funciones

Las **reglas de ámbito** de un lenguaje son las reglas que controlan si un fragmento de código conoce o tiene acceso a otro fragmento de código o de datos.

En C, cada función es un bloque de código discreto. El código de una función es privado a esa función y no se puede acceder a él mediante una expresión (como por ejemplo **goto**) de otra función, a menos que se haga a través de una llamada a esa función.

El código que comprende el cuerpo de una función está oculto al resto del programa y, a no ser que se usen datos o variables **globales**, no puede ser afectado por otras partes del programa ni el afectarlas. Dicho de otro modo, el código y los datos que están definidos dentro de una función no pueden interactuar con el código o los datos definidos dentro de otra función porque las dos funciones tienen un ámbito diferente.

Las variables que están definidas dentro de una función son conocidas como **variables locales**. Una variable local, comienza a existir cuando se entra en la función y se destruye al salir de ella. Así las variables locales no pueden conservar sus valores entre distintas llamadas a la función. La única excepción a esta regla se dá cuando la variable se declara con el especificador de clase de almacenamiento **static**. Esto hace que el compilador trate a la variable como si fuese una variable global, en cuanto almacenamiento se refiere, pero sige limitando su ámbito al interior de la función.

En C, todas las funciones están al mismo nivel de ámbito. Es decir, no se puede **definir** una función dentro de otra función.

5.3 Acceso a una Función

Se puede acceder o llamar a una función especificando su nombre, seguido de una lista de argumentos encerrados entre paréntesis y separados por comas.

La llamada a la función puede aparecer sola como una expresión

simple, o puede ser uno de los operandos de una expresión más compleja.

La llamada a una función puede tener la siguiente forma:

```
[variable =] func(argumentos_actuales);
```

donde **variable** especifica la variables donde va a ser almacenado el valor devuelto por la función, si es que se devuelve algún valor; esto es opcional. **func** es un identificador que corresponde al nombre de la función.

Los argumentos que aparecen en la llamada a la función se denominan **argumentos actuales**, mientras que los **argumentos formales** son los que aparecen en la primera línea de definición de la función.

Otra forma de llamar a los argumentos actuales, es simplemente con el nombre **argumentos**, ó también con el nombre de parámetros actuales.

En una llamada normal a una función, habrá un argumento actual por cada argumento formal. Los argumentos actuales pueden ser constantes, variables simples, o expresiones más complejas. No obstante, cada argumento actual debe ser del mismo tipo de datos que el argumento formal correspondiente.

5.4 Argumentos de Funciones

Si una función va a usar argumentos, se deben declarar variables que acepten los valores de los argumentos. Estas variables son las que reciben el nombre de **parámetros formales** de la función.

Estas variables se comportan como otras variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir.

Se debe estar seguro de que los parámetros formales son del mismo tipo que los argumentos usados para llamar a la función (los parámetros actuales). Si hay un error en los tipos, el compilador no mostrará un mensaje de error, pero se obtendrán resultados inesperados.

A diferencia de muchos otros lenguajes, C generalmente hace algo con cualquier programa sintácticamente correcto, aún cuando el programa contenga posibles discordancias de tipos. Por ejemplo, si una función espera un puntero pero se le llama con un valor, pueden obtenerse resultados inesperados. El uso de **prototipos de funciones** puede ayudar a detectar este tipo de errores.

Al igual que con las variables locales, se pueden hacer asignaciones a los parámetros formales de una función o usarlos

en cualquier expresión válida de C. Aún cuando estas variables realizan la tarea especial de recibir el valor de los argumentos que se pasan a la función, se pueden usar como cualquier otra variable.

Existen dos métodos de pasar argumentos a las funciones:

a) LLamadas por Valor

b) LLamadas por Referencia

El primer método se denomina **llamada por valor**. En este método se copia el valor de un argumento en el parámetro formal de una subrutina o función. De esta forma, los cambios en los parámetros de la subrutina no afectan a las variables que se usan en la llamada.

La **llamada por referencia** es la segunda forma de pasar argumentos a una subrutina. En este método, se copia la **dirección** del argumento en el parámetro formal. Dentro de la subrutina se usa la dirección para acceder al argumento usado en la llamada. Esto significa que los cambios hechos a los parámetros afectan a la variable usada en la llamada a la subrutina.

A pesar de que el convenio de paso de argumentos en C es la llamada por valor, es posible simular una llamada por referencia pasando un puntero del argumento. Como esto hace que se pase la dirección del argumento a la función, es posible cambiar el valor del argumento exterior de la función.

Los punteros se pasan a las funciones como cualquier otro valor, pero como es natural, es necesario declarar los parámetros como tipo puntero. Como ejemplo, mostramos el siguiente programa:

```
void inter(int *x, int *y);

void main(void)
{
   int x, y;

   x = 10;
   y = 20;
   inter(&x, &y);
}

void inter(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

En el ejemplo anterior, a la variable x se le asigna el valor de 10 y a y el valor de 20. Enseguida se llama a la función inter() con las direcciones de x y de y como argumentos.

El objetivo de la función inter es la de intercambiar el valor de sus dos argumentos enteros. Por lo tanto después de la llamada a la función, x tendrá el valor de 20 y y el de 10.

Cuando se llama a una función con un nombre de array como argumento, se pasa a la función un puntero al primer elemento del array (el nombre de un array sin índice es un puntero al primer elemento del array). Esto significa que la declaración del parámetro debe ser de un tipo puntero compatible.

Existen tres formas de declarar un parámetro formal que va a recibir un puntero a un array.

a) Se puede declarar como un array:

```
#include "stdio.h"

void mostrar(int num[10]);

void main(void)
{
  int t[10], i;

  for(i=0; i<10; ++i) t[i] = i;
  mostrar(t);
}


void mostrar(int num[10])
{
  int i;

  for(i=0; i<10; i++) printf("%d ",num[i]);
}
```

Aunque el parámetro num se declara como un array de enteros de diez elementos, el compilador de C automáticamente lo convierte en un puntero a entero. Esto es necesario porque ningún parámetro puede recibir un array de enteros. Al pasar un puntero a un array se debe tener un parámetro de tipo puntero para recibirlo.

b) Se puede declarar como un array sin tamaño.

```
void mostrar(int num[])
{
  int i;

  for(i=0; i<10; i++) printf("%d ",num[i]);
}
```

En el caso anterior, num se declara como un array de enteros de tamaño desconocido. Es importante mencionar que en C no se comprueba los límites de los arrays, el tamaño real del array es irrelevante para el parámetro. Este método de declaración define a num como un puntero a entero.

c) Se puede declarar como un puntero.

```
void mostrar(int *num)
{
  int i;

  for(i=0; i<10; i++) printf("%d ",num[i]);
}
```

Esta es la forma más común y profesional para esta declaración.

En resumen, es importante entender que cuando se usa un array como argumento de una función, es su dirección lo que se pasa a la función, el código que existe dentro de la misma opera sobre el contenido real del array, permitiendo modificar el valor de sus elementos. Esto es una excepción al convenio de C de paso de parámetros por valor.

5.5 Argumentos de main()

En algunas ocasiones es útil pasar información al programa cuando se ejecuta. El método general es pasar información a la función main() mediante el uso de argumentos en la línea de órdenes.

Un **argumento de la línea de órdenes** es la información que sigue al nombre del programa en la línea de órdenes del sistema operativo.

Hay dos argumentos especiales ya incorporados, que son **argc** y **argv**, que se utilizan para recibir los argumentos de la línea de órdenes.

El parámetro **argc** contiene el número de argumentos de la línea de órdenes y es un entero. Siempre vale 1 por lo menos, ya que el nombre del programa cuenta como primer argumento.

El parámetro **argv** es un puntero a un array de punteros a caracteres. Cada elemento del array apunta a un argumento de la línea de órdenes. Todos los argumentos de la línea de órdenes son cadenas. Se puede declarar **argv** como sigue:

```
char *argv[];
```

Como ya sabemos los corchetes vacíos indican que es un array de longitud indeterminada.

Se pueden acceder a los argumentos individualmente indexando **argv**. Por ejemplo, **argv[0]** apunta a la primera cadena, que es siempre el nombre del programa; **argv[1]** apunta al primer argumento, y así sucesivamente.

En teoría se pueden tener hasta 32 767 argumentos, pero la mayoría de los sistemas operativos sólo permiten unos pocos. En la mayoría de los casos se utiizan estos argumentos para indicar un nombre de archivo o una opción.

El utilizar argumentos en la línea de órdenes da a los programas una apariencia profesional y facilita el uso del programa en archivos de procesamiento por lotes.

Es importante mencionar, que cuando no se vayan a usar los parámetros de la línea de órdenes, lo normal es declarar **main()** sin parámetros utilizando la palabra clave **void**. Sin embargo, si se desea, se puede dejar simplemente sin nada entre los paréntesis.

5.6 Vuelta de una Función

Hay dos formas en las que una función puede terminar su ejecución y volver al sitio en que se llamó.

La primera ocurre cuando se ha ejecutado la última sentencia de la función y, conceptualmente, se encuentra la llave } del final de la función.

La segunda, cuando se emplea la sentencia **return**. La mayoría de las funciones utilizan esta sentencia para terminar la ejecución, bien porque se tiene que devolver un valor o bien para simplificar y hacer el código más eficiente.

Todas las funciones, excepto aquellas de tipo void devuelven un valor, el cual se especifica explícitamente en la sentencia **return**. Es importante mencionar que el compilador de C devuelve el valor 0 cuando no se especifica explícitamente un valor de vuelta, pero no se debe contar con ello cuando se pretende tener código portable.

Si una función no se declara como **void**, puede ser usada como operando en cualquier expresión válida de C. Por ejemplo:

```
x = potencia(y);

if(max(x,y) > 100) printf("mayor");

for(c=getchar(); isdigit(c); ) ...;
```

En cambio, no es válido que una función sea el destino de una asignación, ejemplo:

```
intercambia(x,y) = 100;
```

En caso de que el compilador de C, encuentre una expresión como la anterior no compilará el programa.

Las funciones que se encuentran en programas del lenguaje C se pueden agrupar dependiendo de su valor devuelto, en alguno de los siguientes grupos:

El primer grupo es simplemente computacional, es decir son funciones diseñadas específicamente para realizar operaciones con sus argumentos y devolver un valor basado en esos cálculos. Es muy común que a una función computacional sea llamada también como **función pura**. Ejemplos de este grupo de funciones son las **funciones de biblioteca estándar** como lo son **sqrt()** (que calcula la raíz cuadrada) y **sin()** (que calcula el seno de sus argumentos).

El segundo grupo de funciones manipulan la información y devuelven un valor que indica simplemente el éxito o el fallo de esa manipulación. Ejemplo de éste grupo de funciones es la función de biblioteca **fclose()**, la cual se usan para cerrar un archivo. Si la operación de cierre tiene éxito, la función devuelve el valor 0; si no, la función devuelve un código de error.

El último grupo de funciones no tienen un valor de vuelta explícito. En esencia la función es estrictamente de tipo procedimiento y no genera un valor. Un ejemplo de éste grupo de funciones es la función **exit()**, la cual termina la ejecución de un programa. Todas las funciones que no devuelvan valores deben ser declaradas de tipo **void**, ya que esto evita que se usen en expresiones, previniendo así el mal uso accidental.

Cuando no se declara explícitamente el tipo del valor devuelto de una función, el compilador de C asigna automáticamente el tipo **int**. Pero en caso de que se quiera devolver un tipo de dato diferente, se requerirán dos pasos a seguir. El primero, es que se dé a la función un especificador de tipo explícito. y el segundo, el tipo de la función debe estar identificado antes de hacer la primera llamada. Esta es la única forma en que el compilador de C pueda generar un código correcto para funciones que devuelvan valores no enteros.

5.7 Declaración de Funciones

Las funciones se pueden declarar para que devuelvan cualquier tipo de dato válido en C. El método de la declaración es similar al de la declaración de las variables; el especificador de tipo

precede al nombre de la función, y además le dice al compilador qué tipo de datos va a devolver la función. Esta información es muy importante para que el programa se ejecute correctamente, porque tipos de datos diferentes tienen tamaños y representaciones internas diferentes.

Antes de que se pueda usar una función que devuelva un tipo no entero, se debe hacer saber su tipo al resto del programa. La razón es que, a menos que se indique lo contrario, el compilador de C asume que una función va a devolver un valor entero. Si el programa llama a una función, que devuelve un tipo diferente antes de la declaración de esa función, el compilador genera erróneamente el código de la llamada a la función. Para evitar esto, se debe usar una forma de declaración especial cerca del principio del programa para decirle al compilador qué valor va a devolver realmente la función.

Exiten dos formas de declarar una función antes de que sea usada:

a) La Forma Tradicional.

b) El Método de Prototipos.

5.8 Declaración de Funciones en Forma Tradicional

En el método tradicional, se especifica el tipo y el nombre de la función al principio del programa. La sentencia tradicional de declaración del tipo de una función tiene la siguiente forma general:

```
tipo nombre_función();
```

Aún cuando la función tome argumentos, no se especifica ninguno en la declaración del tipo.

Si no existiere la sentencia de declaración de tipo, se daría un error de discordancia entre el tipo de datos que devuelve la función y el tipo de datos que espera la rutina que la llama. Los resultados pueden ser extraños e impredecibles. Si ambas funciones están en el mismo archivo, el compilador detectará la discordancia de tipos y no compilará el programa. Sin embargo, si las funciones están en archivos diferentes el compilador no detectará el error. La comprobación de tipos no se hace al enlazar el programa o al ejecutarlo, sino al compilarlo. Por esta razón, se debe asegurar que los tipos sean compatibles.

Por ejemplo:

```
#include "stdio.h"

float suma();
float uno, dos;
```

```
void main(void)
{
  uno = 5.1;
  dos = 3.1;
  printf("%f", suma());
}


float suma()
{
  return uno + dos;
}
```

La primera declaración del tipo de la función le dice al compilador que la función **suma()** devuelve un tipo de dato en coma flotante. Esto permite al compilador generar correctamente el código de las lamadas a **suma()**. Sin esta declaración, el compilador indicaría un error de discordancia de tipos.

5.9 Prototipo de Funciones

El C del estándar *ANSI* ha ampliado la declaración tradicional de funciones al permitir declarar el **número y los tipos de los argumentos**. A ésta declaración ampliada se le denomina **Prototipo de Función.**

Es importante mencionar que los prototipos de funciones no estaba en el lenguaje original de C, sin embargo, son una de las ampliaciones más importantes que el ANSI ha introducido en C.

Los prototipos de funciones permite que se lleve a cabo una fuerte comprobación de tipos, ya que al usarlos se pueden 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 la siguiente:

```
tipo nombre_func(tipo param_1, tipo param_2, ...,tipo param_N);
```

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 dé un error, por lo que generalmente se incluyen.

Debido a la necesidad de compatibilidad con la versión original de C del **UNIX**, a los prototipos de funciones se les aplican ciertas reglas especiales. En primer lugar cuando se declara el tipo devuelto por la función sin la información de prototipo, el compilador asume que no se dá 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 paréntesis. Por ejemplo, si una función llamada **func()** devuelve un **float** y no tiene parámetros, su prototipo es:

```
float func(void);
```

Esto le indica al compilador que la función no tiene parámetros y que cualquier llamada a la función que utilice argumentos será errónea.

Otro aspecto importante sobre los prototipos, es la forma en que afecta a la **promoción** de tipos 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 promociones de tipos 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.

Los prototipos de funciones permiten localizar errores antes de que se den. Ayuda a verificar que un programa funcione correctamente al no permitir que se llame a funciones con argumentos discordantes.

El lenguaje C originalmente usaba un método de declaración de parámetros diferente. A este antiguo método a veces se le denomina forma **clásica**. Y a la utilización de prototipo de funciones se le denomina forma **moderna**. Siempre es bueno conocer las dos formas de declaración, ya que en la forma clásica existen millones de líneas de código que se encuentran en libros, revistas y en rutinas con cierta antigüedad.

La declaración de parámetros de funciones clásicas consisten en dos partes: una lista de parámetros, que va entre los paréntesis que siguen al nombre de la función y la declaración real de los parámetros, que va entre el paréntesis cerrado y la llave de apertura de la función.

La forma general de definición clásica de parámetros es:

```
tipo nombre_func(param_1, param_2, ... param_N)
tipo param_1;
tipo param_2;
 .
 .
 .
tipo param_N;
{
código de la función
}
```

El siguiente ejemplo muestra las dos difentes formas de una declaración de función:

Forma Moderna

```
float func(int a, int b, char c)
{
.
.
.
}
```

Forma clásica

```
float func(a, b, c)
int a, b;
char c;
{
.
.
.
}
```

5.10 Recursividad

En C, las funciones pueden llamarse a sí mismas. Si una expresión en el cuerpo de una función llama a la propia función se dice que ésta es recursiva.

La recursividad es el proceso de definir algo en términos de sí mismo y a veces se llama **definición circular**.

El ejemplo clásico de recursividad es el cálculo del factorial de un entero.

Como sabemos el factorial de un número n es el producto de todos los números enteros entre 1 y n

Cuando una función se llama a sí misma, se asigna espacio en la pila para las nuevas variables locales y parámetros, y el código de la función se ejecuta con estas nuevas variables desde el principio.

Una llamada recursiva no hace una nueva copia de la función. Sólo son nuevos los argumentos. Al volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la ejecución se reanuda en el punto de la llamada a la función dentro de la función.

Como en todo, la recursividad tiene algunas pequeñas desventajas. La mayoría de las rutinas recursivas no minimizan significativamente el tamaño del código ni ahorran espacio de

memoria. Además, las versiones recursivas de la mayoría de las rutinas se ejecutan un poco más despacio que sus versiones iterativas equivalentes, esto es debido a la repetidas llamadas a la función; pero esto normalmente no será repetitivo.

Muchas llamadas recursivas a una función pueden hacer que se desborde la pila. Debido a que el almacenamiento de los parámetros de la función y de las variables locales se hacen en la pila y cada nueva llamada crea una nueva copia de estas variables, es posible que la pila **sobreescriba** algún dato o la zona de memoria del programa. Esto puede llegar a ocurrir cuando una función recursiva pierde el control.

La principal ventaja de las funciones recursivas es que se pueden usar para crear funciones de algoritmos más claras y más sencillas, como por ejemplo, el algoritmo de ordenación rápida el cual es difícil de implementar de forma iterativa. Otro ejemplo, es precisamente utilizar funciones recursivas para la solución a problemas relacionados a la **inteligencia artificial**.

Es importante remarcar que cuando se escriben funciones recursivas, se debe tener una sentencia if en algún sitio que obligue a la función a volver sin que se ejecute la llamada recursiva. Si no se hace así la función nunca devolverá el control una vez que se le ha llamado.

5.11 Aspectos Importantes de las Funciones

Como ya lo hemos mencionado, las funciones son los bloques constructores de C y son cruciales para la creación de todos los programas, excepto los más simples.

Es importante saber que para algunas aplicaciones especializadas, puede ser necesario eliminar una función y reemplazarla con **código insertado**. El código insertado es el equivalente a las sentencias que se encuentran dentro de una función.

Es preferible utilizar código insertado en lugar de llamadas a funciones únicamente cuando el tiempo de ejecución sea crítico.

El código insertado es más rápido que una llamada a una función, primero porque una instrucción de llamada tiene su tiempo de ejecución y segundo porque si hay argumentos que pasar, estos se tiene que colocar en la pila, lo que también lleva tiempo.

Una función de propósito general es aquella que se va a utilizar en muchas situaciones distintas y por diferentes programadores.

CAPITULO VI

SISTEMA DE E/S

SISTEMA DE E/S

El lenguaje C es prácticamente único en su tratamiento de las operaciones de entrada/salida (E/S), porque no define ninguna palabra clave para realizarlas. Por el contrario, se realizan a través de funciones de biblioteca.

El sistema de E/S en el lenguaje C es una pieza de ingeniería elegante que ofrece un mecanismo flexible a la vez que consistente para transferir datos entre dispositivos. Además se puede dividir en dos categorías básicas:

a) E/S por Consola
b) E/S por Archivos

Se puede acceder a una función de E/S desde cualquier sitio de un programa con simplemente escribir el nombre de la función, seguido de una lista de argumentos encerrados entre paréntesis. Algunas funciones no requieren argumentos, pero deben aparecer los paréntesis vacíos.

La mayoría de las versiones de C incluyen una colección de **archivos cabecera** que proporcionan la información necesaria (como por ejemplo constantes simbólicas) para las distintas funciones de biblioteca. Estos archivos se incluyen dentro de un programa mediante una sentencia **#include** al comienzo del mismo. Como norma general, el archivo cabecera requerido para la E/S estándar se llama **stdio.h**; por lo que cuando queramos utilizar este tipo de funciones es necesario al comenzar el programa, escribir la siguiente sentencia:

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

6.1 E/S por Consola

Las funciones de E/S por consola son aquellas que controlan la entrada por teclado y la salida a través de la pantalla (E/S estándares). La entrada y salida también pueden ser redirigidos a otros dispositivos.

El sistema de E/S por consola es bastante amplio e incluye muchas funciones diferentes, tales como **getchar()**, **putchar()**, **scanf()**, **printf()**, **gets()**, **puts()**, etc. Estas funciones permiten la transferencia de información entre la computadora y los dispositivos de entrada/salida estándar.

Las funciones, **getchar()** y **putchar()**, permiten la transferencia hacia adentro y hacia a fuera de la computadora de caracteres sueltos; **scanf()** y **printf()** son funciones más complicadas, pero permite la transferencia de caracteres sueltos, valores numéricos y cadenas de caracteres; **gets()** y **puts()** permiten la

entrada y salida de cadenas de caracteres.

6.2 Escritura y Lectura de Caracteres

Las funciones más simples de E/S por consola son:

getchar()
putchar()

La función **getchar()**, devuelve un carácter leído del dispositivo de entrada estándar (teclado) y no requiere de argumentos. La función **putchar()** imprime su argumento, que es un carácter en el dispositivo de salida estándar (la pantalla) y en la posición actual del cursor.

Los prototipos de estas funciones son las siguientes:

```
int getchar(void);
int putchar(int c);
```

Como se puede observar la función **getchar()** devuelve un entero, pero el byte de menor orden contiene el carácter. A si mismo la función **putchar()** se declara usando un parámetro de tipo entero, pero el byte de menor orden es el que realmente se muestra en la pantalla. El uso de enteros se debe a la compatibilidad con el compilador de C original de **UNIX**. El archivo de cabecera que se deberá incluir cuando se usen estas funciones deberá ser **stdio.h**.

Si se encuentra una condición de **fin de archivo** cuando se está leyendo un carácter con la función **getchar()**, la función devolverá de forma automática el valor de la constante simbólica **EOF** (este valor se define dentro del archivo stdio.h y normalmente tendrá asignado el valor -1, aunque puede variar de un compilador a otro). La detección **EOF** de esta forma, hace posible descubrir el fin de archivo en el momento y lugar que ocurra, y de ello se puede tomar alguna decisión en los programas.

La función **getchar()** también se puede utilizar para leer cadenas de varios caracteres leyendo en un bucle la cadena carácter a carácter. Pero exiten otras funciones que facilitan más la lectura de cadenas, éstas funciones las veremos más adelante.

La forma general de hacer referencia a la función **getchar()** es la siguiente:

```
variable_carácter = getchar();
```

donde **variable_carácter** es alguna variable de tipo carácter previamente declarada.

Por otra parte, se puede llamar a la función **putchar()** con un argumento de carácter. Aunque **putchar** se declara usando un parámetro entero, sólo el byte de menor orden es el que se muestra en la pantalla.

En general, una referencia a la función **putchar()** se escribiría así:

```
putchar(variable_carácter)
```

en donde **variable_carácter** hace referencia a una variable del tipo carácter que ya ha sido declarada.

La función **putchar** devuelve o bién el carácter escrito, o bien EOF (fin de archivo) si se ha producido un error.

También se puede utilizar esta función para visualizar una constante de cadena de caracteres almacenando la cadena en un array unidimensional de tipo carácter y mediante la utilización de un bucle, se pueden ver los caracteres uno a uno.

Existen otras funciones alternativas a **getchar()** que son:

```
getch()
getche()
```

sus protolipos son:

```
int getch(void);
int getche(void);
```

Para la mayoría de los compiladores, los prototipos de éstas funciones se encuentran en el archivo conio.h .

La función **getch()** espera a que se pulse una tecla e inmediatamente después devuelve un valor. No muestra el carácter en la pantalla.

La función **getche()** es igual que getch() pero sí muestra el carácter en la pantalla.

6.3 Escritura y Lectura de Cadenas

Las funciones **gets()** y **puts()** facilitan la transferencia de cadenas de caracteres entre la computadora y los dispositivos de E/S estándar.

Cada una de estas funciones acepta un sólo argumento. El argumento debe ser un dato que represente una cadena de caracteres (un array de caracteres). La cadena de caracteres puede incluir caracteres de espaciado. En el caso de la función **gets**, la cadena se introducirá por el teclado y terminará con un carácter de nueva línea. Es decir, se pueden escribir caracteres

por el teclado hasta pulsar un salto de carro. El salto de carro no formará parte de la cadena; en su lugar, un terminador nulo es situado al final y después **gets()** acaba.

Es importante hacer notar que no se puede utilizar **gets()** para devolver un salto de carro, en cambio **getchar()** puede hacerlo.

Se pueden corregir errores al momento de estar escribiendo una cadena utilizando la tecla de retroceso antes de pulsar ENTER.

El prototipo de la función **gets()** es:

```
char *gets(char *cad);
```

donde cad es un array de caracteres donde se colocan los caracteres teclados por el usuario. La función devuelve un puntero señalando a la cadena recién leída o un puntero nulo si ocurre un error.

El prototipo de la función **gets()** se encuentra en el archivo stdio.h.

La función **puts()** escribe su argumento de tipo cadena en la pantalla seguido de un carácter de salto de línea. Su prototipo es el siguiente:

```
int puts(char *s);
```

Una llamada a puts requiere mucho menos tiempo que la misma llamada **printf()**, porque **puts()** sólo puede imprimir una cadena de caracteres y no números o conversiones de formato. Por ésto **pust()** requiere mucho menos espacio y se ejecuta más rápido que **print()**.

La función **puts()** devuelve EOF (fin de archivo) si ocurre algún error.

En figura 6.1 se muestra, en forma de resumen, las funciones más simples que realizan las operaciones de E/S por consola.

6.4 E/S por Consola con Formato

Las funciones **scanf()** y **printf()** realizan la entrada y la salida con formato, esto es, pueden leer o escribir datos en varios formatos que pueden ser controlados.

La función **printf()** escribe datos en la consola y la función **scanf()**, su complementaria, lee datos desde el teclado. Ambas funciones pueden operar sobre cualquiera de los tipos de datos existentes, incluyendo caracteres, cadenas y números.

Función	Operación
getchar()	Lee un carácter del teclado. Espera un salto de carro.
getche()	Lee un carácter con eco. No espera un salto de carro. No está definida por el ANSI.
getch()	Lee un carácter sin eco. No espera un salto de carro. No está definida por el ANSI.
putchar()	Escribe un carácter en la pantalla.
gets()	Lee una cadena del teclado.
puts()	Escribe una cadena en la pantalla.

Figura 6.1

6.5 Escritura de Datos con Formato (Función printf())

Se pueden escribir datos en el dispositivo de salida estándar utilizando la función de biblioteca **printf()**. Con esta función se puede escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres.

El prototipo de la función **printf()** se encuentra en el archivo **stdio.h** y es:

```
int printf(char *cadena_control, lista_argumentos);
```

donde **cadena_control** hace referencia a una cadena de caracteres que contiene información sobre el formato de la salida. Los **argumentos** pueden ser constantes, variables simples, nombres de arrays o expresiones más complicadas. También se pueden incluir referencias a funciones.

La función **printf()** devuelve el número de caracteres escritos o bien un valor negativo, si se produce un error.

La **cadena_control** está formada por dos tipos de elementos. El **primer elemento es el carácter** que se mostrará en la pantalla. El **segundo elemento contiene especificadores de formato** que definen la forma en que se muestran los argumentos posteriores.

El **especificador de formato** empieza siempre con un **%** y va seguido por el **código del formato**. Debe haber exactamente el

mismo número de argumentos que especificadores de formato y ambos deben coincidir en su órden de aparición de izquierda a derecha.

En la figura 6.2 se muestra la forma en que está compuesta la cadena de control de la función printf().

Figura 6.2

En la figura 6.3 se presentan los principales especificadores de formato utilizados por la función printf().

Código	Formato
%c	Carácter.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Coma flotante con Notación científica (e minúscula).
%E	Coma flotante con Notación científica (E mayúscula).
%f	Coma flotante sin Exponente.
%g	Usar %e o %f, el más corto.
%G	Usar %E o %F, el más corto.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimales sin signo (letras minúsculas).
%X	Hexadecimales sin signo (letras mayúsculas).
%p	Mostrar un Puntero.
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos.
%%	Imprimir el signo %

Figura 6.3

6.6 Impresión de Caracteres

Si se quisiera imprimir un sólo carácter se utiliza el especificador de formato **%c**. Para imprimir cadenas se utiliza el especificador de formato **%s**.

6.7 Impresión de Números

Si se quiere indicar un número decimal con signo, se pueden utilizar tanto el especificador de formato **%d** como el **%i**, los dos son equivalentes.

Un valor sin signo es representado por **%u**.

El especificador de formato **%f** imprime números en coma flotante. Los especificadores **e** y **E** le indican a la función **printf()** que muestre su argumento de doble longitud en notación científica.

Otro especificador de formato que es de gran ayuda es **%g** o **G**, éste le puede indicar a la función **printf()** que utilice ya sea el especificador de formato **f** o **e** dependiendo de cuál de los dos genere la salida más corta.

Los especificadores **o** y **x** permiten imprimir enteros sin signo en formato octal y hexadecimal respectivamente. Como ya es conocido, en la numeración hexadecimal se incluyen las letras de la **A** a la **F** para representar los números del 10 al 15, si se quiere imprimir las letras en mayúsculas es necesario utilizar el especificador **X** en vez de **x** que las escribe en minúsculas.

6.8 Impresión de una Dirección

El especificador de formato **%p** nos permite imprimir una dirección de la máquina en un formato compatible con el tipo de direccionamiento utilizado en la computadora.

6.9 Especificador %n

Este especificador de formato es muy diferente a los demás. En lugar de hacer que **printf()** imprima algo, produce el efecto de cargar la variable apuntada por su correspondiente argumento con un valor igual al número de caracteres que han sido impresos. En otras palabras, el valor que corresponde al especificador de formato **%n** debe ser un puntero a una variable. Después de terminada la ejecución de **printf()** esa variable tendrá el número de caracteres que se han impreso hasta el punto en el cual encontró **%n**. Un ejemplo de ello se presenta en la siguiente página.

```
#include "stdio.h"

main()
{
  int ejemplo;

  printf("Esto%nes un texto\n", &ejemplo);
  printf("%d",ejemplo);
}
```

Al ejecutar este programa se imprime lo siguiente:

```
Esto es un texto
4
```

El especificador de formato %n se utiliza fundamentalmente para permitir al programa asignar dinámicamente formatos.

6.10 Modificadores de Formato

Algunos especificadores de formato pueden aceptar modificadores que alteran su signifcado lévemente. Por ejemplo, se puede especificar:

a) La longitud mínima de un campo,
b) El número de decimales (Precisión),
c) Ajuste de la salida,
d) Modificadores de formato de enteros,
e) otros modificadores de formato (#, *).

El modificador de formato se sitúa entre el signo % y el código real.

6.11 Longitud Mínima de un Campo

Se puede especificar la longitud mínima de un campo colocando un número entero entre el signo de % y el código de formato. Esto hace que se rellene la salida con espacios para asegurar que el campo alcanza una cierta longitud mínima. En caso de que la cadena o el número sea más largo que el mínimo, se imprimirá toda su longitud. Los huecos se rellenan con espacios por omisión. Si en un momento dado se quisiera que se rellenara con ceros se le pone un cero antes del especificador de longitud del campo, como por ejemplo:

```
printf("valor %05d", num)
```

se rellenaria con ceros un número con menos de 5 dígitos para que su longitud total sea 5.

Es muy frecuente utilizar el modificador de longitud mínima de campo para crear tablas en las cuales las columnas aparezcan alineadas.

6.12 Número de Decimales (Precisión)

Si existe un especificador que tenga una longitud mínima de campo, el **especificador de precisión** le puede seguir.

El especificador de precisión consisten en un punto seguido de un entero. Su significado exacto depende del tipo de dato al que se aplica.

Cuando se aplica un especificador de precisión a datos en coma flotante, éste determina el número de posiciones decimales a imprimir. Por ejemplo, **%10.4f** imprime un número de al menos 10 caracteres con cuatro posiciones decimales.

Si el especificador de precisión se aplica a cadenas, éste determina la longitud máxima del campo. Como por ejemplo, **%5.7s**, ésto hace que se imprima una cadena de al menos 5 caracteres de longitud y no más de 7; en caso de que la cadena sea más larga que la longitud máxima del campo se truncarán los caracteres finales.

Si el especificador de precisión se aplica a enteros se determina el número mínimo de dígitos que aparecerán por cada número. Para alcanzar el número requerido de dígitos se añaden ceros por delante.

6.13 Ajuste de la Salida

Por omisión, todas las salidas están justificadas por la derecha. Esto, si la longitud del campo es mayor que el dato a imprimir. Se puede hacer que la salida sea ajustada a la izquierda colocando un signo menos después del tanto por ciento, como por ejemplo **%-10.2f**, ésto hace que se ajuste a la izquierda un número en coma flotante con dos decimales en un campo de diez caracteres.

6.14 Modificadores de Formato de Enteros

Existen dos modificadores de formato que permite a la función **printf()** imprimir enteros largos (**l**) y cortos (**h**).

Estos modificadores se pueden aplicar a los especificadores de tipo **d**, **i**, **o**, **u**, y **x**.

El modificador **l** indica a **printf()** que lo que sigue es un tipo de dato **largo**. Por ejemplo, **%ld**, significa que se va a imprimir un entero largo.

El modificador **h** indica a la función **printf()** que se va a imprimir un entero corto. Por ejemplo, **%hu**, indica que el dato es del tipo entero corto sin signo.

El modificador **L** puede preceder a los especificadores de coma flotante **e**, **f** y **g** e indica que le sigue uno de doble longitud.

6.15 Otros Modificadores de Formato

Si a los especificadores de formato **g**, **f** o **e** le precede un simbolo **#**, se asegura que aparecerá un punto decimal incluso si no hay dígitos decimales.

Si se le antepone el simbolo **#** al especificador de formato **x** se imprimirá un número hexadecimal con el prefijo **0x**.

Los especificadores de longitud mínima de campo y de precisión pueden ser pasados como **argumentos** en la función **printf()** en lugar de hacerlo mediante **constantes**. Para conseguir esto se usa el signo * como marca de posición. Cuando lea la cadena de formato, **printf()** asociará el * a un argumento que aparezca en el mismo lugar de orden. Por ejemplo:

```
printf("%*.*f", 10, 4, 123.3);
```

la longitud mínima de campo es 10, la precisión es 4 y el valor a imprimir es 123.3 .

6.16 Lectura de Datos con Formato (Función scanf())

La función **scanf()** es la rutina de entrada por consola de propósito general. Puede leer todos los tipos de datos que suministra el compilador y convierte los números automáticamente al formato interno apropiado. La función **scanf()** es la función inversa de **printf()**.

El prototipo de la función **scanf()** es el siguiente:

```
int scanf(char *cadena_control, lista_argumentos);
```

y se encuentra en el archivo cabecera **stdio.h**.

La función **scanf()** devuelve el número de datos a los que se ha asignado un valor con éxito. Si se produce un error, se devuelve **EOF**. La **cadena_control** determina cómo se leen los valores en las variables a las que se hace referencia en la **lista_argumentos**.

La **cadena_control** consta de tres clases de caracteres:

a) Especificadores de Formato.
b) Caracteres de espacio en blanco.
c) Caracteres no de espacios en blanco.

Todas las variables que se utilizan para recibir valores a través de la función **scanf()** se deben pasar por sus **direcciones**. Esto significa que todos los argumentos deben ser punteros a las variables que se usan como argumentos. Como ya sabemos, ésta es la manera en la que C crea una llamada por referencia y permite a una función alterar el contenido de su argumento. Por ejemplo, para leer un entero en la variable **cuenta**, se utilizaría la siguiente llamada a **scanf()**:

```
scanf("%d", &cuenta);
```

Las cadenas se asignarán a arrays de caracteres y el nombre del array, sin índice, es la dirección del primer elemento del array, por lo que para asignar una cadena al array de caracteres llamada **cadena**, se utilizaría:

```
scanf("%s",cadena);
```

Como **cadena** es ya un puntero no es necesario que sea precedido por el operador **&**.

6.17 Especificadores de Formato de Entrada

Los especificadores de formato de entrada van precedidos por el signo **%** e indican a la función **scanf()** que tipo de dato se va a leer.

La figura 6.4 presenta los principales especificadores de formato para la función **scanf()**.

Código	Formato
%c	Leer un único carácter.
%d	Leer un entero decimal.
%i	Leer un entero decimal.
%e	Leer un número de tipo coma flotante.
%f	Leer un número de tipo coma flotante.
%g	Leer un número de tipo coma flotante.
%o	Leer un número octal.
%s	Leer una cadena.
%x	Leer un número hexadecimal.
%p	Leer un puntero.
%n	Recibe un valor entero igual al número de caracteres leidos.
%u	Leer un entero sin signo.
%[]	Muestrea un conjunto de caracteres.

Figura 6.4

6.18 Lectura de Números

Para leer un número decimal se puede utilizar los especificadores de formato **%d** e **%i**.

Para leer un número en coma flotante representado en notación estándar o científica se utilizan los especificadores **%f**, **%e** y **%g**.

Para leer enteros en forma octal y hexadecimal se utilizan los especificadores **%o** y **%x**, el cual puede escribirse tanto en mayúsculas como en minúsculas.

Para introducir un entero sin signo se utiliza el especificador de formato **%u**.

Es importante hacer notar que la función **scanf()** termina de leer un número cuando encuentra el primer carácter no numérico.

6.19 Lectura de Caracteres Individuales

Como ya sabemos, se pueden leer caracteres utilizando la función **getchar()** o una función derivada. La función **scanf()** también se puede utilizar para éste propósito si se utiliza el especificador de formato **%c**. Es importante hacer notar, que los espacios en blanco, tabulaciones, y salto de línea que usualmente se usan como saparadores de campos se leen como cualquier otro carácter y ésto, puede acarrear problemas en los programas.

6.20 Lectura de Cadenas

Se puede utilizar la función **scanf()** para leer una cadena de la secuencia de entrada utilizando el especificador de formato **%s**. La utilización de éste especificador hace que la función lea caracteres **hasta que se encuentre con un tipo de carácter en blanco**. Los caracteres que se leen se asignan al array de caracteres apuntado por el argumento correspondiente y se añade un carácter nulo al final.

Por lo que se refiere a **scanf()**, un carácter en blanco es un espacio, un salto de carro o una tabulación.

La diferencia de la función **gets()** y **scanf()**, es que la primera lee una cadena hasta que se pulsa ENTER o RETURN, mientras que para la segunda lee una cadena hasta el primer caracter en blanco. Esto significa que **scanf()** no se puede utilizar para leer una cadena como "hola como estas", porque el primer espacio termina con el proceso de lectura.

6.21 Lectura de una Dirección

Para leer una dirección de memoria (un puntero), se utiliza el especificador de formato **%p**. No se debe intentar utilizar un entero sin signo o cualquier otro especificador de formato para leer una dirección, porque **%p** hace que **scanf()** lean la dirección en el formato utilizado por la Unidad Central de Proceso (C.P.U.).

6.22 Especificador %n

El especificador **%n** le indica a la función **scanf()** que asigne a la variable apuntada por el correspondiente argumento el número de caracteres leídos desde la secuencia de entrada hasta el punto en el que se encuentra el especificador **%n**.

6.23 Juego de Inspección

El C del estándar ANSI ha añadido a la función **scanf()** una nueva característica llamada **Juego de Inspección**.

Un **juego de inspección** define un conjunto de caracteres que pueden leerse utilizando **scanf()** y ser asignados al correspondiente array de caracteres.

El juego de inspección se define poniendo una cadena con los caracteres que se van a leer entre corchetes. El corchete abierto debe ir precedido de un signo **%**. Por ejemplo, el siguiente juego de inspección indica a la función **scanf()** que lea sólo los caracteres "X", "Y" y "Z":

```
%["XYZ"]
```

Cuando se utiliza un juego de inspección, **scanf()** lee caracteres y los asigna al correspondiente array de caracteres, hasta que se encuentra con un carácter que no está en el juego de inspección. La variable correspondiente debe ser un puntero a un array de caracteres. Después de la ejecución de **scanf()**, este array contendrá una cadena que termina con un carácter nulo y compuesta con los caracteres que han sido leidos.

Se puede especificar un juego inverso si el primer carácter del conjunto es ^. El signo ^ indica a la función **scanf()** que acepte cualquier carácter que no esté definido por el juego de inspección.

También se puede especificar un rango utilizando un guión. Por ejemplo, se pueden aceptar caracteres desde la "A" hasta la "Z":

```
%["A-Z"]
```

Es importante remarcar que el juego de inspección es sensible a mayúsculas y a minúsculas y sí se desea leer tanto de una como de otra se debe especificar por separado.

6.24 Carácter en Blanco

Un carácter en blaco en la cadena de control hace que la función **scanf()** salte uno o más caracteres en blanco de la secuencia de entrada.

Un carácter en blanco puede ser un espacio, una tabulación o un carácter de salto de línea. En esencia, un carácter en blanco en la cadena de control hace que **scanf()** lea, pero no almacene, cualquier número de espacios en blanco hasta que encuentre el primer carácter que no sea de ese tipo.

Un carácter distinto de espacio en blanco en la cadena de control hace que **scanf()** lea y descarte los caracteres que coincidan con él en la secuencia de entrada. Por ejemplo, **%d,%d** hace que la función **scanf()** lea un entero, lea y descarte una coma y posteriormente lea otro entero. Si no se encuentra el carácter especificado, la función **scanf()** termina su ejecución. Si se desea leer y descartar un signo de **%** se debe utilizar **%%** en la cadena de control.

6.25 Modificadores de Formato de Entrada

Al igual que la función **printf()**, **scanf()** permite que se modifiquen algunos de sus especificadores de formato. Entre los **modificadores de formato de entrada** tenemos:

a) Modificador de longitud máxima de campo.
b) Modificador **l**, **h** y **L**.
c) Modificador *

6.26 Modificador de Longitud Máxima de Campo

Este es un entero, situado entre el **%** y el formato, que limita el número de caracteres leídos para ese campo. Por ejemplo:

```
scanf("%20s", cadena);
```

Aquí no se puede leer más de 20 caracteres en **cadena**.

Si la secuencia de entrada está formada por más de 20 caracteres, una llamada porterior a la función comenzará donde la anterior acabó. Por ejemplo, si hubiéramos introducido el alfabeto:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

como respuesta a la anterior llamada de la función **scanf()**, sólo se asignaría a **cadena** los primeros 20 caracteres, o lo que es lo mismo, hasta el carácter "T", debido al especificador de longitud máxima. El resto de los caracteres no se utilizan todavía, pero en caso de que se se realizara otra llamada a

scanf() como:

```
scanf("%s",cadena);
```

se asignaría automáticamente las letras "UVWXYZ" a **cadena**. La asignación a un campo puede terminar antes de que se alcance su longitud máxima, si se encuentra un carácter en blanco. En éste caso, **scanf()** prosigue con el siguiente campo.

6.27 Modificador de Formato l, h y L

Para poder leer un entero largo o un entero corto se utilizan los modificadores de formato **l** y **h**.

Si se quiere leer un entero largo se pone "**l**" delante del especificador de formato. Y lo mismo es para leer un entero corto sólo que en vez de **l** es **h**. Estos modificadores de formato se utilizan con los formatos **d**, **i**, **o** y **x**.

Como sabemos los especificadores **%f**, **%e** y **%g** indican a **scanf()** que asigne los datos a una variable de tipo **float**. Si se pone la letra **l** delante de alguno de estos especificadores, la función **scanf()** asigna los datos a una variable de tipo **double**. Si se usara **L** se le indica a **scanf()** que la variable que recibe los datos es de tipo **long double**.

6.28 Modificador de Formato *

En ocasiones se puede indicar a **scanf()** que lea un campo pero que no lo asigne a ninguna variable. Esto se hace precediendo al código de formato de ese campo un *. Por ejemplo:

```
scanf("%d%*c%d", &x, &y);
```

de esta forma se puede introducir el siguiente par de coordenadas 10,10. Se leerá correctamente la coma, pero no se asignará a nada. La supresión de asignación es especialmente útil cuando sólo se necesita procesar una parte de lo que se introduce.

6.29 E/S por Archivos

El sistema de E/S por archivos del ANSI C es flexible y potente. Permite leer o escribir fácilmente cualquier tipo de dato. Los datos se pueden transferir en su representación binaria interna o en formato de texto normal, lo que hace que se puedan crear archivos que satisfagan cualquier necesidad.

Muchas aplicaciones requieren leer información de un periférico auxiliar de almacenamiento. Tal información se almacena en el

periférico en la forma de un **archivo de datos**, el cual, permite almacenar la información de modo permanente, para ser accedida o alterada cuando sea necesario.

En C existe un conjunto extenso de funciones de biblioteca para crear y procesar archivos de datos. A diferencia de otros lenguajes de programación, en C no se distingue entre archivos secuenciales y de acceso directo (acceso aleatorio). Pero existen dos tipos distintos de archivos de datos, llamados **archivos secuenciales de datos** (o estándar) y **archivos orientados a sistema** (o de bajo nivel). Es más fácil trabajar con los primeros ya que son los más usados.

Los **archivos de datos secuenciales** se pueden dividir en dos categorías. En la primera categoría se encuentran los archivos que contienen **caracteres consecutivos**. Estos caracteres pueden interpretarse como datos individuales, como componentes de una cadena o como números. La manera de interpretarlos es determinada dentro de las funciones de biblioteca usadas para transferir la información o por las especificaciones de formato dentro de las funciones de biblioteca, tales como scanf y printf

La segunda categoría de archivos de datos secuenciales, a menudo llamados **archivos sin formato**, organiza los datos en bloques contiguos de información. Estos bloques representan estructuras de datos más complejos como array y estructuras. Existe un conjunto de funciones de biblioteca para tratar este tipo de archivos. Estas funciones proveen instrucciones simples que pueden transferir array completos o estructuras a/o desde un archivo de datos.

Los archivos orientados al sistema están más relacionados con el sistema operativo de la computadora, que los archivos secuenciales. Es complicado trabajar con ellos, pero su uso puede ser más eficiente para cierto tipo de aplicaciones. Para procesar este tipo de archivos se requiere un conjunto separado de procedimientos con sus funciones de biblioteca correspondientes.

En contraste con las funciones de E/S de archivos de datos del estándar ANSI, el antiguo C del UNIX contiene dos sistemas distintos de rutinas que manejan las operaciones de E/S. El primer método se equipara vagamente con el definido por el estándar ANSI y se llama **sistemas de archivos con buffer** ("con formato" o de "alto nivel"). El segundo es el **sistema de archivos tipo UNIX** ("sin formato" o "sin buffer") y está definido sólo en el antiguo estándar de UNIX. El estándar ANSI no define el sistema de archivos sin buffer porque, entre otras cosas, los dos sistemas son muy redundantes y el sistema tipo UNIX puede no ser útil en ciertos entornos que soporta C. El hecho de que ANSI no haya incluido el sistema de E/S tipo UNIX sugiere que su utilización se irá cada vez más extinguiéndose.

6.30 Funciones para el Manejo de Archivos

El sistema de archivos ANSI está compuesto por varias funciones interrelacionadas. En la figura 6.5 se presentan las funciones más importantes en el manejo de archivos.

Nombre	Función
fopen()	Abre un archivo.
fclose()	Cierra un archivo.
putc()	Escribe un carácter en un archivo.
fputsc()	Escribe un carácter en un archivo.
getc()	Lee un carácter de un archivo.
fgetc()	Lee un carácter de un archivo.
fseek()	Busca un byte específico de un archivo.
fprintf()	Hace lo mismo en archivos, que printf() en consola.
fscanf()	Hace lo mismo en archivos, que scanf() en consola.
feof()	Devuelve cierto si se ha llegado al final del archivo.
ferror()	Devuelve cierto si se produce un error.
rewind()	Inicializa el indicador de posición del archivo al principio de éste.
remove()	Borra un archivo.
fflush()	Vacía un archivo.

Figura 6.5

Para que estas funciones se puedan utilizar en cualquier programa es necesario que se incluya el archivo de cabecera **Stdio.h**. En este archivo se suministran los prototipos de las funciones de E/S y define los siguientes tipos:

size_t fpos_t FILE

El tipo **size_t** es esencialmente el mismo que **unsigned**, al igual que **fpos_t**. El tipo **FILE** es un tipo especial de estructura que establece el área de buffer.

También en **Stdio.h** se define varias macros, las más importante son:

```
EOF
SEEK_SET
SEEK_CUR
SEEK_END
```

La macro **EOF** se define generalmente como -1 y es el valor devuelto cuando una función de entrada intenta leer más allá del final del archivo.

Las otras macros se utilizan con la función **fseek()**, la cual es la función que realiza el acceso directo sobre archivos y que veremos con mayor detenimiento más adelante.

6.31 Puntero a un Archivo

Un **puntero a un archivo** es un puntero a una información que define varias cosas sobre él, incluyendo el **nombre**, el **estado** y la **posición actual** del archivo.

En otras palabras, un puntero a un archivo es una variable de tipo puntero al tipo **FILE**.

Cuando se trabaja con archivos secuenciales de datos, el primer paso es establecer un **área de buffer**, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria de la computadora y el archivo de datos. Esta área de buffer permite leer y escribir información del archivo más rápidamente de lo que sería posible de otra manera. El área del buffer se establece declarando una variable de tipo puntero a archivo. Por ejemplo:

```
FILE *fp;
```

En donde **FILE** es un tipo especial de estructura que establece el área de buffer y ***fp** es la variable puntero que indica el principio de esta área. El tipo de estructura de **FILE** está definido en el archivo **stdio.h**

6.32 Apertura de un Archivo. Función fopen().

Un archivo de datos debe ser **abierto** antes de ser creado o procesado. Esto asocia el nombre de archivo con el área de buffer.

Al abrir un archivo de datos se debe de especificar su uso. Si es de sólo para lectura, sólo para escritura o para lectura/escritura.

Para abrir un archivo se usa la función de biblioteca **fopen**. Por ejemplo:

```
fp = fopen(nombre_archivo, modo_archivo);
```

Donde **nombre_archivo** es un puntero a una cadena de caracteres que representan un nombre válido del archivo y puede incluir una especificación de directorio. El nombre del archivo deberá ser consistente con las reglas para nombrar archivos del sistema operativo.

La cadena **modo_archivo** representa la manera en la que se puede abrir el archivo. En la figura 6.6 se enlistan los diferentes valores que son permitidos.

La función **fopen()** retorna un puntero al principio del área de buffer asociada con el archivo. Un programa nunca debe alterar el valor de ese puntero. Se retorna un valor NULL si no se puede abrir el archivo (por ejemplo, en caso de que no se encuentre).

Si quisiéramos abrir un archivo llamado **prueba** para escritura, se escribirían las siguientes líneas:

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

O bién, se puede escribir el siguiente código que permite detectar cualquier error al abrir el archivo. Errores tales como si el disco estuviera protegido contra escritura o si estuviera lleno.

```
FILE *fp;
if((fp = fopen("prueba",w)) == NULL ) {
    printf("Problemas al abrir el archivo\n");
}
```

Si se utiliza la función **fopen()** para abrir un archivo para escritura, cualquier archivo que ya existiese con ese nombre se borraría y se crearía uno nuevo. Si no existiese ninguno, simplemente crearía uno nuevo.

En caso de que se quisiera añadir información al final de un archivo, se deberia utilizar el modo "a".

Sólo se pueden **abrir** archivos que ya existen para realizar operaciones de lectura. En caso de que el archivo no existiera, se devolvería un error.

Modo	Significado
r	Abre un archivo de texto para lectura.
w	Crea un archivo de texto para escritura. Si ya existe un archivo con ese nombre será destruido y creado uno nuevo en su lugar.
a	Abre un archivo de texto para añadir. (Se añade información nueva al final del archivo) Se creará un archivo nuevo si no existe el archivo especificado.
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. Si ya existe un archivo con ese nombre será destruido y creado uno nuevo en su lugar.
a+	Añade o crea un archivo de texto para lectura/escritura. Se creará un archivo nuevo en caso de que no exista.
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 lectura/escritura.

Figura 6.6

Si se abre un archivo para realizar operaciones de lectura/escritura, y ya existía no será borrado. Sin embargo si no existía será creado.

El estándar ANSI especifica que pueden estar abiertos, por lo

menos, ocho archivos en cualquier momento. Sin embargo la mayoría de los compiladores y entornos de C permiten un número mayor.

6.33 Cierre de un Archivo. Función fclose().

La función **fclose()** cierra un archivo que fué abierto mediante una llamada **fopen()**. Escribe toda la información que todavía se encuentre en el buffer del disco y realiza un cierre formal del archivo a nivel del sistema operativo.

Un eror en el cierre de un archivo puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y errores subsecuentes.

En la mayoría de los casos, existe un límite del sistema operativo al número de archivos abiertos que puede haber simultáneamente, por lo que puede ser necesario cerrar un archivo antes de abrir otro.

La función **fclose()** tiene el siguiente prototipo:

```
int fclose(FILE *fp);
```

donde **fp** es el puntero al archivo devuelto por la llamada a la función **fopen()**. Si se devuelve un valor 0 significa que la operación de cierre ha tenido éxito.

Generalmente la función **fclose()** sólo falla cuando un disco se ha retirado antes de tiempo o cuando no queda espacio libre en disco.

Es una buena práctica de programación cerrar explícitamente los archivos de datos mediante la función **fclose()**, aunque la mayoría de los compiladores de C cerrarán automáticamente los archivos de datos al final de la ejecución del programa, si no está presente una llamada a **fclose()**.

6.34 Escritura y Lectura de un Carácter

El estándar ANSI define dos funciones equivalentes que escriben un carácter:

```
putc()
fputc()
```

Estas dos funciones son idénticas, solamente existen para preservar la compatibilidad con versiones antiguas de C.

La función **putc** escribe caracteres en un archivo que haya sido abierto, previamente para operaciones de escritura utilizando claro, la función **fopen()**. El prototipo de esta función es:

```
int putc(int c, FILE *fp);
```

Donde **fp** es el puntero al archivo devuelto por la función **fopen()** y **c** es el carácter que se va a escribir. El puntero al archivo le dice a la función **putc()** en que archivo de disco debe escribir; **c** se define como **int**, pero sólo se utiliza el byte menos significativo.

Si una operación con la función **putc()** tiene éxito, devuelve el carácter escrito. En otro caso, devuelve **EOF**.

El estándar ANSI también define dos funciones equivalentes para leer un carácter:

```
getc()
fgetc()
```

Estas funciones son idénticas. La función **getc()** lee caracteres de un archivo abierto en modo lectura mediante la función **fopen()**. El prototipo de la función **getc()** es:

```
int (getc(FILE *fp);
```

Donde **fp** es un puntero al archivo de tipo FILE devuelto por **fopen()**. La función **getc()** devuelve un entero; pero el byte mas significativo es cero. **getc()** devuelve **EOF** cuando se ha alcanzado el final del archivo. Por ejemplo si se quiere leer un archivo de texto hasta que se encuentre la marca de fin de archivo, se escribirla lo siguiente:

```
do {
   car = getc(fp);
} while(car !=EOF);
```

6.35 Escritura y Lectura de Cadenas

C incluye las funciones asociadas **fgets()** y **fputs()**, que leen y escriben cadenas de caracteres sobre archivos en disco. Estas funciones trabajan como **getc()** y **putc()** pero en lugar de leer o escribir un sólo carácter, leen o escriben cadenas.

Los prototipos de las funciones se encuentran en el archivo **Stdio.h** y son:

```
int fputs(char *cad, FILE *fp);

int *fgets(char *cad, int longitud, FILE *fp);
```

La función **fputs()** escribe la cadena en el archivo especificado. Si se produce un error devuelve **EOF**.

La función **fgets()** lee una cadena del archivo especificado hasta que se lee un caracter de salto de línea o hasta que se ha leído **longitud-1** caracteres.

La cadena resultante acaba con un carácter nulo. La función devuelve un puntero a cad si se ha ejecutado correctamente y un carácter nulo si se produce un error.

6.36 Fin de Archivo. Función feof().

Como ya sabemos, el sistema de archivos con buffer puede operar sobre datos binarios. Cuando se abre un archivo para entrada binaria, se puede leer un valor entero igual al de la marca **EOF** (fin de archivo). Esto podría hacer que la rutina de lectura indicase una condición de fin de archivo aún cuando el fin físico del mismo no se haya alcanzado. Para resolver éste problema, en C se incluye la función **feof()** que determina cuando se ha alcanzado el fin de archivo leyendo datos binarios. El prototipo de esta función se encuentra en el archivo **Stdio.h** y es:

```
int feof(FILE *fp);
```

La función devuelve cierto si se ha alcanzado el final de archivo; en otro caso, devuelve 0. El siguiente ejemplo lee un archivo binario hasta que se encuentre el final del mismo:

```
while(!feof(fp)) car = getc(fp);
```

Se puede aplicar éste método tanto a archivos de texto como archivos binarios.

6.37 Función rewind()

La función **rewind()** inicializa el indicador de posición, al principio del archivo, indicado por su argumento. Esto es, **rebobina** el archivo.

El prototipo de la función se encuentra en el archivo **Stdio.h** y es:

```
void rewind(FILE *fp);
```

donde **fp** es un puntero a un archivo válido.

6.38 Función ferror()

La función **ferror()** determina si se ha producido un error en una operación sobre un archivo.

El prototipo de la función se encuentra en el archivo **Stdio.h** y es:

```
int ferror(FILE *fp);
```

en donde **fp** es un puntero a un archivo válido.

La función devuelve cierto si se ha producido un error durante la última operación sobre el archivo; en caso contrario, devuelve falso.

Debido a que cada operación sobre un archivo actualiza la condición de error, se debe llamar a la función ferror() inmediatamente después de cada operación de este tipo; si no se hace así, el error puede perderse.

6.39 Función remove()

La función remove() borra el archivo especificado. Su prototipo se encuentra en el archivo de cabecera Stdio.h y es:

```
int remove(char *nombre_archivo);
```

La función devuelve cero si tiene éxito. Si no, devuelve un valor distinto de cero.

6.40 Función fflush()

Si se desea vaciar el contenido de una secuencia de salida, se puede utilizar la función fflush(). El prototipo de la función es:

```
int fflush(FILE *fp);
```

Esta función escribe todos los datos almacenados en el buffer sobre el archivo asociado con fp.

Si se llama a la función fflush() con un puntero nulo se vacían los buffers de todos los archivos abiertos.

La función devuelve 0 si tiene éxito; en otro caso, devuelve EOF.

6.41 Funciones fread() y fwrite()

Para leer y escribir tipos de datos que ocupan más de un byte, el sistema de archivo de C del ANSI suministra dos funciones, fread() y fwrite(). Estas funciones permiten la lectura y escritura de bloques de cualquier tipo de datos. Sus prototipos son:

```
size_t  fread(void *buffer, size_t número_bytes,
              size_t  cuenta,FILE *fp);

size_t  fwrite(void *buffer, size_t número_bytes,
               size_t  cuenta,FILE *fp);
```

En **fread()**, **buffer** es un puntero a una región de memoria donde se pueden escribir los datos leídos del archivo.

En **fwrite()**, **buffer** es un puntero a la información que va a ser escrita en el archivo.

El número de bytes a leer o escribir se especifica con **número_bytes**.

El argumento **cuenta** determina cuántos elementos (cada uno de **número_byes** de longitud) se van a leer o a escribir.

Es importante recordar que el tipo **size_t** (más o menos el mismo que el tipo **unsigned**) al igual que los prototipos de estas funciones están definidos en el archivo de cabecera **Stdio.h**.

fp es un puntero a un archivo abierto con anterioridad.

La función **fread()** devuelve un número de elementos leídos. Este valor puede ser menos que **cuenta** si se encuentra el final del archivo o si se produce un error.

La función **fwrite()** devuelve el número de elementos escritos. Este valor será igual a **cuenta** a menos que se produzca un error.

Las dos funciones pueden leer y escribir cualquier tipo de información, siempre y cuando el archivo se haya abierto para operaciones con datos binarios.

6.42 E/S de Acceso Directo. Función fseek().

Si se utiliza el sistema de E/S con buffer se pueden realizar operaciones de lectura y escritura **directa** con la ayuda de la función **fseek()**, la cual, sitúa el indicador de posición del archivo. El prototipo de la función es:

```
int fseek(FILE *fp, long numbytes, int origen);
```

Aquí, **fp** es un puntero al archivo devuelto por una llamada a la función **fopen()**, **numbytes** que es un entero largo, es el número de bytes a partir del **origen** que supondrá la nueva posición y **origen**, es una de las macros definidas en el archivo de cabecera **stdio.h** que se muestran en la figura 6.7.

De esta manera, para situarse a **numbytes** desde el principio del archivo, **origen** debe ser **SEEK_SET**. Para situarse a partir de la posición actual se utiliza **SEEK_CUR** y para situarse a partir del final del archivo se utiliza **SEEK_END**.

La función **fseek()** devuelve 0 cuando ha tenido éxito y un valor distinto de 0 cuando se produce un error.

Origen	Nombre de la Macro
Principio de Archivo	SEEK_SET
Posición Actual	SEEK_CUR
Fin de Archivo	SEEK_END

Figura 6.7

La función se puede utilizar para recorrer el archivo en múltiplos del tamaño de cualquier tipo de datos, simplemente multiplicando el tamaño de los datos por el número del elemento que se quiere alcanzar. Por ejemplo, si se tiene un archivo con una lista de direcciones, el siguiente fragmento de código hace que se vaya a la décima dirección.

```
fseek(fp, 9 * sizeof(struct tipo_lista), SEEK_SET);
```

6.43 Funciones fprintf() y fscanf()

Las funciones **fprintf()** y **fscanf()** se comportan exactamente como las funciones **printf()** y **scanf()**, excepto en que operan sobre archivos. Los prototipos de estas funciones son;

```
int fprintf(FILE *fp, const char *cadena_control, ...);

int fscanf(FILE *fp, const char *cadena_control, ...);
```

donde **fp** es un puntero al archivo devuelto por un llamada a la función **fopen()**. Las funciones **fprintf()** y **fscanf()** dirigen sus operaciones de E/S al archivo al que apunta **fp**.

La función **fprintf()** devuelve el número de caracteres escritos. En cambio la función **fscanf()** devuelve el número de argumentos que han sido leídos y asignados. Si el valor devuelto es un cero, significa que no se han asignado valores y si es un **EOF**, significa que se ha detectado el final del archivo.

A menudo las dos funciones son la forma más fácil de escribir y leer datos ordenadamente en archivos de disco, pero no siempre son los más eficientes.

Con la función **fprint()** los caracteres son almacenados uno por byte y los números enteros y reales en lugar de ocupar 2, 4 u 8 bytes dependiendo del tipo, requieren de un byte por cada dígito. Por ejemplo el número 105.56 ocuparía 6 bytes. Cuando la cantidad de datos numéricos a almacenar es grande, esta función no es la idónea ya que se ocupa mucho espacio en disco.

Debido a que los datos se escriben con formato ASCII como aparecerían en la pantalla (en lugar de su representación binaria), se produce una sobrecarga extra en cada llamada. Por esto, si lo importante es la velocidad o el tamaño del archivo, se debe utilizar las funciones fread() y fwrite().

6.44 Archivos Estándar

Siempre que se empieza la ejecución de un programa en C, cinco archivos, que se corresponden con dispositivos, son abiertos automáticamente. En la figura 6.8 se nombran éstos archivos y los dispositivos asociados por defecto.

Nombre	Dispositivo
stdin	Dispositivo de entrada estándar (teclado).
stdout	Dispositivo de salida estándar (pantalla).
stderr	Dispositivo de errror estándar (pantalla).
stdaux	Dispositivo auxiliar estándar (puerto serie).
stdprn	Dispositivo de impresión estándar (impresora paralelo).

Figura 6.8

De éstos cinco, dos de ellos, el dispositivo serie y el dispositivo de impresión paralelo, dependen de la configuración de la máquina, por lo tanto pueden no estar presentes.

Estos nombres están definidos como punteros a archivos (son constantes no variables), y pueden ser utilizados en cualquier función que requiera como argumento, un puntero a un archivo. Por ejemplo, se puede usar la función fputs() para mostrar una cadena utilizando una llamada como ésta:

```
fputs("este es un ejemplo", stdout);
```

Siempre se debe tener presente que **stdin**, **stdout** y **stderr** no son variables en el sentido normal y no se les debe asignar un valor mediante la función **fopen()**. Además, de la misma manera que se crean estos punteros a archivos automáticamente al empezar la ejecución del programa, se cierran automáticamente al finalizar la misma; no es recomendable intentar cerrarlos ya que se pueden acarrear problemas.

6.45 Redirección de Archivos Estándar. Función freopen().

La entrada y salida estándar, podrán ser redefinidas utilizando los símbolos <, >, >> o |.

En entornos en los que se permite la redirección de la E/S puede hacerse que **stdin** y **stdout** sean asignados a otros dispositivos, además del teclado y la pantalla. Enseguida se presenta un ejemplo de la redirección de E/S.

```
/* Programa EJEMPLO */

#include "stdio.h"

void main(void)
{
   char cad[80];

   printf("Introduzca una cadena: ");
   gets(cad);
   printf(cad);
}
```

Si se ejecuta el programa **EJEMPLO**, normalmente se imprime el cursor en la pantalla, se lee una cadena del teclado e imprime esta cadena en la pantalla. Sin embargo, en un entorno que permite la redirección de E/S bien **stdin**, bien **stdout** o ambos podrán ser reasignados a un archivo. Por ejemplo, al ejecutar el programa **EJEMPLO** en un entorno DOS o OS/2 de la siguiente forma:

```
EJEMPLO > SALIDA
```

hace que la salidad de EJEMPLO sea escrita en un archivo llamado SALIDA. Pero si se ejecuta el programa **EJEMPLO** de la siguiente forma:

```
EJEMPLO < ENTRADA > SALIDA
```

Se direcciona **stdin** a un archivo llamado ENTRADA y se envía su salida a un archivo llamado SALIDA.

Cuando finaliza la ejecución de un programa en C todas las secuencias redirigidas se inicializan a sus valores por omisión.

Se pueden redirigir los archivos estándar utilizando la función **fropen()**. Esta función asocia un archivo con un archivo nuevo. De esta manera, puede utilizarse para asociar un archivo estándar con un archivo nuevo. Su prototipo es:

```
FILE *freopen(const char *nombre_archivo, const char *modo,
              FILE *secuencia);
```

donde **nombre_archivo** es un puntero al nombre del archivo que se quiere asociar con el archivo apuntado por **secuencia**. El archivo se abre usando el valor **modo**, que puede tener los mismos valores que los utilizados por la función **fopen()**.

Acontinuación se presenta un programa que utiliza la función **freopen()** para redirigir a **stdout** a un archivo llamado SALIDA.

```
/* Programa para Redirigir a stdout */

#include "stdio.h"

void main(void)
{
   char cad[80];

   freopen("SALIDA", "w", stdout);
   printf("Introduzca una cadena: ");
   gets(cad);
   printf(cad);
}
```

En ocasiones es útil redirigir los archivos estándar; Esto puede servir principalmente a procesos de depuración. Sin embargo, no es tan eficiente realizar E/S en disco utilizando **stdin** y **stdout** redirigidos, como utilizar las funciones **fread()** y **fwrite()**.

CAPITULO VII

TIPOS DE DATOS CONGLOMERADOS

TIPOS DE DATOS CONGLOMERADOS

El lenguaje C soporta clases de tipos conglomerados, como es el caso de las estructuras, uniones y enumeraciones. Las tres, proporcionan los medios adecuados para el manejo de agrupaciones de variables diferentes y a la vez relacionadas.

Es importante mencionar que al crear estructuras, uniones y enumeraciones, también se definen **tipos de datos a medida**, la cual, es una característica muy importante del lenguaje C, ya que con ello se dá la posibilidad de crear tipos de datos propios.

7.1 Estructuras

Una **estructura** es una colección de variables que se referencían bajo un único nombre, proporcionando un medio eficaz de mantener junta la información relacionada. Las estructuras son tipos de datos **conglomerados** porque se componen de distintas variables, aunque relacionadas en forma lógica.

Una **definición de estructura** forma una plantilla que puede utilizarse para crear **variables de estructura**. A las variables que componen la estructura se les llaman **elementos de la estructura**.

La forma general de una definición de estructura es:

```
struct tipo nombre_estructura{
       tipo elemento_1;
       tipo elemento_2;
       tipo elemento_3;
       .
       .
       .
       tipo elemento_N;
       } variables_estructura;
```

Donde se puede omitir ya sea el nombre del tipo de la estructura o el nombre de la variable de estructura, pero no ambas.

La palabra clave **struct** indica al compilador que se está definiendo una plantilla de una estructura.

Los elementos de una estructura pueden ser variables ordinarias, punteros, arrays, e incluso otras estructuras. Además los nombres de ellos debe ser todos diferentes; pero el nombre de un elemento puede ser el mismo que el de una variable definida fuera de la estructura.

Cuando una estructura es un elemento de otra estructura, se le denomina **estructura anidada**. El estándar ANSI especifica que las estructuras han de poder anidarse al menos hasta 15 niveles.

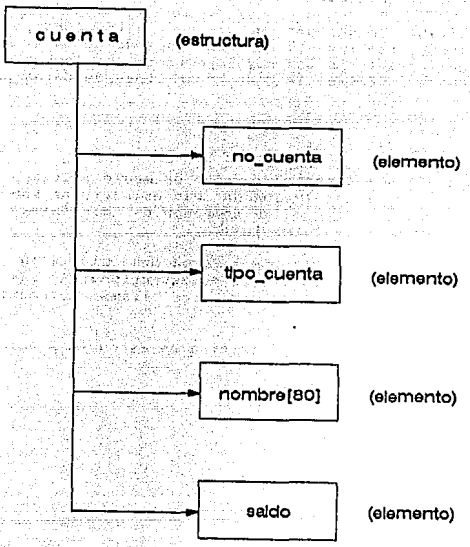

cuenta
(estructura)
no_cuenta
(elemento)
tipo_cuenta
(elemento)
nombre[80]
(elemento)
saldo
(elemento)

Figura 7.1

El código que sigue, muestra como crear (definir) una plantilla de estructura:

```
struct cuenta{
        int   no_cuenta;
        char  tipo_cuenta;
        char  nombre[80];
        float saldo;
};
```

Esta estructura recibe el nombre de **cuenta**, la cual consta de cuatro elementos que son un entero (no_cuenta), un carácter (tipo_cuenta), un array de 80 caracteres (nombre[80]), y una cantidad de coma flotante (saldo). En la figura 7.1 se muestra en forma esquemática, la forma de la estructura cuenta.

En el código no se han declarado variables de estructura, sólo se han definido la forma de los datos. Para declarar una variable de estructura llamada **clientes** de tipo **cuenta** se escribe la siguiente declaración:

```
struct cuenta clientes;
```

Cuando se define una estructura, se está definiendo esencialmente un tipo complejo de variable, formada por diferentes elementos. No existe realmente una variable de ese tipo hasta que se declara.

El compilador de C dispone automáticamente de la suficiente memoria para acomodar todas las variables que componen una **variable estructura**.

En la figura 7.2 se muestra la cantidad en bytes que ocupa en memoria la variable estructura **clientes**.

Miembro de la Estructura	Número de Bytes
no_cuenta	2
tipo_cuenta	1
nombre	80
saldo	4
T O T A L	87

Figura 7.2

Es posible declarar una o más variables de estructura a la vez que se define. Por ejemplo:

```
struct cuenta{
       int   no_cuenta;
       char  tipo_cuenta;
       char  nombre[80];
       float saldo;
} clie1, clie2, clie3;
```

Aquí se define un tipo de estructura llamada **cuenta** y se declaran las variables **clie1**, **clie2** y **clie3** de dicho tipo.

Si se tiene el caso de que sólo se necesitara una variable de estructura, no sería necesario incluir el nombre de la estructura, tal y como lo muestra el siguiente código:

```
struct {
       int   no_cuenta;
       char  tipo_cuenta;
       char  nombre[80];
       float saldo;
} cuen_clie;
```

Una estructura puede ser definida como un elemento de otra estructura. En tales situaciones, la definición de la estructura interna debe aparecer antes que la definición de la estructura externa. Por ejemplo:

```
struct fecha{
       int mes;
       int dia;
       int año;
};

struct cuenta{
       int   no_cuenta;
       char  tipo_cuenta;
       char  nombre[80];
       float saldo;
       struct fecha ultimo_pago;
} clie1, clie2, clie3;
```

La estructura **cuenta** contiene ahora otra estructura de tipo **fecha**, como uno de sus elementos. En la figura 7.3 se muestra en forma esquemática la composición de la estructura **cuenta**.

A los elementos de una variable de estructura se le pueden asignar valores iniciales, prácticamente de la misma forma que a

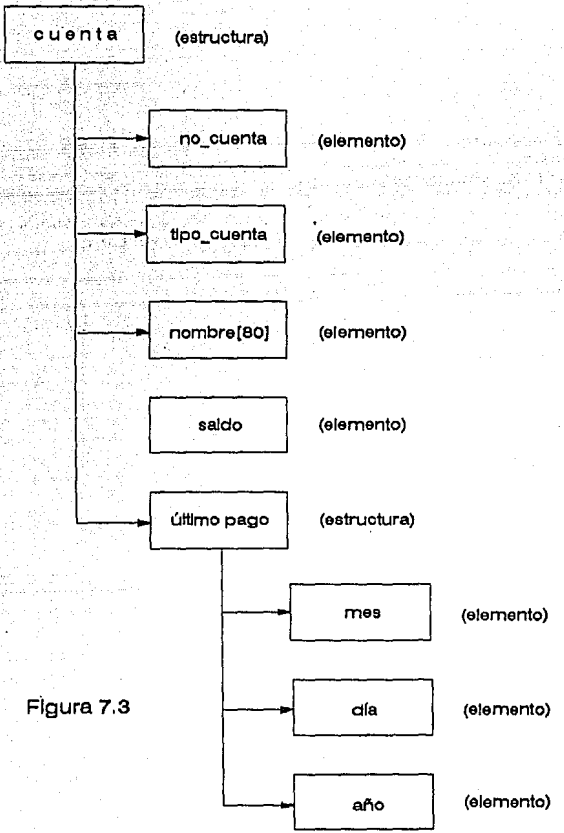

cuenta
(estructura)
no_cuenta
(elemento)
tipo_cuenta
(elemento)
nombre[80]
(elemento)
saldo
(elemento)
último pago
(estructura)
mes
(elemento)
día
(elemento)
año
(elemento)

Figura 7.3

los arrays; éstos valores deben aparecer en el órden en que serán asignados a los correspondientes elementos de la estructura, encerrados entre llaves y separados por comas. La forma general es:

```
tipo struct nombre variable = {valor1, valor2, ..., valor n };
```

En donde **valor1** se refiere al valor del primer elemento, valor2 al valor del segundo, y así sucesivamente. Una variable estructura, como un array, solamente puede ser inicializada si su tipo de alamacenamiento es **extern** o **static**.

En el siguiente ejemplo se muestra la asignación de valores iniciales a los elementos de una variable estructura:

```
/* Definición de la Estructura Tipo cuenta */

struct fecha{
        int mes;
        int dia;
        int año;
};

struct cuenta{
        int   no_cuenta;
        char  tipo_cuenta;
        char  nombre[80];
        float saldo;
        struct fecha ultimo_pago;
};

/* Asignación de Valores Iniciales */

static  struct  cuenta  cliente =
                    {12345, 'R', "Juan Olguin", 1000.00, 5, 5, 92};
```

cliente es una variable de estructura estática del tipo cuenta, cuyos elementos tienen asignados valores iniciales. El primer elemento (no_cuenta) tiene asignado el valor entero 12345, el segundo (tipo_cuenta) tiene asignado el carácter 'R', el tercer elemento (nombre[80]) tienen asignada la cadena "Juan Olguin", el cuarto elemento (saldo) tiene asignado el valor en coma flotante 1000.00, y el último elemento que es una estructura que contiene tres elementos enteros (mes, dia y año), tiene asignado los valores enteros 5, 5 y 92.

7.2 Array de Estructuras

También es posible definir un array de estructuras, esto es, un array en el que cada elemento sea una estructura. Por ejemplo, si

se trabaja con la definición de la estructura tipo cuenta:

```
struct cuenta cliente[100];
```

En esta declaración **cliente** es un array de 100 estructuras. Cada variable estructura **cliente** es una estructura del tipo **cuenta**.

Es importante hacer notar que en la estructura tipo **cuenta** se incluye un array (**nombre[80]**) y otra estructura (**fecha**) como elementos. Así, se tiene un array y una estructura incluidos dentro de otra estructura, que es a su vez un elemento de un array.

Un array de estructuras puede tener asignados valores iniciales, como cualquier otro array. Sólo que hay que recordar que cada elemento del array es una estructura a la que debe asignarse su correspondiente conjunto de valores iniciales como se ilustra en el siguiente ejemplo:

```
struct personal{
       char nombre[80];
       int mes;
       int día;
       int año;
};

static struct personal ingreso[] = {"Juan", 3, 3, 90,
                                    "Jose", 7, 25, 91,
                                    "Jorge", 12,12, 91};
```

ingreso es un array de estructuras cuyo tamaño está sin especificar. Los valores iniciales definirán el tamaño del array y la cantidad de memoria necesaria para almacenar el array.

Hay que hacer notar que cada fila en la declaración de la variable contiene cuatro constantes, las cuales corresponden a **nombre**, **mes**, **día** y **año**, para un elemento del array; como existen tres filas (tres conjuntos de constantes), el array contendrá tres elementos numerados del 0 al 2.

7.3 Elementos de una Estructura

Los elementos de una estructura se procesan generalmente de modo individual, como entidades separadas. Por tanto, es necesario acceder a estos elementos de la siguiente forma:

```
variable_estructura.elemento
```

La **variable_estructura** se refiere al nombre de una variable de un tipo de estructura y **elemento** es el nombre de un elemento dentro de la estructura. El punto que los separa es el denominado

operador punto; el cual, es un miembro del grupo de mayor prioridad de operadores y su asociatividad es de izquierda a derecha.

Si tenemos declarada la siguiente estructura:

```
struct cuenta cliente;
```

y quisiéramos acceder al número de cuenta (no_cuenta) del cliente se accedería de la siguiente forma:

```
cliente.no_cuenta
```

al igual que se accediera al saldo:

```
cliente.saldo
```

En la figura 7.4 se presenta la interpretación que se les dá a algunas expresiones que involucran elementos de una estructura.

Expresión	Interpretación
++cliente.saldo	Incrementa el valor de cliente.saldo
cliente.saldo++	Incrementa el valor de cliente.saldo después de acceder a su valor.
--cliente.no_cuenta	Decrementa el valor de cliente.no_cuenta
&cliente	Accede la dirección de comienzo de cliente
&cliente.no_cuenta	Accede la dirección de cliente.no_cuenta

Figura 7.4

Se pueden escribir expresiones más complejas usando repetidamente el operador punto. Como por ejemplo, si un elemento de una estructura es a su vez otra estructura, entonces el elemento de la estructura más interna puede ser accedido de la

siguiente manera:

```
variable_estructura.elemento.subelemento
```

donde como es lógico, **elemento** se refiere al nombre del elemento dentro de la estructura externa y subelemento al nombre del elemento dentro de la estructura interna. Por ejemplo, si el último elemento de la variable estructura **cliente** es **cliente.ultimopago**, y éste a su vez es una estructura del tipo fecha, para poder acceder al elemento mes se escribiría lo siguiente:

```
cliente.ultimopago.mes
```

Ahora, si un elemento de una estructura es un array, entonces el elemento individual del array puede ser accedido escribiendo:

```
variable_estructura.elemento[expresión]
```

Donde **expresión** es un valor no negativo que indica el elemento del array.

El uso del operador punto puede extenderse a **arrays de estructuras** usando:

```
array[expresión].elemento
```

donde **array** se refiere el nombre del array y **array[expresión]** es un elemento individual del array (es decir, una variable estructura). Por tanto, un **array[expresión].elemento** se referirá un elemento específico dentro de una estructura particular.

Por ejemplo, si se declara un array de estructuras (cliente) del tipo cuenta:

```
struct cuenta cliente[1000];
```

Para acceder a la cuenta del cliente 14 (en realidad 13, ya que el índice empieza de 0), se debería escribir:

```
cliente[13].no_cuenta
```

Y para acceder al saldo y el nombre:

```
cliente[13].saldo
cliente[13].nombre
```

Para acceder al octavo carácter dentro del nombre:

```
cliente[13].nombre[7]
```

Para acceder al mes, día y año del último pago del cliente 14:

```
cliente[13].ultimopago.mes
cliente[13].ultimopago.día
cliente[13].ultimopago.año
```

Para producir un incremento en el valor del día:

```
++cliente[13].ultimopago.día
```

7.4 Asignaciones de Estructuras

En algunas de las versiones más antiguas de C, las estructuras debían ser procesadas elemento por elemento. Con esta restricción, la única operación permisible con la estructura completa es tomar su dirección. Sin embargo, la mayoría de las nuevas versiones permiten asignar una estructura completa a otra siempre que las estructuras tengan la misma composición. Esta característica se incluye en el nuevo estándar ANSI.

Para lograr esto, primero se tendrían que declarar variables de estructura de un mismo tipo:

```
struct cuenta clientes, clientas;
```

Hay que dar por hecho que a todos los elementos de **clientes** se les ha asignado valores individuales. Para copiar estos valores a **clientas** simplemente se escribe la siguiente expresión:

```
clientas = clientes;
```

Cuando se trabaja con las antiguas versiones de C, se requiere que los valores sean copiados elemento a elemento, como por ejemplo:

```
clientas.no_cuenta = clientes.no_cuenta;
clientas.ultimopago.mes = clientes.ultimopago.mes;
```

7.5 Paso de Estructuras a Funciones

Existen varias maneras de pasar información de una estructura a/o desde una función. Se pueden transferir los elementos individuales o las estructuras completas. El mecanismo para realizar la transferencia varía, dependiendo del tipo de transferencia (elementos individuales o estructuras completas) y

de la versión particular de C.

Los elementos individuales de estructura se pueden pasar a una función como argumentos en la llamada a la función y un elemento de una estructura puede ser devuelto mediante la sentencia **return**. Para hacer esto, cada elemento de la estructura se trata como una variable ordinaria.

Cuando se pasa un elemento de una variable de estructura a una función se está realmente pasando el valor de ese elemento. Por tanto, se está pasando una variable simple. Por ejemplo:

```
struct pla{
       char x;
       int y;
       float z;
       char w[10];
} ejem;
```

De la estructura anterior se pueden pasar los elementos a la función **func** de la siguiente manera:

```
func(ejem.x);
func(ejem.y);
func(ejem.z);
func(ejem.w);       /* Aquí en vez de pasar el valor, pasa la
                       dirección de la cadena w. */
func(ejem.w[2]);    /* Aquí sólo pasa el valor del carácter de
                       w[2]. */
```

En caso de que se quiera pasar la **dirección** de elementos individuales de la estructura se deberá colocar el operador & antes del nombre de la estructura. Por ejemplo:

```
func(&ejem.x);
func(&ejem.y);
func(&ejem.z);
func(ejem.w);
func(&ejem.w[2]);
```

Siempre es importante, colocar el operador & antes del nombre de la estructura y no al nombre del elemento individual. Hay que hacer notar que el elemento de cadena **w** ya significa una dirección, por lo que no requiere del operador &.

Una estructura completa puede transferirse a una función pasando un puntero a la estructura como un argumento. En principio es similar al procedimiento usado para transferir un array a una función. Sin embargo, se debe usar una notación explícita para representar que una estructura se pasa como un argumento.

Es importante tener muy en claro que una estructura pasada de esta manera será pasada por **referencia** en vez de por valor. De aquí, si cualquiera de los elementos de la estructura es alterado

dentro de la función, las alteraciones serán reconocidas fuera de la misma.

7.6 Punteros a Estructuras

El lenguaje C permite punteros a estructuras al igual que permite punteros a cualquier otro tipo de variables. Sin embargo, hay algunos aspectos especiales que afectan a este tipo de punteros que se deben conocer.

Al igual que los demás punteros, los punteros a estructuras se declaran poniendo un * delante del nombre de la variable estructura. Por ejemplo, si se tiene definida la estructura:

```
struct dir {
       char nombre[30];
       char calle[40];
       char ciudad[20];
       char estado[3];
       unsigned long int codigo_pos;
};
```

Se puede declarar **puntero_dir** como un puntero a un dato de ese tipo:

```
struct dir *puntero_dir;
```

Existen dos usos principales de los punteros a estructuras: uno es para generar una llamada por referencia a una función y otra para crear listas enlazadas y otras estructuras de datos dinámicas utilizando el sistema de asignación de C.

Existe un importante inconveniente en el paso de cualquier estructura a una función, excepto para las más simples y es el esfuerzo extra que se necesita para introducir (y sacar) todos los elementos de la estructura en (y de) la pila.

Para estructuras simples con pocos elementos, el esfuerzo no es demasiado significativo. Sin embargo, si se utilizan varios elementos o si algunos de ellos son array, el tiempo de ejecución puede degradarse a niveles inaceptables. La solución a este problema es pasar a la función sólo el puntero.

Cuando se pasa a una función un puntero a una estructura, sólo se guarda (o se recupera) en (o de) la pila la dirección de la estructura. Esto significa que la llamada a la función es muy rápida.

Una segunda ventaja, en algunos casos, se encuentra cuando la función ha de referenciar la estructura real y no una copia. Pasando un puntero, se puede modificar los contenidos de los elementos reales de la estructura utilizada en la llamada.

Las siguientes líneas de código muestran el uso de punteros a estructuras:

```
struct ejem{
       int edad;
       char nombre[40];
} persona;

struct ejem *p;      /* Se declara un puntero a la estructura */

  . . .

p = &persona;        /* Se coloca la dirección de la estructura
                        persona en el puntero p.
```

Para acceder a los elementos de una estructura usando un puntero a esa estructura, se usa el operador ->. Por ejemplo, para referenciar el campo **edad**, se escribe:

```
p->edad
```

La -> se denomina usualmente **operador flecha** y consiste en un signo menos seguido de un signo de mayor. Cuando se accede a un elemento de una estructura a través de un puntero a la estructura, se usa la flecha en lugar del operador punto.

Ahora se presenta un programa que muestra el uso de un puntero a una estructura. Este programa imprime en la pantalla las horas, los minutos y los segundos utilizando un retardador de software.

```
/* Visualización de un temporizador software */

#include "stdio.h"

struct mi_hora{                    /* Aquí se define la
       int horas;                     plantilla de la estructura */
       int minutos;
       int segundos;
};

void mostrar(struct mi_hora *t);        /* Definición de
void actualizar(struct mi_hora *t);        funciones */
void retardo(void)

void main(void)
{
   struct mi_hora tiempo;

   tiempo.horas = 0;
   tiempo.minutos = 0;
   tiempo.segundos = 0;
```

```
    for(;;){
      actualizar(&tiempo);
      mostrar(&tiempo);
    }
}



void actualizar(struct mi_hora *t)
{
   t->segundos++;
   if(t->segundos==60){
      t->segundos = 0;
      t->minutos++;
   }
   if(t->minutos==60){
      t->minutos = 0;
      t->horas++;
   }
   if(t->horas==24) t->horas = 0;
   retardo();
}



void retardo(void)
{
   long int t;

   for(t=1; t<128000; t++) ;
}



void mostrar(struct mi_hora *t)
{
   printf("%02d: ", t->horas);
   printf("%02d: ", t->minutos);
   printf("%02d: ", t->segundos);
}
```

En el programa se declara una estructura global llamada **mi_hora** pero no se declaran variables. Dentro de **main()** se declara la variable estructura **tiempo** y se inicializa a 00:00:00. Esto significa que sólo la función **main()** conoce directamente la estructura **tiempo**.

A las funciones de **actualizar()** (que cambia la hora) y **mostrar()** (que imprime la hora) se les pasa la direccion de **tiempo**. En ambas funciones se ha declarado el argumento como un puntero a una estructura de tipo **mi_hora**, lo que permite saber cómo referenciar a los elementos de la estructura. Por ejemplo, para poner a cero la hora cuando el temporizador alcance

24:00:00, se escribiría lo siguiente:

```
if(t->horas==24) t->horas = 0;
```

Esta línea de código indica al compilador que tome la dirección de t (que es tiempo en main()) y asigne 0 a su elemento **horas**.

7.7 Campos de Bits

A diferencia de otros lenguajes, C tiene un método incorporado para acceder a un bit individual dentro de un byte. Esto podría ser de gran ayuda por éstas razones:

a) Si el almacenamiento es limitado, se pueden almacenar varias variables **lógicas** (verdadero/falso) en un byte.

b) Ciertos dispositivos transmiten la información codificada en los bits dentro de bytes.

Aunque todas estas tareas se pueden realizar utilizando los bytes y los operadores a nivel de bits, un campo de bits realza la estructura (y posiblemente la eficiencia) del código.

El método que se utiliza en C para acceder a los bits está basado en la estructura.

Un **campo de bits** es realmente un tipo especial de elemento de estructura que define su tamaño en bits. La forma general de definición de un campo de bits es:

```
struct etiqueta{
      tipo nombre_1:longitud;
      tipo nombre_2:longitud;
        .
        .
        .
      tipo nombre_N:longitud;
} lista_variables;
```

Cada uno de los campos de bit debe declararse como **int**, **unsigned** o **signed**. Sin embargo, cuando uno de ellos es de longitud 1 deben declararse como **unsigned**, ya que un sólo bit no puede tener signo.

Los campos de bits se utilizan frecuentemente cuando se analiza la entrada de un dispositivo hardware. Por ejemplo, el puerto de estado de un adaptador de comunicaciones serie puede devolver el byte de estado organizado tal y como lo muestra la figura 7.5.

Bit	Significado cuando está activado
0	Cambio en la línea listo-para enviar.
1	Cambio en datos_listos.
2	Detección de final.
3	Cambio en la línea de recepción.
4	listo-para-enviar.
5	Datos-listos.
6	LLamada telefónica.
7	Senal recibida.

Figura 7.5

Esta información se puede representar en un byte de estado mediante el siguiente campo de bits:

```
struct tipo_estado{
       unsigned delta_cts: 1;
       unsigned delta_dsr: 1;
       unsigned tr_final:  1;
       unsigned delta_rec: 1;
       unsigned cts:       1;
       unsigned dsr:       1;
       unsigned ring:      1;
       unsigned linea:     1;
} estado;
```

Se pueden usar los campos de bits para permitir que un programa determine si se puede enviar o recibir datos:

```
if(estado.cts) printf("listo para enviar");

if(estado.dsr) printf("datos listos");
```

Para asignar un valor a un campo de bits, sencillamente se utilizaría la forma que su usa para cualquier otro tipo de elemento de estructura. Por ejemplo:

```
estado.ring = 0;
```

En estos ejemplos se usa el operador punto para acceder a cada campo de bit, sin embargo, si se referencia la estructura a través de un puntero, se usaría el operador ->.

En ocasiones no es necesario nombrar cada campo de bit, Esto facilitaría llegar al bit que se desea pasando por alto los que no se usan.

Si del ejemplo anterior sólo nos interesara los bits de cts y dsr, se podría declarar la estructura tipo_estado de la forma:

```
struct tipo_estado{
        unsigned :          4;
        unsigned cts:       1;
        unsigned dsr:       1;
} estado;
```

No es necesario especificar los bits que siguen a dsr si es que no se van a usar.

Las variables de campos de bits tienen ciertas restricciones:

a) No se puede tomar la dirección de una variable de campos de bits.

b) No se pueden construir array de variables de campos de bits.

c) No se pueden solapar los límites enteros.

d) No se puede saber, de máquina a máquina, si los campos se dispondrán de derecha a izquierda o de izquierda a derecha.

Cualquier código que use campos de bits puede tener algunas dependencias con la máquina.

Es posible mezclar elementos de estructuras normales con elementos de campos de bits. Por ejemplo:

```
struct empl{
        struct dir direccion;
        float paga;
        unsigned baja:1;          /* Baja o Activo */
        unsigned por_horas:1;     /* Por horas o en Nómina */
        unsigned deducciones:3;   /* Deducciones */
};
```

7.8 Uniones

En el lenguaje C una **unión** es una posición de memoria que es compartida por dos o más variables diferentes, generalmente de distinto tipo, en distintos momentos.

La definición de una unión es similar al de una estructura. Su forma general se presenta en la siguiente página.

```
union etiqueta{
        tipo nombre_variable;
        tipo nombre_variable;
        tipo nombre_variable;
        .
        .
        .
} variable_unión;
```

donde **union** es la palabra clave requerida y los otros términos tienen el mismo significado que en una definición de estructura.

Las variables unión se pueden declarar como lo muestra la forma general de definición de la unión o bien como:

```
tipo union etiqueta variable_1, variable_2, ... , variable_N;
```

donde **tipo** es un especificador opcional de tipo de almacenamiento, **union** es la palabra clave requerida, **etiqueta** es el nombre que aparece en la definición de unión (plantilla) y **variable_1, variable_2, ..., variable_N** son variables unión del tipo etiqueta.

Las uniones así como las estructuras contienen elementos cuyos tipos de datos pueden ser diferentes. Sin embargo los elementos que componen una unión comparte la misma área de almacenamiento dentro de la memoria de la computadora. Mientras que cada elemento dentro de una estructura tiene asignada su propia área de almacenamiento.

Así, las uniones se usan para ahorrar memoria. Son útiles para aplicaciones que impliquen múltiples elementos donde no se necesita asignar valores a todos los elementos a la vez.

Dentro de una unión, la reserva de memoria requerida para almacenar elementos cuyos tipos de datos son diferentes es manejado automáticamente por el compilador.

Cuando se declara una unión, el compilador crea una variable suficientemente grande para guardar el tipo más grande de variable de la **unión**.

Acontinuación se presenta un pequeño ejemplo de lo que es una declaración de unión:

```
union pla{
      int i;
      char ch;
} ejem;
```

En la unión **ejem**, tanto el entero **i**, como el carácter **ch** comparte la misma posición de memoria. Como sabemos los enteros

ocupan generalmente 2 bytes y los caracteres 1 byte. En la figura 7.6 se muestra la forma en que los elementos de la unión comparte la misma dirección de memoria.

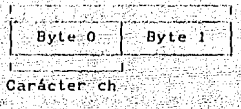


Figura 7.6

Para acceder a un elemento de la unión, se utiliza la misma sintáxis que se utiliza para las estructuras, los operadores flecha y punto.

Si se está trabajando directamente con la unión, se usa el operador punto. Si se accede a la variable unión a través de un puntero, se utiliza el operador flecha.

El uso de una unión puede ayudar a la creación de código independiente de la máquina o portable. No se producen dependencias con la máquina debido a que el compilador sigue la pista de los tamaños actuales de las variables unidas. No es necesario preocuparse por el tamaño de **int**, **char**, **float**, etc..

Las uniones se utilizan frecuentemente donde se necesitan conversiones de tipos, ya que se pueden referir a los datos de la unión de diferentes formas.

7.9 Enumeraciones

Una enumeración es un conjunto de constantes enteras con nombres que especifica todos los valores válidos que una variable de ese tipo puede tener.

Las enumeraciones se define de forma parecida a las estructuras: la palabra clave **enum** señala el comienzo de un tipo enumerado. La forma general de una enumeración es:

```
enum etiqueta {lista_nombres_constantes_enteras} lista_variables;
```

Aquí, tanto la **etiqueta** de la enumeración como la **lista de variables** son opcionales.

Como en las estructuras, aquí también se usa el nombre de la etiqueta para declarar variables de ese tipo.

En el siguiente ejemplo se define una enumeración llamada colores y se declara la variable color de ese tipo:

```
enum colores { azul, amarillo, rojo, verde, blanco,negro };
enum colores color;
```

Dada esa definición y esa declaración, la siguiente sentencia es válida:

```
color = azul;
```

Las variables de tipo **enum** son tratadas como si fuesen de tipo **int**.

Cada identificador de la lista de constantes enteras en una enumeración, tiene asociado un valor. Por defecto, el primer identificador tiene asociado el valor 0, el siguiente el valor 1, y así sucesivamente. Por lo que del ejemplo anterior **color** tendrá el valor 0 (azul = 0).

A cualquier identificador de la lista, se le puede asignar un valor inicial por medio de una expresión constante. Los identificadores sucesivos tomarán valores consecutivos. Por ejemplo:

```
enum colores
{
    azul,
    amarillo,
    rojo,
    verde = 0,
    blanco,
    negro
} color;
```

Los valores asociados a los identificadores son los siguientes:

```
azul     = 0
amarillo = 1
rojo     = 2
verde    = 0
blanco   = 1
negro    = 2
```

A los miembros de una enumeración se les aplica las siguientes reglas:

a) Dos o más miembros pueden tener un mismo valor.

b) Un identificador no puede aparecer en más de un tipo.

c) Un identificador es simplemente un nombre para un entero, no es una cadena.

7.10 typedef

El lenguaje C permite definir explícitamente un nuevo nombre de tipo usando la palabra clave **typedef**.

Realmente no se crea una nueva clase de datos, sino que se define un nuevo nombre para un tipo existente. Este proceso puede ayudar a hacer más transportables los programas con dependencias con las máquinas.

También puede ayudar en la documentación del código, permitiendo usar nombres que sean más descriptivos que los tipos de datos estándar. La forma general de la sentencia **typedef** es:

```
typedef tipo nombre;
```

donde **tipo** es cualquier posible tipo de datos y **nombre** es el nuevo nombre para ese tipo. El nuevo nombre que se define es una adición, no un reemplazamiento del nombre de tipo existente. Por ejemplo:

```
typedef float num;
```

Esta sentencia indica al compilador que se ha de reconocer la palabra **num** como otro nombre de **float**. Por lo tanto se puede escribir lo siguiente:

```
num var;
```

En el ejemplo, **var** es una variable en coma flotante de tipo **num**, que es otro nombre de **float**. También se puede hacer esto:

```
typedef num otro;
```

Aquí se le indica al compilador que ha de reconocer a **otro** como otro nombre de **num**, que es a su vez otro nombre de **float**.

El uso de **typedef** puede hacer que el código sea más fácil de leer y más fácil de transportar a una nueva máquina.

CAPITULO VIII

PREPROCESADOR DE C

El preprocesador de C es una colección de sentencias especiales, llamadas **directivas**, que son ejecutadas al principio del proceso de compilación.

Las directivas del precompilador generalmente aparecen al principio de un programa, pero pueden aparecer en cualquier parte del mismo. Además tendrá sólo aplicación en la porción de programa que sigue a su aparición.

Dentro de las principales directivas que contiene el preprocesador de C se encuentran:

a) Directiva #define
b) Directiva #error
c) Directiva #include
d) Directivas de compilación condicional:
   - Directiva #if y #endif
   - Directiva #else
   - Directiva #elif
   - Directiva #ifdef
   - Directiva #ifndef
e) Directiva #undef
f) Directiva #line
g) Directiva #error
h) Directiva #pragma

8.1 Directiva #define

La directiva **#define**, define un identificador y una cadena que sustituirá al 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_macro cadena
```

Puede existir 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.

Un ejemplo de la directiva **#define** sería:

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

Esto hace que el compilador sustituya la palabra CIERTO por el valor 1 y la palabra FALSO por un 0.

Cada vez que se define un nombre de macro, se puede usar como parte de la definición de otros nombres de macro. Por ejemplo:

```
#define UNO   1
#define DOS   UNO + UNO
#define TRES  UNO + DOS
```

La sustitución de macro es simplemente el reemplazamiento de un identificador por su cadena asociada. De esta forma, si se quisiera definir un mensaje que apareciera en nuestros programas, se escribirían las siguientes líneas:

```
#define RES  "Derechos reservados por Ricardo Ramirez\n"
#define MEN  "Error en la introducción de datos\n"

. . .

printf(RES);
printf(MEN);
```

El compilador sustituirá la cadena "Derechos reservados por Ricardo Ranmirez" o la cadena "Error en la introducción de datos", cada vez que encuentre el identificador RES o MEN respectivamente. Para el compilador, la sentencia printf() aparece realmente como;

```
printf("Derechos reservados por Ricardo Ramirez\n")
printf("Error en la introducción de datos\n")
```

En caso de que la cadena sea más larga que una línea, se puede continuar en la siguiente, poniendo una barra invertida al final de la línea. Por ejemplo:

```
#define EJEMP  "esta es una cadena muy larga \
que se usa como ejemplo"
```

Es muy común usar en programas en C, letras mayúsculas para los identificadores definidos. Este convenio permite a cualquiera que lea un programa, saber de una ojeada dónde se realizan sustituciones. Además, es recomendable poner todos los #define al principio del archivo o en un archivo separado incluído que dispersarlos por todo el programa.

Se puede tener un programa que defina un array y que tenga varias rutinas que accedan a ese array . En vez de codificar directamente el tamaño del array con una constante es mejor definir un tamaño y utilizar ese nombre siempre que se necesite.

De esta forma, para modificar el tamaño del array, si es preciso, sólo se requiere un cambio en un único sitio junto con una recompilación. Por ejemplo:

```
#define MAXIMO 100

float balance[MAXIMO];
```

Otra aplicación útil de la directiva **#define** es que el nombre de macro puede tener argumentos. Cada vez que se encuentre el nombre de macro, los argumentos asociados con él se reemplazan por los argumentos reales del programa. Por ejemplo:

```
#include "stdio.h"
#define ABS(a)   (a) < 0  ?  -(a) : (a)

void main(void)
{
   printf("valor absoluto de -1 y 1: %d %d", ABS(-1), ABS(1));
}
```

Cuando el programa se compila, la a de la definición de macro se sustituye por los valores de -1 y 1. Los paréntesis que rodean a a aseguran que se lleve a cabo adecuadamente la sustitución en todos los casos. Por lo que al ejecutar el programa se tendrá el valor absoluto de -1 y 1.

El uso de sustituciones de macros, en lugar de funciones reales, tiene principalmente una ventaja: incrementan la velocidad del código porque no se gasta tiempo en llamar a la función. Sin embargo, hay que pagar un precio por el aumento de velocidad: el tamaño del programa aumenta debido a la duplicación del código.

8.2 Directiva #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_error
```

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

8.3 Directiva #include

La directiva **#include** hace que el compilador incluya otro archivo fuente en el que tiene ésta directiva. 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 archivo en disco.

Si el nombre del archivo está entre ángulos, el archivo se busca en un directorio especial que está, sólo dispuesto, a archivos incluidos.

Si el nombre del archivo está entre comillas dobles, se busca primero en el directorio actual de trabajo. Si ahí no se encuentra el archivo, se repite la búsqueda como si estuviera entre ángulos.

Los archivos incluídos pueden contener a su vez directivas **#include**. A este tipo de situaciones se les llama **anidamiento de inclusiones**. El número de niveles de anidamiento permitido depende del compilador. El estándar ANSI estipula que al menos se deben permitir ocho niveles de anidamiento.

8.4 Directivas de Compilación Condicional

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

8.5 Directivas #if, #else, #elif y #endif

Las directivas **#if**, **#else**, **#elif**, y **#endif** permiten compilaciones condicionales del programa fuente, dependiendo del valor de una o más condiciones de verdadero/falso.

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 la directiva **#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 expresión que sigue a **#if** se evalúa en tiempo de compilación. Por lo que sólo debe contener constantes e identificadores que se hayan **definido** anteriormente. No es posible utilizar variables.

La directiva **#else** funciona de una manera muy parecida a la sentencia **else**. Esta directiva establece una alternativa para el caso de que **#if** falle.

La directiva **#elif** quiere decir si no y establece una escala del 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_2
      secuencia de sentencias
#elif expresión_3
      secuencia de sentencias
.
.
.
#elif expresión_N
      secuencia de sentencias
#endif
```

La directivas **#if** e **#elif** se puede anidar, hasta un límite que es determinado por la implementación. Por ejemplo:

```
#if MAX > 100
    #if EXP
       int var1=198;
    #elif
       int var1=200;
    #endif
#else
   char salida[100];
#endif
```

8.6 Directivas #ifdef e #ifndef

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

```
#ifdef nombre_macro
    secuencia de sentencias
#endif
```

Si se ha definido previamente el **nombre_macro** en una sentencia **#define**, se compila el bloque de código que sigue a la sentencia.

La forma general de la directiva **#ifndef** es:

```
#ifndef nombre_macro
    secuencia de sentencias
#endif
```

Si no se ha definido previamente el **nombre_macro** mediante la directiva **#define**, se compila el bloque de código.

Tanto **#ifdef** como **#ifndef** pueden usar una sentencia **#else**, pero no una sentencia **#elif**. Además las directivas se pueden anidar hasta un determinado nivel, de la misma forma que se anidan las **#if**.

8.7 Directiva #undef

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

```
#undef nombre_macro
```

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

8.8 Directiva #line

La directiva **#line** cambia los contenidos de **_LINE_** y **_FILE_**, que son identificadores del compilador predefinidos.

El identificador _LINE_ contiene el número de línea de la 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_archivo"
```

donde **número** es cualquier entero positivo que se convierte en el nuevo valor de _LINE_ y **nombre_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 para aplicaciones especiales.

8.9 Directiva #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.

8.10 Operadores del Preprocesador de C

El C del ANSI proporciona dos operadores de preprocesamiento el **#** y **##**. Estos operadores se utilizan cuando se usa una definición de macro con **#define**.

El operador **#** es utilizado solamente con **macros** que reciben argumentos. Este operador precediendo al nombre de un parámetro formal en la macro, hace que el correspondiente parámetro actual pasado en la llamada a la macro, sea incluido entre comillas para ser tratado como un literal. Por ejemplo, si tenemos:

```
#define literal(s)  printf(#s"\n")
```

y se tuviera una ocurrencia como esta:

```
literal(Pulse una tecla para continuar);
```

sería sustituida por:

```
printf("Pulse una tecla para continuar" "\n");
```

que sería equivalente a:

```
printf("Pulse una tecla para continuar\n");
```

El operador **##** al igual que el anterior, también es utilizado con **macros** que reciben argumentos. Este operador permite la concatenación de dos cadenas. Por ejemplo:

```
#define salida(n) printf("elemento " #n " = %d\n", elemento ##n)
```

Si se tuviera una ocurrencia como ésta:

```
salida(1);
```

sería sustituida por:

```
printf("elemento " "1" " = %d\n",elemento1);
```

el cual es equivalente a:

```
printf("elemento 1 = %d\n",elemento1);
```

Estos operadores en realidad se usan poco, pero existen para que el preprocesador pueda manejar cierto casos especiales.

8.11 Macros Predefinidas

El estándar ANSI especifica los siguientes cinco nombres de macros predefinidos:

```
_LINE_
_FILE_
_DATE_
_TIME_
_STDC_
```

Es posible que estos nombres de macros predefinidos no se encuentren todos en un compilador particular o que quizá falten algunos más, por lo que es preferible consultar siempre los manuales del compilador.

El uso de las macros **_LINE_** y **_FILE_** ya se discutieron en la sección dedicada a la directiva **#line**.

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 cualquier otro número es que la implementación varía de la estándar.

CAPITULO IX

APLICACION DEL LENGUAJE C

APLICACION DEL LENGUAJE C

Al trabajar para la Gerencia de Planeación Industrial de Petróleos Mexicanos se ha tenido la fortuna de participar ya sea en el análisis, diseño, desarrollo y mantenimiento de sistemas cuyos objetivos son básicamente, el llevar un perfecto control presupuestal de la S.T.I. (Subdirección de Transformación Industrial de PEMEX).

Se han realizado varios sistemas como por ejemplo, el **Sistema de Avance Físico Financiero de Obras de Inversión y Mantenimiento** cuyo principal objetivo es el de llevar el control y seguimiento de las obras de inversión y trabajos de mantenimiento de la S.T.I.; el sistema consta de los siguientes módulos:

a) Presupuesto Autorizado para la Ejecución de Obras
b) Programa de Avance Finaciero de las Obras.
c) Programa de Avance Fisico de las Obras.
d) Avance Financiero Real.
e) Avance Físico Real.

Esto permite en un determinado momento pronósticar y corregir desviaciones en los programas y ejecuciones de las obras

Otro sistema es el **Sistema de Control de Contratos** cuyo objetivo es el de llevar un estricto control de los contratos de inversión por obras, mantenimiento y otros conceptos celebrados entre la S.T.I. con diferentes compañías, por lo que respecta a las afectaciones presupuestales como el costo del contrato (compromiso), erogaciones efectuadas (devengado) y saldo por ejercer.

El beneficio que nos dá este sistema es el de tener información actualizada y controlada de acuerdo a los diferentes rangos de agrupación como lo es por contrato, proyecto, compañía, centro de trabajo, etc.

Pero el sistema de mayor peso y que trataremos de presentar sus características principales corresponde al **Sistema del Ejercicio Presupuestal** que a su vez es complementado por el **Módulo de Control de Información Financiera (C.I.F.)**.

Todos éstos sistemas fueron desarrollados con el manejador de bases de datos Informix 3.30, pero ante la complejidad e importancia de los mismos se ha tenido que recurrir a la elaboración de rutinas en C que permitiera una mejor manipulación de la información.

Es importante remarcar que el objetivo de esta tésis no es el de mostrar el análisis ni diseño del sistema del Ejercicio Presupuestal ni del módulo CIF, sino el de resaltar rutinas hechas en C y además, mostrar la forma en que se puede realizar la interface entre C y el manejador de base de datos Informix, utilizando ALL-IN C.

9.1 Sistema de Control del Ejercicio Presupuestal (S.C.E.P.) y el Módulo C.I.F.

Este sistema tiene el propósito de llevar acabo un control eficiente y oportuno del ejercicio presupuestal de la S.T.I. tanto en sus fases de:

a) Presupuesto Autorizado Devengable
b) Presupuesto Autorizado Flujo de Efectivo
c) Devengado de Rama
d) Devengado Institucional
e) Comprometido de Rama
f) Ejercicio Flujo de Efectivo (Pagado)

El menú principal del sistema consta de las siguientes opciones:

a) Captura/Actualización/Consulta
b) Reportes
c) Carga de Archivos Planos
d) Preparación de Archivos
e) Procesos Especiales

en cada una de ellas se puede explotar cualquier fase del presupuesto.

Como parte integrante del sistema fué creado el **Módulo de Control de Información Financiera**, el cual permite entre otras cosas la generación del documento denominado CIF, representado en la figura 9.1

Este documento permite registar y actualizar la información presupuestal, correspondiente a las operaciones realizadas en PEMEX y tiene un nivel de detalle tal, que se puede registrar la información en áreas de responsabilidad como centros de trabajo y departamentos.

Teniendo esta información a detalle o lo que es lo mismo a nivel de Concepto de Origen se puede agrupar las afectaciones presupuestales en forma de renglones de gasto que están ya determinados por catálogos Institucionales en PEMEX; logrando con ello, tener la información a un nivel de agrupación tal, que puede ser manipulada para la toma de desiciones al comparar lo ejercido con lo que se tiene presupuestado, tanto en forma mensual como anual.

El módulo CIF fué implantado en las diferentes unidades de control presupuestal que tiene cada una de las refinerías y en la unidad de control de México la cual se encarga exclusivamente de las oficinas centrales. Todas estas dependencias controladas por la S.T.I..

PETROLEOS MEXICANOS
CONTROL DE INFORMACION FINANCIERA
C I F

NO. DE CIF : 55

NUMERO DE DOCUMENTO FUENTE	RECIBO 005
NUMERO DE DOCUMENTO CONTABILIZADOR	
PARTIDA PRESUPUESTAL DE OBRA	
NUMERO DE PROYECTO	
CLAVE DE AUTORIZACION	A1434100034
FECHA DE VENCIMIENTO	31/01/93
IMPORTE TOTAL EN MONEDA EXTRANJERA	0.00
TIPO DE CAMBIO	0.00
MOMENTO CONTABLE	7314
CLAVE DE CENTRO DE TRABAJO AFECTADO	800

DEPARTAMENTO	CONCEPTO DE ORIGEN	IMPORTE POR DEPTO. EN M.N.SIN IVA	CONCEPTO DE ORIGEN DEL IVA	MONTO IVA
40000	320500	15888.88	929200	1588.88

IMPORTE TOTAL MONEDA NACIONAL (SIN IVA)	15888.88
IMPORTE TOTAL DEL IVA	1588.88
FECHA DE ELABORACION	15/01/93

RESPONSABLE DE CONTROL PRESUPUESTAL

FICHA	NOMBRE	FIRMA

FIGURA 9.1

En sí, el objetivo del Módulo CIF es dar cumplimiento a los siguientes puntos:

a) Creación automática del documento CIF.
b) Generación de informes de control de cifs.
c) Generación de informes de las diferentes fases presupuestales.
d) Seguimiento del control del ejercicio presupuestal local.
e) Generación de archivos planos para la concentración de información a nivel central (México) para alimentar al Sistema del Ejercicio Presupuestal.

En las primeras versiones del Módulo CIF la información generada alimentaba directamente al Sistema del Ejercicio Presupuestal en forma mensual, pero ante la liberación del complejo Sistema Institucional de Control del Ejercicio Presupuestal de PEMEX (S.I.C.E.P.) se tuvo que cambiar éste procedimiento.

S.I.C.E.P. es un sistema complejo al que están adscritos todas las subdirecciones y cuyo fin es el control del ejercicio presupuestal de todo PEMEX. El sistema contempla a todos los centros de trabajo y departamentos, por lo que las comunicaciones son extensas y el procesamiento de la información se concentra en diferentes nodos estratégicos establecidos en diferentes partes del país para después, enviar la información agrupada al nodo principal establecido en México.

El Módulo CIF mediante un paquete de comunicaciones se conecta con S.I.C.E.P. y se transmiten los registros capturados durante el día para que sean procesados, realizándose con ello el denominado **cierre diario.**

Al final de mes S.I.C.E.P. realiza el cierre mensual y se transmiten archivos planos, (uno por cada fase presupuestal) por medio de otro paquete de comunicaciones, con el fin de alimentar al Sistema del Ejercicio Presupuestal.

En la figura 9.2 se muestra la forma en que se conectan los sistemas para la obtención del ejercicio presupuestal de la S.T.I..

9.2 Manejador de Base de Datos Relacional Informix 3.30

Informix 3.30 fué creado por Relational Database Systems Inc. en noviembre de 1984, y es utilizado para el desarrollo de los sistemas en la S.T.I..

Informix está compuesto por las siguiente utilerías:

a) dbbuild
b) dbstatus
c) Informer
d) Enter2
e) Ace Report Write
   acego - aceprep
f) perform - formbuild

CONEXION DE SISTEMAS PARA LA OBTENCION DEL EJERCICIO PRESUPUESTAL DE LA S. T. I.

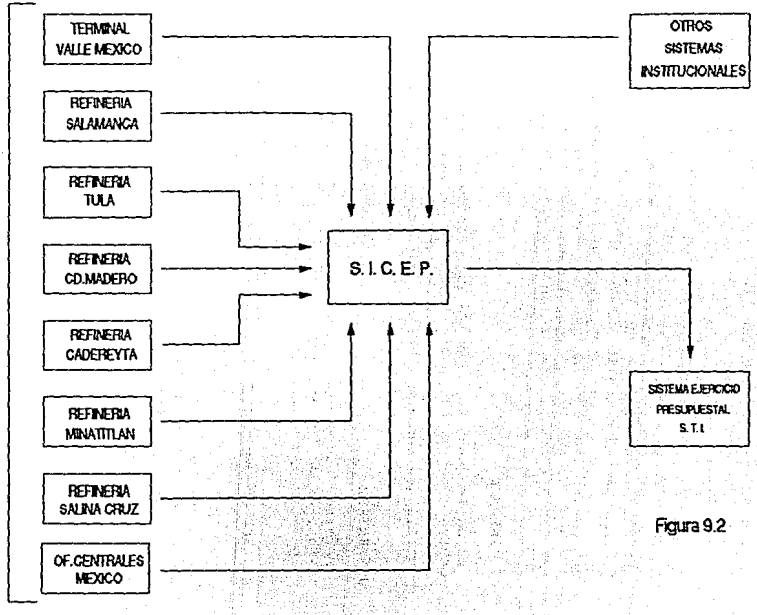


Figura 9.2

a) DBBUILD

Es usado para crear la base de datos, además se puede utilizar también para cambiar la estructura de la misma ya sea para adicionar, borrar, acrecentar o acortar campos de archivos, o incluso adicionar nuevos archivos para que formen parte de la base de datos.

b) DBSTATUS

Es una herramienta diseñada tanto para el programador como para el administrador de la base de datos ya que se puede añadir o borrar índices en archivos, imprimir el esquema de los archivos, borrar archivos o incluso la B.D. completa, generación de archivos planos con o sin delimitadores, chequeo e impresión de estado de archivos, reparación de archivos, etc.

c) INFORMER

Es un lenguaje de búsqueda que permite al usuario, examinar un archivo o partes específicas de mismo. Además se pueden relacionar dos o más archivos para presentar registros en la pantalla que cumplan ciertas condiciones.

d) ENTER2

Permite adicionar, suprimir y actualizar registros de cualquiera de los archivos de una base de datos sin la necesidad de definir algún formato de pantalla.

e) ACE REPORT WRITE

Es similar a informer, se pueden relacionar dos o más archivos dependiendo de ciertas condiciones, y a los registros resultantes se les puede dar un determinado formato para presentar informes con una adecuada presentación.

En Report Write hay instrucciones para clasificar los registros (tomando como referencia uno o más campos ya sea en forma ascendente o descendente), declarar variables útiles en el programa, definir parámetros de la salida (como son los márgenes derecho, izquierdo, etc.), títulos de cabecera o pies de página, y presentación de totales o elaboración de procesos antes o después de determinados grupos de un campo específico.

Después de realizar un programa (utilizando cualquier editor de texto) que especifique sentencias de Report Write, se compila utilizando la utilería ACEPREP y el nombre del programa. Esto generará un archivo con el mismo nombre, pero con la extensión ARC. Después para ejecutar dicho programa es necesario correr la utilería ACEGO y el nombre del programa.

f) FORMBUILD

Se utiliza para compilar un archivo (realizado por cualquier editor de texto) que especifica el formato de una pantalla de captura, actualización o consulta de registros de uno o varios archivos relacionados. Al compilar un archivo de éste tipo se obtiene un archivo con extensión .FRM. Para ejecutar el programa es necesario la utileria PERFORM y el nombre del archivo.

9.3 Interface ALL-II C

ALL-II C es un conjunto de librerías que permiten en un momento dado la manipulación de datos de archivos propios de Informix. En otras palabras, utilizando estas librerías en un programa en C, se permite actualizar, adicionar y borrar datos de cualquier archivo del formato típico de Informix.

Para que estas librerías puedan correr satisfactoriamente es necesario incluir en los programas de aplicaciones, los siguientes archivos de cabecera:

a) perform.h

b) dbio.h

c) stdio.h

Estos permiten establecer el acceso a los archivos propios de Informix.

En Posteriores páginas se mostrará el código de éstos archivos de cabecera.

9.4 Funciones de Biblioteca RDS

Como ya lo hemos mencionado, dentro de ALL-II existe una gran variedad de librerías que permiten en un momento dado, manipular la información contenida en archivos propios de informix.

Por lo tanto, se presentan acontinuación las librerías más utilizadas en los procesos especiales del Sistema del Ejercicio Presupuestal y en la pantalla principal del Módulo C.I.F..

En cada rutina se presenta el tipo y definición de sus parámetros, así como una breve descripción del objetivo de la misma.

9.5 Rutina dbselect()

```
dbselect(flag, name)
int flag;
char *name;
```

Parámetros:

flag.- Es un entero representado por una de las siguientes macros:

```
DBOPEN
DBCLOSE
FILEOPEN
CLOSEFILE
```

Sus valores están incluidos en el archivo de cabecera **dbio.h**.

name.- Es un puntero al nombre de una base de datos o archivo.

Descripción:

La rutina **dbselect()** es usada para abrir y cerrar una base de datos o un archivo.

9.6 Rutina dbstructview()

```
dbstructview(filename, fieldlist,n)
char *filename;
struct dbview *fieldlist;
int n;
```

Parámetros:

filename.- Es un puntero al nombre de un archivo.

fieldlist.- Es un puntero a un array de estructuras del tipo dbview definida en dbio.h.

n.- Es el número de elementos del array.

Descripción:

La rutina dbstructview() permite definir una estructura de tipo dbview cuyos elementos serán idénticos a los campos del archivo filename. En otras palabras podremos crear una vista de un archivo determinado.

Esta rutina permite que un archivo definido por el programador pueda tener una correspondencia uno a uno con el array de estructura fieldlist.

Después de una llamada a dbstructview(); para cada estructura fieldlist, se alojará información acerca de cada campo como lo es la dirección de comienzo, longitud y tipo de dato.

Esta información se puede usar para utilizar datos independientes entre código en C y la base de datos informix.

9.7 Rutina dbselfield()

```
dbselfield(filename,fieldname,flag)
char *filename
char *fieldname
int flag
```

Parámetros:

filename.- Es un puntero al nombre de un archivo.

fieldname.- Es un puntero al nombre de un campo de filename que será usado como llave de lectura por la rutina dbfind().

flag.- Es un entero que puede tener el valor de las siguientes macros:

ACCKEYED para accesar a un registro, seleccionando campos indexados.

ACCESEQUENTIAL para accesar registros en órden fisico.
ACCPRIMARY para accesar registros por la llave primaria.

Descripción:

Esta rutina permite establecer el campo o la forma con la cual se realizarán búsquedas en archivos propios de Informix.

9.8 Rutina dbfind()

```
dbfind(filename,flag,value,length,recbuf)
char *filename;
int flag;
char *value;
int *length;
char *recbuf;
```

Parámetros:

filename.- Es un puntero al nombre del archivo al cual se quiere accesar.

flag.- Es un valor entero que puede ser representado por una de las siguientes macros:

COMPARISON	FIRST
EQUAL	LAST
GTEQ	CURRENT
GREATER	NEXT
PREVIOUS	

value.- Es un puntero para la búsqueda de un valor. Si el campo que está siendo comparado es de tipo entero o double, un puntero a un entero o double será pasado en el parámetro value.

length.- Es un puntero a un entero que recibirá la longitud de la vista leída por dbfind().

recbuf.- Es un puntero

Descripción:

La rutina **dbfind()** permite la búsqueda y acceso a registros de un determinado archivo y alojarlos en el buffer recbuf.

9.9 Rutina dbupdate()

```
dbupdate(filename,recbuf)
char *filename;
char *recbuf;
```

Parámetros:

filename.- Es un puntero al nombre de un archivo en donde un registro será actualizado.

recbuf.- Es un puntero a una área de memoria que contendrá la información a ser actualizada.

Descripción:

Esta rutina permite actualizar registros de un archivo específico.

9.10 Rutina dbadd()

```
dbadd(filename, recbuf)
char *filename;
char *recbuf;
```

Parámetros:

filename.- Es un puntero al nombre de un archivo en el cual un registro será agregado.

recbuf.- Es un puntero a una área de memoria que contendrá la información a ser agregada.

Descripción:

Esta rutina permite agregar o adicionar registros en un archivo.

9.11 Rutina dbdelete()

```
dbdelete(filename)
char *filename;
```

Parámetros:

filename.- Es un puntero al nombre de un archivo del cual un registro será borrado.

Descripción:

Esta rutina borra el registro más reciente de una lectura de un archivo específico por medio de la función **dbfind()**.

9.12 Rutina rtoday()

```
rtoday(pdate)
long pdate;
```

Descripción:

La rutina "**rtoday()**" convierte la fecha del sistema a un valor de tipo long, que es el número de días que han transcurrido desde 1º Enero de 1900.

9.13 Rutina rdatestr()

```
rdatestr (julian,string,type)
long julian;
char *string;
int type;
```

Parámetros:

julian.- Es un long cuyo valor es el número de días trancurridos desde el 1º de Enero de 1900.

string.- Es un puntero a una área de memoria donde se alojará el formato resultante.

type.- Es un valor entero que especifica el formato resultante. Puede ser cualquiera de las siguientes macros definidas en el archivo de cabecera dbio.h:

- DATETYPE
- EDATETYPE
- YDATETYPE

Descripción:

La rutina **rdatestr()** convierte el valor de julian de tipo long en una cadena con terminador null en cualquiera de los siguientes formatos:

Formato DATE	mm/dd/yy
Formato EDATE	dd/mm/yy
Formato YDATE	yy/mm/dd

9.14 Rutina rstoi()

```
rstoi(string,ival)
char *string;
int ival;
```

Parámetros:

string.- Es un puntero a una cadena.

ival.- Es un puntero a un entero.

Descripción:

La rutina rstoi() mira hacia string hasta encontrar un terminador null, los caracteres leídos son convertidos a enteros y depositados en ival. La función regresa un valor diferente a cero si ocurre un error y un cero si se ha tenido éxito.

9.15 Rutina rstrcpy()

```
rstrcpy(from, to)
char *from, *to;
```

Parámetros:

from.- Es un puntero a una cadena a copiar.

to.- Es un puntero a una localización en memoria donde la cadena será copiada.

Descripción:

La rutina rstrcpy copia una cadena de una localización de memoria a otra.

9.16 Rutina rbytecpy()

```
rbytecpy(from,to,length)
char *from;
char *to;
int length;
```

Parámetros:

from.- Es un puntero al comienzo del primer byte que contiene el valor a ser copiado (el número de bytes que contiene un valor depende del tipo en que haya sido declarado).

to.- Es un puntero al comienzo del primer byte donde será copiado el valor.

length.- Número de bytes a ser copiados.

Descripción:

La rutina **rbytecpy** copia el contenido de los bytes de una localización a otra por una longitud especificada por el parámetro length.

9.17 Funciones de Librería Especiales para Perform

Complementando a las funciones de librería de ALL-II C se han diseñado funciones para el control de la pantalla con PERFORM de Informix.

pf_gettype() Determina el tipo y longitud del campo desplegado sobre la pantalla.

pf_getval() Lee el valor de un campo de la pantalla.

pf_putval() Coloca un valor en un campo de la pantalla.

pf_nxfield() Mueve el cursor al campo especificado de la pantalla.

pf_msg() Escribe un mensaje en la parte inferior de la pantalla en video normal o inverso.

Los valores enteros que regresan estas funciones son cero si la función llamada tiene éxito sino se regresa un código de error.

En las siguientes secciones se describen a detalle el objetivo de éstas funciones.

9.18 Rutina pf_gettype()

```
pf_gettype(name,type,len)
char *name;
short *type, *len;
```

Parámetros:

name.- Es un puntero a una cadena que contiene la etiqueta que es usada para identificar un campo dentro de la pantalla.

type.- Es un puntero a un short que será usado por la función, ya que después de una llamada exitosa de la misma se arrojará en él el tipo de dato del campo. Los valores posibles de type pueden ser:

CHARTYPE	INTTYPE
LONGTYPE	DOUBLETYPE
FLOATTYPE	SERIALTYPE
DATETYPE	EDATETYPE
YDATETYPE	MONEYTYPE

Estas macros están definidas en el archivo de cabecera Perform.h

len.- Es un puntero a un short. Después de una llamada satisfactoria a esta función, len contendrá la longitud del campo al cual apunta name.

Descripción:

La rutina pf_gettype regresa el tipo de dato y la longitud de name que es un campo que se desplega en la pantalla.

9.19 Rutina pf_getval()

```
pf_getval(name,pretvalue,valtype,vallen)
char *name, *pretvalue;
short valtype, vallen;
```

Parámetros:

name.- Es una cadena que contiene la etiqueta usada para identificar a un campo dentro de la pantalla.

pretvalue.- Es un puntero a una cadena, un short, un long, un float o un double que es regresado por la función. El tipo de pretvalue es determinado por un valor dado por el programador en el parámetro valtype. Este tipo corresponderá al tipo de dato del campo desplegado; ambos deberán ser numéricos o caracteres.

valtype.- Es un short que indica el tipo del valor al cual pretvalue apuntará, las opciones son:

valtype	tipo
CCHARTYPE	char
CINTTYPE	int
CSHORTTYPE	short
CLONGTYPE	long, date, edate, ydate
CDOUBLETYPE	double, money
CFLOATTYPE	float

vallen.- Es un short que especifica la longitud de la cadena (siempre teniendo en cuenta uno más para el terminador null) que se colocará en pretvalue cuando valtype es CCHARTYPE, y si es diferente vallen es ignorado.

Descripción:

La rutina **pf_getval()** lee el valor de un campo específico dentro de la pantalla de captura/actualización y consulta.

9.20 Rutina pf_putval()

```
pf_putval(pvalue,valtype,name)
char *pvalue;
short valtype;
char *name;
```

Parámetros:

pvalue.- Es un puntero a una cadena, un short, un long, un float o un double que puede ser insertado mediante la llamada a la función pf_putval() en name que es un campo en la pantalla de captura/actualización o consulta.

El tipo de pvalue esta determinado por el valor contenido en el parámetro valtype.

Si se define el tipo carácter y el campo desplegado es de tipo numérico, se realiza la conversión si es posible sino, será introducido el valor de cero en el campo desplegado.

Si el tipo especificado es numérico y el campo desplegado es de tipo carácter se realiza una conversión pero si la cadena no cabe en el campo, será truncadá. Si un valor numérico no cabe en un campo, éste será llenado con asteriscos.

valtype.- Es un short que indica el tipo del valor al cual pvalue apunta. Las opciones son:

valtype	tipo
CCHARTYPE	char
CINTTYPE	int
CSHORTTYPE	short
CLONGTYPE	long, date, edate, ydate
CDOUBLETYPE	double, money
CFLOATTYPE	float

name.- Es una cadena que contiene la etiqueta con que es identificado el campo en la pantalla de captura/actualización y consulta.

Descripción:

La rutina **pf_putval()** coloca un determinado valor en un campo específico dentro de la pantalla de captura/actualización y consulta.

9.21 Rutina pf_nxfield()

```
pf_nxfield(name)
char *name;
```

Parámetros:

name.- Es una cadena que contiene la etiqueta del campo al cual el cursor será enviado.

Descripción:

La rutina **pf_nxfield()** controla la localización del cursor sobre la pantalla cuando se esta en modo de edición ya sea en la alta o actualización de registros.

9.22 Rutina pf_msg()

```
pf_msg(msgstr,reverseflag,bellflag)
char *msgstr;
short reverseflag, bellflag;
```

Parámetros:

msgstr.- Es una cadena que contiene un mensaje que será desplegado en la parte inferior de la pantalla. Se despliega en video normal y además la cadena podrá tener arriba de 80 caracteres. En video inverso, el máximo número de caracteres puede ser menor a 80 ya que el control de ellos en este modo requiere uno o dos espacios más.

reverseflag Es un short que indica si el mensaje será desplegado en video inverso ("1") o en video normal("0");

bellflag.- Es un short que indica si sonará algún timbre cuando el mensaje sea desplegado (1 = sonido , 0 = no sonido).

9.23 Archivos del Módulo C.I.F.

En las siguientes páginas se presentan los esquemas de algunos de los archivos que forman parte de la base de datos del Módulo C.I.F., y que están relacionados con las rutinas que fueron hechas en C. La figura 9.3 muestra la relación que existe entre éstos archivos y la pantalla principal de captura, actualización y consulta de CIF's.

Presupuesto Autorizado Local

```
database  cifs

file  autloc

field  autcto  type  integer              index dups { Cto.Trab. }
field  autger  type  long                 index dups { Gerencia  }
field  autcon  type  character length 3 index dups { Ren.Gasto.}
field  autyer  type  character length 2 index dups { Año }
field  autmon  type  character length 1 index dups { Moneda    }
field  autene  type  double                         { Aut.Ene. }
field  autfeb  type  double                         { Aut.Feb. }
field  autmar  type  double                         { Aut.Mar. }
field  autabr  type  double                         { Aut.Abr. }
field  autmay  type  double                         { Aut.May. }
field  autjun  type  double                         { Aut.Jun. }
field  autjul  type  double                         { Aut.Jul. }
field  autago  type  double                         { Aut.Ago. }
field  autsep  type  double                         { Aut.Sep. }
field  autoct  type  double                         { Aut.Oct. }
field  autnov  type  double                         { Aut.Nov. }
field  autdic  type  double                         { Aut.Dic. }
field  auttot  type  double                         { Aut.Tot. }
field  autkey  type  composite  autcto,autger,autcon,autyer,autmon index
end
```

MODULO C. I. F.

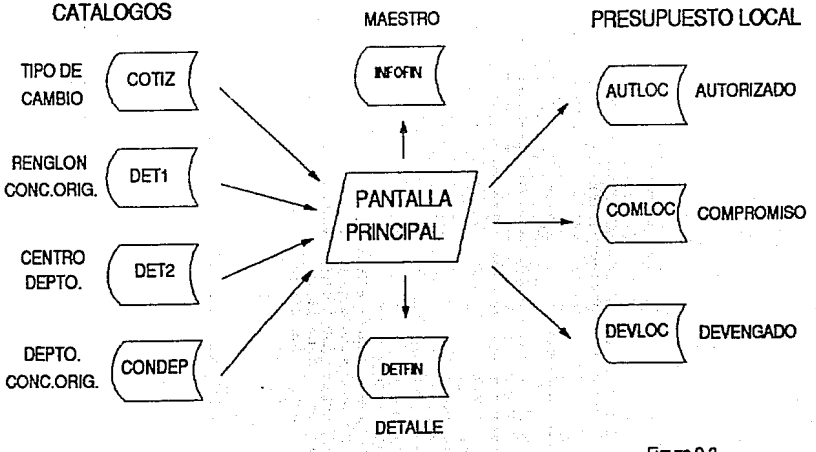


Figura 9.3

Presupuesto Comprometido Local

```
database  cifs

file  comloc

field   comcto   type   integer              index dups { Cto.Trab. }
field   comger   type   long                 index dups { Gerencia }
field   comcon   type   character length 3 index dups { Ren.Gasto }
field   comyer   type   character length 2 index dups { Año }
field   common   type   character length 1 index dups { Mom.Cont. }
field   comene   type   double                         { Comp.Ene. }
field   comfeb   type   double                         { Comp.Feb. }
field   commar   type   double                         { Comp.Mar. }
field   comabr   type   double                         { Comp.Abr. }
field   commay   type   double                         { Comp.May. }
field   comjun   type   double                         { Comp.Jun. }
field   comjul   type   double                         { Comp.Jul. }
field   comago   type   double                         { Comp.Ago. }
field   comsep   type   double                         { Comp.Sep. }
field   comoct   type   double                         { Comp.Oct. }
field   comnov   type   double                         { Comp.Nov. }
field   comdic   type   double                         { Comp.Dic. }
field   comtot   type   double                         { Comp.Tot. }
field   comkey   type   composite  comcto,comger,comcon,comyer,common inde
end
```

Presupuesto Devengado Local

```
database  cifs

file  devloc

field   devcto   type   integer              index dups { Cto.Trab. }
field   devger   type   long                 index dups { Gerencia }
field   devcon   type   character length 3 index dups { Ren.Gasto }
field   devyer   type   character length 2 index dups { Año }
field   devmon   type   character length 1 index dups { Moneda }
field   devene   type   double                         { Dev.Ene. }
field   devfeb   type   double                         { Dev.Feb. }
field   devmar   type   double                         { Dev.Mar. }
field   devabr   type   double                         { Dev.Abr. }
field   devmay   type   double                         { Dev.May. }
field   devjun   type   double                         { Dev.Jun. }
field   devjul   type   double                         { Dev.Jul. }
field   devago   type   double                         { Dev.Ago. }
field   devsep   type   double                         { Dev.Sep. }
field   devoct   type   double                         { Dev.Oct. }
field   devnov   type   double                         { Dev.Nov. }
field   devdic   type   double                         { Dev.Dic. }
field   devtot   type   double                         { Dev.Tot. }
field   devkey   type   composite  devcto,devger,devcon,devyer,devmon inde
end
```

Maestro de CIFs

```
database cifs

file  infofin

field  numero  type long                index dups  { No. Cif }
field  docufu  type character  10                   { Doc. Fuente. }
field  claaut  type character  11  index dups       { Cve. Aut. }
field  vencim  type edate                           { Fecha de Venc. }
field  momcon  type character   2  index dups       { Momento Contable }
field  impmon  type double                          { Importe Mon. Ext. }
field  fichas  type long                            { Ficha Autorizada }
field  folade  type character   8                   { Folio Adefas }
field  infokey type composite   numero, claaut, momcon    index

end
```

Detalle de CIFs

```
database cifs

file detfin

field  numcif  type long                index dups  { No. de Cif }
field  clauob  type character  11  index dups       { Cve. Aut. }
field  mocont  type character   2  index dups       { Mto. Contable }
field  fevenc  type edate                           { Fecha Venc. }
field  numpro  type character   8                   { Num. de Proy. }
field  parpre  type character   8                   { Part. Presup. }
field  moutil  type character   1                   { Moneda }
field  cetrab  type integer                         { Cto. Trabajo }
field  depto   type long                            { Gcia./Depto. }
field  conori  type character   6                   { Cpto. Origen }
field  imppar  type double                          { Importe Parcial }
field  cooiva  type character   6                   { Cpto. IVA }
field  impiva  type double                          { Importe IVA }
field  feelci  type edate                           { Fecha Elaboración }
field  suitch  type character   1                   { Suitch de Estado }
field  defiky  type composite   numcif, clauob, mocont    index dups
field  defikyb type composite   numcif, clauob, mocont, cetrab, dept
                                          conori, imppar     index dup
end
```

Catálogo de Cotizaciones de Moneda Extranjera

```
database  cifs

file cotiz

field  moneda    type char length  1  index dups  { Clave de Moneda }
field  mescot    type char length  2  index dups  { Mes de Cotización }
field  ticambio  type double                     { Tipo de Cambio }
field  monecom   type composite moneda, mescot index

end
```

Catálogo de Renglón de Gasto/Concepto de Origen

```
database cifs

file det1

field  renglas   type char 3  index dups  { Renglón de Gasto   }
field  concepto  type char 6  index       { Concepto de Origen }
field  tipmon    type char 1              { Tipo de Moneda     }
field  gaskey    type composite concepto,tipmon  index

end
```

Catálogo de Centros de Trabajo/Departamentos

```
database cifs

file  det2

field subdira type char  1   index dups      { Clave Subdirección }
field cntrab  type int       index dups      { Clave Cent.Trab. }
field destra  type long      index dups      { Clave del Depto. }
field desdpt  type char 40                   { Descrip. Depto.  }
field unicon  type integer   index dups      { Unidad Control   }
field cuemay  type integer                   { Cuenta mayor     }
field cueope  type char  6   index dups      { Cuenta Operativa }
field keydep  type composite subdira, cntrab index dups
field cenkey  type composite cntrab, destra index
field cenkey1 type composite subdira, cntrab, destra,cueope index

end
```

Catálogo de Departamento/Concepto de Origen

```
database  cifs

file  condep

field  depart  type  long   index dups  { Departamento          }
field  concep  type  char 6  index dups  { Concepto de origen }
field  depcon  type  composite  depart,concep index

end
```

9.24 Pantalla Principal del Módulo C.I.F. (PANCIF.FRM)

La pantalla principal del módulo cif está compuesta por la unión de dos archivos que actuán uno, como archivo maestro (infofin) y otro como archivo de detalle (detfin). En la figura 9.4 se muestran los campos de cada uno de los archivos así como los campos llave que sirven para unir un maestro con varios detalles.

El archivo maestro contendrá información única, que servirá de referencia para capturar registros de detalle.

En la pantalla se podrá agregar, consultar, actualizar y borrar registros tanto del archivo maestro como del archivo de detalle de la base de datos. La figura 9.5 presenta la pantalla tal y como se muestran en el monitor de la PC.

Al iniciar siempre una sesión en la pantalla principal, se llama a la rutina INICIO() la cual abre la base de datos denominada cifs y avisa si está próximo el cierre de mes para llevar acabo el proceso de históricos.

En el archivo maestro (infofin) se capturan los siguientes campos:

a) **Número de Cif.**- Es un número consecutivo que lleva cada unidad de control para identificar en un momento determinado el Cif.

b) **Documento Fuente.**- Puede ser el número de una factura o cualquier cadena de caracteres que sirva de identificación del documento a registrar.

c) **Clave de Autorización.**- Esta clave es una cadena de 11 caracteres que asigna cada unidad de control para la expedición del Cif. El primer caracter identifica, mediante una letra el mes en el cual se elabora el cif, por ejemplo:

A = Enero
B = Febrero
C = Marzo

etc.

El segundo y tercer caracter indican el número de día de elaboración del cif y el cuarto caracter el último dígito del año.

El quinto, sexto y séptimo caracter corresponden a la clave que tiene la unidad de control dentro de Pemex.

Y los restante caracteres se dejan para introducir el número de consecutivo del Cif.

Una vez que es introducida por completo la clave de autorización del Cif, se llama a la rutina UNICON(), cuyo

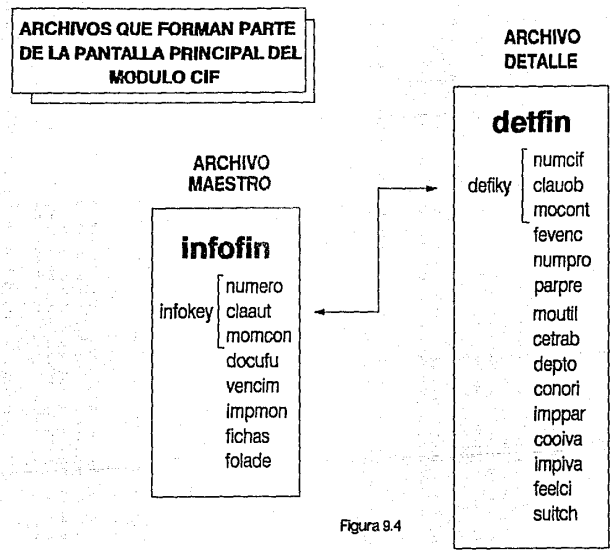

ARCHIVOS QUE FORMAN PARTE DE LA PANTALLA PRINCIPAL DEL MODULO CIF
ARCHIVO MAESTRO
infofin
infokey
numero
claaut
momcon
docufu
vencim
impmon
fichas
folade
ARCHIVO DETALLE
detfin
defiky
numcif
clauob
mocont
fevenc
numpro
parpre
moutil
cetrab
depto
conori
imppar
cooiva
impiva
feelci
suitch

Figura 9.4

PETROLEOS MEXICANOS

MODULO C.I.F.

PANTALLA PRINCIPAL

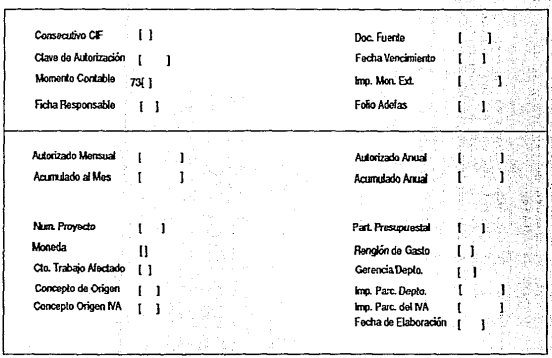


Figura 9.5

propósito es verificar la existencia de la unidad de control en el archivo de catálogo correspondiente (det2). En caso de que exista la clave, el cursor en la pantalla pasa al siguiente campo a capturar, si no existiese la clave se mandará un mensaje de error y el cursor permanecerá en el campo hasta que no se capture la clave correcta o hasta que se aborte el proceso de captura y se dé de alta la clave en el archivo de catálogo

d) **Fecha de Vencimiento.-** Esta es la fecha a la cual el cif tiene que ser registrado en las contadurías de Pemex, y no debe ser menor que la fecha de elaboración ya que si lo es, la pantalla manda inmediatamente un mensaje de error.

e) **Momento Contable.-** En el módulo cif por lo general se manejan dos momentos contables que son:

7313 - Compromiso
7314 - Devengado

Existen otros, pero éstos están muy relacionado con las rutinas en C.

f) **Importe en Moneda Extranjera.-** Si el cif que se está capturando es por alguna moneda extranjera (como dólares, libras, marcos, yenes, francos, etc.) el campo es llenado, en caso de que el importe sea en moneda nacional simplemente se dejan en blanco y se pasa al siguiente campo de captura.

g) **Ficha del Responsable.-** Este campo corresponde a la clave que tiene el jefe de la unidad de control, quién es responsable de la generación del cif.

h) **Folio de Adefas.-** Esta clave corresponde al último campo a capturar en el archivo maestro y se refiere al número de folio de los Adeudos de Años de Anteriores.

Todos estos datos se consideran como información única dentro de un cif, por lo cual, son grabados en el archivo maestro. Pero un cif puede contener varios detalles.

En el archivo de detalle (detfin) se capturan los siguientes campos:

a) **Número de Proyecto.-** En PEMEX cada obra o trabajo de matenimiento se le asigna un número de proyecto el cual determina en un momento dado en donde es realizado (en este caso en cual refinería) , por cual subdirección, y demás características.

b) **Partida Presupuestal.-** Con el número de partida se puede identificar fácilmente la forma en la cual el proyecto de obra afecta al presupuesto.

c) **Moneda.-** Este es un campo de un caracter en donde se identifica el tipo de moneda en el que se hace el cif:

N - Moneda Nacional.
D - Dólares.
Y - Yenes

d) **Centro de Trabajo Afectado.-** Este campo corresponde a la clave que tiene el centro de trabajo que será afectado presupuestalmente con este cif.

e) **Gerencia/Departamento.-** Este campo corresponde a la clave del departamento que también será afectado presupuestalmente.

Es importante remarcar que todos los centro de trabajo de PEMEX (como son las Refinerías y Petroquímicas), incluso las oficinas centrales (Subdirecciones, Coordinaciones, Gerencias, etc) tienen una clave del centro de trabajo y departamento determinadas y pueden ser consultadas por el catalogo institucional de claves de centros de trabajo/departamentos.

Después de capturar la clave del departamento se llama a la rutina CENDEP(), ésta se encarga de verificar la existencia de la relación centro/departamento del archivo de catálogo det2 (catálogo de departamentos).

f) **Concepto de Origen.-** Este campo esta compuesto por seis caracteres cuya combinación de números determina el concepto de origen del cif. Para identificar fácilmente un Concepto de Origen existe el cátalogo Institucional de Conceptos de Origen/Renglón de gasto.

Al capturar el Concepto de Origen, la pantalla verifica si se dió de alta algún proyecto de obra, si existe, determina si el concepto de origen se encuentra dentro del rango de >= "707001" y <= "707010"; ésto se realiza porque los conceptos de origen que se encuentran dentro de este rango deben tener por convención un número de proyecto. Si el concepto de origen se encuentra dentro del rango se corre la rutina GASTOS() la cual verifica en el archivo de catálogo det1 (Catálogo de Conceptos de Origen) si existe el concepto de origen en mancuerda con el tipo de moneda, en caso de encontrar la correspondencia de estos datos, se coloca en pantalla el renglón de gasto al que pertenece el concepto de origen, si no lo encuentra manda un mensaje de error. Ahora en el caso de que el concepto de origen no se encuentre dentro del rango se correrá la rutina GASTOS1() la cual verifica que de acuerdo con el número de proyecto se sepa si se trata de un proyecto de obra de inversion o de un trabajo de mantenimiento y de acuerdo a ello se coloca la clave correspondiente al renglón de gasto.

Enseguida se corre la rutina CONFIR() la cual verifica la relación del concepto de origen/departamento en el archivo **condep**. El objetivo de esta rutina es el de checar que el concepto de origen esté autorizado para afectar presupuestalmente a un departamento determinado.

g) **Importe Parc. Depto.** - Este es el campo en donde se captura el importe del cif el cual afectará presupuestalmente al centro de trabajo y departamento ya capturados.

Antes de que se agrege o actualice el importe se corre la rutina DISCALL(0,r2), la cual muestra en pantalla, tanto en forma mensual como anual, el presupuesto autorizado que se tienen antes de hacer alguna afectación, esto con el fin de que el usuario este conciente del presupuesto con que se cuenta.

Es importante remarcar que el presupuesto se presenta a un nivel de centro de trabajo, departamento, renglon de gasto y moneda.

Después, si el momento contable es 13 (compromiso) se corre la rutina DISCALL(1,r2),y si es 14 DISCALL(2,r2), el primero de los argumentos identifica el archivo (0 - Autorizado, 1 - Compromiso, 2 - Devengado) y el segundo es el renglón de gasto. Para estas dos rutinas se presentan en pantalla las afectaciones presupuestales que se tienen acumuladas tanto al mes como al año y deacuerdo al nivel mencionado con anterioridad.

Una vez que se digita el importe y si éste es diferente a la moneda nacional se corre la rutina CAMBIO() la cual se encarga de multiplicar el importe por el correspondiente tipo de cambio de la moneda y presentar así el importe ya convertido a moneda nacional.

h) **Concepto Origen IVA.**- En pemex se tienen diferentes conceptos de origen para el IVA; éstos van de acuerdo al porcentaje que se tome.

i) **Importe Parc. del IVA.**- Este es el importe resultante del importe parcial por departamento multiplicado por el porcentaje que se haya tomado para el IVA.

j) **Fecha de Elaboración.**- Esta es la fecha en la cual se realiza el cif y por default es la fecha del sistema.

Todos estos campos comforman el detalle de un cif, y como lo mencionamos antes un cif puede tener varios detalles.

Cuando se dá de alta un detalle que tenga momento contable 13 (compromiso) se corre la rutina ACTCALL(1,r2) la cual mediante la llave centro, departamento, renglón y moneda busca su correspondiente en el archivo comloc (Presupuesto Comprometido),

si lo encuentra agrega el importe del cif en el mes de proceso actualizando con ello el archivo, en caso de no encontrarlo simplemente lo dá alta con el importe que tiene el cif. Posteriormente se corre la rutina DISCALL(1,r2) la cual como ya lo mencionamos visualiza en pantalla las afectaciones mensual y anual del presupuesto, en este caso en su fase de compromiso. Ahora si el momento contable es 14 (devengado) se corren estas mismas rutina pero el primer argumento cambia de 1 a 2 y se agrega o actualiza en el archivo devloc (Presupuesto Devengado).

En la página siguiente se muestra el programa fuente de la pantalla principal del módulo cif en donde se puede observar la forma en la que son llamadas las rutina en C.

```
{ pancif - Carga, Actualización y Consulta de CIFS }

database  cifs

screen
{
                    P E T R O L E O S    M E X I C A N O S
                          *** M O D U L O   C I F ***
                          ----- CAPTURA DE CIFS -----

|Consecutivo CIF        [cif ]          Doc. Fuente           [a          ]
|Clave de Autorizacion  [d          ]   Fecha Vencimiento     [e          ]
|Momento Contable    73 [f ]            Importe Mon. Ext.     [g          ]
|Ficha del Responsable  [h      ]       Folio de Adefas       [i          ]
|
|Autorizado Mensual     [r          ]   Autorizado Anual      [r0         ]
|Acumulado al Mes       [r1         ]   Acumulado  Anual      [r3         ]
|
|Num. de Proyecto       [b          ]   Part. Presupuestal    [c          ]
|Moneda                 [j]             Renglon Afectado      [r2 ]
|Cto. Trabajo Afectado  [k    ]         Gerencia/Depto.       [l      ]
|Concepto de Origen     [m    ]         Imp. Parc. Depto.     [n          ]
|Concepto Origen IVA    [o    ]         Imp. Parc. del IVA    [p          ]
|                                       Fecha Elaboracion     [s        ]
|
}
end

attributes
cif = numero  = numcif, reverse, required, noupdate;
a   =  docufu,upshift,queryclear,autonext,required, noupdate;
d   =  claaut  = clauob, upshift, required, picture = "&&&&&&&&&&&", autonext,
       noupdate;
e   =  vencim  = fevenc, required, comments = "Fecha en formato: DDMMAA",
       noupdate;
f   =  momcon  = mocont, required, include = ( 13,14 ),
       autonext, noupdate;
g   =  impmon, format = "&&&&&&&&.&&", noupdate;
h   =  fichas, required, default = 120922, noupdate;
i   =  folade,upshift,queryclear, noupdate;
b   =  numpro,upshift,autonext,queryclear,lookup joining *proye, noupdate;
c   =  parpre,upshift,autonext,queryclear,lookup joining *parti, noupdate;
j   =  moutil, required, include = ("A" to "Z"), default = "N", upshift,
       noupdate;
k   =  cetrab, autonext, noupdate;
l   =  depto,  autonext, noupdate;
m   =  conori, autonext, noupdate,
n   =  imppar, reverse, format = "&&&&&&&&&&&", noupdate;
o   =  cooiva, autonext,include=("929100","929300","929400","       "),
       queryclear, noupdate;
p   =  impiva, format = "&&&&&&&&&&&", noupdate;
s   =  feelci,default = today, noupdate;
r   =  displayonly type double,reverse, format = "&&&&&&&&&&&";
r0  =  displayonly type double, reverse, format = "&&&&&&&&&&&";
r1  =  displayonly type double, reverse, format = "&&&&&&&&&&&";
```

```
r2 =  displayonly type character, reverse;
r3 =  displayonly type double, reverse, format = "##########";

instructions

composite join  infokey = defiky;

infofin master of detfin

on beginning
    call inicio()

after  editadd editupdate of claaut
    call unicon()

after  editadd editupdate of numpro
    if( b = '      ') then
      nextfield = j

before editadd editupdate of moutil
    if ( e < s ) then
      begin
        comments bell reverse "Fecha de Vencimiento menor
                               a la Fecha de Elaboracion!!"
        nextfield = e
      end

after editadd editupdate of depto
    call cendep()

after editadd of conori
    if ( b <> "          ") then
      begin
        if( m >= "707001" and m <= "707010") then
            let r2 = gastos()
        else
            let r2 = gastos1()
      end
    else
        let r2 = gastos()
    call confir()
    if ( m = "300100" or m = "301200" or m = "301500" or m = "301600") then
      begin
        if ( b = "          ") then
          begin
            comments bell reverse "Este concepto requiere
                                   Num. de Obra!!"
            nextfield = m
          end
      end
```

```
before editadd editupdate of imppar
    let r = discall(0,r2)
    if ( f = 13 ) then let r1 = discall(1,r2)
    else if ( f = 14 ) then let r1 = discall(2,r2)


after editadd editupdate of imppar
    if ( j <> "N" ) then call cambio()

before editadd editupdate of cooiva
    if ( r1 > r ) then
       comments bell reverse " !! REBASA AUTORIZADO MENSUAL !! "

after editadd editupdate of cooiva
    if(o="929100") then
       let p = n * 0.06
    else if(o="929300") then
       let p = n * 0.15
    else if(o="929400") then
       let p = n * 0.20
    else
       nextfield = j

after editadd editupdate of impiva
    nextfield = j

after editadd editupdate of detfin
    if ( f = 13 ) then
       begin
         call actcall(1,r2)
         let r1 = discall(1,r2)
       end
    else
       if ( f = 14 ) then
          begin
            call actcall(2,r2)
            let r1 = discall(2,r2)
          end

on ending
    call final()

end
```

9.25 Rutinas en C

La pantalla principal del Módulo CIF llama a las siguientes rutinas en C:

a) inicio()
b) unicon()
c) cendep()
d) confir()
e) verifi()
f) gastos()
g) gastos1()
h) cambio()
i) final()
j) abrebase()
k) cierra()
l) actcall()
m) discall()
n) updcall()

Todas éstas rutinas se encuentran en un archivo denominado Pantalla.c, en él, se incluye el archivo de cabecera pantalla.h que a su vez incluye a los archivos perform.h, stdio.h y dbio.h., después de linkear estos archivos con las librerías del sistema se compila con Laticce obteniendo el archivo ejecutable Pantalla.exe, el cual, al correrlo junto con Pancif.frm:

Pantalla Pancif

Permite que sea llamada cualquier rutina desde la pantalla principal del módulo CIF.

9.26 Rutina inicio()

En esta rutina se abre la base de datos y se checa, deacuerdo a la fecha del sistema, si se debe realizar el cierre de mes.

```
perfvalue inicio()
{
 int Dia;
 dbselect(DBOPEN,"cifs");
 rtoday(&Fechlong);
 rdatestr(Fechlong,Fechstr,DATETYPE);
 Fechstr[2]='\0';
 rstoi(Fechstr+0,&Mes); Mes--;
 Fechstr[5]='\0';
 rstoi(Fechstr+3,&Dia);
 if(Dia > 8 && Dia < 12) {
   printf("\n Segun la computadora, estamos en el mes de
         %s",mesjul[Mes]);
   printf("\n\n  Por lo tanto, se deberan generar HISTORICOS de
         CIF's . . .");
   printf("\n  ( Si ya fue hecho, ignore este mensaje por favor
         )");
   if(!ask("\n\n Desea continuar? (s/n) ", "s")) exit(0);
   }
}
```

```
ask(s,def)
char *s, *def;
{
 char ans[10];
 write(1,s,strlen(s));
 write(1,def,1);
 write(1,"\b",1);
 if(read(0,ans,10) <=1 || ans[0] == def[0]) return(1);
 return(0);
}
```

9.27 Rutina unicon()

Verifica que sea válida la clave de la unidad de control presupuestal que está implícita dentro del campo correspondiente a la clave de autorización del CIF.

```
perfvalue unicon()
{
 struct {
        int uncon;
 } control;

 char temporal[12];
 abrebase(CTLFILE);
 pf_getval("d,temporal,CCHARTYPE,12);
 temporal[7]='\0';
 rstoi(temporal+4,&control.uncon);
 if(dbfind("det2,COMPARISON,0,&Length,&control) != 0) {
   cierra(CTLFILE);
   pf_msg("Unidad de Control Inexistente",1,1);
   pf_nxfield("d");
 }
 else cierra(CTLFILE);
}
```

9.28 Rutina cendep()

Verifica que exista la correspondencia entre centro de trabajo y departamento.

```
perfvalue cendep()
{
 struct {
        int cntrab;
        long destra;
 } cen;
```

```
 abrebase(CENFILE);
 pf_getval("k",&cen.cntrab,CINTTYPE,0);
 pf_putval("l",&cen,destra,CLONGTYPE,0);
 if(dbfind("det2",COMPARISON,0,Length,&cen)!=0) {
   cierra(CENFILE);
   pf_msg("Depto. no adscrito a este centro, DAR DE ALTA!!",1,1);
   pf_nxfield("k");
 }
 cierra(CENFILE);
}
```

9.29 Rutina confir()

Verifica que el concepto de origen capturado exista dentro del Catálogo Institucional de Conceptos de Origen de PEMEX. Si existe, se checa que el departamento que está afectando con ese concepto, esté autorizado para ello.

```
perfvalue confir()
{
 struct{
        char conori[7];
 } cen;

 abrebase(DECFILE);
 pf_getval("m",cen.conori,CCHARTYPE,7);
 if(dbfind("condep",COMPARISON,cen.conori,&Length,&cen)==0) {
   cierra(DECFILE);
   verifi();
 }
 cierra(DECFILE);
}

verifi()
{
 struct {
        long destra;
        char conori[7];
 } cen1;

 abrebase(DECFILE1);
 pf_getval("l",&cen1.destra,CLONGTYPE,0);
 pf_getval("m",&cen1.conori1,CCHARTYPE,7);
 if(dbfind("condep",COMPARISON,0,&Length,&cen1) != 0) {
   cierra(DECFILE1);
   pf_msg("CONCEPTO DE ORIGEN NO AUTORIZADO A ESTE DEPTO.
          !!",1,1);
   pf_nxfield("m");
 }
 cierra(DECFILE1);
}
```

9.30 Rutina gastos()

Esta rutina checa que exista el Concepto de Origen en mancuerna con el tipo de moneda en el catálogo de Renglón/Concepto. Si se encuentra la correspondencia se extrae del catálogo, el renglón de gasto y se visualiza en la pantalla.

```
perfvalue gastos()
{
 struct {
        char renglas[4], concepto[7], tipmon[2];
 } gas;

 abrebase(GASFILE);
 pf_getval("j",gas.tipmon,CCHARTYPE,2);
 pf_getval("m",gas.concepto,CCHARTYPE,7);
 if(dbfind("det1",COMPARISON,0,&Length,&gas) != 0) {
   cierra(GASFILE);
   pf_msg("Concepto de Origen Inexistente, DAR DE ALTA !!",1,1);
   pf_nxfield("m");
 }
 rstrcpy(gas.renglas,Ren);
 strreturn(Ren,3);
 cierra(GASFILE);
}
```

9.31 Rutina gastos1()

Cuando se captura el número de Proyecto, la rutina verifica si se trata de alguna obra de inversión (si la clave comienza con una R) o algún trabajo de mantenimiento (si comienza con RR) y direcciona el registro al renglón de gasto correspondiente.

```
perfvalue gastos1()
{
 pf_getval("b",proye,CCHARTYPE,9);
 proye[2]='\0';
 if(strcmp(proye,"RR") == 0) {
   strreturn("310",3);
 }
 else
   strreturn("317",3);
}
```

9.32 Rutina cambio()

En esta rutina se realizan las conversiones de los importes de monedas extranjeras a moneda nacional, basándose en el catálogo de tipos de cambio.

```
perfvalue cambio()
{
 double tercer;
 struct {
        char moneda[2], mescot[3];
        double ticambio;
 } chan;

 abrebase(COTFILE);
 pf_getval("j",chan.moneda,CCHARTYPE,2);
 pf_getval("s",Fechlong,CLONGTYPE,0);
 rdatestr(Fechlong,Fechstr,EDATETYPE);
 rstrcpy(Fechstr+3,chan.mescot);
 if(dbfind("cotiz",COMPARISON,0,&Length,&chan) == 0) {
   pf_getval("n",&Importe,CDOUBLETYPE,0);
   tercer = Importe * chan.ticambio;
   pf_putval(&tercer,CDOUBLETYPE,"n");
 }
 else
 pf_msg("Cotización NO establecida !!",1,1);
 cierra(COTFILE);
}
```

9.33 Rutina final()

Esta rutina simplemente cierra la base de datos.

```
perfvalue final()
{
 dbselect(DBCLOSE,"cifs");
}
```

9.34 Rutina abrebase()

Esta rutina está formada básicamente por un case y tiene el propósito de abrir un archivo, de crear una vista con los campos que conforman el archivo, y declarar el campo que servirá de llave para accesar al archivo.

```
abrebase(n)
int n;
{
switch (n) {
        case AUTFILE:   dbselect(FILEOPEN,"autloc");
                        dbstructview("autloc",autflds,18);
                        dbselfield("autloc","autkey",ACCKEYED);
                        break;
```

```
        case COMFILE:   dbselect(FILEOPEN,"comloc");
                        dbstructview("comloc",comflds,18);
                        dbselfield("comloc","comkey",ACCKEYED);
                        break;

        case DEVFILE:   dbselect(FILEOPEN,"devloc");
                        dbstructview("devloc",devflds,18);
                        dbselfield("devloc","devkey",ACCKEYED);
                        break;

        case GASFILE:   dbselect(FILEOPEN,"det1");
                        dbstructview("det1",gasflds,3);
                        dbselfield("det1","gaskey",ACCKEYED);
                        break;

        case CENFILE:   dbselect(FILEOPEN,"det2");
                        dbstructview("det2",cenflds,2);
                        dbselfield("det2","cenkey",ACCKEYED);
                        break;

        case COTFILE:   dbselect(FILEOPEN,"cotiz");
                        dbstructview("cotiz",camflds,3);
                        dbselfield("cotiz","monecom",ACCKEYED);
                        break;

        case CTLFILE:   dbselect(FILEOPEN,"det2");
                        dbstructview("det2",conflds,1);
                        dbselfield("det2","unicon",ACCKEYED);
                        break;

        case DECFILE:   dbselect(FILEOPEN,"condep");
                        dbstructview("condep",dcoflds,1);
                        dbselfield("condep","concep",ACCKEYED);
                        break;

        case DECFILE1:  dbselect(FILEOPEN,"condep");
                        dbstructview("condep",dcoflds1,2);
                        dbselfield("condep","depcon",ACCKEYED);
                        break;
}
}
```

9.35 Rutina cierra()

En esta rutina se presenta otro case, en el que cada opción cierra un archivo determinado.

```
cierra(p)
int p;
{
switch(p) {
        case AUTFILE:   dbselect(FILECLOSE,"autloc");
                        break;
```

```
        case COMFILE:   dbselect(FILECLOSE,"comloc");
                        break;

        case DEVFILE:   dbselect(FILECLOSE,"devloc");
                        break;

        case GASFILE:   dbselect(FILECLOSE,"det1");
                        break;

        case CENFILE:   dbselect(FILECLOSE,"det2");
                        break;

        case COTFILE:   dbselect(FILECLOSE,"cotiz");
                        break;

        case CTLFILE:   dbselect(FILECLOSE,"det2");
                        break;

        case DECFILE:   dbselect(FILECLOSE,"condep");
                        break;

        case DECFILE1:  dbselect(FILECLOSE,"condep");
                        break;

}
}
```

9.36 Rutina actcall()

Esta rutina dá de alta un registro deacuerdo al centro de trabajo, departamento, moneda, año de afectación y renglón de gasto ya sea en el archivo del Presupuesto Autorizado, Compromiso, Devengado o Pagado.

```
perfvalue actcall(archivo, renglon)
perfvalue archivo, renglon;
{
 int i;

 abrebase(toint(archivo));
 for(i=0; i<6; i++) comreg.con[i] = ' ';
 rbytecpy(renglon->v_charp,comreg.con,renglon->v_len);
 comreg.con[4]='\0';
 pf_getval("k",&comreg.cto,CINTTYPE,0);
 pf_getval("l",&comreg.ger,CLONGTYPE,0);
 pf_getval("j",&comreg.mon,CCHARTYPE,2);
 pf_getval("s",&Fechlong,CLONGTYPE,0);
 rdatestr(Fechlong,Fechstr,EDATETYPE);
 rstrcpy(Fechstr+6,comreg.yer);
 rstoi(Fechstr+6,&Ae);
 Fechstr[5]='\0';
 rstoi(Fechstr+3,&Mes); Mes--;
```

```
if(dbfind(Nombre[toint(archivo),COMPARISON,0,&Length,&comreg)!=0)
  {
   for(i=0; i<12; i++)
       comreg.mes[i]=0.0;
   pf_getval("n",&Importe,CDOUBLETYPE,0);
   comreg.mes[Mes] = comreg.tot = Importe/1000.0;
   pf_msg("Añadido a Renglón de Gasto . . . ");
   dbadd(Nombre[toint(archivo)],comreg);
  }
  else {
   comreg.tot = 0.0;
   pf_getval("n",&Importe,CDOUBLETYPE,0);
   comreg.mes[Mes] += Importe/1000.0;
   for(i=0; i<12; i++)
      comreg.tot += comreg.mes[i];
   pf_msg("Agregado a Renglón de Gasto . . . ");
   dbupdate(Nombre[toint(archivo),comreg);
  }
  cierra(toint(archivo));
}
```

9.37 Rutina discall()

Esta rutina visualiza el presupuesto mensual y anual (agregado a renglon de gasto) que tiene determinado centro de trabajo al estarle realizando alguna afectación, a la vez que anuncia el posible rebazamiento del presupuesto.

```
perfvalue discall(archivo,renglon)
perfvalue archivo, renglon;
{
 int i;
 Importe = 0.0;
 Impotot = 0.0;
 Impoanu = 0.0;

 abrebase(toint(archivo));
 for(i=0; i<6; i++) comreg.con[i]=' ';
 rbytecpy(renglon->v_charp, comreg.con, renglon->v_len);
 comreg.con[4]='\0';
 pf_getval("k",&comreg.cto,CINTTYPE,0);
 pf_getval("l",&comreg.ger,CLONGTYPE,0);
 pf_getval("j",&comreg.mon,CCHARTYPE,2);
 pf_getval("s",&Fechlong,CLONGTYPE,0);
 rdatestr(Fechlong,Fechstr,EDATETYPE);
 rstrcpy(Fechstr+6,comreg.yer);
 Fechstr[5]='\0';
 rstoi(Fechstr+3,&Mes); Mes--;
 if(dbfind(Nombre[toint(archivo)],COMPARISON,0,Length,&comreg)==0){
   if(toint(archivo) != AUTFILE) {
      pf_getval("n",&Importe,CDOUBLETYPE,0);
      Impotot=(comreg.tot * 1000.0) + Importe;
```

```
      pf_putval("&Impotot,CDOUBLETYPE,"r1");
      Impoanu=(comreg.tot * 1000.0);
      pf_putval("&Impoanu,CDOUBLETYPE,"r3");
   }
   else {
      Impoanu=(comreg.tot * 1000.0);
      pf_putval(&Impoanu,CDOUBLETYPE,"r0");
   }
 Importe=(comreg.mes[Mes] * 1000.0) + Importe;
 }
 cierra(toint(archivo));
 dubreturn(Importe);
}
```

9.38 Rutina updcall()

Esta rutina se encarga se afectar los archivos de presupuestos (Autorizado, Compromiso, Devengado y Pagado) deacuerdo con los campos claves que tiene el CIF.

```
perfvalue updcall(archivo,renglon)
perfvalue archivo, renglon;
{
 int i;
 abrebase(toint(archivo));
 for(i=0; i<6; i++) comreg.con[i]=' ';
    rbytecpy(renglon->v_charp,comreg.con,renglon->v_len);
 comreg.con[4]='\0';
 pf_getval("k",&comreg.cto,CINTTYPE,0);
 pf_getval("l",&comreg.ger,CLONGTYPE,0);
 pf_getval("j",&comreg.mon,CCHARTYPE,2);
 pf_getval("s",&Fechlong,CLONGTYPE,0);
 rdatestr(Fechlong,Fechstr,EDATETYPE);
 rstrcpy(Fechstr+6,comreg.yer);
 rstoi(Fechstr+6,&Ae);
 Fechstr[5]='\0';
 rstoi(Fechstr+3,&Mes); Mes--;
 if(dbfind(Nombre[toint(archivo)],COMPARISON,0,&Length,&comreg)==0){
   comreg.tot = 0.0;
   pf_getval("n",&Importe,CDOUBLETYPE,0);
   comreg.mes[Mes]-=Importe/1000.0;
   for(i=0; i<12; i++)
     comreg.tot += comreg.mes[i];
   dbupdate(Nombre[toint(archivo)],comreg);
 }
 cierra(toint(archivo));
}
```

9.39 Archivo de Cabecera: pantalla.h

Este archivo sirven de alguna manera para definir variables, constantes, funciones y estructuras que se utilizarán en las rutinas contenidas en el archivo pantalla.c, además de incluir otros archivos de cabecera como son, stdio.h, perform.h y dbio.h los cuales ayudan a establecer el ambiente en el cual se está trabajando.

Pantalla.h

```
/* Definiciones para el programa pantalla.c del Sistema CIF */

#include <stdio.h>
#include <perform.h>
#include <dbio.h>

#define AUTFILE 0
#define COMFILE 1
#define DEVFILE 3
#define GASFILE 4
#define DECFILE 5
#define CENFILE 6
#define COTFILE 7
#define CTLFILE 8
#define DECFILE1 9

/* Definiciones de las funciones que ejecutará PERFORM */

extern perfvalue inicio();
extern perfvalue unicon();
extern perfvalue cendep();
extern perfvalue confir();
extern perfvalue actcall();
extern perfvalue discall();
extern perfvalue updcal();
extern perfvalue gastos();
extern perfvalue gatos1();
extern perfvalue cambio();
extern perfvalue final();

struct ufunc userfuncs[] =
{
 "inicio",inicio,"unicon",unicon,"cendep",cendep,"confir",confir,
 "actcall",actcall,"discall",discall,"updcall",updcall,
 "gastos",gastos,"gastos1",gastos1,"cambio",cambio,"final",final,
 0,0
};
```

```
/* Estructuras a utilizarse en el programa */

struct dbview autflds[]=
{
 {"autcto"}, {"autger"}, {"autcon"}, {"autyer"}, {"autmon"},
 {"autene"}, {"autfeb"}, {"autmar"}, {"autabr"}, {"autmay"},
 {"autjun"}, {"autjul"}, {"autago"}, {"autsep"}, {"autoct"},
 {"autnov"}, {"autdic"}, {"auttot"}
};

struct dbview comflds[]=
{
 {"comcto"}, {"comger"}, {"comcon"}, {"comyer"}, {"common"},
 {"comene"}, {"comfeb"}, {"commar"}, {"comabr"}, {"commay"},
 {"comjun"}, {"comjul"}, {"comago"}, {"comsep"}, {"comoct"},
 {"comnov"}, {"comdic"}, {"comtot"}
};

struct dbview devflds[]=
{
 {"devcto"}, {"devger"}, {"devcon"}, {"devyer"}, {"devmon"},
 {"devene"}, {"devfeb"}, {"devmar"}, {"devabr"}, {"devmay"},
 {"devjun"}, {"devjul"}, {"devago"}, {"devsep"}, {"devoct"},
 {"devnov"}, {"devdic"}, {"devtot"}
};

struct dbview gasflds[]=
{
 {"renglas"}, {"concepto"}, {"tipmon"}
};

struct dbview cenflds[]=
{
 {"cntrab"}, {"destra"}
};

struct dbview camflds[]=
{
 {"moneda"}, {"mescot"}, {"licambio"}
};

struct dbview conflds[]=
{
 {"unicon"}
};

struct dbview dcoflds[]=
{
 {"concep"}
};

struct dbview dcoflds1[]=
{
 {"depart"}, {"concep"}
};
```

```
/* Variable Estáticas */

static struct reng {  /* Para Autorizado, Comprometido, Devengado
                         y Pagado */
int    cto;
long   ger;
char   con[4], yer[3], mon[2];
double mes[12];
double tot;
} comreg;

static char *Nombre[]={
"autloc", "comloc", "devloc"
};

static char *mesjul[]={
"ENERO", "FEBRERO", "MARZO", "ABRIL", "MAYO", "JUNIO",
"JULIO", "AGOSTO", "SEPTIEMBRE", "OCTUBRE", "NOVIEMBRE",
"DICIEMBRE"
};

static int Length, Ae, Mes;
static char Ren[4], Fechstr[9], proye[9];
static long Fechlong;
static double Importe;
static double Impotot;
static double Impoanu;
```

9.40 Archivo de Cabecera: Perform.h

Se pueden controlar las sesiones con Perform llamando a algunas rutinas en C, ya sea desde el inicio o final de la sesión, para lograr esto es necesario incluir el archivo de cabecera Perform.h, el cual permite definir ciertos parámetros para lograr una perfecta comunicación o interface entre lo que se captura y visualiza en la pantalla y los archivos que están involucrados.

Para incluir este archivo en un programa de aplicacion sólo basta con escribir la siguiente línea:

```
#include<perform.h>
```

Acontinuación se presenta el contenido de éste archivo de cabecera:

```
/*****************************************************************
 *
 *              Relational Database Systems, Inc.
 *
 *
 *  Title:      perform.h
 *  Sccsid:     @(#)perform.h   2.2     10/31/84  16:23:25
 *  Description:
 *              Header file for C hooks into perform.
 *
 *****************************************************************
 */

/*
 * This is the file which must be included in any C subroutine
 * source file which is to be linked to "libperf.a".
 */


#define CHARTYPE        0
#define INTTYPE         1
#define LONGTYPE        2
#define DOUBLETYPE      3
#define FLOATTYPE       4
#define SERIALTYPE      (0x0100 + LONGTYPE)
#define SERIALSIZE      LONGSIZE
#define DATETYPE        (0x0200 + LONGTYPE)
#define EDATETYPE       (0x0300 + LONGTYPE)
#define YDATETYPE       (0x0400 + LONGTYPE)
#define DATESIZE        LONGSIZE
#define MONEYTYPE       (0x0500 + DOUBLETYPE)
#define MONEYSIZE       DOUBLESIZE
```

```
struct value
    {
    short v_type;                   /* type of value, determines    */
    union                           /* which of following is valid  */
        {
        struct                      /* char value (not null terminated) */
            {
            char *vcp;              /* ptr to char string value     */
            short vidx;             /* unused                       */
            short vlen;             /* len of char string value     */
            } vchar;
        int vint;                   /* integer value                */
        long vlng;                  /* long value (also dates)      */
        float vflo;                 /* float value                  */
        double vdub;                /* double value (also money)    */
        } v_val;
    };

#define v_charp         v_val.vchar.vcp
#define v_index         v_val.vchar.vidx
#define v_len           v_val.vchar.vlen
#define v_int           v_val.vint
#define v_long          v_val.vlng
#define v_float         v_val.vflo
#define v_double        v_val.vdub

extern double round();

/*
 * the following defines are for money types
 *      money is stored as cents in a double value
 *      CENTS converts a double value of dollars to money type (cents)
 *      DOLLARS converts a money type (cents) to double (dollars)
 *      ROUND just rounds a value to integral number
 */
#define CENTS(d)        (round((d)*100.0))
#define DOLLARS(c)      ((c)/100.0)
#define ROUND(c)        (round(c))


typedef struct value * perfvalue;
extern struct value retstack;


#define intreturn(i)    retstack.v_type=INTTYPE;\
                        retstack.v_int=(i);\
                        return(&retstack)

#define lngreturn(i)    retstack.v_type=LONGTYPE;\
                        retstack.v_long=(i);\
                        return(&retstack)

#define floreturn(d)    retstack.v_type=FLOATTYPE;\
                        retstack.v_float=(d);\
                        return(&retstack)
```

```
#define dubreturn(d)    retstack.v_type=DOUBLETYPE;\
                        retstack.v_double=(d);\
                        return(&retstack)

#define strreturn(s,c)  retstack.v_type=CHARTYPE;\
                        retstack.v_charp=(s);\
                        retstack.v_len=(c);\
                        return(&retstack)

extern int toint();     /* toint() takes a pointer to value structure
                         *         as an argument.
                         *         Returns the value (converted to integer)
                         *         of the value structure (v_int).
                         */
extern long tolong();   /* tolong() takes a pointer to value structure
                         *         as an argument.
                         *         Returns the value (converted to long)
                         *         of the value structure (v_long).
                         */
extern double todouble();/* todouble() takes a pointer to value structure
                         *         as an argument.
                         *         Returns the value (converted to double)
                         *         of the value structure (v_double).
                         */
extern long todate();   /* todate() takes a pointer to value structure
                         *         (1st arg) and a type (2nd arg)
                         *         Returns the value (converted to long)
                         *         of the value structure (v_long).
                         *         The type is used only if the value was a
                         *         CHARACTER type -- it is set to DATETYPE,
                         *         YDATETYPE, or EDATETYPE
                         */
extern double tomoney();/* tomoney() takes a pointer to value structure
                         *         as an argument.
                         *         Returns the value (converted to double)
                         *         of the value structure (v_double).
                         */
extern double tofloat();/* tofloat() takes a pointer to value structure
                         *         as an argument.
                         *         Returns the value (converted to double)
                         *         of the value structure (v_double).
                         */

#define intcon  toint
#define longcon tolong
#define dubcon  todouble

struct ufunc
        {
        char *uf_id;
        struct value *(*uf_func)();
        };
```

```
/*
 * The structure declaration for "userfuncs" must be put in the
 * user's data area. A hypothetical case using the user C functions
 * called "userfunc1" and "userfunc2" is shown below.
 *
 *  extern perfvalue userfunc1(); These routines must be externed before
 *  extern perfvalue userfunc2();  the userfuncs structure is initialized
 *
 *  struct ufunc userfuncs[] =
 *       {
 *       "userfunc1", userfunc1,
 *       "userfunc2", userfunc2,
 *                    '----------Pointer to the user function.
 *       |-------------------------The name of the user function
 *                                 as defined in the DEFINE statement of ACE.
 *       0,0 <-------------------Note that this array must be terminated
 *                                  by two zeros.
 *       };
 *
 * These structures are required so that ACE can call the user subroutines
 * at run time.
 */


/*these are c variable type names, used in calling c functions provided*/

#define CCHARTYPE       0
#define CSHORTTYPE      1
#define CINTTYPE        2
#define CLONGTYPE       3
#define CFLOATTYPE      4
#define CDOUBLETYPE     5
#define MAXCTYPE        6
```

9.41 Archivo de Cabecera: dbio.h

El archivo de cabecera dbio.h contiene definiciones en C que son usadas en las llamadas de las rutinas de ALL-II. Esas definiciones son usadas como banderas para indicar a las rutinas los deseos del programador. Por ejemplo, el comando DBSELECT espera recibir cualquiera de estos valores:

```
DBOPEN
DBCLOSE
FILEOPEN
FILECLOSE
```

Basado en este valor la rutina puede determinar si el programador quiere abrir o cerrar un archivo o una base de datos.

Para incluir este archivo en un programa de aplicación solo basta escribir la siguiente línea:

```
#include <dbio.h>
```

Acontinuación se presenta el contenido de éste archivo de cabecera:

```
/*************************************************************
 *
 *              Relational Database Systems, Inc.
 *
 *  Informix -- Relational Database Management System
 *
 *  Title:      dbio.h
 *  Sccsid:     @(#)dbio.h      1.4     10/31/84  16:20:33
 *  Description:
 *              A.L.L. user macro file
 *
 *************************************************************
 */

#define DBOPEN      1
#define DBCLOSE     2
#define FILEOPEN    3
#define RELOPEN     3
#define FILECLOSE   4
#define RELCLOSE    4

#define OPENMASK            0x00ff
#define EXCLUSIVE           0x0100
#define READONLY            0x0200
#define READWRITE           0x0000

#define COMPARISON  1
#define FIRST       2
#define LAST        3
#define NEXT        4
#define PREVIOUS    5
```

```
#define EQUAL       6
#define GTEQ        7
#define GREATER     8
#define CURRENT     9
#define LOCK        0400

/* values passed to dbselfield() */
#define ACCKEYED        0
#define ACCPRIMARY      1
#define ACCSEQUENTIAL   2

struct dbview
    {
    char *vwname;       /* name of attribute    */
    int vwstart;        /* offset in view       */
    int vwtype;         /* type of attribute    */
    int vwlen;          /* length in bytes      */
    };

#define CHARTYPE        0
#define INTTYPE         1
#define INTSIZE         2
#define LONGTYPE        2
#define LONGSIZE        4
#define DOUBLETYPE      3
#define DOUBLESIZE      (sizeof(double))
#define FLOATTYPE       4
#define FLOATSIZE       (sizeof(float))
#define SERIALTYPE      (0x0100 + LONGTYPE)
#define SERIALSIZE      LONGSIZE
#define DATETYPE        (0x0200 + LONGTYPE)
#define EDATETYPE       (0x0300 + LONGTYPE)
#define YDATETYPE       (0x0400 + LONGTYPE)
#define DATESIZE        LONGSIZE
#define MONEYTYPE       (0x0500 + DOUBLETYPE)
#define MONEYSIZE       DOUBLESIZE
#define COMPOSTYPE      (-1)
extern double round();

#define CENTS(d)        (round((d)*100.0))
#define DOLLARS(c)      ((c)/100.0)
#define ROUND(c)        (round(c))

/* values returned by dbindex() */
#define NOINDEX 0
#define NODUPS  1
#define DUPS    2

/* permission flags */
#define PERM_READ       0x0001
#define PERM_UPDATE     0x0002
#define PERM_INSERT     0x0004
#define PERM_DELETE     0x0008
#define PERM_CONTROL    0x0010
#define PERM_ALL        0x001f
```

9.42 Archivo de Cabecera: Stdio.h

En este archivo de cabecera se definen ciertos parámetros necesarios para la entrada y salida estándar de información.

Para incluir este archivo en un programa de aplicación sólo basta con escribir la siguiente línea:

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

A continuación se presenta el contenido de éste archivo de cabecera:

```
/**
*
* This header file defines the information used by the standard I/O
* package.
*
**/
#define _BUFSIZ 512                 /* standard buffer size */
#define BUFSIZ 512                  /* standard buffer size */
#define _NFILE 20                   /* maximum number of files */

struct _iobuf
{
char *_ptr;                         /* current buffer pointer */
int _rcnt;                          /* current byte count for reading */
int _wcnt;                          /* current byte count for writing */
char *_base;                        /* base address of I/O buffer */
char _flag;                         /* control flags */
char _file;                         /* file number */
int _size;                          /* size of buffer */
char _cbuff;                        /* single char buffer */
char _pad;                          /* (pad to even number of bytes) */
};

extern struct _iobuf _iob[_NFILE];

#define _IOREAD 1                   /* read flag */
#define _IOWRT 2                    /* write flag */
#define _IONBF 4                    /* non-buffered flag */
#define _IOMYBUF 8                  /* private buffer flag */
#define _IOEOF 16                   /* end-of-file flag */
#define _IOERR 32                   /* error flag */
#define _IOSTRG 64
#define _IORW 128                   /* read-write (update) flag */

#if SPTR
#define NULL 0                      /* null pointer value */
#else
#define NULL 0L
#endif
#define FILE struct _iobuf          /* shorthand */
#define EOF (-1)                    /* end-of-file code */
```

```
#define stdin (&_iob[0])        /* standard input file pointer */
#define stdout (&_iob[1])       /* standard output file pointer */
#define stderr (&_iob[2])       /* standard error file pointer */

#define getc(p) (--(p)->_rcnt>=0? *(p)->_ptr++:_filbf(p))
#define getchar() getc(stdin)
#define putc(c,p) (--(p)->_wcnt>=0?((int)(*(p)->_ptr++=(c))):_flsbf(
#define putchar(c) putc(c,stdout)
#define feof(p) (((p)->_flag&_IOEOF)!=0)
#define ferror(p) (((p)->_flag&_IOERR)!=0)
#define fileno(p) (p)->_file
#define rewind(fp) fseek(fp,0L,0)
#define fflush(fp) _flsbf(-1,fp)
#define clearerr(fp) clrerr(fp)

FILE *fopen();
FILE *freopen();
long ftell();
char *fgets();

#define abs(x) ((x)<0?-(x):(x))
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<=(b)?(a):(b))
```

9.43 Entrada de Datos al Sistema del Ejercicio Presupuestal

Al realizarse el cierre de operaciones diario en el módulo CIF de cada unidad de control, se corre un programa que genera un archivo plano con los registros del día y con la estructura que muestra la figura 9.6:

NOMBRE DE CAMPO	LONGITUD	DIGITOS	POS.INIC.	POS.FINAL
Cen.Trab.Afectador	3		1	3
Cen.Trab.Afectado	3		4	6
Depto./Gcia.	5		7	11
Concepto Origen	6		12	17
Proyecto	8		18	25
Fecha Elaboración	6		26	31
Fecha Vencimiento	6		32	37
Clave Autorización	11		38	48
Momento Contable	4		49	52
Importe M.N.	16	2	53	68
Moneda	1		69	69
Tipo Cambio	11	2	70	80
Importe Dlls.	16	2	81	96
Número CIF	4		97	100
Documento Fuente	10		101	110
Ficha Responsable	6		111	116
Unidad Control	3		117	119
Folio Adefas	8		120	127
Switch	2		128	129
Conc. Origen I.V.A.	6		130	135
Importe I.V.A.	16		136	151

T O T A L	151			

Figura 9.6

Este archivo se transmite (mediante un paquete de comunicaciones), al ambiente en que se encuentra S.I.C.E.P. (Minicomputador HP-9000), en donde se agrega la información proveniente de las diferentes unidades de control y se consolida a nivel Gerencia, Centro de Trabajo, Renglón de Gasto y Moneda. Tal y como lo muestra la figura 9.7.

Al término del mes y cuando todas las unidades de control han cerrado, S.I.C.E.P. genera seis diferentes archivos planos que contienen:

1) IMP_DEVE.TXT
   Presupuesto Autorizado Devengable

2) IMP_FEFE.TXT
   Presupuesto Autorizado Flujo de Efectivo

TRANSMISION DE INFORMACION
MODULO C.I.F. - S.I.C.E.P.

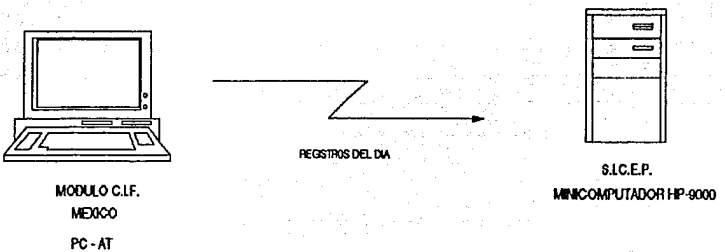


Figura 9.7

3) DEV_ACUM.TXT
Ejercicio del Presupuesto Devengado de Rama

4) AFE_ACUM.TXT
Ejercicio del Presupuesto Devengado Institucional

5) COM_ACUM.TXT
Ejercicio del Presupuesto Comprometido de Rama

6) AFE_FEFE.TXT
Ejercicio del Presupuesto Flujo de Efectivo (Pagado)

y cuya estructura se presenta en la figura 9.8:

NOMBRE DEL CAMPO	LONGITD	POS.INIC.	POS.FINAL
Gerencia	5	1	5
Centro de Trabajo	3	6	8
Renglón de Gasto	3	9	11
Moneda	1	12	12
Año	2	13	14
Enero	15	15	29
Febrero	15	30	44
Marzo	15	45	59
Abril	15	60	74
Mayo	15	75	89
Junio	15	90	104
Julio	15	105	119
Agosto	15	120	134
Septiembre	15	135	149
Octubre	15	150	164
Noviembre	15	165	179
Diciembre	15	180	194

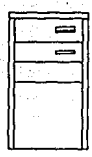
T O T A L	194		

Figura 9.8

Estos archivos son ahora transmitidos a la PC que contiene el **Sistema de Control del Ejercicio Presupuestal**, mediante la ayuda de otro paquete de comunicaciones para su rápida explotación. Tal y como lo muestra la figura 9.9.

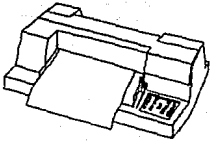
Para cargar estos archivos planos a los archivos propios de informix se corren varios procesos (dependiendo de la fase del presupuesto), los cuales preparan el ambiente y suben la información a los archivos correspondientes de la base de datos.

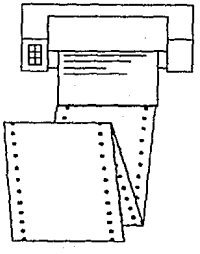
TRANSMISION DE INFORMACION
S.I.C.E.P. - S.C.E.P.



S.I.C.E.P.
MINICOMPUTADOR HP-9000

S.C.E.P.
S. T. L
PC - AT





A NIVEL GCIA/CTRAB/RENGLON/MONEDA

- PRESUPUESTO AUTORIZADO DEVENGABLE
- PRESUPUESTO AUTORIZADO FLUJO DE EFECTIVO
- EJERCICIO DEL PRESUPUESTO COMPROMETIDO (RAMA)
- EJERCICIO DEL PRESUPUESTO DEVENGADO (RAMA)
- EJERCICIO DEL PRESUPUESTO DEVENGADO INSTITUCIONAL
- EJERCICIO DEL PRESUPUESTO FLUJO DE EFECTIVO (PAGADO)

Figura 9.9

En la figura 9.10 se presenta la correspondencia que existe entre los archivos planos extraidos del S.I.C.E.P. y los archivos que forman parte del Sistema de Control del Ejercicio Presupuestal y que representan cada una de las fases presupuestales. Se muestra la estructura de los archivos así como el campo llave que sirve para subir la información.

El programa que se encarga de subir la información fué realizado en c, y existe uno para cada fase presupuestal, los parámetros que cambian son solamente los nombres de los archivos. Como ejemplo se utilizará la fase del devengado de rama. El esquema del archivo así como el programa en c se presentan acontinuación:

ESQUEMA DKDEV.- DEVENGADO DE RAMA

```
database bdpre

file kaldev

field pdger type char length 5 index dups
field pdctr type char length 3 index dups
field pdcon type char length 3 index dups
field pdmon type char length 1 index dups
field pdene type double
field pdfeb type double
field pdmar type double
field pdabr type double
field pdmay type double
field pdjun type double
field pdjul type double
field pdago type double
field pdsep type double
field pdoct type double
field pdnov type double
field pddic type double
field pdtot type double
field pdkey composite pdger, pdctr, pdcon, pdmon, index primary
end
```

ARCHIVOS QUE FORMAN PARTE DEL
SISTEMA DE CONTROL DEL EJERCICIO PRESUPUESTAL

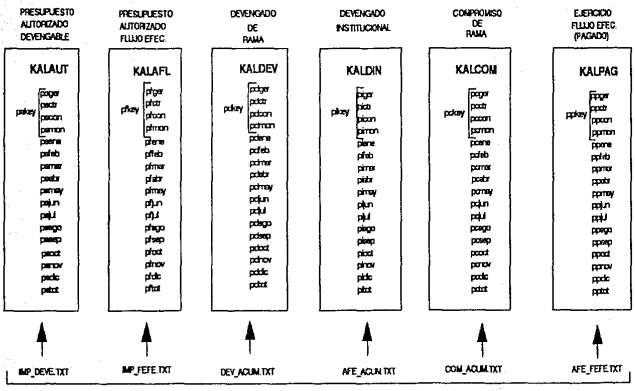


Figura 9.10

```
/*   devram03.c

     Programa  en  "ALL"  que  convierte  un  archivo  plano
     (dev_acum.txt)  a  un  archivo  tipico  de  informix  (Kaldev
     Devengado de Rama)

     Para compilar:
       cc -mp devram03.c -llibdb -lm
*/

#include <stdio.h>
#include <dbio.h>
#define OK 0
double atof();

struct dbview devflds[] =
 {
       {"pdger"}, {"pdctr"}, {"pdcon"}, {"pdmon"},
       {"pdene"}, {"pdfeb"}, {"pdmar"}, {"pdabr"}, {"pdmay"}, {"pdjun"},
       {"pdjul"}, {"pdago"}, {"pdsep"}, {"pdoct"}, {"pdnov"}, {"pddic"},
       {"pdtot"}

 };


struct
 {
       char   pdger[6], pdctr[4], pdcon[4], pdmon[2];
       double pdene, pdfeb, pdmar, pdabr, pdmay, pdjun;
       double pdjul, pdago, pdsep, pdoct, pdnov, pddic, pdtot;
 }devreg;


main()
{

FILE *fopen(), *fp;

  char ger[6], ctr[4], con[4], mon[2], ano[3],
  char aene[15], afeb[15], amar[15], aabr[15], amay[15], ajun[15],
  char ajul[15], aago[15], asep[15], aoct[15], anov[15], adic[15];
  double bene, bfeb, bmar, babr, bmay, bjun, bjul, bago, bsep;
  double boct, bnov, bdic;
  int longi=12, cc;

  dbselect(DBOPEN, "bdpre");
  dbselect(FILEOPEN, "kaldev");
  dbstructview("kaldev", devflds,17);
  dbselfield("kaldev", "pdkey", ACCKEYED);

  if((fp=fopen("dev_acum.txt","r")) == NULL) {
      printf("NO EXISTE EL ARCHIVO DE ENTRADA\n");
      printf("PROCESO TERMINADO\n");
      exit(1);
  }
```

```
 while(fscanf(fp,"%5s%3s%3s%1s%2s%s%s%s%s%s%s%s%s%s%s%s%s",ger,ctr,con,m
    ano,aene,afeb,amar,aabr,amay,ajun,ajul,aago,asep,aoct,anov,adic)>0)
  {
   strncpy(devreg.pdger,ger,5);
   strncpy(devreg.pdctr,ctr,3);
   strncpy(devreg.pdcon,con,3);
   strncpy(devreg.pdmon,mon,1);
   bene = atof(aene);
   bfeb = atof(afeb);
   bmar = atof(amar);
   babr = atof(aabr);
   bmay = atof(amay);
   bjun = atof(ajun);
   bjul = atof(ajul);
   bago = atof(aago);
   bsep = atof(asep);
   boct = atof(aoct);
   bnov = atof(anov);
   bdic = atof(adic);
   devreg.pdene = bene;
   devreg.pdfeb = bfeb;
   devreg.pdmar = bmar;
   devreg.pdabr = babr;
   devreg.pdmay = bmay;
   devreg.pdjun = bjun;
   devreg.pdjul = bjul;
   devreg.pdago = bago;
   devreg.pdsep = bsep;
   devreg.pdoct = boct;
   devreg.pdnov = bnov;
   devreg.pddic = bdic;
   devreg.pdtot = (bene+bfeb+bmar+babr+bmay+bjun+bjul+bago+bsep+boct+bnov+
   dbadd("kaldev",devreg);
  }
  dbselect(FILECLOSE, "kaldev");
  fclose(fp);
  dbselect(DBCLOSE, "bdpre");
}
```

9.44 Beneficios

Entre los principales beneficios que se tienen al utilizar programas en C para la explotación de la información del Sistema de Control del Ejercicio Presupuestal se encuentran:

a) Consolidación de Información al nivel de la Subdirección de Transformación Industrial de PEMEX.

b) Comparación del Presupuesto contra lo ejercido en diferentes niveles de agregación y por cada fase presupuestal, Compromiso, Devengado de Rama, Devengado Institucional y Pagado.

c) Información oportuna, segura y veraz para la toma de desiciones.

d) Control de posibles desviaciones del Presupuesto.

CONCLUSIONES

CONCLUSIONES

Al llegar al término de éste trabajo se ha llegado a la conclusión de que el lenguaje C representa una poderosa herramienta con que puede contar la ingeniería para el desarrollo de sistemas, ya que permite aprovechar al máximo, el rendimiento de una computadora, proporciona el acceso a instrucciones reales de una máquina, manipulación directa de bits, bytes, direcciones de memoria, control de puertos, etc..

C es un lenguaje de propósito general que puede aplicarse en diferentes áreas, además de que permite la programación estructurada, permite economizar expresiones mediante el manipuleo de sus múltiples operadores, tipos de datos (básicos y avanzados), sentencias de control (selección, Iteracción, salto), variables (automáticas, externas, estáticas, de registro) y constantes. Otras utilidades que posee son los arrays, punteros, estructuras, campos de bits, uniones y enumeraciones, características que hacen de C un lenguaje altamente poderoso.

Los array y los punteros juegan un papel muy importante en la programación con lenguaje C. Los array son en realidad posiciones de memoria contigua que almacenan datos de un determinado tipo y que son referenciados por un único identificador. Los punteros son variables que contienen direcciones de memoria ya sea de otras variables, elementos de un array, funciones, archivos y estructuras. El manejo de punteros es lo más atractivo que tiene C, pero a su vez puede ser lo más conflictivo si no se saben operar adecuadamente.

En C se pueden definir tipos de datos a medida es decir, se pueden crear tipos de datos propios deacuerdo a nuestras necesidades, un ejemplo de ello son las estructuras, que son un conjunto de variables de igual o diferentes tipos a las que se les puede hacer referencia mediante un nombre único, proporcionando así, un medio eficaz para mantener junta la información relacionada.

Los campos de bits es un tipo especial de estructura en donde se puede acceder a un bit individual dentro de un byte, este tipo es de gran utilidad ya que ciertos dispositivos transmiten información codificada en los bits dentro de bytes.

Y por último las uniones que representan una posición de memoria que es compartida por dos o más variables de diferentes tipos dando como consecuencia un ahorro de memoria que en determinadas aplicaciones puede ser vital.

El lenguaje C tiene la ventaja de que un programador pueda construir poco a poco sus funciones de biblioteca, es decir construir rutinas para determinadas tareas, e incluirlas cuando sea necesario como archivos de cabecera en programas, o bién se pueden compilar por separado y linkearlas para construir un archivo ejecutable. C puede dividir un problema en partes, en

donde cada una de ellas puede ser una función, sus objetivos serán específicos y fundamentales para solucionar el problema general.

Es importante mencionar que la mayoría de las implementaciones que tiene C, cuenta con una colección de archivos de cabecera que ayudan a proporcionar información necesaria para que se puedan utilizar funciones de biblioteca, como son las funciones de E/S estándar, E/S de archivos, funciones matemáticas, funciones de hora y fecha, funciones para el control de memoria, funciones gráficas, funciones para control de dispositivos, funciones para comunicaciones, etc.

Una característica sumamente valiosa del lenguaje C es que el código es altamente portable ya que es posible adaptar el software escrito para un tipo de máquina en otra; aunque el procesador y el sistema operativo sean diferentes. Sólo basta con recompilar el código fuente usando el compilador de la nueva máquina.

C puede ser usado para la construcción de Sistemas Operativos, manejadores de bases de datos, compiladores, intérpretes, editores, hojas de cálculo, simuladores, paquetes gráficos, juegos de video, sistemas hechos a medida, manipulación de impresoras, movimientos del mouse, programas de modem, etc..

El desarrollar sistemas es un trabajo arduo que requiere tener una visión amplia del objetivo, seguir procedimientos, y fijarse metas. Cuando se agota al máximo el medio para lograrlo, se tiene que valer de una herramienta que ayude a seguir adelante con el proyecto.

Al realizar una nueva versión del Sistema del Ejercicio Presupuestal y del Módulo C.I.F. se tuvo que valer del lenguaje C para cumplir con su objetivo, ya que son sistemas que por su importancia y gran complejidad requieren una alta manipulación de la información para lograr el Control del Ejercicio Presupuestal de la Subdirección de Transformación Industrial de Petróleos Mexicanos.

Como lo mencionamos en su momento el objetivo de éste trabajo no es el mostrar la planeación, análisis y diseño de éstos sistemas, ya que bién podrían ser tema de desarrollo de otro trabajo. Estos sistemas han tenido diversas versiones desde su implemetación original, si alejarse por supuesto de su objetivo principal.

El Módulo CIF en un principio fué desarrollado por dos analistas, alrededor de 1980 y fué implementado además de las oficinas centrales, en las unidades de control de cada una de las 7 refinerías y de los 17 centros petroquímicos, teniendo encuenta que en cada unidad se contara con una PC-XT, disco duro de 20 Mb, un drive de 5 1/4", y una impresora. En la actualidad ante los diversos cambios que ha tenido Petróleos Mexicanos en su estructura el módulo C.I.F. ha cambiado, la última versión,

implantada solamente en México y en las refinerías, requiere de una PC-AT 386, monitor VGA, disco duro mínimo de 80 MB, drives de alta densidad y la compra de paquetes de comunicaciones y modems para poder establecer una comunicación entre el módulo C.I.F. con S.I.C.E.P.

El sistema de Control del Ejercicio Presupuestal, fué desarrollado por un sólo analista, e implantado solamente en las oficinas centrales en México. Al igual que el Módulo C.I.F. ha tenido diversas versiones, comenzó con el mismo equipo que utilizó el C.I.F. agregándole un graficador, en la actualidad se maneja el sistema en una PC-AT 386, 80 MB de capacidad de disco duro, monitor VGA, drives de alta densidad, impresora rápida, modem y también se adquirió otro tipo de paquete de comunicaciones para establecer una interface con S.I.C.E.P.

Todo ésto para obtener el beneficio que acarrea el llevar un control presupuestal de las áreas de la S.T.I., consolidación del ejercicio presupuestal, presupuestar los recursos financieros, para mano de obra, realización de obras de inversión, trabajos de mantenimiento, adquisiones, etc. Aspectos tan importantes para llevar acabo la denominada "Toma de Desiciones".

En este trabajo se presentaron todas las rutinas hechas en C para el mejoramiento de los sistemas, la inclusión de archivos de cabecera para preparar el ambiente en el que se está trabajando, utilización de librerías de ALL-II C que sirven de gran ayuda para poder manipular mejor los archivos de informix y manejar el cursor a nuestro antojo en pantallas de captura, actualización y consulta, las llamadas a las rutinas desde informix, validaciones deacuerdo a archivos de catálogo, conversiones de tipo de cambio, realizar niveles de agregación en archivos, cargar archivos planos en archivos típicos de informix, relacionar archivos y procesar la información, por último la interface que existe entre la pantalla principal del Módulo C.I.F. y las rutinas en C compiladas con Laticce ver. 3.0.

Para concluir, C es un lenguaje realmente confiable, simple, y eficaz, factores que lo hace ser un lenguaje altamente poderoso y útil para la Ingeniería de Sistemas.

BIBLIOGRAFIA

B I B L I O G R A F I A

"C: Manual de Referencia"
Segunda Edición
Herbert Shildt
Editorial: Osborne/McGraw-Hill

"Programación en C"
Byron S. Gottfried
Editorial: McGraw-Hill

"ANSI C a su alcance"
Herbert Shildt
Editorial: Osborne/McGraw-Hill

"Microsoft C Secrets, Shortcuts and Solutions"
Kris Jamsa
Editorial: Microsoft Press

"Curso de Programación con Microsoft C"
Fco. Javier Ceballos
Editorial: Macrobit

"C Manual de Bolsillo"
Alan C. Plantz
Editorial: Addison-Wesley Iberoamericana

"Lenguaje C, Introducción a la Programación"
Kelley/Pohl
Editorial: Addison-Wesley Iberoamericana

"Programación en Turbo C"
Segunda Edición
Herbert Schildt
Editorial: Borland-Osborne/McGraw-Hill

"Turbo C Porgramación Avanzada"
Segunda Edición
Herbert Shildt

"Lenguaje C Biblioteca de Funciones"
Kris Jamsa
Editorial: Osborne/McGraw-Hill

"Introducción al Lenguaje C"
Segunda Edición
Les Hancock
Morris Krieger
Editorial: Byte Books/McGraw-Hill

"Lenguaje C Programación Avanzada"
Herbert Schildt
Editorial: Osborne/McGraw-Hill

"C, Guía para Usuarios Expertos"
Herbert Schildt
Editorial: Osborne/McGraw-Hill

"El Lenguaje de Programación C"
Brian W. Kernighan
Dennis M. Ritchie

"Advance QuickC"
Second Edition
Werner feibel
Editorial: Osborne/McGraw-Hill

"Utilización de C en Iteligencia Artificial"
Herbert Schildt
Editorial: Osborne/McGraw-Hill

"Manual del Compilador Lattice Ver. 3.0"
December 6, 1985
Lattice, Incorporated
P.O. Box 3072
22W600 Butterfield Road
Glen Ellyn, IL 60137