Memoria dinámica en el lenguaje de programación C

Cenid Editorial

Memoria dinámica en el lenguaje de programación C

Joel Ayala de la Vega
Irene Aguilar Juárez
Alfonso Zarco Hidalgo
Hipólito Gómez Ayala

En Cenid estamos a sus órdenes si desea:

☐ Asesoría para elaborar un libro

☐ Publicar un texto
Contáctenos

Teléfono: 3315 420983
www.cenid.org.mx
redesibero@yahoo.com.mx
Pompeya Nº 2705 Col. Providencia
C.P. 44630 Guadalajara, Jalisco, México

Diagramación y cubierta: Esther Ramírez Lara

Memoria dinámica en el lenguaje de programación C

ISBN: 978-607-8435-19-7
Primera edición

Miembro de la Cámara Nacional de la Industria Editorial Mexicana Socio # 3758

Impreso en México / Printed in Mexico

Índice

Índice

UN POCO DE HISTORIA

El lenguaje de programación C históricamente está ligado a la evolución de los sistemas operativos. Para comprender mejor esto, basta remontarse al año 1964, cuando los planes de desarrollo del S. O. MULTICS (Multiplexed Information and Computing Service) eran proporcionar alta disponibilidad, de manera que el servicio de cómputo tenía que igualarse a los servicios de telefonía y a la red eléctrica. Tras un tiempo dicha meta fracasó, debido principalmente a que fue escrito en PL/1, cuyo compilador no se encontraba en el mercado; después, cuando finalmente este salió las dificultades para su manipulación fueron muchas, además de que era un proyecto bastante ambicioso para su época, casi como lo fue el de Charles Babbage en el siglo XIX.

Cuando Bell Labs abandonó el proyecto en 1969, uno de sus investigadores, Ken Thomson, hizo algo interesante. Decidió volver a escribir un nuevo MULTICS (esta vez en lenguaje ensamblador), pero con una filosofía diferente, centrándose en mantener el sistema tan pequeño y simple como fuese posible; de esa mamera fue corrigiendo en MULTICS lo que se veía como una deficiencia. La primera implementación la realizó en una minicomputadora PDP-7. A pesar del poco tamaño de la PDP-7, el sistema de Thomson funcionó. En consecuencia, otro investigador de Laboratorios Bell, Brian Kernighan, lo denominó de manera un tanto burlona UNICS (Sistema de Información y cómputo Unipelxión), haciendo alusión a un eunuco, a un MULTICS castrado. Dicho nombre se aceptó, aunque más adelante fue cambiado a UNIX.

El trabajo de Thomson impresionó de tal forma que posteriormente se unió todo un departamento de Laboratorios Bell. El siguiente paso fue trasladar UNIX de la PDP-7 a una minicomputadora más moderna, la PDP-11/20, y posteriormente a la PDP-11/45 y la PDP-11/70. Estas dos últimas computadoras dominaron el mundo de las minicomputadoras durante gran parte de la década de los años setenta. La PDP-11/45 y la PDP-11/70 tenían hardware de protección a memoria, lo que permitió soportar varios usuarios al mismo tiempo.

El segundo desarrollo fue con respecto al lenguaje con que se escribió UNIX. En esa época, fue cada vez más evidente que no era divertido escribir de nuevo todo el sistema para cada nueva máquina, por lo que en 1970 Thomson decidió volver a escribir UNIX en un lenguaje de alto nivel con su propio diseño, llamado **B.** B era una forma simplicada de BCPL (que a su vez era una forma simplificada de CPL, que al igual que PL/1 nunca funcionó). Debido a ciertas debilidades de B, principalmente la carencia de estructuras, Ritchie diseñó entonces un sucesor de B, llamado C, y escribió un compilador excelente para él. Juntos, Thomson y Ritchie volvieron a escribir UNIX en C. C fue el lenguaje correcto en el momento correcto y ha dominado la programación de sistemas desde entonces. El contar con un lenguaje de alto nivel permitió el avance de los

sistemas operativos, el mismo código podía ser utilizado en las distintas plataformas, propiciando la reutilización de código y reduciendo los tiempos de desarrollo. Esto permitió que Sistemas Operativos basados en UNIX, el sistema BSD, e el sistema GNU/Linux y muchos otros fueran desarrollados en C.

En 1974, Ritchie y Thomson publicaron un artículo memorable relativo a UNIX. Por el trabajo descrito en ese artículo recibieron posteriormente el Premio Turing de la ACM. La publicación de ese artículo estimuló a muchas Universidades para pedir una copia de UNIX. Puesto que la compañía dueña de Laboratorios Bell, AT&T, era entonces un monopolio regulado y no podía entrar al negocio de las computadoras, no tuvo objeción en otorgar licencias de uso UNIX a las universidades a un bajo costo, así que toda una generación de estudiantes se educó con este sistema de computo, lo cual contribuyó a su difusión después de que ellos terminaron su licenciatura y fueron a trabajar en la industria.

Toda licencia de UNIX tenía a su vez un compilador de C; al difundirse UNIX a varias universidades, se propagó el uso de C. Pero C no está ligado a ningún sistema operativo o máquina en concreto y, aunque se lo ha llamado “lenguaje de programación de sistemas” a causa de su utilidad en la escritura de sistemas operativos, ha sido utilizado con el mismo éxito para escribir programas numéricos, programas de procesadores de texto, bases de datos, etcétera.

En 1978, Dennis Ritchie y Brian Kernighan publicaron la primera edición del libro *El lenguaje de programación C*. Este libro fue durante años la especificación informal del lenguaje. El lenguaje descrito en la primera edición de este libro fue conocido como «el C de Kernighan y Ritchie», o simplemente "K&R C".

En 1990 se publicó la estandarización ISO del lenguaje. Este estándar es básicamente el estándar ANSI, con unas pocas modificaciones de formato. A este estándar se le conoce, desde entonces, como C89 o C90, y se trata del mismo lenguaje.

En 1988, basándose en el estándar ANSI que estaba en preparación, Kernighan y Ritchie publicaron la segunda edición de su libro, que todavía es utilizado como una de las referencias principales del lenguaje.

Durante los siguientes años, el lenguaje C permaneció sin demasiados cambios. Sin embargo, como había sucedido antes, a los distintos compiladores se fueron incorporando características adicionales que otros compiladores no tenían, siendo C++ la principal influencia.

A finales de los años noventa se decidió revisar el estándar de C, lo que llevó a la publicación del estándar C99. Este estándar incluye varias características nuevas como son: las funciones inline; la posibilidad de declarar variables en cualquier parte del código; los comentarios de una sola línea utilizando //; los tipos de datos long long int, bool y complex, entre otras.

El proceso de evolución del lenguaje sigue avanzando, y desde 2007 se está trabajando en el armado de un nuevo estándar.

El campo de la documentación ha crecido con el lenguaje de programación C en estas décadas. En 1980, “K&R C” era el único referente del lenguaje de programación C. En este momento se tiene en el mercado una gran variedad de libros que hablan del lenguaje de programación C. Excepto en un área, en los apuntadores y en las operaciones sobre apuntadores es parcialmente inexplorado. La mayoría de los libros y otra documentación ofrecen un capítulo sobre apuntadores en C, pero son generalmente incompletos. Esto es comprensible porque en apuntadores con C se requiere al menos un libro para que valga la pena la explicación.

Para personas que estudian el lenguaje C, las Operaciones de apuntadores en C constituyen uno de los mayores obstáculos. Una buena base del conocimiento en apuntadores permite potencializar la programación. Se puede programar en C sin este recurso, pero la programación con apuntadores permite una mayor expresividad y una mayor eficiencia.

Una nota importante: los compiladores del lenguaje C son un subconjunto de la estandarización C, por lo que funcionan en forma diferente. La compilación depende la versión del compilador y el sistema operativo en el que se esté ejecutando, por lo que tal vez algún ejemplo de este libro no compile en forma apropiada en la versión que se esté usando. La mayoría de los ejemplos de este libro se compilaron tanto en Turbo C versión 3, Turbo C portable, como en Dev-C para Windows 7.

Alfonso Zarco Hidalgo

Ingeniero en Mecánica Agrícola por la Universidad Autónoma de Chapingo. Maestro y Doctor en Ciencias en Hidrociencias por el Colegio de Postgraduados. Teniendo las siguientes publicaciones: en 2010: Elementos para el Desarrollo de una Hidrología Operacional con Sensores Remotos: Suelo Desnudo en Tecnología y Ciencias del Agua, Vol. 1, pp.59-71, abril-junio de 2010; en 2012: Escalamiento Espacial de Medios Heterogéneos Espectrales Usando Invarianzas Temporales en Terra Latinoamericana, Vol. 30, núm. 4, pp. 315-326; en 2013: Estimación de las propiedades ópticas de la vegetación usando medios radiativos equivalentes y espacios n-paramétricos en Terra Latinoamericana, Vol. 31 Núm. 2 pp. 119-134, Abril-Junio de 2013; en 2014: Análisis de criterios de evaluación para la calidad de los materiales didácticos digitales en CTS Revista Ibero Americana de Ciencia, Tecnología y Sociedad, No 25, Vol. 9 y Plataforma de cómputo para el cálculo de desempleo en México en Aportaciones en Ciencias Sociales: Economía y Humanidades con ISBN 9786071203540; en 2015: Comparativa Sintáctica entre los lenguajes de programación Java y Groovy en la Revista Iberoamericana de las Ciencias Computacionales e Informática con ISSN 2007-9915. . Investigador Adjunto en el Colegio de Postgraduados en 1997. Gerente y jefe de Piso en Grease Monky de 1998 a 1999. Docente en el Centro Universitario UAEM Texcoco desde el 2000. Tutor Académico del Centro Universitario UAEM Texcoco desde el 2004. Participación como director y revisor en la dirección de tesis profesionales desde el 2009. Profesor de tiempo completo por la Universidad Autónoma del Estado de México. Perfil PROMEP.

Irene Aguilar Juárez

Ingeniera en Computación por la Universidad Autónoma del Estado de México. Maestra en Ingeniería en Sistemas Computacionales por el Tecnológico de Estudios Superiores de Ecatepec. Candidata a Doctora en Sistemas y Ambiente Educativo por la Universidad de Guadalajara. Profesora de tiempo completo por la Universidad Autónoma del Estado de México. Ha participado en congresos Nacionales e Internacionales, entre ellos el VII congreso Internacional de Educación a Distancia AMECYD 2012, CcITA 2012, CcITA 2013, CONTE 2014 y CcITA2015. Ha publicado en revistas como CTS, RIDE, RECI y PAG. También ha participado como autora de algunos capítulos de libros en libros especializados, obtuvo en el 2012 el reconocimiento Perfil PROMEP.

Hipólito Gómez Ayala

Nació en Texcoco Estado de México en el año 1965, su educación básica y media superior la realizó en Texcoco, estudio la carrera de Matemáticas en la Universidad Autónoma Metropolitana, obteniendo el título de “Matemático” en 1997, estudio el posgrado en la Universidad Autónoma del Estado de México obteniendo el grado de “Maestro en Ciencias de la Computación” en el año 2011. Experiencia docente: Ingresa a la Universidad Autónoma del Estado de México Campus Texcoco en 1997, al Centro Universitario Texcoco Francisco

Ferreira y Arreola de 1997 al año 2005, a la Universidad del Valle de México campus Texcoco del año 2000 al 2009, a Liceo Universidad Fray Pedro de Gante de 1999 a 2011. Y al Tecnológico de Estudios Superiores de Chimalhuacán se incorpora en el año 2008 a la fecha. La Universidad Autónoma del Estado de México le otorgó "Nota Laudatoria" en el año 2005. En el año 2013 el Tecnológico de Estudios Superiores lo nombra profesor de Tiempo Completo y en el 2014 es nombrado Subdirector Académico. Registró el libro "Ecuaciones Diferenciales Paso a Paso", en el Instituto Nacional del Derecho de Autor. Publica en enero 2014 el artículo "La media Derivada", En la revista número 2 "Ciencia y algo mas….", http://cienciayalgomas100.wix.com Cuenta con Diplomado en "Educación", y otro en "Tutoría", se ha actualizado en la toma de diferentes cursos de actualización docente y en su área profesional, concretamente matemáticas y cómputo.

Joel Ayala de la Vega

En 1982 obtuvo el título de Ingeniero Mecánico Electricista por la Universidad La Salle. En 1986 obtuvo el grado de Maestro en Computo Aplicado por el Colegio de Postgraduados. En 1995 Obtuvo la candidatura a doctor en el Departamento en Ciencias de la Computación de la Universidad de Tulane en Nueva Orleans, EU. En 2008 obtuvo el grado de Doctor en Educación por la Escuela Libre de Ciencias Políticas y Administración Pública de Oriente. Ha tomado 3 diplomados y varios cursos de actualización docente. Experiencia Laboral: 1980-1983. Ingeniero de diseño eléctrico industrial en Procesos de México, Ingeniería. 1985-1987. Profesor de asignatura en la Facultad de Ingeniería de la UNAM. 1987-1996. Investigador Docente del Colegio de Postgraduados. 1996-2011. Profesor de asignatura en diversas Universidades (Universidad Pedro de Gante, Universidad del Valle de México, Universidad Autónoma del Estado de México). 2011 a la fecha. Profesor de Tiempo Completo por la Universidad Autónoma del Estado de México. Reconocimientos: 1991-1994. Candidato a investigador por el Sistema Nacional de Investigadores (CONACyT). 2001. Nota Laudatoria por trayectoria académica por la UAEMex. 2014-2017. Perfil PROMEP por la SEP. Cargos académicos: 1998-2006: Presidente de Académica de Ingeniería en Computación del C. U. UAEM Texcoco. 2004-2011: Coordinador de Ingeniería en Computación del C. U. UAEM Texcoco. 2012-2015: Coordinador de la Maestría en Ciencias de la Computación del C. U. UAEM Texcoco. Director de 3 tesis a nivel Doctoral en el área de Educación. Asesor de 4 tesis de la Maestría en Ciencias de la Computación y de 3 tesis en la Maestría en Computo Aplicado. Director de 15 tesis en Ingeniería de la Computación y Revisor de 14 tesis en Ingeniería en Computación. Publicaciones: autor del libro "Sistema Opeativo MS-DOS Ver. 5". Editorial Colegio de Postgraduados en 1990. Autor o coautor de al menos 12 artículos en revistas indexadas o capítulos de libro.

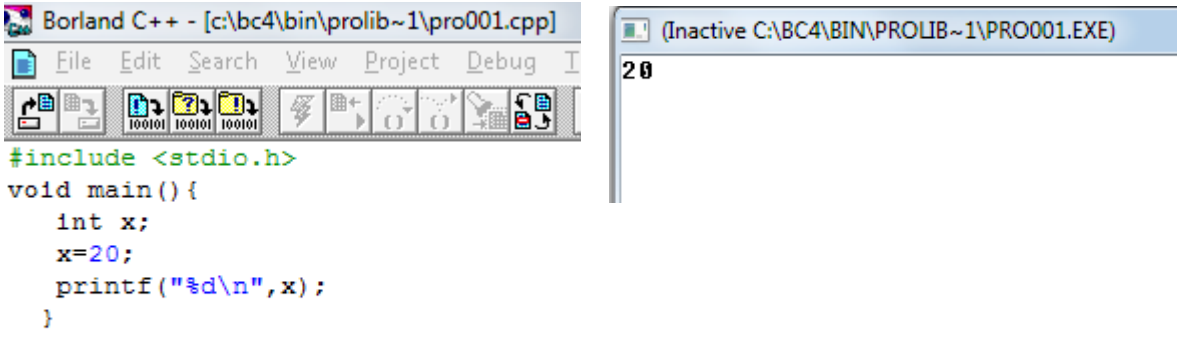
&, UNA CONSTANTE TIPO APUNTADOR

1.1 VARIABLES AUTOMÁTICAS

Un apuntador en el lenguaje C se comprende como una variable especial que guarda una dirección de memoria. Un apuntador no puede ser tratado como una variable estándar, por el adjetivo “especial”, esto indica que los apuntadores pueden ser tratados como variables estándar con algunos operadores, pero no pueden ser tratados en forma estándar con otros operadores. Una programación descuidada (aunque técnicamente correcta) en el uso de apuntadores puede producir una mezcolanza casi imposible de entender.

Las operaciones con apuntadores no son fáciles de entender, mientras no se comprendan los métodos utilizados en el lenguaje C y se fijen en la mente del programador. Estos métodos y reglas son manejadas por el compilador y son relativamente ocultas para el programador.

Iniciemos la discusión usando un programa simple como ejemplo:



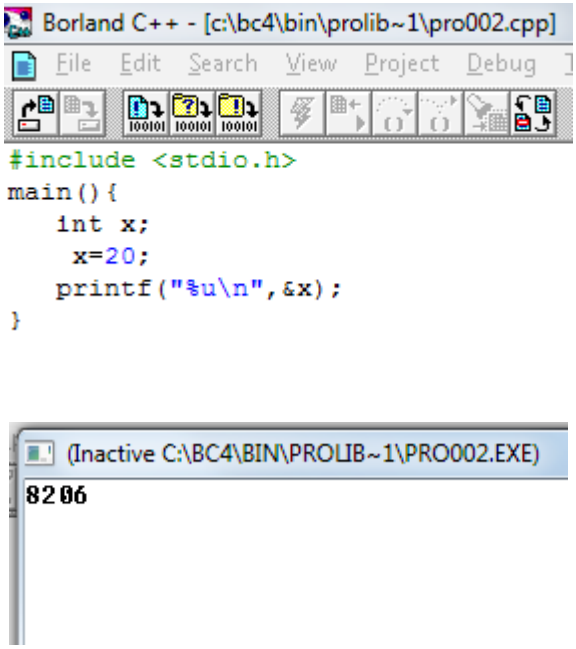
Programa 1.1

En el programa 1.1, -x- es declarada como variable estándar (auto) de tipo entero (int). Es asignado un valor y se muestra en la pantalla usando una función de escritura (printf()). ¿Qué sucede cuando la variable es declarada? De manera sucinta describimos este proceso: se reserva una localidad de la memoria con una extensión de dos bytes, para albergar un número entero (la mayoría de los compiladores utilizados en una microcomputadora soportan 2 bytes para guardar un entero).

La línea de asignación indica que en la localidad se va a almacenar el valor 20. Siendo la declaración una variable entera con signo, indica que el rango de almacenamiento es desde 32.767 hasta -32.768 (El número de bytes depende del compilador, en este caso se están usando las especificaciones de Turbo C).

Cuando la función escribe (printf()) es ejecutada y el valor colocado en –x- se desplegará en la pantalla como un entero.

Lo importante en esta declaración elemental es que se reservan dos bytes de almacenamiento "privado" para la variable –x-. Se puede acceder al espacio de memoria utilizando el operador "&" (también conocido como el "operador de direccionamiento"). En el programa 1.2 se muestra el valor de una dirección de memoria, y al ejecutarse el programa puede aparecer un valor cualquiera que depende del compilador, del sistema operativo y del tipo de máquina. Este valor es la dirección donde se guarda el valor de la variable –x- .



Programa 1.2

Siendo precedida la variable –x- con el ampersand, causa que la localidad de memoria, también llamada el "inicio de la dirección" de la variable, sea retornada en vez del valor entero guardado en la localidad de memoria. -&x- siempre hace referencia al inicio de la

dirección de memoria dada para la variable –x-. Nosotros no podemos cambiar esta dirección de memoria, es fija (o constante).

En la función de impresión (printf()), la conversión "%u" se utiliza para desplegar un entero sin signo. Esto es necesario ya que la dirección se localiza en la parte alta del segmento de la memoria pequeña (64 k bytes utilizados en la mayoría de los compiladores en C para las microcomputadoras). El valor que se muestra en la pantalla depende de la configuración de la máquina, compilador, sistema operativo y uso de la memoria.

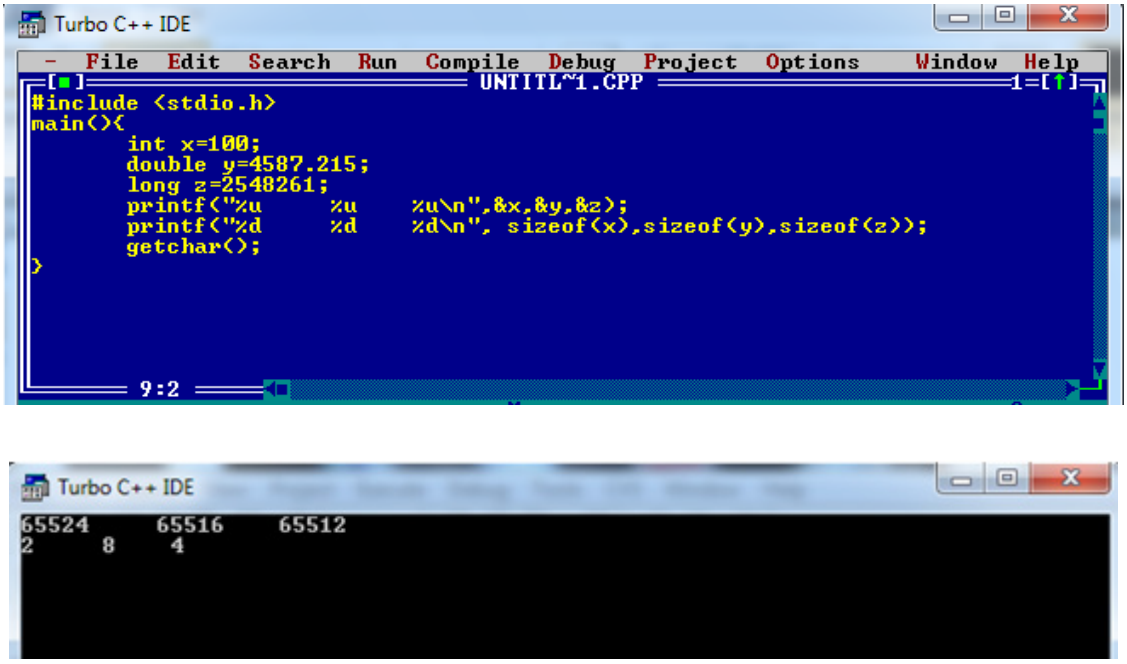
El punto importante a recordar es que la variable –x- es declarada una variable de tipo entero (int) y que &x es una constante (al menos durante la corrida del programa que estamos analizando). No puede ser cambiado y se encuentra atado a la variable x siendo igual al inicio de dirección del almacenamiento de la variable –x-. En otras palabras, no se puede colocar a la dirección de x (-&x-) otro valor. Por definición y diseño, siempre es igual al inicio de memoria asignado a-x-. Por ejemplo, la expresión: &x=9990 es completamente errónea, porque &x solo puede realizar una sola cosa, **retornar el valor que indica la dirección de almacenamiento de la variable –x-**. (Siendo la variable –x- entera, lo que retorna &x es la dirección del primer byte de almacenamiento).

En C, cuando se declara una variable solo una operación básica se ejecuta, se asigna el espacio de memoria para cada variable declarada. Los bytes en esta localidad de memoria no son limpiados, en consecuencia contienen cualquier valor comúnmente denominado basura. Por lo general se comete el error al pensar que el compilador inicializa la variable en cero (lenguajes como BASIC inicializan sus variables en cero). Si el programador asume erróneamente la inicialización de las variables, puede producir graves errores al momento de la ejecución del programa, esto es importante ya que sucede lo mismo con los apuntadores. Todo apuntador no inicializado puede contener basura (Por ejemplo, para Turbo C será un valor entre 0 y 65535). Lo que hace un apuntador es contener o retornar una dirección de memoria. Lo que uno hace o deja de hacer con una dirección de memoria es responsabilidad del programador.

Cuando se declara una variable estándar, uno debe comprender que el espacio de almacenamiento es asignado automáticamente a la variable y se considera un almacenamiento "seguro". Cuando un programa es cargado y ejecutado, existen diferentes porciones de la memoria que son reservados para diferentes cosas. Una porción de área se reserva para el manejo e interface con el sistema operativo, esta es el área que permite al programa ser ejecutada en forma adecuada. El área de almacenamiento de variables no se localiza en tal área, ya que se puede producir una colisión, En consecuencia existe un área reservada

solo para el almacenamiento de variables. Cuando se tienen declaradas variables tipo apuntador, no están restringidas a un área segura como las variables tipo estándar y pueden contener la dirección de cualquier punto de la memoria. **La ejecución de un programa que contenga variables tipo apuntador puede resultar en una catástrofe si el programador no está consciente de lo que está haciendo con tales variables.**

Ahora bien, cuando se tienen que declarar *n* variables se tendrá una vecindad donde se localizarán las variables. Observe el programa 1.3:



Programa 1.3

La función sizeof() retorna como resultado el número de bytes que ocupa cada una de las variables que se envían como parámetro en la función.

En la ayuda de Turbo C se tienen el siguiente número de bytes por variable:

unsigned char	1 byte
char	1 byte
unsigned int	2 bytes
short int	2 bytes
int	2 bytes
unsigned long	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	10 byts

En la primera hilera del programa 1.3 en ejecución se muestra la dirección de memoria en donde se encuentra cada una de las tres variables, en la segunda hilera indica el número de bytes de cada tipo de variable. Observe que el apuntador está apuntando al primer byte de cada variable y que las variables se localizan en la memoria en forma continua (ver figura 1.1).

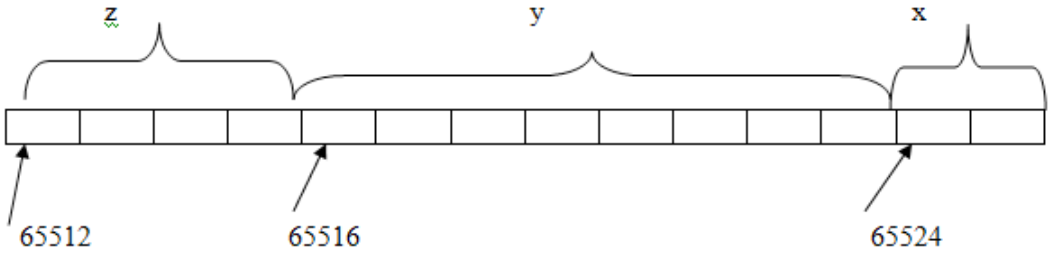
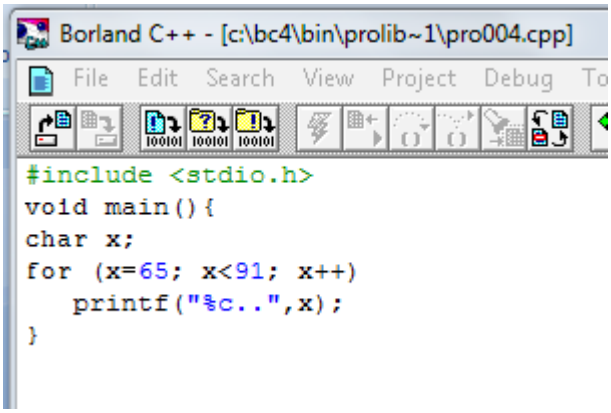


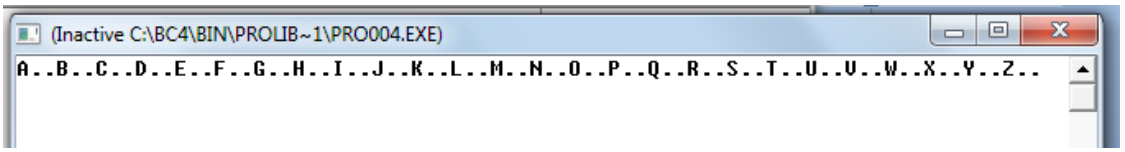
Figura 1.1

1.2 ARREGLOS DE CARACTERES

Cuando se trata con cadenas de caracteres en C, nos acercamos a operaciones puras de apuntadores. Sin embargo, cadenas en C, o arreglos de caracteres, también tienen una similitud con números y variables del tipo numérico. La creación y uso de de arreglos de caracteres es la siguiente discusión en apuntadores.

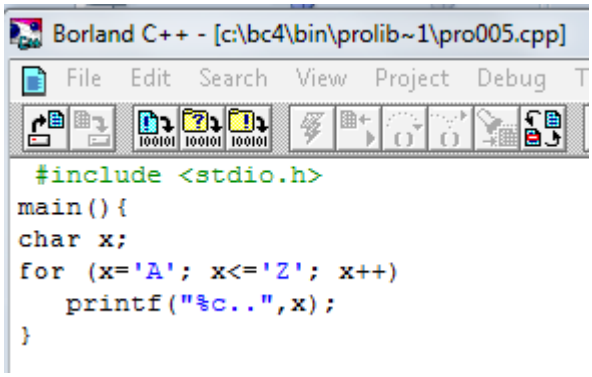
El programa 1.4 despliega en el monitor todas las letras del abecedario:

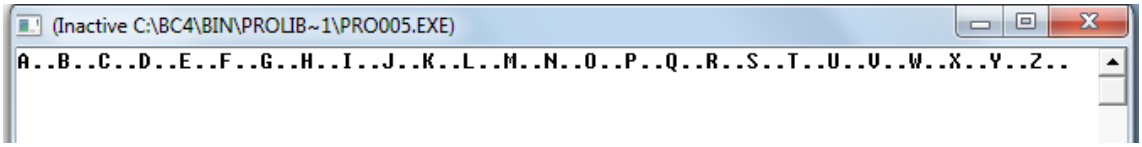




Programa 1.4

La línea de declaración muestra una variable x del tipo carácter. Esto significa que un simple byte es reservado para el uso exclusivo de esta variable. El valor estándar para una variable del tipo carácter consiste en un entero corto sin signo del rango 0 a 255. El valor 65 en decimal representa la –A- en ASCII. También se puede ejecutar el programa como lo muestra el programa 1.5:





Programa 1.5

En el lenguaje C, x= 'A' significa exactamente lo mismo que x=65. 'A' se traduce en compilador como 65.

La única razón por la cual se despliega la letra 'A' en la pantalla en vez de un número se debe a la designación de %c en la función printf(). Esto especifica que el valor en su argumento –x-, se desplegará como un carácter ASCII.

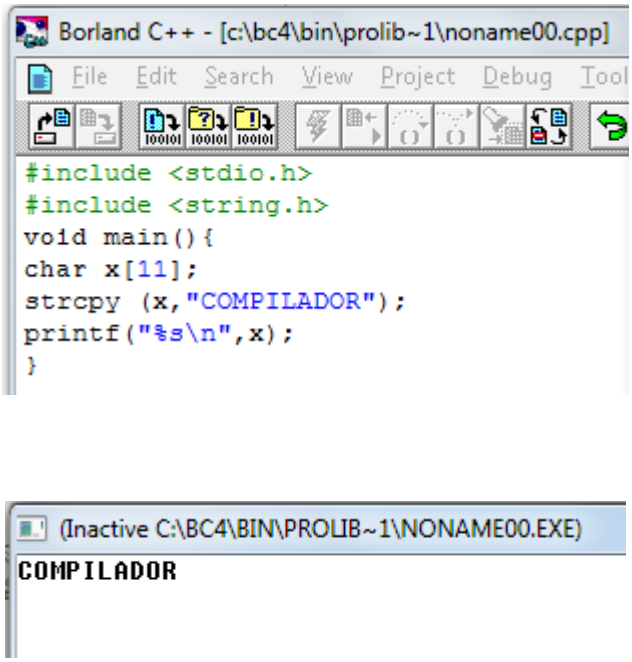
De la misma forma analizada con los enteros, la siguiente línea retorna la dirección de almacenamiento de la variable x:

printf("%u\n", &x)

Si se incluye la siguiente línea, lo que aparecerá en pantalla será el tamaño de almacenamiento de la variable –x-; siendo esta un carácter, da como resultado el valor de uno:

printf("%d\n", sizeof(x));

Es extraño observar dentro de un programa la declaración de una variable tipo carácter, lo más común es utilizar un arreglo de caracteres (o cadena de caracteres). Una cadena de caracteres se maneja como una simple unidad en la mayoría de los lenguajes de cómputo; sin embargo, C no tiene una variable tipo cadena de caracteres. En C todas las cadenas de caracteres son guardadas como un arreglo de caracteres donde cada carácter se almacena en un byte. Al ejecutar el programa, se desplegará en la pantalla la palabra –COMPILADOR-. Ver programa 1.6:



Programa 1.6

Con la declaración de la variable –x-, se coloca un espacio en memoria para once bytes. La razón por la que se solicitan 11 bytes cuando la palabra solo tiene 10 bytes es la siguiente: en C se define como cadena de caracteres a una unidad de caracteres terminados en el carácter NULL. El carácter NULL en ASCII es el valor cero (ocho bits con el valor de cero). La función strcpy() también copia el carácter NULL, por lo que se guarda en memoria la palabra -COMPILADOR\0- donde '\0' es el carácter nulo y la marca de fin de cadena de caracteres.

En este caso, para la función strcpy() (string copy) copiará la constante de la cadena que se encuentra como parámetro de la derecha a la variable colocada como parámetro a la izquierda hasta encontrar el carácter nulo.

La función printf("%s\n", x) escribe en pantalla la cadena de caracteres hasta encontrar el valor nulo. Si por alguna razón la cadena de caracteres no tiene como límite de cadena al valor nulo, se imprimirá basura hasta encontrar un valor nulo. Otro detalle importante es ¿cómo sabe el compilador desde dónde escribir? En este caso, el nombre de la variable

–x- es un apuntador a la dirección de memoria del primer carácter de la cadena. El operador unario ampersand no puede ser usado en un constructor de arreglo:

&x

Porque –x- en sí misma es ya un apuntador. Sin embargo, sí se puede utilizar con indices.

&x[0]

En consecuencia, x[0] es una variable del tipo carácter y &x[0] retorna la dirección de memoria de esta variable, la dirección del primer byte del arreglo. Observe el programa 1.7:

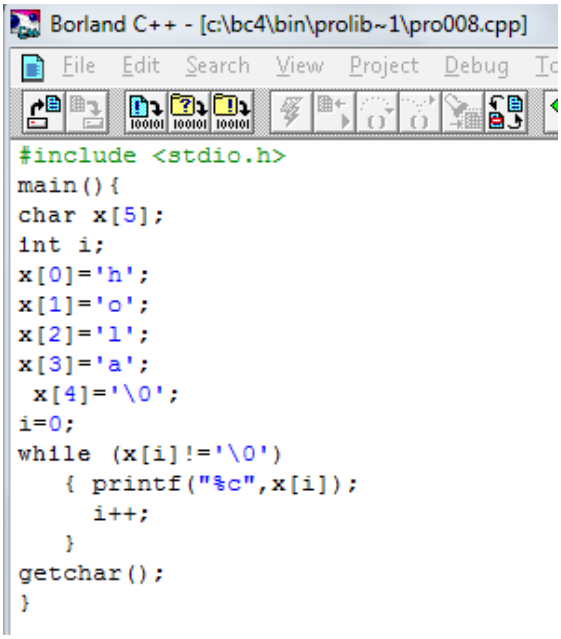
```
#include <stdio.h>
#include <string.h>
main(){
char x[5];
strcpy(x,"hola");
printf("%u  %s  %u  %s\n", &x, x, &x[0], &x[0]);
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\pro007.exe
2293552 hola    2293552 hola
```

Programa 1.7

Por lo tanto, –x- y -&x[0]- son uno y el mismo. El valor que se muestre en su computadora puede variar, sin embargo, se obtendrán dos valores iguales indicando el inicio del arreglo. De esta forma &x[1] mostrará la dirección del segundo byte del arreglo, etcétera.

Otra forma de guardar información en un arreglo se muestra en el programa 1.8:

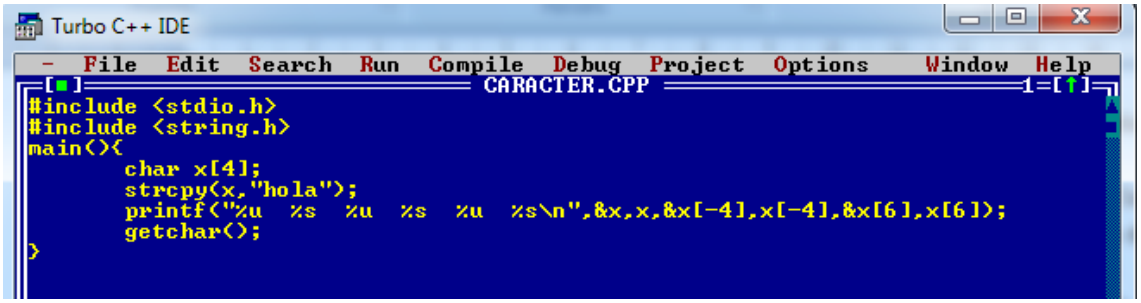


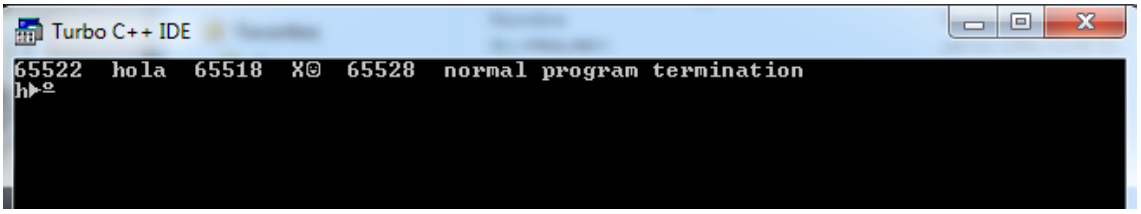
Progama 1.8

El ciclo while muestra en forma muy general la manera como realiza su trabajo la función printf() para %s.

Recuerde que la variable –x- no es igual a “hola”. Solo retorna la dirección de memoria donde se localiza el primer carácter del arreglo.

Una debilidad de C es que no tiene una forma de verificar los límites del arreglo, por lo que se puede desbordar a la izquierda como a la derecha y el compilador no verifica el desbordamiento (otros lenguajes como Pascal, Java, PL/1, etcétera, verifican en el momento de la compilación y además en la ejecución. Si existe un desbordamiento en un arreglo, envía un mensaje de error). Un ejemplo de desbordamiento se muestra en el programa 1.9:





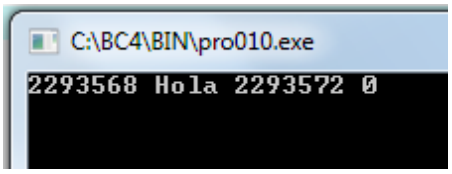
Programa 1.9

Donde:

- &x- y –x- muestran la dirección y contenido del arreglo.
- &x[-4]- muestra la dirección cuatro bytes antes de donde se encuentra la palabra “hola” y posteriormente muestra el contenido que es solo basura.
- &x[6] muestra una dirección 6 bytes después del inicio del arreglo declarado y muestra lo que se tenga en esa posición que puede ser basura, en este caso guardó el comentario “normal programa termination” con salto de línea y basura hasta encontrar el fin de línea “\0”.
- Al ejecutar el programa en una ocasión diferente, se pueden mostrar diferentes elementos.
- Si se ejecuta con otro compilador, puede en un momento dado producir error.

La falta de comprobación de límites simplemente significa que no existe una forma de protección para que no exista una sobrescritura en cualquier arreglo. Por ejemplo, en el programa 1.10 muestra en pantalla:

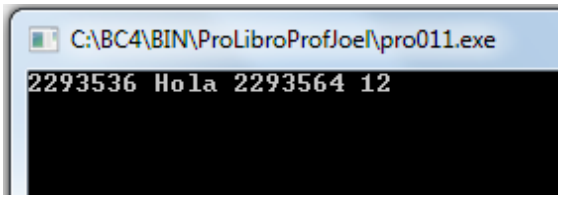
```
#include <stdio.h>
#include <string.h>
main(){
int y=12;
char x[4];
strcpy(x,"Hola");
printf("%u %s %u %d\n",x,x,&y,y);
getchar();
}
```



Programa 1.10

Y el programa 1.11 muestra en pantalla:

```
#include <stdio.h>
#include <string.h>
main(){
int y=12;
char x[5];
strcpy(x,"Hola");
printf("%u %s %u %d\n",x,x,&y,y);
getchar();
}
```



Programa 1.11

En ambos programas se trata de guardar en la variable -x- la palabra “Hola”. Para poder guardar la palabra “Hola” se requieren 5 bytes porque se tiene que guardar también el carácter nulo (“Hola\0”). En consecuencia, en el primer programa se sobrescribió el valor del arreglo ocupando uno de los dos bytes pertenecientes a la variable entera –y-, de esta forma la variable entera muestra en pantalla un cero en lugar de un doce. (Pruebe con los siguientes valores y = 255, y = 256).

Cuando el número de caracteres rebasa al tamaño del arreglo declarado existe la posibilidad de que el arreglo invada memoria que ha sido dada a otras variables por lo que puede sobrescribir causando una secuencia impropia en la ejecución. Un caso extremo puede incluir a porciones de la administración de memoria produciendo un conflicto y siendo necesaria la inicialización de la ejecución del programa o hasta del mismo sistema operativo. O puede realizar borrado de disco duro por una inadvertida escritura a direcciones de interrupción, produciendo un desastre.

Otro ejemplo se muestra en el programa 1.12 con lo que aparece en pantalla:

```
#include <stdio.h>
#include <string.h>
main(){
        char x[10];
        char y[4];
        strcpy(x,"como estas");
        strcpy(y,"bien");
        printf("%u  %s  %u  %s\n",&x,x,&y,y);
        getchar();
        }
```

```
C:\BC4\BIN\ProLibroProfJoel\pro012.exe
2293552  2293548 bien
```

Programa 1.12

Si hacemos un pequeño cambio se obtiene el program 1.13; observe la impresión en pantalla:

```
#include <stdio.h>
#include <string.h>
main(){
        char x[10];
        char y[5];
        strcpy(x,"como estas");
        strcpy(y,"bien");
        printf("%u  %s  %u  %s\n",&x,x,&y,y);
        getchar();
        }
```

```
C:\TC\BIN\arreglo.exe
2293552  como estas  2293536  bien
```

Programa 1.13

En este programa, la sobre posición no es desastrosa. Adicionalmente, el error fue fácilmente reconocido y fácilmente resuelto incluyendo más espacio en una de las variables. Sin embargo, si se tiene un programa mucho más complejo, una sobrescritura puede producir horas de revisión o debugging.

*, UNA VARIABLE TIPO APUNTADOR

2.1 VARIABLES TIPO APUNTADOR Y OPERADORES BÁSICOS

Los apuntadores analizados hasta este momento tienen un valor fijo o constante; es decir, están unidos a una variable que ha sido declarada en algún lugar del programa. En este capítulo se describen variables que han sido declaradas como apuntadores. Estas variables no guardan objetos, al menos como se cree en relación con otras variables. Las variables tipo apuntador guardan la dirección de una locación de memoria. En comparación con los apuntadores analizados previamente, una variable puede apuntar a cualquier área de la memoria.

El programa 2.1 permitirá iniciar la explicación de la declaración de variables tipo apuntador en el lenguaje de programación C. Al ejecutar el programa, los resultados varían dependiendo del compilador y de la computadora, por lo tanto, lo que se muestra en pantalla es lo siguiente.

c:\bc4\bin\prolib~1\joel2001.cpp

```
#include <stdio.h>
main(){
int i, *p;
i=245;
p=&i;
printf("%d %d %u %u %u\n", i, *p, &i, p, &p);
getchar();
}
```

C:\BC4\BIN\PROLIB~1\JOEL2001.EXE

```
245 245 8220 9407 8220
```

Programa 2.1

En la línea de declaración se tienen dos variables, una de ellas es una variable automática del tipo entero (int i) y la otra es una variable apuntador de tipo entero (int *p). El operador unario asterisco (*) es la clave en la declaración.

- *p Es una variable del tipo entero.
- p Apunta a una localidad de memoria del tipo entero.

Para que *p muestre un valor de tipo entero, primero se le debe de indicar la localidad de memoria a la que se está haciendo referencia, por lo que al realizar la igualdad -p=&i-, la variable –p- toma el valor de la dirección de memoria de la variable –i-. Ahora bien, -*p- mostrará el contenido de la localidad de memoria de la variable –i- y -&p- mostrará la dirección de la localidad de memoria de la variable –p-. Observe que un entero necesita de dos bytes para guardar cualquier valor entero dentro de su rango y la variable del tipo apuntador entero utilizará los bytes necesarios para contener una dirección de memoria (ver figura 2.1).

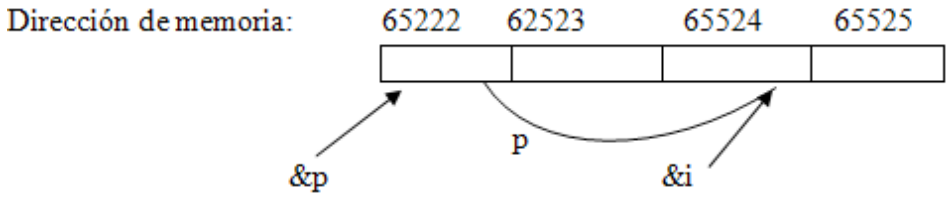


Figura 2.1

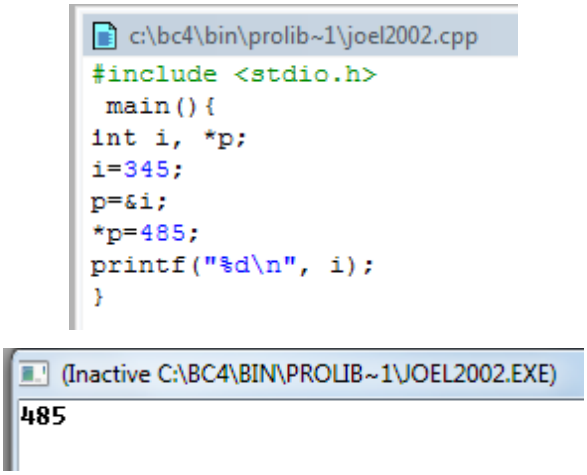
Por lo tanto:

$$\&i \equiv p \quad y \quad *p \equiv i.$$

(Se cumple *p ≡ i. si y solo si primero se realizó la asignación p=&i).

Hasta este momento hemos visto cómo una variable del tipo apuntador puede ser utilizada para retornar un valor de una localidad de memoria a la cual está siendo direccionada.

En el programa 2.2 se muestra cómo la variable de tipo apuntador se utiliza para modificar el valor de la localidad de memoria a la que hace referencia:

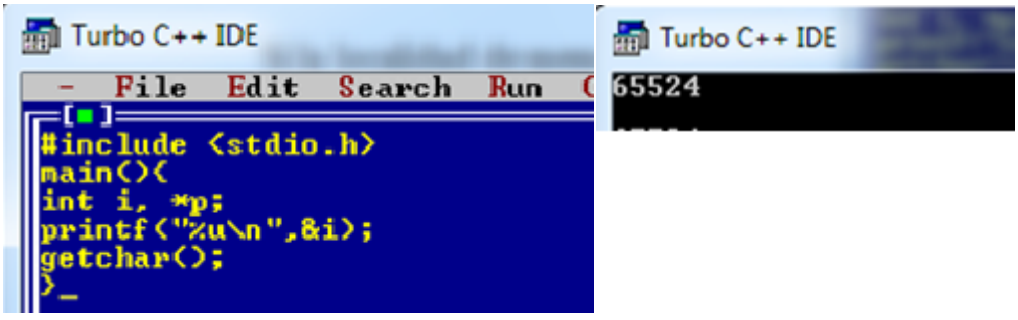


Programa 2.2

Observe que el valor asignado a la variable –i- es 345, posteriormente se le asigna al apuntador –p- la dirección de la localidad de memoria de la variable –i-. En la siguiente línea, la conexión del operador unario -*- al apuntador –p- da como resultado al acceso de la localidad donde hace referencia –p-, por lo que el valor 345 es sobrescrito colocándose el valor 485.

Si la localidad de memoria se conoce, la asignación se puede realizar en forma directa.

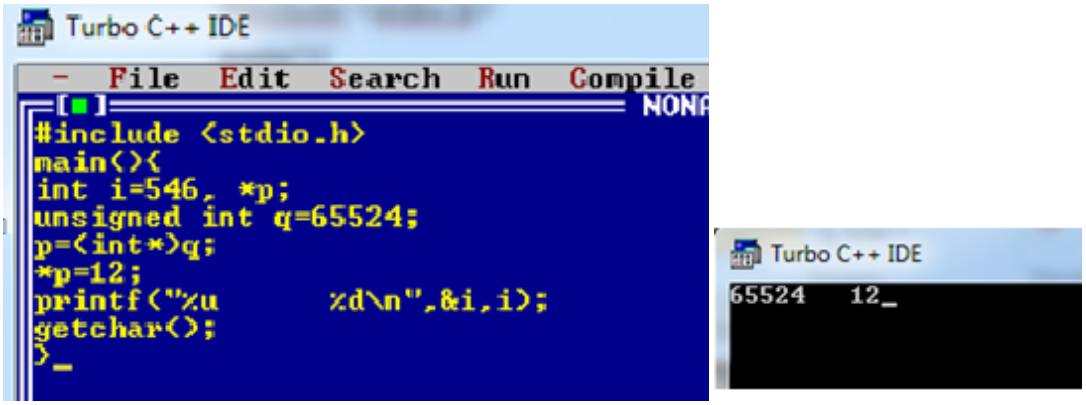
Ejecutemos el programa 2.3:



Programa 2.3

Observe que la variable –i- se encuentra en la localidad 65524.

Ahora, en la misma computadora ejecute el programa 2.4:



Programa 2.4

Esto es porque al apuntador se le asigna la dirección de memoria de la localidad de la variable –i-. Al momento de la asignación se debe de realizar un casting (o moldeado) de una constante entera sin signo a un apuntador tipo entero ya que de otra forma se produce un error de tipo mismatch (los tipos de variable no coinciden). (Este programa en particular fue ejecutado con TC.)

Por lo que:

unsigned q=65524;

p=(int *)q ≡ p=&i //Para una ejecución específica

(El utilizar una variable del tipo unsigned es para poder tener el rango de 0-65536)

Como se recordará, una variable tipo apuntador no se inicializa, por lo que en su localidad de memoria puede apuntar a cualquier dirección de la memoria (basura), uno debe tener la certeza de que el área de memoria es segura. Es decir, que no debe ser utilizada para otros propósitos del cual el programador no pueda tener conocimiento. El programa 2.5 muestra un peligroso uso de un apuntador:

Untitled1.cpp

```
#include <stdio.h>
main(){
int *x;
*x=2541;
printf("%d\n", *x);
getchar();
}
```

Programa 2.5

En este caso, se declara un apuntador del tipo entero, pero no se inicializa con una dirección segura específica. Por lo que puede contener cualquier dirección de memoria. El cuestionamiento es: ¿a qué o a dónde apunta –x-?

Cuando se declara una variable y no se inicializa, esta contiene un valor aleatorio. Esto mismo sucede con los apuntadores, si el apuntador no se inicializa puede apuntar a cualquier área de memoria. Si el programa es tan pequeño como el del ejemplo, puede ser que no suceda nada fuera de lo usual y el programa se ejecute como lo esperado. Existe la posibilidad de que la computadora sea reinicializada o que falle sin alguna razón.

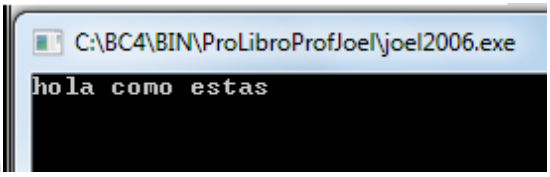
2.2 VARIABLES TIPO APUNTADOR PARA CADENAS DE CARACTERES

Los apuntadores tipo carácter se utilizan con frecuencia para tener acceso a cadenas de caracteres. Cadenas de caracteres y apuntadores a carácter pueden ser tratados en forma indistinta. En el capítulo anterior se comentó que el nombre de un arreglo de caracteres (sin el uso de un subíndice) se considera como apuntador al primer elemento del arreglo de caracteres.

El programa 2.6 muestra la declaración de un apuntador tipo carácter y su uso potencial:

joel2006.cpp

```
#include  <stdio.h>
#include <string.h>
main(){
char a[16], *p;
strcpy(a,"hola como estas");
p=a;
printf("%s\n", p);
getchar();
}
```



Programa 2.6

Al ejecutar el programa se muestra que la variable –p- apunta al inicio de la secuencia de diez caracteres. Observe que el operador ampersand no se utilizó para el acceso a la dirección de memoria del primer elemento ya que la variable –a-, cuando se utiliza sin subíndice, es a la vez el nombre del arreglo y el apuntador al primer elemento del arreglo.

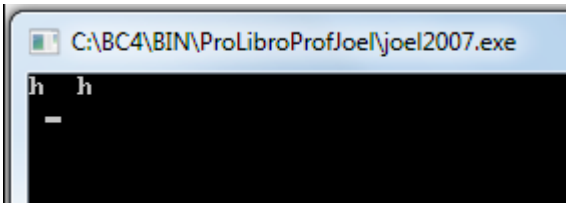
En consecuencia:

$$p=a \equiv p=\&a[0]$$

La variable –p- apunta al primer elemento de la variable, entonces -*p- muestra lo que contiene en el primer elemento del arreglo. Observe el programa 2.7:

joel2007.cpp

```
#include  <stdio.h>
#include <string.h>
main(){
char a[10], *p;
strcpy(a,"hola");
p=a;
printf("%c  %c\n ", * p, a[0]);
getchar();
}
```



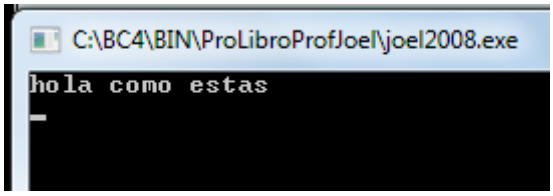
Programa 2.7

Se puede decir que el apuntador apunta a un área segura que es compuesta de 10 bytes consecutivos.

Una forma más sencilla de escribir el primer programa y sin la necesidad de declarar arreglos se muestra en el programa 2.8:

joel2008.cpp

```
#include <stdio.h>
main(){
char  *p;
p="hola como estas";
printf("%s\n",p);
getchar();
}
```



Programa 2.8

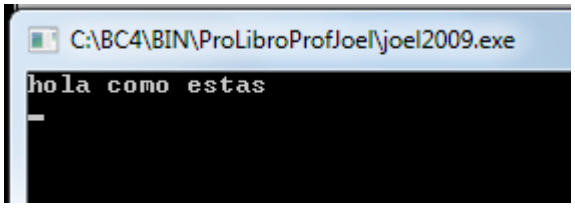
El programa 2.8 realiza lo mismo que el programa 2.6, pero sin la necesidad de la declaración del arreglo de caracteres, además de no necesitar la función strcpy() y la librería string.h.

Pero se asignó la dirección de un objeto a un apuntador y el apuntador no fue inicializado. El punto por analizar es porqué el programa es correcto, recordemos que la prioridad de una asignación es de derecha a izquierda, por lo que la primer acción a realizar es identificar el elemento que está a la derecha de la igualdad, siendo esta una cadena de caracteres constante a la que se le asignará un espacio de memoria o dirección del inicio de este espacio, al cual a su vez se le asignará la variable –p-. Por lo que la variable –p- apunta a un área segura de memoria, el área donde la constante fue almacenada.

El programa descrito es casi lo mismo al programa 2.9:

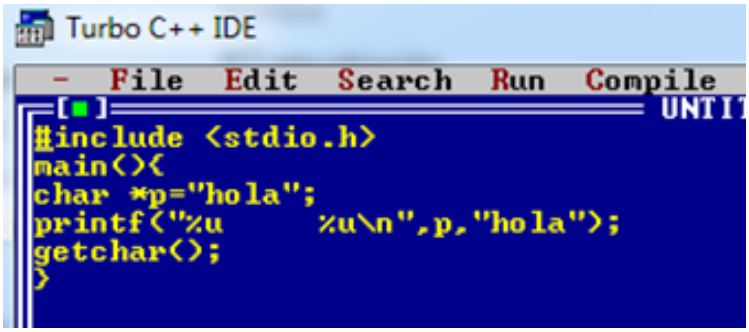
joel2009.cpp

```
#include <stdio.h>
main(){
printf("%s\n","hola como estas");
getchar();
}
```



Programa 2.9

La forma de ver el espacio de las constantes depende del compilador, por ejemplo, se muestra el programa 2.10A en TC:



Programa 2.10A

Se podría pensar que al ejecutarse el programa se desplegaría en el monitor la misma dirección de memoria, pero lo que realmente sucede es que existen dos constantes con el mismo contenido. El compilador destinará dos direcciones diferentes de memoria aunque se tenga el mismo contenido, la variable –p- apuntará al inicio de una constante y al momento de la invocación de la función printf() se asignará el espacio de memoria a la segunda constante.

Observe la ejecución del mismo programa en DEV-C (programa 2.10B) y opine:

```
#include <stdio.h>
main(){
char *p="hola";
printf("%u      %u\n",p,"hola");
getchar();
}
```

```
C:\TC\BIN\Untitled1.exe
4206592      4206592
```

Programa 2.10B

Ahora se observa otra forma del manejo de apuntadores utilizando los ejemplos de los programas 2.11 y 2.12.

joel2010.cpp

```
      #include <stdio.h>
#include <string.h>
main(){
char a[11];
int x;
strcpy(a,"hola amigo");
for (x=0;a[x]!='\0';x++)
        printf("%c",a[x]);
putchar('\n');
getchar();
}
```

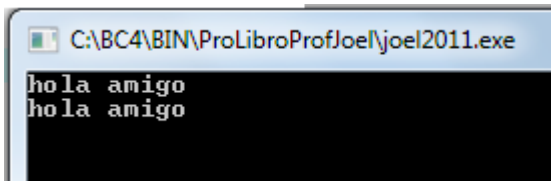
```
C:\BC4\BIN\ProLibroProfJoel\joel2010.exe
hola amigo
```

Programa 2.11

En este ejemplo se observa cómo se puede imprimir cada uno de los caracteres utilizando un operador de repetición como es el “for”.

joel2011.cpp

```
#include <stdio.h>
main(){
char *a;
int x;
a="hola amigo";
for (x=0;*(a+x)!='\0';x++)
        printf("%c",*(a+x));
putchar('\n');
for (x=0;a[x]!='\0';x++)
        printf("%c",a[x]);
putchar('\n');
getchar();
}
```



Programa 2.12

Sabiendo que la variable del tipo char solo ocupa un byte, se puede manejar el comando “for” con un incremento de uno en uno. Ahora bien, se sabe que la variable –a- apunta al primer elemento de la cadena de caracteres, ¿pero si se le suma uno al apuntador? Entonces se estaría apuntando al siguiente elemento del arreglo, por lo tanto:

*a ≡ a[0]
*(a+1) ≡ a[1]
*(a+2) ≡ a[2]
*(a+3) ≡ a[3]

Es más eficiente la declaración de apuntadores a carácter que la declaración de arreglo de caracteres. En el arreglo de caracteres se tiene que definir el tamaño de memoria a utilizar, por ejemplo char b[20], esto indica que se tiene que asignar el espacio de 20 bytes aun si no los usa todos o que exista un desbordamiento produciendo los problemas previamente analizados.

Si se usan apuntadores de carácter, no se tiene que reservar una determinada cantidad fija de elementos. Esto tiene una gran ventaja, ya que si el apuntador va a hacer referencia a una constante del tipo cadena de caracteres, la cadena puede ser de cualquier tamaño y no se desperdicia espacio o no se corre el riesgo de desbordamiento.

Otra forma para mostrar una cadena de caracteres se observa en el programa 2.13:

joel2012.cpp

```
#include <stdio.h>
main(){
char *a;
a="hola amigo";
while (*a!='\0')
        printf("%c",*a++);
putchar('\n');
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2012.exe
hola amigo
```

Programa 2.13

Al ejecutarse la función printf() despliega en el monitor el contenido de la localidad donde está apuntando la variable –a-, al incrementarse el apuntador en uno, se permite desplegar en el monitor el siguiente carácter hasta que se llega al carácter NULL.

Otra forma se observa en el programa 2.14:

joel2013.cpp

```
#include <stdio.h>
main(){
char *a;
a="hola amigo";
while (*a)
        printf("%c",*a++);
putchar('\n');
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2013.exe
hola amigo
```

Programa 2.14

El operador de control while (iteración o mientras) se ejecuta mientras la condición sea verdadera (es decir, mientras que el resultado de la condición sea diferente a cero); el operador romperá el ciclo cuando la condición sea falsa (o sea, hasta que la condición tenga el valor de cero), al recorrer la cadena al llegar al valor NULL, se tiene el valor de cero como condición produciendo que se rompa el ciclo.

En los dos últimos ejemplos se ha modificado la dirección de memoria del apuntador utilizando el operador unario "++", por lo que el apuntador, al final, se queda con la dirección de localidad de memoria donde se localiza el valor NULL. Esto se puede probar con el programa 2.15:

joel2014.cpp

```
#include <stdio.h>
main(){
char *a;
a="hola";
printf("%u\n",a);
while (*a)
        printf("%c",*a++);
putchar('\n');
printf("%u\n",a);
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2014.exe
4206592
hola
4206596
```

Programa 2.15

Para que el apuntador mantenga la dirección de memoria del inicio de la cadena de caracteres se puede utilizar una función como se muestra en el programa 2.16:

joel2015.cpp

```
#include <stdio.h>
void imprime(char *);
main(){
char *a;
a="hola amigo";
printf("%u\n",a);
imprime(a);
printf("%u\n",a);
getchar();
}
void imprime(char *s){
while (*s)
        printf("%c",*s++);
putchar('\n');
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2015.exe

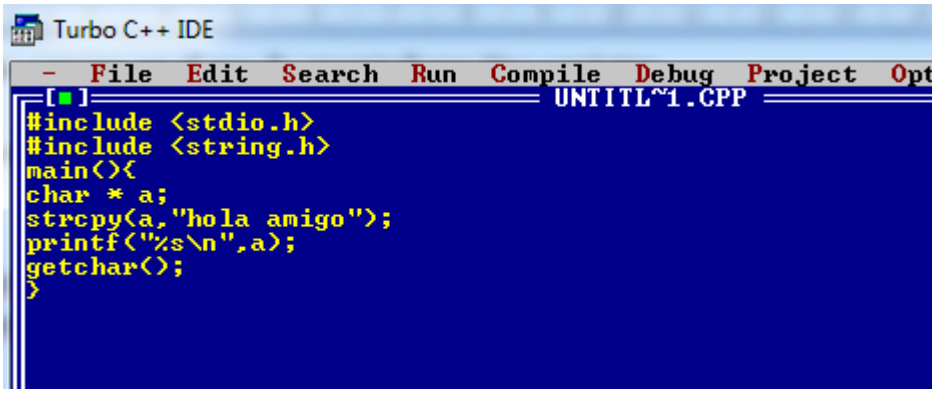
```
4206592
hola amigo
4206592
```

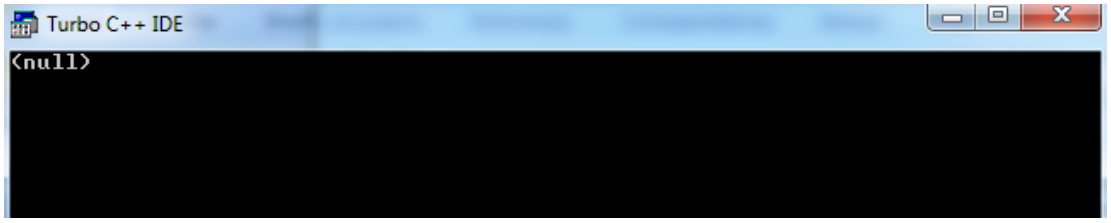
Programa 2.16

El lenguaje de programación C solo tiene pase de parámetros por valor, al invocarse la función -imprime()- se envía como parámetro el valor de la dirección del inicio de cadena. Al ejecutarse la función -imprime()- la variable local –s- recibe como valor la dirección de inicio de la cadena. Al finalizar el operador de repetición while(), la variable –s- apuntará a la localidad cuyo valor es NULL y la función imprime() retornará el control a la función donde fue invocada y el valor de la variable –a- no ha sido alterado. Por lo tanto, al ejecutarse el programa lo que se observa en la variable –a- es la misma dirección de memoria antes de invocar la función –imprime()- y después de la invocación a tal función.

2.3 ALGUNOS ERRORES EN EL USO DE APUNTADORES TIPO CARÁCTER

En el programa 2.17 se tiene un grave error:





Programa 2.17

En Turbo C, el programa se podrá compilar y ejecutar e imprimirá <null>, en DEV-C se podrá compilar, pero no se podrá ejecutar.

En este ejemplo, cuando se declara la variable –a- se inicializa con un valor aleatorio, al invocar la función strcpy() se asignará la constante “hola amigo” a un lugar de memoria que puede ser inseguro produciendo los conflictos ya mencionados cuando un apuntador no se ha inicializado correctamente. El compilador no encontrará error lexicográfico ni sintáctico, pero en la ejecución se pueden generar tres posibles errores. La primera es que el compilador muestre un valor <null>, la segunda es que el compilador aborte el programa y muestre un letrero indicando que existió una violación al tratar de escribir en la dirección 0x0. Y la tercer opción es que ejecute sin problema; en esta opción si se tiene un buen antivirus detectará que se trata de sobre escribir en zona insegura y bloqueará el proceso.

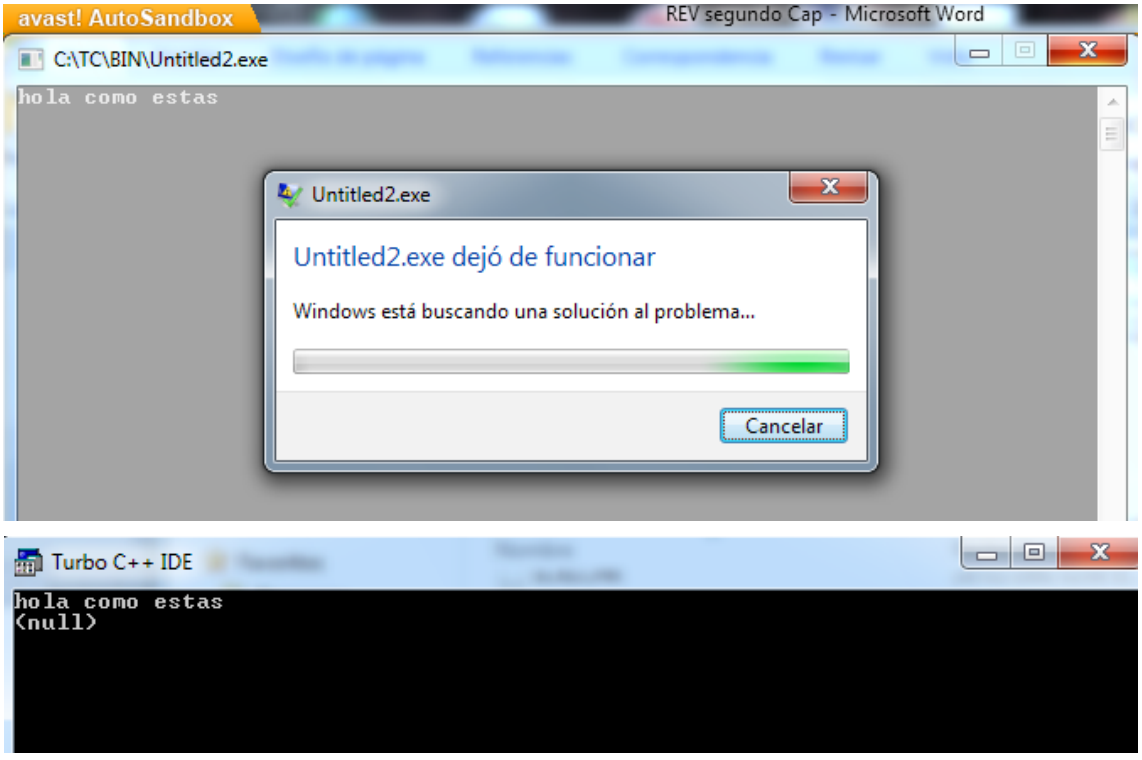
En el programa 2.17 se muestra otro posible error. En este programa la función -gets()- leerá caracteres del teclado hasta que se oprima la tecla “enter” o “return”. En ese momento, lo introducido por el teclado será asignado a la variable que fue enviada como parámetro de la función. Se podrá observar que la variable –a- no está inicializada y tiene

como dirección a memoria un valor aleatorio, por lo que la cadena de caracteres se puede guardar en un momento dado en un área de la memoria insegura. El resultado de la ejecución dependerá del compilador utilizado.

Es importante recordar que una vez que a una variable de carácter se le asigne una dirección de memoria, esta sea una dirección segura.

Al ejecutar el programa 2.18 en DEV-C, se obtiene un desplegado de error después de introducir un texto y en Turbo C se desplegará el mensaje <null>.

```
#include <stdio.h>
main(){
char *a;
gets(a);
printf("%s\n",a);
}
```



Programa 2.18

Observe el programa 2.19:

joel2018.cpp

```
#include <stdio.h>
main(){
        char *a,b[20];
        a=b;
        gets(a);
        printf("%s\n",a);
        getchar();
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2018.exe

```
hola crayola
hola crayola
```

Programa 2.19

En este programa, la variable –b- acepta hasta 19 caracteres en zona segura. Por lo que el apuntador –a- recibe la dirección de inicio del arreglo –b-. De esta forma se puede leer una cadena de caracteres del teclado en zona segura de estos 19 bytes.

Observe el programa 2.20 e indique que es lo que se despliega en el monitor al momento de su ejecución:

joel2019.cpp

```
#include <stdio.h>
#include <string.h>
main(){
        char *a,b[20];
        strcpy(b,"hola mundo");
        a=b;
        strcpy(a,"adios mundo");
        printf("%s\n",a);
        getchar();
}
```

Programa 2.20

Al cuestionarnos lo que muestra en pantalla la ejecución del programa, si se pensó que se muestra en monitor “hola mundo”, es un error. Recuerde que al momento de igualar a=b, la variable –a- tiene como valor el inicio del arreglo donde apunta la variable –b-, por lo que se sobrescribirá la palabra “adiós mundo”, siendo esta frase la que se muestra.

2.4 ARREGLOS DE APUNTADORES

Un arreglo es una colección de datos homogéneos, y un apuntador es una variable especial que guarda una dirección de memoria. Ahora bien, es posible y práctico contar con un arreglo de apuntadores. Observe el programa 2.21:

joel2020.cpp

```
#include <stdio.h>
#include <string.h>
main(){
        char b[3][20];
        strcpy(b[0],"hola mundo");
        strcpy(b[1],"estoy contento");
        strcpy(b[2],"con la vida");
        printf("%s %u\n",b,b);
        for (int i=0; i<3;i++)
            printf("%s %u\n",b[i],b[i]);
        getchar();
        }
```

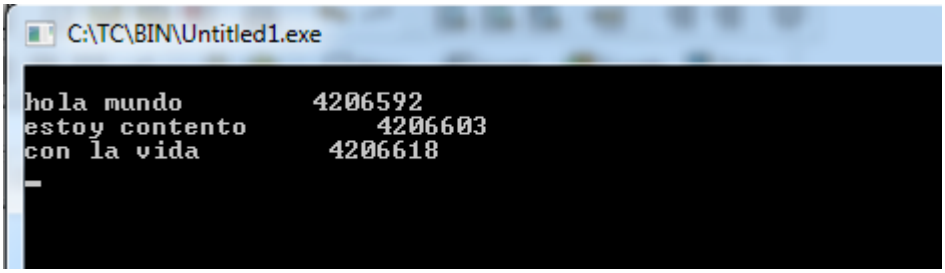
```
C:\BC4\BIN\ProLibroProfJoel\joel2020.exe
hola mundo 2293504
hola mundo 2293504
estoy contento 2293524
con la vida 2293544
```

Programa 2.21

En este programa, el arreglo bidimensional –b- acepta 3 cadenas con un máximo de 19 caracteres. Observe que el compilador reservará espacio para 20 caracteres (incluyendo el fin de cadena) para cada una de las cadenas. La variable –b- es un apuntador que guarda el inicio del primer elemento, en este caso b=b[0], b[0] guarda el inicio de la primer cadena de caracteres, b[1] guarda el inicio de la segunda cadena de caracteres, y b[2] guarda el inicio de la tercera cadena.

Ahora, observe el programa 2.22:

```
#include <stdio.h>
main(){
        char *b[3];
        b[0]="hola mundo"; //La cadena es de 11 bytes incluyendo el byte NULL
        b[1]="estoy contento";//La cadena es de 15 bytes incluyendo el byte NULL
        b[2]="con la vida";
        for (int i=0; i<3;i++)
            printf("%s        %u\n",b[i],b[i]);
        getchar();
        }
```



Programa 2.22

En este caso se tiene un arreglo de tres apuntadores a carácter. En la ejecución solo se guardará espacio para cada una de las cadenas en forma individual, y al ejecutarse el programa se obtendrá el siguiente desplegado:

hola mundo	4206592	
estoy contento	4206603	//Una diferencia de 11 bytes
con la vida	4206618	//Una diferencia de 15 bytes

Como se observó en el ejemplo anterior, existe una diferencia (sutil) entre apuntadores y arreglos. Un ejemplo claro es el manejo de cadenas es:

char *p[10];
char mc[10][20];

Aquí es totalmente válido direccionarse en el lenguaje C a p[5][6] y mc[5][6]. Sin embargo:

- En el ejemplo anterior, -mc- es un arreglo verdadero de 200 elementos de dos dimensiones del tipo char.
- El acceso de cada uno de los elementos del arreglo –mc- en memoria se hace bajo la siguiente fórmula: 20*renglón + columna + dirección base.
- En cambio, la variable –p- solo tiene 10 apuntadores a elementos.
- Una desventaja de la variable –mc- es que solo tiene 200 caracteres disponibles en zona segura.
- Una ventaja de la variable –p- es que cada apuntador puede apuntar a cadenas de caracteres de diferente longitud.
- Reiterando, un apuntador es una variable. Si la variable –ap- es un apuntador tipo carácter y la variable –c- es un arreglo de tipo carácter, se permite realizar las siguientes instrucciones: “ap=c; ap++;”. Sin embargo, no es posible realizar las siguientes instrucciones: “c=ap; c++;”, es ilegal. Recuerde que “un apuntador es una variable, un arreglo NO ES una variable”.

Por ejemplo, la declaración:

char semanam[][15]={“Lunes”, ”Martes”, “Miércoles”, “Jueves”, “Viernes”, “Sábado”, “Domingo”`}

La variable semanam es una matriz con 15 columnas, lo cual se puede observar en la figura 2.2.

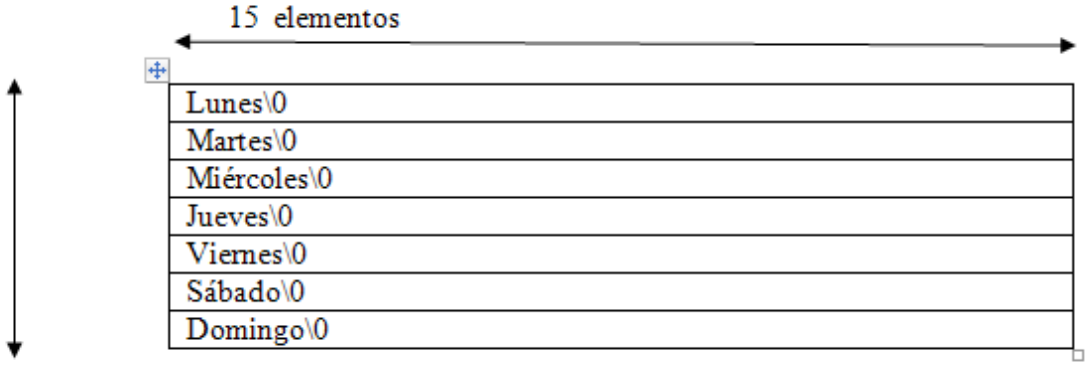


Figura 2.2

El arreglo de apuntadores semanap:

char *semanap[]={“Lunes”, ”Martes”, “Miércoles”, “Jueves”, “Viernes”, “Sábado”, “Domingo”`}

Se representa gráficamente en la figura 2.3:

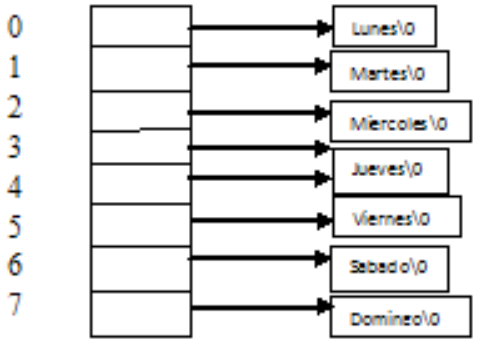


Figura 2.3

Se observa con claridad que en la variable –semanap- tiene un uso más eficiente del espacio.

2.5 MANEJO DE ESTRUCTURAS (REGISTROS)

Una estructura es una colección de variables de diferentes tipos de datos. Por supuesto, los arreglos permiten un medio de almacenar variables dentro de una unidad. En el caso de los arreglos, los valores dentro del arreglo deben de ser homogéneos y en el caso de las estructuras, los valores son heterogéneos.

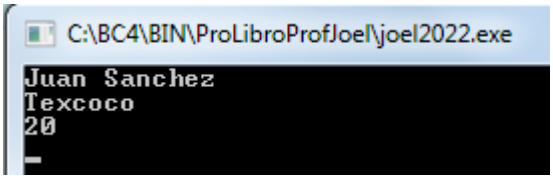
Existen diferentes formas de crear una estructura, la más común es:

```
struct alumno{
        char nombre[30];
        char dirección [50];
        char sexo;
        int edad;
};
```

El nombre de la estructura es en sí un tipo de dato. Por lo tanto, se pueden declarar variables tipo nombre de la estructura. Por ejemplo, el programa 2.23 muestra en pantalla los elementos guardados en la estructura –alumno-:

joel2022.cpp

```
#include <stdio.h>
#include <string.h>
struct alumno{
        char nombre[30];
        char direccion [50];
        int edad;
};

main(){
        alumno a;
        strcpy(a.nombre,"Juan Sanchez");
        strcpy(a.direccion,"Texcoco");
        a.edad=20;
        printf("%s\n%s\n%d\n",a.nombre,a.direccion,a.edad);
        getchar();
}
```



Programa 2.23

Algunos compiladores permiten la declaración de variable de la siguiente forma:

alumno a;

Otros compiladores requieren que el tipo de variable sea mas especifico, por lo que se debe de anteponer la palabra struct:

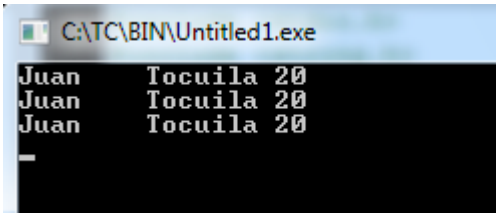
struct alumno a;

La forma de tener acceso a un elemento de la estructura es:

a.nombre.

Los apuntadores a estructuras son muy comunes en la programación en C. Cuando se utiliza un apuntador a estructuras, se requiere un operador especial para poder tener acceso a un elemento de una estructura. El operador es "->" (Un guión medio seguido del signo mayor que). El programa 2.24 muestra el uso de un apuntador tipo estructura:

```
#include <stdio.h>
#include <string.h>
struct alumno{
char a[20];
char b[30];
int edad;
};
main(){
 struct alumno alumnos,*b;
 b=&alumnos;
 strcpy(b->a,"Juan");
 strcpy(b->b,"Tocuila");
 b->edad=20;
 printf("%s\t%s\t%d\n", alumnos.a, alumnos.b, alumnos.edad);
 printf("%s\t%s\t%d\n",b->a,b->b,b->edad);
 printf("%s\t%s\t%d\n",(*b).a,(*b).b,(*b).edad);
 getchar();
}
```



Programa 2.24

En este programa la variable –alumnos- es del tipo alumno y la variable –b- es un apuntador del tipo alumno. El operador especial "." se emplea para tener acceso a la estructura al momento de usar la variable –alumnos- y el operador especial "->" se emplea para tener acceso a cada uno de los elementos de la estructura al momento de utilizar la variable –b-. El último operador es una forma sencilla de acceso a elementos de estructura. Como se observa en el programa, el operador especial "->" tiene la siguiente equivalencia:
b->edad ≡ (*b).edad

El operador "." tiene mayor precedencia con respecto al operador unario "*", por lo que el operador unario requiere colocarse entre paréntesis.

Como se muestra en el programa 2.25, se pueden tener arreglos de estructuras que se pueden manipular con apuntadores.

joel2024.cpp

```
#include <stdio.h>
#include <string.h>
struct alumno{
char a[20];
char b[30];
int edad;
};

main(){
struct alumno alumnos[2],*b;
b=alumnos;   //o también se puede colocar b=&alumnos[0];
strcpy(b->a,"Juan");
strcpy(b->b,"Tocuila");
b->edad=20;
strcpy((b+1)->a,"Luis");
strcpy((b+1)->b,"Taxco");
(b+1)->edad=40;
printf("^%s\t%s\t%d\n",alumnos[0].a,alumnos[0].b,alumnos[0].edad);
printf("^%s\t%s\t%d\n",b->a,b->b,b->edad);
printf("^%s\t%s\t%d\n",(*b).a,(*b).b,(*b).edad);

printf("^%s\t%s\t%d\n",alumnos[1].a,alumnos[1].b,alumnos[1].edad);
printf("^%s\t%s\t%d\n",(b+1)->a,(b+1)->b,(b+1)->edad);
printf("^%s\t%s\t%d\n",(*(b+1)).a,(*(b+1)).b,(*(b+1)).edad);
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2024.exe
^Juan   Tocuila 20
^Juan   Tocuila 20
^Juan   Tocuila 20
^Luis   Taxco   40
^Luis   Taxco   40
^Luis   Taxco   40
```

Programa 2.25

En este ejemplo se muestran dos formas diferentes de imprimir el contenido de los registros utilizando los dos operadores especiales para el acceso de los elementos de cada registro.

Observe que al incrementar la variable-b- en uno, el incremento se realizará con respecto al número de bytes al tamaño de la estructura.

2.6 APUNTADORES A APUNTADORES

Los apuntadores, como cualquier otra variable, también tienen dirección de memoria, esto es, al momento de ser declarados los apuntadores tienen su propia área de memoria. En un modelo pequeño de memoria un apuntador requerirá 2 bytes para el almacenamiento de una dirección de memoria y para un modelo grande de memoria se requerirán 4 bytes. Se sabe que toda variable tiene dos valores, el primero es la dirección donde se guardará la información, el segundo es el valor del objeto o dato que será guardado en esa dirección.

En el caso de apuntadores, el primer valor es la dirección donde se guardará la información del apuntador y el segundo valor es el valor del objeto, pero en este caso, el objeto es otra dirección de memoria a la que está haciendo referencia tal apuntador.

Es obvio que los apuntadores tienen una dirección fija donde guarden su objeto. Observemos el programa 2.26:

joel2025.cpp

```
#include <stdio.h>
main(){
        int *x;
        char *y;
        long *z;
        double *w;
        printf("%u\t%u\t%u\t%u\n",&x,&y,&z,&w);
        getchar();
}
```

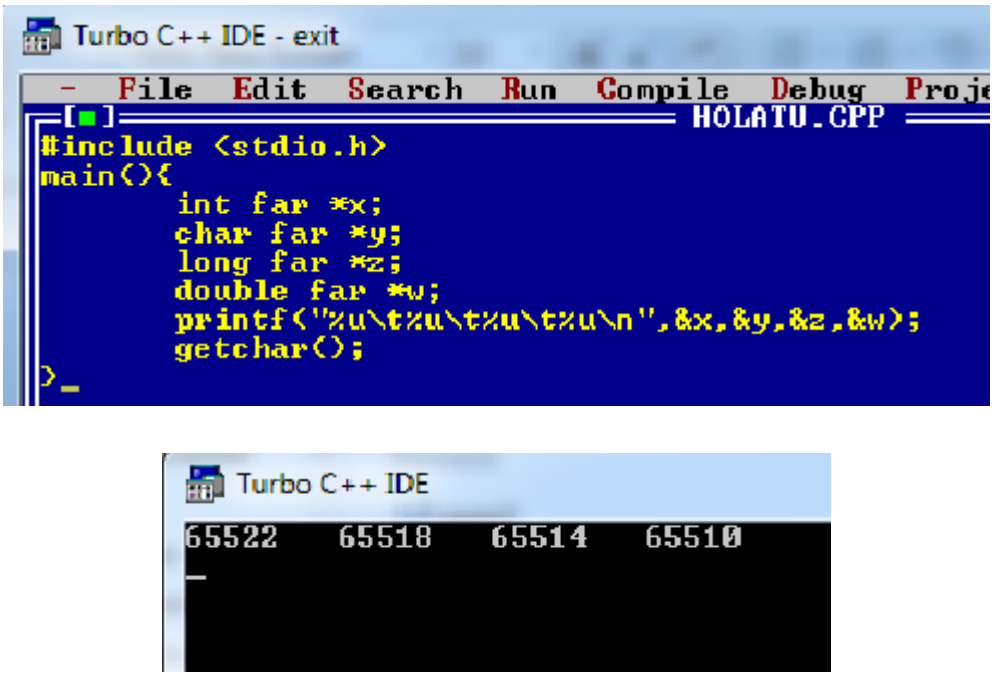
```
C:\BC4\BIN\ProLibroProfJoel\joel2025.exe
2293572 2293568 2293564 2293560
```

Programa 2.26

Lo que se observa en pantalla es la dirección de memoria donde cada apuntador guardará la dirección de memoria a la que se hace referencia.

Como se observará, para un modelo pequeño se requieren solo dos bytes para poder hacer referencia a una dirección de memoria.

Ahora, observe el programa 2.27 ejecutado en TC:



Programa 2.27

Ya que se está manejando un modelo de memoria grande, se requieren cuatro bytes para manejar la información a guardar.

Se puede tener una variable que apunte a la dirección de memoria de la variable tipo apuntador. El programa 2.28 muestra el uso de un apuntador a apuntador o doble apuntador:

joel2027.cpp

```
#include <stdio.h>
main(){
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%d\t%d\t%d\n",x,*p,**ptr);
getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2027.exe
1000    1000    1000
```

Programa 2.28

En este programa -**ptr- es declarado un doble apuntador, o apuntador a apuntador, de tipo entero, por lo que da acceso al objeto que guarda la variable –p- y de esta forma accederá al objeto guardado por la variable –x-.

Observe el programa 2.29:

joel2028.cpp

```
#include <stdio.h>
main(){
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%u\t%u\t%u\n",&x,p,*ptr);
getchar();
}
```

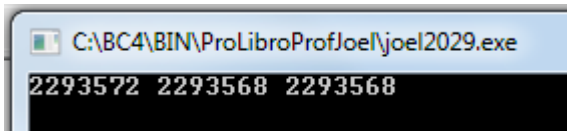
```
C:\BC4\BIN\ProLibroProfJoel\joel2028.exe
2293572 2293572 2293572
```

Programa 2.29

La variable –p- guarda la dirección de la variable –x- donde hace referencia, y la variable -*ptr- guarda la dirección donde hace referencia –p- (siendo ptr un doble apuntador), lo que se muestra en pantalla es la misma dirección de memoria para las tres variables. Observe programa 2.30:

joel2029.cpp

```
#include <stdio.h>
main(){
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%u\t%u\t%u\n",&x,&p,ptr);
getchar();
}
```



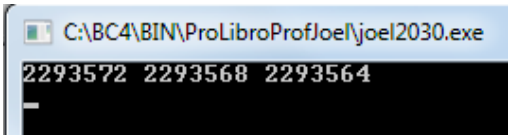
Programa 2.30

Aquí la variable -&p- muestra la dirección de memoria donde la variable –p- guardará el valor del objeto y ptr hará referencia al valor del objeto guardado en su área de memoria.

El programa 2.31 muestra el apuntador al doble apuntador:

joel2030.cpp

```
#include <stdio.h>
main(){
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%u\t%u\t%u\n",&x,&p,&ptr);
getchar();
}
```



Programa 2.31

-&ptr- es la dirección de memoria donde –ptr- guarda el valor del objeto.

El programa 2.32 muestra el uso de apuntadores a apuntadores tipo carácter:

joel2031.cpp

```
#include <stdio.h>
#include <string.h>
main(){
        char a[5], *b, **c;
        strcpy(a,"Hola");
        b=a;
        c=&b;
        printf("%s\t%c\t%c\n",a,*b,**c);
        getchar();
        }
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2031.exe
Hola    H       H
```

Programa 2.32

El operador unario "*" se utiliza para diferenciar a los objetos buscados. Un doble apuntador se puede pensar como un apuntador anidado. Observe el programa 2.33:

```
#include <stdio.h>
#include <string.h>
main(){
        char a[5], *b, **c;
        strcpy(a,"Hola");
        b=a;
        c=&b;
        printf("%u\t%u\t%u\n",a,b,*c);
        getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2032.exe
2293552 2293552 2293552
```

Programa 2.33

En este caso, -*c- es la dirección de memoria del objeto anidado. La expresión -**c- es el objeto al final de la pila, por lo que se direccionará al primer carácter de la cadena. Observe que si solicita la impresión de una cadena para la variable –c- (%s) en vez de la impresión de un carácter (%c), se producirá un error de ejecución.

En C se puede crear un mayor número de anidamientos. El programa 2.34 muestra un ejemplo de anidamiento:

joel2033.cpp

```
#include <stdio.h>
main(){
      int x, *p, **p2, ***p3, ****p4;
      x=100;
      p=&x;
      p2=&p;
      p3=&p2;
      p4=&p3;
      printf("%d\n",****p4);
      getchar();
   }
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2033.exe
100
```

Programa 2.34

Este programa muestra la capacidad de anidamiento de las operaciones con apuntadores, por supuesto, es difícil encontrar en la práctica un anidamiento tan profundo.

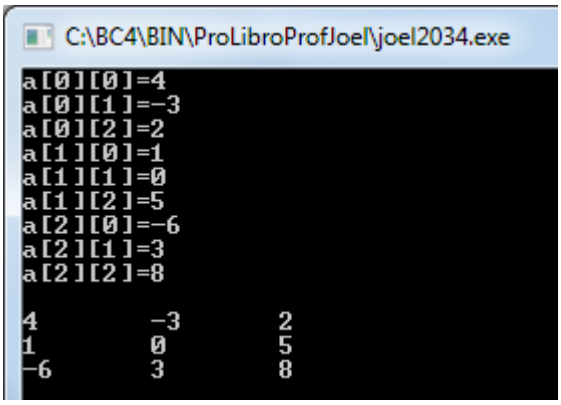
2.7 MANEJO DE MATRICES CON APUNTADORES

Una matriz es un arreglo bidimensional de números. Las matrices se usan generalmente para describir sistemas de ecuaciones lineales, sistemas de ecuaciones diferenciales o representar una aplicación lineal.

Siendo una matriz un arreglo bidimensional, se puede manipular por medio de corchetes o por anidamiento de apuntadores como se muestra en el programa 2.35:

joel2034.cpp

```
#include <stdio.h>
main(){
    int a[3][3],i,j;
for (i=0;i<3;i++)
        for (j=0;j<3;j++){
            printf("a[%d][%d]=", i, j);
            scanf("%d",(*(a+i)+j));
        }
    for (i=0;i<3;i++)
      { printf ("\n");
        for (j=0;j<3;j++)
            printf("%d\t",*(*(a+i)+j));

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



Programa 2.35

Analice los programas anteriores sobre apuntadores dobles y trate de comprender el funcionamiento del último programa.

ALLOCATE, ASIGNAR MEMORIA EN FORMA DINÁMICA

3.1 APUNTADORES Y ASIGNACIÓN DE MEMORIA

En la mayoría de los programas explicados en los capítulos anteriores, los apuntadores se asignaron a localidades de memoria correspondientes a una variable en particular. (Algunos lo conocen como peek/poke, peek se refiere a leer el contenido de una celda de memoria de una determinada dirección y poke establece el contenido de una celda de memoría). Los apuntadores se asignaron a localidades de memoria específicamente con el propósito de leer el contenido de una variable o escribir el contenido de una variable en un área de memoria específica. Estas localidades de memoria se conocen en tiempo de compilación (también conocidas como variables estáticas ya que al compilar se conoce la cantidad de memoria a ser utilizada, por ejemplo: char a[10]).

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

stdlib.h (*std-lib: standard library* o biblioteca estándar) es el archivo de cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras. Es compatible con C++ donde se conoce como cstdlib.

alloc.h (*allocate library* o biblioteca para asignación dinámica de memoria) es el archivo de cabecera para C que nos da la capacidad de asignar y liberar memoria de forma dinámica. No todos los compiladores tienen esta biblioteca, por lo que se debe de verificar si el compilador a utilizar contiene tal herramienta.

Ambas bibliotecas tienen varias funciones en común. Las que se utilizan en este capítulo son:

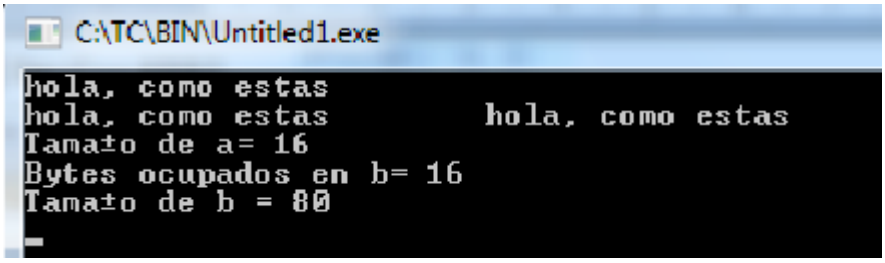
GESTIÓN DE MEMORÍA DINÁMICA	
malloc	void * malloc(size_t size) La función malloc reserva espacio en el heap a un objeto cuyo tamaño es especificado en bytes (size). El heap es un área de memoria que se utiliza para guardar estructuras dinámicas en tiempo de ejecución. Si se puede realizar la asignación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

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

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

El programa 3.1 muestra el uso de la función –malloc()-:

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



Programa 3.1

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

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

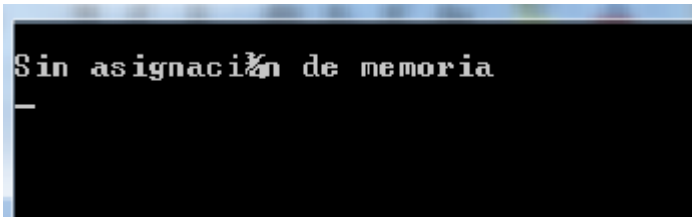
La mayor prioridad la tiene la función -strlen()-, por lo que retorna el número de bytes ocupados en la variable –b-. Observe que no se utilizó la función –sizeof()- ya que esta función retorna el tamaño de bytes reservados en la variable –b- (en este caso sería 80). Una vez determinado el tamaño, la función –malloc()- reserva el espacio en bytes en el heap, posteriormente se realiza el casting (se crea el espacio de memoria a lo indicado dentro del paréntesis, en este caso, se devuelve un apuntador a carácter al inicio de la cadena). El

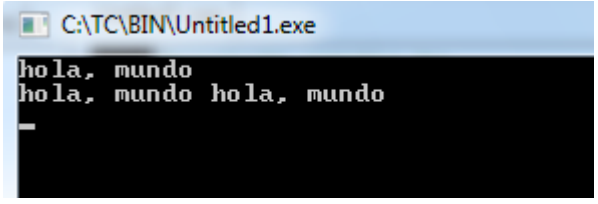
casting es necesario ya que la función –malloc()- retorna un apuntador tipo void (un apuntador sin tipo). Cuando es un apuntador tipo void, se puede tener la dirección de cualquier tipo de variable pero sin tener el permiso de acceso a ella para escribir o para leer. Si se requiere tener el acceso para leer o escribir, se debe moldear al tipo de variable al que se quiere tener acceso (o casting).

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

En el programa 3.2 se muestra un ejemplo de la forma de detectar el retorno de NULL en la función malloc(). Si al ejecutar el programa, el ejecutor del programa oprime la tecla return (enter), se mostrará en monitor “Sin asignación de memoria”. De otra forma se mostrará dos veces lo escrito por el ejecutor del programa:

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

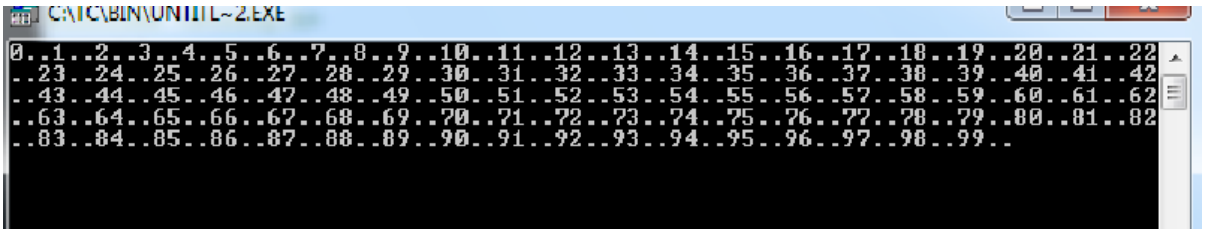




Programa 3.2

Uno puede tener un apuntador que controle a un gran bloque de memoria para algún propósito en específico. El programa 3.3 declara un apuntador entero y utiliza la función malloc para retornar un gran bloque de enteros:

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



Programa 3.3

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

En el programa anterior, el manejo del bloque de bytes se puede realizar sin problema con los subíndices, esto es:
*x ≡ x[0]
*(x+1) ≡ x[1]
Para saber la dirección de memoria de un arreglo se puede utilizar:
x ≡ &x[0]
x + 1 ≡ &x[1]

El programa 3.4 muestra la asignación de memoria en forma dinámica:

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

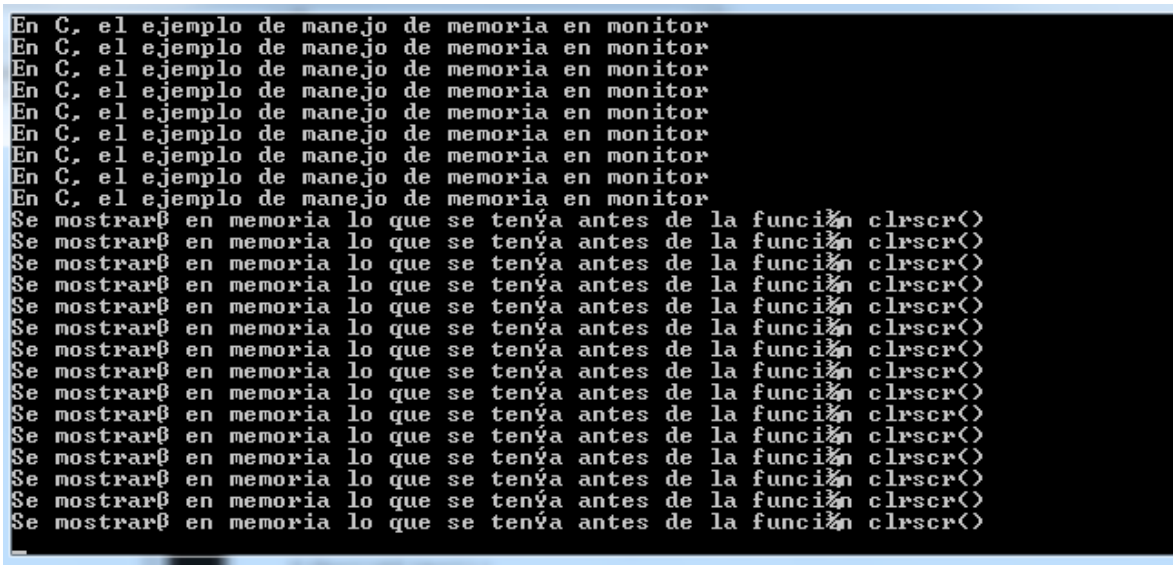
```
La variable -a-  apunta a la localidad de memoria 8523744 8523744
a[0]=100        *(a+0)=100      Loc 8523744     Loc 8523744
a[1]=101        *(a+1)=101      Loc 8523748     Loc 8523748
a[2]=102        *(a+2)=102      Loc 8523752     Loc 8523752
a[3]=103        *(a+3)=103      Loc 8523756     Loc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

Programa 3.4

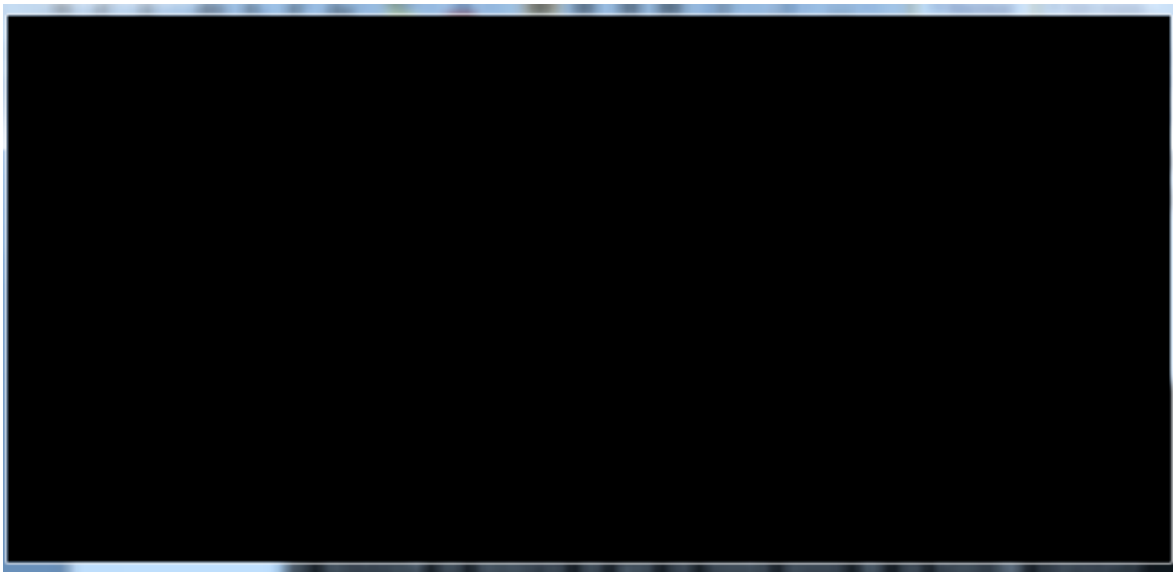
El programa 3.5 muestra un uso práctico de –malloc()-. En este caso, el contenido de la pantalla es guardado en un block de memoria. La pantalla será limpiada y el contenido será desplegado nuevamente. El inicio de la memoria para una pantalla a color es 0xb8000000.

```
//El programa solo se puede ejecutar en Turbo C o Turbo C portable
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>
#include <conio.h>
main(){
char far *a,*b;
int i;
a=(char far*)0xb8000000; //se apunta al inicio de la memoria de la pantalla activa
if ((b=(char *)malloc(4000))==NULL){
        printf("No se pudo asignar memoria\n");
        exit(0);
}
clrscr();
for (i=0; i<10; i++){
        printf("En C, el ejemplo de manejo de memoria en monitor\n");
for (i=0;i<15;i++)
        printf("Se muestra en memoria lo que se tenía antes de la funcion clrscr()\n");
for (i=0;i<4000;i++)  //Se guarda en otra direccion de memoria el contenido en pantalla*(b+i)=*(a+i),
sleep(2);        //Se detiene la pantalla por 3 segundos
clrscr();        //Se limpia la pantalla
sleep(2);        //Se detiene la pantalla por 3 segundos para poder observar la diferencia
for (i=0; i<4000; i++) //Se coloca en pantalla de nuevo el contenido antes de limpiarla*(a+i)=*(b+i);
getchar();
}
```

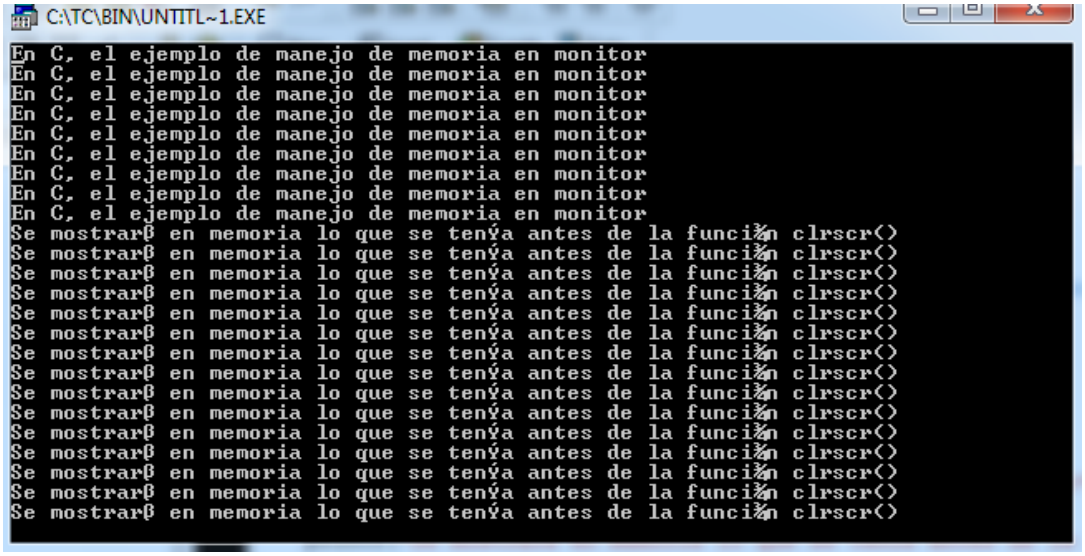
Al ejecutarse el programa se observan las siguientes pantallas:



Se limpia la pantalla:



Y tres segundos después se muestra en pantalla el contenido que se tenía antes de limpiarla:



Programa 3.5

El modificador “far” es utilizado cuando se compila en modelos de memoria diminuta, pequeña o compacta y fuerza al apuntador a colocarse en un punto lejano de la memoria. En este caso, la dirección a apuntar será el buffer de la pantalla.

El programa 3.6 muestra una forma sencilla de guardar una pantalla en disco:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main(){
FILE *fp;
char far *a;
int i;
clrscr();
if((fp=fopen(“pantalla1”,”w”))==NULL){
        printf(“Error: No se pudo abrir el archive\n”);
        exit(0);
}
a=(char far *)0xb8000000; //0xb8000000 es la dirección del inicio de buffer de monitor
for (i=0; i<10; i++)
        printf(“En C, el ejemplo de manejo de memoria en monitor\n”);
for (i=0;i<15;i++)
        printf(“Se muestra en pantalla lo guardado en archivo\n”);
for (i=0;i<4000; i++)
```

```
        fputc(*a++,fp);
fclose(fp);
}
```

Programa 3.6

Para poder manejar un archivo, se requiere abrirlo mediante una función -fopen()- de la biblioteca -stdio.h-. –fopen()- acepta como argumento un nombre externo de archivo y devuelve un nombre interno que deberá emplearse en subsecuentes lecturas o escrituras de archivo.

El nombre interno es un apuntador (llamado apuntador de archivo) a una estructura que mantiene información sobre el archivo. El usuario no necesita conocer los detalles de la estructura ya que dicha estructura es parte de las definiciones contenidas en -stdio.h- y se llama FILE. Solo es necesario declarar el apuntador del archivo. Observe que FILE es el nombre de un tipo, como int, no es un nombre de una estructura; se define internamente mediante un typedef.

La llamada a –fopen()- en un programa es:

fp=fopen(name, mode)

Aquí el primer argumento es el nombre del archivo y el segundo argumento es el modo que indica el uso que se le va a dar al archivo. Los modos permitidos son lectura (“r”), escritura (“w”), o añadir (“a”).

Si se intenta escribir o añadir a un archivo inexistente, se preocupará en crear el archivo. La apertura de un archivo existente para escribir en él provocará la pérdida del contenido anterior. Es un error leer de un archivo que no existe. Existen más situaciones que provocan error (como intentar leer un archivo sin tener permiso necesario). Si existe un error, -fopen()- devuelve el apuntador NULL.

En el programa anterior se intenta abrir un archivo en forma normal, si no es posible se retorna un valor NULL, por lo que se enviará un mensaje de error y terminará la ejecución del programa. Si no es así, por medio del ciclo –for- se crea alguna información en pantalla y posteriormente se emplea un ciclo –for- para guardar en disco lo que se tiene en el buffer de pantalla byte por byte. La función -fputc()- se emplea para escribir en disco cada uno de los bytes a almacenar.

Cuando se haya terminado de realizar la escritura, el archivo se cierra y se tiene en disco la pantalla.

Para recuperar información de una pantalla se puede emplear el programa 3.7:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main(){
FILE *fp;
char far *a;
int i;
clrscr();
a=(char far *)0xb8000000;
if((fp=fopen("pantalla1","w"))==NULL){
        printf("Error: No se pudo abrir el archive\n");
        exit(0);
}
a=(char far *)0xb8000000;
while ((i=fgetc(fp))!=EOF)
        *a++=i;
getchar();
fclose(fp);
}
```

Programa 3.7

En el ciclo while se extraen cada uno de los bytes por medio de la función -fgetc()- hasta encontrar la constante –EOF- (End Of File).

Otra forma de programar el ciclo while es:

```
while ((*a=fgetc(fp))!=EOF)
*a++;
```

Otra forma:

```
while((*a++=fgetc(fp))!=EOF);
```

EOF es una constante en el archivo de cabecera -stdio.h- con valor de -1, se puede emplear en forma directa

```
while((*a++=fgetc(fp))!=-1);
```

o, dependiendo del compilador, con un casting:

```
while ((*a++=fgetc(fp))!=(char)-1);
```

La función -calloc()-, a comparación de la función -malloc()-, limpia la memoria. Esto significa que todo block asignado de memoria será inicializado en cero. El programa 3.8 es similar a un ejemplo anterior en este capítulo y va a permitir hacer la comparación entre las dos funciones.

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int *x,y;
    if((x=(int *)calloc(400,2))==NULL){
                        printf("No asignacion de memoria");
                        exit(0);
    }
    for (y=0; y<400; y++)
       *(x+y)=88;
}
```

Programa 3.8

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

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

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

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

```
calloc(400, sizeof(int));
```

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

malloc(400*sizeof(int))

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

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

free(ptr).

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

La función -realloc(ptr, tamaño)- cambia el tamaño del objeto apuntado por –ptr- al tamaño especificado por –tamaño-. El contenido del objeto no cambiará hasta que se defina el nuevo tamaño del objeto. Si el nuevo tamaño es mayor, el valor de la porción nueva se adjudicará al objeto. Si –ptr- es un puntero nulo, la función –realloc()- se comporta igual que la función –malloc()-. De lo contrario, si –ptr- no es igual a un puntero previamente retornado por la función –malloc()-, -calloc()- o –realloc(), o si el espacio ha sido desadjudicado por una llamada a la función –free()- o- realloc(), el comportamiento no estará definido.

Por ejemplo, el programa 3.9 lee y escribe los valores de un arreglo –V- de reales. El número de valores se conoce durante la ejecución. La memoria se asignará en el momento de ejecución elemento por elemento.

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

Programa 3.9

3.2 MANEJO DE MATRICES EN FORMA DINÁMICA

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

En particular, se posibilitará:

Crear matrices en forma dinámica (en el momento de ejecución)

Destruir matrices en forma dinámica.

Tener acceso mediante índices o apuntadores.

Para poder trabajar la matriz en forma dinámica se requiere una variable del tipo doble apuntador. El primer apuntador hará referencia al inicio de un arreglo de apuntadores y cada apuntador del arreglo de apuntadores tendrá la referencia del inicio de un arreglo de enteros u otro tipo de variable, ver figura 3.1:

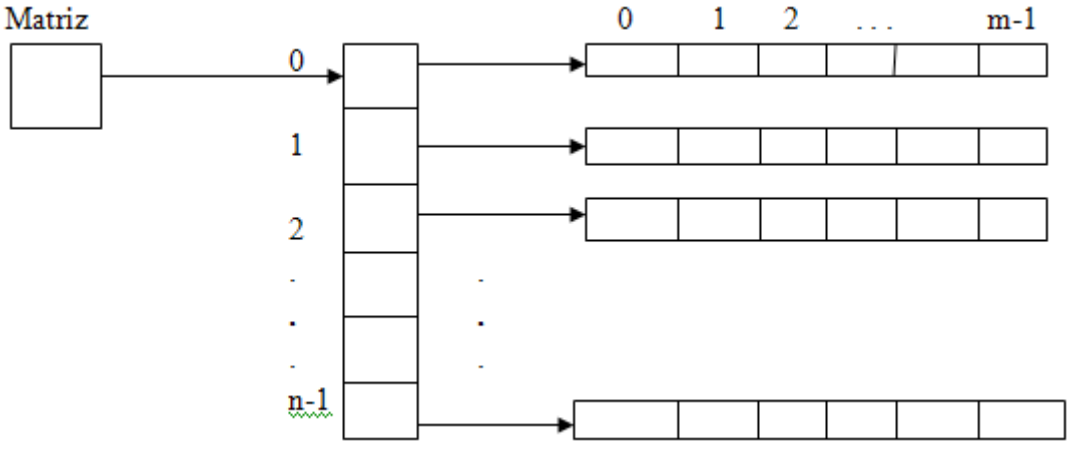


Figura 3.1

El programa 3.10 muestra la forma en que debe ser manipulado cada uno de los apuntadores para la creación dinámica de la matriz bidimensional:

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

      getchar();

      getchar();

}
```

Programa 3.10

La siguiente línea del código anterior crea el arreglo de apuntadores:
pm=(int **)malloc(sizeof(int *)*f);

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

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

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

3.3 MANEJO DE ARREGLOS DE REGISTROS EN FORMA DINÁMICA

El programa 3.11 muestra una forma de crear un arreglo de registros en forma dinámica:

```
#include <stdio.h>
#include <stdlib.h>
struct alumno{
        char a[20];
        char b[30];
        int edad;
};
main(){
alumno *b;
int max,i;
printf("Da el numero de registros a manejar=>");
scanf("%d",&max);
b=(alumno *)malloc(sizeof(alumno)*max);
for (i=0;i<max;i++){
        printf("Da el nombre del elemento %d=>",i);
        scanf("%s",(b+i)->a);
```

```
        printf("Da la direccion del elemento %d=>",i);
        scanf("%s",(*(b+i)).b);
        printf("Da la edad del elemento %d=>",i);
        scanf("%d",&(b+i)->edad);
}
for (i=0;i<max;i++){
        printf("%s\t%s\t%d\n",(b+i)->a,(b+i)->b,(b+i)->edad);
        printf("%s\t%s\t%d\n",(*(b+i)).a,(*(b+i)).b,(*(b+i)).edad);
}
getchar();
getchar();
}
```

Programa 3.11

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

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

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

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

Nota: El operador –new()- se puede emplear en lugar del operador -malloc()- y el operador -delete()- se puede emplear en lugar del operador -free()-. Si se decide emplear el operador –new()-, para liberar memoria se tiene que utilizar forzosamente el operador -delete()-. Si se utiliza el operador –malloc()-, para liberar memoria se tiene que utilizar forzosamente el operador –free()-. No es recomendable mezclar operadores.

Capítulo IV

FUNCIONES

4.1 PASO DE PARÁMETROS

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

PASO DE PARÁMETROS POR VALOR

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

PASO DE PARÁMETROS POR REFERENCIA

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

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

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

Nótese que el paso de parámetros por valor no implica que no puedan ocurrir cambios fuera del procedimiento mediante el uso de parámetros. Si el parámetro tiene un tipo de apuntador o referencia, entonces el valor es una dirección, y puede utilizarse para cambiar

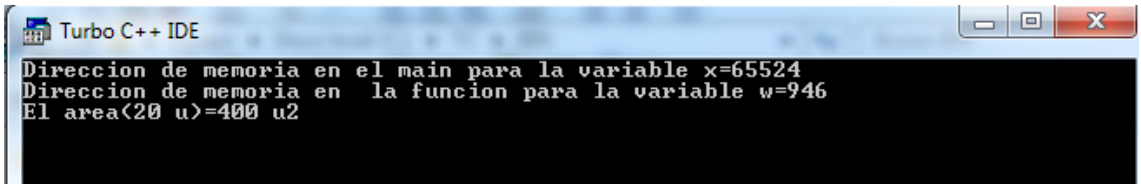
la memoria por fuera del procedimiento. Por ejemplo, la siguiente función en C cambia de manera definitiva el valor del entero a la cual el parámetro –p- apunta:

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

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

Observemos el programa 4.1:

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



Programa 4.1

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

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

Observe que al ejecutarse el programa 4.1, la variable –x- tiene una dirección de memoria independiente a la variable –w- y la variable –w- solo recibe una copia del valor guardado en la localidad de la variable- x-. (Observe que la función área_cuadrado() dentro de su declaración indica que va a retornar un valor del tipo entero).

¿Qué sucede si en vez de enviar el valor de la variable se envía su dirección?

Observe el programa 4.2:

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

Programa 4.2

En el programa 4.2, como en el programa anterior, la función main() tiene las variables locales enteras –x- y –y-. La función área_cuadrado() tiene la variable local apuntador a entero –w-.

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

4.2 PASO DE PARÁMETROS EN VECTORES

Como recordaremos, al declarar el nombre de un vector lo que realmente se está haciendo es declarar un apuntador al inicio del arreglo. Por lo que al enviar un vector como parámetro se enviará la dirección de inicio del vector (en este caso, por las características del lenguaje C, para vectores y matrices solo se acepta el paso de parámetros por referencia). En el programa 4.3, las funciones –imprime()- e -imprime1()- muestran las formas en las que se puede manejar un vector. Como se observa, se puede manejar como un vector o como un apuntador; es indistinto.

```
#include <stdio.h>
void imprime(int *);
void imprime1(int []);
main(){
int i ,a[]={1,2,3,4,5};
for (i=0;i<5;i++)

   printf("%u..",(a+i));
   putchar('\n');

imprime(a);
imprime1(a);
getchar();
}

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

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

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

Programa 4.3

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

Un detalle importante es que el lenguaje C no verifica los acotamientos tanto inferior como superior, por lo que puede sobrescribir o puede mostrar basura. Observe el programa 4.4:

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

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

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

Programa 4.4

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

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

4.3 PASO DE PARÁMETROS EN MATRICES ESTÁTICAS

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

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

Por lo tanto, la notación:

A[n][m]

Nos indica que los elementos del arreglo están guardados en renglón por renglón.

Cuando se pasa un arreglo bidimensional a una función se debe especificar el número de columnas ya que el número de renglones es irrelevante.

La razón de lo anterior es nuevamente los apuntadores. C requiere conocer cuántas son las columnas para que pueda brincar de renglón en renglón en la memoria.

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

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

o aún como:

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

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

Por lo tanto:
*int (*a)[35]* declara un apuntador a un arreglo de 35 enteros y, por ejemplo, si hacemos la siguiente referencia a+2, nos estamos refiriendo a la dirección del primer elemento que se encuentra en el tercer renglón de la matriz supuesta, mientras que int *a[35]; está declarando un arreglo de 35 apuntadores a enteros. Observe el programa 4.5:

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

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

void matriz1(int b[][3]){
int i,j;
for (i=0;i<H;i++){
   for (j=0;j<3;j++)
     printf("%d\t",b[i][j]);
     putchar('\n');
}
}
```

```
1       2       3
4       5       6
7       8       9
10      11      12

1       2       3
4       5       6
7       8       9
10      11      12
```

Programa 4.5

En este programa las funciones matriz() y matriz1() se emplean indistintamente la notación tipo apuntador como la notación matricial. Al igual que en vectores, en matrices C no cuida el límite inferior ni el superior. Por ejemplo, el programa 4.6 muestra:

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

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

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

```
-2      1976998106      16      1       2       3
1       2       3       4       5       6
4       5       6       7       8       9
7       8       9       10      11      12
10      11      12      0       0       2293624
0       0       2293624 4198887 1       5443440
4198887 1       5443440 5444912 -1      2293616

2028240130      -2      1976998106      16      1       2       3
16      1       2       3       4       5       6
3       4       5       6       7       8       9
6       7       8       9       10      11      12
9       10      11      12      0       0       2293624
12      0       0       2293624 4198887 1       5443440
2293624 4198887 1       5443440 5444912 -1      2293616
```

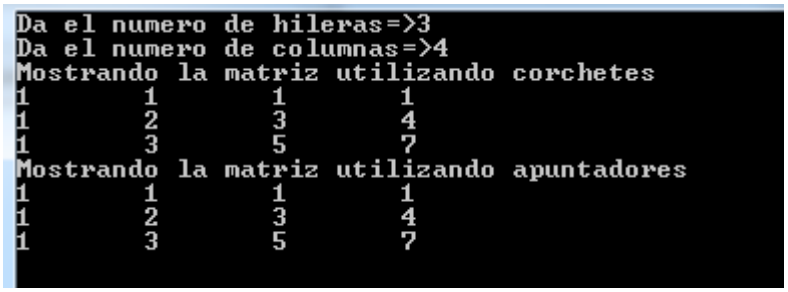
Programa 4.6

4.4 PASO DE PARÁMETROS CON MATRICES DINÁMICAS

Si el usuario no conoce el tamaño de la matriz se puede utilizar la memoria en forma dinámica como se analizó en el capítulo 3. El argumento de la matriz se manejará en la función llamada como un apuntador doble, además se debe de incluir el número de hileras y el número de columnas para que la función llamada pueda manejar en forma adecuada la dimensión de la matriz. Estos elementos se muestran en el programa 4.7:

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

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

```
for (j=0;j<col;j++)
printf("%d\t",*(*(b +i)+j));
putchar('\n');
  }
}
```



Programa 4.7

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

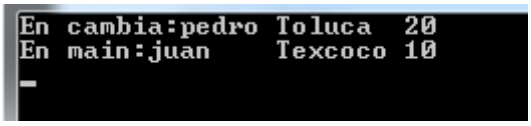
Un punto interesante es que en algunos compiladores el programa compila bien pero no ejecuta en forma apropiada (el programa ejecutó bien en dev-c++, en Turbo C y en Turbo C portable, no siendo así en Borland C).

4.5 PASO DE PARÁMETROS EN REGISTROS

Si se desea enviar como parámetro solo un registro, el paso parámetro será por valor. Por ejemplo, observe el programa 4.8:

```
#include <stdio.h>
#include <string.h>
struct alumno{
      char a[10];
      char b[10];
      int c;
      };
void cambia(alumno );
main(){
      alumno alumnos;
      strcpy(alumnos.a,"juan");
      strcpy(alumnos.b,"Texcoco");
      alumnos.c=10;
      cambia(alumnos);
      printf("En main:%s\t%s\t%d\n",alumnos.a,alumnos.b,alumnos.c);
```

```
      getchar();
      }

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



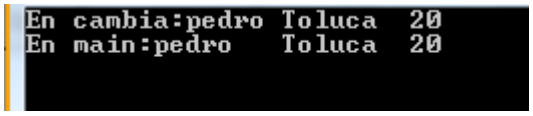
Programa 4.8

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

Para poder realizar paso de parámetros por referencia se tiene que realizar el envío de la dirección de memoria en forma explícita con el operador -&- y la recepción del parámetro se realizará con el operador -*-. Observe el programa 4.9:

```
#include <stdio.h>
#include <string.h>
struct alumno{
      char a[10];
      char b[10];
      int c;
      };
void cambia(alumno *);
main(){
      alumno alumnos;
      strcpy(alumnos.a,"juan");
      strcpy(alumnos.b,"Texcoco");
      alumnos.c=10;
      cambia(&alumnos);
      printf("En main:%s\t%s\t%d\n",alumnos.a,alumnos.b,alumnos.c);
      getchar();
      }

void cambia(alumno *b){
    strcpy(b->a,"pedro");
```

```
    strcpy(b->b,"Toluca");
    b->c=20;
    printf("En cambia:%s\t%s\t%d\n",b->a,b->b,b->c);
    }
```



Programa 4.9

Si se desea enviar como parámetro un arreglo de registros se procede como se muestra en el programa 4.10:

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

void cambia(alumno *);
void cambia1(alumno[]);
main(){
      int i;
      alumno alumnos[3];
      strcpy(alumnos[0].a,"juan"); strcpy(alumnos[0].b,"Texcoco"); alumnos[0].c=10;
      strcpy(alumnos[1].a,"luis"); strcpy(alumnos[1].b,"Tepexpan"); alumnos[1].c=20;
      strcpy(alumnos[2].a,"lucas"); strcpy(alumnos[2].b,"Tocuila"); alumnos[2].c=10;
      cambia(alumnos);
      putchar('\n');
      for (i=0;i<3;i++)
         printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
      putchar('\n');
      cambia1(alumnos);
      putchar('\n');
      for (i=0;i<3;i++)
         printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
      getchar();
}
```

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

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

```
En cambia:juana Texas   100
En cambia:luisa Tampa   200
En cambia:lola  Tulane  300

En main:juana   Texas   100
En main:luisa   Tampa   200
En main:lola    Tulane  300

En cambia:Lula  Taxco   100
En cambia:Cuca  Sonora  200
En cambia:Peche Tula    300

En main:Lula    Taxco   100
En main:Cuca    Sonora  200
En main:Peche   Tula    300
```

Programa 4.10

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

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

4.6 APUNTADORES A FUNCIONES

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

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

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

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

Lo anterior es útil cuando se deben usar distintas funciones, quizás para realizar tareas similares con los datos. Por ejemplo, se pueden pasar como parámetros los datos y la función que serán usados por alguna función de control. Como se verá más adelante, la biblioteca estándar de C tiene funciones para ordenamiento (qsort()) y para realizar búsqueda (bsearch()), a las cuales se les pueden pasar funciones como parámetros (ver ejemplos en capítulo 5).

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

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

Aquí –pf- es un apuntador a una función que no envía parámetros y retorna un tipo de dato entero. Observe que se ha declarado el apuntador y en este momento no se ha dicho a qué variable va a apuntar.
Supongamos que se tiene una función –*int f()*;- entonces simplemente se debe de escribir:

```
pf= f;
```

Para que la variable –pf- apunte al inicio de la función -f()-.
Los apuntadores a funciones se declaran en el área de prototipos, por ejemplo:

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

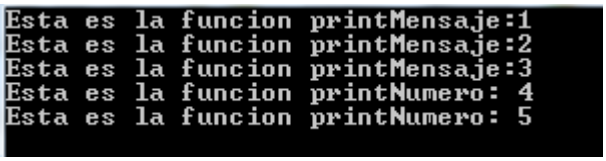
En consecuencia, en el cuerpo del programa se pueden tener asignaciones como:

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

Los cuales son equivalentes.

Observe el programa 4.11:

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



Programa 4.11

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

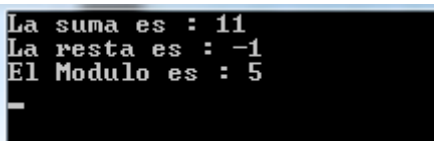
En la función principal, llamamos a la función –printMensaje()- con el valor uno como parámetro, en la línea siguiente asignamos al puntero a función (*funcPuntero) el valor de –printMensaje()- y utilizamos el apuntador para llamar a la misma función de nuevo. Por tanto, las dos llamadas a la función –printMensaje()- son idénticas gracias a la utilización del puntero -(*funcPuntero)-.

Dado que hemos asignado el nombre de una función a un apuntador a función, y el compilador no da error, el nombre de una función debe ser un apuntador a una función. Esto es exactamente lo que sucede. Un nombre de una función es un apuntador a esa función, pero es un apuntador constante que no puede ser cambiado. Sucede lo mismo con los vectores: el nombre de un vector es un apuntador constante al primer elemento del vector.

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

Veamos el código del programa 4.12:

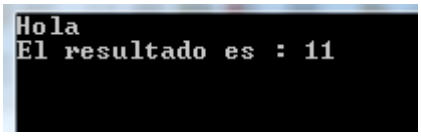
```
#include <stdio.h>
#include <conio.h>
int MiFuncionSuma(int,int);
int MiFuncionResta(int,int);
int MiFuncionModulo(int,int);
main(){
        int (*ptrFn) (int,int);
        ptrFn = MiFuncionSuma;
        printf("La suma es : %d\n", ptrFn(5, 6));
        ptrFn = MiFuncionResta;
        printf("La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



Programa 4.12

En el programa 4.13 se observa la declaración del apuntador dentro de la función –main()- y se muestra la forma en que se puede enviar como parámetro un apuntador a función que retorna a su vez un apuntador tipo entero.

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

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

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



Programa 4.13

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

Ahora se realizará un ejemplo donde la función tiene como parámetro a un apuntador a una función.

Primero se definirá la variable tipo apuntador a función:

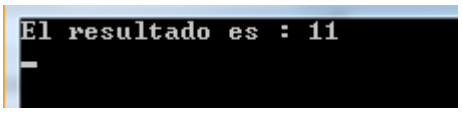
```
int (*ptrFn) (int, int);
```

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

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

En el programa 4.14 se mostrará el empleo de este tipo de declaración:

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

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



Programa 4.14

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

void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve **void.**
void (*fptr)(int);	fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un **int** y un char como argumentos y devuelve un **int**.
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a int.
CUANDO EL VALOR DEVUELTO POR LA FUNCIÓN ES A SU VEZ UN PUNTERO (A FUNCIÓN, O DE CUALQUIER OTRO TIPO), LA NOTACIÓN SE COMPLICA UN POCO MÁS:	
int const * (*fptr)();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un **int** constante
float (*(*fptr)(char))(int);	fptr es un puntero a función que recibe un **char** como argumento y devuelve un puntero a función que recibe un **int** como argumento y devuelve un **float**.
void * (*(*fptr)(int))[5];	fptr es un puntero a función que recibe un **int** como argumento y devuelve un puntero a un array de 5 punteros-a-**void** (genéricos).
char (*(*fptr)(int, float))();	fptr es un puntero a función que recibe dos argumentos (**int** y **float**), devolviendo un puntero a función que no recibe argumentos y devuelve un **char**.
long (*(*(*fptr)())[5])();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven **long**.
void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve **void**.
void (*fptr)(int);	fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int.**
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a **int.**
int (* afptr[10])(int);	afptr es una matriz de 10 apuntadores a función

Capítulo V

ALGUNOS EJEMPLOS INTERESANTES

INTRODUCCIÓN

En este capítulo se muestran algunos programas en los cuales se utilizan los apuntadores en forma extensa. Para su mayor comprensión se requiere que el lector haya tomado o esté tomando un curso sobre estructura de datos, ya que se requiere el conocimiento de funciones recursivas y listas enlazadas. Se espera que al lector le permita tener una mejor visión de la programación de apuntadores y su importancia dentro de la programación.

5.1 MATRICES

5.1.1 CÁLCULO DE DETERMINANTES EN FORMA RECURSIVA

El cálculo del determinante por menores de una matriz se puede definir en forma recursiva como:

$$DET(N, A_N) \rightarrow \begin{cases} \sum_{j=0}^{N-1} DET(N\text{-}1, A_{N-1}) * a[0][j]*(-1)^j & \text{Si } N>2 \\ a[0][0]*a[1][1]-a[1][0]*a[0][1] & \text{Si } N=2 \\ a[0][0] & \text{Si } N=1 \end{cases}$$

En el programa 5.1, en la llamada recursiva DET(N-1AN-1), al eliminarse una hilera y una columna a la matriz, se eliminará la primera hilera y la j-ésima columna:

```
#include <stdio.h>
#include <stdlib.h>
//PROTOTIPOS
int signo(int);
int determinante(int ** ,int);
int ** subm(int , int ** , int);
main(){
      int f,i,j;
      int **pm;
      printf("Da el tamaño de la matriz=>");
      scanf("%d",&f);
      pm=(int **)malloc(sizeof(int *)*f);
      for (j=0;j<f;j++)
            pm[j]=(int*)malloc(sizeof(int)*f);
      for (i=0;i<f;i++)
            for (j=0;j<f;j++){
        printf("a[%d][%d]=",i,j);
              scanf("%d",&pm[i][j]);
        }
```

```
        printf("LA MATRIZ ES:\n");
        for (i=0;i<f;i++){
                for (j=0;j<f;j++)
                        printf("%d\t",pm[i][j]);
              putchar('\n');
        }
        printf("\nDeterminante(A)=%d\n",determinante(pm,f));
        getchar();
        getchar();
}
int determinante(int ** a, int tam){
if (tam==1)
   return a[0][0];
if (tam==2)
   return a[0][0]*a[1][1]-a[1][0]*a[0][1];
int m=0;
int tam1=tam-1;
for (int i=0;i<tam;i++)
    m=signo(i)*determinante(subm(i,a,tam1),tam1)*a[0][i]+m;
return m;
}
int ** subm(int k, int ** a,int tam){
        int ** b;
        int i,j;
        b=(int **)malloc(sizeof(int *)*tam);
        for (j=0;j<tam;j++)
                b[j]=(int*)malloc(sizeof(int)*tam);
                   for (i=0; i<tam;i++){
                           int m=0;
                           for (j=0; j<tam+1;j++)
                                   if (j != k){
                                           b[i][m]=a[i+1][j];
                                           m++;
                                   }

                   }
                   return b;
}
int signo(int i){
                   if (i%2==0)
                           return 1;
                   return -1;
}
```

```
Da el tama±o de la matriz=>4
a[0][0]=0
a[0][1]=1
a[0][2]=2
a[0][3]=-3
a[1][0]=1
a[1][1]=3
a[1][2]=2
a[1][3]=-5
a[2][0]=2
a[2][1]=4
a[2][2]=3
a[2][3]=1
a[3][0]=3
a[3][1]=-2
a[3][2]=-8
a[3][3]=1
LA MATRIZ ES:
0       1       2       -3
1       3       2       -5
2       4       3       1
3       -2      -8      1

Determinante(A)=91
```

Programa 5.1

5.1.2 PROBLEMA DEL AGENTE VIAJERO

El problema del agente viajero responde a la siguiente pregunta: dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen? Este es un problema NP-duro y es muy importante en la investigación de operaciones y en la ciencia de la computación.

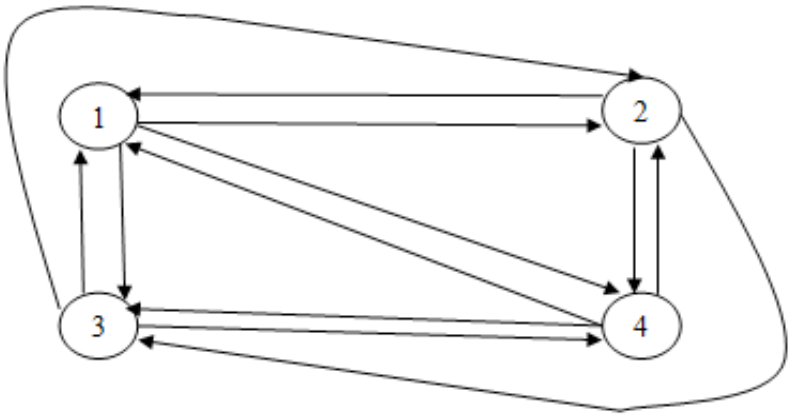
El problema fue formulado por primera vez en 1930 y es uno de los problemas de optimización más estudiados. Es usado como prueba para muchos métodos de optimización. Aunque el problema es computacionalmente complejo, una gran cantidad de heurísticas y métodos exactos son conocidos, de manera que algunas instancias desde cien hasta miles de ciudades pueden ser resueltas.

Su complejidad se debe a que es un problema de permutaciones. Un problema de permutaciones usualmente será mucho más difícil de resolver que problemas donde se escogen subconjuntos ya que existen n! permutaciones de n objetos mientras que solo existen 2^n diferentes subconjuntos de n objetos (n! > O(2^n). Observe la tabla 5.1.

n	2^n	n!
1	2	1
2	4	2
4	16	24
8	256	40320

Tabla 5.1

El problema del agente viajero se puede representar en forma gráfica dirigida donde el tamaño de las aristas se dan en la matriz C, como se muestra en la figura 5.1:



$$C = \begin{vmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{vmatrix}$$

Fig. 5.1 Gráfica dirigida cuya longitud de cada arista se localiza en la matriz C.

Donde para recorrer de la ciudad uno a la ciudad dos se tiene una distancia de diez unidades de longitud y para recorrer de la ciudad tres a la ciudad dos se requieren trece unidades de longitud, por ejemplo.

De esta forma, una gira en la gráfica es un ciclo dirigido que incluye todos los vértices en V. El costo de la gira es la suma de los costos de las aristas en la gira. El problema del agente viajero es encontrar una gira que minimice los costos.

5.1.2.1 SOLUCIÓN POR PROGRAMACIÓN DINÁMICA

PRINCIPIO DE OPTIMALIDAD

Cuando hablamos de *optimizar* nos referimos a buscar alguna de **las mejores soluciones** de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estra-

tegia voraz. En otros casos, aunque no sea posible aplicar la estrategia voraz, se cumple el **principio de optimalidad de Bellman** enunciado en 1957 y que dicta que "dada una secuencia óptima de decisiones, **toda subsecuencia** de ella es, a su vez, óptima". En este caso sigue siendo posible el ir tomando decisiones elementales, con la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica. Aunque este principio parece evidente no siempre es aplicable y, por tanto, es necesario verificar que se cumple para el problema en cuestión.

En la siguiente discusión se comentará sobre un recorrido que inicia en el vértice 1 y termina en el mismo vértice, siendo el recorrido el del mínimo costo. Siendo V el conjunto de vértices, se puede decir que toda gira consiste de una arista <1, k> para algún k ϵ V – {1} y un trayecto desde el vértice k al vértice 1. El trayecto desde el vértice k al vértice 1 va a través de cada vértice en V – {1, k}. De aquí que el principio de optimización se mantiene. Sea g(i, S) la longitud del trayecto más corto iniciando en el vértice i, yendo a través de todos los vértices en S y terminando en el vértice 1. g(1, V – {1}) es la longitud de una gira óptima de un agente viajero. Desde el principio de optimalidad se deduce que:

$$g(1, V-\{1\}) = \min_{2<=k<=n}\{ c_{i\,k} + g(k, V-\{1,k\})\} \quad \ldots\ldots\ldots 1$$

Generalizando, se obtiene (para i ¢ S):

$$g(i, S) = \min_{j \epsilon S} \{ c_{i\,j} + g(j, S-\{j\})\} \quad \ldots\ldots\ldots\ldots\ldots 2$$

Se puede resolver g(1, V-{1}) si conocemos g(k, V-{1,k}) para todas las opciones de k. Los valores de g pueden ser obtenidos usando (2). Claramente, g(i, ϕ) = $C_{i,1}$, $1 \leq i \leq n$. De aquí podemos usar (2) para obtener g(i,S) para todo S de tamaño 1, entonces se puede obtener g(i,S) para S con |S| = 2, etcétera. Cuando |S| < n – 1, los valores de i y S para el cual se necesita

g(i, S) son tales que i ≠ 1; 1 $\notin$ S e i $\in$S.

Ejemplo: Considere la gráfica dirigida de la figura 5.1, a partir de la cual se obtiene:
$g(2,\phi)=c_{2,1}$ =5; $g(3,\phi)=c_{3,1}$ =6; $g(4,\phi)=c_{4,1}$ =8;

Utilizando a (2) se obtiene:

g(2, {3}) = $c_{2,3}$ + g(3,ϕ) = 15;	g(2, {4}) = c_{24} + g(4,ϕ) = 18;
g(3, {2}) = 18;	g(3, {4}) = 20;
g(4, {2}) = 13;	g(4, {3}) = 15;

Ahora, calculamos g(i, S) con |S|=2, i ≠ 1, 1¢ S e i ¢ S.

$g(2, \{3,4\})=\min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25$

$g(3, \{2,4\})=\min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25$

$g(4, \{2,3\})=\min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23$

Finalmente, de (1) obtenemos:

$g(1, \{2,3,4\}) = \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2,4\}), c_{14} + g(4, \{2,3\})\}$

$= \min \{35, 40, 43\}$

$= 35.$

Una forma de programar el agente viajero por programación dinámica es utilizando la recursividad. Enla figura 5.2 se muestra el árbol recursivo para el ejemplo anterior:

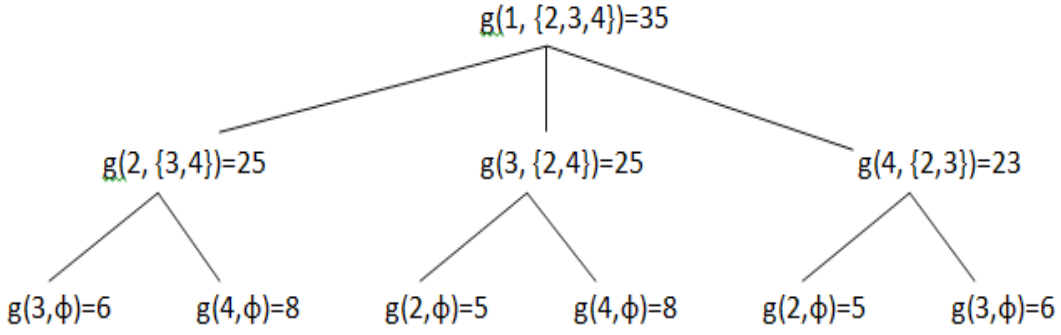


Figura 5.2

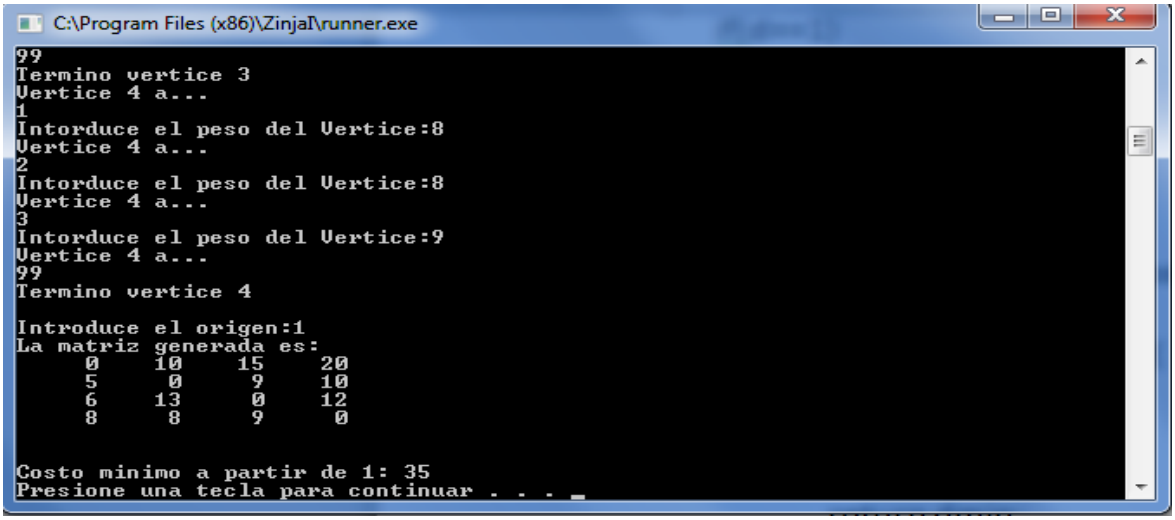
Observe que cada nodo es óptimo para su recorrido hacia la raíz, por lo que se mantiene el principio de optimalidad.

El programa 5.2 muestra la programación del agente viajero en forma recursiva utilizando el principio de optimalidad:

```
#include<stdio.h>
#include<stdlib.h>
void generar(int **,int);
int tsp(int **,int *,int,int,int,int);

main(){
      int **cost,*m,d,o,v,i,j,r;
      printf("       ***************    TSP    ***************\n");
      printf("\nIntroduce el numero de ciudades: ");
      scanf("%d",&v);
      cost=(int **)malloc(sizeof(int *)*v);
```

```
    for(i=0;i<v;i++)
                cost[i]=(int *)malloc(sizeof(int)*v);
    m=(int *)malloc(sizeof(int)*v);
    for(i=0;i<v;i++)
      m[i]=0;
    for(i=0;i<v;i++)
      for(j=0;j<v;j++){
            if(i==j)
                  cost[i][j]=0;
            else
                  cost[i][j]=9999;
      }
    generar(cost,v);
    printf("\nIntroduce el origen:");
    scanf("%d",&r);
    printf("La matriz generada es:  \n");
    for(i=0;i<v;i++){
    for(j=0;j<v;j++){
    printf("%6d",cost[i][j]); } printf("\n"); }
    printf("\n\nCosto minimo a partir de %d: %d\n",r,tsp(cost,m,v,r-1,v,r-1));
    system("PAUSE");
    }

void generar(int **cost, int tam){
    int i,nv=0,p,ad;
    printf("Para terminar de introducir los costos introduce 99\n");
    while(nv<tam){
          printf("Vertice %d a... \n",nv+1);
          scanf("%d",&ad);
          if(ad==99){
          printf("Término vértice %d\n",nv+1);
          nv++;
}
          else if(ad>tam)
          printf("La ciudad no existe \n");
          else{
              printf("Introduce el peso del vertice:");
              scanf("%d",&p);
              cost[nv][ad-1]=p;
          }
     }
}
```

```
int tsp(int **cost,int *m,int d,int o,int v,int r){
    if(d==1)
    return cost[o][r];
    int dist,dmin=999;
    m[o]=1;
    for(int  i =0;i<v;i++)
     if(m[i]==0){
              dist=cost[o][i]+tsp(cost,m,d-1,i,v,r);
              m[i]=0;
                       if(dist<dmin){
                       dmin=dist;
                       }
              }
              return dmin;
    }
```



Programa 5.2

5.2 PROGRAMACIÓN DE LISTAS ENLAZADAS

Una lista es una colección de elementos llamados generalmente nodos. El orden entre los nodos se establece por medio de apuntadores (realizando el enlace entre nodos), es decir, direcciones o referencias a otros nodos. Las operaciones más comunes son:

- Recorrido de la lista
- Inserción de un elemento
- Borrado de un elemento
- Búsqueda de un elemento

Los tipos de datos abstractos más conocidos son la pila, la lista ordenada, la lista doblemente enlazada ordenada y el árbol binario, su estudio es común dentro de la Asignatura "Estructura de Datos" de cualquier licenciatura cuyo campo de estudio es el cómputo y son un ejemplo claro del manejo de memoria en forma dinámica. Por lo que en las siguientes sub-secciones se mostrará la programación de estos elementos. (Parte de las definiciones escritas en estas sub-secciones fueron tomadas de [Cairo, 1993])

5.2.1 PROGRAMACIÓN DE UNA PILA

Una pila (stack en inglés) es una lista de elementos a la cual se pueden insertar o eliminar elementos solo por uno de los extremos (apilar o retirar un elemento de la pila). En consecuencia, los elementos de una pila serán eliminados en orden inverso al que se insertaron. Es decir, el último elemento que se mete en la pila es el primero que se saca.

Debido al orden en el cual se insertan o eliminan elementos de una pila a esta estructura, también se le conoce como estructura LIFO (Last-In, First-Out: Último en entrar, primero en salir).

Las pilas pertenecen al grupo de estructuras de datos lineales, porque los componentes ocupan lugares sucesivos en la estructura.

Algunos de los ejemplos donde se utiliza la pila son:

- Evaluación de expresiones en notación posfija.
- Reconocedores sintácticos de lenguajes independientes del contexto.
- Implementación de la recursividad.

Las operaciones elementales que pueden realizarse en la pila son:

- Colocar un elemento en la pila.
- Eliminar un elemento de la pila.
- Mostrar la pila.

La estructura a usar en el programa es la siguiente:

```
struct nodo{
        int dato;
        nodo * siguiente;
}
```

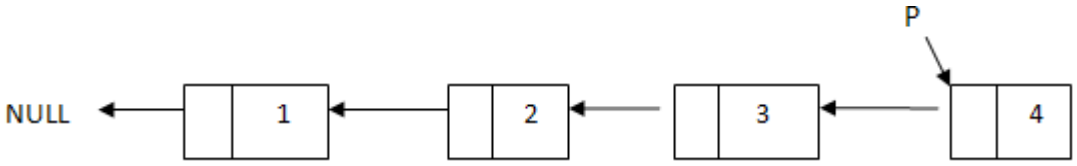
Esta estructura dinámica se puede representar en forma gráfica de la siguiente manera:

Observe que la variable –siguiente- es un apuntador a la misma estructura, esto nos permite poder realizar un enlace con estructuras del mismo tipo (de ahí el nombre de estructuras enlazadas).

En la figura 5.3 se muestra la forma en que se manipula la pila, por lo que el primer elemento introducido es la estructura donde la variable –dato- guarda el valor uno y el último elemento en la pila es la estructura donde la variable –dato- guarda el valor cuatro.

La impresión de la pila será:

4..3..2..1..



Al eliminar un elemento de la pila, esta se verá de la siguiente forma:

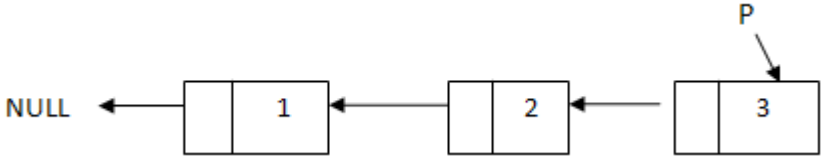


Figura 5.3

El programa 5.3 muestra la programación de una pila en forma dinámica en el que se permite insertar, mostrar o eliminar elementos:

```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
        int dato;
        nodo * siguiente;
};

//PROTOTIPOS
int menu();
void * crear (void *);
void * eliminar(void *);
void mostrar(void *);

main(){
void *p=NULL;
int eleccion;
do{
        eleccion=menu ();
        switch(eleccion){
                case 1:p=crear(p);continue;
                case 2:p=eliminar(p);break;
                case 3:mostrar(p);continue;
                default:printf("FIN DE LAS OPERACIONES");
        }
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion,x;
do{
        printf("\n\t\tMENU PRINCIPAL\n");
        printf("\t1.-Introducir un elemento a la pila\n");
        printf("\t2.-Eliminar un elemento de la pila\n");
        printf("\t3.-Mostrar el contenido de la pila\n");
        printf("\t4.-Salir\n");
        scanf("%d",&eleccion);
}while(eleccion<1||eleccion>4);
putchar('\n');
```

```
return(eleccion);
}
void *crear(void *p){
nodo *q,*aux;
putchar('\n');
printf("Indica el valor a introducir a la pila=>");
q=(nodo*)malloc(sizeof(nodo));
scanf("%d",&q->dato);
q->siguiente=NULL;
if(p==NULL)    //La pila esta vacia
        p=q;
else{
    q->siguiente=(nodo*)p;
    p=q;
}
return(p);
}

void * eliminar(void * s){
nodo *p,* aux;
if(s==NULL)
        printf("\npila vacia\n");
else{
        p=(nodo*)s;
        s=p->siguiente;
        free(p);
}
return(s);
}

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf("PILA VACIA");
else
        do{
                printf("%d..",s->dato);
                s=s->siguiente;
        }while(s!=NULL);
}
```

Programa 5.3

Para poderse mantener la pila en memoria se requiere un apuntador principal al final de la pila, el apuntador utilizado en el programa es del tipo –void-, se requiere en cada función invocada realizar un casting para la manipulación de la pila. En la función –crear()- se tiene el casting:

q->siguiente=(nodo*)p;

Cada vez que se ejecuta la función –crear()-, en la instrucción:

q=(nodo*)malloc(sizeof(nodo));

La función –sizeof()- indica el tamaño de la estructura –nodo- en bytes. Posteriormente, la función –malloc()- reserva memoria del tamaño indicado en la función –sizeof()- y finalmente la memoria reservada se utilizará como tipo nodo y se regresa un apuntador a una variable del tipo apuntador a nodo.

De esta forma se crea en forma dinámica una estructura del tipo nodo. Y en el código:

q->siguiente=(nodo*)p;

Se enlazan dos estructuras. Observe que la variable –p- es un apuntador del tipo void y la variable –q- es un apuntador del tipo nodo, si se realiza una igualdad de la forma:

q->siguiente=p;

se tendrá un error del tipo mismatch (las variables no son compatibles), esto se entiende ya que se está asignando una variable del tipo general (void) a una variable del tipo específico (nodo *).

5.2.2 PROGRAMACIÓN DE UNA LISTA SIMPLEMENTE ENLAZADA ORDENADA

Una lista ordenada es una lista enlazada con un orden, siendo el orden una estructura interna que debe mantenerse en todo momento.

Las operaciones elementales que pueden realizarse en la lista ordenada son:

- Colocar un elemento en la lista.
- Eliminar un elemento de la lista.
- Mostrar la lista.

Para colocar un elemento en la lista ordenada se deben de observar cuatro posibles casos:

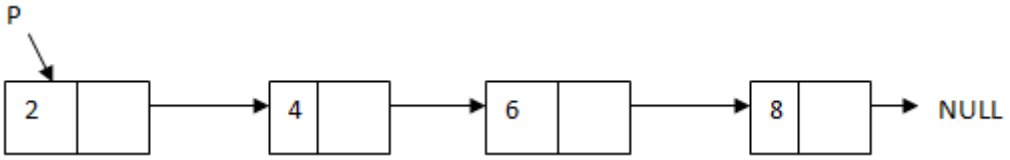
- La lista está vacía.
- El elemento a colocar es menor a todos los elementos que se encuentran en la lista.
- El elemento a colocar es mayor a todos los elementos que se encuentran en la lista.
- El elemento a colocar debe de estar entre la lista.

Cada caso se observa en la función –crear()- del siguiente programa.

- Para eliminar un elemento de la lista se tienen los siguientes casos:
- El elemento a eliminar es el primero y único de la lista.
- El elemento a eliminar es el primero y no es el único.
- El elemento a eliminar es el último de la lista.
- El elemento a eliminar se encuentra entre la lista.

Cada caso se observa en la función –eliminar()- del siguiente programa.

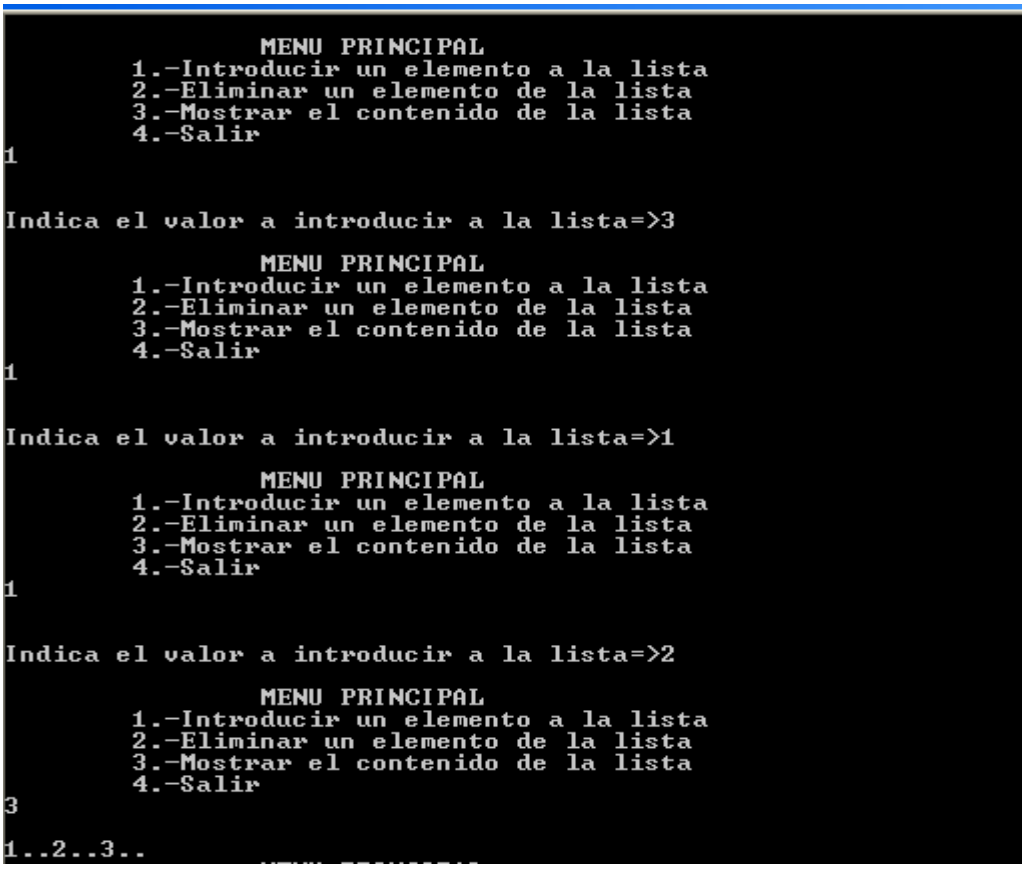
Por lo que el programa 5.4 ordena los elementos de menor a mayor creando el espacio del elemento a guardar en forma dinámica. Además, permite eliminar un elemento sin perder el orden establecido:




```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
        int dato;
        nodo * siguiente;
};
//PROTOTIPOS
int menu();
void * crear (void * );
void * eliminar(void * );
void mostrar(void * );

main(){
void *p=NULL;
int eleccion;
do{
        eleccion=menu ();
        switch(eleccion){
                case 1:p=crear(p);continue;
                case 2:p=eliminar(p);break;
                case 3:mostrar(p);continue;
                default:printf("FIN DE LAS OPERACIONES");
        }
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion,x;
do{
        printf("\n\t\tMENU PRINCIPAL\n");
```

```
        printf("\t1.-Introducir un elemento a la lista\n");
        printf("\t2.-Eliminar un elemento de la lista\n");
        printf("\t3.-Mostrar el contenido de la lista\n");
        printf("\t4.-Salir\n");
        scanf("%d",&eleccion);
}while(eleccion<1||eleccion>4);
putchar('\n');
return(eleccion);
}



void *crear(void *p){
nodo *q,*aux,*aux1;
int x;
putchar('\n');
printf("Indica el valor a introducir a la lista=>");
scanf("%d",&x);
q=(nodo*)malloc(sizeof(nodo));
q->dato=x;
q->siguiente=NULL;
if(p==NULL)//Lista vacia
        p=q;
else{
        aux1=aux=(nodo*)p;
        if (x<aux->dato){    //Elemento menor a cualquier elemento de la lista
                q->siguiente=aux;
                p=q;
        }
        else{   //el elemento se colocara entre la lista o sera el ultimo
                while(aux!=NULL&&aux->dato<x){
                        aux1=aux;
                        aux=aux->siguiente;
                }
                aux1->siguiente=q;
                q->siguiente=aux;

        }
}
return(p);
}
```

```
void * eliminar(void * s){
if(s==NULL)
        printf("\nLISTA VACIA\n");
else{
        nodo *p,* aux;
        int x;
        printf("Da el elemento a eliminar=>");
        scanf("%d",&x);
        aux=p=(nodo*)s;
        if(p->siguiente==NULL&&aux->dato==x)//solo hay un elemento en la lista
                s=NULL;
        else {
                while(p->siguiente!=NULL && p->dato<x){
                        aux=p;
                        p=p->siguiente;
                }
                if (p!=NULL)
                        if (p->dato==x && p==s) //No existio movimiento
                                s=p->siguiente;
                        else if(p->dato==x)
                                aux->siguiente=p->siguiente;
                        else
                                printf("DATO NO LOCALIZADO\n");
                else
                        printf("DATO NO ENCOTRADO\n");
        }
        free(p);
}
return(s);
}

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf("LISTA VACIA");
else
        do{
                printf("%d..",s->dato);
                s=s->siguiente;
        }while(s!=NULL);
}
```



Programa 5.4

5.2.3 PROGRAMACIÓN DE UNA LISTA DOBLEMENTE ENLAZADA ORDENADA

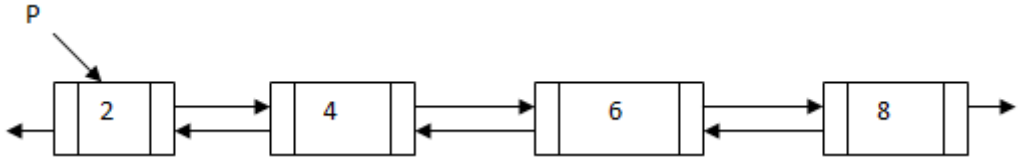
La lista puede ser doblemente enlazada por lo que permite ordenar la lista y recorrerla de izquierda a derecha y de derecha a izquierda, permitiendo en este caso mostrar los números ingresados en forma ascendente o en forma descendente. (Es el preámbulo para poder programar un árbol binario).

Los casos para la inserción y eliminación son similares a las listas simplemente enlazadas ordenadas con la única diferencia de que se tiene que realizar el doble enlace al momento de introducir un nuevo elemento.

Por lo que la estructura tiene dos apuntadores:

```
struct nodo{
        int dato;
        nodo * izq;   //Apuntador al nodo izquierdo
        nodo * der;  //Apuntador al nodo derecho
}
```

Gráficamente quedaría de la siguiente forma:



El programa 5.5 permite introducir elementos en forma ordenada, mostrar elementos en forma ascendente y en forma descendente y eliminación de elementos.

```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
        int dato;
        nodo * izq;
        nodo * der;
};

//PROTOTIPOS
int menu();
void * crear (void * );
void * eliminar(void * );
void mostrarasc(void * );
void mostrardesc(void *);
main(){
void *p=NULL;
int eleccion;
do{
        eleccion=menu();
        switch(eleccion){
                case 1:p=crear(p);continue;
                case 2:p=eliminar(p);break;
                case 3:mostrarasc(p);continue;
                case 4:mostrardesc(p);continue;
                default:printf("FIN DE LAS OPERACIONES");
        }
```

```
}while(eleccion<5);
return 0;
}

int menu(){
int eleccion,x;
do{
        printf("\n\t\tMENU PRINCIPAL\n");
        printf("\t1.-Introducir un elemento a la lista\n");
        printf("\t2.-Eliminar un elemento de la lista\n");
        printf("\t3.-Mostrar el contenido de la lista en forma ascendente\n");
        printf("\t4.-Mostrar el contenido de la lista en forma descendente\n");
        printf("\t5.-Salir\n");
        scanf("%d",&eleccion);
}while(eleccion<1||eleccion>5);
putchar('\n');
return(eleccion);
}
void *crear(void *p){
nodo *q,*aux,*aux1;
int x;
putchar('\n');
printf("Indica el valor a introducir a la lista=>");
scanf("%d",&x);
q=(nodo*)malloc(sizeof(nodo));
q->dato=x;
q->izq=NULL;
q->der=NULL;
if(p==NULL)   //La lista esta vacia
        p=q;
else{
        aux1=aux=(nodo*)p;
        if (x<aux->dato){      //el dato es menor a todos los datos localizados en la lista
                q->der=aux;
                aux->izq=q;
                p=q;
        }
        else{
                while(aux!=NULL && aux->dato<x){
                        aux1=aux;
                        aux=aux->der;
                }
```

```
                aux1->der=q;
                q->izq=aux1;
                if (aux!=NULL){  //El dato se colocara entre la lista
                        q->der=aux;
                        aux->izq=q;
                }
        }
}
return(p);
}
void * eliminar(void * s){
if(s==NULL)
        printf("\nLISTA VACIA\n");
else{
        nodo *p,* aux,*aux1;
        int x;
        printf("Da el elemento a eliminar=>");
        scanf("%d",&x);
        aux=p=(nodo*)s;
        if(p->der==NULL && aux->dato==x)          //solo hay un elemento en la lista
                s=NULL;
        else {
                while(p->der!=NULL && p->dato<x){
                        aux=p;
                        p=p->der;
                }
                if (p!=NULL){
                        if (p->dato==x && p==s){ //No existio movimiento
                                s=aux1=p->der;
                                if(aux1!=NULL)
                                        aux1->izq=NULL;
                        }
                        else if(p->dato==x){
                                aux1=aux->der=p->der;
                                if (aux1!=NULL)
                                        aux1->izq=aux;
                        }
                        else
                                printf("DATO NO LOCALIZADO\n");
                }
                else
                        printf("DATO NO ENCOTRADO\n");
```

```
        }
        free(p);
}
return(s);
}
void mostrarasc(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf("LISTA VACIA");
else
        do{
                printf("%d..",s->dato);
                s=s->der;
        }while(s!=NULL);
}

void mostrardesc(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
        printf("LISTA VACIA");
else{
        while(s->der!=NULL)
                s=s->der;
        do{
                printf("%d..",s->dato);
                s=s->izq;
        }while(s!=NULL);
}
}
```

Enseguida se muestra la ejecución del programa donde se introducen los números 3, 1 y 2. Posteriormente se muestra la impresión de los elementos en forma ascendente y en forma descendente.

```
                MENU PRINCIPAL
        1.-Introducir un elemento a la lista
        2.-Eliminar un elemento de la lista
        3.-Mostrar el contenido de la lista en forma ascendente
        4.-Mostrar el contenido de la lista en forma descendente
        5.-Salir
1

Indica el valor a introducir a la lista=>3

                MENU PRINCIPAL
        1.-Introducir un elemento a la lista
        2.-Eliminar un elemento de la lista
        3.-Mostrar el contenido de la lista en forma ascendente
        4.-Mostrar el contenido de la lista en forma descendente
        5.-Salir
1

Indica el valor a introducir a la lista=>1

                MENU PRINCIPAL
        1.-Introducir un elemento a la lista
        2.-Eliminar un elemento de la lista
        3.-Mostrar el contenido de la lista en forma ascendente
        4.-Mostrar el contenido de la lista en forma descendente
        5.-Salir
1

Indica el valor a introducir a la lista=>2

                MENU PRINCIPAL
        1.-Introducir un elemento a la lista
        2.-Eliminar un elemento de la lista
        3.-Mostrar el contenido de la lista en forma ascendente
        4.-Mostrar el contenido de la lista en forma descendente
        5.-Salir
3

1..2..3..
                MENU PRINCIPAL
        1.-Introducir un elemento a la lista
        2.-Eliminar un elemento de la lista
        3.-Mostrar el contenido de la lista en forma ascendente
        4.-Mostrar el contenido de la lista en forma descendente
        5.-Salir
4

3..2..1..
```

Programa 5.5

5.2.4 PROGRAMACIÓN DE UN ÁRBOL BINARIO DE BÚSQUEDA

Los árboles representan las estructuras no lineales y dinámicas más importantes en cómputo. Un árbol binario es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No puede tener más de dos hijos. Si algún hijo tiene como referencia a NULL, es decir que no almacena ningún dato, entonces este es llamado un nodo externo o nodo hoja. En el caso contrario, el hijo es llamado un nodo interno.

En teoría de grafos, se usa la siguiente definición: “Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 3”. De esta forma solo existe un camino entre un par de nodos.

Un árbol binario con raíz es un grafo que tiene uno de sus vértices, llamado raíz, de grado no mayor a 2. Con la raíz escogida, cada vértice tendrá un único padre, y nunca más de dos hijos.

Un árbol binario tiene múltiples aplicaciones. Se los puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos de un proceso, para representar la historia de un campeonato de tenis, para representar un árbol genealógico y para representar expresiones algebraicas construidas con operadores binarios. Esto solo para citar algunos de sus múltiples usos.

El árbol binario de búsqueda es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación.

Un árbol binario de búsqueda se define de la siguiente forma: “Para todo nodo T del árbol debe cumplirse que todos los valores de los nodos del subárbol izquierdo de T deben ser menores o iguales al valor del nodo T. En forma similar, todos los nodos del subárbol derecho de T deben ser mayores o iguales al valor del nodo T”.

LA INSERCIÓN DE UN ÁRBOL BINARIO DE BÚSQUEDA

La inserción es una operación que se puede realizar eficientemente en un árbol binario de búsqueda. Los pasos a realizarse para insertar un elemento en un árbol binario de búsqueda son:

- Debe compararse la clave a insertar con la raíz del árbol. Si es mayor, debe avanzarse hacia el subárbol derecho. Si es menor, debe avanzarse hacia el subárbol izquierdo.
- Repetir el paso 1 hasta que se cumpla alguna de las siguientes condiciones:
- El subárbol derecho es igual a vacío, o el subárbol izquierdo es igual a vacío, en cuyo caso se procederá a insertar el elemento en el lugar que le corresponde.
- La clave que quiere insertarse es igual a la raíz del árbol; en cuyo caso no se realiza la inserción.

EL BORRADO EN UN ÁRBOL BINARIO DE BÚSQUEDA

La operación de borrado es un poco más complicada que la de inserción. Esta consiste en eliminar un nodo del árbol sin violar los principios que definen justamente a un árbol binario de búsqueda. Se deben distinguir los siguientes casos:

- Si el elemento a borrar es terminal u hoja, simplemente se suprime y su nodo padre debe de apuntar a NULL.
- Si el elemento a borrar tiene un solo descendiente, entonces tiene que sustituirse ese descendiente.

- Si el elemento a borrar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Además, debemos recordar que antes de eliminar un nodo, debemos localizarlo en el árbol. Para esto, se utilizará el algoritmo de búsqueda presentado anteriormente.

RECORRIDO EN UN ÁRBOL BINARIO

Una de las operaciones más importantes a realizar en un árbol binario es el recorrido de los mismos, recorrer significa visitar los nodos del árbol en forma sistemática, de tal manera que todos los nodos del mismo sean visitados en una sola vez.

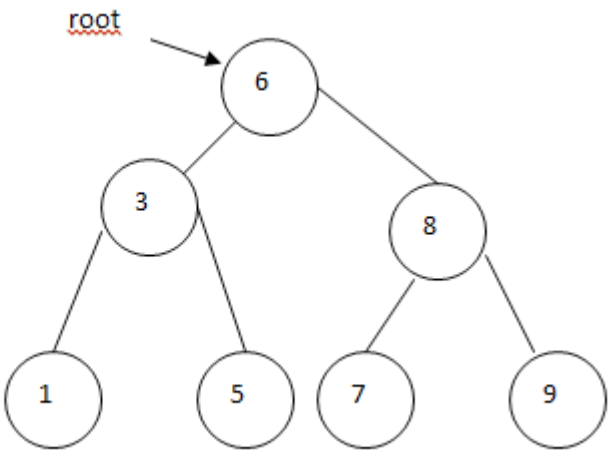
Existen 3 formas diferentes de efectuar el recorrido y todas ellas de naturaleza recursiva, las cuales son:

1. **Recorrido en preorden:** en el que se procesa el nodo y después se procesan recursivamente sus hijos.

2. **Recorrido en postorden:** donde el nodo dado se procesa después de haber procesado recursivamente a sus hijos.

3. Recorrido en inorden: en este se procesa recursivamente el hijo izquierdo, luego se procesa el nodo actual y finalmente se procesa recursivamente el hijo derecho.

Ejemplificando con el siguiente árbol se tiene:



Recorrido en preorden:
6-3-1-5-8-7-9

Recorrido en postorden:
1-5-3-7-9-8-6

Recorrido en inorden:
1-3-5-6-7-8-9

La programación de un árbol binario de búsqueda se muestra en el programa 5.6:

```
#include<stdio.h>
#include <stdlib.h>
struct nodo{
        int dato;
        nodo * izq;
        nodo * der;
};

//´PROTOTIPOS
void * crear (void *);
void mostrar(void *);
void * eliminar(void *);
void preorden(void *);
void postorden(void *);
void inorden(void *);
int menu();

main(){
void * root=NULL;
int x;
do{
        x=menu();
        switch(x){
        case 1:root=crear(root);break;
        case 2:mostrar(root);break;
        case 3:root=eliminar(root);break;
        }
}while (x>0 && x<4);
return 0;
}
```

```
void mostrar(void *r){
int x;
do{
        printf(«\n\tSUBMENU MOSTRAR\n»);
        printf(«1. MOSTRAR EL ARBOL EN PREORDEN\n»);
        printf(«2. MOSTRAR EL ARBOL EN POSTORDEN\n»);
        printf(«3. MOSTRAR EL ARBOL EN INORDEN\n»);
        printf(«4. SALIR\n»);
        printf(«Da la opcion->»);
        scanf("%d",&x);
}while(x>4 || x < 1);

if (r==NULL)
        printf("ARBOL VACIO\n");
else
        switch(x){
                case 1:preorden(r);break;
                case 2:postorden(r);break;
                case 3:inorden(r);break;
        }
}

int menu(){
int x;
do{
        printf(«\n\tMENU\n»);
        printf("1. INSERTAR UN ELEMENTO\n");
        printf("2. MOSTRAR EL ARBOL\n");
        printf("3. ELIMINAR UN ELEMENTO DEL ARBOL\n");
        printf("4. SALIR\n");
        printf("Da la opcion->");
        scanf("%d",&x);
}while(x>4 || x < 1);
return(x);
}

void * eliminar(void * r){
/*Se emplea una variable del tipo bandera para:
        Si band=1,  no se localizo el dato en el arbol
        Si band =2, se localizo el dato en el arbol
*/
if (r==NULL)
```

```
        printf("ARBOL VACIO\n");
else{
        printf("DA EL VALOR DEL DATO A ELIMINAR=>");
        int x,band=0;
        scanf("%d",&x);
        nodo *aux,*q=(nodo *)r;
        aux=NULL;
        do
                if(q->dato<x)
                        if(q->der!=NULL){
                                aux=q;
                                q=q->der;
                        }
                        else
                                band=1;
                else if(q->dato>x)
                        if(q->izq!=NULL){
                                aux=q;
                                q=q->izq;
                        }
                        else
                                band=1;
                else band=2;
        while(band==0);
        if(band==2){
                printf("DATO A ELIMINAR=>%d\n",q->dato);
                if(aux!=NULL)
                       printf("VALOR DEL DATO EN EL AUXILIAR=>%d\n",aux->dato);
                else
                        printf("DATO LOCALIZADO EN LA RAIZ\n");
                nodo * d;
                if (q->izq==NULL && q->der==NULL){
                        printf("EL DATO SE LOCALIZO EN UN NODO HOJA\n");
                        if (r==q){
                                printf("Y ES NODO RAIZ UNICO\n");
                                r=NULL;
                        }
                        else if(aux->izq==q)
                                aux->izq=NULL;
                        else if(aux->der==q)
                                aux->der=NULL;
                         d=q;
```

```
}
else if (q->izq==NULL && q->der!=NULL){
// El dato a eliminar solo tiene ramificacion derecha
        d=q;
        q=q->der;
        if (r==d)
                r=d->der;
        else if(aux->der==d)
                aux->der=d->der;
        else
                aux->izq=d->der;
}
else if (q->izq!=NULL && q->der==NULL){
// El dato a eliminar solo tiene ramificacion izquierda
        d=q;
        q=q->izq;
        if(r==d)
                r=d->izq;
        else if(aux->der==d)
                aux->der=d->izq;
        else
                aux->izq=d->izq;
}
else{
// El dato a eliminar tiene las dos ramificaciones
        d=q->izq;
        if(d->der==NULL){
                q->dato=d->dato;
                q->izq=d->izq;
        }
        else{
                do{
                        aux=d;
                        d=d->der;
                }while(d->der!=NULL);
                if (d->izq!=NULL)
                        aux->der=d->izq;
                else
                        aux->der=NULL;
                q->dato=d->dato;
        }
}
```

```
                free(d);
        }//if (q->dato==x)
        else
                printf("DATO NO LOCALIZADO\n");
}//else de arbol no vacio
return(r);
}

void * crear(void * p){
/*Se utiliza una variable tipo bandera para:
        Si band=1, el valor a insertarse esté repetido
        Si band=2, Se llegó a una hoja donde se insertará el nuevo elemento
        Si band=3, el árbol está vacio
*/
int x,band=0;
nodo * q, * aux;
printf(«Da un valor entero=>»);
scanf(«%d»,&x);
aux=(nodo *)p;
if (aux==NULL)
        band=3;
else
        do
                if (aux->dato==x)
                        band=1;//valor repetido
                else if (aux->dato>x){
                        if(aux->izq!=NULL)
                                aux=aux->izq;
                        else
                                band=2;
                }
                else{
                        if(aux->der!=NULL)
                                aux=aux->der;
                        else
                                band=2;
                }
        while (band<1);
if (band>1){
                q=(nodo*)malloc(sizeof(nodo));
                q->der=q->izq=NULL;
                q->dato=x;
```

```
                if (band==3)
                        p=q;
                else if (aux->dato<x)
                        aux->der=q;
                else
                        aux->izq=q;
        }
else
        printf(«VALOR REPETIDO\n»);

return(p);
}

void preorden(void * r){
        nodo *w;
        w=(nodo*)r;
        if (r!=NULL){
                printf("%d..",w->dato);
                preorden(w->izq);
                preorden(w->der);
        }
}
void postorden(void * r){
        nodo *w;
        w=(nodo*)r;
        if (r!=NULL){
                postorden(w->izq);
                postorden(w->der);
                printf("%d...",w->dato);
        }
}
void inorden(void * r){
        nodo *w;
        w=(nodo*)r;
        if(w!=NULL){
                inorden(w->izq);
                printf("%d…",w->dato);
                inorden(w->der);

        }
}
```

Programa 5.6

5.2.5 ARREGLO DE APUNTADORES

El programa 5.7 simula una agenda telefónica donde se guarda el nombre en forma alfabética, dirección y teléfono de personas. Cada nombre se guarda en un arreglo de caracteres cuyo tamaño se determina en forma dinámica y es guardado en una lista simplemente enlazada en orden alfabético, por lo que existirá un apuntador para cada letra del alfabeto (se crea un arreglo de 26 apuntadores tipo void, y cada apuntador apuntará a una lista simplemente enlazada donde contendrá nombres ordenados de personas que inicien con la misma letra). El menú permite introducir los nombres con su dirección y teléfono y se guardan en forma alfabética, muestra todos los nombres ordenados en forma alfabética o los nombres que inician con una letra en particular, o elimina un nombre en particular junto con el número telefónico y dirección.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct agenda{
      char * nombre;
      char * direccion;
      char telefono[11];
      struct agenda * siguiente;
};
struct alfabeto{
      struct agenda * orden;
};
//PROTOTIPOS
int menu();
void crear(alfabeto []);
void mostrar(alfabeto []);

main(){
      int i,x;
      struct alfabeto amigos[26];
      for (i=0;i<26;i++)
         amigos[i].orden=NULL;
     do{
        x=menu();
        switch(x){
                case 1: crear(amigos); break;
                case 2: mostrar(amigos); break;
        }
     }while(x<4);
}
```

```
void crear(alfabeto cuates[]){
    int i,j;
    char n[40];
    struct agenda * q, * aux, * aux1;
    q=(agenda *)malloc(sizeof(agenda));
    q->siguiente=NULL;
    do{
      printf("Dar el nombre con mayusculas:");
      for (i=0;(n[i]=getchar())!='\n';i++);
      n[i]='\0';
    }while(n[0]<'A' || n[0]>'Z');
    q->nombre=(char *)malloc(sizeof(i));
    for (j=0;j<=i;j++)
       q->nombre[i]=n[i];
    do{
      printf("Dar la direccion:");
      for (i=0;(n[i]=getchar())!='\n';i++);
      n[i]='\0';
    }while(n[0]<'A' || n[0]>'Z');
    q->direccion=(char *)malloc(sizeof(i));
    for (j=0;j<=i;j++)
       q->direccion[i]=n[i];

    printf("Da el telefono con 10 digitos");
    gets(q->telefono);
    i=q->nombre[0]-'A';
    if (cuates[i].orden->siguiente==NULL)
      cuates[i].orden->siguiente=q;
    else{
              aux= cuates[i].orden;
              int x;
              x=strcmp(q->nombre, aux->nombre);
              if (x<0){
                       q->siguiente=aux;
                       cuates[i].orden->siguiente=q;
              }
              else{
                        while (aux!=NULL && strcmp(q->nombre, aux->nombre) >0 ){
                   aux1=aux;
                                 aux=aux->siguiente;
          }
                        aux1->siguiente=q;
```

```
                    q->siguiente=aux;
            }
             }
}

void mostrar (alfabeto cuates[]){
    char x;
    agenda * aux;
    do{
      printf("Da la letra del alfabeto que quieres mostrar:");
      scanf("%c",&x);
    }while(x<'A' || x>'Z');
     aux= cuates[x-'A'].orden;
     if (aux->siguiente==NULL)
        printf("No existen nombres que inicien con la letra %c.\n",x);
     else{
          while (aux->siguiente!=NULL){
               printf("%s\t%s\t%s\n",aux->nombre,aux->direccion,aux->telefono);
               aux=aux->siguiente;
               getchar();
               }
          }
    }
int menu(){
    int x;
    do{
       printf("AGENDA DE CUATES\n");
       printf("1. INTRUDUCIR AMIGO\n");
       printf("2. MOSTRAR SERIE DE AMIGOS CON EL MISMO APELLIDO PATERNO\n");
       printf("3. ELIMINAR DE LA AGENDA A UN ELEMENTO\n");
       printf("4. Salir de la aplicacion\n");
       printf("INDIQUE SU ELECCION:");
       scanf("%d",&x);
    }while (x<1 || x>4);
    return x;
}
```

Programa 5.7

5.3 APUNTADORES A FUNCIONES

En el programa 5.8 se ordena un vector de enteros, haciendo uso de una función de comparación y una para cambiar los datos. La función de ordenamiento recibe un apuntador a una función.

```
#include <stdio.h>
#include <conio.h>
 // Función para comparar dos valores. Retorna 1 o -1
int Comparacion(int valor1,int valor2){
    return valor1 > valor2 ? 1 : -1;
}
// Función para intercambiar el valor de dos posiciones en un vector
void CambiarDatos(int posI,int posJ,int *Vector){
     int aux=Vector[posI];
     Vector[posI]=Vector[posJ];
     Vector[posJ]=aux;
}
// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (* max  )(int , int), int *Vector,int maximo){
     for (int i=0;i<maximo-1;i++){
          for (int j=i+1;j<maximo;j++){
               if ( max (Vector[i],Vector[j])>0)
                    CambiarDatos(i,j,Vector);
          }
     }
}
// Función para imprimir un vector
void Imprime_Vector(int *Vector,int  maximo){
     for (int i=0;i< maximo;i++)
          printf("%2d\n",Vector[i]);
}
main(){
       int Vector[10]={5,76,4,8,50,2,44,7,23,6};
       printf("Original: \n");
              Imprime_Vector(Vector,10);
              Ordenamiento(Comparacion,Vector,10);
              printf("Ordenado: \n");
              Imprime_Vector(Vector,10);
              getchar();
}
```

Programa 5.8

Se reescribe el mismo ordenamiento utilizando la variable Vector como un apuntador y se utiliza aritmética de apuntadores para desplazarse en las posiciones en la memoria, (ver programa 5.9):

```
#include <stdio.h>
#include <conio.h>
// Función para comparar dos valores. Retorna 1 o -1
int Comparacion(int *valor1, int *valor2){
        return *valor1 > *valor2 ? 1 : -1;
}
// Función para intercambiar dos valores
void CambiarDatos(int *datoI, int *datoJ){
        int aux=*datoI;
        *datoI=*datoJ;
        *datoJ=aux;
}
// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (* fun )(int * , int *), int *Vector,int maximo){
        int *vi,*vj;
        vi =Vector;
        for (int i=0; i<maximo-1; i++){
            vj = vi;
            vj++;
            for (int j=i+1; j<maximo; j++){
         if ( fun(vi,vj) > 0)
                    CambiarDatos(vi,vj);
                vj++;
            }
            vi++;
        }
}

// Función para imprimir un vector
void Imprime_Vector(int *Vector,int maximo){
        int *vi;
        vi = Vector;
        for (int i=0;i<maximo;i++)
        {
            printf("%2d\n",*vi);
            vi++;
        }
    }
```

```
int main(){
        int Vector[10]={5,76,4,8,50,2,44,7,23,6};
        printf("Original: \n");
        Imprime_Vector(Vector,10);
        Ordenamiento(Comparacion,Vector,10);
        printf("Ordenado: \n");
        Imprime_Vector(Vector,10);
        getchar();
    }
```

Programa 5.9

5.4 MATRICES DE APUNTADORES A FUNCIÓN

Antes de continuar, se puede comentar que una matriz permite guardar datos del mismo tipo, por lo que se puede utilizar una matriz para guardar apuntadores a función.

Por ejemplo:
Primero definamos un apuntador a función que recibe dos parámetros del tipo entero y retorna un entero:

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

A partir de la definición anterior, se definirá una matriz de cinco apuntadores a función:

```
int (*ptrFn[5]) (int,int);
```

Ahora, observe el programa 5.10 donde solicita información de cinco personas y las ordena utilizando el método de la burbuja:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct StPersona{
        int edad;
        int Salario;
        char *Nombre;
};

// Función para comparar por edad. Retorna 1 o -1
int ComparacionxEdad(StPersona  *Persona1, StPersona *Persona2){
        return Persona1->edad > Persona2->edad ? 1 : -1;
}
```

```
// Función para comparar por salario. Retorna 1 o -1
int ComparacionxSalario(StPersona  *Persona1, StPersona *Persona2){
        return Persona1->Salario > Persona2->Salario ? 1 : -1;
}

// Función para comparar por nombre. Retorna 1 o -1
int ComparacionxNombre(StPersona  *Persona1, StPersona *Persona2){
   return strcmp(Persona1->Nombre, Persona2->Nombre);
}

// Función para intercambiar dos valores

void CambiarDatos(StPersona *datoI, StPersona *datoJ){
        StPersona aux = *datoI;
        *datoI = *datoJ;
        *datoJ = aux;
}

// Función para ordenar, recibe un puntero a una función
void Ordenamiento (int (* fun)(StPersona * , StPersona *), StPersona *Vector,int  nti  ){
        StPersona *vi,*vj;
        vi = Vector;
                for ( int i =0; i < nti-1; i++){
                vj=vi;
                vj++;
                for (int j=i+1;j<nti;j++){
                      if ( fun(vi,vj)>0)
                         CambiarDatos(vi,vj);
                vj++;
               }
                 vi++;
         }
}

// Función para imprimir un vector
void Imprime_Vector(StPersona *Vector,int  nti  ){
        StPersona *vi;
        vi = Vector;
        for ( int i =0; i<nti; i++){
              printf("Edad : %d\n", vi->edad );
              printf("Nombre : %s\n", vi->Nombre);
              printf("Salario : %d\n", vi->Salario );
```

```
                vi++;
            }
}

//Función que permite leer un nombre
char * nombre(){
    char * nomb;
    char a[50];
     int i ;
    for (i=0;(a[i]=getchar())!='\n';i++);
    a[i]='\0';
    nomb=(char *)malloc(sizeof(char)*i);
    strcpy(nomb,a);
    return nomb;
}
int main(){
      char *b[3], a[80];
      b[0]="edad";
      b[1]="salario";
      b[2]="nombre";
            StPersona *Vector;
            Vector=new StPersona[5];
             int i ;
            for (i=0;i<5;i++){
                  printf ("Edad : ");
                  scanf("%d", &Vector[i].edad);
                  printf ("Salario : ");
                  scanf("%d", &Vector[i].Salario);
                  getchar();
                  printf ("Nombre : ");
                  Vector[i].Nombre=nombre();
            }
            int (*ptrFn[3]) (StPersona *, StPersona *); // Matriz de 3 punteros a funciones
            printf("Original: \n");
            Imprime_Vector(Vector,5);
            ptrFn[0] = ComparacionxEdad;
            ptrFn[1] = ComparacionxSalario;
            ptrFn[2] = ComparacionxNombre;

//Invocacion a cada uno de los ordenamientos
printf("\n ****************************************** \n");
            for (i=0;i<3;i++){
```

```
            Ordenamiento(ptrFn[i], Vector,5);
            printf("\nOrdenado por   %s: \n", b[i]);
            Imprime_Vector(Vector,5);
        }
        getchar();
}
```

Programa 5.10

En el código se han escrito 3 funciones de ordenamiento, los cuales son invocados en el último "for". Observe que en la función -Ordenamiento()- se envía como parámetro un elemento de una matriz (ptrFn[]) que es un apuntador de función.

5.5 APUNTADORES A FUNCIONES COMO PARÁMETROS DE FUNCIONES DE LA BIBLIOTECA ESTÁNDAR EN C

La función de biblioteca *–qsort()-* localizado en -stdlib- es muy útil y está diseñada para ordenar un arreglo usando un valor como llave de cualquier tipo para ordenar en forma ascendente.

El prototipo de la función *–qsort()-* es:

*void qsort(void *base, size_t nmiemb, size_t tam, int (*comparar)(const void *, const void*));*

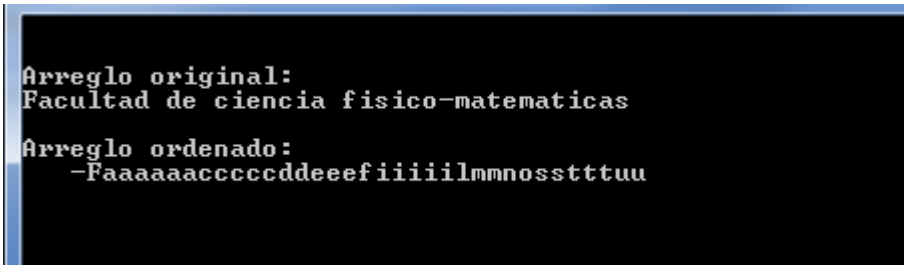
El argument *–base-* apunta al comienzo del vector que será ordenado, *-nmiemb-* indica el tamaño del arreglo, *-tam-* es el tamaño en bytes de cada elemento del arreglo y el argumento final *–comparar-* es un apuntador a una función.

La función *–comparar-* del programa 5.11 debe regresar un determinado valor entero de acuerdo al resultado de la comparación que debe ser:

- Menor que cero: Si el primer valor es menor que el segundo.
- Cero: Si el primer valor es igual que el segundo.
- Mayor que cero: Si el primer valor es mayor que el segundo.

Observe que en la función *–comparar()-* la define el usuario y en el momento de la invocación se envía el nombre sin parámetros (en la misma forma en que se envía como parámetro una matriz). La función *–comparar-* tiene como parámetros a dos constantes del tipo apuntador tipo void para que la función *–qsort-* se le permita manipular a cualquier tipo de variable.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int comp(const void *i, const void *j);
main(){
int i;
char cad[]=”Facultad de Ciencias fisico-matematicas”;
printf(“\n\nArreglo original:\n”);
for (i=0;i<strlen(cad);i++)
        printf(“\%c”,cad[i]);
qsort(cad,strlen(cad),sizeof(char),comp);
printf(“\n\nArreglo ordenado:\n”);
for (i=0;i<strlen(cad); i++)
        printf(“%c”, cad[i]);
printf(“%c”);
getchar();
}

int comp(const void *i, const void *j){
char *a, *b;
a=(char *)i;
b=(char *)j;
return *a - *b;
}
```



Programa 5.11

Observe que dentro de la función –comp()- se realiza un –cast- para forzar el tipo –void *- al tipo –char *-.

ANEXO

UN EJEMPLO DE MÉTODOS NUMÉRICOS

En este anexo se muestra un programa de Métodos Numéricos utilizando memoria dinámica, y con el cual se calculan: a) las raíces de una ecuación polinomial, b) solución de ecuaciones lineales, c) el determinante, d) ecuaciones, y valores característicos.

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

//PROTOTIPOS PARA ECUACIONES POLINOMIALES
void newton(void);
void solucion(float*, int);
void imprime_coeficientes(float*, int);
float *leen(float*, int);
void escriben(float *, int);
void descartes(float*, int);
void grafica(int, int, int);
int *genera_factores(float, int*, int*);
void imprime_factores(int*, int);
float divide(float*, float, int);
float doblediv(float*, float, int);
void r_enteras(float*, int*, int, int);
void r_reales(float*, int*, int, int);
//PROTOTIPOS PARA ECUACIONES LINEALES
int gauss(float**, int, int, int*);
float **lee(float**, int, int*);
void imprime(float**, int, int);
void resultado(float **, int, int);
int menu(void);
void determinante(void);
void sistecuaciones(void);
//PROTOTIPOS PARA ECUACIONES Y VALORES CARACTERÍSTICOS
void caracteristico(void);
void resultado1(float *, int);
void multiplica(float **, float *, int);
void acomoda(float **, float *, int, int);
```

```
int main() {
    setbuf(stdout, NULL);
    //se defne un valor para solicitar opcion del menu
    int x;
    //realiza loop mientras no se obtenga un valor valido para el menu
    do {
        x = menu();
        getchar();
    } while (x < 5);
    //sale de main
    return 0;
    return (EXIT_SUCCESS);
}

int menu() {
    int x;
    //muestra menu
    printf("        MENU\n");
    printf("1.- Solución de polinomios.\n");
    printf("2.- Solución de ecuaciones lineales.\n");
    printf("3.- Cálculo del determinante.\n");
    printf("4.- Ecuación característica, valores característicos.\n»);
    printf(«5.- Salir del sistema.\n»);
    printf(«Seleccione su opción=>\n»);
    //lee valor de entrada para opcion del menu
    scanf("%d", &x);
    switch (x) {
        case 1:newton();
            break;
        case 2:sistecuaciones();
            break;
        case 3:determinante();
            break;
        case 4:característico();
    }
    return (x);
}
//FUNCIONES PARA ECUACIONES POLINOMIALES

void newton() {
    //define variables a utilizar en el programa
```

```
    //p -> arreglo donde se almacerá los coeficientes del polinomio
    //max -> constante que define el tamaño maximo del arreglo
    float *polinomio = NULL;

    int grado_del_polinomio;
    printf("DA EL GRADO DEL POLINOMIO=>");
    scanf("%d", &grado_del_polinomio);
    polinomio = leen(polinomio, grado_del_polinomio);
   //se envia el arreglo polinomio y el grado para que se procesen y se genere una solucion
    solucion(polinomio, grado_del_polinomio);
}

void solucion(float *polinomio, int grado_del_polinomio) {
    //variables
    //nun_fact -> numero de factores
    //fact -> es el arreglo de los fatores que se generan¡
    printf(«EL POLINOMIO QUE ME DISTE ES:\n»);
    //imprimira¡ el polimonio
    imprime_coeficientes(polinomio, grado_del_polinomio);
    //realizara el analisis de la cantida de raices positivas, negativas o complejas existentes
    descartes(polinomio, grado_del_polinomio);
    //se obtendran la cantidad de factores
    //envia el último elemento del polinomio
    int *factores = NULL;
    int num_factores = 0;
     factores = genera_factores(*(polinomio + grado_del_polinomio), factores, &num_facto-
res);
    //imprime los fatores encontrados
    imprime_factores(factores, num_factores);
    //evaluara las raices, para encontrar aquellas que son enteras
    r_enteras(polinomio, factores, grado_del_polinomio, num_factores);
    //evaluara las raices para encontrar las que son reales
    r_reales(polinomio, factores, grado_del_polinomio, num_factores);
    getchar();
}

void r_reales(float *polinomio, int *factores, int grado_del_polinomio, int num_factores) {
    //j -> servira para iterar el arreglo de factores
    //bandera -> servira para mostrar el titulo solo la primera vez
    int j = 0, bandera;
    //a -> almacenara el valor de f(x)
    //e -> almacenará el error
```

```
   //a1 -> almacenará el valor de f(x1)
   //error -> diferencia entre la raiz en la iteracion i y la anterior
   //error1 -> almacena el error de la iteracion anterior
   float a, e, a1, x, x1, error, error1;
   printf("\n********** RAICES REALES **********\n");
   printf("Da el error a aceptar:");
   scanf("%f", &e);
   printf("          x          x1          f(x)          f(x1)\n");
   //itera por todos los factores
   do {
//define el valor del error tan grande que la primera vez pueda comparar
//error1 = 100000;
      //servira para identificar si error dismunuye
      bandera = 0;
      //amacena el factor en una variable
      x = *(factores + j);
      //almacena el factor siguiente en otra variable
      x1 = *(factores + j + 1);
      //obtiene los valores de a, por medio de division sintetica
      a = divide(polinomio, x, grado_del_polinomio);
      a1 = divide(polinomio, x1, grado_del_polinomio);
      printf("  %13G  %13G  %13G  %13G\n", x, x1, a, a1);
      //si los valores indican cambio de signo, hay raiz en ese intevalo
      if (a * a1 < 0) {
         //evalua punto medio de las raices
         x = (x + x1) / 2;
         //aqui empieza el metodo de newton
         do {
            //realiza la formula de newton
            x1 = x - dobledIv(polinomio, x, grado_del_polinomio);
            //calcula la diferencia entre la raiz calculada y la anterior
            error = x - x1;
            //realiza el valor absoluto
            if (error < 0)
               error = -error;
            //si el error calculado es mayor que el anterior
            //hay divergencia
            if (error > error1)
               bandera = 1;
            //se almacenan los valores obtenidos en lugar de las actuales
            error1 = error;
            x = x1;
```

```
            //en caso de haber haber divergencia
            if (bandera == 1)
                printf("EL METODO DIVERGE EN EL INTERVALO\n");
            //continua iterando en caso de que el error disminuya y no se levante la bandera
        } while (error > e && bandera == 0);
        printf("RAIZ=%f\n", x1);
    }
    j++;
} while (j < num_factores - 1);
}

void r_enteras(float *polinomio, int *factores, int grado_del_polimonio, int num_factores) {
    //variables
    //bandera -> usada en caso de encontrar la primera raiz entera
    //j -> servira para iterar en el arreglo de los fatores
    int bandera = 0, j = 0;
    //a ->
    //b -> almacena cada factor, comenzando por el ultimo factor
    float a, b;
    //recorre el arreglo de factores
    do {
        b = *(factores + j);
        //realiza la divison para identificar las raices
        a = divide(polinomio, b, grado_del_polimonio);
        //si el resultado de la division sintetica es 0, definimos que es una raiz
        if (a == 0) {
            //se ejecuta solo la primera vez
            if (bandera == 0) {
                printf("\n*** RAICES ENTERAS ***\n");
                bandera = 1;
            }
            //imprime la raiz
            printf("RAIZ ENTERA:%d\n", *(factores + j));
        }
        j++;
    } while (j <= num_factores);
}

float divide(float *p, float x, int n) {
    //a -> es inicializado con el primer
    //x -> es el factor que se evalua en la division
    //n -> grado
```

```
    float a = *(p);
    int i;
    for (i = 1; i <= n; i++)
        //a ira acumulando los resultados de la division sintetica
        a = a * x + *(p + i);
    //a -> f(x)
    return (a);
}

float doblediv(float *p, float f, int n) {
    //c -> es f'(x)
    //a -> f(x)
    //b -> es auxiliar, donde se va almacenando f'(x)
    //i -> servira para iterar
    float c, a = *p, b = *p;
    int i;
    for (i = 1; i <= n; i++) {
        a = a * f + *(p + i);
        b = b * f + a;
        if (i == n - 1)
            c = b;
    }
    //regresa f(x)/f'(x)
    return (a / c);
}

void imprime_factores(int *fac, int n) {
    printf("Los factores posibles son:\n");
    int i;
    for (i = 0; i < n; i++)
        printf("%d..", *(fac + i));
}

int *genera_factores(float f, int *factores, int *num_fatores) {
    //f -> el ultimo coeficiente del polimonio
    //factor -> es el arreglo donde se almacenaran los factores que se generan
    //si el valor del último coeficiente es menor a 0, cambio de signo
    if (f < 0)
        f = -f;
    //convierte el último coeficiente a un entero
    int i = (int) f;
    //cont -> contendra la cantidad de factores
```

```
    //j -> servira para mantener el valor inicial de i
    int cont, j = i;
    printf("********** FACTORES *******\n");
    cont = 0;
    printf("El número a factorar es:%d\n", j);
    //mientras i sea mayor que 1, generando factores
    while (i >= 1) {
        //obtiene el modulo, para saber si i es un factor de j
        if (j % i == 0) {
            //guarda el valor de i(factor encontrado) en el arreglo
            factores = (int *) realloc((int *) factores, ((cont + 1) * 2) * sizeof (int));
            factores[cont] = i;
            cont++;
        }
        i--;
    }
    //almacenar los factores dentro del arreglo como negativos
    //k -> servira para iniciar en la última posicion de los factores generados
    int k = cont - 1;
    for (i = cont; i < 2 * cont; i++) {
        *(factores + i) = -(*(factores + k));
        k--;
    }
    printf("El número de factores es:%d\n", i);
    //regresa la cantidad de factores encontrados por 2
    *num_fatores = 2 * cont;
    return factores;
}

void descartes(float *p, int n) {
    //variables
    //i -> itera en el arreglo
    //cont -> servirá para contar los cambios de signo entre los coefientes
    int i, cont = 0;
    //pmenos -> copia del polimnomio donde se iran cambiando de signo
    //simulando el cambio de x -> -x
    float a[n], *pmenos;
    pmenos = a;
    //identifica si el grado es par o impar
    if (n % 2 == 0)
        //si es par, el contador iniciará en 1, para garantizar que comienza en potencia impar
        cont = 1;
```

```
    //realiza una copia de p en pmenos
    for (i = 0; i <= n; i++)
        *(pmenos + i) = *(p + i);
    //cambiar de signo los coeficientes de las potencias impares
    do {
        *(pmenos + cont) = -(*(p + cont));
        cont += 2;
    } while (cont < n);

    printf("\nEL POLINOMIO NEGATIVO ES:\n");
    //imprime los nuevos valores de los coeficientes
    imprime_coeficientes(pmenos, n);
    //pos -> cantidad de raices positivas
    //neg -> cantidad de raices negativas
    int pos = 0, neg = 0;
    //recorre ambos arreglos, para identificar la catidad de cambios de signo
    for (i = 0; i < n; i++) {
        //si la multiplicacion de los elementos resulta con un valor menor que 0
        //se asume que uno es positivo y otro negativo, por lo tanto, hay cambio de signo
        if ((*(p + i)) * (*(p + i + 1)) < 0)
            pos++;
        //si la multiplicacion de los elementos resulta con un valor menor que 0
        //se asume que uno es positivo y otro negativo, por lo tanto, hay cambio de signo
        if ((*(pmenos + i)) * (*(pmenos + i + 1)) < 0)
            neg++;
    }
    //imprime la cantidad de cambios de signo
    grafica(pos, neg, n);
}

void grafica(int p, int n, int grado) {
    //variables
    //comp1 -> cantidad de raices complejas para el primer caso
    //comp2 -> cantidad de raices complejas para el segundo caso
    //p2 -> positivos para el caso 2
    //n2 -> negativos para el caso 2
    int comp1 = 0, comp2 = 0, p2 = p, n2 = n;
    //si positivos es mayor a 2, disminuye 2, para el segundo caso
    if (p > 2)
        p2 = p - 2;
    //si negativos es mayor a 2, disminuye 2, para el segundo caso
    if (n > 2)
```

```
        n2 = n - 2;
    //si positivos y negativos es menor al grado, entonces existen complejas
    if (p + n < grado)
        comp1 = grado - p - n;
     //si positivos y negativos es menor al grado, entonces existen complejas, para el se-
gundo caso
    if (p2 + n2 < grado)
        comp2 = grado - p2 - n2;
    //imprime los casos
    printf("\nREGLA DE DESCARTES\n");
    printf("            caso I        caso II\n");
    printf(" positivas   %d             %d\n", p, p2);
    printf(" negativas   %d                    %d\n", n, n2);
    printf(" complejas   %d                    %d\n", comp1, comp2);
    printf(" total:      %d                    %d\n", p + n + comp1, p2 + n2 + comp2);
}

void escriben(float *p, int n) {
    //variables
    //i -> servira para iterar el arreglo de coeficientes
    //m -> disminuir el grado de x con cada coeficiente
    int i, m;
    //inicializa a m con el valor de n
    m = n;
    for (i = 0; i <= n; i++) {
        printf("(%f)X**%d", *(p+i), m);
        //imprime el simbolo de +, siempre y cuando no sea el ultimo elemento
        if (i <= n - 1)
            printf(" + ");
        //disminuye el exponente de x en cada vuelta
        m--;
    }
}

void imprime_coeficientes(float *p, int n) {
    //variables
    //i -> servira para iterar el arreglo de coeficientes
    int i;
    for (i = 0; i <= n; i++) {
        printf("(%f)X**%d", *(p + i), n - i);
        //imprime el simbolo de +, siempre y cuando no sea el ultimo elemento
        if (i <= n - 1)
```

```
            printf(" + ");
        }
    }

    float *leen(float *polinomio, int grado_del_polinomio) {
        //define las variables a utilizar
        //i -> servira para iterar hasta llegar al valor de n
        int i;
        //solicitara que se ingresen los coeficientes
        for (i = 0; i <= grado_del_polinomio; i++) {
            printf("da el coeficiente de la variable de grado %d=>", grado_del_polinomio - i);
            polinomio = (float *) realloc((float *) polinomio, (i + 1) * sizeof (float));
            scanf("%f", &polinomio[i]);
        }
        return polinomio;
    }

    //FUNCIONES PARA SISTEMAS MATRICIALES

    void determinante() {
                //a-> matriz de la cual se calculara el determinante
                //piv->pivote para realizar el metodo de gauss
                // j-> bandera que me indica si la matriz a tiene vectores linealmente dependientes
                //k-> me sirve para iterar

            int i, j, k, piv = 1;
            float **a=NULL;
            a = lee(a, 0,&i);
            // se manda a la matriz a para que se llene y guarde el numero de filas
            printf("El determinante a resolver es:\n");
            // se manda a imprimir la matriz a
            imprime(a, i, 0);
            //se manda a diagonalizar la matriz a mediante el metodo de gauss a una triangular superior
            j = gauss(a, i, 0, &piv);
            // si no existe dependencia lineal se calcula el determinante
            if (j == 0) {
                printf("la matriz diagonalizada es:\n");
                // se imprime la matriz triangular superior
                imprime(a, i, 0);
                //recordar que el determinante de una matriz triangular superior es el producto de
```

```
la diagonal
          float det = 1; //inicializa el det en uno para no alterar el producto e irlo acumulando
           for (k = 0; k < i; k++)
               det = det * (*(*(a + k) + k)); //producto de la diagonal
           printf("DETERMINANTE=%f\n", det * piv);  //imprime el determinante
       } else
           printf("DEPENDENCIA LINEAL");

}

void sistecuaciones() {
    int j, piv, num_ecuaciones = 0;
    float **matriz = NULL;
    matriz = lee(matriz, 1, &num_ecuaciones);
    printf("El sistema a resolver es:\n");
    imprime(matriz, num_ecuaciones, 1);
    j = gauss(matriz, num_ecuaciones, 1, &piv);

        if (j == 0) {
            printf("la matriz diagonalizada es:\n");
            imprime(matriz,num_ecuaciones , 1);
            printf("El resultado es:\n");
            resultado(matriz, num_ecuaciones, 0);
        }
                    else {
            printf("SISTEMA CON DEPENDENCIA LINEAL");
            getchar();
            getchar();
        }


}

void resultado1(float *a, int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("X[%d]=%15G\n", i, *(a+i));
    getchar();
    getchar();
}

void resultado(float **b, int n, int h) {
```

```
    int k, i;
    float x[n];

    x[n - 1]= *(*(b+(n-1))+n)/ (*(*(b+(n-1))+(n-1)));
    for (k = n-2; k >= 0; k--) {
          x[k]=*(*(b + k) + n);
       for (i = k + 1; i < n; i++)
           x[k]=x[k] - (*(*(b + k) + i))*x[i];
       x[k] = x[k] / *(*(b + k) + k);
    }
    if (h == 0)
       resultado1(x, n);
    else {
       resultado1(x, n);
       for (i = n; i > 0; i--)
          x[i] = x[i - 1];
       x[0] = 1;
       printf("     ECUACION CARACTERISTICA:\n");
       solucion(x, n);
    }
}

int gauss(float **a, int n, int s, int *p) {
    int k, i, j,c,f, puedo = 0;
    float m, M, temp;
    if (s==0){
          f=n;
          c=n;
    }
    else{
          f=n;
          c=n+1;
    }

    for (k = 0; k < f - 1; k++) {
       //busqueda del mayor abajo de la diagonal
       int piv = 0;
       M = fabs(*(*(a + k) + k));
       for (i = k; i < f - 1; i++) {
          temp = fabs(*(*(a + i + 1) + k));
          if (M < temp) {
             *p = -1 * *p;
```

```
            piv = i + 1;
            M = temp;
        }
    }
    //cambio de hilera
    if (M > fabs(*(*(a + k) + k))) {
        printf("ANTES DE PERMUTACION:\n");
        imprime(a, n, s);
        for (j = k; j < c; j++) {
            temp = *(*(a + k) + j);
            *(*(a + k) + j) = *(*(a + piv) + j);
            *(*(a + piv) + j) = temp;
        }
        printf("PERMUTACION:\n");
        imprime(a, n, s);
    }
    if (*(*(a + k) + k) == 0) {
        puedo = 1;
        break;
    } else {

        for (i = k + 1; i < f; i++) {
            m = -((*(*(a + i) + k)) / (*(*(a + k) + k)));

            for (j = k; j < c; j++) {
                *(*(a + i) + j) = (*(*(a + i) + j)) + ((m) * (*(*(a + k) + j)));

            }
        }
    }
}
printf("DESPUES DE LA TRANSFORMACION N\n»);
imprime(a, n, s);
if (*(*(a + n - 1)+(n - 1)) == 0)
    puedo = 1;
return (puedo);
}

float **lee(float **matriz, int k, int *num_ecuaciones) {
        int columnas, i, j, filas;
printf("DA EL NUMERO DE HILERAS DE LA MATRIZ:");
scanf("%d", &filas);
```

```
    columnas = filas+1;
    if (k == 0)
        columnas = filas;
    printf("da los elementos de la matriz A.\n");
    matriz = (float **) malloc(sizeof (float) * filas);
    for (k = 0; k < filas; k++)
        matriz[k] = (float *) malloc(sizeof (float) * columnas);
    for (i = 0; i < filas; i++) {
        for (j = 0; j < columnas; j++) {
            printf("Dame el valor a[%d][%d]=  ", i, j);
            scanf("%f", (*(matriz + i) + j));
        }
    }
    *num_ecuaciones = filas;
    return matriz;
}


void imprime(float **matriz, int m, int x) {
    int i, j, n = m+1;
    if (x == 0)
        n = m;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            float temp = *(*(matriz + i) + j);
            if (fabs(temp) < .000001 && j < i)
                temp = 0;
            printf("%13G", temp);
        }
        printf("\n");
    }
    getchar();
}
//ECUACIONES CARACTERISTICAS Y VALORES CARACTERISTICOS

void multiplica(float **a, float *b, int n) {
    float c[n];
    int i, j;
    for (i = 0; i < n; i++) {
        c[i] = 0;
        for (j = 0; j < n; j++)
            c[i] = *(*(a+i)+j) * (*(b+j)) + c[i];
```

```
    }
    for (i = 0; i < n; i++)
        *(b+i) = c[i];
}

void acomoda(float **a, float *b, int n, int k) {
    int i, m = k, s = 1;
    if (k == -1) {
        m = n;
        s = -s;
    }
    for (i = 0; i < n; i++)
        *(*(a+i)+m)=s * (*(b+i));

}

void caracteristico() {
        int num_ecuaciones,i, j;
        float **a=NULL,**c;
        printf("ECUACIONES Y VALORES CARACTERISTICOS.\n");
        a = lee(a, 0,&num_ecuaciones);
        float b[num_ecuaciones];
        printf("LA MATRIZ A TRABAJAR ES:\n");
        imprime(a, num_ecuaciones, 0);
        b[0] = 1;
        for (i = 1; i < num_ecuaciones; i++)
            b[i] = 0;
       c=(float **)malloc(sizeof(float)*num_ecuaciones);

            for (i = 0;i < num_ecuaciones; i++)
                  c[i]=(float*)malloc(sizeof(float)*num_ecuaciones);
            for(i = 0; i < num_ecuaciones; i++)
                for (j = 0; j < num_ecuaciones; j++){
                                    *(*(c+i)+j)=0;

                  }


        acomoda(c, b, num_ecuaciones, (num_ecuaciones) - 1);
        for (i = (num_ecuaciones) - 2; i >= -1; i--) {
            printf("Multiplicación No:%d\n", (num_ecuaciones) - i - 1);
            multiplica(a, b, num_ecuaciones);
```

```
            resultado1(b, num_ecuaciones);
            acomoda(c, b, num_ecuaciones, i);
            imprime(c, num_ecuaciones, 1);
        }
        int p;
        j = gauss(c, num_ecuaciones, 1, &p);
        if (j == 0)
            resultado(c, num_ecuaciones, 1);
}
```

REFERENCIAS BIBLOGRÁFICAS

La teoría sobre las librerías del manejo de memoria dinámica fue extraída del libro:
The standard C library. P. J. Pauger. Prentice Hall International, 1992.

Los conceptos sobre manejo de archivos fueron extraídos del libro:
Programación en C. Byron Gottfried. Serie Schaum, Mc Graw Hill, 1999.

Para comprender el concepto del determinante se recomienda leer cualquier libro sobre algebra lineal.

Los conceptos de programación dinámica y el algoritmo del agente viajero fueron extraídos del libro:
Fundamentals of computer algorithms, Ellis Horowitz, Sartaj Sahni. Computer Science Press, 1978.

COMENTARIOS

Para comprender en forma sencilla los conceptos de estructuras de datos, donde se incluyen la teoría del manejo de una pila, una lista, una lista doblemente enlazada y un árbol de búsqueda binario, se recomienda leer el libro:

Estructura de Datos, Osbaldo Cairó y Silvia Guardati, Mac Graw Hill.

El anexo comprende un programa de métodos numéricos. Para comprender su teoría con facilidad se recomienda leer el libro:

Apuntes de métodos numéricos, Rafael Iriarte V. Balderrama, Hugo E. Borras García, Rossynela Duran Cuevas. Editado por la Facultad de Ingeniería de la Universidad Nacional Autónoma de México.

Y si se desea un tratado más completo sobre el área se recomienda leer el libro:

Análisis Numérico, Richard. L. Burden, J. Douglas Faires. Grupo Editorial Iberoamérica.

Memoria dinámica en el lenguaje de programación C

Se terminó de imprimir en enero de 2016

Editorial Centro de estudios e investigaciones
para el desarrollo docente. Cenid AC
Pompeya Nº 2705 Col. Providencia
C.P. 44630 Guadalajara, Jalisco, México
Teléfono: 3315 420983

www.cenid.org.mx