FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACION Y DOCUMENTACION
"ING. BRUNO MASCANZONI"

El Centro de Información y Documentación Ing. Bruno Mascanzoni tiene por objetivo satisfacer las necesidades de actualización y proporcionar una adecuada información que permita a los ingenieros, profesores y alumnos estar al tanto del estado actual del conocimiento sobre temas específicos, enfatizando las investigaciones de vanguardia de los campos de la ingeniería, tanto nacionales como extranjeras.

Es por ello que se pone a disposición de los asistentes a los cursos de la DECFI, así como del público en general los siguientes servicios:

- **Préstamo interno.**
- **Préstamo externo.**
- **Préstamo interbibliotecario.**
- **Servicio de fotocopiado.**
- **Consulta a los bancos de datos: librunam, seriunam en cd-rom.**

Los materiales a disposición son:

- **Libros.**
- **Tesis de posgrado.**
- **Noticias técnicas.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1980 a la fecha.**

En las áreas de ingeniería industrial, civil, electrónica, ciencias de la tierra, computación y, mecánica y eléctrica.

El CID se encuentra ubicado en el mezzanine del Palacio de Minería, lado oriente.

El horario de servicio es de 10:00 a 19:30 horas de lunes a viernes.

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

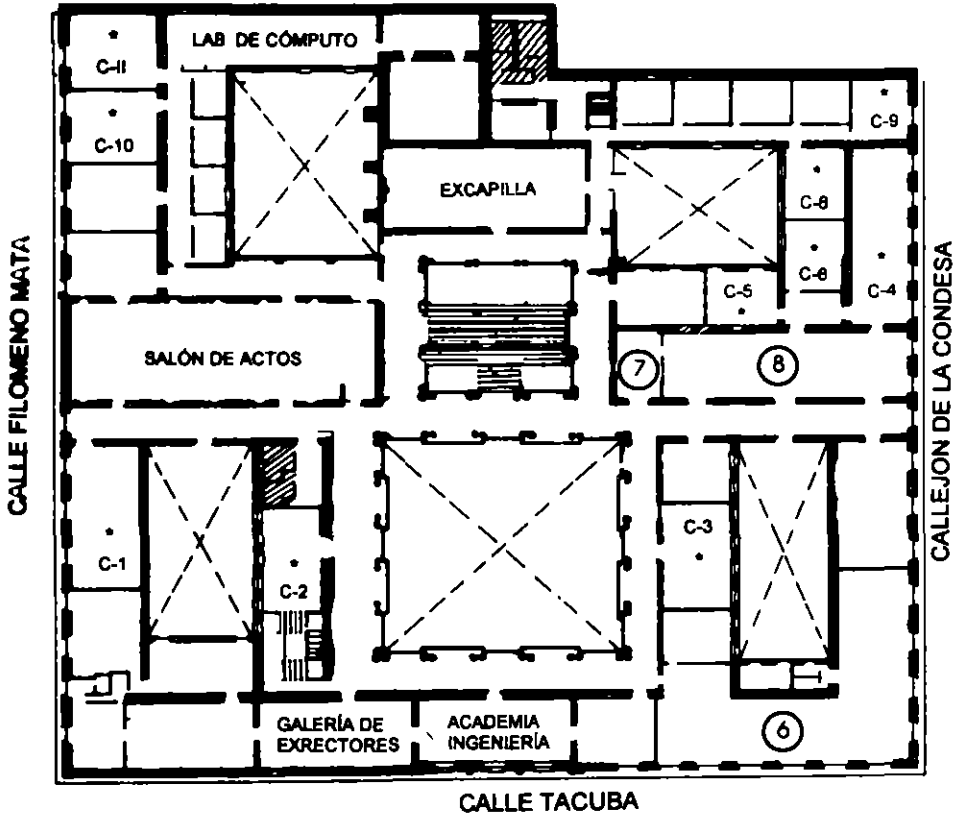
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

División de Educación Continua.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

PALACIO DE MINERIA



Ier. PISO

GUÍA DE LOCALIZACIÓN

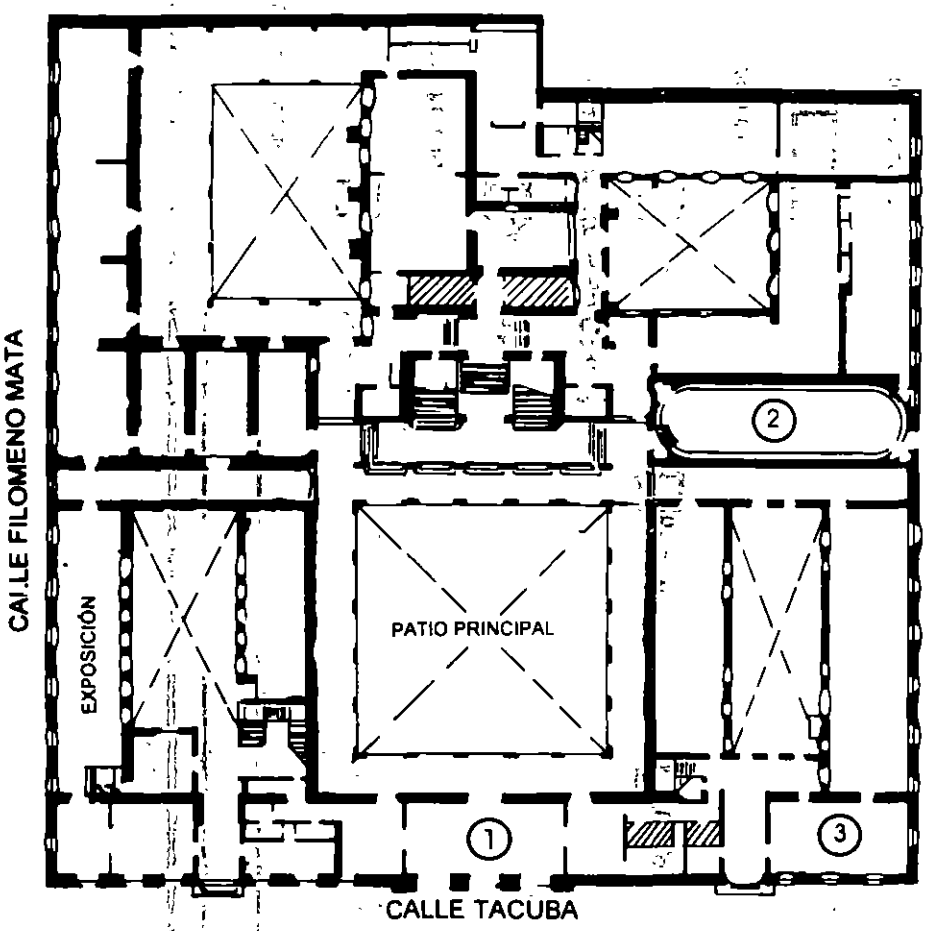
1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

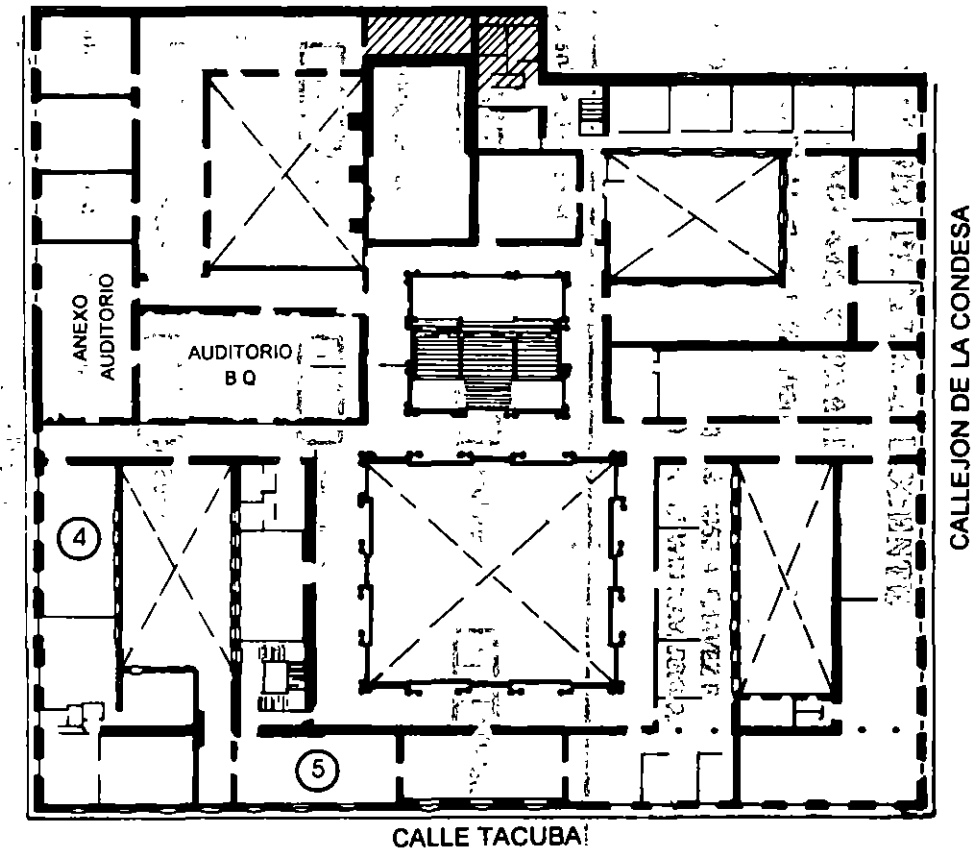
* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

PALACIO DE MINERIA



PLANTA BAJA



MEZZANINNE

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ - ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ

ΕΘΝΙΚΗ ΑΡΧΗ ΑΞΙΟΛΟΓΗΣΗΣ ΚΑΙ ΠΙΣΤΟΠΟΙΗΣΗΣ

ΔΗΛΩΣΗ ΣΥΜΜΕΤΟΧΗΣ ΣΤΗΝ ΕΞΕΤΑΣΗ

Α) ΝΕΟΣ ΥΠΟΨΗΦΙΟΣ

Β) ΝΕΟΣ ΥΠΟΨΗΦΙΟΣ ΜΑΘΗΤΗΣ ΕΠΑ.Λ.

Γ) ΝΕΟΣ ΥΠΟΨΗΦΙΟΣ ΕΠΑΝΑΛΗΠΤΗΣ

Δ) ΝΕΟΣ ΥΠΟΨΗΦΙΟΣ ΕΠΑΝΑΛΗΠΤΗΣ ΜΑΘΗΤΗΣ ΕΠΑ.Λ.

Σημειώστε με «Χ» τη σχετική επιλογή.

1. Επιθυμώ να εξεταστώ στα μαθήματα για τα οποία δηλώνω συμμετοχή στο εξεταστικό κέντρο:

Μαθηματικά	Φυσική	Χημεία	Βιολογία
Ιστορία	Αρχαία Ελληνικά	Λατινικά	Θρησκευτικά
Νεοελληνική Γλώσσα και Γραμματεία	Νεοελληνική Λογοτεχνία	Αγγλικά	Γαλλικά
Γερμανικά	Ιταλικά	Ισπανικά	Ρωσικά
Τουρκικά	Αραβικά	Κινεζικά	Εβραϊκά

2. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στην Α΄ Λυκείου:

Ναι ☐ Όχι ☐

3. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Β΄ Λυκείου:

Ναι ☐ Όχι ☐

4. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

5. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

6. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

7. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

8. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

9. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

10. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

11. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

12. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

13. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

14. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

15. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

16. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

17. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

18. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

19. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

20. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

21. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

22. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

23. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

24. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

25. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

26. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

27. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

28. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

29. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

30. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

31. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

32. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

33. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

34. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

35. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

36. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

37. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

38. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

39. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

40. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

41. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

42. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

43. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

44. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

45. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

46. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

47. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

48. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

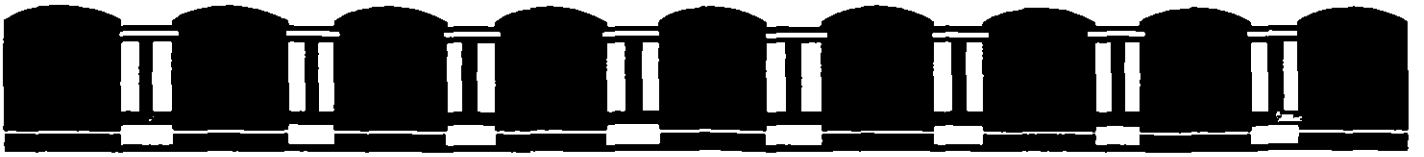
Ναι ☐ Όχι ☐

49. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐

50. Η εξέταση θα γίνει στην ξένη γλώσσα που επέλεξα στη Γ΄ Λυκείου:

Ναι ☐ Όχι ☐



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

LENGUAJE " C"

II PARTE

Palacio de Mineria. Calle de Tacuba 5, primer piso, Delegación Cuauhtémoc, 06000 México, D.F. Tel. 521-4022 y 23, Fax 510-0573 Apdo. Postal M-2285

ARREGLOS
Y
APUNTADORES

ARREGLOS

Un arreglo es una colección de elementos del mismo tipo.

Un arreglo se define de la siguiente forma:

tipo nombre[tamaño];

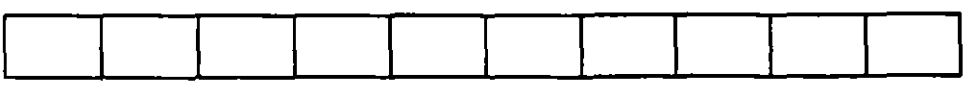
donde:	tipo	tipo de los elementos que constituyen el arreglo.
	nombre	identificador del arreglo.
	tamaño	número de elementos del arreglo.

Para accesar a los elementos de un arreglo hay que hacer referencia a ellos mediante un índice. El primer elemento del arreglo es tiene como índice 0, por lo tanto, para el último el índice es *tamaño-1*.

Cuando se define un arreglo se reservan localidades contiguas de memoria para almacenar cada uno de sus alementos, aún cuando el arreglo sea multidimensional.

Ejemplo:

int v[10];



v[0] v[1] v[2] v[3] v[4] v[5] v[6] v[7] v[8] v[9]

Inicialización de arreglos

Los arreglos externos y estáticos numéricos inicializan sus elementos con cero.

Los arreglos externos y estáticos de caracteres, inicializan sus elementos con el caracter cuyo código ASCII es '\0'.

Los arreglos pueden inicializarse de forma explícita de la siguiente forma:

```
int    x[5] = { 2, 6, 8, 12, 28};
```

En una inicialización explicita:

- El número de inicializadores puede ser menor que el número de elementos en el arreglo, en este caso los elementos restantes se inicializan con cero:

```
int    x[10] = { 4, 5, 7};
```

- Es un error el que el número de inicializadores sea mayor que el tamaño del arreglo.

- No es necesario especificar su dimensión, la dimensión sera el número de inicializadores:

```
int x[] = { 1, 5, 5, 7};      /* Se define un arreglo de 4 elementos: int x[4] */
```

Arreglos de caracteres

En el lenguaje C no existe el tipo *string* o *cadena*, sin embargo, una cadena de caracteres puede ser representada con un arreglo de caracteres.

Para poder llevar a cabo operaciones con la cadena de caracteres, por ejemplo, concatenaciones o busqueda de subcadenas en la misma, es necesario determinar en que posición dentro del arreglo termina la cadena.

Las funciones estandar para manejo de arreglos de caracteres asumen que el final de una cadena lo determina el caracter '\0', esto no impide que el programador pueda construir funciones para manipulación de cadenas utilizando sus convenciones.

Es importante remarcar, que si se desea trabajar con arreglos de cadenas utilizando las funciones de la libreria estandar, se debe tomar en cuenta esta convención. Todas las librerias disponibles para C toman esta convención como un estandar.

Por lo tanto, si se desea manipular una cadena de longitud N, se debera definir un arreglo de tamaño N+1, ya que se tiene que considerar un elemento adicional para el terminador.

Es responsabilidad del programador asegurarse de que no se excedan los límites de la cadena.

Las constantes cadena se escriben entre comillas:

```
"HOLA MUNDO"    /* La cadena tiene 11 elementos (incluyendo el terminador) */
```

Es un error común utilizar constantes como **'a'** y **"a"** en forma indistinta; hay que recordar, que la primera representa un solo caracter y la segunda una cadena de dos caracteres.

La inicialización de arreglos de caracteres se puede hacer de una forma semejante a la inicialización de arreglos numéricos

```
char   mensaje[] = { 'H', 'O', 'L', 'A', ' ', 'M', 'U', 'N', 'D', 'O', '\0'};
```

o bien:

```
char    cadena[] = "HOLA MUNDO";
```

Ejemplo:

```
/*
**      Programa 5_1
**
**      Este programa simplemente lee el nombre de una persona y lo almacena
**      como una cadena de caracteres; sin embargo, no se utilizan las funciones
**      de la libreria estandar para manejo de strings.
**
*/

#include    <stdio.h>
#define     MAX_NOM_LONG  100

main()
{
        int     i=0;
        char    nombre[N],
                c;

        printf("\tDame tu nombre: ");
        while ((c = getchar()) != '\n')Cada caracter leido es almacenado como elemento del arreglo */
                nombre[i++] = c;
        nombre[i] = '\0';                                       /* Se agrega el terminador */
        printf("\nMuchas gracias %s por haber dado tu nombre\n",  nombre);
}
```

Arreglos multidimensionales

En un arreglo multidimensional, las localidades de memoria que se reservan, al igual que en un arreglo unidimensional, son contiguas; sin embargo, se utiliza una función de mapeo de direcciones en base a las dimensiones del arreglo para determinar la posición en donde se encuentra cada elemento.

El programador no debe preocuparse por el mapeo que se hace al "arreglo unidimensional".

Los arreglos multidimensionales más utilizados son aquellos de dos dimensiones, por ejemplo, en la representación de matrices.

La forma de definir un arreglo multidimensional, por ejemplo de dos dimensiones, es la siguiente:

```
int   matriz[10][10];  /* Se reserva localidades de memoria continua para 100 enteros */
```

En este caso el primer índice representa los renglones y el segundo las columnas; sin embargo, esto no implica que el compilador maneje un arreglo de dos dimensiones como un conjunto de renglones y columnas. Este manejo depende totalmente del programador, de tal forma que se podría manipular el arreglo de tal forma que el primer índice representara a las columnas y el segundo a los renglones.

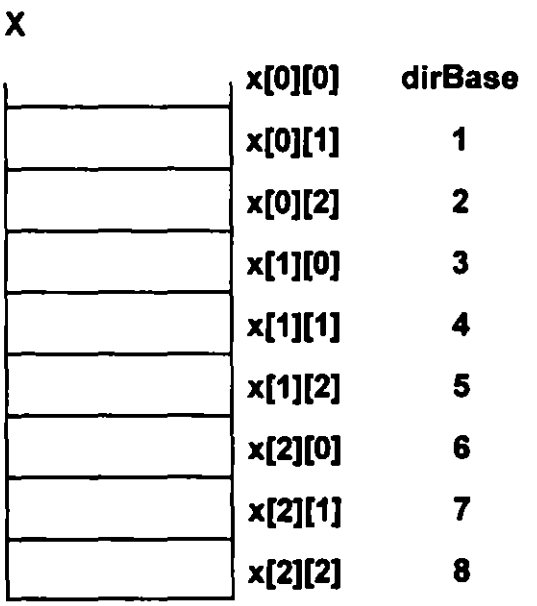
La función de mapeo o transformación de direcciones se utiliza para calcular la posición de un elemento en el espacio de memoria ocupado por el arreglo en base a sus índices; por ejemplo, para un arreglo bidimensional, la función sería la siguiente:

dirBase + n*i + j

donde:	dirBase	dirección de inicio del arreglo
	n	tamaño de la segunda dimensión
	i	primer índice del elemento
	j	segundo índice del elemento

Ejemplo:

```
int    x[3][3];
```



Para el ejemplo anterior, el elemento x[1][2] se encuentra 3*1 + 2 = 5 posiciones después de la dirección de inicio del arreglo.

Como se puede observar, para la función de mapeo de un arreglo bidimensional, no se utiliza la segunda dimensión con la que fue definido el arreglo.

Cuando se inicializa un arreglo multidimensional (N dimensiones) es necesario especificar por lo menos N-1 dimensiones para que se pueda llevar a cabo el mapeo de direcciones.

La inicialización de arreglos multidimensionales es muy parecida a la de los arreglos unidimensionales:

Ejemplo:

Un arreglo de dos dimensiones, es un arreglo de arreglos:

```
int x[3][3] = { { 3, 6, 9},  { 8, 5, 1},  { 1, 1, 5}};
```

Sin ambargo, se almacena como un arreglo unidimensional:

```
int x[3][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};
```

Si se inicializan todos sus elementos, no se necesita indicar la primera dimensión, en este caso se crea un arreglo con tantos "renglones" como sean necesarios para alojar a todos los valores de la inicialización:

```
int x[][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};
```

LABORATORIO

1. Escriba una función que haga una búsqueda secuencial de un elemento sobre un arreglo. La función debe regresar como valor la posición en donde se encuentra el elemento ó -1 si no se encuentra.

APUNTADORES

Todas las variables se almacenan en memoria ocupando cierto número de bytes dependiendo del tipo de la misma. El lugar en donde se almacena la variable determina su dirección de almacenamiento.

Un apuntador es una variable que almacena una dirección de memoria.

Los apuntadores son utilizados para manejo de memoria dinámica, para simular paso de parámetros por variable y para la creación de estructuras de datos complejas. El manejo de apuntadores es la base para el desarrollo de aplicaciones complejas que tienen por objetivo la eficiencia del código que utilizan.

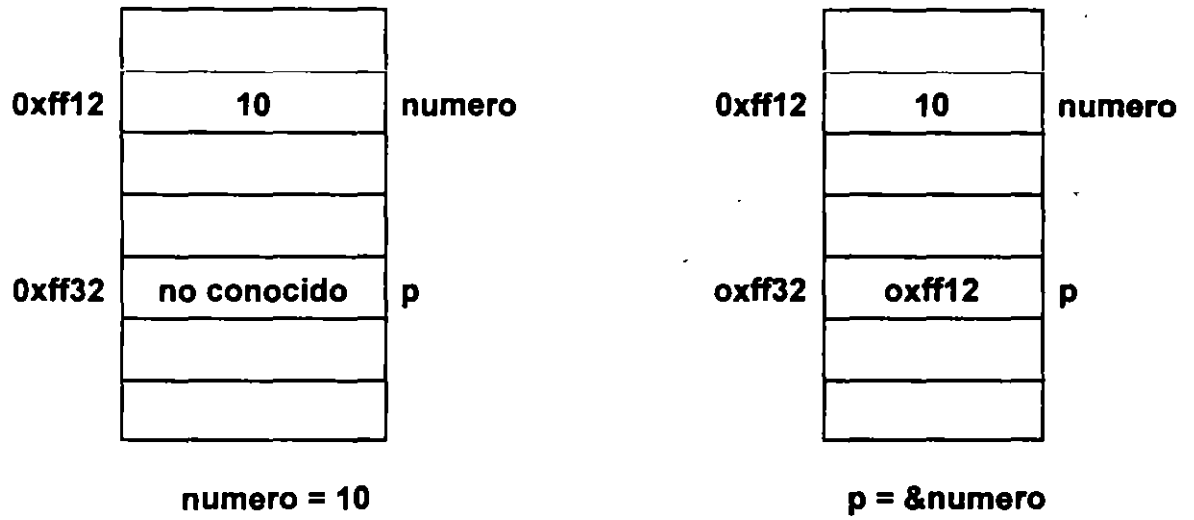
Un apuntador puede contener la dirección de una variable entera, caracter, double, de un apuntador, etc.; el tipo de esta variable, determina el tipo de apuntador: apuntador a entero, apuntador a carácter, apuntador a double, apuntador a apuntador, etc. Por ejemplo, para definir un apuntador a entero:

```
int          *p;
```

En este caso se esta definiendo una variable apuntador a entero. La dirección que contiene el apuntador, al momento de ser definido, es una dirección desconocida (en general, no es una dirección de memoria que se pueda accesar sin peligro de dañar otros procesos), puesto que este no ha sido inicializado.

Para asignar a un apuntador una dirección válida se puede utilizar el operador de dirección **&**, el cuál obtiene la dirección de una variable. Por ejemplo, supongase el siguiente fragmento de código:

```
int    numero;
int    *p;

numero = 10;
p = &numero;
```

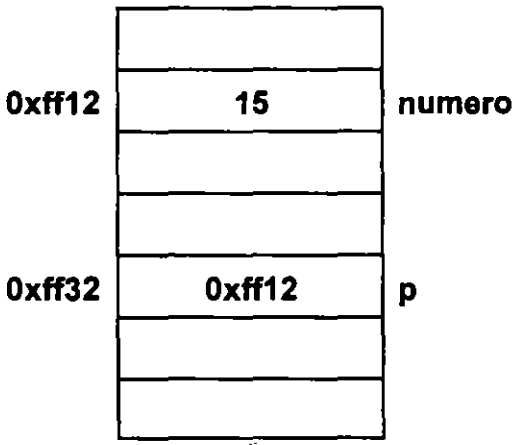


El ejemplo anterior asigna a **p** la dirección de **numero** y se puede accesar el valor de la variable **numero** directamente o indirectamente por medio del apuntador p.

El apuntador de dirección & es unuario y solamente se aplica a variables.

Para poder hacer referencia a la memoria direccionada por el apuntador, se utiliza el operador de desreferencia o indirección *. Por ejemplo, para cambiar el valor de la variable entera **numero** en forma indirecta (en este caso, el ejemplo no muestra un caso común del manejo de apuntadores):

***p = 15;**



***p = 15**

en este caso, la expresión ***p** es una variable entera y se puede utilizar en cualquier contexto que acepte expresiones enteras:

Ejemplo:

Suponga las siguientes definiciones:

```
int       x = 2, z, *p, *q;
double    d;
```

Analizemos la siguiente secuencia de operaciones:

```
p = &x;                                          /* p contiene la direccion de x */
z = *p + 1;          /* A z se le asigna el valor de x + 1 (puesto que p apunta a x) => y = 3 */
d = sqrt(*p);          /* A d se le asigna la raiz cuadrada de el valor de x => d = sqrt(2) */

*p = 0;                                          /* El valor de x ahora es cero */
*p += 1;                                     /* El valor de x se incrementa en uno=> x=1 */
(*p)++;          /* El valor de x se vuelve a incrementar (son importantes los parentesis,
                 debido a la presedencia de los operadores */

q = p;                                  /* El apuntador q tambien se le asigna la direccion de x */
```

Los apuntadores se pueden inicializar al momento de ser definidos:

```
int     x = 4,
        *p = &x;
```

No se puede asignar a un apuntador una dirección de una variable que no es del tipo del apuntador:

```
int             *p;
double          f;

p = &f;     /* no es valido */
```

Generalmente muchos programas no conocen la cantidad de memoria que van a requerir, por ejemplo, si queremos crear un arreglo para guardar los nombres de los clientes de una empresa, de que tamaño debera ser el arreglo?, ¿qué sucede si se reserva más o menos memoria de la que se necesita?. Un programa eficiente solo debería ocupar la memoria que realmente requiere. Para solucionar este problema, se maneja el concepto de **memoria dinámica**, es decir, se reserva memoria a tiempo de ejecución.

Para reservar memoria a tiempo de ejecución, se utiliza la función **malloc,** la cuál reserva n bytes de memoria y regresa como valor un apuntador generico (void *) el cuál se puede asignar a una variable apuntador utilizando una conversión explicita:

```
int     *p;
p = (int *)malloc(sizeof(int));
```

el operador sizeof obtiene el número de bytes que ocupa un tipo.

Apuntadores como parámetros de funciones

La forma de pasar parámetros a las funciones es por valor, esto implica, que la función no puede cambiar los valores de los parámetros actuales.

Para que en una función pueda cambiar el valor de una variable, no definida en la misma función, en primera instancia, es necesario que este definida como externa:

Ejemplo:

```
/*
**      Programa 5_2
**
**      Este programa muestra el manejo de variables externas como una alternativa al
**      paso de parámetros por referencia
**
*/

#include    <stdio.h>
#define     N     10

int     max;
void    maximo(int,int);

main()
{
        int i, num;

        printf("Proporciona 10 numeros:\n");
        scanf("%d", &num);
        max = num;
        for (i=1; i < N; i++)
        {
                scanf("%d", &num);
                maximo(max,num);
        }
        printf("\nEl numero mayor es: %d\n", max);
}

void    maximo(int x,int y)
{
        max = x > y ? x : y;

}
```

Otra forma de regresar un valor a la función que hace la llamada es con el uso de la sentencia return en una función.

Ejemplo:

```
/*
**      Programa 5_3
**
**      Este programa muestra el regreso de valores en la cláusula return, como una
**      alternativa para obtener valores de regreso de la llamada a una funcion.
**
*/

#include    <stdio.h>
#define     N     10

int maximo(int,int);

main()
{
        int     i, num, max;

        printf("Proporciona 10 numeros:\n");
        scanf("%d", &num);
        max = num;
        for (i=1; i < N; i++)
        {
                scanf("%d", &num);
                max = maximo(max, num);
        }
        printf("\nEl numero mayor es: %d\n", max);
}

int     maximo(int x, int y)
{
        return ( x > y ? x : y);
}
```

Sin embargo:

- El uso de variables externas no es muy recomendable por la programación estructurada.

- El uso de return solamente permite el regreso de un valor.

- Muchas funciones necesitan regresar más de un valor.

Cuando se utilizan apuntadores como parámetros, el valor de las variables, direccionadas por el apuntador, puede cambiar en la llamada a una función.

Consideremos el diseño de la función swap, esta función deberá intercambiar el valor de sus dos parámetros:

```
void    swap(int a, int b)
{
        int aux;

        aux = a;
        a = b;
        b = aux;
        printf("\nLos valores son a = %d y b = %d", a, b);
}
```

¿Cuál será la salida generada por el programa 5_4 que hace utiliza la función swap previamente definida?

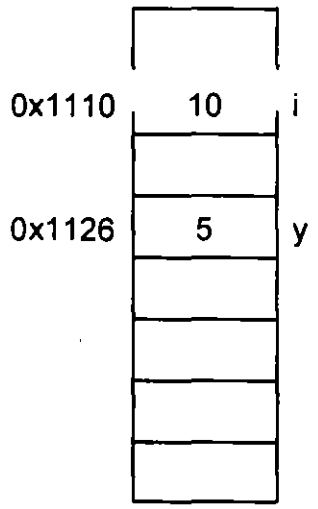
```
/*
**      Programa 5_4
**
**      Programa que implementa la función swap sin manejo de apuntadores
**
*/

#include <stdio.h>

void swap(int,int);

main()
{
        int     i = 10, y = 5;

        swap(i, y);
        printf("\nLos valores son i = %d y y = %d", i, y);
}

void    swap(int a, int b)
{
        int aux;

        aux = a;
        a = b;
        b = aux;
        printf("\nLos valores son a = %d y b = %d", a, b);
}
```

Consideremos el definir apuntadores a entero como parametros, como lo muestra el programa 5_5:

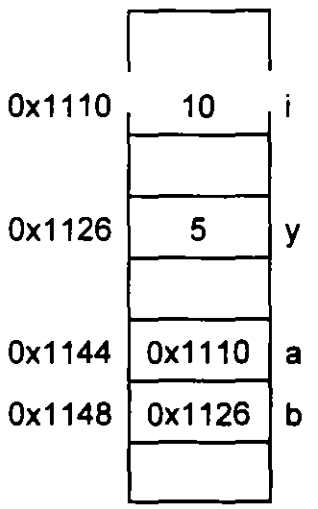
```
/*
**      Programa 5_5
**
**      Programa que implementa la función swap conmanejo de apuntadores
**
*/

#include <stdio.h>

void swap(int *, int *);

main()
{
        int     i = 10, y = 5;

        swap(&i, &y);
        printf("\nLos valores son i = %d y y = %d", i, y);
}

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

        aux = *a;
        *a = *b;
        *b = aux;
        printf("\nLos valores son a = %d y b = %d", *a, *b);
}
```

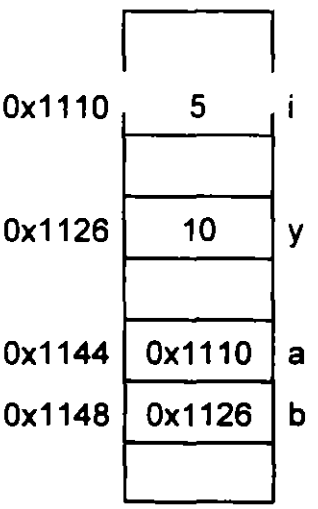
¿ Cuál es la salida que produce el programa?

El comportamiento de la función swap se muestra con la siguiente figura:

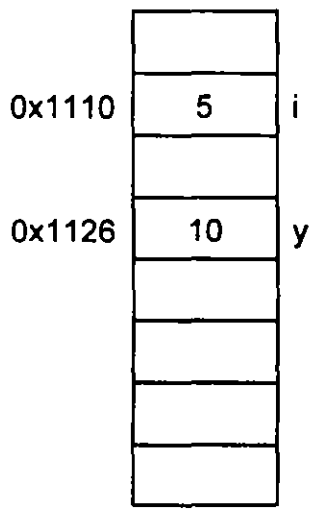


antes de la llamada a swap





en la llamada a swap



después de la llamada a swap

Para simular el paso de parámetros por variable, se debe considerar:

- Definir los parámetros formales como apuntadores.
- En la definición de la función, hay que utilizar el operador de indirección al momento de accesar la memoria direccionada por los apuntadores.
- El parámetro actual en la llamada a la función debe ser una dirección.

Apuntadores y arreglos

Existe una gran relación entre apuntadores y arreglos.

El nombre de un arreglo es una variable donde se guarda la dirección de inicio del arreglo, su valor no puede ser modificado.

Muchas veces los apuntadores y los arreglos son utilizados con el mismo propósito; sin embargo, cabe recordar que un apuntador es una variable y un arreglo es un apuntador constante.

Cuando se define un arreglo de tamaño N, se reservan N localidades contiguasde memoria. Por otra parte, cuando se define un apuntador solamente se reserva el espacio para la variable que representa.

Suponga las siguientes definiciones:

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

las siguientes expresiones son validas:

```
p = x;                          /* es equivalente a p = &x[0] */

p =  x + 1;                     /* utilizando notación de apuntadores, p apunta a x[1] */

p++;                            /* p apunta ahora a x[2] */

*(x + 5) = 10;                  /* x[5] = 10 */

*(p + 10) = 15;                 /* es válido pero se accesa una  localidad de memoria no válida */
```

la siguientes expresiones no son válidas:

```
x++;                            /* x es un apuntador constante */

*(x + 10) = 20;                 /* x es la dirección de un arreglo  y la localidad máxima que se puede
                                   accesar es x + 9 */
```

Ejemplo:

```
/*
**      Programa 5_6
**
**      Programa que ayuda a comprender el manejo de apuntadores y arreglos.
**      Intente determinar la salida del programa sin ejecutarlo.
**
*/

#include      <stdio.h>

main()
{
        char        x[] = "ESTA ES UNA CADENA EJEMPLO";
        char        *p = x,
                    *q = x + 2;

        printf("%d %c %c %d %d %c", *p, p[0], *p + 1, *(p +5),  q[2] + 3, *x);
        p+=2;

        printf("%d %c %c", p[0], **&p + 5, *(p + 4));

        printf("%c %c %c", *(p++), *p, *(p + 1));
}
```

¿ Cuál es la salida del programa ?

Arreglos como parámetros de funciones

Cuando un arreglo es pasado como parámetro a una función, en realidad se esta pasando una dirección, ya que el nombre de un arreglo es la dirección del primer elemento del mismo.

El parámetro formal puede ser definido como arreglo a como apuntador, el parametro actual puede ser un apuntador o un arreglo.

En el caso de que el parámetro formal sea definido como un arreglo unidimensional no es necesario especificar el tamaño, en este caso es responsabilidad del programador no rebasar los límites del arreglo en la función.

En el caso de arreglos multidimensionales es necesario especificar todas las dimensiones a excepción de la primera, esta no es necesaria ya que no se utiliza en la fórmula de transformación de direcciones.

En el caso de arreglos multidimensionales, las dimensiones especificadas en el parámetro formal pueden no ser las mismas que las del parámetro actual. La función lo único que recibe es una dirección de inicio e información para accesar los elementos por medio de la fórmula de mapeo.

Ejemplo:

```
/*
**      Programa 5_6
**
**      Programa que muestra el paso de arreglos como parametros en funciones.
**
*/
#include <stdio.h>

void  f(int [][3]);

main()
{
        int     matriz[4][4] = { { 1, 2, 3, 4 },
                                 { 5, 6, 7, 8 },
                                 { 9, 10, 11, 12 },
                                 { 13, 14, 15, 16}};

        f(matriz);

}

/*
**      f
**
**      Esta funcion recibe como parametro un arreglo bidimensional. En la funcion,
**      el arreglo se manipula con 3 "columnas".
**
**      PARAMETROS:
**              a       direccion de inicio del arreglo.
**
*/
void    f(int a[][3])
{
        int     i;

        for(i=0; i < 5; i++)
                printf("%d ", a[i][2]);
}
```

Aritmética de apuntadores

La aritmética de apuntadores es una de las características más eficaces del lenguaje C.

Si p es un apuntador a un tipo de dato e inicialmente tiene una dirección x, por ejemplo la dirección 0x1876; la expresión p + 1 no es necesariamente la dirección 0x1877. El incremento no es unitario necesariamente, sino que depende del número de bytes que se necesiten para almacenar un elemento del tipo al cual direcciona el apuntador.

En el caso de que p fuera una apuntador a int y que el tipo int ocupará 2 bytes, p++ ocasionaría que p apuntara dos bytes adelante de su dirección original.

Se pueden manejar sumas y restas de direcciones exclusivamente.

En forma general se puede decir que la aritmética de apuntadores no es igual a la de enteros.

Funciones para manejo de caracteres y cadenas

Las funciones que se mencionan a continuación, permiten determinar la naturaleza de un carácter, todas ellas reciben como parámetro un valor numérico o char.

Para poder utilizar estas funciones es necesario especificar el header **ctype.h**

Todas estas funciones regresan un valor verdadero, si el carácter es del tipo que se esta validando y un valor de cero en cualquier otro caso.

NOMBRE DE FUNCION	PROPOSITO
isalnum(int c)	Determina si c es alfanumérico
isalpha(int c)	Determina si c es letra
iscntrl(int c)	Determina si c es carácter de control
isdigit(int c)	Determina si c es dígito
isprint(int c)	Determina si c es imprimible
islower(int c)	Determina si c es minúscula
isspace(int c)	Determina si c es blanco
isupper(int c)	Determina si c es mayúscula

El lenguaje C cuenta con un conjunto de funciones para manejo de cadenas, recuerde que en este caso se considera una cadena a un arreglo de caracteres terminado con '\0'. Los prototipos de estas funciones se encuentran en el archivo **string.h**

strcpy

char *strcpy(char *s1, char *s2)

Copia s2 a s1, incluye en s1 el carácter '\0'.

Regresa como valor s1.

strncpy

char *strncpy(char *s1, char *s2, int n)

Copia n caracteres de s2 a s1, incluye en s1 el carácter '\0'.

Regresa como valor s1.

strcat

char *strcat(char *s1, char *s2)

Concatena s2 a s1 incluye en s1 el carácter '\0'.

Regresa como valor s1.

strncat

char *strncat(char *s1, char *s2, int n)

Concatena n caracteres de s2 a s1 incluye en s1 el carácter '\0'.

Regresa como valor s1.

strcmp

int strcmp(char *s1, char *s2)

Compara las cadenas s1 y s2, no compara la longitud de ellas, sino el orden lexicográfico de cada uno de sus caracteres.

Regresa:

menor a cero si s1 < s2,
mayor a cero si s1 > s2,
cero si s1 = s2

strncmp

int strncmp(char *s1, char *s2, int n)

Compara a lo más n caracteres de las cadenas s1 y s2.

Regresa:

menor a cero si s1 < s2,
mayor a cero si s1 > s2,
cero si s1 = s2

strchr

char *strchr(char *s1, int c)

Busca la primera ocurrencia del carácter c en s1.

Regresa la dirección de la primera ocurrencia de c en s1o NULL si c no esta en s1.

strlen

int strlen(char *s1)

Calcula la longitud de s1 no contando el terminador.

Regresa la longitud de la cadena.

A continuación se muestra la implementación de algunas de ellas:

```
int strcmp(char *s, char *t)
{
        for( ; *s == *t ; s++, t++)
                if (*s ==  '\0')
                        return 0;
        return *s - *t;
}


char *strcat(char *s1, char *s2)
{
        char            *aux = s1;

        while(*aux++)
                ;
        --aux;
        while (*aux++ = *s2++)
                ;
        return s1;
}


int strlen(char *s)
{
        int     n=0;

        while(*s++)
                n++;
        return n;
}
```

LABORATORIO

1. Escriba una función que determine si una cadena de caracteres es un palíndromo. Un palíndromo es una cadena de caracteres que se lee igual hacia adelante que hacia atrás.

2. Escriba una función llamada substr que busque la ocurrencia de una cadena en otra y que regrese como valor la posición a partir de la que se encuentra; en caso de que no se encuentre, la función deberá regresar -1.

Arreglos de apuntadores

El lenguaje C permite la definición de tipos complejos a partir de los tipos básicos, de esta forma se pueden crear arreglos de apuntadores, arreglos de arreglos de apuntadores, etc.

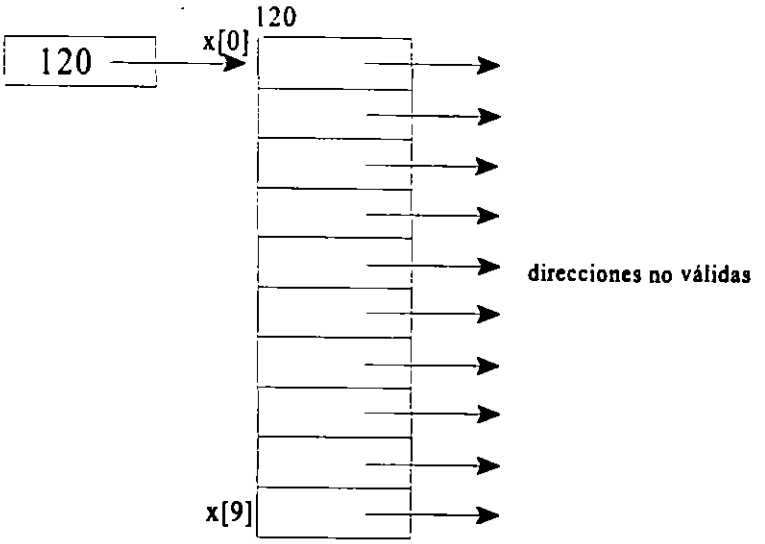
Uno de los tipos complejos más utilizados son los arreglos de apuntadores, ya que son mucho más flexibles que los arreglos multidimensionales, además de que generalmente, requieren de menos memoria.

La forma de definir un arreglo de apuntadores a carácter sería la siguiente:

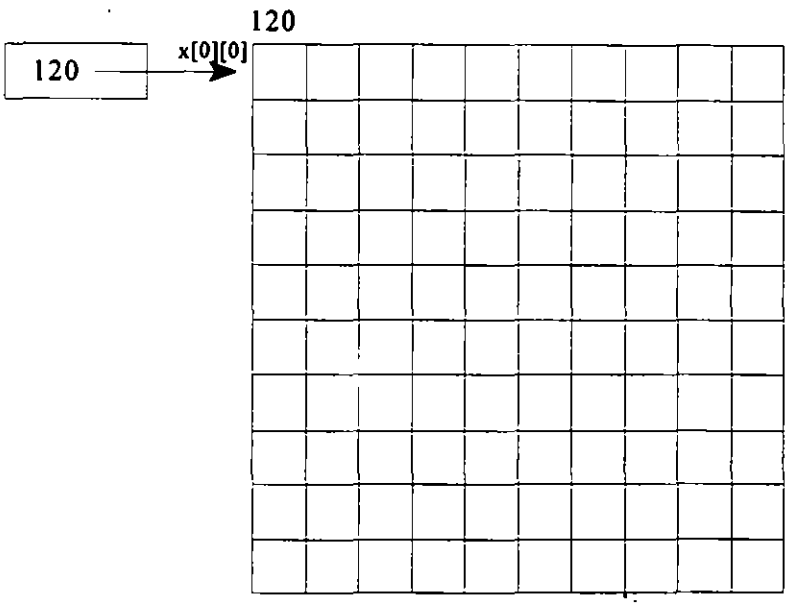
```
char        *x[10];
```

La representación interna de este arreglo es muy diferente a la de un arreglo bidimensional de caracteres, aunque la forma de hacer referencia a cada uno de sus elementos sea muy parecida.

```
char    *x[10];
```




```
char    x[10][10];
```



los elementos se almacenan en localidades contiguas

Una vez creado el arreglo de apuntadores, se deberá asignar a cada uno de los elementos direcciones de memoria previamente reservadas, esto se puede hacer dinamicamente a tiempo de ejecución de la forma siguiente:

x[0] = (char *)malloc(..);

De esta forma cada uno de los elementos se puede considerar como un arreglo de N caracteres, donde N no esta limitado más que por la cantidad de memoria que pueda tener disponible la máquina.

Ejemplo:

```
/*
**      Programa 5_7
**
**      Este programa permite la implementación de una agenda en donde se
**      guardan los nombres de los amigos de una persona. La agenda es
**      ordenada posteriormente.
**      Se utiliza un arreglo de apuntadores a caracter para la implementación
**      de la agenda, de esta forma no se desperdicia memoria por los
**      nombres cortos.
**
*/
#include <stdio.h>

#define     MAX         20    /* Máximo número de elementos en el arreglo */
#define     MAX_NOM     35    /* Longitud maxima de un nombre de persona */

void   ordena(char *[], int);
void   imprime(char *[], int);

main()
{
        char    *agenda[MAX],
                nombre[MAX_NOM];
         int    i=0,j;
```

```
        printf("\nProporciona el nombre de tus amigos:\n");
        while (gets(nombre) != NULL)
        {
                /*
                ** Se utiliza la variable nombre como un buffer para leer los nombres
                ** de las personas. Una vez leido el nombre, se reserva la memoria
                ** necesaria para almacenarlo, la dirección de inicio de esta se asigna
                ** a un elemento del arreglo y se copia el nombre en dicha direccion.
                */
                agenda[i] = (char *)malloc((strlen(nombre)+1) * sizeof (char));
                strcpy(agenda[i], nombre);
                i++;
         }
        ordena(agenda, i);
        printf("\n\nLista Ordenada:\n");
        imprime(agenda, i);
}

/*
**      ordena
**
**      Esta funcion ordena las cadenas apuntadas por los elementos de un arreglo de
**      apuntadores a caracter. Se utiliza el metodo conocido como burbuja.
**
**      PARAMETROS:
**              x               arreglo de apuntadores a caracter.
**              n               numero de elementos del arreglo.
**
**      RETURN:
**              ninguno
*/
void    ordena(char *x[], int n)
{
        int     i,j;
        char    *aux;

        for(i=0; i < n - 1; i++)
                for (j = n - 1; i<j; j--)
                        /*
                        ** Las cadenas se tienen que comparar caracter por caracter,
```

```
                ** como lo hace la funcion strcmp.

                */
                if ((strcmp(x[j-1], x[j])) > 0)
                {
                        /*
                        ** Solo se intercambian direcciones.
                        */
                        aux = x[j-1];
                        x[j-1] = x[j];
                        x[j] = aux;
                }
}

/*
**      imprime
**
**      Esta funcion imprime las cadenas de un arreglo de apuntadores a caracter.
**
**      PARAMETROS:
**              x               arreglo de apuntadores a caracter.
**              n               numero de elementos del arreglo.
**
**      RETURN:
**              ninguno
*/
void    imprime(char *x[], int n)
{
        int     i;

        for(i=0; i < n; i++)
                printf("%s\n", x[i]);
}
```

Parámetros para main

En la función main se pueden utilizar dos parámetros para establecer comunicación con el sistema operativo al momento que se lleva a cabo la ejecución del programa.

Los argumentos se llaman **argc** y **argv.** El primero de ellos almacena el número de argumentos en la línea de comandos que recibe el programa ejecutable y el segundo es una arreglo de apuntadores a char en donde se almacenan dichos argumentos.

El primer elemento argv[0] contiene el nombre del programa ejecutable.

Ejemplo:

```
/*
**      Programa 5_8
**
**      Este programa despliega los argumentos con los que se ejecuta el programa. Para
**      su ejecucion debera teclear a nivel de sistema operativo:
**
**      lab_5_8.exe  argumento1 argumento2 ... argumentoN
**
**      donde:          argumento1, ..., argumentoN  son cualquier palabra.
*/
#include <stdio.h>

main(int argc, char **argv)
{
        int i;

        for(i=0 ; i < argc; i++)
                printf("%s ", argv[i]);
        putchar('\n');
```

```
}
```

LABORATORIO

1. Modifique el programa 5_7, para que una vez que se han proporcionado y ordenado los elementos de la agenda, se hagan búsquedas sobre esta; es decir, se proporcionara un nombre y el programa determinara si se encuentra dado de alta o no.

ESTRUCTURAS
Y
UNIONES

typedef

C es un lenguaje que puede ampliarse con facilidad, por ejemplo, con la definición de símbolos mediante los #define y creando funciones de propósito general para uso de todos los usuarios.

También puede ampliarse el lenguaje al definir tipos de datos que se construyen con los tipos ya existentes.

También se pueden definir tipos mucho muy complejos con el uso de estructuras y apuntadores.

C proporciona diversos tipos fundamentales, como char, int, double, etc. y varios tipos derivados como arreglos y apuntadores; también proporciona la declaración **typedef,** que permite la definición de nuevos tipos a partir de los ya existentes o previamente definidos.

La forma de definir un nuevo tipo es como se muestra:

```
typedef     tipo_base   nuevo_tipo;
```

Ejemplos:

```
typedef     int         ENTERO;
typedef     char        CARACTER;
typedef     ENTERO      VECTOR[10];
typedef     CARACTER    *STRING;
typedef     VECTOR      MATRIZ[10];
```

Estos tipos definidos se pueden utilizar para la definición de variables, parametros de funciones, tipo del valor de retorno de las funciones, etc., de la misma forma en como se utilizan los tipos estándar:

```
STRING    lista[N];        /* Se define un arreglo de apuntadores a caracter */
MATRIZ    a,b,c;           /* Se definen tres arreglos bidimensionales de enteros */
```

Las definiciones de tipo permiten la documentación de programas. Normalmente las definiciones de tipo junto con los prototipos y las definiciones de símbolos con #define se colocan en archivos de encabezados (archivos con extensión .h).

Cuando los tipos de datos no tienen las mismas características en diferentes ambientes, el empleo de typedef permite que los programas sean transportables. Por ejemplo: el tipo int en los sistemas UNIX es de cuatro bytes y en otros es de dos, si la aplicación requiere del manejo de enteros en cuatro bytes se podría definir un tipo ENTERO (este se utilizaría para la definición de variables) y dependiendo del sistema, este tipo estaría asociado a un int o a un long. No se tendrían que modificar todas las definiciones de variables, solamente definir el tipo de sistema:

```
#ifdef      UNIX

typedef     int    ENTERO;

#endif

#ifdef      MS_DOS

typedef     long   ENTERO;

#endif
```

ESTRUCTURAS

Las estructuras permiten la representación de elementos cuyos componentes son de diferentes tipos.

Por ejemplo, suponga que se quiere representar mediante una estructura de datos la información de un empleado:

- Número de cuenta (ej. 1287BDG)
- Nombre
- Edad
- Dirección
- Teléfono
- Sexo (M, F)
- Tipo (V = vendedor, A = administrativo,
  T = técnico, D = directivo)
- Salario

La información del empleado contiene atributos de diferentes tipos, el nombre es un arreglo de caracteres, el sexo es un solo caracter, el salario es float, la edad es de tipo entero, etc.

La definición de la estructura para manejar la información de un empleado se muestra a continuación:

```
struct emp                                  /* Identificador para el nuevo tipo */
{
        char        noCta[8],               /* Se definen cada uno de los atributos */
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad;
        float       salario;
};
```

Una vez definida la estructura, esta puede ser usada como tipo para la definición de variables:

```
struct emp  lista[N];
struct emp  empleado1;
```

Se puede hacer uso de typedef para hacer una definición de tipo más manejable:

```
typedef struct          /* El identificador de estructura (emp) en este caso es opcional */
{                   /* ya que se utilizara el identificador de tipo EMPLEADO y no struct emp*/
        char        noCta[8],
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad;
        float       salario;
} EMPLEADO;
```

Con la definición anterior, se podrían definir variables de la siguiente forma:

```
EMPLEADO  lista[N];
EMPLEADO  empleado1;
```

En la definición de el tipo EMPLEADO, la etiqueta emp después de struct es opcional, cuando esta se coloca, permite que se pueda utilizar también el identificador de tipo *struct emp* para la definición de variables. Es necesaria la etiqueta cuando la estructura tiene elementos del tipo que se esta definiendo.

Los nombres de los miembros de la estructura son únicos dentro de ella.

La forma de accesar los elementos de una estructura es por medio del operador "." (punto):

```
empleado1.edad = 27;
empleado1.sexo = 'F';
strcpy(empleado1.nombre,"Jessica Briseño");
```

Una expresión de la forma:

variable_estructura.atributo

se utiliza como cualquier expresión del tipo asociado al atributo.

Son válidas las expresiones de asignación de estructuras. En este caso, se copia el contenido de cada uno de los atributos, por ejemplo, si *empleado* y *empleado1* son estructuras del mismo tipo, la siguiente sentencia es válida:

```
empleado1 = empleado;
```

sin embargo, si el campo *nombre* en la estructura *struct emp* se hubiera definido como:

```
...
char  *nombre;
...
```

la asignación de un valor para el atributo *nombre* se tendría que realizar utilizando memoria dinámica:

```
empleado.nombre = (char *)malloc(strlen("Jessica Briseño")+1);
strcpy(empleado.nombre,"Jessica Briseño");
```

en este caso la asignación

```
empleado1 = empleado
```

provocaría que los atributos *nombre* de ambas estructuras apuntaran a la misma dirección de memoria, de tal forma que si se ejecuta la siguiente instrucción:

```
free(empleado.nombre)
```

el atributo *nombre* de la estructura *empleado1* direcciona memoria marcada como disponible, la cuál puede ser adquirida en cualquier momento dando como resultado comportamientos indeseables de las variables.

Ejemplo:

```
/*
**      Programa 6_1
**
**      Programa que implementa el manejo de números complejos con el uso de
**      estructuras.
**
*/

#include <stdio.h>

typedef struct
{
        float   real,
                imaginaria;
} COMPLEJO;

COMPLEJO  suma(COMPLEJO, COMPLEJO);
COMPLEJO  resta(COMPLEJO, COMPLEJO);

main()
{
        COMPLEJO  a,b,c;

        printf("\n\nProporciona dos numeros complejos (parte_real  parte_imaginaria):\n");
        printf("1=> ");
        scanf("%f",&a.real);
        scanf("%f",&a.imaginaria);
        printf("2=> ");
        scanf("%f",&b.real);
        scanf("%f",&b.imaginaria);
        c = suma(a,b);
        printf("\nLa suma de los numeros:\n%5.2f + %5.2fi\n%5.2f + %5.2fi\nes:\n\n"
                "%5.2f + %5.2fi\n", a.real, a.imaginaria, b.real, b.imaginaria,
                c.real, c.imaginaria);
        c = resta(a,b);
        printf("\n\ny la resta:\n\n%5.2f + %5.2fi\n", c.real, c.imaginaria);
}
```

```
/*
**      suma
**
**      Funcion que implementa la suma de numeros complejos.
**
**      PARAMETROS:
**              x               primer numero complejo
**              y               segundo numero complejo
**
**      RETURN:
**              suma de los dos numeros complejos
*/
COMPLEJO suma(COMPLEJO x, COMPLEJO y)
{
        COMPLEJO z;

        z.real = x.real + y.real;
        z.imaginaria = x.imaginaria + y.imaginaria;
        return z;
}
/*
**      resta
**
**      Funcion que implementa la resta de numeros complejos.
**
**      PARAMETROS:
**              x               primer numero complejo
**              y               segundo numero complejo
**
**      RETURN:
**              resta de los dos numeros complejos
*/
COMPLEJO resta(COMPLEJO x, COMPLEJO y)
{
        COMPLEJO z;

        z.real = x.real - y.real;
        z.imaginaria = x.imaginaria - y.imaginaria;
        return z;
}
```

Inicialización de estructuras

Las estructuras pueden ser inicializadas de una forma muy parecida a como se inicializan los arreglos:

```
EMPLEADO empleado = { "811CAFA",
                      "C. Jéssica Briseño C.",
                      "Norte 86B 4729",
                      "379-00-00",
                      'F',
                      'D',
                      24,
                      6500 };
```

Arreglos de estructuras

El lenguaje C permite la creación de arreglos de elementos cuyos tipos pueden ser cualquiera previamente definido, por ejemplo, se pueden definir arreglos de estructuras:

```
EMPLEADO        lista[N];
```

Así, para poder accesar el salario del primer empleado en la lista se tendría que hacer referencia a lista[0].salario

LABORATORIO

1. Implemente la agenda del capítulo anterior (programa 5_7 del capítulo "ARREGLOS Y APUNTADORES") con un arreglo de estructuras, de tal forma que se almacene el nombre , dirección y telefono de sus amigos.

2. Modifique el programa de agenda para que se puedan hacer consultas de telefonos de los amigos.

UNIONES

Una unión define a un conjunto de elementos de diferentes tipos que comparten el espacio de memoria asignado a la union.

Una unión al igual que una estructura, es un tipo derivado. La sintaxis de las uniones es semejante a la de las estructuras, solamente que en lugar de especificar el tipo *struct* se utiliza *union.*

Como los atributos de la union comparten la memoria asignada, el compilador asigna una porción de almacenamiento que pueda acomodar al más grande de los atributos especificados.

La notación para accesar a un atributo de una unión es idéntica a la que emplean las estructuras.

Como la union tiene asignado solo un espacio de memoria, el valor que contiene es interpretado dependiendo del atributo por el que se accese, elegir el correcto es responsabilidad del programador.

Ejemplo:

```
union numero {
        int             entero;
        double          real;
}

struct          numero          num;

num.real = 13.56;
printf("%f",num.real);          /* Al accesar la union con real obtendremos como valor 13.56 */
printf("%d",num.entero); /* Al accesar la union con enetro el valor obtenido es desconocido */
```

en este ejemplo, se crea una variable del tipo de la union definida. A la union se le asigna el espacio de memoria necesario para almacenar el atributo que ocupa mas espacio, en este caso *real*. El espacio de memoria de la union se puede manejar como de tipo *int* o como de tipo *double*, en el ejemplo, se accesa como *double*. Si hacemos referencia a ese espacio mediante el atributo *entero*, despues de la asignacion, el espacio de memoria se manejara como de tipo *int* y el valor obtenido es, en general, impredecible.

Ejemplo:

Suponga que se quiere representar la información de los empleados de una empresa mediante una estructura de datos. En la empresa existen diferentes tipos de empleados (vendedor, administrativo, directivo y técnico) y para el calculo de la nómina es importante considerar todos los puntos que influyen en el cálculo de las percepciones mensuales.

Los técnicos y directivos reciben un sueldo mensual, un bono adicional y un pago mensual para gastos de automovil; los vendedores perciben además de su sueldo base comisiones y premios, a los administrativos se les paga las horas extras trabajadas.

Para todos los empleados se tiene que almacenar su numero de cuenta, nombre, sexo, direccion, telefono, edad y año de ingreso.

Los tipos utilizados serían los siguientes:

```
/* Estructura para definir la informacion del sueldo de un empleado de confianza */

typedef  struct
{
        float   sueldoBase;
        float   bono;
        float   gastosAuto;
} CONFIANZA;


/* Estructura para definir la informacion del sueldo de un empleado administrativo */

typedef struct
{
        float   sueldoBase;
        int     horasExt;
} ADMON;


/* Estructura para definir la informacion del sueldo de un vendedor */

typedef struct
{
        float   sueldoBase;
        float   comision;
        float   premio;
} VENDEDOR ;


/* Union  para definir la informacion del sueldo de un empleado de la empresa */

typedef union
{
        CONFIANZA       confianza;
        ADMON           admon;
        VENDEDOR        vendedor;
} SALARIO;
```

La estructura EMPLEADO se definiria entonces, de la siguiente forma:

```
typedef struct
{
        char        noCta[8],
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad,
                    anioIngreso;
        SALARIO     salario;
} EMPLEADO;
```

El atributo tipo es de gran importancia, ya que sirve para determinar como accesar la union:

```
EMPLEADO        emp;

        ...

if (emp.tipo = 'V')                                    /* Si se trata de un vendedor */
        sueldoMensual = emp.salario.vendedor.sueldoBase +
                        emp.salario.vendedor.comision + emp.salario.vendedor.premio;
        ...
```

LABORATORIO (OPCIONAL)

1. Modifique el programa agenda del laboratorio anterior, para que cuando se trate de registrar clientes se almacene ademas teléfono de oficina, fax y nombre de la empresa que representa; para los amigos hay que almacenar solamente el telefono de su casa y la fecha de cumpleaños.

Apuntadores a estructuras

Se pueden definir apuntadores a estructuras para poder referenciarlas indirectamente.

Desde un apuntador también se pueden accesar los atributos de una estructura.

Los apuntadores a estructuras se pueden utilizar para manejar paso de parámetros por referencia cuando se utilizan estructuras como parámetros.

Los apuntadores de estructuras son la base de la implementación más eficiente de estructuras de datos como pilas, listas lineales, gráficas y árboles.

La forma natural de accesar los atributos de una estructura por medio de un apuntador es un poco confusa:

```
EMPLEADO *p;
...

(*p).tipo = 'D';    /* Son necesarios los parentesis, por la presedencia de los operadores */
```

Una forma alterna, y la más utilizada es con el operador "->":

```
EMPLEADO *p;
...

p->tipo = 'D';
```

Ejemplo:

```
/*
**      lab6_2.c
**
**      Archivo main del programa de nomina.
*/
/*
**      Programa 6_2
**
**      Este programa genera un reporte de sueldos de una empresa. En la empresa
**      existen tres tipos de empleados: de confianza, administrativos y vendedores. La
**      forma de calcular el sueldo varia dependiendo del tipo de empleado.
**
*/
#include    <stdio.h>
#include    "emp.h"

main()
{
        EMPLEADO nomina[MAX_EMP];
        float       pagoEmp,
                    totalNomina=0;
        int         i, numEmp;

        printf("Cuantos empleados forman la nomina? ");
        scanf("%d",&numEmp);
        fflush(stdin);
        for(i=0 ; i < numEmp ; i++)
                leeDatos(&nomina[i]);
        printf("Reporte de NOMINA\n\n");
        printf("No. Cta.\tEmpleado\t\tPago\n\n");
        for(i=0 ; i < numEmp ; i++)
        {
                pagoEmp = calculaPago(nomina[i]);
                totalNomina += pagoEmp;
                printf("%-12.12s\t%-12.12s\t%10.2f\n",nomina[i].noCta,
                        nomina[i].nombre, pagoEmp);
        }
        printf("\n\n\t\tTOTAL: %12.2f\n",totalNomina);
```

```
}

/*
**      emp.c
**
**      Archivo de definicion de funciones para calculo de los pagos de empleados.
*/
#include      <stdio.h>
#include      "emp.h"

/*
**      leeDatos
**
**      Funcion que lee los datos de un empleado.
**
**      PARAMETROS:
**              empleado            empleado para el que se leera su informacion.
*/
void    leeDatos(EMPLEADO      *empleado)
{
        /*
        **      Datos comunes a todos los empleados:
        */
        printf("No. Cta   =>");
        gets(empleado->noCta);
        printf("Nombre    =>");
        gets(empleado->nombre);
        printf("Direccion =>");
        gets(empleado->direccion);
        printf("Tel.      =>");
        gets(empleado->tel);
        printf("Sexo      =>");
        empleado->sexo = getchar();
        printf("Edad      =>");
        scanf("%d",&empleado->edad);
        printf("A&o ingreso=>");
        scanf("%d",&empleado->anioIngreso);
        fflush(stdin);
        printf("Tipo (C confianza, A administrativo, V vendedor) =>");
        empleado->tipo = getchar();
```

```
fflush(stdin);
```

```
        /*
        **      Datos particulares dependiendo del tipo de empleado.
        */
        if(empleado->tipo == 'C' || empleado->tipo == 'c')
        {
                printf("Sueldo    =>");
                scanf("%f",&empleado->salario.confianza.sueldoBase);
                printf("Bono      =>");
                scanf("%f",&empleado->salario.confianza.bono);
                empleado->salario.confianza.gastosAuto = GASTOS_AUTO;
                fflush(stdin);
        }
        else if (empleado->tipo == 'A' || empleado->tipo == 'a')
        {
                printf("Sueldo    =>");
                scanf("%f",&empleado->salario.admon.sueldoBase);
                printf("Horas Ext. =>");
                scanf("%d",&empleado->salario.admon.horasExt);
                fflush(stdin);
        }
        else
        {       printf("Sueldo    =>");
                scanf("%f",&empleado->salario.vendedor.sueldoBase);
                printf("Comisiones =>");
                scanf("%d",&empleado->salario.vendedor.comision);
                printf("Premios   =>");
                scanf("%d",&empleado->salario.vendedor.premio);
                fflush(stdin);
        }
}
```

```
/*
**      calculaPago
**
**      Funcion encargada de calcular el sueldo de un empleado en base a sus datos de
**      nomina. A los empleados de confianza se les descuenta el 15% por concepto
**      de impuesto sobre la suma que resulte de su sueldo base mas su ayuda para
**       gastos de auto, su bono no tiene ningun descuento. A los administrativos se les
**      descuenta solamente el 8% sobre su sueldo base y cada hora extra trabajada se
**      paga de acuerdo a una tarifa establecida con el sindicato. A los vendedores se les
**      paga sueldo base, comisiones y premios, pero todos estos conceptos causan 10%
**      de impuesto retenido.
**
**      PARAMETROS
**              empleado            informacion del empleado
**
**      RETURN:
**              pago mensual del empleado
*/
float   calculaPago(EMPLEADO  empleado)
{
        float   pago;

        if(empleado.tipo == 'C' || empleado.tipo == 'c')
                pago = (empleado.salario.confianza.sueldoBase +
                        empleado.salario.confianza.gastosAuto) * 0.85 +
                        empleado.salario.confianza.bono;

        else if(empleado.tipo == 'A' || empleado.tipo == 'a')
                pago = (empleado.salario.admon.sueldoBase +
                    empleado.salario.admon.horasExt * PAGO_HORA_EXT) * 0.92;

        else
                pago = (empleado.salario.vendedor.sueldoBase +
                        empleado.salario.vendedor.comision +
                        empleado.salario.vendedor.premio) * 0.9;
        return pago;
}
```

```
/*
**      emp.h
**
**      Archivo de definicion de tipos, prototipos y simbolos para las funciones de calculo
**      de la nomina de empleados.
*/

#ifndef     EMP_H
#define     EMP_H

/*
**      Estructura que define los datos particulares de un empleado de confianza.
*/
typedef  struct
{
        float    sueldoBase;
        float    bono;
        float    gastosAuto;
} CONFIANZA;

/*
**      Estructura que define los datos particulares de un empleado administrativo.
*/
typedef struct
{
        float    sueldoBase;
        int  horasExt;
} ADMON;

/*
**      Estructura que define los datos particulares de un vendedor.
*/
typedef struct
{
        float    sueldoBase;
        float    comision;
        float    premio;
} VENDEDOR ;
```

```
/*
**      Union que define la informacion del salario de un empleado, puede ser
**      informacion de un vendedor, de un administrativo o de un empleado de
**      confianza.
*/
typedef union
{
        CONFIANZA     confianza;
        ADMON         admon;
        VENDEDOR      vendedor;
} SALARIO;

/*
**      Estructura para definir la informacion de un empleado.
*/
typedef struct
{
    char        noCta[8],
                nombre[35],
                direccion[35],
                tel[10],
                sexo,
                tipo;
    int         edad,
                anioIngreso;
    SALARIO         salario;
} EMPLEADO;

#define     MAX_EMP         20          /* Numero maximo de empleados */
#define     GASTOS_AUTO     550         /* Ayuda para gastos de automovil */
#define     PAGO_HORA_EXT   50          /* Pago por hora extra para los empleados administrativos */

/*
**      Prototipos
*/
void    leeDatos(EMPLEADO *);
float   calculaPago(EMPLEADO);
void    leeFloat(int *);

#endif
```

RESUMEN DE OPERADORES

Operador	Asociatividad
() [] ->	izquierda a derecha
! ~ ++ -- + - * & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
>> <<	izquierda a derecha
> >= < <=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= &= *= /= %= ^=	derecha a izquierda
|= <<= >>=	
, (operador coma)	izquierda a derecha

LISTAS LINEALES

LISTA

Las listas son estructuras de datos que dan mayor flexibilidad de programación a los usuarios, con un ahorro considerable de memoria por las operaciones que pueden practicarse sobre ella.

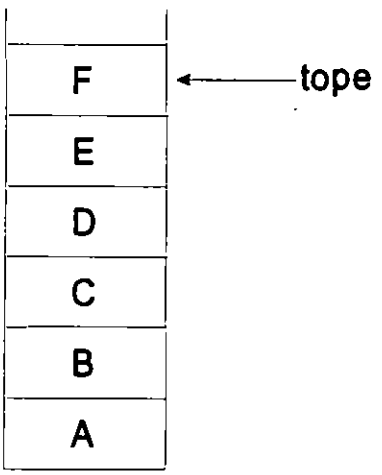
Una lista es una estructura de datos que tiene un número variable de nodos.

Una lista lineal, es una lista cuyos nodos están ordenados por un solo criterio, en donde el último y el primer nodo no tienen sucesor y antecesor respectivamente.

PILA

Una pila o stack es una estructura de datos lineal, en la cual las operaciones se realizan por uno de los extremos de la lista.

Una pila se puede representar mediante la siguiente figura:

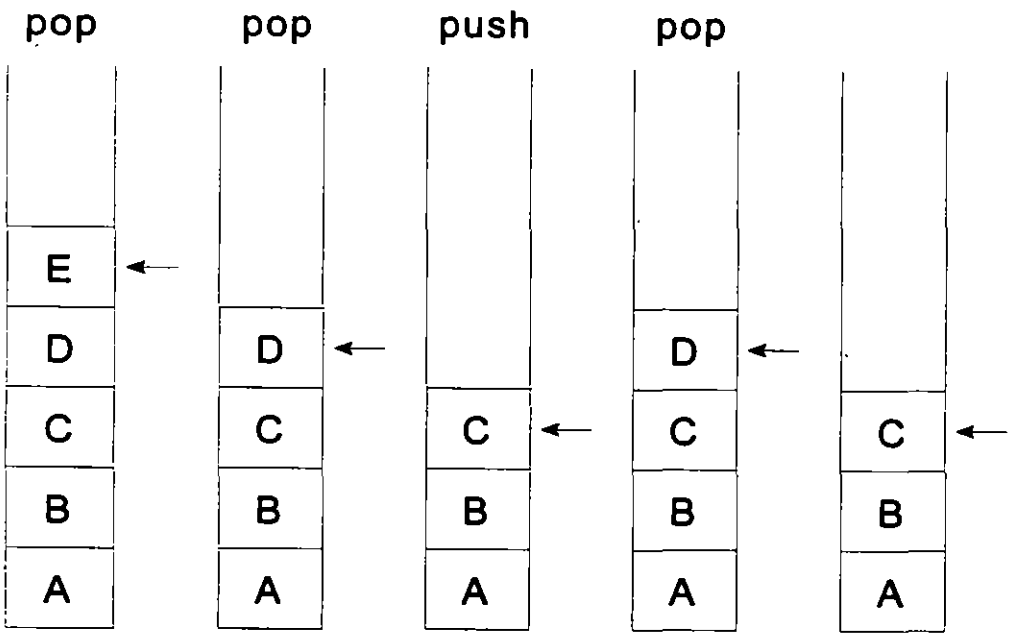


Uno de los extremos de la pila se designa como el tope de la misma y es por donde se colocan nuevos elementos o se retiran.

En el caso de que se agreguen nuevos elementos a la pila, el tope se moverá hacia arriba para indicar al último elemento en entrar a la pila.

En el caso de que se retire un elemento de la pila el tope se mueve hacia abajo, para indicar hacia el nuevo elemento que se encuentra en el extremo de la pila.

En la siguiente figura se muestra el comportamiento de la pila al insertar y borrar elementos de ella.



Por la forma en que se agregan y retiran elementos de la pila, el método ha sido llamado LIFO (las input first output). esto significa que solamente puede ser retirado de la pila el último elemento agregado.

Las operaciones que se llevan a cabo sobre una pila se conocen como **push** (para insertar) y **pop** (para borrar un elemento).

Cuando la pila contiene un solo elemento y se lleva a cabo una operación de pop, la pila resultante no contiene elementos y se llama **pila vacía**.

Aunque la operación push es aplicable teóricamente en cualquier momento, no ocurre lo mismo con la operación pop, la cual no puede aplicarse a una pila vacía.

Representación de pilas en C

Antes de programar la solución de un problema que use una pila, debe decidirse como representarla mediante las estructuras de datos existentes en nuestro lenguaje de programación.

Como se verá, hay varias maneras de representar una pila en el lenguaje C. Consideremos ahora la más sencilla: un *arreglo*.

Sin embargo, una pila y un arreglo son dos cosas diferentes. El número de elementos de una arreglo es fijo y se asigna en la definición de este. Por otra parte, una pila es un objeto dinámico cuyo tamaño cambia constantemente mediante las operaciones push y pop.

Si se quiere implementar una pila con un arreglo se deberá definir un arreglo lo bastante grande para admitir el tamaño máximo de la pila. Es necesario otro elemento para guarde la posición del elemento tope de la pila.

Por lo tanto puede declararse una pila, como una estructura que contiene dos elementos: una arreglo para guardar los elementos de la pila y un entero para guardar la posición del elemento tope de la pila:

```
# define STACKSIZE      100

typedef     struct {
                int     tope;
                int     elementos[STACKSIZE];
}       STACK;
```

Para definir una pila:

```
STACK       s;
```

Las funciones push y pop se definirían de la siguiente forma:

```
void push(STACK *s, int dato) {

if ( s->tope == STACKSIZE )
        printf("Pila llena");
else
        s->elementos[s->tope++] = dato;
}


int  pop(STACK *s) {

if ( s->tope == 0 )
        printf("Pila vacía");
else
        return s->elementos[s->tope--];
}
```

Para que no existiera restricción en cuanto al tipo de elementos en la pila se podría definir de la siguiente forma:

```
# define STACKSIZE      100

typedef       union {
        int      intEle;
        float   floatEle;
        char    charEle;
} ELEMENTO;


typedef       struct {
              int     tope;
              ELEMENTO elementos[STACKSIZE];
}       STACK;
```

Pilas dinámicas

Una de las características principales de una pila es que su tamaño es dinámico, es decir su tamaño no esta previamente definido.

Para definir una pila dinámica utilizaremos la siguiente estructura:

```
typedef     struct s {
            int     dato;
            struct s *liga;
} ELEMENTO;

typedef  ELEMENTO       *STACK;
```

De esta forma se puede definir una pila como:

```
STACK       pila= NULL;
```

Inicialmente la pila esta vacía y el único espacio en memoria ocupado es el del apuntador.

Para implementar las funciones push y pop se deben considerar los siguientes puntos:

- La pila esta vacía cuando la variable de tipo STACK tiene como valor NULL.
- Cuando se lleva a cabo la operación pop, se deberá liberar la memoria ocupada por el elemento saliente.
- Cuando se lleva a cabo la operación push se deberá alojar memoria para el elemento que se va a colocar en el tope de la pila.
- Idealmente no existe límite en cuanto al número de elementos que pueden entrar a la pila.

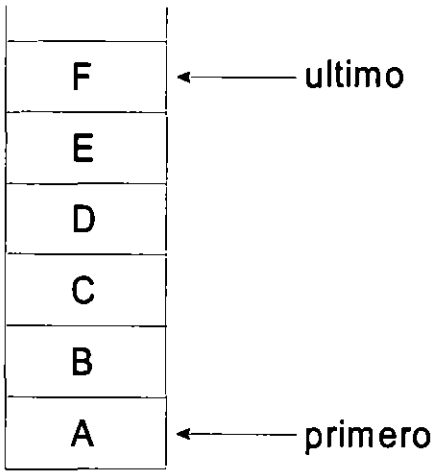
La implementación de estas funciones sería:

```
void push(STACK *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
aux->liga = *s;
*s = aux;
}


int  pop(STACK *s) {

int          dato;
ELEMENTO  *aux;

if ( *s == NULL )
        printf("Pila vacía");
else {
        dato = (*s)->dato;
        aux = *s;
        *s = (*s)->liga;
        free(aux);
        return dato;
}
}
```

COLA

Una cola es una estructura de datos lineal, en la cual la operación de inserción de un elemento se realiza por uno de los extremos de la lista y la extracción por el otro.

Una cola se puede representar mediante la siguiente figura:

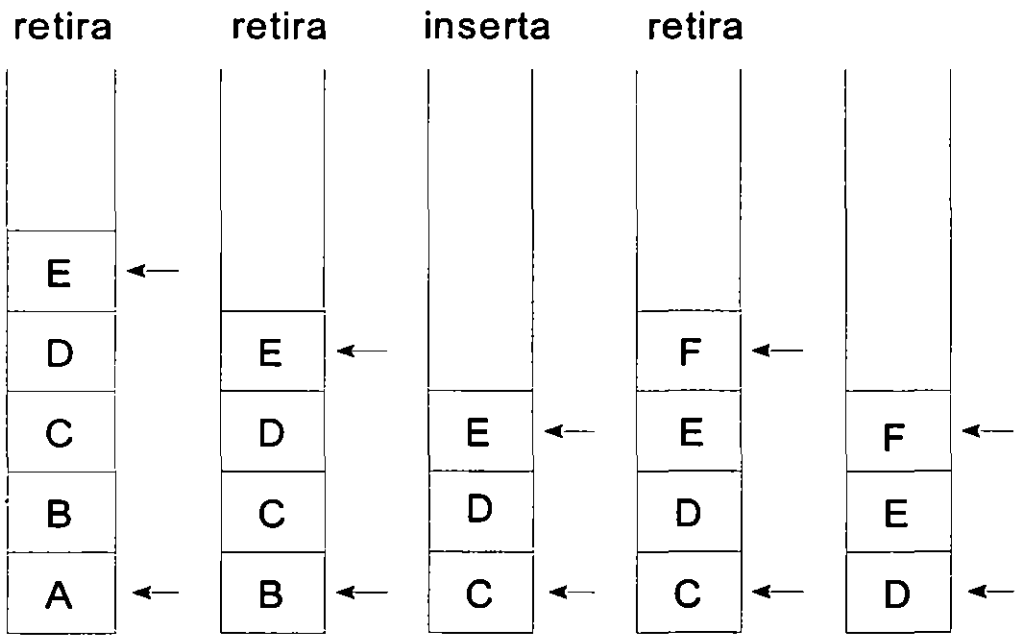


Uno de los extremos de la cola se designa como el último de la misma y es por donde se colocan nuevos elementos.

En el caso de que se agreguen nuevos elementos a la cola, este elemento será el último de la cola.

En el caso de que se retire un elemento de la cola, este elemento será el que entro antes que todos los elementos actuales de la cola. Al retirar al primer elemento de la cola, se mueven hacia abajo los demás elementos.

En la siguiente figura se muestra el comportamiento de la cola al insertar y borrar elementos de ella.



Por la forma en que se agregan y retiran elementos de la cola, el método ha sido llamado FIFO (first input first output). esto significa que solamente puede ser retirado de la cola el primer elemento agregado.

Las operaciones que se llevan a cabo sobre una cola se conocen como **insertar** y **retirar**.

Al igual que con la pila, las formas de representar una cola son muchas; sin embargo debido a que una de las características principales de una cola es que su tamaño es dinámico, consideraremos únicamente esta forma de implementación.

Para definir una cola dinámica utilizaremos la misma estructura que para la pila:

```
typedef     struct s {
            int     dato;
            struct s *liga;
} ELEMENTO;

typedef  ELEMENTO      *COLA;
```

De esta forma se puede definir una cola como:

```
COLA        cola= NULL;
```

Inicialmente la cola esta vacía y el único espacio en memoria ocupado es el del apuntador.

Para implementar las funciones insertar y retirar se deben considerar los siguientes puntos:

- La cola esta vacía cuando la variable de tipo COLA tiene como valor NULL.

- Cuando se lleva a cabo la operación retirar, se deberá liberar la memoria ocupada por el elemento saliente.

- Cuando se lleva a cabo la operación insertar se deberá alojar memoria para el elemento que se va a colocar en uno de los extremos de la cola.

- Idealmente no existe límite en cuanto al número de elementos que pueden entrar a la cola.

La implementación de las funciones para manejo de colas se muestra a continuación:

```
void inserta(COLA *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
aux->liga = *s;
*s = aux;
}


int  retira(COLA *s) {

int          dato;
ELEMENTO  *aux;

if ( *s == NULL )
        printf("Cola vacía");
else {
        aux = *s;
        while ( aux->liga != NULL )
                aux = aux->liga;
        dato = aux->dato;
        if ( aux == *s)
                *s = NULL;
        free(aux);
        return dato;
}
}
```

LISTAS NO LINEALES
ARBOLES

En el capítulo anterior se expuso que una lista lineal es una estructura de datos que expresa las relaciones entre los nodos por un solo criterio o en una sola dimensión.

Las listas no lineales son estructuras de datos más complejas, cuyas relaciones entre sus nodos son en más de una dimensión.

Una de las listas no lineales más utilizadas, principalmente en la implementación de sistemas operativos y DBMS's, son los árboles.

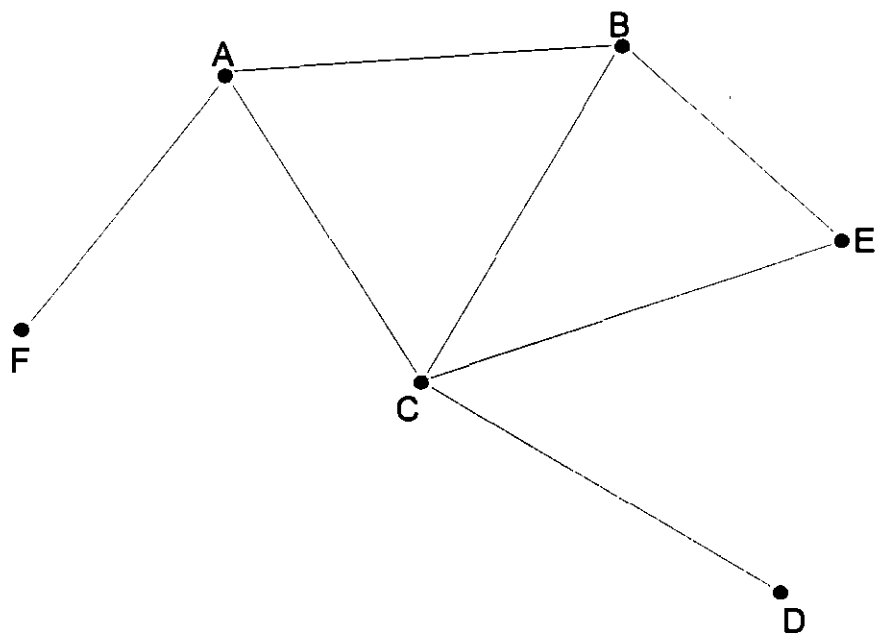
ARBOLES

Para poder definir el concepto de un árbol, tendremos que hacer referencia primero a lo que es una gráfica.

Una gráfica es un conjunto de pares ordenados:

$$G = \{ (a,b), (a,c), (c,e), (b,e), (c,d), (a,f) \}$$

representada de la siguiente forma:



Arco dirigido

Si en un par ordenado (a,b) es importante considerar que a es el nodo inicial y b el nodo final, estaremos hablando de un arco dirigido.

Grado externo de un nodo

El grado externo de un nodo u es el número de arcos que salen de él.

Grado interno de un nodo

El grado externo de un nodo u es el número de arcos que llegan a él.

Trayectoria

Si en una gráfica con arcos dirigidos ciertos pares ordenados pueden ser colocados en una secuencia de la forma:

$$(a_1, a_2), (a_2, a_3), (a_3, a_4), \ldots, (a_{n-1}, a_n)$$

el conjunto de arcos es llamado una trayectoria desde a_1 hasta a_n. Si $a_1 = a_n$, la trayectoria es un ciclo.

Cuando los arcos de la secuencia son distintos, la trayectoria es simple. Si los arcos son distintos y contienen a todos los nodos de la gráfica, la trayectoria es hamiltoniana.

La longitud de una trayectoria es el número de arcos que la componen.

Lazo o loop

Un arco que une un vértice consigo mismo se llama lazo. La dirección de una lazo no tiene ningún significado.

Definición de Arbol

Con los conceptos anteriores podemos definir un árbol:

Un árbol es una gráfica en la que:

- El número de nodos es igual al número de arcos más uno.
- Todos los nodos son de grado interno uno, excepto un nodo llamado raíz, de grado cero.
- No hay ciclos.
- Cualquier trayectoria es simple.
- Entre cualquier par de nodos solo hay una trayectoria.
- Cualquier arco es un arco de desconección.

En la terminología que se emplea para el estudio de los árboles encontraremos entre otros, los términos siguientes:

Se define como grado o grado externo de un nodo al número de sus subárboles.

Una hoja o nodo terminal es un nodo de grado cero.

Un nodo ramal es un nodo de grado mayor de cero.

El nivel de un nodo es el nivel de su antecesor directo más uno. El nivel de la raíz es uno.

Es frecuente que los nodos de un árbol reciban nombres, tales como: el nodo a es padre

de b, c, d, si existe un arco de a a b, uno de a a c y otro de a a d.

ARBOLES BINARIOS

Los árboles binarios son aquellos cuyos nodos tienen un grado externo menor o igual a dos.

Los árboles binarios tienen muchas aplicaciones en sistemas operativos y bases de datos.

Las operaciones que se definen para un árbol binario son: inserción y recorrido.

El recorrido de un árbol binario es el procedimiento de visitar cada uno de sus nodos, con el objeto de sistematizar la recuperación de la información almacenada.

Una de las formas más simples de recorrer un árbol es de la raíz hacia los nodos hoja (top-down).

El recorrido top-down de un árbol binario puede ser de tres formas diferentes:

- preorden
- inorden
- postorden

En el recorrido preorden:

- se visita la raíz
- se recorre el subárbol izquierdo
- se recorre el subárbol derecho

En el recorrido inorden:

- se recorre el subárbol izquierdo
- se visita la raíz
- se recorre el subárbol derecho

En el recorrido postorden:

- se recorre el subárbol izquierdo
- se recorre el subárbol derecho
- se visita la raíz

La forma de implementar los algoritmos de recorrido es en forma recursiva y no recursiva; sin embargo, la forma recursiva es mucho más entendible, por lo que es la que se presentará.

Para ello consideremos la siguiente estructura:

```
typedef struct x {
                    int             dato;
                    struct x  *ligaIzq,
                              *ligaDer;
}       ELEMENTO;

typedef       ELEMENTO  *ARBOL;
```

Para definir una variable:

```
ARBOL       arbol=NULL;
```

Las funciones de recorrido se muestran a continuación:

```
int preOrden(ARBOL  a) {

if ( a != NULL ) {
        printf("%d ", a->dato);
        preOrden(a->ligaIzq);
        preOrden(a->ligaDer);
}
}
```

```
int inOrden(ARBOL  a) {

if ( a != NULL ) {
        inOrden(a->ligaIzq);
        printf("%d ", a->dato);
        inOrden(a->ligaDer);
}
}


int postOrden(ARBOL  a) {

if ( a != NULL ) {
        postOrden(a->ligaIzq);
        postOrden(a->ligaDer);
        printf("%d ", a->dato);
}
}
```

LABORATORIO

Escriba un programa que ordene una secuencia de números utilizando un árbol binario.

METODOS DE ORDENAMIENTO E INSERCION

METODOS DE ORDENAMIENTO

El ordenamiento es una de las operaciones más importantes que se practica sobre una estructura de datos.

Ordenar una estructura de datos es establecer un orden de precedencia entre los elementos de la estructura, de acuerdo a uno o más campos que se seleccionen para tal fin.

Es frecuente encontrar operaciones de ordenamiento como parte de los procesos para el manejo de datos.

Ordenar un conjunto de datos puede parecer una operación trivial, pero en realidad es una operación costosa que deberá realizarse solamente cuando sea estrictamente necesario y en este caso seleccionar el método apropiado.

Consideraciones para la selección del método de ordenamiento

Para seleccionar un método de ordenamiento en una cierta situación, es aconsejable considerar los siguiente:

1. El tipo de memoria en la que se encuentran los datos. Esta puede ser de acceso directo y de alta velocidad, de acceso directo y de mediana velocidad, y de acceso secuencial.

2. Las características del sistema operativo para el manejo de archivos y de la memoria.

3. Los tiempos de acceso a los dispositivos.

4. La cantidad, el tipo y la distribución inicial de los datos.

5. La eficiencia del método. Para establecer la eficiencia de un método de ordenamiento, basta con determinar el número promedio de comparaciones y de intercambios, así como la cantidad de memoria que requiere.

Para tener una idea del comportamiento de los algoritmos de ordenamiento, los mejores métodos realizan un número de comparaciones proporcional a $n\log_2 n$, donde n es el número de elementos a ordenar.

Ordenamientos internos

Los métodos de ordenamiento que se utilizan para un conjunto de datos almacenados en una memoria de acceso directo de alta velocidad, son llamados métodos de ordenamiento interno.

Los métodos de ordenamiento interno se encuentran clasificados en métodos de selección, intercambio, inserción, distribución e intercalación de acuerdo al principio en el que se basan.

A continuación se listan los algoritmos más importantes en cada caso:

Métodos por selección:	directa repetitiva torneo heap
Métodos por intercambio:	burbuja transposición de pares y nones embudo quick
Métodos por inserción:	directa binaria de doble entrada shell

Métodos por distribución: cubetas

Métodos por intercalación: merge

Métodos por Selección: selección directa

Los métodos por selección como su nombre lo indica, seleccionan del conjunto de datos, según el criterio que se siga, al mayor o al menor de los datos y lo excluye para proceder sobre los restantes de forma similar.

El método de Selección Directa consiste en seleccionar del conjunto de datos el elemento más pequeño en valor y excluirlo del conjunto de datos para repetir el procedimiento sobre los restantes.

El número de comparaciones que este algoritmo realiza es:

$$(n_2 - n)/2$$

```
/*      Algoritmo de ordenamiento: Método de selección directa

        Intercambia el elemento iesimo de un arreglo de N
        elementos (0 < i < N) con el elemento menor del arreglo.

*/

#include <stdio.h>
#define swap(a,b)   {int t; t=a; a=b; b=t;}
#define maxN        100

void seleccion(int a[], int N) {

        int i, j, min;

        for(i=0; i<N-1; i++) {
                min = i;
                for(j= i+1; j < N; j++)
                        if(a[j] < a[min])
                                min=j;
                swap(a[min],a[i]);
        }
}


main() {
        int N = 0, i, a[maxN];

        while(scanf("%d",&a[N]) == 1)
                N++;
        seleccion(a,N);
        for(i=0; i < N; i++)
                printf("%d ",a[i]);
}
```

Métodos por Intercambio: burbuja

Los métodos por intercambio transponen o intercambian sistemáticamente pares de datos que se encuentran fuera de orden hasta que dejen de existir.

El nombre de burbuja se debe a la manera como los datos se mueven dentro del conjunto, aparentando ser burbujas en el agua subiendo a la superficie.

El algoritmo se inicia comparando las llaves n y n-1, y las intercambia si n<n-1, compara después n-1 y n-2 y las intercambia si n-1<n-2. Este procedimiento se practica hasta comparar los datos 1 y 2 y, cuando esto sucede, el dato menor ha alcanzado su posición final.

```
/*      Algoritmo de ordenamiento: Método de la burbuja

        En varias pasadas sobre un arreglo, se intercambian
        elementos adyacentes de ser necesario.
        Después de varias pasadas no se necesitan más intercambios,
        el arreglo queda ordenado.

*/

#include <stdio.h>
#define swap(a,b)   {int t; t=a; a=b; b=t;}
#define maxN        100

void bubble(int a[], int N) {

        int i, j;

        for(i=N-1; i>=0; i--)
                for(j= 1; j <= i; j++)
                        if(a[j-1] > a[j])
                                swap(a[min],a[i]);
}

main() {
        int N = 0, i, a[maxN];

        while(scanf("%d",&a[N]) == 1)
                N++;
        bubble(a,N);
        for(i=0; i < N; i++)
                printf("%d ",a[i]);
}
```

Métodos por Intercambio: quickSort

El algoritmo de quicksort es un procedimiento recursivo en el que se intercambian los datos para colocarlos en orden con respecto a uno de ellos, llamado pivote, de tal manera que a la derecha del pivote quedan los elementos mayores a el y a la izquierda los menores.

Este proceso se repite sobre la lista de datos a la derecha del pivote y sobre la lista de datos a la izquierda del pivote.

El algoritmo es muy eficiente para llaves que se encuentran aleatoriamente distribuidas.

```
/*      Algoritmo de ordenamiento: Método de QuickSort

        Este método consiste en dividir el arreglo a ordenar
        en dos subarreglos formados a partir de un elemento llamado
        pivote.

*/

#include <stdio.h>
#define swap(a,b)   {int t; t=a; a=b; b=t;}
#define maxN        100

void quickSort(int a[], int l, int r) {

        int i, j, v;

        if (r>1) {
                v=a[r];
                i=l-1;
                j=r;
                for(;;) {
                        while(a[++i] < v)
                                ;
                        while(a[--j] > v)
                                ;
                        if (i >= j)
                                break;
                        swap(a[i],a[j]);
                }
                swap(a[i],a[r]);
                quickSort(a,l,i-1);
                quickSort(a,i+1,r);
        }
}
```

```
main() {

        int N = 0,
        i,
        a[maxN+1];

        while(scanf("%d",&a[N]) == 1)
                N++;
        quickSort(a,1,N);
        for(i=0; i < N; i++)
                printf("%d ",a[i]);

}
```

Métodos por Inserción: Inserción Directa

Los métodos por inserción suponen que el conjunto de llaves se encuentra ordenado. Para una llave K que se desea agregar al conjunto, se determina el lugar que debe ocupar y se mueven los datos una posición para insertarlo en su posición correcta.

```
/*      Algoritmo de ordenamiento: Método de Insercion Directa

        Los elementos del arreglo se ordenan insertando el elemento a[i] en la posición
        adecuada dentro del conjunto ordenado de elementos a[1]...a[i-1].

*/


#include <stdio.h>
#define maxN        100


void insercion(int a[], int N, int x) {

        int i, j, v;

        for(i=0; a[i]<x && i < N; i++)
                ;
        for(j = N; j > i; j--)
                a[j] = a[j-1];
        a[i] = x;
}
```

```
main() {

        int N = 0,
        dato,
        a[maxN+1];


        while(scanf("%d",&dato) == 1)
                insercion(a, ++N, dato);

        for(i=0; i < N; i++)
                printf("%d ",a[i]);

}
```

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

L E N G U A J E " C "

(PARTE II)

ESTRUCTURAS DE DATOS

MATERIAL DIDACTICO

COMPLEMENTO

AGOSTO 1995

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg Cuauhtemoc 06000 Mexico, D.F. APDO. Postal M-2285
Telefonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

LABORATORIO

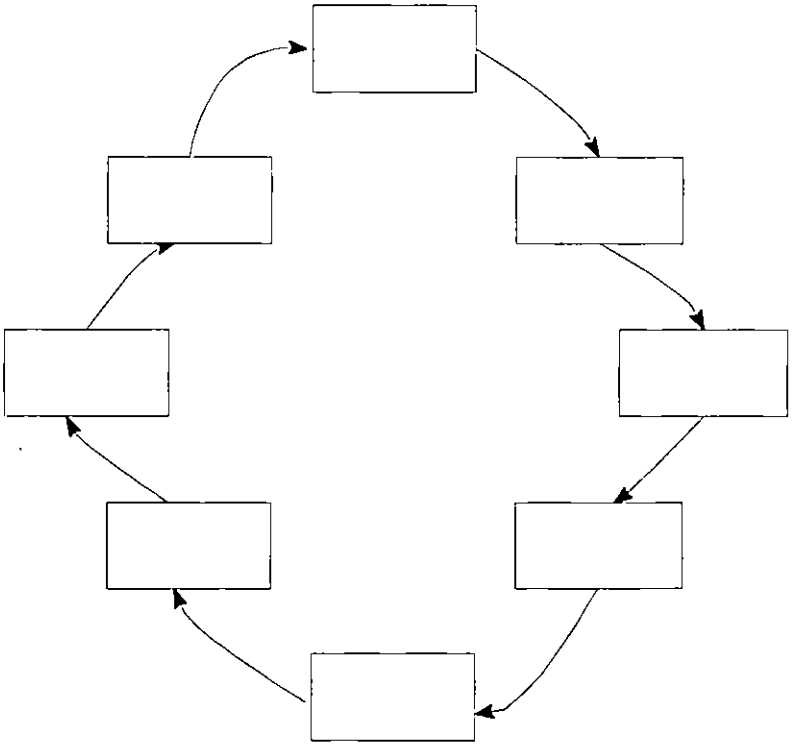
Escriba un programa que evalue una expresión en notación polaca. El programa deberá utilizar un stack para la evaluación. Las expresiones a evaluar tendrán operandos de un solo digito.

LISTA CIRCULAR

Una lista circular es aquella estructura de datos que tiene como característica fundamental un orden en el que, al último elemento le sigue el primero.

Las operaciones que se definen sobre la lista circulár son las de insertar y extraer y su comportamiento depende de su manejo puede ser como cola o como pila.

Una cola se puede representar mediante la siguiente figura:



La implementación de una lista circular, considerando la representación como la de una cola es la siguiente:

```
typedef     struct s {
            int     dato;
            struct s *liga;
} ELEMENTO;

typedef  ELEMENTO      *COLA;
```

De esta forma se puede definir una cola como:

```
COLA        cola= NULL;
```

Para implementar las funciones insertar y retirar se deben considerar los siguientes puntos:

- La cola esta vacía cuando la variable de tipo COLA tiene como valor NULL.
- Cuando la cola tiene un solo elemento este apunta así mismo.

La implementación de las funciones se presenta a continuación:

```
void inserta(COLA *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
if ( *s == NULL)
        aux->liga = aux;
else {
        aux->liga = *s;
        while ( (*s)->liga != aux->liga)
                *s = (*s)->liga;
        (*s)->liga = aux;
}
*s = aux;
}
```

```
int  extrae(COLA *s) {
int          dato;
ELEMENTO  *aux;

if ( *s == NULL )
        printf("Cola vacía");
else {
        aux = *s;
        if ( aux->liga == *s) {
                dato = aux->dato;
                free(aux);
                *s = NULL;
        } else {
                while ( aux->liga->liga != *s )
                        aux = aux->liga;
                dato = aux->liga->dato;
                free(aux->liga);
                aux->liga = *s;
        }
        return dato;
}
}
```

LABORATORIO

Escriba un programa que implemente el juego de José.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

LENGUAJE "C" (PARTE II)

ESTRUCTURA DE DATOS

MATERIAL DIDACTICO

AGOSTO, 1995

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 Mexico, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

FUNCIONES Y EL PREPROCESADOR

FUNCIONES

Las funciones son elementos que permiten el desarrollo de programas modulares. En el lenguaje C no existe el concepto de procedimiento o subrutina, todos los modulos de un programa se implementan en base a funciones.

Un programa es un conjunto de definiciones de variables y funciones. La comunicación entre funciones es por parámetros y valores regresados por las funciones, y a través de variables externas.

La función que controla la ejecución del programa se llama **main.**

Las funciones pueden presentarse en cualquier orden dentro del archivo fuente (la función main no se tiene necesariamente que definir primero), o en diferentes archivos, simpre y cuando las funciones no se dividan.

Todas las funciones se definen al mismo nivel, no se puede definir una función en otra (la definición de funciones no se puede anidar).

Las funciones pueden ser recursivas, salvo main.

La sintaxis para la definición de una función es la siguiente:

```
tipo_retorno      nombre (lista_parametros )
{

      definiciones de variables y sentencias

}
```

donde:	tipo_retorno	es el tipo asociado a el valor regresado por la función.
	nombre	identificador de la función.
	lista_parametros	lista de parametros en la siguiente forma: tipo parametro

Para cada parámetro en la lista de parámetros se debe especificar su tipo.

Por default, las funciones tienen un tipo de retorno *int*.

Para funciones que no regresan valores (por ejemplo aquellas que actuan como procedimientos), se puede especificar como tipo de retorno *void*.

Ejemplo:

```
double maximo(double x, double y)
{

      /*.Funcion que recibe dos parametros de tipo double y su valor de retorno es de tipo double */

}
```

Declaración y definición de una función

Se define una función cuando se indica su nombre, el tipo del valor de retorno, el número de parámetros que recibe y su tipo, así como las sentencias que la forman.

Se declara una función cuando únicamente se indica su nombre, el tipo del valor de retorno, el número de parámetros que recibe y su tipo. A la declaración de una función se le conoce comunmente como **prototipo**.

Una función se debe definir una vez y se puede declarar más de una vez en un programa.

Si una función no se declara o define antes de que aparezca una llamada a ella, el compilador asume que regresa un valor de tipo *int* y que el valor, tipo y número de sus parámetros corresponden a los que aparecen en esa llamada.

Ejemplo:

```
double  maximo(double, double);          /* Prototipo o declaracion de la funcion*/

main()
{
        double x=5, y=8, z;

        z = maximo(x,y);

            ...

}

double  maximo(double x, double y)       /* Definicion de la función */
 {
        double        max;
        ...

        return max;
}
```

En el ejemplo anterior, sino se especifica la declaración de la función, cuando se hace la llamada a la misma el compilador asume que el tipo del valor de retorno es int. En el momento en el que el compilador encuentra la defincion de la función genera un error indicando que la función ya habia sido definida, aunque realmente el compilador asumio una definición.

Valores de regreso

Una función puede regresar un valor, el cual puede ser utilizado en la expresión en donde se hizo la llamada como cualquier constante del tipo indicado para dicho valor.

El valor de retorno puede ser cualquier expresión indicada en una cláusula *return*.

return [(expresión)];

La cláusula return termina la ejecución de una función y pasa el control a la función que hizo la invocación.

Si se indica una expresión en la cláusula return, esta es evaluada y se regresa el resultado de esta a la función que hizo la llamada.

En la definición de la función, el valor de retorno no necesariamente debe de ser del tipo indicado para este, ya que al momento de ejecución se hace una conversión implicita (si existe) del valor de regreso al tipo definido para este. De esta forma, se puede regresar el valor de una variable entera en una función con tipo de valor de retorno double:

```
double      maximo(int a, int b)
{
      int   max;
      . . .

      return max;
}
```

Por lo tanto, la expresión del return debe de ser del mismo tipo que el especificado en el tipo de retorno, o bien, debe de poderse llevar a cabo una conversión implicita de dicha expresión al tipo del valor de retorno.

En una función pueden existir más de una cláusula return, en el caso de que no exista ninguna, la función termina al alcanzar su última sentencia y el valor de retorno es indefinido.

Ejemplo:

```
double  maximo(double x, double y)      /* Funcion que obtiene el mayor de dos numeros */
{
        if (x > y)
                return x;
        return y;
}
```

Paso de parámetros

Los **parámetros actuales** de una función son la lista de valores asociados a cada uno de los parámetros, con los que se hace una llamada a una función.

Los **parámetros formales** de una función son la lista de variables que aparecen como parámetros en la definición de la función y no guardan ninguna relación en cuanto a nombre o tipo con las variables que pudieran ser tomadas como parámetros actuales en alguna llamada a la función.

El paso de parámetros en las funciones es por valor, es decir, solamente se utiliza el valor de las expresiones indicadas como parámetros.

Cuando se hace una llamada a una función:

1. Cada expresión en la lista de parámetros actuales es evaluada (no existe un orden de evaluación).

2. Se crean variables que corresponden a los parámetros formales y los valores de los parámetros actuales se copian a estas variables.

3. Las sentencias de la función se ejecutan.

4. Si existe una cláusula return, el control del programa pasa a la función que hizo la llamada.

5. Si la cláusula return incluye una expresión, el valor de esta es convertido (si es válido) a el tipo especificado como tipo de retorno y el valor es regresado a la función que hizo la llamada.

6. Si no existe cláusula return o esta no contiene una expresión, la función regresa un valor desconocido.

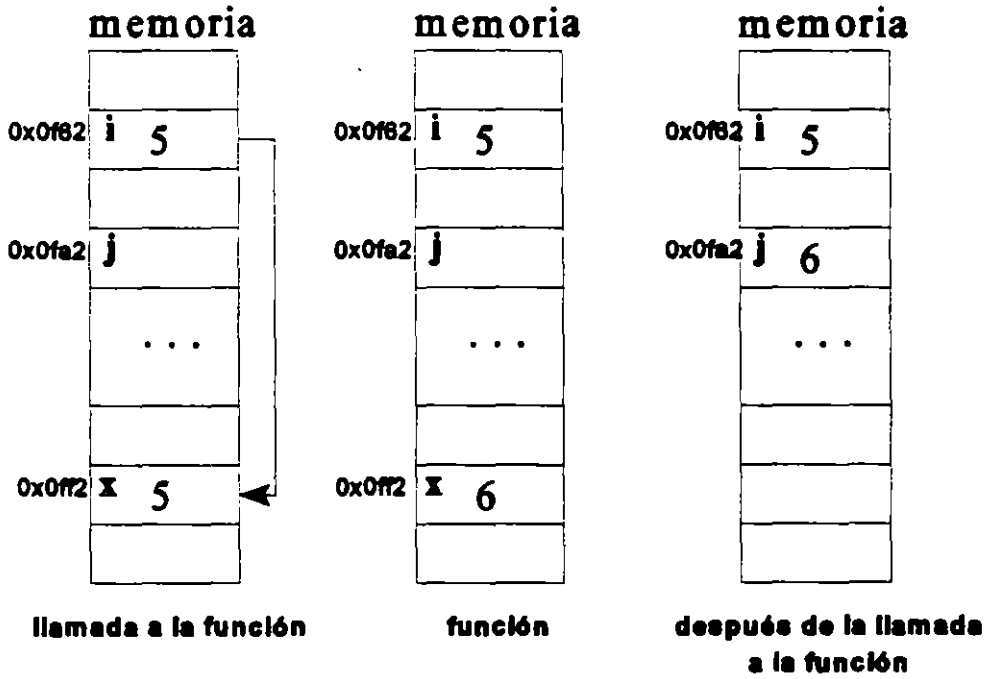
7. Las variables que representan a los parámetros formales se destruyen.

Ejemplo:

```
/*
**      Programa 4_1
**
**      Este programa muestra el comportamiento del paso de parametros por valor de
**      las funciones.
**
*/
#include      <stdio.h>

int incrementa(int);

main()
{
        int     i, j;

        i = 5;
        printf("\nValores antes de la llamada a la funcion: i=>%d, j=>%d\n",i,j);
        j = incrementa(i);
        printf("\nValores despues de la llamada a la funcion:i=>%d, j=>%d\n",i,j);
}
/*
**      incrementa
**
**      Funcion que incrementa en forma unitaria el valor pasado como parametro.
**
**      PARAMETROS
**              n       valor a incrementar
**
**      RETURN
**              valor incrementado
*/
int incrementa(int x)
{
        x++;
        printf("\nValor en la llamada a la funcion: %d\n",x);
        return x;
}
```

¿Qué salida genera el ejemplo anterior? ¿Cambia el valor de i depués de la llamada?

Cuando se hace la llamada a la función se crea una variable(x) en la cual se copia el valor del parámetro actual(i). En la función se utiliza el parámetro formal:



llamada a la función **función** **después de la llamada a la función**

Variables automáticas

Las variables automáticas son aquellas que se definen en un bloque o bien aquellas que se definen en una función.

Para estas variables se reserva espacio de memoria cada que se ejecuta el bloque o función.

Cuando termina la ejecución del bloque o función, estas variables se destruyen y se libera el espacio de memoria que ocupan.

Solo pueden ser accesadas desde el bloque o función que las define.

Los parámetros formales de una función son variables automáticas.

Se indican mediante la palabra *auto*, pero es opcional:

```
main()
{
        int         i,j,k;                  /* Variables automaticas en una funcion */
        auto int    x,y;
        ...

        for (i=0; i< N ; i++)
        {
                int     a,b;                /* Variables automaticas en un bloque */
                ...
        }
}
```

Variables externas

Las variables externas son aquellas que se definen fuera de cualquier función.

Para estas variables, se reserva espacio de memoria cuando se definen y permanecen hasta el termino del programa.

La declaracion de una variable externa indica el tipo de ella, mientras que una definición, además reserva espacio de memoria para ella.

Para la declaración de una variable externa es necesario el calificativo *extern*.

Una variable externa se puede declarar muchas veces; pero solamente debe existir una definición de ella. Esto es especialmente útil cuando se tiene un programa con varios archivos fuentes, en todo el programa solamente se deberá definir una sola vez una variable; sin embargo, en todos los archivos en donde se desee utilizar, deberá ser declarada, de lo contrario los archivos fuente no podran ser compilados.

Todas las funciones que aparecen después de la definición(en el mismo archivo) de una variable externa pueden accesar a esta.

Ejemplo:

main.c

```
int varGlobal;
main()
{
   . . .
}
```

fun1_2.c

```
extern int varGlobal;
fun1()
{
   . . .
}
fun2()
{
   . . .
}
```

fun3.c

```
fun3()
{
   . . .
}
```

En este ejemplo la variable *varGlobal* se define en el archivo *main.c*, por lo que podra ser utilizada en la función *main*. Por otra parte, la variable es declarada en el archivo *fun1_2.c*, por lo que podra ser utilizada en las funciones *fun1* y *fun2*. En el archivo *fun3.c* no existe declaración alguna para la variable, por lo tanto, no podra ser utilizada en la función *fun3*.

Variables estáticas

Las variables estáticas son automáticas a un bloque o función; pero retienen su valor entre cada llamada a la función o ejecución del bloque en donde fueron definidas.

Las variables estáticas, solamente se inicializan una vez y conservan su valor entre cada llamada a la función.

Ejemplo:

```
/*
**      Programa 4_2
**
**      Este programa calcula el mayor de N numeros proporcionados por el usuario.
**      Muestra el comportamiento de las variables estaticas.
**
*/
#include    <stdio.h>
#define     N     10

int     maximo(short, int);

main()
{
        int num,i;

        printf("Proporciona %d numeros:\n",N);
        printf("1=> ");
        scanf("%d",&num);
        maximo(0, num);
        for ( i=2; i<= N; i++)
        {
                printf("%d=> ",i);
                scanf("%d",&num);
                maximo(1, num);
        }
        printf("\nEl numero mayor fue: %d\n",maximo(2,0));
}
```

```
/*
**      maximo
**
**      Funcion que controla las operaciones sobre el numero mayor del programa.
**
**      PARAMETROS:
**              opcion      operaciones a aplicar sobre el numero mayor:
**                          0   inicializar con el valor del parametro n.
**                          1   comparar el numero mayor con el parametro n para
**                              obtener el nuevo valor mayor.
**                          2   unicamente regresa el valor mayor.
**              n           valor inicial (opcion=0) o valor a comparar con el mayor
**                          (opcion =1).
**
**      RETURN:
**              El valor mayor
**              -1 en caso de que el parametro opcion no sea valido.
*/
int     maximo(short opcion, int n)
{
        static int      max=0;

        switch (opcion)
        {
        case 0:         max = n;
                        return max;
        case 1:         max = (max > n ? max : n);
        case 2:         return max;
        default:        return -1;
        }
}
```

Inicialización

En ausencia de una inicialización explícita, las variables externas y estáticas se inicializan en cero.

En ausencia de una inicialización explícita, las variables automáticas se inicializan con valores indefinidos.

Las variables escalares se pueden inicializar cuando se definen:

```
int     x=1,
        j=5;
char    c='S';
```

Para variables externas y estáticas, el inicializador debe ser una expresión constante.

Para variables automáticas, el inicializador puede ser una constante o cualquier expresión que contenga valores previamente definidos, incluso llamadas a función:

Reglas de alcance

El alcance de una variable determina la parte del programa donde se esta se puede utilizar:

1. Para variables automáticas, el alcance de estas es el bloque o función en donde fueron definidas.

2. Las variables automáticas con el mismo nombre que estén en funciones diferentes no tienen relación. Lo mismo es válido para los parámetros formales de una función.

3. El alcance de una variable o función externa (todas las funciones son externas por default) abarca desde el lugar en donde se declaran hasta el fin del archivo fuente.

4. Si se hace referencia a una variable externa antes de su definición, o si esta definida en un archivo fuente diferente al que se esta utilizando, es obligatoria una declaración extern.

5. Para hacer llamadas a una función que se encuentra definida en un archivo fuente diferente, es conveniente hacer una declaración previa de la misma en el archivo en donde se desea hacer la llamada.

6. Cuando existe una definición de una variable externa y una automática con el mismo identificador, las referencias a través del identificador serán hacia la variable automática.

7. La declaración *static* aplicada a una variable o función externa, limita el alcance de ese objeto solamente al resto del archivo fuente.

Ejemplo:

```
/*
**      Programa 4_3
**
**      Este programa muestra el comportamiento de las reglas de alcance.
**      ¿Cual es la salida del programa? intentalo sin ejecutarlo.
**
*/
int  fun1(int);
int  fun2(int);

int     x=5,   y;

main()
{
        int x=10;

        printf("Valores al inicio del programa x=> %d, y=>%d\n",x,y);
        x = fun1(x);
        y = fun1(y);
        printf("Valores en main x=> %d, y=>%d\n",x,y);
        x = fun2(y);
        printf("Valores en main x=> %d, y=>%d\n",x,y);
}

int fun1(int  y)
{
        x += y++;
        printf("Valores en fun1 x=> %d, y=>%d\n",x,y);
        return x++;
}

int fun2(int  x)
{
        int  y=x;

        y +=
        printf  'alores en fun2 x=> %d, y=>%d\n",x,y);
        retur
}
```

Recursividad

Una función recursiva es aquella que se llama así misma directa o indirectamente. Es decir esta definida en terminos de la misma.

recursividad directa:

```
int     fun1()
{
        . . .
        fun1();
        . . .
}
```

recursividad indirecta:

```
int     fun1()
{
        . . .
        fun2();
        . . .
}

int     fun2()
{
        . . .
        fun1();
        . . .
}
```

Las funciones de C (excepto main) pueden ser recursivas.

Cada llamada recursiva reserva espacio para las variables automáticas que se definen en ella, los valores de estas variables para cada llamada se mantienen en un stack.

Las funciones recursivas deben incluir además de la llamada o llamadas recursivas sentencias para asegurar que la recursión terminará en algún momento.

Ejemplo:

```
/*
**      Programa 4_4
**
**      Este programa obtiene el factorial de un numero. El factorial de un numero es el
**      resultado de la multiplicacion del numero por el factorial del numero anterior.
**      Se toma como base que el factorial de o es 1.
*/
#include    <stdio.h>

int     factorial(int);

main()
{
        int     num;

        printf("Proporciona un numero=> ");
        scanf("%d",&num);
        printf("El factorial de %d es=> %d\n",factorial(num));
}
int     factorial(int n)
{
        return  (n == 0) ? 1: n*factorial(n-1);
}
```

El stack que se guardando los valores de las variables entre llamadas recursivas, cuando num es igual a 5 se muestra a continuación (cada renglon representa una llamada a la función):

n	return
0	1
1	1*factorial(0)
2	2*factorial(1)
3	3*factorial(2)
4	4*factorial(3)
5	5*factorial(4)

LABORATORIO

1. Haga un programa que obtenga el numero mayor y el menor de una serie de N números. Se deben utilizar dos funciones, una para obtener el mayor y otra para el menor. No utilice variables externas, tampoco estaticas. Las dos funciones deberan recibir dos parametros (los numeros a comparar) y regresar el valor mayor o menor, segun sea el caso.

2. Reconstruya su programa de empaquetamiento y desempaquetamiento del laboratorio anterior, de tal forma que se tengan 3 funciones: una que empaqueta la información proporcionada, otra que desempaqueta y otra que presenta la representación binaria del número en donde se guarda la información empaquetada.

3. Escriba una función que obtenga el número de Fibonacci n por medio de una función recursiva. Ejemplo: fib(8), debe dar como resultado 13.

NOTA: La definición de la serie de Fibonnaci se presenta en los laboratorios del capítulo 3.

EL PREPROCESADOR

C proporciona ciertas facilidades por medio de un preprocesador. El preprocesador actúa antes que el compilador ejecutando las instrucciones que comienzan con el carácter #, de tal forma que el archivo que se compila es realmente un archivo "preprocesado" que ya no incluye las instrucciones del preprocesador.

El alcance de las instrucciones que ejecuta el preprocesador es a nivel de archivo (definición de símbolos y macros son válidas solamente en el archivo en donde se hace la definición).

include

El preprocesador sustituye esta instrucción por el contenido del archivo especificado.

Sintaxis:

#include <archivo>
#include "archivo"

Cuando el nombre de archivo esta limitado por **<>**, el preprocesador busca el archivo en un directorio asignado por default (en UNIX generalmente es /usr/include y en compiladores para MS-DOS y windows este directorio es configurable), o en los directorios indicados al momento de la compilación.

Cuando el nombre de archivo esta limitado por **" "**, el preprocesador busca el archivo en el directorio de trabajo actual.

No existe restricción en cuanto al contenido del archivo.

El archivo que se sustituye puede contener instrucciones para el preprocesador, las cuales son ejecutadas una vez que se hace la sustitución.

El lenguaje C cuenta con un conjunto de librerias estandar, las cuales tienen asociadas "archivos de encabezados" (generalmente con extensión h) los cuales contienen los símbolos, tipos y prototipos utilizados con la librería en cuestion:

Archivo de encabezado	Contenido (prototipos, tipos y ctes. para)
alloc.h	alojación y liberación de memoria.
conio.h	control del display.
ctype.h	funciones de propósito general para caracteres.
dir.h	manejo de las estructuras asociadas a los directorios del sistema operativo.
errno.h	manejo de errores.
math.h	funciones matematicas.
signal.h	manejo de señales en el sistema operativo.
stdio.h	entrada/salida.
stdlib.h	funciones de propósito general.
string.h	manejo de strings.
sys/times.h	funciones para manejo de tiempos.
sys/types.h	estructuras de propósito general.

define

Esta instrucción del procesador permite la definición de símbolos y macros que son sustituidos a partir de que se da la definición.

Sintaxis:

```
#define     simbolo     token_string
#define     macro(simb1, ...,simbN)  token_string
```

#define simbolo token_string

El preprocesador sustituye cualquier ocurrencia del *simbolo* por el *token_string*. La sustitución se hace a partir de la definición solamente en el archivo en donde esta se encuentra. La sustitución no se lleva a cabo en comentarios y cadenas de caracteres encerradas por "".

Ejemplos:

```
#define     TRUE            1
#define     PI          3.1415
#define     SEG_X_DIA  (60 * 60 * 24)
```

El preprocesador solamente lleva a cabo la sustitución, si existen errores en la expresión reemplazada, estos son reportados por el compilador, en el lugar en la línea en donde se hizo la sustitución.

Las constantes simbólicas ayudan en la documentación al reemplazar lo que de otra forma sería una constante enigmática con un identificador nemónico, haciendo más portátil el programa permitiendo que se alteren en un solo lugar las constantes que pueden ser dependientes del sistema.

Una constante definida con **#define** puede revocarse con **#undef** (a partir del lugar en donde aparece, la definición ya no es válida):

#undef simbolo

#define macro(simb1, ...,simbN) token_string

Esta alternativa de la proposición **#define** sirve para la definición de macros, las cuales pueden entenderse como funciones pero relamente no lo son ya que no generan código.

No deben de existir blancos entre el identificador de macro y el primer paréntesis.

Ejemplo:

#define CUADRADO(x) ((x) * (x))

El parámetro "x" se sustituye cuando se encuentra una ocurrencia de la macro CUADRADO, en este caso si en algún lugar del archivo aparece CUADRADO(3), la sustitución se hara por ((3) * (3)).

Los paréntesis parecen excesivos, sin embargo son necesarios, suponiendo que la definición fuera la siguiente:

#define CUADRADO(x) (x * x)

si se utiliza la macro en la siguiente forma:

CUADRADO(7 + i)

la sustitución será:

(7 + i * 7 + i) lo que probablemente no generara los resultados deseados

Las macros son utilizadas para la sustitución de funciones que se pueden hacer en una línea de código:

#define MIN(x, y) ((x) < (y) ? (x) : (y))

De esta forma la macro puede servir para cualquier tipo de parámetros numéricos.

Cuando se hace la sustitución no se verifica sintaxis ni aspectos léxicos.

Ejemplo:

```
/*
**      Programa 4_5
**
**      Este programa calcula el mayor de N numeros proporcionados por el usuario.
**      Esta version proporciona una mejor documentacion gracias a la definicion de
**      simbolos.
**
*/
#define     N           10
#define     SET         0
#define     MATCH       1
#define     GET         2
#define     ERROR       -1

int     maximo(short, int);

main()
{
        int num,i;

        printf("Proporciona %d numeros:\n",N);
        printf("1=> ");
        scanf("%d",&num);
        maximo(SET, num);
```

```
        for ( i=2; i<= N; i++)
        {
                printf("%d=> ",i);
                scanf("%d",&num);
                maximo(MATCH, num);
        }
        printf("\nEl numero mayor fue: %d\n",maximo(GET,0));
}

/*
**      maximo
**
**      Funcion que controla las operaciones sobre el numero mayor del programa.
**
**      PARAMETROS:
**              opcion          operaciones a aplicar sobre el numero mayor:
**                      SET             inicializar con el valor del parametro n.
**                      MATCH           comparar el numero mayor con el parametro n para
**                                      obtener el nuevo valor mayor.
**                      GET             unicamente regresa el valor mayor.
**
**              n               valor inicial (opcion=0) o valor a comparar con el mayor
**                              (opcion =1).
**
**      RETURN:
**              El valor mayor
**              ERROR en caso de que el parametro opcion no sea valido.
*/
int     maximo(short opcion, int n)
{
        static int      max=0;

        switch (opcion)
        {
        case SET:       max = n;
                        return max;
        case MATCH:  max = (max > n ? max : n);
        case GET:       return max;
        default:        return ERROR;
        }
}
```

Compilación condicional

El preprocesador incluye algunas instrucciones que permiten llevar a cabo tareas de rastreo en los programas. También sirven para que un archivo de encabezado pueda incluirse sin ningún problema en cualquier programa, evitando conflictos por dobles definiciones.

Las líneas de control

```
#ifdef      simbolo
sentencias
#endif
```

permiten incluir las **sentencias** en el archivo a compilar, siempre y cuando se haya definido el **simbolo** anteriormente, la proposición **#ifndef** lo hace cuando el **simbolo** no ha sido definido.

Para la implementación de rastreo condicional de un programa se podría utilizar el siguiente esquema:

```
#define     DEBUG                    /* Solo se valida la definicion de DEBUG */
                                     /* por lo que no es necesario indicar el valor asociado al simbolo */
...
#ifdef      DEBUG
        printf("Valores obtenidos del proceso en el paso 1: %d, %d, %d\n",x,y,z);
#endif
```

Suponga que se crea un programa para cálculo de una nomina. Este programa utiliza una serie de funciones generales para el proceso. Las funciones estan definidas en un archivo: *funciones.c,* el cual tiene asociado su archivo de encabezados *(funciones.h)* en donde se definen los prototipos, simbolos y tipos utilizados en tales funciones.

Existe por otra parte un archivo llamado *tipos.h* en donde se definen los tipos de datos utilizados en todo el proceso de cálculo de nomina, dichas definiciones son incluidas en el archivos de encabezados funciones.h. En el archivo fuente del programa principal se incluyen los archivos *tipos.h* y *funciones.h:*

main.c

```
#include "tipos.h"
#include "funciones.h"
```

tipos.h

```
#define N 100
/* otras definiciones */
```

funciones.h

```
#include "tipos.h"

/* definiciones para funciones.c */
```

El código generado por el preprocesador antes de la sustitución de los simbolos finales sería:

```
#define N 100                          /* Codigo incluido por tipos.h*/
/* otras definiciones */

#define N 100                          /* Codigo incluido por funciones.h */
/* otras definiciones */

/* definiciones para funciones.c */
```

Para evitar dobles definiciones, principalmente de tipos, se pueden utilizar las sentencias **#ifdef** y **#ifndef** en *tipos.h:*

```
#ifndef     _TIPOS_H    /* Las instrucciones solo se incluyen cuando _TIPOS_H no ha sido */
#define     _TIPOS_H    /* definido, lo cual solo se da una sola vez, ya que en el mismo */
                        /* bloque de ifndef-endif se define */

#define     N     100
/* otras definiciones */

#endif
```

División de un programa en varios archivos

Un programa en C puede constar de varios archivos fuente; pero en ellos solamente debe definirse una sola función main.

Se pueden hacer llamadas a una función desde cualquier función, no importando el archivo fuente en donde se defina; pero debe de existir una declaración en el archivo en donde se encuentra la función que hace la llamada.

Las variables globales pueden ser accesadas desde cualquier función; pero, debe haber una declaración de la variable en el archivo fuente en donde se utilice.

Es recomendable que se definan archivos de encabezado que contengan los símbolos, tipos y prototipos de las funciones definidas en un archivo fuente, para que de esta forma se utilice la proposición **#include** con el archivo de encabezados en todos los archivos fuente donde se pretenda utilizar dichas funciones.

```
/*
**      minimo
**
**      Funcion que controla las operaciones sobre el numero menor del programa.
**
**      PARAMETROS:
**              opcion      operaciones a aplicar sobre el numero menor:
**                      SET     inicializar con el valor del parametro n.
**                      MATCH   comparar el numero menor con el parametro n para
**                              obtener el nuevo valor menor.
**                      GET     unicamente regresa el valor menor.
**              n       valor inicial (opcion=0) o valor a comparar con el menor
**                      (opcion =1).
**
**      RETURN: El valor menor
**              ERROR en caso de que el parametro opcion no sea valido.
*/

int     minimo(short opcion, int n)
{
        static int min=0;

        switch (opcion)
        {
        case SET:   min = n;
                    return min;
        case MATCH: min = (min > n ? min : n);
        case GET:   return min;
        default:    return ERROR;
        }
}
```

```
/*
**      maxMin.h
**
**      Archivo para definicion de prototipos y simbolos utilizados en las funciones del
**      archivo fuente maxMin.c
**
*/

#ifndef     _MAX_MIN_H
#define     _MAX_MIN_H

#define     SET         0
#define     MATCH       1
#define     GET         2
#define     ERROR       -1

int     maximo(short,int);
int     minimo(short,int);

#endif
```

LABORATORIO

1. Implemente el programa de empaquetamiento y desempaquetamiento en tres archivos, uno para el main, otro para las funciones de empaquetamiento y desempaquetamiento y otro para la función que imprime la representación binaria de un número.

Página intencionalmente blanca.