CAPÍTULO 6

FUNCIONES

6.1 Concepto de función

Una función es una parte o módulo de un programa que realiza una tarea específica. Está constituida por un conjunto de sentencias y puede ser llamada desde cualquier lugar dentro de un programa.

6.2 Estructura de una función

La estructura general de una función en C es la siguiente:

```
Tipo_de_retorno nombre_de_la_función(Lista de Parámetros)
{
        cuerpo de la función
        return Expresión;
}
```

Donde:

Tipo_de_retorno es el tipo de datos del valor devuelto por la función o la palabra **void** si la función no devuelve ningún valor.

nombre_de_la_función es el identificador o nombre de la función. El nombre de una función comienza por una letra o subrayado y puede contener tantas letras, números o subrayados como se desee. Cuando desde un programa se llama a una función, se hace utilizando este nombre.

Lista de Parámetros es una lista de declaraciones de los argumentos separados por comas.

Expresión es el valor que devuelve la función.

En el siguiente ejemplo, la función suma tiene un argumento entero N (dato de entrada) y calcula el sumatorio de 1 a N, siendo este valor (de tipo float) el devuelto por la función:

```
float  suma ( int  N)
{
      int  i;
      float total=0.;
        for(i=1;i<=N;i++)
          total+=total;
      return  total;
}
```

Una función se puede utilizar para realizar alguna tarea específica que no consista en obtener un valor y devolverlo; entonces, estas tareas se disponen en sentencias en el cuerpo de la función y el comando return puede no aparecer. Además, el tipo de esta función será void. (Cuando el tipo es void se está indicando explícitamente que la función no devuelve un valor; puede omitirse la declaración de tipo void, aunque el programa en este caso considera la función

como int (tipo por defecto); como realmente no se devuelve un valor no nos importa el tipo que considere el programa).

En el siguiente ejemplo, la función escribe no tiene ningún argumento y no devuelve ningún valor. Cuando es llamada desde cualquier parte de un programa ejecuta las instrucciones presentes en su cuerpo, es decir, escribe en pantalla la frase correspondiente:

```
void escribe ( )
{
    printf("Escribo desde la función ESCRIBE");
}
```

6.3 Prototipo de una función

La declaración de una función se llama **prototipo**. Los prototipos consisten en una sentencia idéntica a la de la cabecera de la función pero con la diferencia de que los prototipos terminan en punto y coma. En C no es estrictamente necesaria la declaración de prototipos, pero es conveniente para que el compilador pueda hacer chequeos en las llamadas a las funciones. Los prototipos de las funciones usadas en un programa se incluyen en la cabecera del programa (antes de que comience cualquier función) para que así sean reconocidas en todo el programa.

De las dos funciones que aparecen en los ejemplos anteriores, los prototipos serían:

```
float sum (int N);
    void escribe ( );
```

6.4 Estructura general de un programa que utiliza varias funciones externas

Hasta el momento, sólo se había empleado la función main( ), que delimita la parte principal del programa, y las funciones intrínsecas. Lo habitual es que un programa conste de varias funciones externas además de éstas. Veamos a continuación la estructura de un programa que consta de tres funciones externas f1, f2 y f3 de tipo void:

```
# include .....

/*prototipos de las funciones*/
void f1( );
void f2( );
void f3( );

 void main( )
{
  /*sentencias correspondientes a la parte principal del programa*/
 }
 void f1( )
 {
   /* sentencias correspondientes a la función f1*/
  }
 void f2( )
  {
   /* sentencias correspondientes a la función f2*/
   }
 void f3( )
  {
   /*sentencias correspondientes a la función f3*/
   }
```

Desde cualquier función del programa se puede llamar a otra de las funciones existentes incluyendo la sentencia de llamada, con la sintaxis:

nombre_función (argumentos);

entonces, el flujo del programa pasa a la función a la que se ha llamado, se ejecutan las instrucciones correspondientes a ella y cuando se encuentra con su llave de cierre o el comando return regresa a la sentencia inmediatamente posterior a la de la llamada.

Cuando la función se utiliza para devolver un valor, se sabe que en la función aparece la sentencia return valor; en ese momento se regresa a la función desde la que se ha realizado la llamada y, si ese valor quiere recogerse en ella, se deberá asignar a una variable de la forma:

X= nombre_función (argumentos);

X tomará el valor retornado por nombre_función.

A continuación se muestran dos ejemplos de programas que utilizan funciones que no retornan valores:

Ejemplo 1:

```
void f1( );
void f2( );
void main( )
{
f2( );
printf(" a todos \n");
 }
void f1( )
{
printf(" días");
 }
void f2( )
{
printf("Buenos");
f1( );
 }
```

La salida a pantalla es:

Buenos días a todos

Ejemplo 2:

```
#include <stdio.h>
void producto(int x,int y);
 void main (  )
 {
   producto(2,3);
  }
 void producto(int x,int y)
{
  printf("%d",x*y);
}
```

La función producto recibe los valores 2 y 3 para los argumentos x e y, e imprime en pantalla el producto. La salida a pantalla es:

6

A continuación, se va a volver a realizar el ejemplo anterior pero, en este caso, la función producto devuelve el valor del producto a la función principal y es ésta la que lo imprime en pantalla:

```
int producto(int x, int y);
void main (  )
{
int x;
/*llamada a la función,
    el valor del producto se almacena en la variable x*/
x=producto(2,3);
printf("%d",x);
}

int producto(int x, int y)
{
       return x*y;
}
```

6.5 Parámetros de una función

Cuando se llama a una función, en general, se pasan a ésta unos valores, que son los llamados parámetros o argumentos. Se sabe que una función puede no contener argumentos (en ese caso se disponen los paréntesis de la función vacíos) o contener varios parámetros, que se colocan separados por comas dentro de los paréntesis.

Los argumentos que aparecen en la llamada a la función se denominan *argumentos reales o actuales*.

Los argumentos que aparecen en la cabecera de la función son los *argumentos formales*.

El siguiente ejemplo clarifica la explicación anterior:

```
#include ……..
float suma (float x, float y) ;

void main( )
{
float x,y;
.....
.....
suma (x,y);
.....
}

float suma (float x, float y)
{
…….
…….
}
```

Argumentos reales

Argumentos formales

Los argumentos reales y formales correspondientes deben coincidir en tipo, en caso contrario se producirá un error en la compilación.

En la llamada a la función se establece una conexión entre el argumento real y formal correspondiente que se denomina *paso de parámetros*. Se puede realizar mediante dos técnicas diferenciadas:

- Paso de parámetros por valor
- Paso de parámetros por referencia

- **Paso de parámetros por valor***:* cuando se llama a una función, ésta recibe una copia de los valores de los *parámetros reales o actuales*, que son guardados en los *parámetros formales* correspondientes. Si se modifica el valor del *parámetro formal* dentro de la función, el cambio sólo afecta a la función y no tiene efecto fuera de ella (el *parámetro real* no se verá afectado por el cambio).

Ejemplo:

```
#include<stdio.h>
void func(int i);

void main( )
{
   int i=6;
   func(i);
   printf("i=%d \n",i);
 }

 void func(int i)
 {
   i++;
   printf("i=%d \n",i);
 }
```

Este programa imprime como salida:

```
i=7
i=6
```

- **Paso de parámetros por referencia*:*** cuando una función debe modificar el valor del *parámetro formal* y que el cambio afecte al *parámetro real* correspondiente, se usa el método de paso por referencia o dirección.

  Con este método, el compilador pasa la dirección en memoria del *parámetro real* considerado, por ello, el símbolo & debe preceder al nombre del argumento, y el *parámetro formal* correspondiente de la función debe declararse como puntero.

  Ejemplo:

```
#include<stdio.h>
void  intercambio(int *a, int *b);
void main (  )
{
int i=3, j=90;
printf("i=%d   j=%d \n",i,j);
intercambio(&i,&j);        /* Llamada a la función */
printf("i=%d   j=%d \n",i,j);
 }
void intercambio(int *a,int *b)
 {
 int  aux=*a;
  *a=*b ;
  *b=aux ;
 }
```

  La salida a pantalla de este programa es :

```
i=3      j=90
i=90     j=3
```

El método por defecto de pasar parámetros en C es siempre por VALOR. Sin embargo, en el caso de tablas, éstas se transmiten siempre por REFERENCIA o DIRECCIÓN.

6.6 Paso de tablas a funciones

6.6.1 Tablas unidimensionales

Supongamos que se hace una llamada desde la función principal **main** a una función cualquiera, por ejemplo, **paso_tabla**, queriendo transmitir a ésta un vector **v**. La llamada a la función sería:

```
void main( )
{
 int v[10];
 paso_tabla (v);
 ....................
```

El argumento real es v, es decir la constante puntero que apunta a v[0] o lo que es lo mismo, la dirección de v[0]. Lo que se transmite, por tanto, es una dirección, por eso la transmisión es por referencia. Para recoger esta dirección desde la función paso_tabla se tienen tres posibilidades. Se puede poner como argumento formal:

1- un puntero al tipo de datos de la tabla:

```
paso_tabla (int *p)
```

2- una tabla del mismo tamaño y tipo que la transmitida:

```
paso_tabla (int p[10])
```

3- una tabla de tamaño indeterminado:

```
paso_tabla (int p[ ])
```

En el siguiente ejemplo la función tabla recibe por referencia un vector de 3 componentes:

```
#include <stdio.h>
void tabla (int *b);/*también se puede poner b[3] ó b[ ]*/

void main( )
{
      int a[3]={1,2,3},i;
      tabla(a);
      for(i=0;i<3;i++)printf("%d ", a[i]);
}

void tabla (int *b)/*también se puede poner b[3] ó b[ ]*/
{
   int sum=0,i;
   for (i=0;i<3;i++)
   {
      sum=sum+b[i];
            b[i]=sum;
   }
}
```

La salida a pantalla es:

1 3 6

Si se quisiera pasar a la función la dirección del segundo elemento de la tabla, se tendría que indicar explícitamente una transmisión por referencia, ya que los elementos de la tabla se comportan como una variable normal (no son punteros):

```
void tabla (int *b);
void main( )
{
         int a[3]={1,2,3},i;
 tabla(&a[1]);
  for(i=0;i<3;i++)printf("%d ", a[i]);
}
```

```
void tabla (int *b)
{
   (*b)++;
    b++;
   (*b)++;
}
```

La salida a pantalla es:

1 3 4

El ejemplo anterior es equivalente al siguiente programa, en el que el vector b (argumento formal) recoge la segunda y sucesivas componentes del vector a (argumento real):

```
void tabla (int b[ ]);
main( )
{
  int a[3]={1,2,3},i;
 tabla(&a[1]);
   for(i=0;i<3;i++)printf("%d ", a[i]);
 }
  void tabla (int b[ ])
 {
  b[0]++;
  b[1]++;
 }
```

6.6.2 Tablas multidimensionales

Cuando se quieren transmitir tablas multidimensionales a funciones, en el *argumento formal* es necesario especificar el tamaño de todas las dimensiones salvo el de la primera.

Ejemplo:

```
#include <stdio.h>
void tablasum (int b[][3]);/*también se puede poner b[3][3] */
 void main(  )
 {
 int a[3][3]={1,2,3,4,5,6,7,8,9},i,j;
 tablasum(a);
 for(i=0;i<3;i++)
        for(j=0;j<3;j++) printf("%d ", a[i][j]);
}
void tablasum (int b[][3])/*también se puede poner b[3][3]*/
{
 int sum=0,i,j;
  for (i=0;i<3;i++)
       for(j=0;j<3;j++)
          {
                 sum=sum+b[i][j];
                 b[i][j]=sum;
          }
}
```

La salida a pantalla es:

1 3 6 10 15 21 28 36 45

6.7 Ámbito de una variable

El ámbito o alcance de una variable determina cuáles son las funciones desde las que esa variable es reconocida.

Si una función reconoce una variable, la variable es visible en esa función, por tanto, el ámbito es la zona de un programa en la que es visible una variable. En C hay cuatro clases de ámbito para las variables:

- **Ámbito de programa (variables globales)**

Las variables que tienen ámbito de programa pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman **globales**. Para hacer que una variable sea global se debe declarar al principio del programa, fuera de cualquier función.

Ejemplo:

```
#include <stdio.h>
int x;
void cua( );
void main( )
{
        printf("Introduce un entero\n");
        scanf("%d",&x);
        cua( );
}

void cua ( )
{
        printf("El numero %d al cuadrado es: %d",x,x*x);
}
```

En este programa la variable **x** es global y "se ve" desde las dos funciones del programa.

Si se introduce *5* como dato, la salida es:

El numero 5 al cuadrado es 25

Con este ejemplo se comprueba, que la utilización de variables globales es otra forma de transmisión de valores entre funciones, ya que, gracias a que x es una variable global, la función cua no necesita argumentos.

- **Ámbito de archivo fuente**

Un programa puede estar guardado en un archivo fuente o en varios. Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra **static**, tiene ámbito de archivo fuente. Las variables con este ámbito se pueden referenciar desde el punto del programa en el que estén declaradas hasta el final del archivo fuente, pero no se podrían ver desde los otros archivos fuente que forman el programa (si hay más de uno).

Ejemplo: `static int i;` (Aquí i tiene ámbito de archivo fuente).

- **Ámbito de función (variables locales)**

Una variable que tiene ámbito de una función se puede referenciar desde cualquier parte de la función.

Las variables declaradas dentro del cuerpo de la función se dice que son **locales** a la función. Estas variables no pueden ser utilizadas fuera de la función en la que están definidas. En este ejemplo:

```
void    funcional ( void)
{
double  x,y,z;  /*  Ámbito de función  */
.....
}
```

las variables x, y, z son locales a la función funcional. Si aparecen unas variables con los mismos nombres en otra parte del programa, serían unas variables distintas.

- **Ámbito de bloque**

Una variable declarada en un bloque tiene ámbito de bloque y puede ser referenciada en cualquier parte de éste, desde el punto en que esté declarada hasta el final del bloque. Estas variables no son visibles fuera del bloque.

Véase el siguiente ejemplo:

```
float   func( int j)
{
if (j>10)
{
 int i;
 for (i=0;i<20;i++)
 func1(i);
 }
    /*  Aquí ya no es visible  i  */
}
```

6.8 Funciones en línea. Macros con argumentos.

En C se pueden definir funciones en una sola línea, utilizando macros con argumentos por medio de la sintaxis siguiente:

```
#define  Nombremacro (parámetros sin tipos)  expresión_texto
```

Ejemplo:

```
#include<stdio.h>
#define f(x)  (x*x+2*x-1)
void main(  )
{
    float x;
    for (x=0.0;x<=6.5; x+=0.3)
    printf("%6.2f", f(x));
}
```

6.9 Clases de almacenamiento

La clase de almacenamiento de una variable se refiere a la permanencia de una variable en memoria. Las variables dentro de una función pueden presentar dos tipos de almacenamiento y se clasifican en:

- **Variables automáticas**

Una variable automática tiene asignado espacio de memoria automáticamente a la entrada de la función y se libera el espacio tan pronto se sale de dicha función. Es decir, la variable se borra de la memoria cuando termina la función.

Las variables que se declaran dentro de una función se dice que son automáticas (auto), salvo que en la declaración se incluya explícitamente un tipo de almacenamiento distinto.

- **Variables estáticas**

Las variables estáticas no se borran de la memoria cuando la función termina y, en consecuencia, retienen sus valores entre llamadas sucesivas a la función. Al contrario que las variables automáticas, una variable estática se inicializa sólo una vez, en la primera llamada a la función. Para declarar una variable como estática se le antepone el adjetivo **static** en su declaración.

Véanse a continuación dos ejemplos que clarifican la diferencia entre variables automáticas y estáticas. Sea el programa:

```
#include<stdio.h>
 void dev(int *i, int *s);

 void main( )
 {
    int s=0, i=1;
    dev(&i,&s);
    dev(&i,&s);
    printf("s=%d \n",s);
 }
```

```
  void dev(int *i,int *s)
{
  int  j=1;
  j=j+5;
 *s=j+*i;
 }
```

La variable j tiene almacenamiento automático; cada vez que se llama a la función dev se inicializa a 1.
La salida de este programa es

s=7

Si se modifica el programa haciendo que la variable j tenga almacenamiento estático, es decir:

```
#include<stdio.h>
 void dev(int *i, int *s);
 void main(  )
 {
  int  s=0, i=1;
  dev(&i,&s);
  dev(&i,&s);
  printf("s=%d \n",s);
}
void dev(int *i,int *s)
{
 static int  j=1;
  j=j+5;
 *s=j+*i;
 }
```

la salida es a pantalla es:

s=12

Esto es debido a que j retiene el valor (j=6) de la primera llamada a la función dev.

6.10 Enunciados de las prácticas

Ejercicio 1.

Diseñar un programa C que conste de la función main, en la que se lea por teclado dos valores enteros a y b y, con estos datos, llame a una función suma que calcule la suma de sus dos argumentos enteros x e y, y que retorne ese valor; en la función principal se imprimirán por pantalla los valores de los sumandos y el valor de la suma.

Ejercicio 2.

La mayor utilidad de una función es que calcula "algo" para diferentes valores de los parámetros. En el siguiente programa hay dos funciones,

AreaCirculo
AreaEsfera

y en la función principal se va a calcular el *Área del círculo* y el *Área de la esfera* para diferentes valores del radio, sin más que llamar repetidas veces a las mismas funciones pero con diferentes argumentos de llamada.

Analizar este programa interpretando todas las instrucciones.

Nota: aparece una constante llamada PI, declarada mediante una sentencia define

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

#define PI          3.141592

/* Prototipos de las funciones   */

float AreaCirculo(float);
float AreaEsfera(float);

void main( )
{
    float radio;
    puts("=== Areas ===");
    puts("Radio\tCirculo\tEsfera");
```

```
     puts("------\t------\t------");

   /*
    * Para cada valor del radio, se imprime el radio y las
    *  áreas de las superficies para el círculo y la esfera
    */
     for (radio = 0.0; radio <= 2.01; radio += 0.2)
         printf("%6.2lf\t%6.3lf\t%6.3lf\n", radio,
               AreaCirculo(radio), AreaEsfera(radio));
}


float  AreaCirculo(float radio)
{
   return PI * (radio * radio);
}

float AreaEsfera(float radio)
{
   return 4.0*PI * (radio * radio);
}
```

Ejercicio 3.

Un año es bisiesto si cumple que es divisible entre 4 pero no entre 100, con la excepción de los múltiplos de 400, que sí son bisiestos.

Diseñar una función **bisiesto**, de retorno lógico, que devuelva verdadero si un valor entero agno es un año bisiesto; diseñar también la función main que permita comprobar el funcionamiento correcto del programa.

Ejercicio 4. (Números primos)

Un número entero es primo si sus únicos divisores son el 1 y el propio número; diseñar una función llamada **EsPrimo**, de retorno lógico, que devuelva verdadero si su único argumento, de tipo entero, es un número primo. Construir también el programa principal de comprobación.

Ejercicio 5. (Método de Newton)

El método de Newton trata de resolver ecuaciones del tipo f(x)=0.

Haciendo el desarrollo limitado de Taylor de f alrededor de un punto x_0:

$$f(x)=f(x_0)+f'(x_0)(x-x_0)=0$$

se obtiene $$x=x_0-f(x_0)/f'(x_0)$$

Aplicando iterativamente el método anterior se llega al algoritmo de **Newton**:

- Partiendo de un x_0 supuestamente cerca de la solución se va iterando del siguiente modo:

$$x^{k+1}=x^k-(f(x^k)/f'(x^k)) \quad (1)$$

- Como no se conoce la solución exacta, el algoritmo se detiene cuando dos soluciones consecutivas difieren en una cantidad lo suficientemente pequeña que llamaremos tolerancia (ε). Es decir, cuando $|x^{k+1}-x^k|<\varepsilon$.

Escribir un programa en C que obtenga una raíz de la ecuación $x^2-4=0$ tomando como valor inicial $x_0=1$. El programa debe tener tres funciones: la primera de nombre **newton** para el desarrollo del algoritmo según la ecuación (1), la segunda de nombre **f** para definir f(x) y la tercera de nombre **df** para definir la derivada de f.

Ejercicio 6. (Ordenación de un vector por el método de la Burbuja).

Para ordenar (de menor a mayor) los elementos de un vector por el *método de la burbuja*, se procede a ir comparando, dos a dos, elementos consecutivos, de tal manera que si están desordenados, se intercambian los valores. Este proceso aplicado una vez, de izquierda a derecha, consigue situar el elemento mayor a la derecha. Si se repite el proceso, el elemento más grande de los restantes queda situado al lado izquierdo del mayor y así sucesivamente. Se va a aplicar el método al siguiente ejemplo:

Sea el vector a ordenar 5 4 3 2 1

El proceso es como sigue (se han resaltado en letra negrita las parejas de elementos a comparar en cada iteración):

5 4 3 2 1	→	4 5 3 2 1
4 **5 3** 2 1		4 3 5 2 1
4 3 **5 2** 1		4 3 2 5 1
4 3 2 **5 1**		4 3 2 1 5
4 3 2 1 5		3 4 2 1 5
3 **4 2** 1 5		3 2 4 1 5
3 2 **4 1** 5		3 2 1 4 5
3 2 1 4 5		2 3 1 4 5
2 **3 1** 4 5		2 1 3 4 5
2 1 3 4 5		1 2 3 4 5

Escribir un programa en C que lea un vector v de dimensión menor que 1000 y llame a una función de nombre **burbuja** que se encargue de ordenarlo de acuerdo al algoritmo de la burbuja.

Ejercicio 7. (Ordenación de un vector por el método de Selección)

Para ordenar los elementos de un vector de n componentes por el *método de selección* se procede del siguiente modo:

- Se calcula el mínimo de las n componentes y se intercambia con el primero.
- En el vector resultante, se calcula el mínimo de los n-1 últimos elementos y se intercambia con el segundo, si no coincide con él.
- En el vector resultante se calcula el mínimo de los n-2 últimos elementos y se intercambia con el tercero si no coincide con él.
- Así sucesivamente hasta llegar a la componente n-1.

Por ejemplo, si el vector a ordenar es 5 2 4 6 1 3, el proceso a seguir es el siguiente (en cada iteración se ha puesto en letra negrita el mínimo y subrayado la posición en la que hay que situarlo):

5 2 4 6 **1** 3	→	**1** 2 4 6 5 3
1 **2** 4 6 5 3		1 **2** 4 6 5 3
1 2 4 6 5 **3**		1 2 **3** 6 5 4
1 2 3 6 5 **4**		1 2 3 **4** 5 6
1 2 3 6 **5** 4		1 2 3 4 **5** 6

Escribir un programa en C que lea un vector v de dimensión menor que 1000 y que llame a una función de nombre **seleccion** que se encargue de ordenarlo de acuerdo al algoritmo de selección.

Ejercicio 8. (Ordenación de un vector por el método de Inserción)

Para ordenar los elementos de un vector por el *método de inserción* se procede del siguiente modo:

- Se considera el segundo elemento, si es menor que el primero, se intercambia con él.
- En el nuevo vector se considera el tercer elemento, si es menor que el primero, se inserta en el primer lugar desplazando un lugar a la derecha a los dos primeros elementos, si es mayor que el primero pero menor que el segundo, se intercambia con este último.
- En el nuevo vector, se toma el cuarto elemento y se sitúa en el lugar correspondiente, desplazando a los mayores que él un lugar a la derecha.
- Se procede de esta forma hasta llegar al n-simo elemento.

Por ejemplo, si el vector a ordenar es 5 2 4 6 1 3, el proceso a seguir es el siguiente (se ha señalado en letra negrita el elemento a comparar con los anteriores en cada iteración):

5 **2** 4 6 1 3	→	**2** 5 4 6 1 3
2 5 **4** 6 1 3		2 **4** 5 6 1 3
2 4 5 **6** 1 3		2 4 5 **6** 1 3
2 4 5 6 **1** 3		**1** 2 4 5 6 3
1 2 4 5 6 **3**		1 2 **3** 4 5 6

Escribir un programa en C que lea un vector v de dimensión menor que 1000 y que llame a una función de nombre **insercion** que se encargue de ordenarlo de acuerdo al algoritmo de inserción.

Ejercicio 9.

Dado un conjunto de puntos en el plano, se define su diámetro como la distancia máxima entre dos de ellos. Diseñar un programa que calcule el diámetro de un conjunto formado por n puntos (n<1000). Las coordenadas de los puntos deben situarse en una matriz `p[n][2]`. Se debe utilizar una función **`dist`** que calcule la distancia entre dos puntos y una función **`max`** para calcular el máximo de dos números reales.

Ejercicio 10. (Puntos de silla de una matriz).

Dada una matriz, se dice que uno de sus coeficientes es un PUNTO DE SILLA si es el máximo de su fila y mínimo de su columna, o bien, si es mínimo de su fila y máximo de su columna.

Por ejemplo, en la matriz:

1	2	[3]
[5]	6	7
4	1	3

son puntos de silla los coeficientes que están recuadrados.

Diseñar un programa en C que calcule los puntos de silla de una matriz de dimensión n (n<100) y que informe de la fila y columna de la matriz donde se encuentran.

Ayuda: El pseudocódigo que resuelve este problema se puede plantear así:

> Lectura de los coeficientes de a
> Bucle a los coeficientes a.
> Si a[i][j] es punto de silla, imprimir: "a[i][j] es punto de silla".
> Fin del bucle

La verificación de que `a[i][j]` es punto de silla se puede realizar mediante una función **`silla(`**`a,i,j,n`**`)`** que devuelva 1 si `a[i][j]` es punto de silla y 0 en caso contrario.

Esta función utilizará a su vez, dos funciones enteras **`mn`**`(l,v,n)` y **`mx`**`(l,v,n)` que devolverán 1 si el elemento `v[l]` es el mínimo o máximo respectivamente del vector `v` y 0 en caso contrario.

6.11 Solución a las prácticas del capítulo 6

Ejercicio 1.

A continuación se ofrece una solución válida al ejercicio:

```
#include <stdio.h>
/*
 * Prototipo de función.
 */
int suma(int x, int y);

void main(  )
{
   int a,b;
   printf("Introduce los sumandos:");
   scanf("%d %d",&a,&b);
   printf("El resultado de sumar %d y %d es %d",a,b,suma(a,b));
}

int suma(int x, int y)
{
   int sum;
   sum=x+y;
   return sum;
}
```

Nota: En la función suma se ha utilizado una variable auxiliar llamada sum; se podría evitar el empleo de esta variable tal como se indica a continuación:

```
int suma(int x, int y)
{
    return (x+y);
}
```

Ejercicio 2.

La salida del programa es la siguiente:

```
=== Areas ===
Radio   Circulo Esfera
------  ------  ------
  0.00   0.000   0.000
  0.20   0.126   0.503
  0.40   0.503   2.011
  0.60   1.131   4.524
  0.80   2.011   8.042
  1.00   3.142  12.566
  1.20   4.524  18.096
  1.40   6.158  24.630
  1.60   8.042  32.170
  1.80  10.179  40.715
  2.00  12.566  50.265
```

Ejercicio 3.

Una posible solución, incluyendo la función main y la función bisiesto es la ofrecida a continuación:

```
#include <stdio.h>

int bisiesto(int agno);

void main(  )
{
   int agno;
   printf("\n Introduce el agno a investigar");
   scanf("%d",&agno);
   if(bisiesto(agno))
          printf("\n El agno %d es bisiesto",agno);
   else
          printf("\n El agno %d no es bisiesto",agno);
}
```

```
int bisiesto(int agno)
{
   return  agno%4==0&&agno%100!=0||agno%400==0 ;
}
```

Nota: observar que en la expresión

```
agno%4==0&&agno%100!=0||agno%400==0
```

no se han utilizado paréntesis, puesto que se ha hecho uso de las prioridades de los operadores: primero se hacen las operaciones aritméticas (resto de dividir entre ..), después las operaciones relacionales, a continuación la operación lógica && (y lógico), y por último la operación lógica || (o lógico).
Si no se es experto en el orden, es recomendable (aunque no aporta legibilidad al programa), hacer uso de cuantos paréntesis sean necesarios. Observar la diferencia si se hiciese esto (se han añadido espacios en blanco para facilitar la lectura):

```
(((agno%4) == 0 )&& ((agno%100)!=0)) || ((agno%400)== 0)
```

Ejercicio 4.

Un programa válido es el siguiente:

```
#include<stdio.h>

int EsPrimo(int);

void main(  )
{
    int n;
    printf("\n Introducir un numero entero\n");
    scanf("%d",&n);
    if(EsPrimo(n))
          printf(" \n El numero %d es primo",n);
    else
          printf(" \n El numero %d no es primo",n);
}
```

```
int EsPrimo(int n)
{
    int i;
    for (i=2;i*i<=n;i++) if(n%i==0) return 0;
    return 1;
}
```

Observaciones:

1. La condición de repetición del bucle (i*i<=n) se explica de la siguiente forma: el divisor (distinto de n) más grande posible es n/2, así que una condición válida sería i<=n/2. Pero si n/2 es un número entero que es divisor de n, entonces 2 también es divisor, y ya se probó en una iteración anterior si el 2 dividía a n. De esta forma no hace falta probar con el n/2. Entonces, el siguiente divisor más grande posible es n/3, pero por un razonamiento análogo, tampoco hace falta probarlo ya que antes se había probado con el 3. Este proceso se puede repetir hasta que n/i = i, o sea hasta que $i^2 = n$. Una manera más formal de verlo es la siguiente: si d es un divisor de n, entonces n/d es un divisor de n. Así que en vez de probar con estos dos números sólo hace falta probar con el más pequeño.
2. Si n es divisor de algún valor i, la función vale falso (0) y debe retornar ese valor a la función llamadora sin necesidad de investigar si existe algún otro divisor; si no se ha cumplido la condición n%i==0 para ninguno de los valores de i entonces la función debe retornar el valor 1.

Ejercicio 5.

A continuación se ofrece una posible solución:

```
/*  Método  de Newton */
#include<stdio.h>
#include<math.h>

/* Prototipos  */

double newton(double x) ;
double f(double x) ;
double df(double x);


void main(  )
{
    int k=0; double x0,x1,tol;
    printf("Introduce la Tolerancia \n");
    scanf("%lf",&tol);
    printf("Introduce el valor inicial del algoritmo \n");
    scanf("%lf",&x0);

    while(1)
    {
        x1=newton(x0);
        printf("x1=%f \n",x1);
        k=k+1;
        if(fabs(x1-x0)<tol) break;
        x0=x1;
    }
    printf("Numero de Iteraciones: %d \n",k);
    printf("Raiz aproximada: %f \n",x1);
}
```

```
double newton(double x)
{
   return x-(f(x)/df(x));
}


double f(double x)
{
   return x*x-4;
}


double df(double x)
{
   return 2*x;
}
```

Si se ejecuta este programa con el valor de tol 0.000001 y x0=1, se obtiene:

```
x1=2.500000
x1=2.050000
x1=2.000610
x1=2.000000
x1=2.000000

Numero de Iteraciones:  5
Raiz aproximada:   2.000000
```

Ejercicio 6. Método de la Burbuja.

El siguiente programa ordena un vector de números según el algoritmo de la burbuja:

```
#include<stdio.h>
#define DIM 1000
void burbuja(float v[ ],int n);

void main(  )
{
    int i,n;
    float v[DIM];
    printf("¿Dimension del vector? \n");
    scanf("%d",&n);
    printf("¿Coeficientes del vector? \n");
    for(i=0;i<n;i++)
    scanf("%f",&v[i]);
    burbuja (v,n);
    printf("El vector ordenado es \n");
    for(i=0;i<n;i++)
                printf("v[%d]=%f \n", i,v[i]);
}

void burbuja(float v[ ],int n)
{
    int i, j, logic; float aux;
    for(i=n-1; i>=1; i--)
    {
        logic=1;
        for(j=1; j<=i; j++)
       {
          if(v[j-1]>v[j])
          {
            aux=v[j];
            v[j]=v[j-1];
            v[j-1]=aux;
            logic=0;
          }
      }
      if (logic) return ; /* ordenación conseguida */
    }
}
```

Ejercicio 7. Método de Selección

El siguiente programa ordena un vector de números según el algoritmo de selección:

```
#include<stdio.h>
#define DIM 1000
void seleccion(float v[ ],int n);
void main(  )
{
    int i,n;
    float v[DIM];
    printf("¿Dimension del vector? \n");
    scanf("%d",&n);
    printf("¿Coeficientes del vector? \n");
    for(i=0;i<n;i++)
    scanf("%f",&v[i]);
    seleccion(v,n);
    printf("El vector ordenado es \n");
    for(i=0;i<n;i++)
        printf("v[%d]=%f \n", i,v[i]);
}
void seleccion(float v[ ],int n)
{
    int i, j, k;   float m;
    for(i=0;i<n-1;i++)
    {
        m=v[i];
        k=i;
    /* Búsqueda del menor componente */
        for(j=i+1; j<n; j++)
        {
            if(v[j]<m)
            {
              m=v[j];
              k=j;
            }
        }
      /* Intercambios */
        v[k]=v[i];
        v[i]=m;
    }
}
```

Ejercicio 8. Método de Inserción.

El siguiente programa ordena un vector de números según el método de inserción:

```
#include<stdio.h>
#define DIM 1000

void insercion(float v[ ], int n);

void main(  )
{
    int i,n;
   float v[DIM];
   printf("¿Dimension del vector? \n");
   scanf("%d",&n);
   printf("¿Coeficientes del vector? \n");
   for(i=0;i<n;i++)
   scanf("%f",&v[i]);
   insercion(v,n);
   printf("El vector ordenado es \n");
   for(i=0;i<n;i++)
      printf("v[%d]=%f \n", i,v[i]);
}

void insercion (float v[ ], int n)
{
   int i,j; float k;
   for(i=1; i<n; i++)
   {
     k=v[i];
     /* Inserta  v[i] en la sucesión ordenada  v[0],……..v[i-1] */
     j=i-1;
    while( j>=0  && v[j]>k)
    {
       v[j+1]=v[j];
       j=j-1;
       v[j+1]=k;
    }
  }
}
```

A continuación se muestra una solución alternativa a la función insercion:

```
void insercion (float v[ ], int n)
{
   int i,j,k; float m;
   for(i=1; i<n; i++)
   {
     m=v[i];
     /* compara  v[i] con los elementos anteriores */
     for(j=0;j<i;j++)
     {
        if(v[i]<v[j])
        {
/*se desplaza un lugar a la derecha los elementos entre la posición j y la i*/
         for(k=i;k>j;k--)v[k]=v[k-1];
         v[j]=m;
         break;
         }
      }
   }
}
```

Ejercicio 9.

A continuación se ofrecen dos soluciones a este ejercicio:

Solución 1:

Esta primera solución hace uso explícito de punteros y es más corta y elegante que la segunda solución que veremos más adelante.

```
#include<stdio.h>
#include<math.h>
float dist(float* a, float* b);
float max(float s, float t);

void main( )
{
    int i,j,n;
    float p[1000][2],diametro=0;
    printf("¿número de puntos del conjunto? \n");
    scanf("%d",&n);
```

```
    printf("¿Coordenadas de los puntos? \n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<2;j++)
            scanf("%f",&p[i][j]);
    }

/* Algoritmo Iterativo  sobre las n(n-1)/2  distancias */
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<=n-1; j++)
            diametro=max(diametro,dist(p[i],p[j]));
    }
    printf("el diámetro es : %f \n",diametro);
}

float dist(float* p, float* q)
{
    return sqrt((*p-*q)*(*p-*q)+(*(p+1)-*(q+1))*(*(p+1)-*(q+1)));
}

float max(float s, float t)
{
    if(s>=t) return s;
    else      return t;
}
```

Solución 2:

En esta solución no se hace un uso explícito de los punteros.

```
#include<stdio.h>
#include<math.h>
float dist(float a[ ], float b[ ]);
float max(float s, float t);

void main(  )
{
    int i,j,n,k;
    float p[1000][2],a[2],b[2],diametro=0;
    printf("¿número de puntos del conjunto? \n");
    scanf("%d",&n);
```

```
    printf("¿Coordenadas de los puntos? \n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<2;j++)
            scanf("%f",&p[i][j]);
    }

/* Algoritmo Iterativo  sobre las n(n-1)/2  distancias */

    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<=n-1; j++)
        {
             for(k=0;k<2;k++)
             {
                a[k]=p[i][k];
                b[k]=p[j][k];
              }
             diametro=max(diametro,dist(a,b));
         }
    }
    printf("el diámetro es : %f \n",diametro);
}


float dist(float a[ ], float b[ ])
{
    return sqrt((a[0]-b[0])* (a[0]-b[0])+ (a[1]-b[1])* (a[1]-b[1]));
}



float max(float s, float t)
{
    if(s>=t) return s;
    else       return t;
}
```

Ejercicio 10.

A continuación se ofrece un programa válido:

```
#include<stdio.h>
#define DIM 100

int mn(int l, float v[ ],int n);
int mx(int l, float v[ ],int n);
int silla(float a[ ][DIM],int i,int j, int n);

void main(  )
{
int i,j,n;
float a[DIM][DIM];
printf("¿Dimension de la matriz a? \n");
scanf("%d",&n);


printf("Introduce los coeficientes de a por filas \n");


for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        scanf("%f",&a[i][j]);
}


for(i=0;i<n;i++)
{
   for(j=0;j<n;j++)
   if (silla(a,i,j,n)) printf("pto. de silla en i=%d j=%d de valor %f \n",i,j,a[i][j]);
}
}
```

```
int silla(float a[ ][DIM],int i, int j,int n)
{
    int k;
    float f[DIM],c[DIM];
    for(k=0;k<n;k++)
   {
        f[k]=a[i][k];
       c[k]=a[k][j];
   }
   if(mn(j,f,n) && mx(i,c,n) || mx(j,f,n) && mn(i,c,n)) return 1;
   else return 0;
}



int mn(int l,float v[ ],int n)
{
int k;
for(k=0;k<n;k++)
  {
   if( v[l]>v[k]) return 0;
   }
 return 1;
}



int mx(int l,float v[ ],int n)
{
int k;
for(k=0;k<n;k++)
  {
   if( v[l]<v[k]) return 0;
   }
 return 1;
}
```