CAPÍTULO 7

FUNCIONES

CONTENIDO

INTRODUCCIÓN

Una función es un miniprograma dentro de un programa. Las funciones contienen varias sentencias bajo un solo nombre, que un programa puede utilizar una o más veces para ejecutar dichas sentencias. Las funciones ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables. En otros lenguajes como BASIC o ensamblador se denominan *subrutinas*; en Pascal, las funciones son equivalentes a *funciones* y *procedimientos*.

Este capítulo examina el papel (rol) de las funciones en un programa C. Las funciones existen de modo autónomo, cada una tiene su ámbito. Como ya conoce, cada programa C tiene al menos una función **main()**; sin embargo, cada programa C consta de muchas funciones en lugar de una función **main()** grande. La división del código en funciones hace que las mismas se puedan reutilizar en su programa y en otros programas. Después de que escriba, pruebe y depure su función, se puede utilizar nuevamente una y otra vez. Para reutilizar una función dentro de su programa, sólo se necesita llamar a la función.

Si se agrupan funciones en bibliotecas otros programas pueden reutilizar las funciones, por esa razón se puede ahorrar tiempo de desarrollo. Y dado que las bibliotecas contienen rutinas presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

La mayoría de los programadores no *construyen* bibliotecas, sino que, simplemente, las utilizan. Por ejemplo, cualquier compilador incluye más de quinientas funciones de biblioteca, que esencialmente pertenecen a la *biblioteca estándar* ANSI (American National Standards Institute). Dado que existen tantas funciones de bibliotecas, no siempre será fácil encontrar la función necesaria, más por la cantidad de funciones a consultar que por su contenido en sí. Por ello, es frecuente disponer del manual de biblioteca de funciones del compilador o algún libro que lo incluya.

La potencia real del lenguaje es proporcionada por la biblioteca de funciones. Por esta razón, será preciso conocer las pautas para localizar funciones de la biblioteca estándar y utilizarlas adecuadamente. En este capítulo aprenderá:

- Utilizar las funciones proporcionadas por la biblioteca estándar ANSI C, que incorporan todos los compiladores de C.
- Los grupos de funciones relacionadas entre sí y los archivos de cabecera en que están declarados.

Las funciones son una de las piedras angulares de la programación en C y un buen uso de todas las propiedades básicas ya expuestas, así como de las propiedades avanzadas de las funciones, le proporcionarán una potencia, a veces impensable, a sus programaciones. La compilación separada y la recursividad son propiedades cuyo conocimiento es esencial para un diseño eficiente de programas en numerosas aplicaciones.

CONCEPTOS CLAVE

- Biblioteca de funciones,
- Compilación independiente.
- Función.
- Modularización.
- Parámetros de una función.
- Pasar parámetros por valor.
- Paso por referencia.
- Recursividad.
- Sentencia return.
- Subprograma.

7.1. CONCEPTO DE FUNCIÓN

C fue diseñado como un *lenguaje de programación estructurado,* también llamado *programación modular*. Por esta razón, para escribir un programa se divide éste en varios módulos, en lugar de uno solo largo. El programa se divide en muchos módulos (rutinas pequeñas denominadas *funciones),* que producen muchos beneficios: aislar mejor los problemas, escribir programas correctos más rápido y producir programas que son mucho más fáciles de mantener.

Así pues, un programa C se compone de varias funciones, cada una de las cuales realiza una tarea principal. Por ejemplo, si está escribiendo un programa que obtenga una lista de caracteres del teclado, los ordene alfabéticamente y los visualice a continuación en la pantalla, se pueden escribir todas estas tareas en un único gran programa (función **`main()`**).

```
int main()
{
  /* Código C para obtener una lista de caracteres */
  ...
  /* Código C para alfabetizar los caracteres */
  ...
  /* Código C para visualizar la lista por orden alfabético */
  ...
  return 0
}
```

Sin embargo, este método no es correcto. El mejor medio para escribir un programa es escribir funciones independientes para cada tarea que haga el programa. El mejor medio para escribir el citado programa sería el siguiente:

```
int main()
{
  obtenercaracteres();          /* Llamada a una función que obtiene los
                                   números */
  alfabetizar();                /* Llamada a la función que ordena
                                   alfabéticamente las letras */
  verletras();                  /* Llamada a la función que visualiza
                                   letras en la pantalla */
  return 0;                     /* retorno al sistema */
}

int obtenercaracteres()
{
  /*
     Código de C para obtener una lista de caracteres
  */
  return(0);                    /* Retorno a main() */
}

int alfabetizar()
{
  /*...
     Código de C para alfabetizar los caracteres
  */
  return(0) ;                   /* Retorno a main() */
}

void verletras()

  /*...
```

```
      Código de C para visualizar lista alfabetizada
   */

   return                          /* Retorno a main() */
}
```

Cada función realiza una determinada tarea y cuando se ejecuta `return` se retorna al punto en que fue llamada por el programa o función principal.

Consejo

Una buena regía para determinar la longitud de una función (número de líneas que contiene) es que no ocupe más longitud que el equivalente a una pantalla.

7.2. ESTRUCTURA DE UNA FUNCIÓN

Una función es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Las funciones permiten al programador un grado de abstracción en la resolución de un problema.

Las funciones en C no se pueden anidar. Esto significa que una función no se puede declarar dentro de otra función. La razón para esto es permitir un acceso muy eficiente a los datos. En C todas las funciones **son** externas o globales, es decir, pueden ser llamadas desde cualquier punto del programa.

La estructura de una función en C se muestra en la Figura 7.1.

```
tipo-de-retorno nombreFunción (listaDeParámetros)
{
      cuerpo de la función
      return expresión
}
```

tipo-de-retorno	Tipo de valor devuelto por la función o la palabra reservada `void` si la función no devuelve ningún valor.
nombreFunción	Identificador o nombre de la función.
listaDeParámetros	Lista de declaraciones de los parámetros de la función separados por comas.
expresión	valor que devuelve la función.

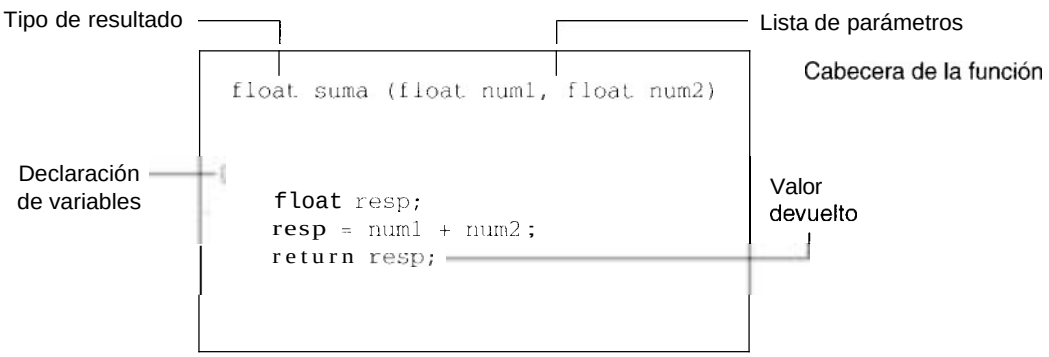


Figura 7.1. Estructura de una función.

Los aspectos más sobresalientes en el diseño de una función son:

- *Tipo* ***de resultado.*** Es el tipo de dato que devuelve la función C y aparece antes del nombre de la función.
- ***Lista de parúmetros.*** Es una lista de parámetros ***tipificados*** (con tipos) que utilizan el formato siguiente:

  tipo1 **parámetrol,** *tipo2 parámetro2,* **...**

- ***Cuerpo de lafunción.*** Se encierra entre llaves de apertura ({) y cierre (}). No hay punto y coma después de la llave de cierre.
- *Paso* ***de parámetros.*** Posteriormente se verá que el paso de parámetros en C se hace siempre por valor.
- ***No se pueden declararfunciones anidadas.***
- ***Declaración local.*** Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- ***Valor devuelto por lafunción.*** Mediante la palabra reservada `return` se devuelve el valor de la función.

Una llamada a la función produce la ejecución de las sentencias del cuerpo de la función y un retorno a la unidad de programa llamadora después que la ejecución de la función se ha terminado, normalmente cuando se encuentra una sentencia `return`.

Ejemplo 7.1

Lasfunciones cuadrado() y ***suma()*** *muestran dos ejemplos típicos de ellas.*

```
/* función que calcule los cuadrados de números enteros
   sucesivos a partir de un número dado (n), parámetro
   de la función y hasta obtener un cuadrado que sea
   mayor de 1000
*/

void cuadrado(int n)
{
  int cuadrado=0, q=0;
  while (q <= 1000)           /*el cuadrado ha de ser menor de 1000 */
  {
    q = n*n;
    printf("El cuadrado de: %d  es %d \n",n,q);
    n++;
  }
  return;
}
/*
   Calcula la suma de un número dado (parámetro) de elementos leídos de la
   entrada estándar(teclado).
 */
float suma (int num-elementos)
{
  int indice;
  float total = 0.0;

  printf("\n \t Introduce %d números reales\n",num_elementos);
  for (indice = 0; indice < num-elementos; indice++)
  {
     float x;
     scanf("%f ";&x);
```

```
    total += x;
  }
  return total;
}
```

7.2.1. Nombre de una función

Un nombre de una función comienza con una letra o un subrayado (_) y puede contener tantas letras, números o subrayados como desee. El compilador ignora, sin embargo, a partir de una cantidad dada (32 en Borland/Inprise C, 248 en Microsoft). C es sensible a mayúsculas, lo que significa que las letras mayúsculas y minúsculas son distintas a efectos del nombre de la función.

```
int max (int x, int y);                  /* nombre de la función max */
double media (double xl, double x2);     /* nombre de la función media */
double MAX (int* m, int n);              /* nombre de función MAX,
                                            distinta de max */
```

7.2.2. Tipo de dato de retorno

Si la función no devuelve un valor `int`, se debe especificar el tipo de dato devuelto (de retorno) por la función; cuando devuelve un valor `int`, se puede omitir ya que **por** defecto el C asume que todas las funciones son enteras, a pesar de ello siempre conviene especificar el tipo aun siendo de tipo `int`, para mejor legibilidad. El tipo debe ser uno de los tipos simples de C, tales como `int`, `char` o `float`, o un puntero a cualquier tipo C, o un tipo `struct`.

```
int max(int x, int y)                    /* devuelve un tipo int */
double media(double xl, double x2)       /* devuelve un tipo double */
float func0() {...}                      /* devuelve un float */
char func1() {...}                       /* devuelve un dato char */
int *func3() {...}                       /* devuelve un puntero a int */
char *func4() {...}                      /* devuelve un puntero a char */
int func5() {...}                        /* devuelve un int (es opcional)*/
```

Si una función no devuelve un resultado, se puede utilizar el tipo `void`, que se considera como un tipo de dato especial. Algunas declaraciones de funciones que devuelven distintos tipos de resultados son:

```
int calculo_kilometraje(int litros, int kilometros);
char mayusculas(char car);
float DesvEst (void);
struct InfoPersona BuscarRegistro(int num_registro);
```

Muchas funciones no devuelven resultados. La razón es que se utilizan como ***subrutinas*** para realizar una tarea concreta. Una función que no devuelve un resultado, a veces se denomina ***procedimiento.*** Para indicar al compilador que una función no devuelve resultado, se utiliza el tipo de retorno **`void`**, como en este ejemplo:

```
void VisualizarResultados(float Total, int num-elementos);
```

Si se omite un tipo de retorno para una función, como en

```
VerResultados(float Total, int longitud);
```

el compilador supone que el tipo de dato devuelto es `int`. Aunque el uso de `int` es opcional, por razones de claridad y consistencia se recomienda su uso. Así, la función anterior se puede declarar también:

```
int VerResultados(float Total, int longitud);
```

7.2.3. Resultados de una función

Una función puede devolver un Único valor. El resultado se muestra con una sentencia `return` cuya sintaxis es:

```
return (expresión)
return;
```

El valor devuelto ***(expresión)*** puede ser cualquier tipo de dato excepto una función o un ***array.*** Se pueden devolver valores múltiples devolviendo un puntero o una estructura. El valor de retorno debe seguir las mismas reglas que se aplican a un operador de asignación. Por ejemplo, no se puede devolver un valor `int`, si el tipo de retorno es un puntero. Sin embargo, **si** se devuelve un `int` y el tipo de retorno es un `float`, se realiza la conversión automáticamente.

Una función puede tener cualquier número de sentencias `return`. Tan pronto como el programa encuentra cualquiera de las sentencias `return`, devuelve control a la sentencia llamadora. La ejecución de la función termina si no se encuentra ninguna sentencia `return`; en este caso, la ejecución continúa hasta la llave final del cuerpo de la función.

Si el tipo de retorno es `void`, la sentencia `return` se puede escribir como `return;` sin ninguna expresión de retorno, o bien, de modo alternativo se puede omitir la sentencia `return`.

```
void func1(void)
{
  puts("Esta función no devuelve valores");
}
```

El valor devuelto se suele encerrar entre paréntesis, pero su uso es opcional. En algunos sistemas operativos, como **DOS,** se puede devolver un resultado al entorno llamador. Normalmente el valor 0, se suele devolver en estos casos.

```
int main()
{
  puts("Prueba de un programa C, devuelve 0 al sistema ");
  return 0;
}
```

Consejo

Aunque no es obligatorio el uso de la sentencia `return` en la Última Linea, se recomienda su uso, ya que ayuda a recordar el retorno en ese punto a la función llamadora.

Precaución

Un error típico de programación es olvidar incluir la sentencia **`return`** o situarla dentro de una sección de código que no se ejecute. Si ninguna sentencia return se ejecuta, entonces el resultado que devuelve la función es impredecible y puede originar que su programa falle o produzca resultados incorrectos. Por ejemplo, suponga que se sitúa la sentencia return dentro de una sección de código que se ejecuta condicionalmente, tal como:

```
if (Total >= 0.0)
  return Total;
```

Si `Total` es menor que cero, no se ejecuta la sentencia return y el resultado de la función es un valor aleatorio (C puede generar el mensaje de advertencia "`Function should return a value`", que le ayudará a detectar este posible error).

7.2.4. Llamada a una función

Las funciones, para poder ser ejecutadas, han de ser *llamadas* o *invocadas*. Cualquier expresión puede contener una *llamada a una función* que redirigirá el control del programa a la función nombrada. Normalmente la llamada a una función se realizará desde la función principal `main()`, aunque naturalmente también podrá ser desde otra función.

Nota

La función que llama a otra función se denomina *función llamadora* y la función controlada se denomina *función llamada*.

La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina (se alcanza la sentencia `return`, o la llave de cierre (`}`) si se omite `return`) el control del programa vuelve y retorna a la función `main()` o a la función llamadora si no es `main`.

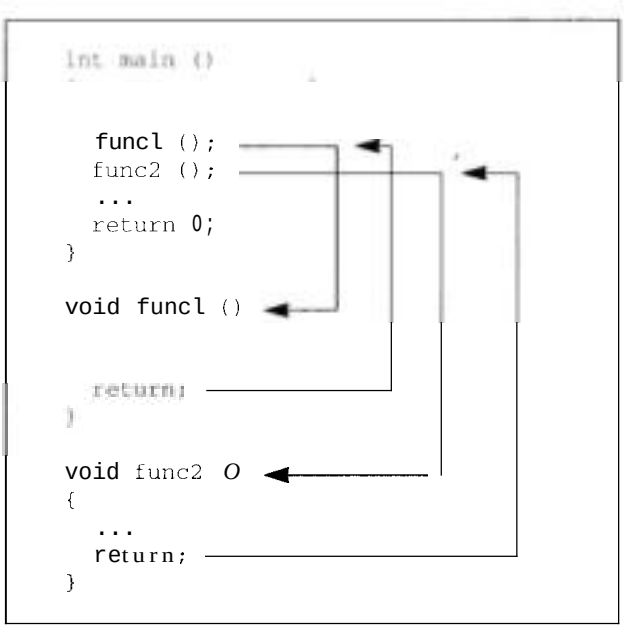


Figura 7.2. Traza de llamadas de funciones.

En el siguiente ejemplo se declaran dos funciones y se llaman desde la función main().

```
#include <stdio.h>

void funcl(void)
{
  puts ("Segunda función");
  return;
}

void func2 (void)
{
  puts ("Tercera función");
  return;
}

int main()
{
  puts("Primera función llamada main()");
  funcl();                          /* Segunda función llamada */
  func2();                          /* Tercera función llamada */
  puts ("main se termina");
  return 0;                         /* Devuelve control al sistema */
}
```

La salida de este programa es:

```
Primera función llamada main()
Segunda función
Tercera función
main se termina
```

Se puede llamar a una función y no utilizar el valor que se devuelve. En esta llamada a función: `func();` el valor de retorno no se considera. El formato `func()` sin argumentos es el más simple. Para indicar que la llamada a una función no tiene argumentos se sitúa una palabra reservada `void` entre paréntesis en la declaración de la función y posteriormente en lo que se denominará *prototipo;* también, con paréntesis vacíos.

```
int main()

  func();                       /* Llamada a la función */
  ...
}

void func(void)                 /* Declaración de la función */
{
  printf ("Función sin argumentos \n");
}
```

Precaución

No se puede definir una función **dentro de otra. Todo código** de la función debe ser listado secuencialmente, a lo largo de todo el programa. Antes de que aparezca el código de una función, debe aparecer la llave de cierre de la función anterior.

Ejemplo 7.2

La función max devuelve el número mayor de dos enteros.

```
#include <stdio.h>
int max(int x, int y)
{
  if (x < y)
    return y;
  else
    return x;
}

int main()
{
  int m, n;
  do {
    scanf("%d %d",&m,&n);
    printf("Maximo de %d,%d es %d\n",max(m, n)); /*llamada a max*/
  }while(m != 0);
  return 0;
}
```

Ejemplo 7.3

Calcular la media aritmética de dos números reales.

```
#include <stdio.h>
double media(double xl, double x2)
{
  return(x1 + x2)/2;
}

int main()
{
  double numl, num2, med;
  printf("Introducir dos números reales:");
  scanf("%lf %lf",&num1,&num2);
  med = media(num1, num2);
  printf ("El valor medio es %.4lf  \n", med);
  return 0;
}
```

7.3. PROTOTIPOS DE LAS FUNCIONES

La *declaración* de una función se denomina *prototipo*. Los prototipos de una función contienen la cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma. Específicamente un prototipo consta de los siguientes elementos: nombre de la función, una lista de argumentos encerrados entre paréntesis y un punto y coma. En C no es estrictamente necesario que una función se declare o defina antes de su uso, no es necesario incluir el prototipo aunque si es recomendable para que el compilador pueda hacer chequeos en las llamadas a las funciones. Los prototipos de las funciones llamadas en un programa se incluyen en la cabecera del programa para que así sean reconocidas en todo el programa.

C recomienda que se declare una función si se llama a la función antes de que se defina.

Sintaxis

tipo_retorno nombre_función (lista_de_declaración_parámetros);

tipo_retorno	Tipo del valor devuelto por la función o palabra reservada `void` si no devuelve un valor.
nombre_función	Nombre de la función.
lista_declaración_parámetros	Lista de declaración de los parámetros de la función, separados por comas (los nombres de los párametros son opcionales, pero es buena práctica incluirlos para indicar lo que representan).

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función está siendo llamada correctamente, con respecto al número y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia.

```
double FahrACelsius(double tempFahr);      /* prototipos válidos */
int max(int x, int y);
int longitud(int h, int a);
struct persona entrad(void);
char* concatenar(char* cl, char*  c2);
double intensidad(double, double);
```

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la función `main()`. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los argumentos reales de la llamada a la función son los mismos que el número y tipo de argumentos **formales** en la función llamada. Si se detecta una inconsistencia, se visualiza un mensaje de error. Sin prototipos, un error puede ocurrir si un argumento con un tipo de dato incorrecto se pasa a una función. En programas complejos, este tipo de errores son difíciles de detectar.

En C, la diferencia entre los conceptos ***declaración*** y ***dejnición*** es preciso tenerla clara. Cuando una entidad se ***declara,*** se proporciona un nombre y se listan ***sus*** características. Una ***definición*** proporciona un nombre de entidad y reserva espacio de memoria para esa entidad. Una ***definición*** indica que existe un lugar en un programa donde «existe» realmente la entidad definida, mientras que una ***declaración*** es sólo una indicación de que algo existe en alguna posición.

Una ***declaración*** de la función contiene sólo la cabecera de la función y una vez declarada la función, la ***definición*** completa de la función debe existir en algún lugar del programa, antes o después de `main()`.

En el siguiente ejemplo se escribe una función `area()` de rectángulo. En la función `main()` se llama a `entrada()` para pedir la base y la altura; a continuación se llama a la función `area()`.

```
#include <stdio.h>

float area_rectangulo(float b, float a);/* declaración */
float entrada();              /* prototipo o declaración */

int main()
{
  float b, h;
  printf("\n Base del rectangulo: ");
  b = entrada();
  printf("\n Altura del rectangulo: ");
  h = entrada();
  printf("\n Area del rectangulo: %.2f",area_rectangulo(b,h));
```

```
  return 0;
}
/* devuelve número positivo */
float entrada()
{
 float m;
 do {
  scanf("%f",&m);
 } while (m<=0.0);
 return m;
}
/* calcula el area de un rectángulo */
float area_rectangulo(float b, float a)
{
  return (b*a);
}
```

En este otro ejemplo se declara la función media

```
#include <stdio.h>
double media (duble xl, double x2);        /*declaración de media*/
int main()
{
   ...
  med = media(num1, num2);
   ...
}
double media(double xl, double x2)         /* definición  */
{
  return (xl + x2)/2;
}
```

- Declaraciones de una función
  - Antes de **que una** función pueda ser invocada, debe ser ***declarada.***
  - **Una declaración de una función** contiene sólo la cabecera de la función (llamado también *prototipo*)

```
tipo_resultado nombre(tipo1 param1, tipo2 param2, ...);
```

- **Los nombres de los parámetros se pueden omitir**

```
char* copiar (char*, int);
char* copiar (char * buffer, int n);
```

La comprobación de tipos es una acción realizada por el compilador. El compilador conoce cuales son los tipos de argumentos que se han pasado una vez que se ha procesado un prototipo. Cuando se encuentra una sentencia de llamada a una función, el compilador confirma que el tipo de argumento en la llamada a la función es el mismo tipo que el del argumento correspondiente del prototipo. Si no son los mismos, el compilador genera un mensaje de error. Un ejemplo de prototipo:

```
int procesar(int a, char b, float c, double d, char *e);
```

El compilador utiliza sólo la información de los tipos de datos. Los nombres de los argumentos, aunque se aconsejan, no tienen significado; el propósito de los nombres es hacer la declaración de tipos más fácil para leer y escribir. La sentencia precedente se puede escribir también así:

```
int procesar(int, char, float, double, char *);
```

Si una función no tiene argumentos, *se* ha de utilizar la palabra reservada `void` como lista de argumentos del prototipo (también se puede escribir paréntesis vacíos).

```
int muestra(void);
```

Ejemplos

```
1./* prototipo de la función cuadrado */
double cuadrado(double);

int main()
{
  double x=11.5;
  printf("%6.2lf al cuadrado = %8.4lf \n",x,cuadrado(x));
  return 0;
}

double cuadrado(double n)
{
  return n*n;
}
```

```
2. /* prototipo de visualizar-nombre */
void visualizar_nombre(char*);

void main()
{
  visualizar-nombre("Lucas El Fuerte");
}

void visualizar_nombre(char* nom)
{
  printf("Hola %s \n",nom);
}
```

7.3.1. Prototipos con un número no especificado de parametros

Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por puntos suspensivos (...). Por ejemplo,

```
int muestras(int a, ...);
int printf(const char *formato, ...);
int scanf(const char *formato, ...);
```

Para implementar una función con lista variable de parámetros es necesario utilizar unas macros (especie de funciones en línea) que están definidas en el archivo de cabecera `stdarg.h`, por consiguiente lo primero que hay que hacer es incluir dicho archivo.

```
#include <stdarg.h>
```

En el archivo está declarado el tipo `va_list`, un puntero para manejar la lista de datos pasada a la función.

```
val-list puntero;
```

La función va-start() inicializa puntero, de tal forma que referencia al primer parámetro variable. El prototipo que tiene:

void va-start(va-list puntero, ultimofijo);

El segundo argumento es el Último argumento fijo de la función que se está implementando. Así para la función muestras(int a, ...);

```
va-start(puntero,a);
```

Con la función va-arg() se obtienen, consecutivamente, los sucesivos argumentos de la lista variable. El prototipo que tiene

```
tipo va_arg(va_list puntero, tipo);
```

Donde tipo es el tipo del argumento variable que es captado en ese momento, a su vez es el tipo de dato que devuelve va-arg(). Para la función muestras() si los argumentos variables son de tipo int:

```
int m;
m  = va_arg(puntero,int);
```

La Última llamada que hay que hacer en la implementación de estas funciones es a va-end(). De esta forma se queda el puntero preparado para siguientes llamadas. El prototipo que tiene va-end() :

```
void  va_end(va_list puntero).
```

Ejercicio 7.1

Una aplicación completa de una función con lista de argumentos variables ***es maximo(int, ...)****, que calcula el máximo de n argumentos de tipo double, donde n es el argumento fijo que* ***se*** *utiliza.*

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

void maximo(int n,...);

int main(void)
{
    puts("\t\tPRIMERA BUSQUEDA DEL MAXIMO\n");
    maximo(6,3.0,4.0,-12.5,1.2,4.5,6.4);
    puts("\n\t\tNUEVA BUSQUEDA DEL MAXIMO\n");
    maximo(4,5.4,17.8,5.9,-17.99);
    return 0;
}

void maximo(int n, ...)
{
  double mx,actual;
  va-list puntero;
  int i;
  va_start(puntero,n);
  mx = actual = va_arg(puntero,double);
  printf("\t\tArgumento actual: %.21f\n",actual);
  for (i=2; i<=n; i++)
  {
     actual = va_arg(puntero,double);
     printf("\t\tArgumento actual: %.21f\n",actual);
     if (actual > mx)
     {
```

```
            mx = actual;
          }
        }
        printf("\t\tMáximo de la lista de %d números es %.2lf\n",n,mx);
        va_end(puntero);
    }
```

7.4. PARÁMETROS DE UNA FUNCIÓN

C siempre utiliza el método de *parámetros por valor* para pasar variables a funciones. Para que una función devuelva un valor a través de un argumento hay que pasar la dirección de la variable, y que el argumento correspondiente de la función sea un puntero, es la forma de conseguir en C un paso de *parámetro por referencia.* Esta sección examina el mecanismo que C utiliza para pasar parámetros a funciones y cómo optimizar el paso de parámetros, dependiendo del tipo de dato que se utiliza. Suponiendo que se tenga la declaración de una función **circulo** con tres argumentos

```
void circulo(int x, int y, int diametro);
```

Cuando se llama a **circulo** se deben pasar tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión, como en el siguiente ejemplo:

```
circulo(25, 40, vueltas*4);
```

7.4.1. Paso de parámetros por valor

Paso por valor (también llamadopaso *por copia)* significa que cuando C compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro variable local, el cambio sólo afecta a la función y no tiene efecto fuera de ella.

La Figura 7.3 muestra la acción de pasar un argumento por valor. La variable real *i* no se pasa, pero el valor de *i*, 6, se pasa a la función receptora.

En la técnica de paso de parámetro por valor, la modificación de la variable (parámetro pasado) en la función receptora no afecta al parámetro argumento en la función llamadora.

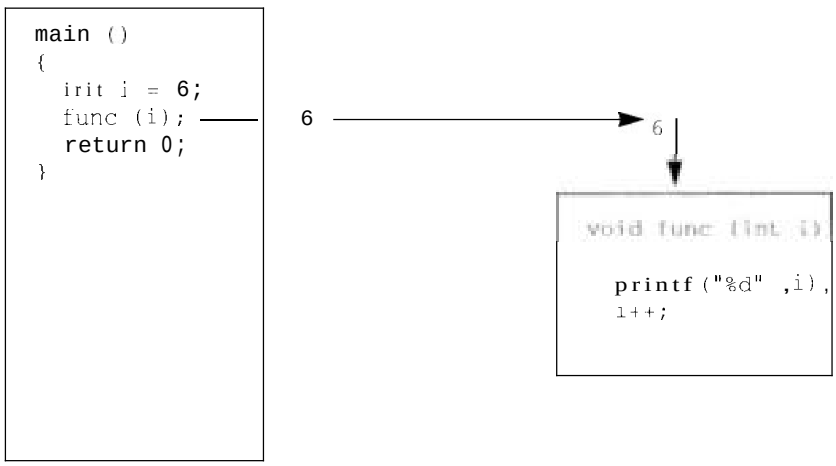


Figura 7.3. Paso de la variable *i* **por** valor.

Nota

El **método por defecto de pasar parámetros es por valor, a menos que se pasen arrays. Los arrays se** pasan **siempre por dirección.**

El siguiente programa muestra el mecanismo de paso de parámetros por valor.

```
/*
   Muestra el paso de parámetros por valor
   Se puede cambiar la variable del parámetro en la función
   pero su modificación no puede salir al exterior
*/
#include <stdio.h>
void DemoLocal(int valor);
void main(void)
{
  int n = 10;
  printf("Antes de llamar a DemoLocal, n = %d\n",n);
  DemoLocal(n);
  printf("Después de llamada a DemoLocal, n = %d\n",n);
}
void DemoLocal(int valor)
{
  printf("Dentro de DemoLocal, valor = %d\n",valor);
  valor = 999;
  printf("Dentro de DemoLocal, valor = %d\n",valor);
}
```

Al ejecutar este programa se visualiza la salida:

```
Antes de llamar a DemoLocal, n = 10
Dentro de DemoLocal, valor = 10
Dentro de DemoLocal, valor = 999
Después de llamar a DemoLocal, n = 10
```

7.4.2. Paso de parámetros por referencia

Cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por ***referencia*** o *dirección.*

En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (la variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado. Para pasar una variable por referencia, el símbolo & debe preceder al nombre de la variable y el parámetro variable correspondiente de la función debe declararse como puntero.

```
float x;
int y;
entrada(&x,&y);
   . . .
void entrada(float* x, int* y)
```

C permite utilizar punteros para implementar parámetros por referencia, ya que por defecto en C el paso de parámetros es por valor.

```
/* método de paso por referencia, mediante punteros */
void intercambio(int* a, int* b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}
```

En la llamada siguiente, la función **intercambio()** utiliza las expresiones **a* y **b* para acceder a los enteros referenciados por las direcciones de las variables *i* y *j*:

```
int i = 3, j = 50;
printf("i = %d y j = %d \n", i,j);
intercambio(&i, &j);
printf("i = %d y j = %d \n", i,j);
```

La llamada a la función **intercambio()** debe pasar las direcciones de las variables intercambiadas. El operador & delante de una variable significa *«dame la dirección **de** la variable»*.

```
double x;
&x ;  /* dirección en memoria de x */
```

Una variable, o parámetro puntero se declara poniendo el asterisco (*) antes del nombre de la variable. Las variables *p*, *r* y *q* son punteros a distintos tipos.

```
char* p;   /* variable puntero a char */
int * r;   /* variable puntero a int */
double* q; /* variable puntero a double */
```

7.4.3. Diferencias entre paso de variables por valor y por referencia

Las reglas que se han de seguir cuando se transmiten variables por valor y por referencia son las siguientes:

- los parámetros valor reciben copias de los valores de los argumentos que se les pasan;
- la asignación a parámetros valor de una función nunca cambian el valor del argumento original pasado a los parámetros;
- los parámetros para el paso por referencia (declarados con *, punteros) reciben la dirección de los argumentos pasados; a estos les debe de preceder del operador &, excepto los arrays;
- en una función, las asignaciones a parámetros referencia (punteros) cambian los valores de los argumentos originales.

Por ejemplo, la escritura de una función **potrat()** para cambiar los contenidos de dos variables, requiere que los datos puedan ser modificados.

Paso por valor

```
float a, b;

potrat1(float x,float y)
{

}
```

*Paso **par** referencia*

```
float a, b;

potrat2(float* x,float* y)
{

}
```

Sólo en el caso de potrat2 los valores de a y b se cambiarán. Veamos una aplicación completa de ambas funciones:

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

```
void potrat1(float, float);
void potrat2(float*, float*)
void main()
{
   float a, b;
   a = 5.0; b = 1.0e2;
   potratl(a,b);
   printf("\n a = %.1f  b = %.1f",a,b);
   potrat2(a,b);
   printf("\n a = %.1f  b = %.1f",a,b);
}
void potrat1(float x, float y)
{
   X = x*x;
   Y = sqrt(y);
I
void potrat2(float* x, float* y)
{
   *x = (*x)*(*x);
   *y = sqrt(*y);
}
```

La ejecución del programa producirá:

```
a = 5.0 b = 100.0
a = 25.0 b = 10.0
```

Nota

Todos **los parámetros en C se pasan por valor. C no tiene parámetros por referencia, hay que hacerlo con punteros y el operador** &.

Se puede observar en el programa cómo se accede a los punteros, el operador * precediendo al parámetro puntero devuelve el contenido.

7.4.4. Parámetros const de una función

Con el objeto de añadir seguridad adicional a las funciones, se puede añadir a una descripción de un parámetro el especificador `const`,que indica al compilador que sólo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación.

```
void f1(const int, const int*);
void f2(int, int const*);
void f1(const int x, const int* y)
{
  x = 10;   /* error por cambiar un objeto constante*/
  *y = 11;  /* error por cambiar un objeto constante*/
  y = &x;   /* correcto */
}
void f2(int x,  int const* y)
{
  x = 10;   /* correcto */
  *y = 11;  /*  error */
  y = &x;   /*  correcto */
}
```

La Tabla 7.1 muestra un resumen del comportamiento de los diferentes tipos de parámetros.

Tabla 7.1. *Paso* de **parámetros** en C.

Parámetro especificado como:	Item pasado por	Cambia item dentro de la función	Modifica parámetros al exterior
`int item`	*valor*	*Si*	*NO*
`const int item`	*valor*	*NO*	*No*
`int* item`	*por* ***dirección***	*Si*	*Si*
`const int* item`	***por*** *dirección*	*No* ***su contenido***	*NO*

7.5. FUNCIONES EN LINEA, MACROS CON ARGUMENTOS

Una función normal es un bloque de código que se llama desde otra función. El compilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. **A** continuación, el compilador genera código que sitúa cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

```
float  fesp(float x)
{
  return (x*x + 2*x -1);
}
```

Las funciones en línea sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función es una expresión, **su** código es pequeño y se utiliza muchas veces en el programa. Realmente no son funciones, el preprocesador expande o sustituye la expresión cada vez que es llamada. **Así** la anterior función puede sustituirse:

```
#define fesp(x) (x*x + 2*x -1)
```

En este programa se realizan cálculos de la función para valores de x en un intervalo.

```
#include <stdio.h>
#define fesp(x) (x*x + 2*x -1)

void main()
{
   float x;
   for (x = 0.0; x <=6.5; x += 0.3)
     printf("\t f(%.1f) = %6.2f ",x, fesp(x));
}
```

Antes de que el compilador construya el código ejecutable de este programa, el preprocesador sustituye toda llamada a `fexp(x)` por la expresión asociada. Realmente es como si hubiéramos escrito

```
printf("\t f(%.1f) = %6.2f ",x, (x*x + 2*x -1));
```

Para una *macro con argumentos (función en línea),* el compilador inserta realmente el código en el punto en que se llama, esta acción hace que el programa se ejecute más rápidamente, ya que no ha de ejecutar el código asociado con la llamada a la función.

Sin embargo, cada invocación a una macro puede requerir tanta memoria como se requiera para contener la expresión completa que representa. Por esta razón, el programa incrementa su tamaño, aunque es mucho más rápido en su ejecución. Si se llama a una macro diez veces en un programa, el compilador inserta diez copias de ella en el programa. Si la macrofunción ocupa 0.1K, el tamaño de su programa se incrementa en 1K (1024 bytes). Por el contrario, si se llama diez veces a la misma función

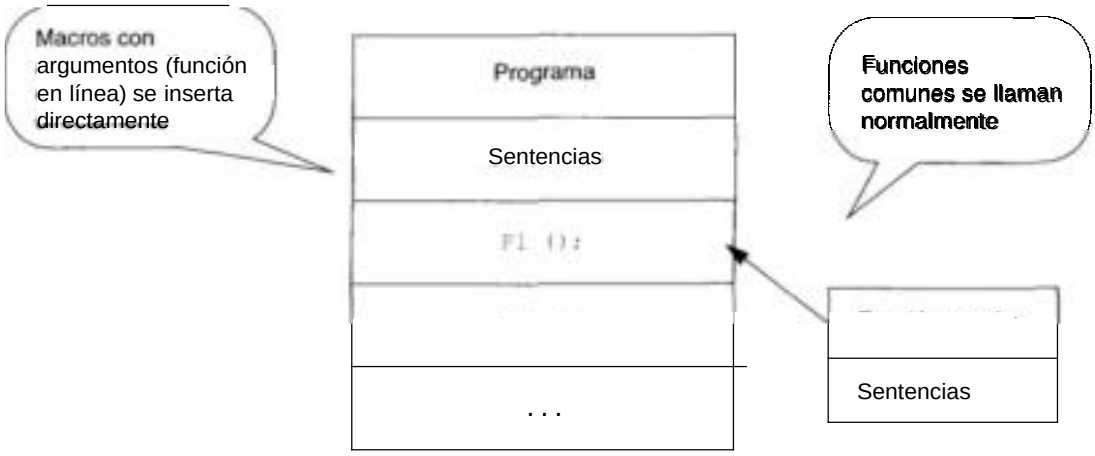


Figura 7.4. Código generado por una función fuera de línea.

con una función normal, y el código de llamada suplementario es 25 bytes por cada llamada, el tamaño se incrementa en una cantidad insignificante.

La Figura 7.5 ilustra la sintaxis general de una macro con argumentos.

```
#define NombreMacro(parámetros sin tipos) expresión-texto
```

REGLA: La definición de una macro sólo puede ocupar una línea. Se puede prolongar la línea con el carácter \ al final de la línea.

Figura 7.5. Código de una macro con argumentos.

La Tabla 7.2 resume las ventajas y desventajas de situar un código de una función en una macro o fuera de línea (función normal):

Tabla 7.2. Ventajas y desventajas de macros.

	Ventajas	Desventajas
Macros (funciones en línea)	Rápida de ejecutar.	Tamaño de código grande.
Funciones fuera de línea	Pequeño tamaño de código.	Lenta de ejecución.

7.5.1. Creación de macros con argumentos

Para crear una macro con argumentos utilizar la sintaxis:

```
#define NombreMacro(parámetros sin tipos) expresión-texto
```

La definición ocupará sólo una línea, aunque si se necesitan más texto, situar una barra invertida (\) al final de la primera línea y continuar en la siguiente, en caso de ser necesarias más líneas proceder de igual forma; de esa forma se puede formar una expresión más compleja. Entre el nombre de la macro y los paréntesis de la lista de argumentos no puede haber espacios en blanco. Por ejemplo, la función media de tres valores se puede escribir:

```
#define MEDIA3(x,y,z) ((x) + (y) + (z))/3.0
```

En este segmento de código se invoca a MEDIA3

```
double a = 2.9;
printf("\t %lf ", MEDIA3(a,4.5,7));
```

En esta llamada a MEDIA3 se pasan argumentos de tipo distinto. Es importante tener en cuenta que en las macros con argumentos *no hay comprobación de tipos.* Para evitar problemas de prioridad de operadores, es conveniente encerrar entre paréntesis cada argumento en la expresión de definición e incluso encerrar entre paréntesis toda la expresión.

En la siguiente macro, la definición de la expresión ocupa más de una línea.

```
#define FUNCION3(x)  {                      \
                        if  ((x) <-1.0 )    \
                          (-(x)*(x) +3);    \
                        else if  ((x)<=1)   \
                          (2*(x)+5);        \
                        else                \
                          ((x)*(x)-5);      \
                     }
```

Al tener la macro más de una sentencia, encerrarla entre llaves hace que sea una sola sentencia, aunque sea compuesta.

Ejercicio 7.2

Una aplicación completa de una macro con argumentos es `VolCono()`*, que calcula el volumen de la figura geométrica Cono.*

$$(V = \frac{1}{3}\pi r^2 h)$$

```
#include <stdio.h>
#define Pi 3.141592

#define VOLCONO(radio,altura) ((Pi*(radio*radio)*altura)/3.0)

int main()
{
  float radio, altura, volumen;

  printf("\nIntroduzca radio del cono: ");
  scanf("%f",&radio);
  printf ("Introduzca altura del cono: ");
  scanf("%f",&altura);
  volumen = VOLCONO(radio, altura);
  printf("\nEl volumen del cono es: %.2f",volumen);
  return 0;
}
```

7.6. ÁMBITO (ALCANCE)

El *ámbito* o *alcance* de una variable determina cuáles son las funciones que reconocen ciertas variables. Si una función reconoce una variable, la variable es *visible* en esa función. El ámbito es la zona de un programa en la que es visible una variable. Existen cuatro tipos de ámbitos: *programa, archivofiente, función* y *bloque.* Se puede designar una variable para que esté asociada a uno de estos ámbitos. Tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito.

Normalmente la posición de la sentencia en el programa determina el ámbito. Los especificadores de clases de almacenamiento, `static, extern, auto` y `register`, pueden afectar al ámbito. El siguiente fragmento de programa ilustra cada tipo de ámbito:

```
int i;                  /* Ámbito de programa */
static int j;           /* Ámbito de archivo */
float func(int k)       /* k, ámbito de función */
{

      int m;            /* Ámbito de bloque */
      ...
   }
}
```

7.6.1. Ámbito del programa

Las variables que tienen *ámbito* ***de*** *programa* pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman *variables globules*. Para hacer una variable global, declárela simplemente al principio de un programa, fuera de cualquier función.

```
int g, h;               /*  variables globales */
main()
{
  ...
}
```

Una variable global es visible **(«se** conocen) desde su punto de definición en el archivo fuente. Es decir, si se define una variable global, cualquier línea del resto del programa, no importa cuantas funciones y líneas de código le sigan, podrá utilizar esa variable.

```
#include <stdio.h>
#include <math.h>
float ventas, beneficios;           /* variables globales */
void f3(void)


}
void f1(void)
{
...
}

void main()
{
...
}
```

Consejo

Declare todas **las variables en la parte superior de su programa. Aunque se pueden definir tales variables entre dos funciones, podría realizar cualquier cambio en su programa de modo más rápido, si sitúa las variables globales al principio del programa.**

7.6.2. Ámbito del archivo fuente

Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada `static` tiene *ámbito de archivo fuente*. Las variables con este ámbito se pueden referencia desde el punto del programa en que están declaradas hasta el final del archivo fuente. Si un archivo fuente tiene más de una función, todas las funciones que siguen a la declaración de la variable pueden referenciarla. En el ejemplo siguiente, *i* tiene ámbito de archivo fuente:

```
static int i;
void func(void)
{

}
```

7.6.3. Ámbito de una función

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. Las variables locales no se pueden utilizar fuera del ámbito de la función en que están definidas.

```
void calculo(void)
{
   double x, r, t ;  /* Ámbito de la función */
   ...
}
```

7.6.4. Ámbito de bloque

Una variable declarada en un bloque tiene *ámbito de bloque* y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declaradas dentro de una función tienen ámbito de bloque de la función; no son visibles fuera del bloque. En el siguiente ejemplo, *i* **es** una variable local:

```
void funcl(int x)
{
  int i;
  for (i = x; i < x+10; i++)
    printf("i = %d \n",i*i);
}
```

Una variable local declarada en un bloque anidado sólo es visible en el interior de ese bloque.

```
float func(int j)
{
  if (j > 3)
  I
    int i;
    for (i = 0; i < 20; i++)
      funcl(i);
  }
/* aquí ya no es visible i */
};
```

7.6.5. Variables locales

Además de tener un ámbito restringido, las variables locales son especiales por otra razón: existen en memoria sólo cuando la función está activa (es decir, mientras se ejecutan las sentencias de la función). Cuando la función no se está ejecutando, sus variables locales no ocupan espacio en memoria, ya que no existen. Algunas reglas que siguen las variables locales son:

- Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir la misma variable `test`. Cada variable es distinta y pertenece a su función específica.
- Las variables locales de las funciones no existen en tnemoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden compartir la misma memoria para sus variables locales (pero no al mismo tiempo).

7.7. CLASES DE ALMACENAMIENTO

Los especificadores de clases (tipos) de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes: `auto, extern, register, static` y `typedef`.

7.7.1. Variables automáticas

Las variables que se declaran dentro de una función se dice que son automáticas (auto), significando que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada `auto` es opcional.

```
auto int Total;        es igual que        int Total;
```

Normalmente no se especifica la palabra `auto`.

7.7.2. Variables externas

A veces se presenta el problema de que una función necesita utilizar una variable que *otra función* inicializa. Como las variables locales sólo existen temporalmente mientras se está ejecutando su función, no pueden resolver el problema. ¿Cómo se puede resolver entonces el problema? En esencia, de lo que se trata es de que una función de un archivo de código fuente utilice una variable definida en otro archivo. Una solución es declarar la variable local con la palabra reservada `extern`. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro lugar.

```
/* variables externas: parte 1 */
/* archivo fuente exter1.c */
#include <stdio.h>

extern void leerReal(void);/* función definida en otro archivo; en este
                              caso no es necesario extern */

float f;

int main()
{
    leerReal();
    printf("Valor de f = %f",f);
    return 0;
}
```

```
/*variables externas: parte 2 */
/* archivo fuente exter2.c */
#include <stdio.h>

void leerReal(void)
{
    extern float f;

    printf("Introduzca valor en coma flotante: ");
    scanf("%f",&f);
}
```

En el archivo EXTER2.C la declaración externa de *f* indica al compilador que *f* se ha definido en otra parte (archivo). Posteriormente, cuando estos archivos se enlacen, las declaraciones se combinan de modo que se referirán a las mismas posiciones de memoria.

7.7.3. Variables registro

Otro tipo de variable C es la *variable registro*. Precediendo a la declaración de una variable con la palabra reservada `register`, se sugiere al compilador que la variable se almacene en uno de los registros hardware del microprocesador. La palabra register es una sugerencia al compilador y no una orden. La familia de microprocesadores 80x86 no tiene muchos registros hardware de reserva, por lo que el compilador puede decidir ignorar sus sugerencias. Para declarar una variable registro, utilice una declaración similar a:

```
register int k;
```

Una variable registro debe ser local a una función, nunca puede ser global al programa completo.

El uso de la variable `register` no garantiza que un valor se almacene en un registro. Esto sólo sucederá si existe un registro disponible. Si no existen registros suficientes, C ignora la palabra reservada `register` y crea la variable localmente como ya se conoce.

Una aplicación típica de una variable registro es como variable de control de un bucle. Guardando la variable de control de un bucle en un registro, se reduce el tiempo que la CPU requiere para buscar el valor de la variable de la memoria. Por ejemplo,

```
register int indice;
for (indice = 0; indice < 1000; indice++)...
```

7.7.4. Variables estáticas

Las variables estáticas son opuestas, en su significado, a las variables automáticas. Las *variables estáticas* no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable `static` se inicializa sólo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada `static`.

```
func_uno()
{
  int i;
  static int j = 25;          /*j, k variables estáticas */
  static int k = 100;
  ...
}
```

Las variables estáticas se utilizan normalmente para mantener valores entre llamadas a funciones.

```
float ResultadosTotales(float valor)
{
  static float suma;

  suma = suma + valor;
  return suma;
}
```

En la función anterior se utiliza `suma` para acumular sumas a través de sucesivas llamadas a `ResultadosSota1es`.

Ejercicio 7.3

Una aplicación de una variable static en una función es la que nos permite obtener la serie de números de fibonacci. El ejercicio lo plantearnos: dado un entero n, *obtener los* n *primeros números de la serie de fibonacci.*

Análisis

La secuencia de números de fibonacci: 0, 1, 1, 2, 3, 5, 8, 13..., se obtiene partiendo de los números 0, 1 y a partir de ellos cada número se obtiene sumando los dos anteriores:

$$a_n = a_{n-1} + a_{n-2}$$

La función fibonacci tiene dos variables estáticas, x e y. Se inicializan x a 0 e y a 1; a partir de esos valores se calcula el valor actual, y, se deja preparado x para la siguiente llamada. Al ser variables estáticas mantienen el valor entre llamada y llamada.

```
#include <stdio.h>
long int fibonacci();
int main()
{
  int n,i;
  printf("\nCuantos numeros de fibonacci ?: ");
  scanf("%d",&n);
  printf("\nSecuencia de fibonacci: 0,1");
  for (i=2; i<n; i++)
    printf(",%ld",fibonacci());
  return 0;
}
long int fibonacci()
{
  static int x = 0;
  static int y = 1;
  y = y + x;
  x = y - x;
  return y;
}
```

Ejecución

```
Cuantos numeros de fibonacci ? 14
Secuencia de fibonacci: 0,1,1,2,3,5,8,13,21,34,55,89,144,233
```

7.8. CONCEPTO Y USO DE FUNCIONES DE BIBLIOTECA

Todas las versiones del lenguaje **C** ofrecen con una biblioteca estándar de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las *funciones estándar* o *predefinidas*, como así se denominan las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo *archivo de cabecera.*

Los nombres de los archivos de cabecera estándar utilizados en nuestro programa se muestran a continuación encerrados entre corchetes tipo ángulo:

<assert.h>	<ctype.h>	<errno.h>	<float.h>
<limits.h>	<math.h>	<setjmp.h>	<signal.h>
<stdarg.h>	<stdef.h>	<stdio.h>	<string.h>
<time.h>			

En los módulos de programa se pueden incluir líneas `#include` con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez.

Para utilizar una función o un macro, se debe conocer su número de argumentos, sus tipos y el tipo de sus valores de retorno. Esta información se proporcionará en los prototipos de la función. La sentencia `#include` mezcla el archivo de cabecera en su programa.

Algunos de los grupos de funciones de biblioteca más usuales son:

- E/S estándar (para operaciones de Entrada/Salida);
- matemáticas (para operaciones matemáticas);
- rutinas estándar (para operaciones estándar de programas);
- visualizar ventana de texto;
- de conversión (rutinas de conversión de caracteres y cadenas);
- de diagnóstico (proporcionan rutinas de depuración incorporada);
- de manipulación de memoria;
- control del proceso;
- clasificación (ordenación);
- directorios;
- fecha y hora;
- de interfaz;
- diversas;
- búsqueda;
- manipulación de cadenas;
- gráficos.

Se pueden incluir tantos archivos de cabecera como sean necesarios en sus archivos de programa, incluyendo sus propios archivos de cabecera que definen sus propias funciones.

En este capítulo se estudiarán las funciones más sobresalientes y más utilizadas en programación.

7.9. FUNCIONES DE CARÁCTER

El archivo de cabecera <CTYPE.H> define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones devuelven un resultado de valor verdadero (distinto de cero) o falso (cero).

Para utilizar cualquiera de las funciones (Tabla **7.3)** no se olvide incluir el archivo de cabecera CTYPE.H en la parte superior de cualquier programa que haga uso de esas funciones.

Tabla 7.3. Funciones de caracteres.

Función	Prueba (test) de
`int isalpha(int c)`	Letra mayúscula o minúscula.
`int isdigit(int c)`	Dígito decimal.
`int isupper(int c)`	Letra mayúscula (A-Z).
`int islower(int c)`	Letra minúscula (a-z).
`int isalnum(int c)`	`letra o dígito; isalpha(c)||isdigit(c)`
`int iscntrl(int c)`	Carácter de control.
`int isxdigit(int c)`	Dígito hexadecimal.
`int isprint(int c)`	Carácter imprimible incluyendo ESPACIO.
`int isgraph(int c)`	Carácter imprimible excepto ESPACIO.
`int isspace(int c)`	ESPACIO, AVANCE DE PÁGINA, NUEVA LÍNEA, RETORNO DE CARRO, TABULACIÓN, TABULACIÓN VERTICAL.
`int ispunct(int c)`	Carácter imprimible no espacio, dígito o letra.
`int toupper(int c)`	Convierte a letras mayúsculas.
`int tolower(int c)`	Convierte a letras minúsculas.

7.9.1. Comprobación alfabética y de dígitos

Existen varias funciones que sirven para comprobar condiciones alfabéticas:

- **`isalpha(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra mayúscula o minúscula. Se devuelve un valor falso si se pasa un carácter distinto de letra a esta función.
- **`islower(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra minúscula. Se devuelve un valor falso (0), si se pasa un carácter distinto de una minúscula.
- **`isupper(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra mayúscula, falso con cualquier otro carácter.

Las siguientes funciones comprueban caracteres numéricos:

- **`isdigit(c)`**
  Comprueba si `c` es un dígito de 0 a 9, devolviendo verdadero (distinto de cero) en ese caso, y falso en caso contrario.
- **`isxdigit(c)`**
  Devuelve verdadero si `c` es cualquier dígito hexadecimal (0 a 9, A a F, o bien a a f) y falso en cualquier otro caso.

Las siguientes funciones comprueban argumentos numéricos o alfabéticos:

- **`isalnum(c)`**
  Devuelve un valor verdadero, si `c` es un dígito de 0 a 9 o un carácter alfabético (bien mayúscula o minúscula) y falso en cualquier otro caso.

Ejemplo 7.4

Leer un carácter del teclado y comprobar si es una letra.

```
/*
  Solicita iniciales y comprueba que es alfabética
*/
#include <stdio.h>
#include <ctype.h>
int main()
{
  char inicial;
  printf("¿Cuál es su primer carácter inicial?: ");
  scanf("%c",&inicial);
  while (!isalpha(inicial))
  {
    puts ("Carácter no alfabético ");
    printf("¿Cuál es su siguiente inicial?: ");
    scanf("%c",&inicial);
  }
  puts("¡Terminado!");
  return 0;
}
```

7.9.2. Funciones de prueba de caracteres especiales

Algunas funciones incorporadas a la biblioteca de funciones comprueban caracteres especiales, principalmente a efectos de legibilidad. Estas funciones son las siguientes:

- **iscntrl(c)**
  Devuelve verdadero si c es un *carácter de control* (códigos ASCII 0 a 31) y falso en caso contrario.
- **isgraph(c)**
  Devuelve verdadero si c es un carácter imprimible (no de control) excepto espacio; en caso contrario, se devuelve falso.
- **isprint(c)**
  Devuelve verdadero si c es un carácter imprimible (código ASCII 32 a 127) incluyendo un espacio; en caso contrario, se devuelve falso.
- **ispunct(c)**
  Devuelve verdadero si c es cualquier carácter de puntuación (un carácter imprimible distinto de espacio, letra o dígito); falso, en caso contrario.
- **isspace(c)**
  Devuelve verdadero si c es carácter un espacio, nueva línea (\n), retorno de carro (\r), tabulación (\t) o tabulación vertical (\v).

7.9.3. Funciones de conversión de caracteres

Existen funciones que sirven para cambiar caracteres mayúsculas a minúsculas o viceversa.

- **tolower(c)**
  Convierte el carácter c a minúscula, si ya no lo es.
- **toupper(c)**
  Convierte el carácter c a mayúscula, si ya no lo es.

Ejemplo *7.5*

El programa MAYMIN1.C *comprueba si la entrada es una* V *o una* H.

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

int main()
{
  char resp;      /* respuesta del usuario */
  char c;

  printf("¿Es un varón o una hembra (V/H)?: ");
  scanf("%c",&resp);
  resp=toupper(resp);
  switch (resp)
  {
    case 'V':
     puts("Es un enfermero");
     break;
    case 'H':
     puts("Es una maestra");
     break;
    default:
     puts("No es ni enfermero ni maestra");
     break;
  }
  return 0;
}
```

7.10. FUNCIONES NUMÉRICAS

Virtualmente cualquier operación aritmética es posible en un programa C. Las funciones matemáticas disponibles son las siguientes:

- matemáticas;
- trigonométricas;
- logarítmicas;
- exponenciales;
- aleatorias.

La mayoría de las funciones numéricas están en el archivo de cabecera MATH.H; las funciones **abs** y **labs** están definidas en MATH.H y STDLIB.H, y las rutinas div y **ldiv** en STDLIB.H.

7.10.1. Funciones matemáticas

Las funciones matemáticas usuales en la biblioteca estándar son:

- **ceil(x)**
  Redondea al entero más cercano.
- **fabs(x)**
  Devuelve el valor absoluto de x (un valor positivo).
- **floor(x)**
  Redondea por defecto al entero más próximo.

- **fmod(x, y)**
  Calcula el resto *f* en coma flotante para la división *x/y*, de modo que x = *i**y+*f*, donde i es un entero, *f* tiene el mismo signo que x y el valor absoluto de *f* es menor que el valor absoluto de *y*.
- **pow(x, y)**
  Calcula x elevado a la potencia *y* (x^y). Si x es menor que o igual a cero, *y* debe ser un entero. Si *x* es igual a cero, *y* no puede ser negativo.
- **pow10(x)**
  Calcula 10 elevado a la potencia x (IO); x debe ser de tipo entero.
- **sqrt(x)**
  Devuelve la raíz cuadrada de *x;* x debe ser mayor o igual a cero.

7.10.2. Funciones trigonométricas

La biblioteca de C incluye una serie de funciones que sirven para realizar cálculos trigonométricos. Es necesario incluir en su programa el archivo de cabecera MATH. H para utilizar cualquier función.

- **acos(x)**
  Calcula el arco coseno del argumento x. El argumento x debe estar entre –1 y 1.
- **asin(x)**
  Calcula el arco seno del argumento x. El argumento x debe estar entre –1 y 1.
- **atan(x)**
  Calcula el arco tangente del argumento x.
- **atan2(x,y)**
  Calcula el arco tangente de x dividido por y.
- **cos(x)**
  Calcula el coseno del ángulo x ; x se expresa en radianes.
- **sin(x)**
  Calcula el seno del ángulo x;x se expresa en radianes.
- **tan(x)**
  Devuelve la tangente del ángulo x ; x se expresa en radianes.

Regla

Si necesita pasar un ángulo expresado en *grados* a radianes, para poder utilizarlo con las funciones trigonométricas, multiplique los grados por pi/180, donde pi = 3.14159.

7.10.3. Funciones logarítmicas y exponenciales

Las funciones logarítmicas y exponenciales suelen ser utilizadas con frecuencia no sólo en matemáticas, sino también en el mundo de la empresa y los negocios. Estas funciones requieren también el archivo de inclusión MATH. H.

- **exp(x), expl(x)**
  Calcula el exponencial e , donde e es la base de logaritmos naturales de valor 2.7 18282.

  ```
  valor = exp(5.0);
  ```

  Una variante de esta función es expl, que calcula e utilizando un valor long double (largo doble).

- **log(x), logl(x)**
  La función **log** calcula el logaritmo natural del argumento x y logl(x) calcula el citado logaritmo natural del argumento x de valor long double (largo doble).
- **log10(x), log10l(x)**
  Calcula el logaritmo decimal del argumento x, de valor real double en log10(x) y de valor real long double en log10l(x); x ha de ser positivo.

7.10.4. Funciones aleatorias

Los números aleatorios son de gran utilidad en numerosas aplicaciones y requieren un trato especial en cualquier lenguaje de programación. C no es una excepción y la mayoría de los compiladores incorporan funciones que generan números aleatorios. Las funciones usuales de la biblioteca estándar de C son: **rand, random, randomize** y **srand.** Estas funciones se encuentra en el archivo STDLIB.H.

- **rand(void)**

La función **rand** genera un número aleatorio. El número calculado por **rand** varía en el rango entero de 0 a RAND—MAX. La constante RAND-MAX se define en el archivo STDLIB.H en forma hexadecimal (por ejemplo, 7FFF). En consecuencia, asegúrese incluir dicho archivo en la parte superior de su programa.

Cada vez que se llama a **rand()** en el mismo programa, se obtiene un número entero diferente. Sin embargo, si el programa se ejecuta una y otra vez, se devuelven el mismo conjunto de números aleatorios. Un método para obtener un conjunto diferente de números aleatorios es llamar a la función **srand()** o a la macro **randomize.**

La llamada a la función **rand()** se puede asignar a una variable o situar en la función de salida printf().

```
test = rand();
printf("Este es un número aleatorio %d\n",rand());
```

- **randomize(void)**

La macro randomize inicializa el generador de números aleatorios con una semilla aleatoria obtenida a partir de una llamada a la función **time.** Dado que esta macro llama a la función **time,** el archivo de cabecera TIME.H se incluirá en el programa. No devuelve ningún valor.

```
/* progrma para generar 10 números aleatorios */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

int main(void)
{
  int i;

  clrscr(); /* limpia la pantalla */
  randomize();
  for (i=1; i<=10; i++)
   printf("%d  ",rand());

  return 0;
}
```

- **srand(semilla)**

La función **srand** inicializa el generador de números aleatorios. Se utiliza para fijar el punto de comienzo para la generación de series de números aleatorios; este valor se denomina semilla.

Si el valor de semilla es 1, se reinicializa el generador de números aleatorios. Cuando se llama a la función **rand** antes de hacer una llamada a la función **srand,** se genera la misma secuencia que si se hubiese llamado a la función **srand** con el argumento semilla tomando el valor 1.

- **random(num)**

  La macro random genera un número aleatorio dentro de un rango especificado (0 y el límite superior especificado por el argumento num). Devuelve un número entero entre 0 y num-1.

```
/*
  programa para generar encontrar el mayor de 10 números aleatorios
  entre 0 y 1000
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#define TOPE 1000
#define MAX(x,y) ((x)>(y)?(x):(y))

int main(void)
{
  int mx,i;

  clrscr();
  randomize();
  mx = random(TOPE);
  for (i=2; i<=10; i++)
  {
   int y;
   y = random(TOPE);
   mx = MAX(mx,y);
  }
  printf("El mayor número aleatorio generado: %d", mx);
  return 0;
}
```

En este otro ejemplo de generación de números aleatorios, se fija la semilla en 50 y se genera un número aleatorio.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(void)
{
  clrscr();
  srand(50);
  printf("Este es un número aleatorio: %d",rand());
  return 0;
}
```

7.11. FUNCIONES DE FECHA Y HORA

La familia de microprocesadores 80x86 tiene un sistema de reloj que se utiliza principalmente para controlar el microprocesador, pero se utiliza también para calcular la fecha y la hora.

El archivo de cabecera TIME.H define estructuras, macros y funciones para manipulación de fechas y horas. La fecha se guarda de acuerdo con el calendario gregoriano.

Las funciones **time,clock, _strdate** y **_strtime** devuelven la hora actual como el número de segundos transcurridos desde la medianoche del 1 de enero de 1970 (hora universal, GMT), el tiempo de CPU empleado por el proceso invocante, la fecha y hora actual, respectivamente.

La estructura de tiempo utilizada incluye los miembros siguientes:

```
struct tm
{
   int tm_sec;     /* segundos */
   int tm_min;     /* minutos */
   int tm_hour;    /* horas */
   int tm_mday;    /* día del mes 1 a 31 */
   int tm_mon;     /* mes, 0 para Ene, 1 para Feb,... */
   int tm_year;    /* año desde 1900 */
   int tm_wday;    /* días de la semana desde domingo (0-6)*/
   int tmjday;     /* día del año desde el 1 de Ene(0-365) */
   int tm_isdt;    /* siempre 0 para gmtime */
};
```

- **clock(void)**
  La función **clock** determina el tiempo de procesador, en unidades de click, transcurrido desde el principio de la ejecución del programa. Si no se puede devolver el tiempo de procesador se devuelve –1.

```
inicio = clock();
fin = clock();
```

- **time(hora)**
  La función **time** obtiene la hora actual; devuelve el número de segundos transcurridos desde la medianoche (00:00:00) del 1 de enero de 1970. Este valor de tiempo se almacena entonces en la posición apuntada por el argumento ***hora.*** Si ***hora*** es un puntero nulo, el valor no se almacena. El prototipo de la función es:

```
time-t time(time-t *hora);
```

  El tipo time_h está definido como tipo long en time.h.

- **localtime(hora)**
  Convierte la fecha y hora en una estructura de tipo tm. **Su** prototipo es

```
struct tm *localtime(const time-t *tptr);
```

- **mktime(t)**
  Convierte la fecha en formato de calendario. Toma la información del argumento y determina los valores del día de la semana (tm_wday) y del día respecto al inicio del año, también conocido como fecha juliana (tm_yday). Su prototipo es

```
time-t mktime(struct tm *tptr)
```

  La función devuelve –1 en caso de producirse un error.

  En este ejercicio se pide el año, mes y día; escribe el día de la semana y los días pasados desde el 1 de enero del año leído. Es utilizado un array de cadenas de caracteres, su estudio se hace en capítulos posteriores.

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

char *dias[] = { " ","Lunes", "Martes", "Miercoles",
                 "Jueves", "Viernes", "Cabado","Domingo"};

int main(void)
{
```

```
  struct tm fecha;
  int anyo, mes, dia;

  /*  Entrada: año, mes y dia */
  printf("Año: ");
  scanf("%d", &anyo);
  printf("Mes: ");
  scanf("%d", &mes);
  printf("Dia:  ");
  scanf("%d", &dia);

/*  Asigna fecha a la estructura fecha, en formato establecido */

  fecha.tm_year = anyo - 1900;
  fecha.tm_mon  = mes - 1;
  fecha.tm_mday = dia;
  fecha.tm_hour = 0;
  fecha.tm_min  = 0;
  fecha.tm_sec  = 1;
  fecha.tm_isdst = -1;

/*  mktime encuentra el día de la semana y el día del año.
    Devuelve -1 si error.
*/
  if (mktime(&fecha) == -1)
  {
    puts(" Error en la fecha.");
    exit(-1);
  }

/* El domingo, la función le considera dia 0 */
   if (fecha.tm_wday == 0)
      fecha.tm_wday = 7;

  printf("\nDia de la semana: %d; dia del año: %d",
                          fecha.tm_wday,fecha.tm_yday+1);

/*  Escribe el dia de la semana */
  printf("\nEs el dia de la semana, %s\n", dias[fecha.tm_wday]);
  return 0;
}
```

Ejercicio 7.4

Una aplicación de `clock()` para determinar el tiempo de proceso de un programa que calcula el factorial de un número.

El factorial de *n! = n*(n-1)*(n-2) ... 2*1*. La variable que vaya a calcular el factorial, se define de tipo `long` para poder contener un valor elevado. El número, arbitrariamente, va a estar comprendido entre *3* y 15. El tiempo de proceso va a incluir el tiempo de entrada de datos. La función `clock()` devuelve el tiempo en unidades de click, cada `CLK_TCK` es un segundo. El programa escribe el tiempo en ambas unidades.

```
/*
   En este ejercicio se determina el tiempo del procesador para
   calcular el factorial de un número requerido, entre 3 y 15.
*/

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

```
int main(void)
{
  float inicio, fin;
  int n, x;
  long int fact;

  inicio = clock();
  do {
    printf(" Factorial de (3 <x< 15): ");
    scanf("%d",&x);
  }while (x<=3 || x>=15);

  for (n=x,fact=1; x; x--)
     fact*=x;
  fin = clock();

  printf("\n Factorial de %d! = %ld",n,fact);
  printf("\n Unidades de tiempo de proceso: %f,\t En segundos: %f",
          (fin-inicio), (fin-inicio)/CLK_TCK);

  return 0;
}
```

7.12. FUNCIONES DE UTILIDAD

C incluyen una serie de funciones de utilidad que se encuentran en el archivo de cabecera `STDLIB.H` y que se listan a continuación.

- **`abs(n), labs(n)`**

  ```
  int abs(int n)
  long labs(long n)
  ```

  devuelven el valor absoluto de *n*.

- **`div(num, denom)`**

  ```
  div-t div(int num, int denom)
  ```

  Calcula el cociente y el resto de `num`, dividido por `denom` y almacena el resultado en `quot` y `rem`, miembros `int` de la estructura `div_t`.

  ```
  typedef struct
  {
    int quot;          /* cociente */
    int rem;           /* resto */
  } div-t;
  ```

  El siguiente ejemplo calcula y visualiza el cociente y el resto de la división de dos enteros.

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

  int main(void)
  {
    div-t resultado;

    resultado = div(16, 4);
    printf("Cociente %d", resultado.quot);
    printf("Resto %d", resultado.rem);
    return 0;
  }
  ```

- **ldiv(num,denom)**

  Calcula el cociente y resto de num dividido por denom, y almacena **los** resultados de quot y rem, miembros long de la estructura ldiv_t.

```
typedef struct
{
  long int quot;      /* cociente */
  long int rem;       /* resto    */
} ldiv_t;

resultado = ldiv(1600L, 40L);
```

7.13. VISIBILIDAD DE UNA FUNCIÓN

El *ámbito* de un elemento es su visibilidad desde otras partes del programa y la *duración* de un elemento es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuando se crea y cuando se hace disponible. El ámbito de un elemento en C depende de donde se sitúe la definición y de los modificadores que le acompañan. En resumen, se puede decir que un elemento definido dentro de una función tiene *ámbito local* (alcance local), o si se define fuera de cualquier función, se dice que tiene un *ámbito global.* La Figura *7.6* resume el modo en que se ve afectado el ámbito por la posición en el archivo fuente.

Existen dos tipos de clases de almacenamiento en C: auto y static. Una variable auto es aquella que tiene una *duración automática.* No existe cuando el programa comienza la ejecución, se crea en algún punto durante la ejecución y desaparece en algún punto antes de que el programa termine la ejecución. Una variable static es aquella que tiene una *duración fija.* **El** espacio para el elemento de programación se establece en tiempo de compilación; existe en tiempo de ejecución y se elimina sólo cuando el programa desaparece de memoria en tiempo de ejecución.

Las variables con ámbito global se denominan *variables globales* y son las definidas externamente a la función *(declaración externa).* Las variables globales tienen el siguiente comportamiento y atributos:

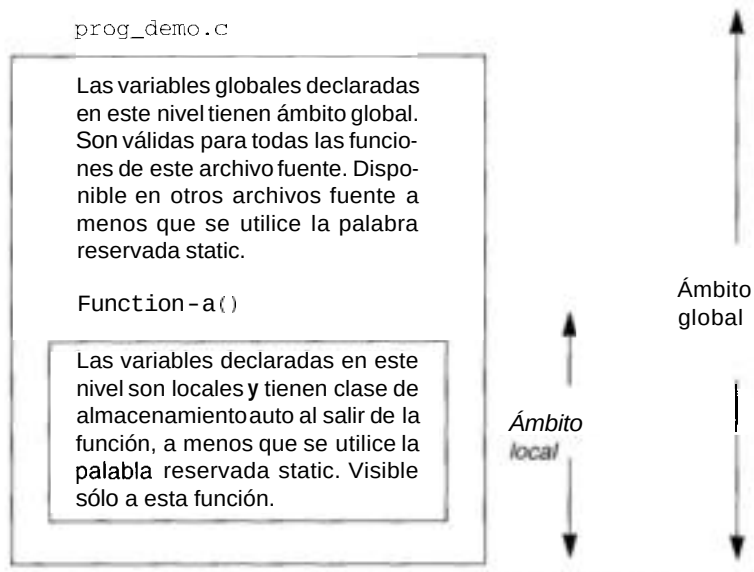


Figura *7.6.* Ámbito de variable local y global.

- ***Las variables globales tienen duración estática por defecto.*** El almacenamiento se realiza en tiempo de compilación y nunca desaparece. Por definición, una variable global no puede ser una variable auto.
- ***Las variables globales son visibles globalmente en el archivo fuente.*** Se pueden referenciar por cualquier función, a continuación del punto de definición.
- ***Las variables globules están disponibles, por defecto, a otros archivos fuente.*** Esta operación se denomina ***enlace externo.***

7.13.1. Variables locales frente a variables globales

Además de las variables globales, es preciso considerar las variables locales. Una ***variable local*** está definida solamente dentro del bloque o cuerpo de la función y no tiene significado ***(vida)*** fuera de la función respectiva. Por consiguiente, si una función define una variable como local, el ámbito de la variable está protegido. La variable no se puede utilizar, cambiar o borrar desde cualquier otra función sin una programación específica mediante el paso de valores (parámetros).

Una *variable local* es una variable que se define dentro de una función.

Una *variable global* es una variable que puede ser utilizada por todas las funciones de un programa dado, incluyendo `main()`.

Para construir variables globales en C, se deben definir fuera de la función **`main()`.** Para ilustrar el uso de variables locales y globales, examine la estructura de bloques de la Figura 7.7. Aquí la variable global es `x0` y la variable local es `x1`. La función puede realizar operaciones sobre `x0` y `x1`. Sin embargo, **`main()`** sólo puede operar con `x0`, ya que `x1` no está definida fuera del bloque de la función **`funcion1()`**. Cualquier intento de utilizar `x1` fuera de **`funcion1()`** producirá un error.

```
int  x0 ;                    /* variable global */
funcion1 (...)               /* prototipo funcional */

int  main
{
   ...
   ...
   ...
}

   funcioril (...)
   {
      int xl                 /* variable local */
      ...


   }
```

Figura 7.7. `x0` es global al programa completo, mientras que `x1` es local a la función `funcion1()`.

Exainine ahora la Figura 7.8. Esta vez existen dos funciones, ambas definen x1 como variable local. Nuevamente x0 es una variable global. La variable x1 sólo se puede utilizar dentro de las dos funciones. Sin embargo, cualquier operación sobre x1 dentro de **funcioni()** no afecta al valor de x1 en **funcion2()** y viceversa. En otras palabras, la variable x1 de **funcioni()** se considera una variable independiente de x1 en **funciona().**

Al contrario que las variables, *las funciones son externas por defecto.* Es preciso considerar la diferencia entre *definición* de una función y *declaración.* Si una declaración de variable comienza con la palabra reservada **extern,** no se considera definición de variable. Sin esta palabra reservada es una definición. Cada definición de variable es al mismo tiempo una declaración de variable. Se puede utilizar una variable sólo después de que ha sido declarada (en el mismo archivo). Únicamente las definiciones de variables asignan memoria y pueden, por consiguiente, contener inicializaciones. Una *variable sólo se define una vez,* pero se puede *declarar* tantas veces como se desee. Una declaración de variable al nivel global (externa a las funciones) es válida desde esa declaración hasta el final del archivo; una declaración en el interior de una función es válida sólo en esa función. En este punto, considérese que las definiciones y declaraciones de variables globales son similares a las funciones; la diferencia principal es que se puede escribir la palabra reservada **extern** en declaraciones de función.

Figura 7.8. x0 es global al programa completo, x1 es local tanto funcion1() como a funcion2(), pero se tratan como variables independientes.

La palabra reservada **extern** se puede utilizar para notificar al compilador que la declaración del resto de la línea no está definida en el archivo fuente actual, pero está localizada en otra parte, en otro archivo. El siguiente ejemplo utiliza **extern:**

```
/* archivo con la funcion main(): programa.c */
int total;
extern int suma;
extern void f(void);
void main(void)

/*
  archivo con la definición de funciones y variable: modulo.c
*/
int suma;
void f(void)
...
```

Utilizando la palabra reservada **extern** se puede acceder a símbolos externos definidos en otros módulos. `suma` y la función `f()` se declaran externas.

Las **funciones son externas por defecto,** al **contrario que las variables.**

7.13.2. Variables estáticas y automáticas

Los valores asignados a las variables locales de una función se destruyen cuando se termina la ejecución de la función y no se puede recuperar su valor para ejecuciones posteriores de la función. Las variables locales se denominan *variables automáticas,* significando que se pierden cuando termina la función. Se puede utilizar `auto` para declarar una variable

```
auto int ventas;
```

aunque las variables locales se declaran automáticas por defecto y, por consiguiente, el uso de `auto` es opcional y, de hecho, no se utiliza.

Las *variables estáticas* (`static`), por otra parte, mantienen su valor después que una función se ha terminado. Una variable de una función, declarada como estática, mantiene un valor a través de ejecuciones posteriores de la misma función. Haciendo una variable local estática, su valor se retiene de una llamada a la siguiente de la función en que está definida. Se declaran las variables estáticas situando la palabra reservada `static` delante de la variable. Por ejemplo,

```
static int ventas = 10000;
static int dias = 500;
```

Este valor se almacena en la variable estática, sólo la primera vez que se ejecuta la función. Si su valor no está definido, el compilador almacena un cero en una variable estática por defecto.

El siguiente programa ilustra el concepto estático de una variable:

```
#include <stdio.h>
/* prototipo de la función   */
void Ejemplo_estatica(int);

void main()
{
  Ejemplo_estatica(1);
  Ejemplo_estatica(2);
  Ejemplo_estatica(3);
}

/* Ejemplo del uso de una variable estática  */
```

```
void Ejemplo_estatica(int Llamada)
{
  static int Cuenta;
  if (Llamada == 1)
     Cuenta = 1;
  printf("\n El valor de Cuenta en llamada nº %d es: %d",
             Llamada,Cuenta);
  ++Cuenta;
}
```

Al ejecutar el programa se visualiza:

```
El valor de Cuenta en llamada nº 1 es: 1
El valor de Cuenta en llamada nº 2 es: 2
El valor de Cuenta en llamada nº 3 es: 3
```

Si quita la palabra reservada `static` de la declaración de `Cuenta`, el resultado será:

```
El valor de Cuenta en llamada nº 1 es: 1
El valor de Cuenta en llamada nº 2 es: 1046
```

no se puede predecir cuál es el valor de `Cuenta` en llamadas posteriores **a** la primera.

Las variables globales se pueden ocultar de otros archivos fuente utilizando el especificador de almacenamiento de clase static.

Para hacer una variable global privada al archivo fuente **(y,** por consiguiente, no Útil **a** otros módulos de código) se le hace preceder por la palabra `static`. Por ejemplo, las siguientes variables se declaran fuera de las funciones de un archivo fuente:

```
static int m = 25;
static char linea_texto[80];
static int indice-linea;
static char bufer[MAXLOGBUF];
static char *pBuffer;
```

Las variables anteriores son privadas al archivo fuente. Observe este ejemplo:

```
#define OFF 0
#define ON 1
...
static unsigned char maestro = OFF;
...

main()

...
}

funcion-a()
I

...
}
```

`maestro` se puede utilizar tanto en **`funcion-a()`** como en **`main()`,** en este archivo fuente, pero no se puede declarar como **`extern`** a otro archivo fuente.

Se puede hacer también una declaración de función `static`. Por defecto, todas las funciones tienen enlace externo **y** son visibles a otros módulos de programa. Cuando se sitúa la palabra reservada `static` delante de la declaración de la función, el compilador hace privada la función al archivo fuente. Se puede, entonces, reutilizar el nombre de la función en otros módulos fuente del programa.

7.14. COMPILACIÓN SEPARADA

Hasta este momento, casi todos los ejemplos que se han expuesto en el capítulo se encontraban en un sólo archivo fuente. Los programas grandes son más fáciles de gestionar si se dividen en varios archivos fuente, también llamados *módulos,* cada uno de los cuales puede contener una o más funciones. Estos módulos se compilan y enlazan por separado posteriormente con un *enlazador*, o bien con la herramienta correspondiente del entorno de programación. Cuando se divide un programa grande en pequeños, los únicos archivos que se recompilan son los que se han modificado. El tiempo de compilación se reduce, dado que pequeños archivos fuente se compilan más rápido que los grandes. Los archivos grandes son difíciles de mantener y editar, ya que su impresión es un proceso lento que utilizará cantidades excesivas de papel.

La Figura 7.9 muestra cómo el enlazador puede construir un programa ejecutable, utilizando módulos objetos, cada uno de los cuales se obtiene compilando un módulo fuente.

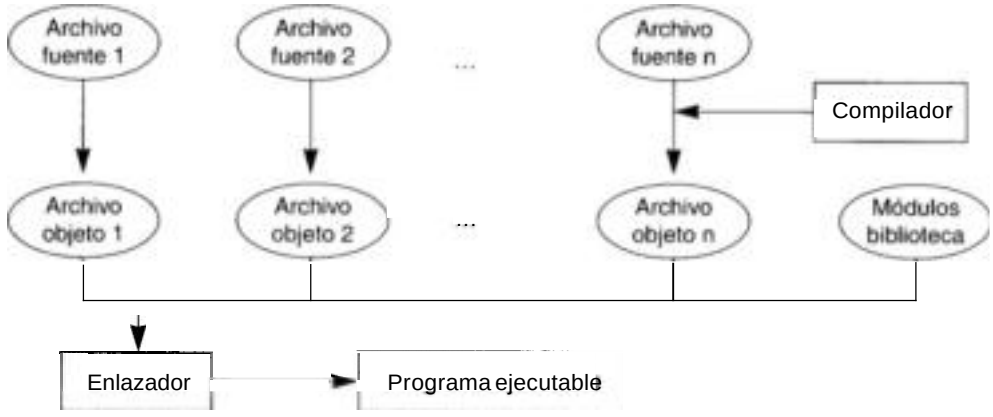


Figura 7.9. Compilación separada.

Cuando se tiene más de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por defecto. Si desea, por razones de legibilidad —no recomendable—, puede utilizar la palabra reservada `extern` con un prototipo de función y en la cabecera.

Se puede desear restringir la visibilidad de una función, haciéndola visible sólo a otras funciones en un archivo fuente. Una razón para hacer esto es tener la posibilidad de tener dos funciones con el mismo nombre en diferentes archivos. Otra razón es reducir el número de referencias externas y aumentar la velocidad del proceso de enlace.

Se puede hacer una función no visible al exterior de un archivo fuente utilizando la palabra reservada `static` con la cabecera de la función y la sentencia del prototipo de función. Se escribe la palabra `static` antes del tipo de valor devuelto por la función. Tales funciones no serán públicas al enlazador, de modo que otros módulos no tendrán acceso a ellas. La palabra reservada **`static`**, tanto para variables globales como para funciones, es Útil para evitar conflictos de nombres y prevenir el uso accidental de ellos. Por ejemplo, imaginemos un programa muy grande que consta de muchos módulos, en el que se busca un error producido por una variable global; si la variable es estática, se puede restringir su búsqueda al módulo en que está definida; si no es así, se extiende nuestra investigación a los restantes módulos en que está declarada (con la palabra reservada `extern`).

Como regla general, son preferibles las variables locales a las globales. Si realmente es necesario o deseable que alguna variable sea **global,** es preferible hacerla estática, lo que significa que será «local» en relación al archivo en que está definida.

Ejemplo 7.6

Supongamos dos módulos: `MODULO1` *y* `MODULO2`*. En el primero se escribe la función main(), hace referencia a funciones y variables globales definidas en el segundo módulo.*

```
/* MODULO1.C */
#include <stdio.h>

void main()
{
  void f(int i), g(void);
  extern int n;              /* Declaración de n (no definición) */
  f(8);
  n++;
  g();
  puts("Fin de programa.";
}

/* MODUL02.C */

#include <stdio.h>
int n = 100;                 /* Definición de n (también declaración) */

static int m = 7;

void f(int i)
{
  n += (i+m);
}

void g(void)
{
  printf("n = %d\n",n);
}
```

`f` y *g* **se** *definen* en el módulo 2 y ***se*** *declaran* en el módulo 1. Si se ejecuta el programa, se produce la salida

```
n = 116
Fin de programa.
```

Se puede hacer una función invisible fuera de un archivo fuente utilizando la palabra reservada `static` con la cabecera y el prototipo de la función.

7.15. VARIABLES REGISTRO (`register`)

Una *variable registro* (`register`) es similar a una variable local, pero en lugar de ser almacenada en la pila, se almacena directamente en un registro del procesador (tal como *ay* o *bx*). Dado que el número

de registros es limitado y además están limitados en tamaño, el número de variables registro que un programa puede crear simultáneamente es muy restringido.

Para declarar una variable registro, se hace preceder a la misma con la palabra reservada `register` ;

```
register int k;
```

La ventaja de las variables registro es su mayor rapidez de manipulación. Esto se debe a que las operaciones sobre valores situados en los registros son normalmente más rápidas que cuando se realizan sobre valores almacenados en memoria. Su uso se suele restringir a segmentos de código mucha veces ejecutados. Las variables registro pueden ayudar a optimizar el rendimiento de un programa proporcionando acceso directo de la CPU a los valores claves del programa.

Una variable registro debe ser local a una función; nunca puede ser global al programa completo. El uso de la palabra reservada `register` no garantiza que un valor sea almacenado en un registro. Esto sólo sucederá si un registro está disponible (libre). Si no existen registros disponibles, C crea la variable como si fuera una variable local normal.

Una aplicación usual de las variables registro es como variable de control de bucles `for` o en la expresión condicional de una sentencia `while`, que se deben ejecutar a alta velocidad.

```
void usoregistro(void)
{
  register int k;
  puts("\n Contar con una variable registro.');
  for (k = 1; k <= 100; k++)
    printf("%8d",k);
}
```

7.16. RECURSIVIDAD

Una *función recursiva* es una función que se llama a **sí** misma directa o indirectamente. La *recursividad* o *recursión directa* es el proceso por el que una función se llama a **sí** misma desde el propio cuerpo de la función. La *recursividad* o ***recursión** indirecta* implica más de una función.

La recursividad indirecta implica, por ejemplo, la existencia de dos funciones: **`uno()`** y **`dos()`**. Suponga que **`main()`** llama a **`uno()`**, y a continuación **`uno()`** llama a **`dos()`**. En alguna parte del proceso, **`dos()`** llama a **`uno()`** —una segunda llamada a **`uno()`**—. Esta acción es recursión indirecta, pero es recursiva, ya que **`uno()`** ha sido llamada dos veces, sin retornar nunca a su llamadora.

Un proceso recursivo debe tener una condición de terminación, ya que si no puede continuar indefinidamente.

Un algoritmo típico que conduce a una implementación recursiva es el cálculo del factorial de un número. El factorial de *n* (`n!`).

```
n! = n * (n-1) * (n-2)* ... * 3 * 2 * I
```

En consecuencia, el factorial de 4 es igual a 4*3*2*1, el factorial de 3 es igual a 3*2*1. **Así** pues, el factorial de 4 es igual a 4 veces el factorial de 3. La Figura 7.10 muestra la secuencia de sucesivas invocaciones a la función factorial.

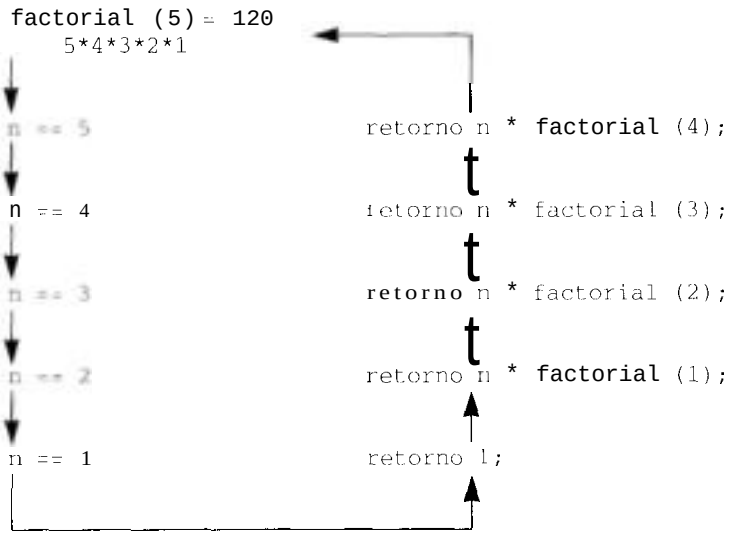


Figura 7.10. Llamadas a funciones recursivas para `factorial(5)`.

Ejemplo 7.7

Realizar el algoritmo de la función factorial.

La implementación de la función recursiva `factorial` es:

```
double factorial(int numero)
{
  if (numero > 1)
    return numero * factorial(numero-1);
  return 1;
}
```

Ejemplo 7.8

Contar valores de 1 a 10 de modo recursivo.

```
#include <stdio.h>
void contar(int cima);
int main()
{
  contar(10);
  return 0;
}
void contar(int cima)
{
  if (cima > 1)
    contar(cima-1);
  printf("%d ",cima);
}
```

Ejemplo 7.9

Determinar si un número entero positivo es par o impar; con dos funciones que se llaman mutuamente: recursividad indirecta.

```
#include <stdio.h>

int par(int n);
int impar(int n);

int main(void)
{
   int n;

/*  Entrada: entero > 0 */
   do {
     printf("\nEntero > 0:  ");
     scanf("%d", &n);
   } while (n<=0);

/*  Llamda a la funci¢n par() */
   if (par(n))
     printf("El numero %d es par.",n);
   else
     printf("El numero %d es impar.",n);
   return 0;
}

int par(int n)
{
  if (n == 0)
    return 1;     /* es par */
  else
    return impar(n-1);
}

int impar(int n)
{
  if (n == 0)
    return 0;     /* es impar */
  else
    return par(n-1);
}
```

La función `par()` llama a la función `impar()`, ésta a su vez llama a la función `par()`. La condición para terminar de hacer llamadas es que `n` sea cero; el cero se considera par.

7.17. RESUMEN

Las funciones son la base de la construcción de programas en C. Se utilizan funciones para subdividir problemas grandes en tareas más pequeñas. El encapsuiamiento de las características en funciones, hace los programas más fáciles de mantener. El uso de funciones ayuda al programador a reducir el tamaño de su programa, ya que se puede llamar repetidamente y reutilizar el código dentro de una función.

En este capítulo habrá aprendido lo siguiente:

- el concepto, declaración, definición y uso de una función;
- las funciones que devuelven un resultado lo hacen a través de la sentencia `return;`
- los parámetros de funciones se pasan por valor, para un paso por referencia se utilizan punteros;
- el modificador `const` se utiliza cuando se desea que los parámetros de la función sean valores de sólo lectura;
- el concepto y uso de prototipos, cuyo uso es recomendable en C;
- la ventaja de utilizar macros con argumentos, para aumentar la velocidad de ejecución;
- el concepto de *ámbito* o *alcance* y *visibilidad,* junto con el de *variable global* y *local*;
- clases de almacenamiento de variables en memoria: `auto, extern, register` y `static.`

La biblioteca estándar C de funciones en tiempo de ejecución incluye gran cantidad de funciones. Se agrupan por categorías, entre las que destacan:

- manipulación de caracteres;
- numéricas;
- tiempo y hora;
- conversión de datos;
- búsqueda y ordenación;
- etc.

Tenga cuidado de incluir el archivo de cabecera correspondiente cuando desee incluir funciones de biblioteca en sus programas.

Una de las características más sobresalientes de C que aumentan considerablemente la potencia de los programas es la posibilidad de manejar las funciones de modo eficiente, apoyándose en la propiedad que les permite ser compiladas por separado.

Otros temas tratados han sido:

- *Ámbito* o las *reglas de visibilidad* de funciones y variables.
- En entorno de un programa tiene cuatro tipos de ámbito: de programa, archivo fuente, función y bloque. Una variable está asociada a uno de esos ámbitos y es invisible (no accesible) desde otros ámbitos.
- Las *variables globules* se declaran fuera de cualquier función y son visibles a todas las funciones. Las variables locales se declaran dentro de una función y sólo pueden ser utilizadas por esa función.

```
int i;          /* variable global,
                ámbito de programa
                */
static int j    /* ámbito de archivo
                */

main()
{
  int d, e;     /* variable local,
                ámbito de función */
  ...
}

func(int j)
{
  if (j > 3)
  {
   int i;       /* ámbito de bloque */
   for (i = 0; i < 20; i++)
      func2(i);
  }
  /* i ya no es visible */
  }
```

- *Variables automáticas* son las variables, por defecto, declaradas localmente en una función.
- *Variables estáticas* mantienen su información, incluso después que la función ha terminado.

Cuando se llama de nuevo la función, la variable se pone al valor que tenía cuando se llamó anteriormente.

- *Funciones recursivas* son aquellas que se pueden llamar a sí mismas.
- Las *variables registro* se pueden utilizar cuando se desea aumentar la velocidad de procesamiento de ciertas variables.

7.18. EJERCICIOS

7.1. Escribir una función que tenga un argumento de tipo entero y que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.

7.2. Escribir una función lógica de dos argumentos enteros, que devuelva *true* si uno divide al otro y *false* en caso contrario.

7.3. Escribir una función que convierta una temperatura dada en grados Celsius a grados Fahrenheit. La fórmula de conversión es:

$$F = \frac{9}{5}C + 32$$

7.4. Escribir una función lógica *Dígito* que determine si un carácter es uno de los dígitos de 0 a 9.

7.5. Escribir una función lógica *Vocal* que determine si un carácter es una vocal.

7.6. Escribir una función Redondeo que acepte un valor real Cantidad y un valor entero Decimales y devuelva el valor Cantidad redondeado al número especificado de Decimales. Por ejemplo, Redondeo (20.563,2) devuelve 20.56.

7.7. Determinar y visualizar el número más grande de dos números dados, mediante un subprograma.

7.8. Escribir un programa recursivo que calcule los *N* primeros números naturales.

7.19. PROBLEMAS

7.1. Escribir un programa que solicite del usuario un carácter y que sitúe ese carácter en el centro de la pantalla. El usuario debe poder a continuación desplazar el carácter pulsando las letras A (arriba), B (abajo), I (izquierda), D (derecha) y F (fin) para terminar.

7.2. Escribir una función que reciba una cadena de caracteres y la devuelva en forma inversa (ñola' se convierte en ´aloh´).

7.3. Escribir una función que determine si una cadena de caracteres es un palíndromo (un palíndrorno es un texto que se lee igual en sentido directo y en inverso: radar).

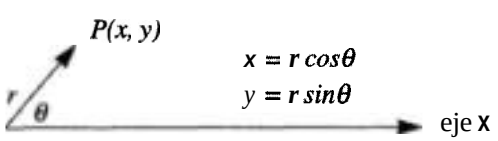
7.4. Escribir un programa mediante una función que acepte un número de día, mes y año y lo visualice en el formato

```
dd/mm/aa
```

Por ejemplo, los valores *8,* 10 y **1946** se visualizan como

```
8/10/46
```

7.5. Escribir un programa que utilice una función para convertir coordenadas polares a rectangulares.



7.6. Escribir un programa que lea un entero positivo y a continuación llame a una función que visualice sus factores primos.

7.7. Escribir un programa, mediante funciones, que visualice un calendario de la forma:

L	M	M	J	V	S	D
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	23
27	28	29	30			

El usuario indica únicamente el mes y el año.

7.8. Escribir un programa que lea los dos enteros positivos *n* y *b* que llame a una función *CambiarBase* para calcular y visualizar la representación del número *n* en la base *b*.

7.9. Escribir un programa que permita el cálculo del mcd (máximo común divisor) de dos números por el algoritmo de Euclides. (Dividir *a* entre *b*, se obtiene el cociente *q* y el resto *r* si es cero *b* es el mcd, si no se divide *b* entre *r*, y así sucesivamente hasta encontrar un resto cero, el último divisor es el mcd.) La función mcd() devolverá el máximo común divisor.

7.10. Escribir una función que devuelva el inverso de un número dado (1234, inverso 4321).

7.11. Calcular el coeficiente del binomio con una función factorial.

$$\binom{m}{n} = \frac{m!}{n!\,(m-n)!} \qquad \text{donde} \qquad m! = \begin{cases} 1 & si\ m = 0 \\ 1,2,3\ldots m & si\ m \neq 0 \end{cases}$$

7.12. Escribir una función que permita calcular la serie:

$$1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n \,.\, (n+10) \,.\, (2n+1)}{6} = n * (n+10) * (2 * n+1) / 6$$

7.13. Escribir un programa que lea dos números *x* y *n* y en una función calcule la suma de la progresión geométrica.

$$1 + x + x^2 + x^3 + \ldots + x^n$$

7.14. Escribir un programa que encuentre el valor mayor, el valor menor y la suma de los datos de entrada. Obtener la media de los datos mediante una función.

7.15. Escribir una función que acepte un parámetro x($x \neq 0$) y devuelva el siguiente valor:

$$\frac{1}{x^3\left(\frac{e^{1.435}}{x} - 1\right)}$$

7.16. Escribir una función con dos parámetros, *x* y *n*, que devuelva lo siguiente:

$$x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2} \qquad \text{si} \quad x >= 0$$

$$x + \frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} \qquad \text{si} \quad x <= 0$$

7.17. Escribir una función que tome como parámetros las longitudes de los tres lados de un triángulo (*a*, *b* y *c*) y devuelva el área del triángulo.

$$Área = \sqrt{p\,(p-a)\,(p-b)\,(p-c)}$$

$$\text{donde} \quad p = \frac{a+b+c}{2}$$

7.18. Escribir un programa mediante funciones que realicen las siguientes tareas:

a) Devolver el valor del día de la semana en respuesta a la entrada de la letra inicial (mayúscula o minúscula) de dicho día.

b) Determinar el número de días de un mes.

7.19. Escribir un programa que lea una cadena de hasta diez caracteres que representa a un número en numeración romana e imprima el formato del número romano y su equivalente en numeración arábiga. Los caracteres romanos y sus equivalentes son:

M	1000	L	50
D	500	X	10
C	100	V	5
		I	1

Compruebe su programa para los siguientes datos:

LXXXVI (86), CCCXIX (319), MCCLIV (1254).

7.20. Escriba una función que calcule cuántos puntos de coordenadas enteras existen dentro de un triángulo del que se conocen las coordenadas de sus tres vértices.

7.21. Escribir un programa que mediante funciones determine el área del círculo correspondiente a la circunferencia circunscrita de un triángulo del que conocemos las coordenadas de los vértices.

7.22. Dado el valor de un ángulo escribir una función que muestra el valor de todas las funciones trigonométricas correspondientes al mismo.