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

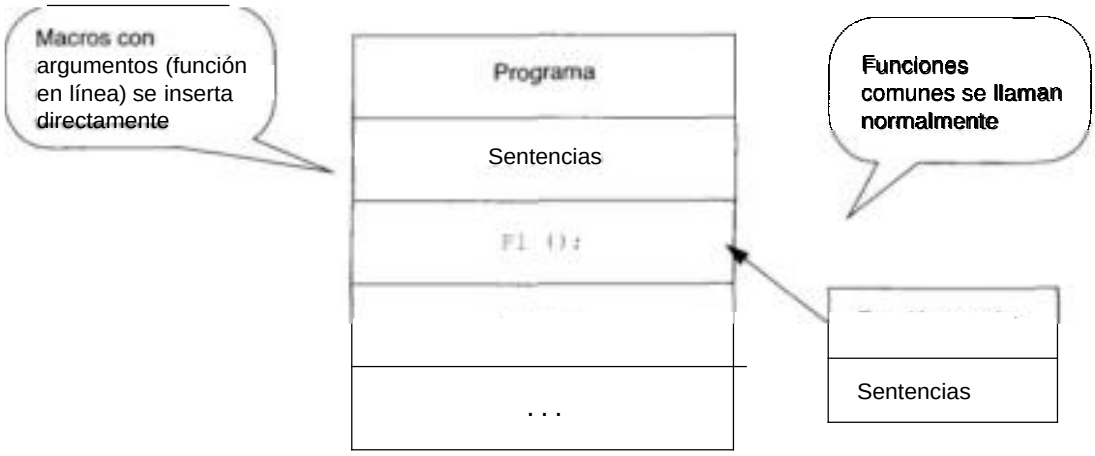

Figura 7.4. Código generado por una función fuera de línea.

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
```