Capítulo 3

Administración de la memoria en C.

Los datos se almacenan en uno de los tres **segmentos** de memoria que el programador dispone.

La zona estática para datos, que permite almacenar variables globales durante la ejecución de un programa.

El stack que permite almacenar los argumentos y variables locales durante la ejecución de las funciones.

El heap que permite almacenar variables adquiridas dinámicamente durante la ejecución de un programa.

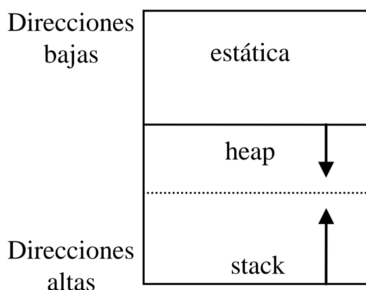

Figura 3.1. Segmentos de memoria.

Los segmentos son provistos por el sistema operativo.

3.1. Manejo estático de la memoria.

La zona estática para datos, permite almacenar variables globales y estáticas.

Si se encuentra una variable definida fuera de las funciones, se la considera global; el compilador le asigna un espacio determinado y genera la referencia para accesarla en la zona estática. El tamaño de las variables no puede cambiarse durante la ejecución del programa, es asignado en forma estática.

El tiempo de vida de las variables de la zona estática es la duración del programa.

Estas variables son visibles para todas las funciones que estén definidas después de ellas.

Si se precede con la palabra **static** a una variable local a una función, ésta también es ubicada en la zona estática, y existe durante la ejecución del programa; no desaparece al terminar la ejecución de la función, y conserva su valor, entre llamados a la función.

3.2. Manejo automático de la memoria en C.

3.2.1. Asignación, Referencias y tiempo de vida.

El compilador asigna un espacio determinado para las variables y genera las referencias para accesar a las variables del stack y de la zona estática. El tamaño de las variables de estas zonas no puede cambiarse durante la ejecución del programa, es asignado en forma estática.

El tiempo de vida de las variables de la zona estática es la duración del programa; las variables denominadas automáticas, o en la zona del stack, existen durante la ejecución de la función que las referencia.

Los argumentos y variables locales, son asignados y desasignados en forma dinámica durante la ejecución de las funciones; pero en forma automática, el programador no tiene responsabilidad en ese proceso.

Esta organización permite direccionar eficientemente variables que serán usadas frecuentemente; a la vez posibilita ahorrar espacio de direccionamiento ya que se puede reutilizar el espacio de memoria dedicado a la función cuando ésta termina; y también posibilita el diseño de funciones recursivas y reentrantes, asociando un espacio diferente para las variables por cada invocación de la función.

Es importante notar que varias funciones pueden emplear los mismos nombres para las variables locales y argumentos y esto no provoca confusión; existe independencia temporal de las variables de una función. Si se emplea una global, con igual nombre que una local, dentro de la función se ve la local; y fuera existe la global.

Por la misma razón, no pueden comunicarse los valores de variables locales de una función a otra.

Cuando un programa se carga en la memoria, desde el disco, sólo se traen la zona de códigos y los datos de la zona estática. Las zonas de stack y heap, son creadas en memoria antes de la ejecución del programa.

3.2.2. Argumentos y variables locales.

Cada función al ser invocada crea un **frame** o registro de activación en el stack, en el cual se almacenan los valores de los argumentos y de las variables locales. Los valores de los argumentos son escritos en memoria, antes de dar inicio al código asociado a la función. Es responsabilidad de la función escribir valores iniciales a las variables locales, antes de que éstas sean utilizadas; es decir, que aparezcan en expresiones en donde se leen estas variables.

Ejemplo 3.1. Función con dos argumentos de tipo valor, con dos locales, retorno de entero.

La siguiente definición de función, tiene argumentos, variables locales y un retorno de tipo entero.

```
int función1(int arg1, int arg2)
{
   int local1;
   int local2=5;
   /* no puede usarse local1 para lectura, por ejemplo: local2=local1+2; es un error  */

   local1=arg1 + arg2 + local2;
   return ( local1+ 3);
}
```

A continuación un ejemplo de uso de la función.
Si una variable x, se define fuera de las funciones, se la considera global y se le asigna espacio en la zona estática. Si en el texto escrito, su definición aparece antes de la función1, se dice que es visible por ésta, o que está dentro del alcance de la función.

```
int x=7;
/*En este punto aún no existen las variables: local1, local2, arg1 y arg2.  */

x = función1(4, 8);
/*Desde aquí en adelante no existe espacio asociado a los argumentos y variables locales de la función 1 */
```

El diagrama de la Figura 3.1, ilustra el espacio de memoria asignado a las variables, después de invocada la función y justo después de la definición de la variables local2. La variable local1, no está iniciada y puede contener cualquier valor.

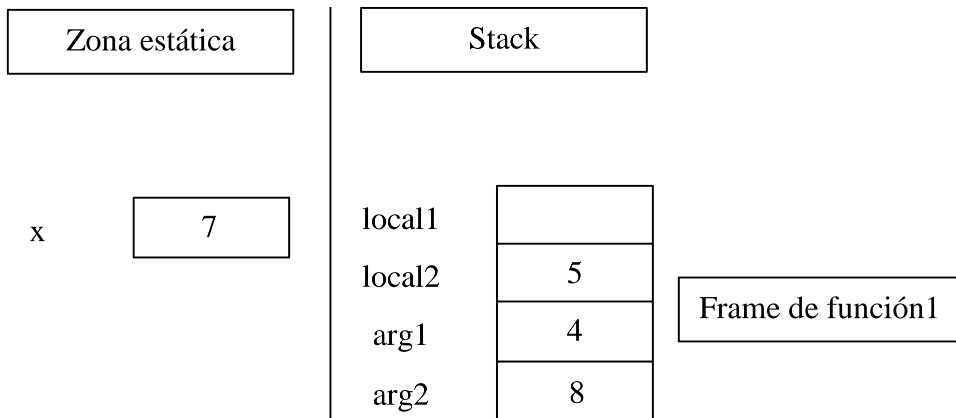

Figura 3.1a. Stack después de invocar a la función

Al salir de la función, el espacio de memoria asignado a las variables, puede visualizarse según:

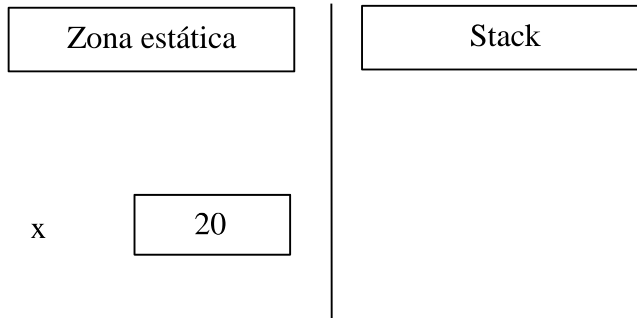

Figura 3.2. Stack al salir de la función.

3.2.3. Visión lógica del stack.

Los diagramas anteriores describen lógicamente el espacio asignado en la memoria para las variables estáticas y automáticas. Cada compilador implementa físicamente la creación de estos espacios de manera diferente; en el frame de la función se suelen almacenar: la dirección de retorno, los valores de los registros que la función no debe alterar; además del espacio para locales y argumentos. Adicionalmente cada compilador establece convenios para pasar los valores y obtener el retorno de la función, en algunos casos lo hace a través de registros, en otras a través del stack. El uso detallado del stack según lo requiere un programador assembler es cubierto en cursos de estructuras de computadores.

Se describe aquí una *visualización lógica* del segmento del stack, que es la que requiere un programador en lenguajes de alto nivel.

Ejemplo 3.2. Riesgos de la desaparición del frame.

La siguiente función plocal retorna un puntero a una variable local, lo cual es un gran error, ya que al salir de la función, deja de existir la local; y el puntero retornado apunta a una dirección del stack que no está asignada.

```
int* plocal(void)
{
    int local;
    // ****
    return(&local); // retorna puntero a local
}
```

La función que invoca a plocal posiblemente produzca una falla seria del programa, o generará un error de difícil depuración.

Tampoco puede tomarse la dirección de una variable local, declarada de tipo registro. Ya que los registros no tienen asociada una dirección de la memoria. El calificar una variable local o a un argumento de tipo **register**, es una indicación para que el compilador intente emplear registros en su manipulación, con ventajas temporales de acceso.

3.2.4. Copia de argumentos.

Puede llamarse a una función con los valores de los argumentos o variables locales (o más generalmente mediante una expresión de éstas) de otra función. Sin embargo la función que es llamada, crea una copia de los valores de sus argumentos, en su propio frame. Lo cual garantiza la independencia de funcionamiento de las funciones, ya que una función que es invocada por otra, puede cambiar sus argumentos sin alterar los argumentos de la que la invocó.

A su vez esta organización implica crear la copia, lo cual puede ser muy costoso en tiempo si el tipo del argumento tiene un gran tamaño en bytes. Para solucionar el problema de la copia, se puede pasar una referencia a un argumento de grandes dimensiones, esto se logra pasando el valor de un puntero (se copia el valor del puntero, el cual ocupa normalmente el tamaño de un entero). Esto se verá en pasos de argumentos por referencia.

Ejemplo 3.3. Una función f que invoca a otra función g.

Se tienen las siguientes definiciones de las funciones. Nótese que los nombres de los argumentos y una de las locales tienen iguales nombres.

```
int g(int a, int b)
{   int c;
    printf("Al entrar en g:  a = %d b = %d  \n", a, b);
    a = a + b;  /*cambia argumento a */
    c = a + 4;
    printf("Antes de salir de g:  a = %d b = %d  c = %d \n", a, b, c);
    return( c );
}

int f(int a, int b)
{   int c;
    int d=5;
    c = a+b+d;
    printf("Antes  de  g: a = %d b = %d  c = %d d = %d \n", a, b, c, d);
    d = g( a, b+c );   /*se copian los valores en el frame de g  */
    a = a + d;
    b = b + a;
    printf("Después de g: a = %d b = %d  c = %d d = %d \n", a, b, c, d);
    return( d + 2);
}
```

Si consideramos x definida en la zona estática, el siguiente segmento:

```
x=3;
x=f(5,6);
printf(" x = %d \n", x);
```

Genera la siguiente salida.

```
Antes  de  g: a = 5 b = 6  c = 16 d = 5
Al entrar en g:  a = 5 b = 22
```

Antes de salir de g: a = 27 b = 22 c = 31
Después de g: a = 36 b = 42 c = 16 d = 31
x = 33

Se ilustran los frames, con el estado de las variables, después de los printf.

Antes de invocar a g, se tiene el esquema de la Figura 3.3:

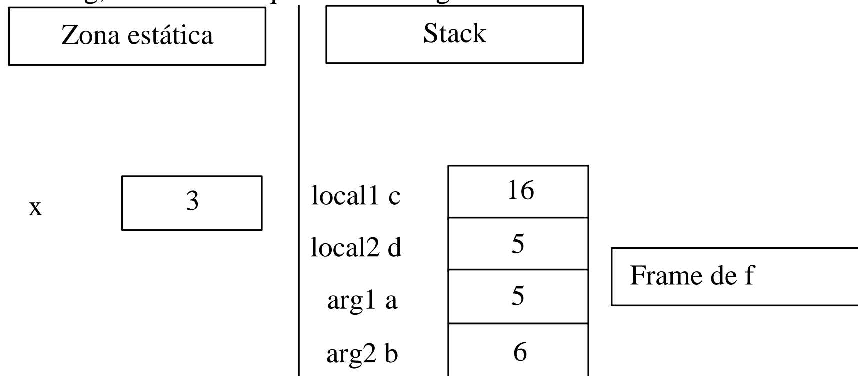

Figura 3.3. Stack antes de invocar a la función g.

Al entrar en g, se copian valores en los argumentos. Los frames se apilan hacia arriba, lo cual se muestra en la Figura 3.4.

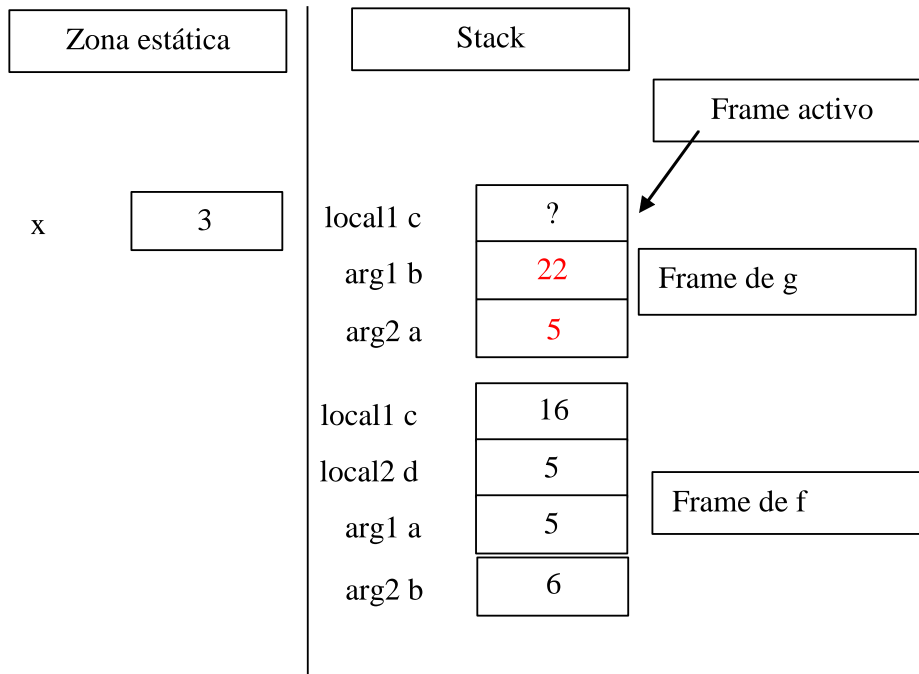

Figura 3.4. Al entrar en la función g.

Antes de salir de la función g:

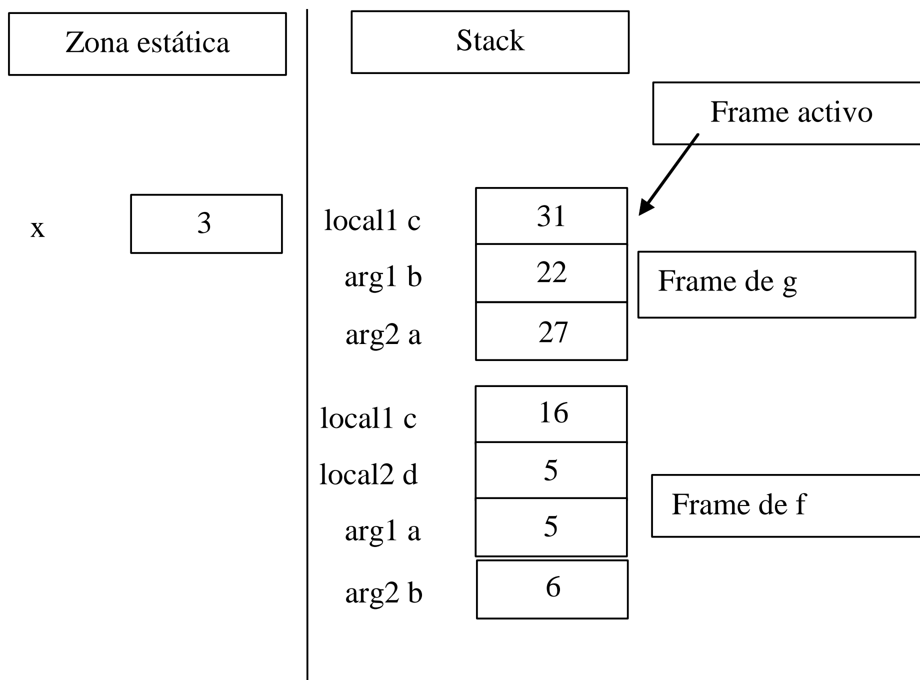

Figura 3.5. Stack justo antes de salir de la función g.

Al salir de la función g, ya no está disponible el frame de g, y el frame activo es el de la función f:

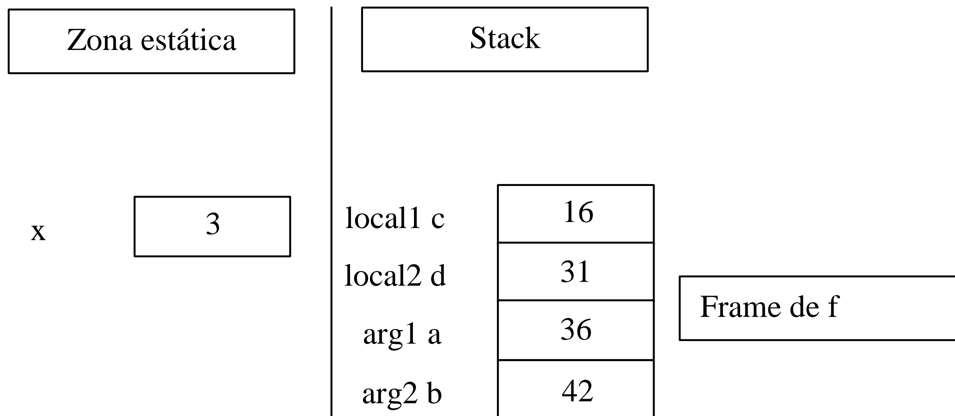

Figura 3.6. Stack al salir de la función g.

Al salir de la función f, se desvanece su frame, como se ilustra en la Figura 3.7:

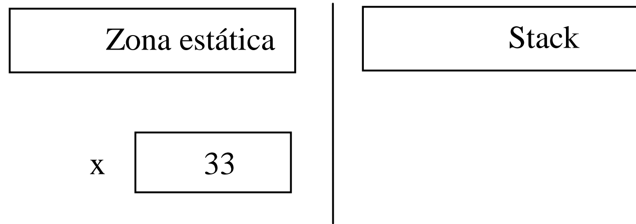

Figura 3.7. Stack al salir de la función f.

Ejemplo 3.4. La misma función anterior invocada dos veces.

Veamos un caso en el cual se ejecuta dos veces la misma función1, del Ejemplo 3.1.

```
int x=4;
x = función1(4, función1(2,3)) + x;
```

Se ilustra un diagrama del espacio, cuando ha terminado la ejecución de la asignación a local1, dentro de la ejecución de la función, pero antes de salir por primera vez de ésta. Los frames se apilan hacia arriba, en la gráfica. Se ha marcado como activo, el frame de la segunda invocación a la función1.

Si una función invoca a otra, en este caso invoca a la misma función, mantiene sus locales y argumentos. Dichas variables existen hasta el término de la ejecución de la función.

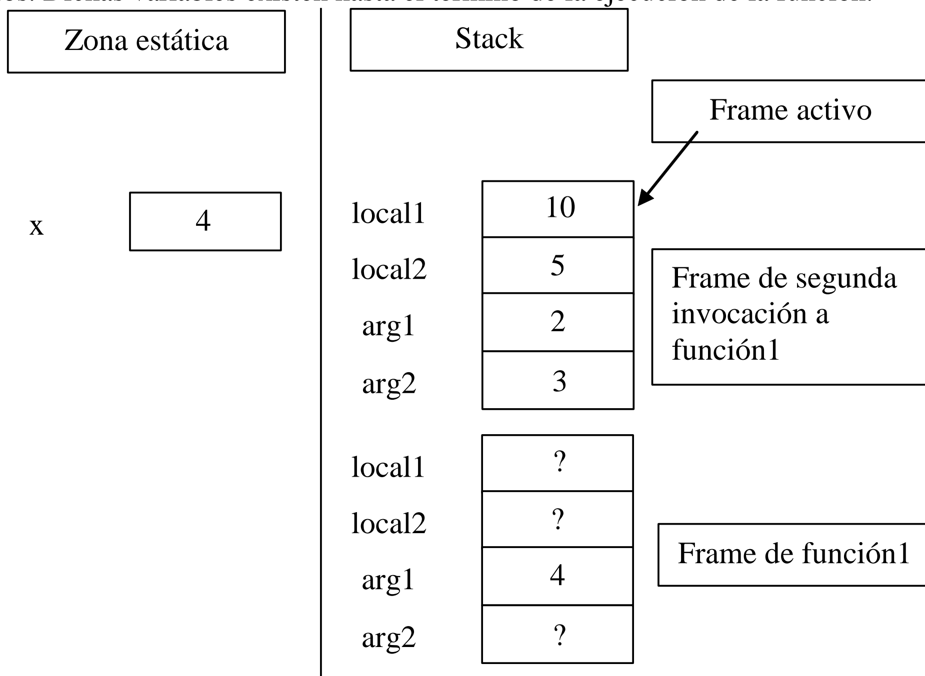

Figura 3.8. Stack después de la segunda invocación a f.

Una vez que retorna, por primera vez de la función, se tiene el valor 13 de arg2, de la primera invocación, y desaparece el frame que se creó en la segunda invocación. Se ilustra la zona después de la asignación a local1.

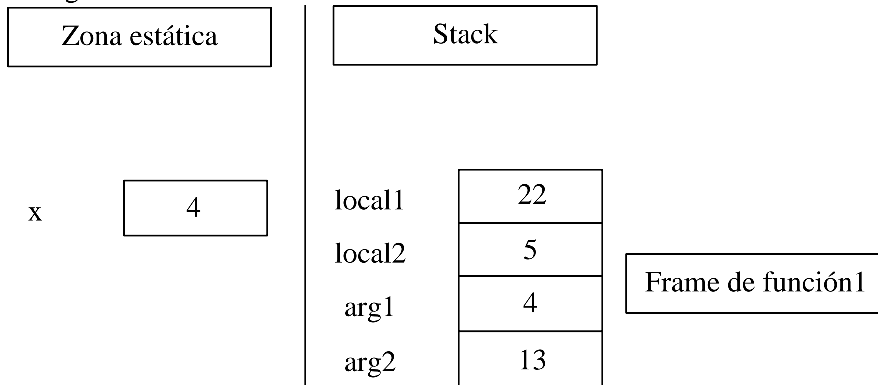

Figura 3.9. Al salir de la segunda invocación.

Finalmente queda x con valor 29, y no está disponible el frame de la función en el stack.

3.2.4. Recursión.

La organización de las variables automáticas, a través de un stack, permite el desarrollo de algoritmos recursivos.

Se define recursión como un proceso en el cual una función se llama a sí misma repetidamente, hasta que se cumpla determinada condición.
Un algoritmo recursivo puede usarse para computaciones repetitivas, en las cuales cada acción se plantea en términos de resultados previos. Dos condiciones deben tenerse en cuenta en estos diseños: Una es que cada llamado a la función conduzca a acercarse a la solución del problema; la otra, es que se tenga un criterio para detener el proceso.

En general los diseños recursivos requieren más espacio de memoria y ocupan mayor tiempo en ejecutarse. Sin embargo generan diseños simples cuando las estructuras de datos quedan naturalmente definidas en términos recursivos. Es el caso de los árboles y de algunas funciones matemáticas.

La recursión es un método para resolver problemas en forma jerárquica (top-down), se parte reduciendo el problema final (top) en partes más simples, hasta llegar a un caso (bottom), en el cual se conoce la solución; y se devuelve ascendiendo hasta el tope, calculando con los valores que se van obteniendo. Cada vez que se activa una invocación de una función recursiva, se crea espacio para sus variables automáticas en un frame; es decir cada una tiene sus propias variables. El cambio de las locales de una invocación no afecta a las locales de otra invocación que esté pendiente (que tenga aún su frame en el stack). Las diferentes encarnaciones de las funciones se comunican los resultados a través de los retornos.

En un **procedimiento iterativo**, también denominado bottom-up, se parte de la base conocida y se construye la solución paso a paso, hasta llegar al caso final.

Más adelante veremos una estructura básica de datos, denominada stack de usuario (no confundir con el stack que maneja las variables automáticas). Se puede demostrar que un algoritmo recursivo siempre se puede plantear en forma iterativa con la ayuda del stack de usuario. Y un programa iterativo, que requiera de un stack de usuario, se puede plantear en forma recursiva (sin stack).

Si existe una forma recursiva de resolver un problema, entonces existe también una forma iterativa de hacerlo; y viceversa.

Consideremos la función matemática factorial, que tradicionalmente está definida en forma recursiva (a través de sí misma).

factorial(0) = 1
factorial(n) = n * factorial(n-1)

La condición para detener la recursión, el caso base, es que factorial de cero es uno. También se puede detener con factorial(1) = 1.

Ejemplo 3.5. Diseño recursivo.

El siguiente diseño recursivo, condiciona la re-invocación de la función cuando se llega al caso base:

```
unsigned int factorial( unsigned int n)
{
  if ( n==0) return (1);
  else return n*factorial(n-1);
}
```

El diseño está restringido a valores de n positivos y pequeños; ya que existe un máximo entero representable, y la función factorial crece rápidamente.

Si se invoca: factorial(4), se producen cinco frames en el stack.
El último frame es producido por la invocación de factorial(0), hasta ese momento ninguna de las funciones ha retornado (todas están ejecutando la acción asociada al else, pero no pueden retornar ya que requieren para calcular el producto, el valor de retorno de la función).
Existen cinco argumentos, de nombre n, con valores diferentes. La ejecución, del caso base (n=0), no invoca nuevamente a la función, ya que ésta está condicionada, y retorna el valor 1; lo cual desactiva el frame con n=0, y pasa a completar la ejecución del llamado factorial(1) que estaba pendiente. En este momento: conoce n, que es uno, y el valor retornado por factorial(0), que también es uno; entonces retorna 1, y elimina el frame.
Sigue la ejecución de factorial(2) del mismo modo, hasta eliminar el último frame, retornando el valor 24.

Ejemplo 3.6. Diseño iterativo.

Similar cálculo se puede realizar en forma iterativa.

```
unsigned int factorial(unsigned int n)
{  int i;
   unsigned int producto = 1;

   for (i = 1; i <= n; i++)   producto *= i;
   return (producto);
}
```

3.2.6. Parámetros pasados por referencia y valor único de retorno.

La metodología empleada en C de pasar los argumentos de las funciones por valor, logra la independencia temporal de las variables, lo cual obliga a copiar los valores de los argumentos en el frame de la función, antes de ejecutar las acciones asociadas a ésta.

Otra limitación del diseño es que una función sólo puede retornar un valor, de esta forma la función sólo se comunica con una única variable mediante la asignación del valor retornado.

Ejemplo 3.7. Comunicación del valor retornado por una función.

Veamos un ejemplo simple:
Se tiene visible una variable x. Se modifica el valor almacenado en x, con el resultado de la invocación a una función f, de dos argumentos, que retorna un entero:

```
int x;

x= f(3, 5);
```

En un caso más general se pueden pasar expresiones como argumentos actuales:

```
x = f(a +b, a*b);
```

Donde a y b deben ser variables visibles, o dentro del alcance de la función, e inicializadas si son variables locales. Antes de invocar a la función, se evalúan las expresiones y se copian los valores resultantes en el espacio de memoria del frame de la función. Luego de la ejecución de las acciones de la función, el único valor retornado por la función es copiado en el espacio asignado a x.

El programador debe asegurar que los tipos declarados para los argumentos formales, en la definición de la función, sean compatibles con los tipos resultantes de las expresiones en el momento del llamado a la función. También es responsabilidad del programador que el tipo declarado para el retorno de la función sea compatible con el tipo de la variable que recibe el resultado de la función.

En un caso más general, el resultado de la función puede formar parte de una expresión, cuyo tipo debe ser compatible con el de la variable a la cual se asigna dicha expresión:

$$x = f(a, b) + c$$

Estas limitaciones pueden superarse si la función escribe o lee *variables globales*. En este caso no hay restricciones para el número de variables que una función puede modificar. Lo anterior también elimina el tener que copiar los valores en argumentos, pudiéndose diseñar funciones con menor número de argumentos de entrada. Esta práctica genera efectos laterales de difícil depuración, en un ambiente en el que varias personas diferentes diseñan las funciones de un programa.

El concepto de puntero permite pasar la referencia a una variable que existe fuera de la función, eliminando, de esta forma, la copia de variables de grandes dimensiones, ya que sólo es preciso copiar el valor del puntero.

También puede modificarse la comunicación de más de un valor si la función retorna el valor de un puntero a una colección (arreglo, string, estructura) agrupada de datos. De esta forma se mantiene el concepto de que una función se comunica con el resto del ambiente, sólo a través de sus argumentos y por único valor de retorno.

La evitación de la copia consiste en mantener en memoria sólo una copia de la variable de interés. Se pasan como argumentos los valores de los punteros que referencian a las variables que se desea ver o modificar, si se desea sólo referenciar variables aparecen éstas precedidas del operador que obtiene la dirección, el &. Mediante la indirección de los punteros se pueden leer o escribir variables externas, dentro de la función, lo cual implica la aparición de asteriscos en el código.
La copia local contiene el valor del puntero, y a través de la indirección se puede modificar la variable externa a la función, además del valor retornado; lo cual debe ser considerado un efecto lateral, y siempre presenta riesgos de generar errores de difícil depuración.

Ejemplo 3.8. Se desea diseñar función que retorne dos resultados.

Sean estos resultados, la suma y la resta de dos variables o valores.

Paso por referencia.

Se desea obtener la suma y diferencia de dos valores cualesquiera, éstos pueden ser pasados por valor.

Si se elige que el retorno de la función entregue la suma de los argumentos, es preciso agregar un argumento que pase el valor de la dirección de variable externa a la función, en donde se escribirá la diferencia:

```
int f(int x, int y, int *dif)
{
  *dif = x – y ;   //escribe la diferencia en variable fuera de la función. Aparece operador *.
  return (x + y); //retorna la suma
}
```

Es conveniente que el código de la función sólo escriba una vez, en la variable pasada por referencia. Lo cual delimita los efectos laterales.
Nótese que la declaración del tipo de la variable por referencia, *recuerda cómo debe ser empleada* la variable dentro de la función. Desde este punto de vista conviene colocar el asterisco precediendo a la variable.

Un ejemplo de invocación, escribiendo la suma en c y la diferencia en d, variables que se asumen definidas:

```
c = f(3, 4, &d);  //aparece el operador &
```

Donde los valores 3 y 4, pueden ser reemplazados por expresiones con valores enteros.

Si se tuviera definida una variable para almacenar el valor del puntero a la variable d, según:

```
int *vpd = &d;  // vpd es de tipo puntero a entero
```

El llamado puede efectuarse sin el &, según:

```
c = f(3, 4, vpd);
```

Lo cual refleja que se está pasando el valor del puntero.

Si se tuviera definido el tipo: puntero a variable de tipo entero según:

```
typedef  int*  pi;
```

La codificación de los argumentos de la función podría escribirse sin asteriscos.

```
int f(int x, int y, pi   pvar)
{
  *pvar = x – y ;   //De todas maneras aparece operador *.
   return (x + y); //retorna la suma
}
```

Ejemplo de uso:

```
pi pd=&d;
c = f(3, 4, pd);
```

Retorno de estructura.

Esta forma posibilita devolver más de un resultado en el retorno de la función.

Se define la estructura dosretornos, para almacenar el resultado.

```
struct  dosretornos
{
    int suma;
    int diferencia;
};                              //declaración de tipo

struct dosretornos g( int x, int y)
{
    struct dosretornos temp;
    temp.suma = x + y;
    temp.diferencia = x-y;
    return (temp);
}
```

Un ejemplo de uso, si se tiene definida una variable c de tipo dos retornos:

```
struct dosretornos c;                  //definición de variable

c = g(3, 4);
```

La suma queda en c.suma, la diferencia en c.diferencia.

Retorno en arreglo.

Un caso particular es cuando todos los valores que se desean retornar son de igual tipo. En esta situación se puede efectuar los retornos en un arreglo.

Se tiene un arreglo de dos posiciones:

```
int a[2];  //variable global
```

La función puede escribir en la variable global

```
void g2( int x, int y)
{
    *a = x + y;   //por global
    *(a+1) = x-y;
}
```

Los resultados quedan en las primeras posiciones del arreglo.

```
  g2(5, 2);
  printf(" %d %d\n", a[0], a[1]);
```

Se puede diseñar una función g3, que escriba en las dos primeras posiciones de un arreglo que se pasa por referencia:

```
void g3( int x, int y, int *arr)
{
    *arr = x + y;
    *(arr+1) = x-y;
}
```

Ahora el llamado se efectúa pasando como tercer argumento el nombre del arreglo.

```
  g3(3, 4, a);
  printf(" %d %d\n", a[0], a[1]);
```

3.2.6. Evitación de la copia de argumentos que ocupan muchos bytes.

En general, en el lenguaje C, los arreglos, strings y estructuras se manipulan con funciones a las cuales se les pasa una referencia. Las funciones pueden retornar una referencia a una agrupación.

Ejemplo 3.9. Árbol binario.

Veamos un primer ejemplo de una estructura que puede ocupar grandes dimensiones.
Se tienen los tipos de datos: nodo y pnodo, definidos según:

```
typedef struct tnode
  {
   int valor;
   struct tnode *left;
   struct tnode *right;
  } nodo, * pnodo;
```

Y se ha formado un árbol binario, con los siguientes nodos.

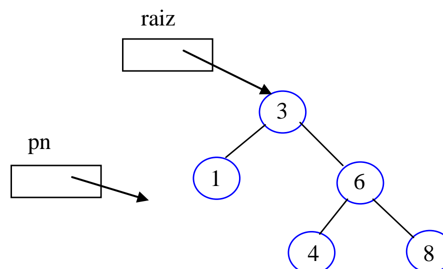

Figura 3.10. Árbol Binario.

Nótese que para cada nodo: los valores de los nodos que están vinculados por la derecha son mayores que los valores de los nodos que están conectados por la izquierda. En la Figura 3.10, se tienen dos variables de tipo puntero a nodo, una apunta a un nodo denominado raíz, el otro (pn) es una variable auxiliar.

Los nodos que no tienen descendientes deben tener valores nulos en los punteros left y right.

Se desea diseñar una función que encuentre el nodo con mayor valor.

No es razonable pasar como argumento el árbol completo a la función, por el espacio requerido en el stack, y porque sería necesario además copiar la estructura completa. Entonces podemos pasar un puntero a un nodo como argumento, esto sólo requiere crear el espacio y copiar el valor de un puntero (normalmente el tamaño ocupado por un entero). El retorno de la función, será un puntero al nodo que tenga el mayor valor.

```
pnodo busca_max(pnodo T)
{  if (T != NULL)
      while (T->right != NULL)  T = T->right; /* Iterativo. Siempre por la derecha */
   return(T);
}
```

La función considera que se la podría invocar con un argumento apuntando a un árbol vacío, en ese caso se decide retornar un puntero con valor nulo, que indica un árbol (o subárbol vacío).
Teniendo en cuenta la propiedad del árbol binario, basta descender por la vía derecha, hasta encontrar un nodo sin descendiente u hoja.

Un ejemplo de uso:

```
pn = busca_max(raiz);
```

En el caso del ejemplo, retornaría un puntero al nodo con valor 8.

Otro ejemplo de uso:

```
pn = busca_max(raiz->left);
```

Retornaría un puntero al nodo con valor 1.

Ejemplo 3.10 Manipulación de strings.

Veremos la forma de tratar funciones que manipulen arreglos o strings.
Antes de desarrollar el ejemplo repasaremos la estructura de datos asociada a un string.

Definición de string.

a) Arreglo de caracteres.

La siguiente definición reserva espacio para un string como un arreglo de caracteres.

La definición de un string como arreglo de caracteres, debe incluir un espacio para el carácter fin de string (el carácter NULL, con valor cero). Quizás es mejor definir el terminador de string como null (o eos: end of string), para evitar confusiones con el valor de un puntero nulo.

char str[6]; /*crea arreglo de chars con espacio para 6 caracteres. Índice varía entre 0 y 5 */
str[5] = NULL; //coloca el fin de string.

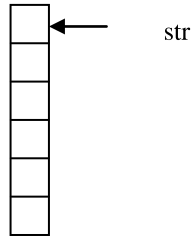

Figura 3.11. Representación en memoria de un string.

Con la definición anterior el string almacenado en el arreglo puede tener un largo máximo de cinco caracteres.

El nombre del arreglo **str**, es un puntero constante que apunta al primer carácter del string, por ser constante no se le puede asignar nuevos valores o modificar. Las direcciones de memoria deben considerarse consecutivas; en la dirección más alta se almacena el fin de string.

b) Puntero a carácter.

La definición de un string como un puntero a carácter, puede ser inicializada asignándole una constante de tipo string. La que se define como una secuencia de cero o más caracteres entre comillas dobles; el compilador agrega el carácter '\0' automáticamente al final.
Si dentro del string se desea emplear la comilla doble debe precedérsela por un \.

En caso de escribir, en el texto de un programa, un string de varias líneas, la secuencia de un \ y el retorno de carro (que es invisible en la pantalla) no se consideran parte del string.

char * str1 = "123456789"; /* tiene 10 caracteres, incluido el NULL que termina el string.*/

Un argumento de tipo puntero a carácter puede ser reemplazado en una lista de parámetros, en la definición de una función por un arreglo de caracteres sin especificar el tamaño. En el caso del ejemplo anterior, podría escribirse: char str1[ ]. La elección entre estas alternativas suele realizarse según sea el tratamiento que se realice dentro de la función; es decir, si las expresiones se elaboran en base a punteros o si se emplea manipulación de arreglos.

En la variable str1 se almacena la dirección de la memoria en donde se almacena el primer byte del string, el cual en este caso es el número 1, con equivalente hexadecimal 0x31.

Nótese que str ocupa el espacio con que fue definido el arreglo, mientras que str1 es un puntero.

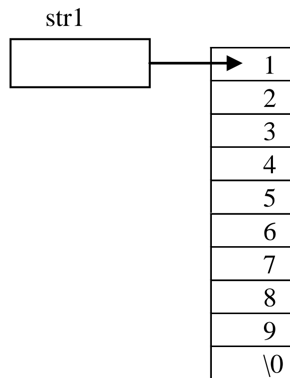

Figura 3.12. Puntero a carácter y el string vinculado.

c) Strcpy. Copia el string fuente en el string destino.

Se detiene la copia después de haber copiado el carácter nulo del fin del string.
Retorna la dirección del string destino.

```
char *strcpy(char * destino, register const char * fuente)
{   register char * cp= destino;
    while(*cp++ = *fuente++) continue;
    return destino;
}
```

Los argumentos y valor de retorno son punteros a carácter. Lo cual evita la copia de los argumentos. A continuación se dan explicaciones de diversos aspectos de la sintaxis del lenguaje.

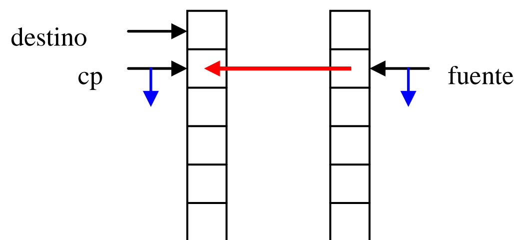

Figura 3.13. Copia de string.

El diagrama ilustra los punteros fuente y cp, después de haberse realizado la copia del primer carácter. Se muestra el movimiento de copia y el de los punteros.

Cuando el contenido de *fuente es el carácter NULL, primero lo copia y la expresión resultante de la asignación toma valor cero, que tiene valor falso para la condición, terminando el lazo while. Copiando correctamente un string nulo.

La instrucción continue puede aparecer en el bloque de acciones de un while, do o for. Su ejecución lleva a reevaluar la condición de continuación del bloque de repetición más interno (en caso de bloques anidados). En el caso de la función anterior podría haberse omitido la instrucción continue; ya que un punto y coma se considera una acción nula.

El operador de postincremento opera sobre un left value (que recuerda un valor que puede colocarse a la izquierda de una asignación). Un lvalue es un identificador o expresión que está relacionado con un objeto que puede ser accesado y cambiado en la memoria.

El uso de estos operadores en expresiones produce un efecto lateral, en el sentido que se efectúan dos acciones. Primero se usa el valor del objeto en la expresión y luego éste es incrementado en uno.

El operador de indirección (el *) y el operador ++ tienen la misma precedencia, entonces se resuelve cuál operador recibe primero el operando mediante su asociatividad, que en el caso de los operadores unarios es de derecha a izquierda. Es decir *fuente++ se interpreta según:

(* (fuente++)) .

La expresión toma el valor del puntero fuente y lo indirecciona, posteriormente incrementa en uno al puntero.

En la expresión (* fuente) ++, mediante el uso de paréntesis se cambia la asociatividad, la expresión toma el valor del objeto apuntado por fuente, y luego incrementa en uno el valor del objeto, no del puntero.

Puede evitarse la acción doble relacionada con los operadores de pre y postincremento, usando éstos en expresiones que sólo contengan dichos operadores. En el caso de la acción de repetición:

```
while(*cp++ = *fuente++) continue;
```

Puede codificarse:

```
while( *cp = *fuente) {cp++, fuente++};
```

Sin embargo los programadores no suelen emplear esta forma. Adicionalmente no producen igual resultado, ya que en la primera forma los punteros quedan apuntando una posición más allá de los caracteres de fin de string; la segunda forma deja los punteros apuntando a los terminadores de los strings. Ambas formas satisfacen los requerimientos de srtcpy.

La primera forma sólo tendría ventajas si el procesador tiene mecanismos de direccionamientos autoincrementados, y si el compilador emplea dichos mecanismos al compilar la primera forma.

Cuando en la lista de parámetros de una función aparece la palabra reservada **const** precediendo a una variable de tipo puntero, el compilador advierte un error si la función modifica la variable a la que el puntero apunta. Además cuando se dispone de diferentes tipos de memorias (RAM, EEPROM o FLASH) localiza las constantes en ROM o FLASH. Si se desea que quede en un segmento de RAM, se precede con **volatile**, en lugar de const.

No se valida si el espacio a continuación de destino puede almacenar el string fuente sin sobreescribir en el espacio asignado a otras variables. Este es un serio problema del lenguaje, y se lo ha empleado para introducir código malicioso en aplicaciones que no validen el rebalse de buffers.

Esta función tiene su prototipo definido en <string.h>

Un ejemplo de uso.

```
#include <string.h>
#include <stdio.h>
char string[10];  /*crea string con espacio para 10 caracteres  */
char * str1 = "abcdefghi";   /* tiene 10 caracteres, incluido el NULL que termina el string.*/

int main(void)
{ strcpy(string, str1);
  printf("%s\n", string);
  printf(str1);   //sin string de formato
  return 0;
}
```

Note que en la invocación se pasan los nombres de los strings, que son considerados punteros constantes. En lugar de string, se podría haber escrito: &string[0].

No se ocupa el retorno de la función, en este caso se usa como procedimiento no como función.

Arreglos de grandes dimensiones no conviene definirlos dentro de la función, ya que podrían producir un rebalse del stack; es preferible definirlos en zona estática o en el heap.

Tradicionalmente se menciona que el diseño de strcpy puede ser difícil de entender. Se muestran a continuación, dos diseños basados en arreglos; y su evolución a códigos basados en punteros.

```
void strcpy1(char destino[], const char fuente[])
 {    int i = 0;
     while (1)
    {
       destino[i] = fuente[i];
       if (destino[i] == '\0') break; // copió fin de string
       i++;
    }
}
```

Debido a que en el lenguaje C, la asignación es una expresión, y por lo tanto tiene un valor, se puede escribir:

```
void strcpy2(char destino[], const char fuente[])
{    int i = 0;
     while ((destino[i] = fuente[i]) != '\0')   i++;
}

// Moviendo los punteros. Resolviendo precedencia por asociatividad.
void strcpy3(char *destino, const char *fuente)
{
   while ((*destino++ = *fuente++) != '\0') ; //Trae el valor, luego incrementa puntero.
}

// Finalmente el fin de string '\0' equivale a valor lógico falso
void strcpy4(char *destino, const char *fuente)
{  while (*destino++ = *fuente++) ; }
```

Las versiones 3 y 4, reflejan en sus argumentos que el diseño está basado en punteros.

La última versión, strcpy4 puede ser difícil de entender. Empleando un compilador que optimice el código assembler en velocidad, entrega costos similares para los cuatro diseños.

El assembler es para el microcontrolador MSP430, el que dispone de instrucciones con autoincrementos.

```
strcpy1o2:    mov.b   @R14+, 0x0(R12)      ;M[R12] = M[R14]; R14++;
              mov.b   @R12+, R15           ;R15 = M[R12]; R12++;
              tst.b    R15                 ;test de R15;
              jne      strcpy1o2           ;Si R15 no es igual a cero repite bloque
              ret                          ;Si R15 es cero retorna

strcpy3o4:    mov.b   @R14+, R15           ; R15      = M[R14]; R14++;
              mov.b   R15, 0x0(R12)        ; M[R12] = R15
              inc.w   R12                  ; R12++;
              tst.b   R15
              jne     strcpy3o4
              ret
```

En este caso la versión mediante arreglos emplea una instrucción menos que las versiones con punteros. La decisión de cual es el mejor código resultará de la comparación de los ciclos de reloj que tarda la ejecución de las instrucciones del bloque repetitivo, ya que cada instrucción puede durar diferentes ciclos de reloj.

La invocación a las funciones se logra pasando los argumentos vía registros R12 y R14.

```
mov.w    #str1, R10      ; dirección de str1 (destino) en R10
mov.w    #str2, R11   ; dirección de str2 (fuente) en R11
mov.w   @R11,R14  ; R14 = M[R11] = contenido str2
mov.w   @R10,R12  ; R12 = M[R10] = contenido str1
call         strcpy
```

La moraleja de esto es escribir código del cual se tenga seguridad de lo que realiza.

Y dejar a los compiladores optimizantes el resto.

3.3. Manejo dinámico de la memoria en C.

3.3.1. Asignación, Referencias y tiempo de vida.

El compilador asigna un espacio determinado para las variables y genera las referencias para accesar a las variables del stack y de la zona estática. El tamaño de las variables no puede cambiarse durante la ejecución del programa, es asignado en forma estática.

El tiempo de vida de las variables de la zona estática es la duración del programa; las variables denominadas automáticas, o en la zona del stack, existen durante la ejecución de la función que las referencia. Los frames en el stack, son asignados y desasignados en forma dinámica durante la ejecución de las funciones; pero en forma automática, el programador no tiene responsabilidad en ese proceso.

En el heap el programador debe solicitar la asignación de espacio, establecer las referencias entre el espacio asignado y las variables en las otras zonas, liberar el espacio, desasignar las referencias. Cualquier equivocación lleva a errores, en tiempo de ejecución, difíciles de depurar.

Este mecanismo permite al programador tener un mayor control de la memoria, tanto en tamaño como en tiempo de vida, pero al mismo tiempo le da la responsabilidad de administrarla correctamente. Un arreglo en la zona estática debe ser definido con un tamaño determinado, el cual no puede cambiar durante la ejecución del programa, sin embargo en el heap se puede solicitar un arreglo del tamaño que se desee, siempre que no exceda el tamaño máximo asignado al heap.

Escribir programas que manejen el heap es notablemente más difícil, por esta razón lenguajes más modernos efectúan automáticamente la programación del heap, y no le permiten al programador realizar esta tarea. Algunos errores de programas que manejan el heap, compilan correctamente, sin embargo al ejecutarlos se producen errores difíciles de depurar.

3.3.2. Funciones para el manejo del heap.

En <stdlib.h> están los prototipos de las funciones de biblioteca que asignan y desasignan bloques de memoria. Describiremos las dos fundamentales.

3.3.2.1. void * malloc(size_t *tamaño)*

Solicita un bloque contiguo de memoria del segmento heap, del *tamaño* especificado en bytes, y retorna un puntero de tipo genérico, el cual puede ser moldeado (cast) a cualquier tipo determinado, al inicio del bloque; retorna NULL si no existen bloques del tamaño solicitado dentro del heap.

El programador debe asignar el valor retornado a una variable de tipo puntero, estableciendo la referencia; dicho puntero debe existir en alguna de las otras zonas de memoria. La única forma de establecer la referencia es mediante un puntero, que inicialmente debe apuntar a NULL.

El contenido del bloque debe ser inicializado antes de ser usado, ya que inicialmente contiene los valores que estaban previamente almacenados en el bloque del heap.

Suele emplearse el operador **sizeof**(tipo o nombre_de_variable) que retorna un valor entero sin signo con el número de bytes con que se representa el tipo o la variable, para calcular el tamaño del bloque que debe solicitarse.

3.3.2.2. void free(void * *puntero*)

Free libera el bloque apuntado por *puntero* y lo devuelve al heap. El valor del puntero debe haber sido obtenido a través de malloc; produce errores difíciles de depurar invocar a free con un puntero no devuelto por malloc.

Después de ejecutar free, el programador no puede volver a referenciar datos a través de ese puntero; debería asignarle NULL, para prevenir errores. Tampoco puede liberarse un bloque más de una vez.

Es importante liberar el bloque cuando el programa no siga utilizando los datos que almacenó en el bloque, de esta forma puede volver a utilizarse dicho espacio. En caso de no hacerlo, van quedando bloques inutilizables en el heap, lo cual origina en el largo plazo la fragmentación y finalmente el rebalse de éste.

El administrador del heap, que es invocado a través de las funciones anteriores, mantiene sus propias estructuras de datos en las cuales registra los bloques que están ocupados y los que están disponibles, y también el tamaño de los bloques. También interactúa con el sistema operativo para solicitar memoria adicional en caso de crecimiento del heap.

Ejemplo 3.11. Arreglo dinámico de enteros.

El siguiente ejemplo emplea una función para crear, usar y liberar un arreglo dinámico de enteros de tamaño determinado por el argumento *size* de la función.

```
void UsaArregloDinámico(unsigned int size)
{
   int * Arreglo;
   int i;

   /*Usar el arreglo antes de asignarlo, provoca errores */

   if ( (Arreglo = (int *) malloc(size * sizeof(int)) ) == NULL) {
      printf ("Memoria insuficiente para Arreglo\n");
      exit(1);
   }
    /* Se puede usar el Arreglo. Pero sus valores no están iniciados.  */
   for(i=0; i<size; i++) Arreglo[i]=0;  //

   /* aquí puede usarse el arreglo…..*/

   free(Arreglo);  /*después de devolver el bloque no se puede referenciar el Arreglo  */
}
```

Es práctica recomendable condicionar la asignación inicial del puntero Arreglo, a que exista espacio disponible en el heap. Y salir del programa, a través de exit y comentado la causa, en caso de no existir memoria disponible. Una alternativa es codificar, mediante assert (requiere incluir assert.h), que aborta la ejecución si su argumento es falso:

```
Arreglo = malloc(size * sizeof(int)) ;
assert(Arreglo != NULL);
/*Si la ejecución continua, significa que había espacio, y que se referenció el puntero */
```

Una vez depurado el programa, conviene emplear el código condicional en lugar de assert, ya que esta función almacena el sendero de la función que la invoca, además del texto de la condición y de los números de líneas del archivo fuente.

Debido a que el puntero Arreglo es variable local, debe liberarse el bloque dentro de la función. Después de salir de la función, queda indefinida la variable Arreglo.
El esquema empleado en este ejemplo, es que la función que solicita el espacio, sea la responsable de liberarlo.

El administrador del heap, también dispone de la función **realloc** que permite cambiar dinámicamente el tamaño de un bloque, y a la vez se encarga de copiar los valores previos de manera eficiente. El nuevo tamaño puede ser mayor o menor que el original; si es menor sólo se copian los datos existentes; si es mayor, los nuevos elementos se consideran no iniciados. El nuevo espacio sigue siendo contiguo, el administrador se encarga de conseguir un nuevo bloque, si es necesario, de copiar los valores, y de liberar el bloque original.

```
Arreglo = realloc(Arreglo,  sizeof(int)*newsize);
assert(Arreglo != NULL);
```

Ejemplo 3.12. Strings dinámicos.

La función CreaString ilustra un uso común del heap, el cual es permitir almacenar strings de largo variable con eficiente uso de la memoria. Si se emplean variable estáticas, el programador debe asegurar que el tamaño de éstas sea suficiente para almacenar los strings, lo cual lleva a reservar un espacio, que en la mayoría de las ocasiones no será empleado. A pesar de esto puede que se presente un string aún mayor que el espacio fijo reservado, llevando a un rebalse del buffer, con resultados impredecibles en la ejecución.

```
char*  CreaString(const char* fuente){
    char* nuevoString;
    nuevoString = (char*) malloc(strlen(fuente) + 1);
                    /*agrega espacio para el fin de string '\0' */
     assert(nuevoString != NULL);
     strcpy(nuevoString, fuente);
     return(nuevoString);}
```

La función CreaString retorna un puntero al string creado en el heap, e iniciado con el valor del string fuente. Las funciones strlen y strcpy tienen sus prototipos en <string.h>
La función que invoca a CreaString es responsable de liberar el espacio, lo cual se ilustra en el siguiente segmento:

```
char * texto;

texto = CreaString("espacio   ");
/*A partir de aquí se puede usar el string texto.  */

free(texto);
```

La omisión de la liberación del espacio limita la reutilización de la memoria dedicada al heap, pudiendo producir el rebalse de éste, ocasionando errores de difícil depuración. El programador debe estar consciente de cuáles funciones invocan a malloc, para acompañarlas de la liberación del espacio.

Ejemplo 3.13. Matriz de enteros, de r renglones y n columnas.

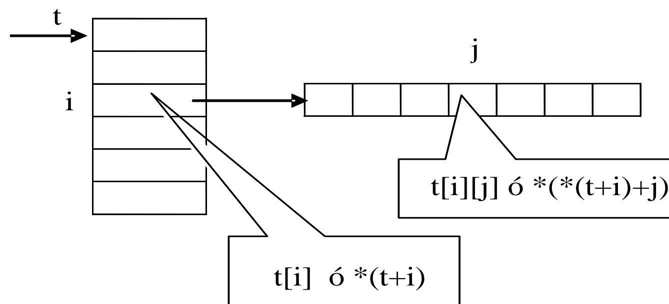

Figura 3.14. Matriz. Arreglo de punteros a renglones.

La matriz se inicia con el valor val, y retorna un puntero a un puntero a entero.

```
int **CreaMatriz(int r, int c, int val)
  { int i, j;
    int **t = malloc(r * sizeof(int *));   /*crea arreglo de punteros a enteros */
            assert(t !=NULL);
      for (i = 0; i < r; i++)
            {  t[i] = malloc(c * sizeof(int));   /*crea arreglo de c enteros  */
               assert(t[i] !=NULL);
            }
      for (i = 0; i < r; i++)
          for (j = 0; j < c; j++)
              t[i][j] = val;
    return t;
  }

void BorreMatriz(int ** p, int r)
  { int i;
      int **t = p;
      for (i = 0; i < r; i++) free(t[i]);  /*deben liberarse primero los renglones */
      free(t);
   }
```

El siguiente segmento ilustra el uso de las funciones:

```
int **m;
m=CreaMatriz(5, 6, -1);
/*se usa la matriz*/

BorreMatriz(m, 5);  /*una vez empleada. Se libera el espacio*/
```

La creación de listas, árboles, colas, stacks, grafos, etc. puede realizarse eficientemente en el heap.

Ejemplo 3.14. Crear nodo de un árbol binario.

Se declara la estructura de un nodo.

```
typedef struct tnode
  {
    int valor;
    struct tnode *left;
    struct tnode *right;
  } nodo, * pnodo;
```

La siguiente función solicita y asigna el nodo obtenido en el heap, si es que había espacio disponible, retornado un puntero al nodo, y dejando iniciadas las variables del nodo.

```
pnodo CreaNodo(int dato)
{
  pnodo pn=NULL;
  if ( (pn= (pnodo) malloc(sizeof(nodo))) ==NULL) exit(1);
  else
   {
     pn->valor=dato; pn->left=NULL; pn->right=NULL;
   }
   return(pn);
}

void LiberaNodo( pnodo pn)
{
   free( pn);   //Libera el espacio
}
```

El siguiente segmento ilustra el uso de las funciones.

```
pnodo root=NULL;  /* el espacio de la variable root existe desde su definición.
root=CreaNodo(5);       //se pega el nodo a la raíz

LiberaNodo(root);
```

Índice general.

CAPÍTULO 3 ..... **1**

ADMINISTRACIÓN DE LA MEMORIA EN C. **1**

3.1. MANEJO ESTÁTICO DE LA MEMORIA. 1
3.2. MANEJO AUTOMÁTICO DE LA MEMORIA EN C. 2
3.2.1. Asignación, Referencias y tiempo de vida. *2*
3.2.2. Argumentos y variables locales. *2*
Ejemplo 3.1. Función con dos argumentos de tipo valor, con dos locales, retorno de entero. 3
3.2.3. Visión lógica del stack. *4*
Ejemplo 3.2. Riesgos de la desaparición del frame. 4
3.2.4. Copia de argumentos. *5*
Ejemplo 3.3. Una función f que invoca a otra función g. 5
Ejemplo 3.4. La misma función anterior invocada dos veces. 8
3.2.4. Recursión. *9*
Ejemplo 3.5. Diseño recursivo. 10
Ejemplo 3.6. Diseño iterativo. 11
3.2.6. Parámetros pasados por referencia y valor único de retorno. *11*
Ejemplo 3.7. Comunicación del valor retornado por una función. 11
Ejemplo 3.8. Se desea diseñar función que retorne dos resultados. 12
Paso por referencia. 12
Retorno de estructura. 13
Retorno en arreglo. 14
3.2.6. Evitación de la copia de argumentos que ocupan muchos bytes. *15*
Ejemplo 3.9. Árbol binario. 15
Ejemplo 3.10 Manipulación de strings. 16
Definición de string. 16
a) Arreglo de caracteres. 16
b) Puntero a carácter. 17
c) Strcpy. Copia el string fuente en el string destino. 18
3.3. MANEJO DINÁMICO DE LA MEMORIA EN C. 22
3.3.1. Asignación, Referencias y tiempo de vida. *22*
3.3.2. Funciones para el manejo del heap. *22*
3.3.2.1. void * malloc(size_t *tamaño)* 23
3.3.2.2. void free(void * *puntero*) 23
Ejemplo 3.11. Arreglo dinámico de enteros. 23
Ejemplo 3.12. Strings dinámicos. 25
Ejemplo 3.13. Matriz de enteros, de r renglones y n columnas. 25
Ejemplo 3.14. Crear nodo de un árbol binario. 26
ÍNDICE GENERAL. 28
ÍNDICE DE FIGURAS. 29

Índice de figuras.

FIGURA 3.1. SEGMENTOS DE MEMORIA. 1
FIGURA 3.1A. STACK DESPUÉS DE INVOCAR A LA FUNCIÓN 3
FIGURA 3.2. STACK AL SALIR DE LA FUNCIÓN. 4
FIGURA 3.3. STACK ANTES DE INVOCAR A LA FUNCIÓN G. 6
FIGURA 3.4. AL ENTRAR EN LA FUNCIÓN G. 6
FIGURA 3.5. STACK JUSTO ANTES DE SALIR DE LA FUNCIÓN G. 7
FIGURA 3.6. STACK AL SALIR DE LA FUNCIÓN G. 7
FIGURA 3.7. STACK AL SALIR DE LA FUNCIÓN F. 8
FIGURA 3.8. STACK DESPUÉS DE LA SEGUNDA INVOCACIÓN A F. 8
FIGURA 3.9. AL SALIR DE LA SEGUNDA INVOCACIÓN. 9
FIGURA 3.10. ÁRBOL BINARIO. 15
FIGURA 3.11. REPRESENTACIÓN EN MEMORIA DE UN STRING. 17
FIGURA 3.12. PUNTERO A CARÁCTER Y EL STRING VINCULADO. 18
FIGURA 3.13. COPIA DE STRING. 18
FIGURA 3.14. MATRIZ. ARREGLO DE PUNTEROS A RENGLONES. 25