CAPÍTULO 5

ÁMBITO Y VIDA DE LAS VARIABLES

Este breve capítulo pretende completar algunos conceptos presentados en el capítulo 3 y que, una vez hemos visto algo de código en el capítulo 4, serán ahora más sencillos de presentar y de comprender. También presentamos una breve descripción de cómo se gestiona el almacenamiento de los datos dentro del ordenador.

Ámbito y Vida.

Entendemos por **ámbito de una variable** el lugar, dentro de un programa, en el que esta variable tiene significado. Hasta el momento todas nuestras variables han tenido como ámbito todo el programa, y quizá ahora no es sencillo hacerse una idea intuitiva de este concepto; pero realmente, no todas las variables están “en activo” a lo largo de todo el programa.

Además del ámbito, existe otro concepto, que podríamos llamar **extensión o tiempo de vida**, que define el intervalo de tiempo en el que el espacio de memoria reservado por una variable sigue en reserva; cuando la variable "muere", ese espacio de memoria vuelve a estar disponible para otros usos que el ordenador requiera. También este concepto quedará más aclarado a medida que avancemos en este breve capítulo.

El almacenamiento de las variables y la memoria.

Para comprender las diferentes formas en que se puede crear una variable, es conveniente describir previamente el modo en que se dispone la memoria de datos en el ordenador.

Hay diferentes espacios donde se puede ubicar una variable declarada en un programa:

1. **Registros**. El registro es el elemento más rápido de almacenamiento y acceso a la memoria. La memoria de registro está ubicada directamente dentro del procesador. Sería muy bueno que toda la memoria fuera de estas características, pero de hecho el número de registros en el procesador está muy limitado. El compilador decide qué variables coloca en estas posiciones privilegiadas. El programador no tiene baza en esa decisión. El lenguaje C permite sugerir, mediante algunas palabras clave, la conveniencia o inconveniencia de que una determinada variable se cree en este espacio privilegiado de memoria.
2. **La Pila**. La memoria de pila reside en la memoria RAM (Random Access Memory: memoria de acceso aleatorio) De la memoria RAM es de lo que se habla cuando se anuncian "los megas" o "gigas" que tiene la memoria de un ordenador.

   El procesador tiene acceso y control directo a la pila gracias al "puntero de pila", que se desplaza hacia abajo cada vez que hay que

reservar más memoria para una nueva variable, y vuelve a recuperar su posición hacia arriba para liberar esa memoria. El acceso a la memoria RAM es muy rápido, sólo superado por el acceso a registros. El compilador debe conocer, mientras está creando el programa, el tamaño exacto y la vida de todas y cada una de las variables implicadas en el proceso que se va a ejecutar y que deben ser almacenados en la pila: el compilador debe generar el código necesario para mover el puntero de la pila hacia abajo y hacia arriba. Esto limita el uso de esta buena memoria tan rápida. Hasta el capítulo 10, cuando hablemos de la asignación dinámica de la memoria, todas las variables que empleamos pueden existir en la pila, e incluso algunas de ellas en las posiciones de registro.

3. **El montículo**. Es un espacio de memoria, ubicada también en la memoria RAM, donde se crean las variables de asignación dinámica. Su ventaja es que el compilador no necesita, al generar el programa, conocer cuánto espacio de almacenamiento necesita asignar al montículo para la correcta ejecución del código compilado. Esta propiedad ofrece una gran flexibilidad al código de nuestros programas. A cambio hay que pagar un precio con la velocidad: lleva más tiempo asignar espacio en el montículo que tiempo lleva hacerlo en la pila.

4. **Almacenamiento estático**. El almacenamiento estático contiene datos que están disponibles durante todo el tiempo que se ejecuta el programa. Más adelante, en este capítulo, veremos cómo se crean y qué características tienen las variables estáticas.

5. **Almacenamiento constante**. Cuando se define un valor constante, éste se ubica habitualmente en los espacios de memoria reservados para el código del programa: lugar seguro, donde no se ha de poder cambiar el valor de esa constante.

Variables Locales y Variables Globales

Una variable puede definirse fuera de la función principal: en el programa, pero no en una función. Esas variables se llaman **globales**, y son válidas en todo el código que se escriba en ese programa. Su espacio de memoria queda reservado mientras el programa esté en ejecución. **Diremos que son variables globales, que su ámbito es todo el programa y que su vida perdura mientras el programa esté en ejecución**.

Veamos como ejemplo el siguiente código:

```
long int Fact;
#include <stdio.h>
void main(void)
{
      short int n;
      printf("Introduce el valor de n ... ");
      scanf("%hd",&n);
      printf("El factorial de %hd es ... ",n);
      Fact = 1;
      while(n) Fact *=n--;
      printf("%ld",Fact);
}
```

La variable *n* es local: su ámbito es únicamente el de la función principal *main*. La variable *Fact* es global: su ámbito se extiende a todo el programa.

Advertencia: salvo para la declaración de variables globales (y declaración de funciones, que veremos más adelante), el lenguaje C no admite ninguna otra sentencia fuera de una función.

Ahora mismo este concepto nos queda fuera de intuición porque no hemos visto aún la posibilidad de crear y definir en un programa otras funciones, aparte de la función principal. Pero esa posibilidad existe, y en ese caso, si una variable es definida fuera de cualquier función, entonces esa variable es accesible desde todas las funciones del programa.

No se requiere ninguna palabra clave del lenguaje C para indicar al compilador que esa variable concreta es global.

Se recomienda, en la medida de lo posible, **no hacer uso de variables globales**. Cuando una variable es manipulable desde cualquier ámbito de un programa es fácil sufrir efectos imprevistos.

Una variable será **local** cuando se crea en un bloque del programa, que puede ser una función, o un bloque interno de una función.

Por ejemplo:

```
long x = 12;
// Sólo x está disponible.
{
      long y = 25;
// Tanto x como y están disponibles.
}
// La variable y está fuera de ámbito. Ha terminado su vida.
```

El ámbito de una variable local será el del bloque en el que está definida. En C, puede declararse una variable local, con un nombre idéntico al de una variable global; entonces, cuando en ese ámbito local se haga referencia al nombre de esa variable, se entenderá la variable local: en ese ámbito no se podrá tener acceso a la variable global, cuyo nombre “ha sido robado” por una local. Por ejemplo:

```
long x = 12;
// Sólo x está disponible.
{
      long x = 25;
// En este bloque la única variable x accesible vale 25.
}
// La única variable x en este ámbito vale 12.
```

También pueden definirse variables locales del mismo nombre en ámbitos diferentes y disjuntos, porque al no coincidir en ámbito en ninguna sentencia, no puede haber equívoco y cada variable, del mismo nombre, existe sólo en su propio ámbito. Por ejemplo:

```
long x = 12;
// Sólo x está disponible.
{
      long y = 25;
```

```
// Tanto x como y están disponibles.
}
// La variable y está fuera de ámbito. Ha terminado su vida.
{
      long y = 40;
// Tanto x como y están disponibles.
// Esta variable y no es la misma que la otra.
/* La declaración de la variable y es correcta, puesto que
   la anterior declaración de una variable con el mismo
   nombre fue en otro ámbito. */
}
// La variable y está fuera de ámbito. Ha terminado su vida.
```

Veamos un ejemplo sencillo de uso de diferentes variable locales:

```
unsigned short i;
for(i = 2 ; i < 10000 ; i++)
{
      unsigned short suma = 1;
      for(unsigned short j = 2 ; j <= i / 2 ; j++)
            if(i % j == 0) suma += j;
      if(suma == i)printf("%hu",i)
}
```

En este código, que como vimos permite buscar los números perfectos entre los primeros 10000 enteros, declara dos variables (*j* y *suma*) en el bloque de la estructura del primer ***for***; la variable *j* está declarada aún más local, en el interior del segundo ***for***. Al terminar la ejecución del ***for*** gobernado por la variable *i*, esas dos variables dejan de existir; la variable *j* muere nada más se abandona el espacio de ejecución de la sentencia iterada por el segundo ***for***. Si a su vez, la estructura ***for*** más externa estuviera integrada dentro de otra estructura de iteración, cada vez que se volviera a ejecutar ese ***for*** se volverían a crear esas dos variables, que tendrían el mismo nombre, pero no necesariamente las mismas direcciones de memoria que antes.

Hay una palabra en C para indicar que la variable es local. Es la palabra reservada ***auto***. Esta palabra rara vez se utiliza, porque el compilador descubre siempre el ámbito de las variables gracias al lugar donde se recoge la declaración.

Un ejemplo muy simple puede ayudar a presentar estas ideas de forma más clara:

```
#include <stdio.h>

long b = 0, c = 0;
void main(void)
{
      for(long b = 0 ; b < 10 ; b++) c++;
      printf("El valor de b es %ld y el de c es %ld", b, c);
}
```

Las variables *b* y *c* han sido declaradas globales. Y ambas han sido inicializadas a cero. Luego, dentro de la función principal, se ha declarado, local dentro del ***for***, la variable *b*. Y dentro del ***for*** se han variado los valores de las variables *b* y *c*.

¿Cuál es la salida que ofrecerá por pantalla este código? Por lo que respecta a la variable *c* no hay ninguna duda: se ha incrementado diez veces, y su valor, después de ejecutar la estructura ***for***, será 10. Pero, ¿y *b*? Esta variable ha sufrido también una variación y ha llegado al valor 10. Pero… ¿cuál de los dos variables *b* ha cambiado?: la de ámbito más local. Y como la sentencia que ejecuta la función *printf* ya está fuera de la estructura ***for***, y para entonces la variable local *b* ya ha muerto, la variable b que muestra la función printf no es otra que la global: la única viva en este momento. La salida que mostrará el programa es la siguiente: *El valor de b es 0 y el de c es 10*.

Una advertencia importante: ya se ha visto que se pueden declarar, en ámbitos más reducidos, variables con el mismo nombre que otras que ya existen en ámbitos más globales. Lo que no se puede hacer es declarar, en un mismo ámbito, dos variables con el mismo nombre. Ante esa circunstancia, el compilador dará error y no compilará.

Una última observación sobre las variables locales: E**l lenguaje C requiere que todas las variables se definan al principio del bloque donde tienen su ámbito**: esas declaraciones de variables deben ser las primeras sentencias en cada bloque que tenga variables locales en él. Así, cuando el compilador crea el bloque, puede asignar el espacio exacto requerido para esas variables en la pila de la memoria. En C++ es posible diseminar las declaraciones de las distintas variables

a lo largo del bloque, definiéndolas en el momento en que el programador requiere de su uso. Si se programa en un entorno de C++, se podrá por tanto diseminar esas declaraciones; pero eso es propiedad de C++, y el programa generado no podría ser compilado en un compilador de C.

Es conveniente por tanto, cuando se pretende aprender a programar en C, imponerse la disciplina de agrupar todas las declaraciones al principio de cada bloque, como es exigido en la sintaxis de C. Aunque el programa compilase en un compilador de C++, el código sería sintácticamente erróneo desde el punto de vista de un compilador de C.

Variables estáticas y dinámicas.

Con respecto a la extensión o tiempo de vida, las variables pueden ser estáticas o dinámicas. Será **estática** aquella variable que una vez definida, persiste hasta el final de la ejecución del programa. Y será **dinámica** aquella variable que puede ser creada y destruida durante la ejecución del programa.

No se requiere ninguna palabra clave para indicar al compilador que una variable creada es dinámica. Sí es en cambio necesario indicar al compilador, mediante la palabra clave ***static***, cuando queremos que una variable sea creada estática. Esa variable puede ser local, y en tal caso su ámbito será local, y sólo podrá ser usada cuando se estén ejecutando sentencias de su ámbito; pero su extensión será la misma que la del programa, y siempre que se vuelvan a las sentencias de su ámbito allí estará la variable, ya creada, lista para ser usada. Cuando terminen de ejecutarse las sentencias de su ámbito esas posiciones de memoria no serán accesibles, porque estaremos fuera de ámbito, pero tampoco podrá hacerse uso de esa memoria para otras variables, porque la variable estática seguirá “viva” y esa posición de memoria sigue

almacenando el valor que quedó de la última vez que se ejecutaron las sentencias de su ámbito.

Cuando se crea una variable local dentro de una bloque, o dentro de una función, el compilador reserva espacio para esa variable cada vez que se llama a la función: mueve en cada ocasión hacia abajo el puntero de pila tanto como sea preciso para volver a crear esa variable. Si existe un valor inicial para la variable, la inicialización se realiza cada vez que se pasa por ese punto de la secuencia.

Si se quiere que el valor permanezca durante la ejecución del programa entero, y no sólo cada vez que se entra de nuevo en el ámbito de esa variable, entonces tenemos dos posibilidades: La primera consiste en crear esa variable como global, extendiendo su ámbito al ámbito de todo el programa (en este caso la variable no queda bajo control del bloque donde queríamos ubicarla, o bajo control único de la función que la necesita, sino que es accesible (se puede leer y se puede variar su valor) desde cualquier sentencia del programa); La segunda consiste en crear una variable ***static*** dentro del bloque o función. El almacenamiento de esa variable no se lleva a cabo en la pila sino en el área de datos estáticos del programa. La variable sólo se inicializa una vez —la primera vez que se llama a la función—, y retiene su valor entre diferentes invocaciones.

Veamos el siguiente ejemplo, donde tenemos dos variables locales que sufren las mismas operaciones: una estática (la variable que se ha llamado *a*) y la otra no (la que se ha llamado *b*):

```
#include <stdio.h>

void main(void)
{
      for(long i = 0 ; i < 3 ; i++)
            for(long j = 0 ; j < 4 ; j++)
            {
                  static long a = 0;
                  long b = 0;

                  for(long j = 0 ; j < 5 ; j++, a++, b++);
```

```
                printf("a = %3ld.   b = %3ld.\n", a, b);
            }
}
```

El programa ofrece la siguiente salida por pantalla:

a = 5. b = 5.
a = 10. b = 5.
a = 15. b = 5.
a = 20. b = 5.
a = 25. b = 5.
a = 30. b = 5.
a = 35. b = 5.
a = 40. b = 5.
a = 45. b = 5.
a = 50. b = 5.
a = 55. b = 5.
a = 60. b = 5.

Variables en registro.

Cuando se declara una variable, se reserva un espacio de memoria para almacenar sus sucesivos valores. Cuál sea ese espacio de memoria es cuestión que no podemos gobernar del todo. Especialmente, como ya se ha dicho, no podemos decidir cuáles son las variables que deben ubicarse en los espacios de registro.

Pero el compilador, al traducir el código, puede detectar algunas variables empleadas de forma recurrente, y decidir darle esa ubicación preferente. En ese caso, no es necesario traerla y llevarla de la ALU a la memoria y de la memoria a la ALU cada vez que hay que operar con ella.

El programador puede tomar parte en esa decisión, e indicar al compilador que alguna o algunas variables conviene que se ubiquen en los registros de la ALU. Eso se indica mediante la palabra clave ***register***.

Si al declarar una variable, se precede a toda la declaración la palabra ***register***, entonces esa variable queda creada en un registro de la ALU.

Una variable candidata a ser declarada ***register*** es, por ejemplo, las que actúan de contadoras en estructuras ***for***.

También puede ocurrir que no se desee que una variable sea almacenada en un registro de la ALU. Y quizá se desea indicar al compilador que, sea cual sea su opinión, una determinada variable no debe ser almacenada allí sino en la memoria, como una variable cualquiera normal. Para evitar que el compilador decida otra cosa se le indica con la palabra ***volatile***.

El compilador tomas las indicaciones de ***register*** a título orientativo. Si, por ejemplo, se ha asignado el carácter de ***register*** a más variables que permite la capacidad de la ALU, entonces el compilador resuelve el conflicto según su criterio, sin abortar el proceso de compilación.

Variables ***extern***

Aunque estamos todavía lejos de necesitar este tipo de declaración, presentamos ahora esta palabra clave de C, que hace referencia al ámbito de las variables.

El lenguaje C permite trocear un problema en diferentes módulos que, unidos, forman una aplicación. Estos módulos muchas veces serán programas independientes que después se compilan por separado y finalmente se "linkan" o se juntan. Debe existir la forma de indicar, a cada uno de esos programas desarrollados por separado, la existencia de variables globales comunes para todos ellos. Variables cuyo ámbito trasciende el ámbito del programa donde se declaran, porque abarcan todos los programas que luego, "linkados", darán lugar a la aplicación final.

Se podrían declarar todas las variables en todos los archivos. C en la compilación de cada programa por separado no daría error, y asignaría tanta memoria como veces estuvieran declaradas. Pero en el enlazado daría error de duplicidad.

Para evitar ese problema, las variable globales que deben permanecer en todos o varios de los módulos de un programa se declaran como ***extern*** en todos esos módulos excepto en uno, donde se declara como variable global sin la palabra ***extern***. Al compilar entonces esos módulos, no se creará la variable donde esté puesta la palabra ***extern***, y permitirá la compilación al considerar que, en alguno de los módulos de linkado, esa variable sí se crea. Evidentemente, si la palabra ***extern*** se coloca en todos los módulos, entonces en ninguno se crea la variable y se producirá error en el linkado.

El identificador de una variable declarada como ***extern*** es conveniente que no tenga más de seis caracteres, pues en los procesos de linkado de módulos sólo los seis primeros caracteres serán significativos.

En resumen…

Ámbito:

El ámbito es el lugar del código donde las sentencias pueden hacer uso de una variable.

Una variable **local** queda declarada en el interior de un bloque. Puede indicarse ese carácter de local al compilador mediante la palabra auto. De todas formas, la ubicación de la declaración ofrece suficientes pistas al compilador para saber de la localidad de cada variable. Su ámbito queda localizado únicamente a las instrucciones que quedan dentro del bloque donde ha sido creada la variable.

Una variable es **global** cuando queda declarada fuera de cualquier bloque del programa. Su ámbito es todo el programa: cualquier sentencia de cualquier función del programa puede hacer uso de esa variable global.

Extensión:

La extensión es el tiempo en que una variable está viva, es decir, en que esa variable sigue existiendo en la memoria.

Una variable global debe existir mientras el programa esté en marcha, puesto que cualquier sentencia del programa puede hacer uso de ella.

Una variable local sólo existe en el intervalo de tiempo transcurrido desde la ejecución de la primera sentencia del bloque donde se ha creado esa variable y hasta que se sale de ese bloque. Es, tras la ejecución de la última sentencia del bloque, el momento en que esa variable desaparece. Si el bloque vuelve a ejecutarse entonces vuelve a crearse una variable con su mismo nombre, que se ubicará donde antes, o en otra dirección de memoria diferente: es, en todo caso, una variable diferente a la anterior.

Se puede forzar a que una variable local exista durante toda la ejecución del programa. Eso puede hacerse mediante la palabra reservada de C ***static***. En ese caso, al terminar la ejecución de la última instrucción del bloque donde está creada, la variable no desaparece. De todas formas, mientras no se vuelva a las sentencias de ese bloque, esa variable no podrá ser reutilizada, porque fuera de ese bloque, aún estando viva, está fuera de su ámbito.

Ubicación: Podemos indicar al compilador si queremos que una variable sea creada en los registros de la ALU, utilizando la palabra reservada ***register***. Podemos indicarla también al compilador que una variable no se cree en esos registros, mediante la palabra reservada ***volatile***. Fuera de esas indicaciones que da el programador, el compilador puede decidir qué variables se crean en la ALU y cuáles en la memoria principal.

No se ha dicho nada en este capítulo sobre la creación de espacios de memoria con valores constantes. Ya se presentó la forma de hacerlo en el capítulo 2 sobre tipos de datos y variables en C. Una variable declarada como ***const*** quedará almacenada en el espacio de memoria de las instrucciones. No se puede modificar (mediante el operador

asignación) el valor de una variable definida como ***const***. Por eso, al crear una variable de esta forma hay que asignarle valor en su declaración.

Ejercicios

40. ***Haga un programa que calcule el máximo común divisor de dos enteros que el usuario introduzca por consola. El usuario podrá hacer tantos cálculos como quiera, e interrumpirá la búsqueda de nuevos máximos comunes divisores cuando introduzca un par de ceros.***

El programa mostrará por pantalla, cada vez que ejecute el código del bucle, un valor contador que se incrementa y que indica cuántas veces se está ejecutando a lo largo de toda la aplicación. Esa variable contador será declarada como static.

```
#include <stdio.h>
void main(void)
{
      unsigned short a, b, mcd;
      do
      {
            printf("Valor de a ... ");
            scanf("%hu",&a);
            printf("Valor de b ... ");
            scanf("%hu",&b);
            if(a == 0 && b == 0) break;
            while(b)
            {
                  static unsigned short cont = 0;
                  mcd = b;
                  b = a % b;
                  a = mcd;
                  cont++;
                  printf("\ncont = %hu", cont);      }
            printf("\n\nEl mcd es %hu.", mcd);
      }while(1);
```

```
}
```

Cada vez que se ejecuta el bloque de la estructura ***do–while*** se incrementa en uno la variable *cont*. Esta variable se inicializa a cero únicamente la primera vez que se ejecuta la sentencia ***while*** de cálculo del máximo común divisor.

Observación: quizá podría ser interesante, que al terminar de ejecutar todos los cálculos que desee el usuario, entonces se mostrara por pantalla el número de veces que se ha entrado en el bucle. Pero eso no es posible tal y como está el código, puesto que fuera del ámbito de la estructura ***while*** que controla el cálculo del máximo común divisor, la variable *cont*, sigue viva, pero estamos fuera de ámbito y el compilador no reconoce ese identificador como variable existente.