

(Versión de fecha 13 de diciembre de 2024)

1 ÁMBITO DE LAS VARIABLES

El *ámbito* de una variable es la zona de un programa en la que una variable es visible. Se suele denominar también el *alcance* de la variable o su *zona de visibilidad*

Se consideran cuatro posibles ámbitos para las variables:

- Programa
- Archivo fuente
- Función
- Bloque

En general, el lugar del código en el que se declara una variable determina su ámbito de visibilidad, pero se pueden utilizar las cláusulas *extern* y *static* para modificar el comportamiento por defecto de la declaración.

2 ÁMBITO DEL PROGRAMA: VARIABLES GLOBALES

Cualquier variable que se declare fuera de las funciones, tiene ámbito global: a partir del punto en el que se declara, es accesible por cualquier función del programa. Lo habitual y recomendable es declarar las variables globales al principio del programa, antes de cualquier declaración de función.

Observa el código del siguiente ejemplo. Se declara una variable global llamada *num* a la que se asigna el valor 125. A partir de ese momento, la variable *num* es accesible desde la función *main()* y desde cualquier otra función del programa.

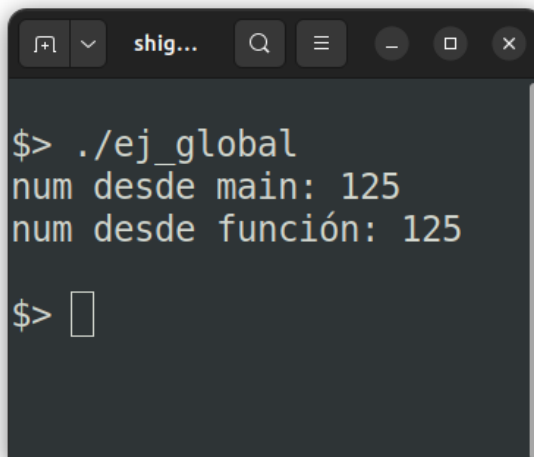
Ejemplo 1 Ejemplo de variable global

```
#include <stdio.h>

int num = 125; // variable global

void print_global() {
    printf("num desde función: %d\n", num);
}

int main() {
    printf("num desde main: %d\n", num);
    print_global();
    return 0;
}
```



```
shig...
$> ./ej_global
num desde main: 125
num desde función: 125
$> 
```

Ya hemos usado anteriormente el ámbito global, al declarar los tipos de datos personalizados con la cláusula *typedef*, por ejemplo y también al declarar constantes con *#define*.

Buenas prácticas de programación: variables globales

En general, no se considera una buena práctica de programación el uso de variables de ámbito global. El ámbito global se debe usar solo para tipos de datos personalizados y valores constantes.

3 VARIABLES LOCALES Y VARIABLES DE BLOQUE

Las variables *locales* son las que se declaran dentro del código de una función. Solo son visibles en la función en la que se han declarado.

Observa el siguiente ejemplo:

Ejemplo 2 Ejemplo de variables locales

```
#include <stdio.h>

void funcion_1() {
    int x = 10;
    printf("x= %d\n", x);
}

void main() {
    int y = 20;
    printf("y= %d\n", y);
    return 0;
}
```

En el código anterior, la variable *x* se declara dentro de *funcion_1()* y solo es accesible desde dentro de la función. Si se intentara acceder a *x* desde dentro de *main()*, se produciría un error de compilación. Lo mismo ocurre con la variable *y*: se declara dentro de *main()* y solo es accesible desde *main()*; si se intentara acceder a *y* desde *funcion_1()*, se produciría un error.

Las llaves, {}, delimitan un bloque de código. Pues bien, las variables que se declaran dentro de un bloque de código, solo son visibles dentro del bloque en el que se declaran.

Cuando un bloque está declarado dentro de otro, el interior (*bloque hijo*), puede acceder a las variables del exterior (*el bloque padre*), pero no al contrario.

El *cuerpo* de una función es el código encerrado entre las llaves de apertura y cierre de la misma. Dentro de una función, pueden existir otros bloques de código, como se muestra en el siguiente ejemplo:

Ejemplo 3 Ámbito de bloques anidados

```
#include <stdio.h>

int main() {
    int x = 10;
    {
        int y = 20;
        printf("x= %d\n", x); // 'x' es accesible desde el bloque hijo
        printf("y= %d\n", y);
    }
    printf("x= %d\n", x);
    printf("y= %d\n", y); // Error: 'y' no existe en este ámbito
}
```

Un caso habitual de bloques de código dentro del cuerpo de una función son los bucles y las bifurcaciones. El siguiente ejemplo muestra que dichos bloques también delimitan su propio ámbito, para las variables que se declaren dentro de los mismos:

Ejemplo 4 Ámbito de bloque en bucles y bifurcaciones

```
#include <stdio.h>

int main() {
    // Las variables declaradas en un bucle solo son visibles dentro del bucle
    for(int i=0; i<5; i++) {
        printf("i= %d\n", i);
    }
    printf("i= %d\n", i); // Error: ''i no existe fuera del bucle

    // Las bifurcaciones también delimitan un ámbito de bloque
    if(x>0) {
        int z = 30;
        printf("z= %d\n", z);
    }
    printf("z= %d\n", z); // Error: ''z no existe fuera del if
}
```

Una buena forma de ver las variables que son visibles en cada momento es utilizar el depurador de VSCode. Si ejecutas los ejemplos anteriores paso a paso, en la sección de variables locales verás cuáles son visibles en cada momento de la ejecución del programa.

4 PROGRAMAS FORMADOS POR VARIOS FICHEROS FUENTE

Vamos a ver primero un ejemplo para demostrar la necesidad del uso de prototipos de funciones cuando se compila. Observa el siguiente código:

```
#include <stdio.h>

int main() {
    int valores[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("El valor maximo es: %d\n", maximo(valores, 10));
}

int maximo(int array[], int num_elementos) {
    int max = array[0];
    for (int i = 1; i < num_elementos; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

Si compilas el código anterior, se producirá un error de compilación. El problema es que en *main()* estamos llamando a la función *maximo()*, que aún no está declarada.

Para poder usar una función, primero hay que declararla. Refiriéndonos al ejemplo anterior, habría dos opciones: poner el código de la función *maximo()* antes de *main()* o poner el prototipo de *maximo()* antes de *main()*, con lo que ya podremos poner su código en cualquier otro sitio.

En el ejemplo siguiente se ha optado por poner al principio del programa el prototipo de la función *maximo()* y su código a continuación de *main()*:

```
#include <stdio.h>

int maximo(int array[], int num_elementos); // Prototipo de la función

int main() {
    int valores[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("El valor maximo es: %d\n", maximo(valores, 10));
}

int maximo(int array[], int num_elementos) {
    int max = array[0];
    for (int i = 1; i < num_elementos; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

Una vez puesto el prototipo de la función antes de su primera utilización, el código compila y ejecuta perfectamente.

Cuando un programa consta de varios ficheros fuente diferentes, hay que seguir determinadas reglas para que las funciones existentes en un fichero se puedan utilizar en los otros ficheros. Vamos a reproducir el ejemplo anterior, pero utilizando dos ficheros: uno para la función *maximo()* y otro para el programa principal con la función *main()*.

El código del fichero *maximo.c* es el siguiente:

```
// Fichero maximo.c
#include <stdio.h>

int maximo(int array[], int num_elementos) {
    int max = array[0];
    for (int i = 1; i < num_elementos; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

El código del fichero *main.c* es el siguiente:

```
// Fichero main.c
#include <stdio.h>

int main() {
    int valores[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("El valor maximo es: %d\n", maximo(valores, 10));
}
```

En Windows, para compilar los dos ficheros y generar un ejecutable llamado *main.exe* habría que teclear la siguiente orden en el terminal:

```
gcc main.c maximo.c -o main.exe
```

En el caso de Mac o Linux, la instrucción para compilar sería la siguiente:

```
gcc main.c maximo.c -o main
```

En ambos casos, le estamos diciendo al compilador que utilice los dos ficheros fuente para generar el ejecutable llamado *main.exe* (o *main*, en Mac o Linux).

Como resultado, el compilador emitirá un error similar a cuando compilábamos todo el código en un solo fichero.

Para evitar el error de compilación, tenemos que declarar la función *maximo()* dentro del fichero *main.c*, antes de utilizarla, de manera similar a lo que hicimos en el caso de un único fichero:

```
// Fichero main.c
#include <stdio.h>

int maximo(int array[], int num_elementos); // Prototipo de maximo()

int main() {
    int valores[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("El valor maximo es: %d\n", maximo(valores, 10));
}
```

Ahora, si volvemos a compilar los dos ficheros, el ejecutable se generará sin que el compilador genere ningún error.

En estos casos es habitual utilizar la cláusula *extern* en el prototipo de la función, para indicar que el código está en otro fichero:

```
extern int maximo(int array[], int num_elementos);
```

No es obligatorio usar *extern*, pero mejora la legibilidad del código y, por ello, es recomendable su utilización. Piensa que, en una situación real, el fichero *main.c* tendrá otros prototipos de funciones cuyo código está en el mismo fichero. De esta manera, queda claro qué prototipos corresponden a funciones situadas en *main.c* y qué prototipos están en otros ficheros.

Quizás, una forma mejor de organizar este tipo de programas es utilizar ficheros de cabecera. Consiste en crear un fichero de cabecera *.h*, para cada fichero que aporte funciones, con los prototipos de las mismas. El fichero que quiera utilizar las funciones, tendrá que hacer un *#include* del fichero de cabecera correspondiente. La organización sería la siguiente:

```
// Fichero maximo.c
#include <stdio.h>

int maximo(int array[], int num_elementos) {
    int max = array[0];
    for (int i = 1; i < num_elementos; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

```
// Fichero maximo.h
#include <stdio.h>

int maximo(int array[], int num_elementos);
```

```
// Fichero main.c
#include <stdio.h>
#include "maximo.h"

int main() {
    int valores[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("El valor maximo es: %d\n", maximo(valores, 10));
}
```

Ahora, el prototipo de la función externa *maximo()* se pone en el fichero de cabecera *maximo.h* y, en el fichero *main.c*, se utiliza una cláusula *#include* para introducir el prototipo. La orden de compilación es la misma que se ha utilizado anteriormente y el ejecutable que se obtiene es también el mismo.

En el fichero de cabecera se puede utilizar o no la cláusula *extern* en los prototipos, aunque no tiene ningún efecto práctico y en los ficheros de cabecera se da por hecho que el código de las funciones que se declaran está en el fichero del mismo nombre y extensión *.c*.

Ruta de los ficheros en las cláusulas *#include*

Observa que, en las cláusulas *#include* del ejemplo anterior, se utiliza una forma diferente de indicar la librería *stdio.h*, que va entre corchetes, que la librería *maximo.h*, que se ha escrito entre comillas.

Cuando el nombre del fichero de cabecera se pone entre corchetes *< >*, el compilador buscará el fichero en los directorios que tiene asignados para librerías del sistema^a.

Si el fichero de cabecera se encierra entre comillas, el compilador lo buscará en la ruta que se especifique entre las comillas. En este ejemplo, que solo pone el nombre del fichero *maximo.h*, lo buscará en el directorio actual.

^aSi has instalado como compilador el paquete MSYS64 en Windows, los ficheros de cabecera del sistema, como el fichero *stdio.h*, los puedes encontrar en el directorio *C:\msys64\ucrt64\include*.

5 Más SOBRE LA CLÁUSULA *EXTERN*

Cuando un programa utiliza varios ficheros fuente diferentes, la cláusula *extern* le indica al compilador que una variable o una función se ha declarado en otro archivo.

El programa del siguiente ejemplo consta de dos ficheros fuente: *fichero_1.c* y *fichero_2.c*. En *fichero_1.c* se declara una variable global *PI*. Además, se declara que la función *print_global()* es *externa*, esto es, que su definición está en otro fichero distinto; dentro de *main()*, se utiliza la función *print_global()*. Por su parte, *fichero_2.c* declara que la variable *PI* es externa y la utiliza dentro de su función *print_global()*.

Ejemplo 5 Ejemplo de uso de la cláusula extern

```
// fichero_1.c
#include <stdio.h>

double PI = 3.1416;

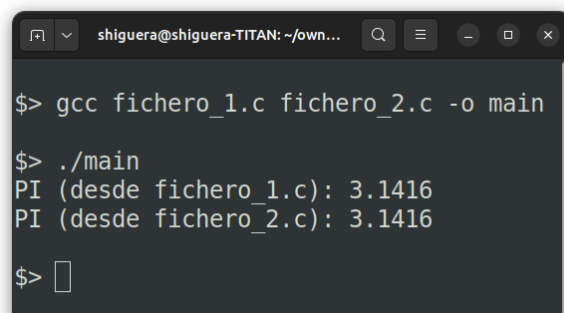
extern void print_global();

int main() {
    printf("PI (desde fichero_1.c): %.4f\n", PI);
    print_global(); // Usando la función externa
    return 0;
}

-----
// fichero_2.c
#include <stdio.h>

extern double PI;

void print_global() {
    printf("PI (desde fichero_2.c): %.4f\n", PI);
}
```



```
shiguera@shiguera-TITAN: ~/own...
$> gcc fichero_1.c fichero_2.c -o main
$> ./main
PI (desde fichero_1.c): 3.1416
PI (desde fichero_2.c): 3.1416
$> 
```

En la figura de la parte derecha del ejemplo anterior puedes ver la instrucción que se ha utilizado para compilar el programa en un ordenador con Linux: se compilan *fichero_1.c* y *fichero_2.c* y se genera el ejecutable llamado *main*:

```
gcc fichero_1.c fichero_2.c -o main
```

En un ordenador con Windows, la instrucción para compilar los ficheros anteriores sería la siguiente:

```
gcc fichero_1.c fichero_2.c -o main.exe
```

6 LA CLÁUSULA *STATIC*

La cláusula *static* se comporta de diferente manera según en qué contexto se utilice.

Primer uso de *static*:

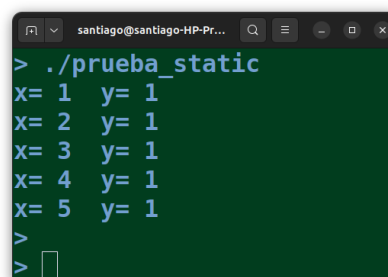
En general, las variables locales de una función se destruyen al acabar la ejecución. En cada llamada a la función, las variables locales se vuelven a inicializar. Esto es siempre así, salvo que la variable local se declare *static*, en cuyo caso solo se inicializa la primera vez que se llama a la función y, en sucesivas llamadas, el valor que tuviera se guarda hasta la siguiente ejecución. Observa el siguiente ejemplo:

Ejemplo 6 Ejemplo de variable local *static*

```
#include <stdio.h>

void prueba_static() {
    static int x = 0;
    int y = 0;
    x++;
    y++;
    printf("x= %d y= %d\n", x, y);
}

int main() {
    for (int i = 0; i < 5; i++) {
        prueba_static();
    }
    return 0;
}
```



```
santiago@santiago-HP-Pr...
> ./prueba_static
x= 1 y= 1
x= 2 y= 1
x= 3 y= 1
x= 4 y= 1
x= 5 y= 1
>
> 
```

En el ejemplo anterior, la función *prueba_static()* declara dos variables: *x* e *y*. La variable *x* es *static* y, por tanto, su valor se conserva en sucesivas llamadas a la función. En cambio, la variable *y* es una variable local normal y se inicializa cada vez que se entra en la función:

Segundo uso de *static*:

Cuando la cláusula *static* se aplica a variables globales o a funciones, garantiza que dicho elemento (variable o función) solo exista en la unidad de compilación en la que se encuentre declarado.

De alguna forma, esta forma de actuar de la cláusula *static* sería similar a la cláusula *private* de la que disponen otros lenguajes.