Fundamentos de Programación – Programación Estructurada en C:
6.- FUNCIONES

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

6.- FUNCIONES

6.- FUNCIONES

Una función es un bloque de código discreto, independiente del Programa Principal, que puede ser llamado enviándole (o no) unos datos, para que realice una determinada tarea y/o proporcione unos resultados.

El correcto diseño de un programa, requiere **modularidad**, es decir, dividir la tarea general que ha de hacer el Programa Principal, en tareas (módulos) más sencillas y concretas. Cada una de esas tareas sería realizada por una función, o, dicho de otra forma, cada módulo que compone el programa total, se implementa mediante una función.

Así, un bloque de sentencias que se ejecute en varias partes en el Programa Principal, se escribe una sola vez en el fichero fuente, dentro de una función, y lo único que habría que hacer en el Programa Principal sería llamar a esa función cada vez que se necesite.

La modularidad facilita el desarrollo y mantenimiento de los programas, y minimiza la cantidad de errores posibles, el uso de memoria y el trabajo innecesario.

Las librerías de funciones son conjuntos de funciones compiladas, que se guardan en un fichero bajo un determinado nombre, listas para ser utilizadas desde cualquier programa compatible con el lenguaje de las mismas. En C, tal y como vimos en **2.1.- Comando #include**, para poder usar funciones de una librería determinada, hay que incluir dicha librería en el proceso de Enlazamiento, para lo cual, hay que indicar su inclusión mediante la instrucción al precompilador *#include*.

6.1.- DECLARACIÓN, DEFINICIÓN Y LLAMADA DE FUNCIONES

6.1.1.- Declaración de una función

Tal y como indicamos en el punto **1.1.2.- Declaraciones**, todos los elementos nuevos que se vayan a usar en un programa en C, hay que declararlos antes de usarlos. Hasta ahora, los únicos elementos nuevos que aparecían en un programa, eran las variables. Tal y como todas las variables de un programa han de ser declaradas antes de ser usadas, **las funciones nuevas que desarrolle el programador en un fichero, han de ser declaradas antes de ser utilizadas**.

La declaración de una función se hace mediante el **prototipo**. La sintaxis general del prototipo de una función es la siguiente:

<tipo> <nombre>**(**[<lista de tipos de argumentos>]**);**

<tipo>: Indica el tipo de dato del valor que devuelve la función al terminar su ejecución. Sirve para que el compilador sepa, cada vez que se encuentra con una llamada a una función, con qué tipo de dato se va a encontrar el procesador cuando vuelva al terminar la ejecución de la función. Hay funciones que no devuelven ningún valor, en cuyo caso, se pone la palabra *void* en lugar del tipo.

<nombre>: Todas las funciones, como las variables, han de tener un nombre que las identifique de forma única en el programa; es decir, no puede haber dos funciones que tengan el mismo nombre (ni variables que tengan el mismo nombre que funciones). Los nombres de las funciones han de ser identificadores válidos en C, por lo que han de cumplir las 3 normas indicadas en **1.1.2.- Declaraciones**.

<lista de tipos de argumentos>: Las funciones, para poder ejecutarse, pueden necesitar recibir una serie de valores a la hora de ser llamadas. Esos valores recibidos, se conocen como **argumentos** o **parámetros** de la función. En la declaración de la función, se indica, mediante una lista de tipos de datos separados por comas (,), el **número** de argumentos que recibirá la función en la llamada, y el **tipo** de dato que le corresponderá a cada uno. Es opcional, ya que puede haber funciones que no necesiten recibir ningún dato en la llamada.

Como hemos dicho antes, las declaraciones de las funciones han de hacerse antes de que éstas sean usadas. Para evitar errores por olvido de declaración, lo que se suele hacer es **declarar todas las funciones** que se van a utilizar en un programa **al comienzo del fichero fuente** (.c), en la 1ª Sección o Sección de Descripciones, **antes de** la 2ª Sección, que es la Sección del Programa Principal, que comienza con la línea ***int main()***. La función *main* es la función que primero se **define** en todo programa, y la única que no hace falta declarar.

6.1.2.- Definición de una función

La **definición** de una función es la **redacción del código** necesario para que la función realice la(s) tarea(s) para la(s) que fue concebida.

Sintaxis general de la definición de una función:

<tipo> <nombre>**(**[<lista de declaraciones de argumentos>]**)**	ENCABEZAMIENTO
{ [<declaraciones de variables>] <resto de código ejecutable> **return**[**(**<expresión>**)**]**;** **}**	CUERPO

Tal y como se ve en la sintaxis, el ENCABEZAMIENTO de la definición de una función es muy parecido al prototipo de la función. Hay 2 diferencias:

- Entre los paréntesis del prototipo, basta con poner los tipos de los argumentos que recibirá la función en la llamada, y en el orden en el que se pasarán en la llamada. En el ENCABEZAMIENTO sin embargo, se escribe una **declaración** completa de cada uno de los argumentos (es al leer esta línea cuando el procesador hace la reserva de memoria para cada uno de ellos). Esos argumentos son **argumentos formales**, y sólo pueden ser usados dentro del CUERPO de la función, ya que cuando se acabe la ejecución de la misma, se destruyen (los bytes de memoria ocupados por las mismas pasan a ser considerados como *no reservados*. En cualquier caso, para que no haya errores de compilación, la lista de **tipos** de datos **del prototipo** y los **tipos** de los **argumentos** declarados en el **encabezamiento**, tienen que coincidir en orden y número.
- El **prototipo** de una función termina con **;**, sin embargo, el **encabezamiento** de la definición **no**.

El CUERPO de la definición de una función es un bloque de sentencias, se escribe entre llaves. Dentro del mismo, se encuentran las siguientes partes:

<declaraciones de variables>: Las funciones pueden hacer uso de variables propias, que han de ser declaradas, como todas las variables, antes de ser usadas. Esas variables se llaman **variables locales**. Su uso se producirá en la parte de <código ejecutable> de la función, por eso han de declararse al inicio de la misma.

Las variables locales permanecen ocultas al resto del programa, lo que quiere decir que sólo pueden ser usadas en sentencias que estén en la parte de <resto de código ejecutable> de la función en la que han sido declaradas. **Se crean** (se hace la reserva memoria) cada vez que se llame a la función (cada vez que se ejecuta la sentencia de declaración de las mismas), **y se destruyen** (se libera la memoria ocupada) cada vez que se termina la ejecución de la función (como los argumentos formales).

Las funciones también pueden acceder a **variables globales**, que son **aquellas declaradas fuera de toda función** (incluso fuera del *main*, que es la función principal; las variables declaradas en el *main* son variables locales del *main*, y no variables globales...). Sin embargo, aunque sea posible usarlas, suelen dar muchos problemas, por lo que hay que evitar su uso; así, en esta asignatura se considera como erróneo el uso de variables globales. En lenguajes de programación más avanzados que C (en Java, por ejemplo), no existen las variables globales.

Como ya hemos visto, una función no tiene por qué tener variables locales. Puede haber funciones que sólo usen los argumentos formales, o ni siquiera eso.

<resto de código ejecutable>: Conjunto de sentencias con las que se implementa la tarea para la que la función fue pensada. En estas sentencias se puede acceder a las variables locales (declaradas justo en la anterior parte del CUERPO) y a los argumentos declarados en el ENCABEZAMIENTO.

De hecho, los argumentos también son variables locales de la función, ya que no pueden ser accedidas desde fuera de la misma. La diferencia con el resto de variables locales es que los argumentos se declaran en el ENCABEZAMIENTO, y no dentro del CUERPO, y que se inicializan con el valor recibido en la llamada a la función.

return[<expresión>]**;**: Es la última sentencia que debe ejecutarse antes de salir de la función. Devuelve el control a la parte del programa que realizó la llamada a la función. Aunque escribir más de una sentencia **return**[<expresión>]**;** no dé ningún problema de compilación (no es un error sintáctico), ha de evitarse el hacerlo, ya que suele inducir a errores (cuando tenemos un programa con miles de líneas de código, es más difícil entender su funcionamiento si la salida de las funciones se puede dar desde más de una sentencia).

Si el <tipo> de la función es *void*, significa que no devuelve ningún valor, por lo que la sentencia será **return;** (o también está admitido no poner nada). Sin embargo, cuando la función devuelve un valor, la sentencia será

return <expresión>**;**

donde lo primero que hace el procesador es evaluar la expresión, convertir, si fuera necesario, el valor obtenido al tipo de dato que la función ha de devolver, y volver con ese valor a la sentencia desde la que se hizo la llamada a la función.

Ejemplo:

```
return a;

return a+b;
```

El tipo de dato resultante de la evaluación de la expresión, deberá coincidir con el tipo de la función, o ser, por lo menos, convertible implícitamente en ella, tal y como ocurría al dar un valor a una variable mediante el Operador Asignación (1.3.1.1. Variables). **Si la conversión no pudiese hacerse** de forma implícita, se produciría un **aviso de compilación** (*warning*), que **se arregla** indicando la conversión de forma explícita (**casting**).

6.1.3.- Llamada a una función

La llamada a una función se hace, incluyendo en una expresión o formando una sentencia (línea terminada en ;), el nombre de la función y la lista encerrada entre paréntesis de los argumentos necesarios separados por comas. Los argumentos indicados en una llamada, son **argumentos actuales**, y pueden ser variables, constantes o expresiones.

Ejemplo:

```
funcion_1(a,'b',c+d);
```

Cuando el procesador se encuentra con la llamada a una función, **lo 1º** que necesita hacer es guardar de alguna forma, **dónde** se ha producido la llamada a la función (en qué instrucción del programa) **y en qué ámbito** (qué variables son visibles en ese punto del programa). Eso hay que hacerlo para, después de ejecutar las instrucciones indicadas en la definición de la función, volver al punto de llamada y poder seguir ejecutando las siguientes instrucciones.

Lo siguiente, será **obtener los valores de los argumentos actuales**. La obtención de valor de los argumentos actuales que son variables o constantes es inmediata, pero en el caso de expresiones, hay que evaluarlas.

Una vez que se tienen los valores de todos los argumentos actuales, la ejecución se pasa al ENCABEZAMIENTO de la función. Eso quiere decir, que el procesador busca en el fichero ejecutable dónde está la definición de la función, y pasa a ejecutar lo indicado en su 1ª línea. En el ENCABEZAMIENTO de todas las funciones, lo que se indica que se ejecute, son las declaraciones de los argumentos formales, es decir, **se hace la reserva de memoria necesaria para cada uno de los argumentos** indicados entre paréntesis (según su tipo de dato), **y una vez hecha esa reserva de memoria, se inicializan los valores de cada uno con los valores de los argumentos actuales** recibidos en la llamada, según corresponda. Por eso es por lo que no se puede inicializar el valor de un argumento formal en el ENCABEZAMIENTO de la función, es en la llamada donde se les da valor con los argumentos actuales.

Al llegar la ejecución de un programa al ENCABEZAMIENTO de una función, se crea un nuevo ámbito, en el que sólo serán accesibles las variables concretas de esa función (las creadas mediante los argumentos formales y las locales internas a la función). Ese ámbito, sólo será válido mientras se ejecute la función, y se destruirá al ejecutar la sentencia *return*. El procesador siempre ha de saber en qué ámbito se encuentra en cada momento, por eso, cuando se crea uno nuevo, ha de guardar, de alguna forma, el ámbito anterior. Así, al ejecutar la sentencia *return* de un ámbito, se destruye el ámbito actual, y se vuelve, al punto del programa desde el que se llamó a la función, con el ámbito correspondiente en el momento de la llamada.

La llamada a una función puede aparecer en cualquier sentencia, tanto dentro del Programa Principal (función *main*) como en cualquier otra función, es decir, **las funciones pueden llamarse entre sí**. La única condición, es que antes de escribir una llamada a una función, ha de haberse escrito su declaración. Por eso, para evitar errores por olvido, se ponen todas las declaraciones de todas las funciones de un fichero fuente, antes del *main* (que será la primera función que pueda hacer uso de alguna otra función). De hecho, las librerías que se importan al fichero mediante los *#include*, contienen declaraciones de funciones ya compiladas en una librería, pero que hay que declararlas como las demás.

Incluso, **una función puede llamarse a sí misma**. Se dice entonces que **es recursiva**. La recursividad es la capacidad que tiene un elemento de un programa, para definir algo en términos de sí mismo. Para usar esta propiedad de forma adecuada, hay que poner algún límite que evite un bucle infinito del cual el programa no pueda salir.

Ejemplo:

```
#include <stdio.h>

double factorial(int);

int main()
{
    double valor;
    //Sit. 1
    valor=factorial(3);

    printf("El valor del factorial de 4 es : %.0lf",valor);

    return 0;
}

double factorial(int n)
{
    double resultado;
    //Sit. 2: función llamada con n=3
    //Sit. 3: función llamada con n=2
    //Sit. 4: función llamada con n=1

    if(n==1)
        resultado=1; //Sit. 5
    else
        resultado=factorial(n-1)*n;

    return(resultado);
}
```

/* Lo que ocurre en memoria, es lo siguiente:

Sit. 1:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	valor	1000 ... 1007	-	8 bytes

Sit. 2:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	valor	1000 ... 1007	-	8 bytes
factorial(3)	n	2000 ... 2003	3	4 bytes
	resultado	2010 ... 2017	-	8 bytes

El ámbito main aparece ensombrecido porque aunque exista, no es accesible en esta situación.

Sit. 3:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	valor	1000 ... 1007	-	8 bytes
factorial(3)	n	2000 ... 2003	3	4 bytes
	resultado	2010 ... 2017	-	8 bytes
factorial(2)	n	3000 ... 3003	2	4 bytes
	resultado	3010 ... 3017	-	8 bytes

Sit. 4:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	valor	1000 ... 1007	-	8 bytes
factorial(3)	n	2000 ... 2003	3	4 bytes
	resultado	2010 ... 2017	-	8 bytes
factorial(2)	n	3000 ... 3003	2	4 bytes
	resultado	3010 ... 3017	-	8 bytes
factorial(1)	n	4000 ... 4003	1	4 bytes
	resultado	4010 ... 4017	-	8 bytes

Sit. 5:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	valor	1000 ... 1007	-	8 bytes
factorial(3)	n	2000 ... 2003	3	4 bytes
	resultado	2010 ... 2017	-	8 bytes
factorial(2)	n	3000 ... 3003	2	4 bytes
	resultado	3010 ... 3017	-	8 bytes
factorial(1)	n	4000 ... 4003	1	4 bytes
	resultado	4010 ... 4017	1	8 bytes

*/

6.2.- PASO DE VARIABLES A UNA FUNCIÓN

Cuando se llama a una función, se evalúan los argumentos actuales y los valores obtenidos se usan para inicializar los argumentos formales de la función, que se destruirán al terminar la ejecución de la misma. Así, hay que tener en cuenta que los cambios que sufran los valores de los argumentos formales no serán accesibles desde fuera de la función, ya que los argumentos formales sólo existen dentro de la función, en el ámbito propio de esa función, que será destruido cuando se acabe la ejecución para la que se creó.

En cada punto del programa, sólo un ámbito es válido, activo; el ámbito actual. Eso no significa que el resto de ámbitos creados a lo largo de la ejecución hayan desaparecido de memoria. Sólo habrán desaparecido los ámbitos de funciones cuya ejecución se haya terminado (se haya llegado a ejecutar *return*); el resto, seguirán existiendo en memoria, pero como ámbitos pasivos. Así, sólo las variables del ámbito actual son válidas en cada punto del programa; sin embargo, todas las direcciones de la memoria usada por el programa, son accesibles desde cualquier ámbito, incluso direcciones de memoria reservadas para variables de ámbitos pasivos.

Según lo explicado hasta ahora, una función puede realizar todos los cálculos que sean necesarios pero sólo puede devolver un único valor (del tipo indicado en el prototipo y encabezamiento). Así, si quisiéramos usar una función para obtener varios valores, está claro que todos esos valores no se podrían recuperar después de la ejecución de la llamada mediante *return*; (sólo se podría obtener un valor). El resto, se obtendrán mediante el uso de direcciones del ámbito anterior.

6.2.1.- Paso de variables por valor

Una variable ha sido pasada **por valor** a una función, cuando el valor con el que se inicializa el argumento formal correspondiente, es **el valor de la variable**. Así, los cálculos que se hagan dentro de la función con ese valor **no afectan al valor de la variable fuera de la función** (dentro de la función, la variable original es de un ámbito pasivo). Al fin y al cabo, la variable corresponderá al ámbito anterior al correspondiente a la ejecución de la función, por lo que no será accesible desde la misma. Dentro de la función sólo se trabaja con una variable **nueva** que ha recibido el valor de la variable original al inicio. Las 2 variables incluso pueden llamarse de igual forma, pero al pertenecer a distintos ámbitos, no hay problemas en compilación ni ejecución; sólo son dos zonas de memoria diferentes, identificadas con el mismo nombre. Se ha visto claramente en el ejemplo de funciones recursivas.

Ejemplo:

```
#include <stdio.h>

void aumentar(int);

int main()
{
    int var=0;

    printf("Valor: %d",var); //Sit. 1.
    aumentar(var);
    printf("Valor: %d",var); //Sit. 4.

    return 0;
}

void aumentar(int var)
{ //Sit. 2
    var=var+10;
    printf("Valor en la función: %d",var); //Sit. 3.
    return();
}

/*
```

Sit. 1:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes

Sit. 2:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes
aumentar	var	2000 ... 2003	0	4 bytes

Sit. 3:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes
aumentar	var	2000 ... 2003	10	4 bytes

Sit. 4:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes

*/

6.2.2.- Paso de variables por referencia

Una variable ha sido pasada **por referencia** a una función, cuando el valor con el que se inicializa el argumento formal correspondiente, es **el valor de la dirección de la variable**. De esta forma, el valor que se recibe en el argumento formal de la función es la dirección de la variable; los cambios hechos a ese valor (a la dirección) no se verán desde fuera de la función, pero **a través de una dirección se puede acceder al contenido**, y así sí que se puede cambiar el valor contenido en la variable.

De esta forma, una dirección correspondiente a una variable del ámbito anterior, que en la ejecución de la función es un ámbito pasivo, llega hasta la función. C proporciona unas variables especiales que, mediante una dirección válida, pueden acceder al valor contenido en la misma (y pueden incluso cambiarlo). Esas variables especiales, se conocen como **punteros**. Son variables como las vistas hasta ahora, porque al fin y al cabo son zonas de memoria cuyo valor cambia durante la ejecución del programa (no son constantes), pero en vez de servir para guardar valores de datos, se usan para almacenar direcciones de comienzo de otras variables.

Así, los argumentos formales que recogen direcciones de variables (o dicho de otra forma, que recogen variables pasadas por referencia), serán **punteros**.

El tamaño en bytes de una variable puntero depende del sistema. Para los ejemplos de esta asignatura, consideraremos que se usan 4 bytes. Cuando se dice que un puntero apunta a una dirección, lo único que se dice es que en los 4 bytes que ocupa el puntero, está representada esa dirección.

La sintaxis de declaración de un puntero es:

<tipo> *<nombre>**;**

El * antes del nombre en la declaración, indica que ese nombre se usará para identificar a un puntero. En la declaración, también se indica el tipo de dato que se va a contener a partir de la dirección de memoria a la que apunta el puntero. Así, el tipo indicado sirve para que el procesador sepa cuántos bytes ha de tener en cuenta a partir de la dirección apuntada, para acceder al valor del

dato; la declaración de un puntero siempre supone la reserva de 4 bytes seguidos en memoria, pero el procesador tendrá que tener en cuenta que los valores de direcciones almacenadas en ese puntero, tendrán que ser direcciones de zonas de memoria reservadas para variables de cierto tipo. Es decir, el mismo puntero, no se puede usar para almacenar direcciones de memoria de variables de diferentes tipos.

Para que una variable se pase por referencia, y se pueda cambiar su valor dentro de una función, hacen falta **dos** cosas:

- Que como **argumento actual** en la llamada a la función, se pase la **dirección** de la variable. La dirección de una variable se indica anteponiendo "***&***" a su nombre. Si la dirección de la variable se tuviera ya en un puntero, como argumento actual se indicaría sólo el nombre del puntero.

- **Que** la dirección de la variable **se recoja en un puntero**, **y** que a través de ese puntero, **se acceda y se cambie el valor** contenido en la dirección. Para acceder al contenido de un puntero en cualquier sentencia (menos en la declaración del mismo), se antepone '*******' al nombre del puntero.

Así, el * en C significa tres cosas:

- Multiplicación. Será un Operador Binario de orden de precedencia 3.

- En la declaración de un puntero, significa que el nombre que viene a continuación se usará en el resto de la función como identificador de un puntero, de una dirección de memoria. Será Operador Unario.

- Delante del nombre de un puntero, en cualquier sentencia menos en la declaración, significa que la expresión *<puntero> representa el valor contenido a partir de la zona de memoria cuya dirección de comienzo es la contenida como valor en el puntero, siendo esa zona de memoria tan grande como lo requiera el tipo de dato indicado en la declaración del puntero. Será Operador Unario.

Ejemplo:

```
#include <stdio.h>

void aumentar(int *);

int main()
{
    int var=0;

    printf("Dirección: %u",&var);
    printf("Valor: %d",var); //Sit. 1.
    aumentar(&var);
    printf("Valor: %d",var); //Sit. 4.

    return 0;
}

void aumentar(int *var)
{
    //Sit. 2.
    *var=(*var)+10;
    printf("Valor del puntero: %u",var); //Sit. 3.
    printf("Valor en la función: %d",(*var));
}

/*
```

Sit. 1:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes

Sit. 2:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	0	4 bytes
aumentar	* var	2000 ... 2003	1000	4 bytes

Sit. 3:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	10	4 bytes
aumentar	* var	2000 ... 2003	1000	4 bytes

Sit. 4:

Ámbito	Variable	Dirección	Memoria	Tamaño
main	var	1000 ... 1003	10	4 bytes

*/

Las funciones pueden devolver cualquier valor mediante la instrucción ***return***, incluso valores de direcciones, es decir, pueden devolver punteros, pero eso sólo se puede hacer de forma correcta en casos concretos, que ya se verán más adelante.

6.3.- DIAGRAMAS DE FLUJO

El algoritmo de primer nivel que se haya diseñado, se codificará en el fichero fuente como la función *main*. El resto de algoritmos, serán funciones inventadas por el programador, codificándose cada una en la función correspondiente.

Los datos de entrada recibidos por un algoritmo, serán argumentos formales en las funciones. Si los datos de entrada cambian a lo largo del algoritmo, y se quiere que ese cambio se vea después de terminar el algoritmo, esos datos también serán de salida. Todas las variables que siendo de entrada también lo sean de salida (datos de entrada/salida), tendrán que ser pasadas por referencia en la función correspondiente en C.

Los datos que sólo son de entrada, podrán pasarse sólo por valor a la función.

Cuando haya más de un dato de salida, sólo uno podrá devolverse en la función con *return*. El resto, habrá que hacerlo pasando las variables correspondientes por referencia (aunque en el algoritmo correspondiente no aparezcan como datos de entrada). Esto significa que la cantidad de argumentos formales de la función no tiene por qué coincidir con la de datos de entrada indicados en el algoritmo (puede ser necesario pasar variables por referencia por el hecho de que *return* no puede devolverlas, pero no porque en el algoritmo se hayan considerado datos de entrada).

6.4.- CRITERIOS EN LA CREACIÓN Y REDACCIÓN DE FUNCIONES

En nuestra asignatura, a la hora de escribir funciones, habrá que tener en cuenta los siguientes puntos:

- Una función nunca debe ser mayor que una pantalla, o la cara de una hoja; si excediera, hay que dividirla en funciones más pequeñas.
- La salida de un bloque ha de ser NO TRAUMÁTICA. Eso quiere decir que no se pueden usar instrucciones *goto*, *continue*, *exit* o *break* (esta última se puede usar sólo en los *case* de los *switch*).
- Usar nombres descriptivos para las variables, funciones y constantes. Se puede usar la llamada Notación Húngara, según la cual, los nombres de variables, funciones y constantes empiezan por una letra minúscula que indica su tipo.