Tema 4: Subprogramación

Objetivos

- Comprender el concepto de subprograma para escribir programas y facilitar así las tareas de diseño descendente, depuración...
- Saber diseñar, codificar e invocar una función para que haga una determinada tarea.
- Entender y saber realizar el seguimiento de subprogramas recursivos.

Índice

- **Funciones**
- Declaración de funciones. Prototipo
- Definición de funciones
- Llamada a la función
- Parámetros
- Recursividad

Funciones

- **Definición y finalidad**:
  - Una **función** es un **conjunto de sentencias** a las que se les asocia un nombre (identificador) y que *pueden* generan un valor nuevo, calculado a partir de los datos que *puede* recibir (argumentos).
  - Una función en C es un **segmento independiente de código**, diseñado para **realizar una tarea específica**.
  - Las funciones **facilitan el desarrollo y mantenimiento** de los programas, evitan errores y ahorran memoria y trabajo innecesario.
  - Mediante el uso de funciones se consigue un **código más limpio y claro.**

Utilidades de los Subprogramas

- Descomponer el programa en partes:
  - Facilita el diseño descendente y la modularidad
  - Mejora la depuración y mantenimiento
  - Las partes son reutilizables
- Programa más legible
- Reducir el tamaño del programa

Funciones

```
#include <stdio.h>
int cuadrado (int y);

void main ()
{
    int x;

    for (x=1; x<=10; x++)
        printf ("%d  ", cuadrado (x) );

}
```

Prototipo (declaración) de la función

Llamada a la función

```
int cuadrado (int y)
{
    return  y * y;
}
```

Definición de la función

Funciones

- **Elementos de la función:**
  - **Identificador**: es el nombre que sirve para invocar o llamar a la función.
  - **Parámetros** es el conjunto de datos que se le pueden facilitar a la función para que realice su tarea.
  - **Cuerpo** o conjunto de sentencias. Las que realizan la tarea para la que ha sido definida la función.
  - **Valor de retorno.** Sólo para las funciones que generan algún valor.

Ejemplo

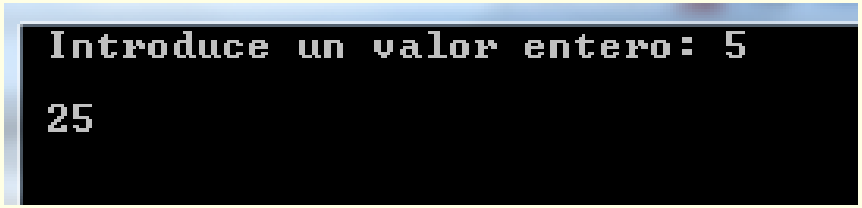
```
#include <stdio.h>
int cuadrado (int y);

void main ()
{
    int   x;
      printf (" Introduce un valor entero: ");
      scanf("%d", &x);
      printf ("\nEl cuadrado de %d es %d", x, cuadrado(x));
}

int cuadrado (int y)
{
   return  y * y;
}
```

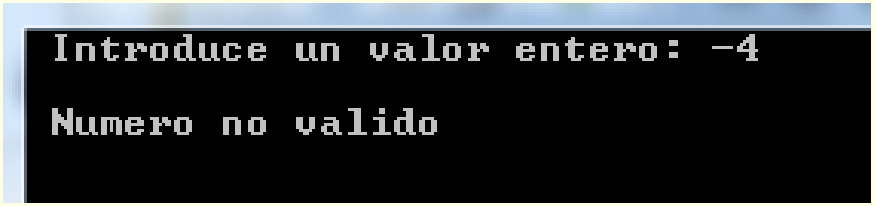
```
Introduce un valor entero: 3

El cuadrado de 3 es 9
```

Obsérvese que la variable x sigue teniendo su valor inicial 3

Ejemplo

```
#include <stdio.h>
int cuadrado (int y);
int positivo (int y);
void main ()
{
    int   x;
    printf (" Introduce un valor entero: ");
    scanf("%d", &x);
    if (positivo(x))
       printf ("%d ", cuadrado(x));
    else printf ("Numero no valido");
}
int cuadrado (int y)
{
    y = y * y;
    return  y;
}
int positivo (int y)
{
    return y > 0;
}
```





Funciones en C

- La primera función que aparece en todo programa C es la **función principal**, o **main**. La función main es la única función que no puede ser utilizada por ninguna otra.
- En un programa se pueden encontrar además de la función main, **funciones creadas por el programador** para esa aplicación, o **funciones ya creadas** e implementadas y compiladas en librerías pertenecientes al estándar de ANSI C o de creación propia.
- Todas las variables que se definen en una función son **variables locales**, es decir, se conocen sólo en la función en la que se definen.

Índice

- Funciones
- **Declaración de funciones. Prototipo**
- Definición de funciones
- Llamada a la función
- Parámetros
- Recursividad

Declaración de funciones: Prototipo

- La declaración se realiza a través de su **prototipo**. Sintaxis de la declaración (prototipo):

tipo-función nombre-función ([tipo1 [var1]],... ,[tipoN [varN]]);

tipo-función: declara de qué tipo es el **valor que devolverá** la función (puede ser cualquier tipo, si no devuelve ningún valor entonces es de tipo **void**).

nombre _ función: cualquier **identificador** que indique lo que hace la función.

tipo1,... tipoN: declara de qué **tipo** es cada uno de los valores que la función recibirá como **parámetros** al ser invocada. En la declaración del prototipo es opcional escribir el nombre de las variables.

Ejm: `int cuadrado (int y);` /* prototipo (declaración) de la función */

`int cuadrado (int);`

Declaración de funciones: Prototipo

El **prototipo** de una función presenta el modo en que esa función debe ser empleada. Qué **valores**, de qué **tipo** y en qué **orden** debe recibir la función los argumentos al ser invocada.

- Todas las **declaraciones** de función deben **preceder** a la definición del cuerpo de la función main.
- Los **paréntesis** son **obligatorios**, aunque no tenga parámetros.
- La **única función que no requiere prototipo es main(),** puesto que es la primera función que se ejecuta cuando comienza el programa.
- **No** se puede declarar, ni definir, **una función dentro de otra función.**

Funciones. Ejemplos

Realizar la **declaración** de una función **esLetra**, en lenguaje C, que reciba un carácter y devuelva un valor entero lógico indicando si el carácter es una letra o no lo es. Es decir devolverá 1 ó 0

```
Declaración:    int esLetra (char car);
```

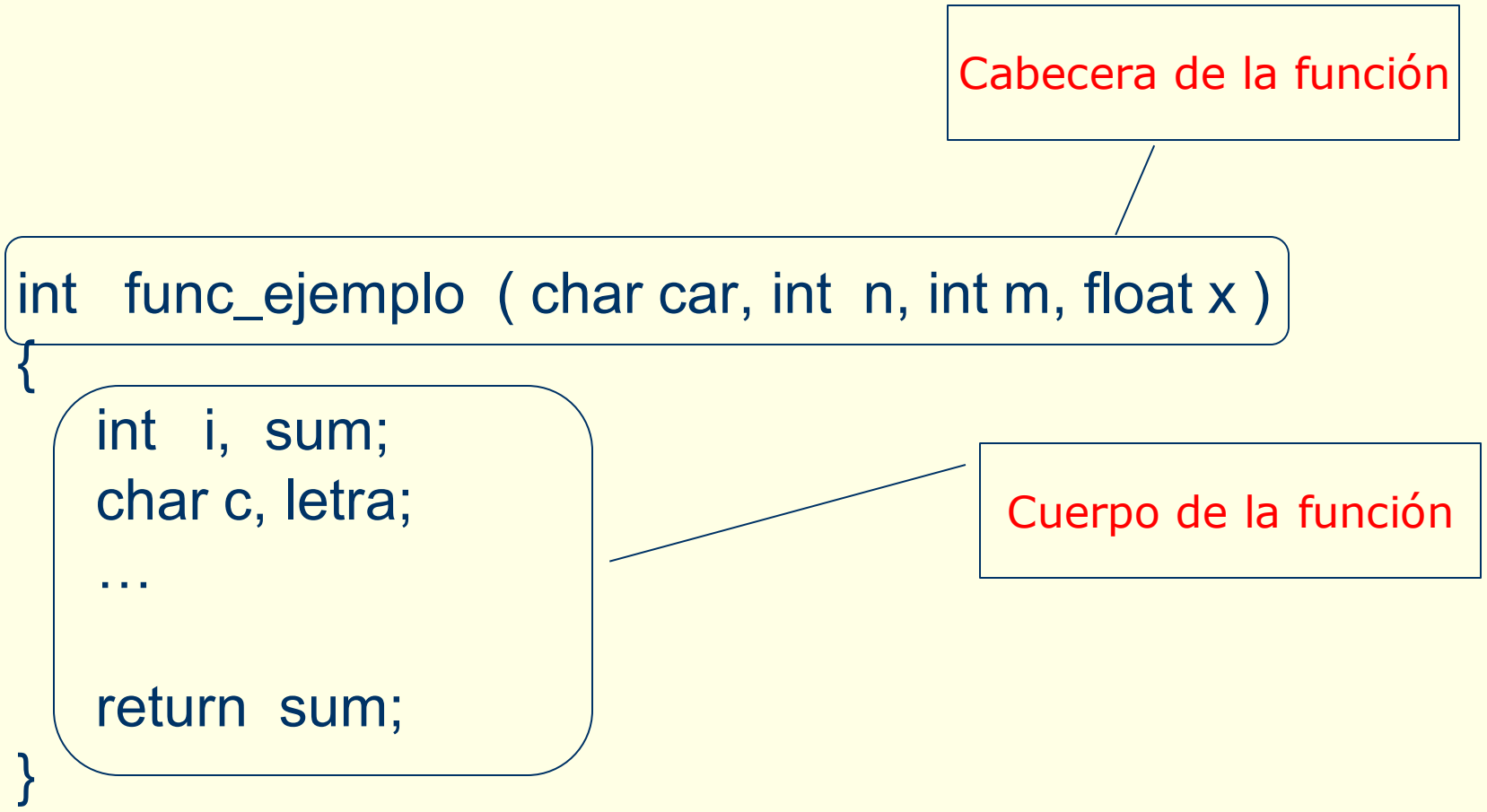
Realizar la **declaración** de una función **esLetra**, en lenguaje C, que reciba un carácter y escriba un mensaje en pantalla indicando si el carácter es una letra o si el carácter no es una letra.

```
Declaración:    void esLetra (char car);
```

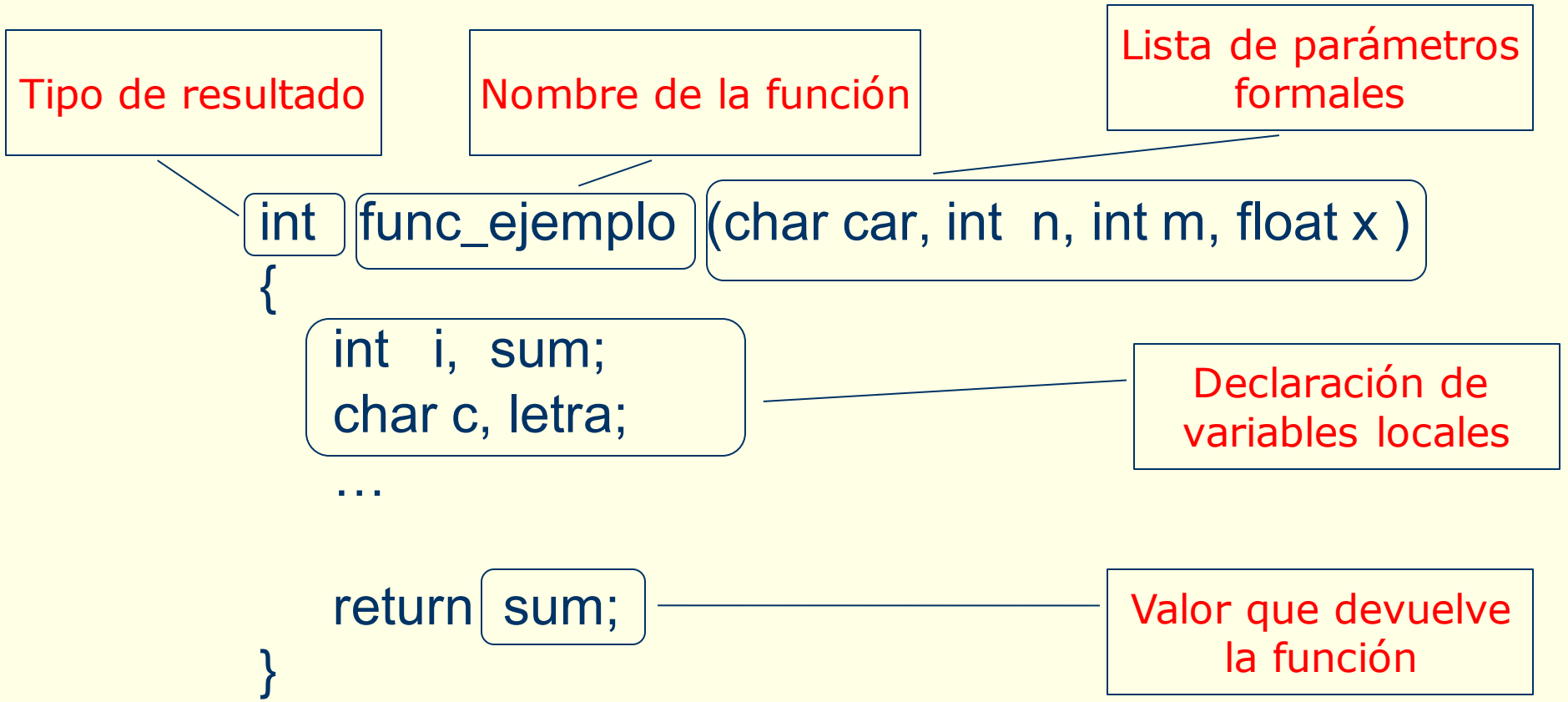
Índice

- Funciones
- Declaración de funciones. Prototipo
- **Definición de funciones**
- Llamada a la función
- Parámetros
- Recursividad

Definición de funciones



Definición de funciones



Definición de funciones

- **El formato de definición de una función es:**

```
tipo_función  nombre_función ([tipo1 var1],...,[tipoN varN])
   {
        [declaraciones de variables locales]
        [sentencias]
        [return  [expresión];]
    }
```

cuerpo

tipo_función, nombre_función: igual que en el prototipo

([tipo1 var1],...,[tipoN varN]) lista de parámetros formales, los nombres de las variables no son opcionales

Definición de funciones

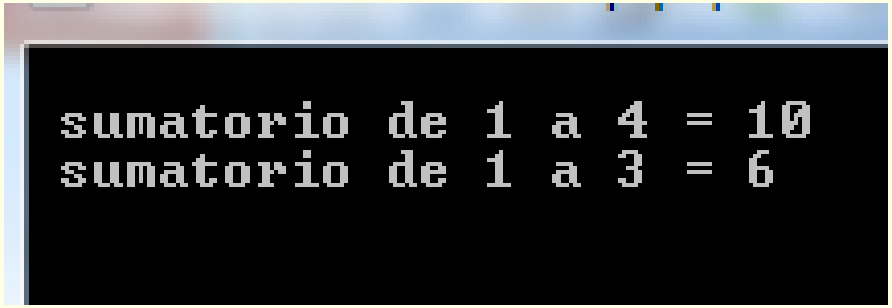
([tipo1 var1],…,[tipoN varN]) lista de **parámetros formales**, los nombres de las variables no son opcionales.

Son los **identificadores** de las variables que recogen los valores que se le pasan en la llamada.

Son **variables locales** a la función. Es decir: se crean cuando la función es llamada y se destruyen cuando termina su ejecución.

Funciones. Ejemplo

```
#include <stdio.h>
int  sumatorio (int n);                /* prototipo (declaración) de la función */
void main ()
{
    int  x, y;
    x = 4;
    printf ("\n sumatorio de 1 a %d = %d", x,  sumatorio(x));
    y = 3;
    printf ("\n sumatorio de 1 a %d = %d", y,  sumatorio(y));
}

int  sumatorio (int  n)
{    int   i,  sum;
     sum = 0;
     for (i=1; i<=n; i++)
          sum  = sum  + i;
     return  sum;
}
```



Definición de funciones

- La **lista de parámetros** es una lista de nombres de variable separados por comas con sus tipos asociados.
- Los **parámetros** reciben los **valores** que le pasan en la llamada (de los argumentos).
- Una función puede **no tener parámetros**, los paréntesis son obligatorios en cualquier caso.
- Si no existe parámetros puede **expresarse de dos formas:**

  tipo_función nombre_función ()

  tipo_función nombre_función (void)
- Todos los **parámetros** deben **declararse individualmente**, no ocurre como en las declaraciones de variables.

Ejem.- cabecera de una declaración de función:

```
int  f (int i, int k, float j)    /*correcto */
int  f (int i, k, float j)        /*incorrecto, le falte el tipo a K */
```

Definición de funciones

- El **cuerpo** de una función aparece entre **{ }** aunque esté formado por una sola sentencia.
- Las variables definidas dentro de la función son **variables locales**. Una variable local comienza a existir cuando se entra en la función y se destruye al salir de ella (se podrán usar nombres idénticos en otras funciones).
- Los **parámetros** son también valores **locales** a la función.

- Terminación de una función: **return**

  Fuerza la salida de la función. Toda función que no devuelva tipo void debe tener al menos una sentencia return y a continuación el valor que devuelve la función.

Resultado de la función: sentencia return

- La forma general de la sentencia return es:

return [expresión];

- Si el tipo de datos de la expresión no coincide con el tipo de retorno de la función, el tipo de datos de la expresión se convierte al de retorno de la función.
- El valor devuelto puede ser **cualquier tipo de dato** conocido por el lenguaje C
- Cuando el tipo de resultado es **void** (la función no devuelve ningún valor) el formato será: **return;** o no se pone la sentencia return y la función termina cuando se encuentra **}**

Función main

- La primera función que aparece (se ejecuta) en todo programa C es la función principal (función main)
- Todo programa ejecutable tiene una y sólo una función main
- Es la única función que no puede ser utilizada (invocada) por nadie.
- No puede llamarse a sí misma.

Funciones

Realizar la definición de una función **esLetra**, en lenguaje C, que reciba un carácter y devuelva un 1 si es una letra y 0 si no es una letra.

Definición:

```
int esLetra (char c)
{
  return(c>='A'&& c<='Z')|| (c>='a'&& c<='z');
}
```

De otra forma:

```
int esLetra (char c)
{
  int vocal;
  if ((c>='A'&& c<='Z')|| (c>='a'&& c<='z')) vocal=1;
  else vocal=0;
  return vocal;
}
```

Funciones

Realizar la definición de una función **esLetra**, en lenguaje C, que reciba un carácter y escriba un mensaje en pantalla indicando si es una letra o no lo es.

Definición:

```
void esLetra (char c)
{
  if ((c>='A'&& c<='Z')|| (c>='a'&& c<='z'))
     printf("Es letra");
  else
     printf("No es letra");
}
```

Índice

- Funciones
- Declaración de funciones. Prototipo
- Definición de funciones
- **Llamada a la función**
- Parámetros
- Recursividad

Llamada a la función

- La forma de invocar (o llamar) una función es:

nombre_función ([argumento1],..., [argumentoN]);

- Los **paréntesis** son **obligatorios**, aunque no tenga argumentos.

- Los argumentos pueden ser nombres de variables o expresiones.

- Los argumentos (**parámetro actual o real**) **se corresponden uno a uno** con los parámetros (**parámetros formales**): El primer valor se asigna al primer parámetros, el segundo al segundo...

- La llamada a una función **puede ser:**
  - una **sentencia** cuando no devuelve ningún valor
  - o **formar parte de una expresión** cuando devuelve algún valor

Funciones

```
#include <stdio.h>
int  restar (int n, int m);
int main ()
{
      int  x, y, result;
      x = 4; y = 5;
      printf ("\n  resta de  %d  y  %d = %d", x, y,  restar ( x, y ) );
      result = restar( 12, x+y );
      printf ("\n  resta de  %d  y  %d = %d", 12, x+y,  result);
      return 0;
}

int  restar ( int  n, int  m )
{     int   resta;              /* se puede prescindir de resta y poner  return n-m  */
      resta =  n - m;
      return  resta;
}
```

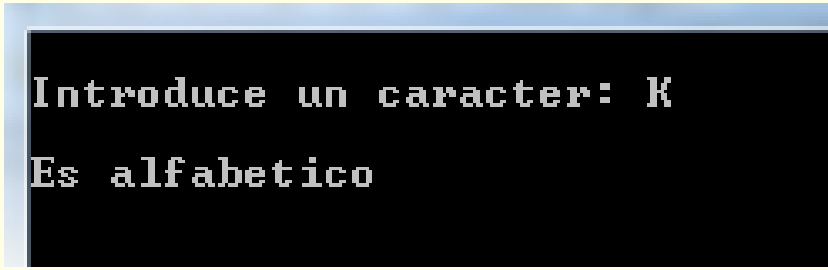
Argumentos.
Parámetros reales

Parámetros.
Parámetros formales

Funciones

Escribir un **programa** que lea un carácter y nos indique si es un carácter alfabético o no lo es. Se tendrá que utilizar una función **esLetra,** que reciba un carácter y devuelva un 1 si es una letra y 0 si no es una letra.

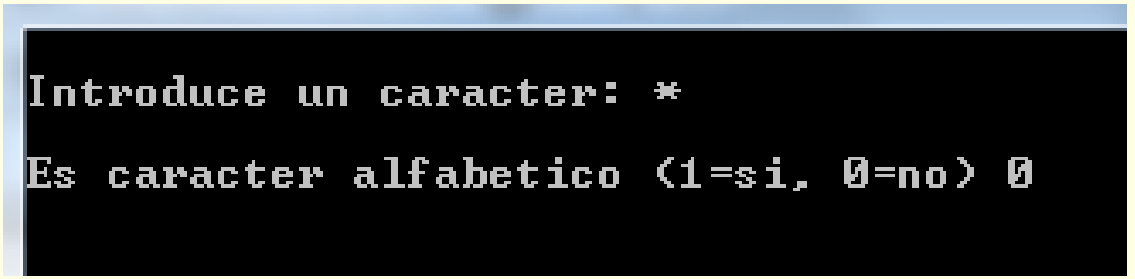
```
#include <stdio.h>
int esLetra (char c);
int main ()
{
  char car;
  printf ("\nIntroduce un caracter: ");
  scanf("%c", &car);
  if (esLetra(car))
     printf ("\nEs alfabetico");
  else  printf ("\nNo es alfabetico");
  return 0;
}
int esLetra (char c)
{
  return((c>='A'&& c<='Z')|| (c>='a'&& c<='z'));
}
```



Funciones

Escribir un **programa** que lea un carácter y nos indique si es un carácter alfabético o no lo es. Se tendrá que utilizar una función **esLetra,** que reciba un carácter y devuelva un 1 si es una letra y 0 si no es una letra.

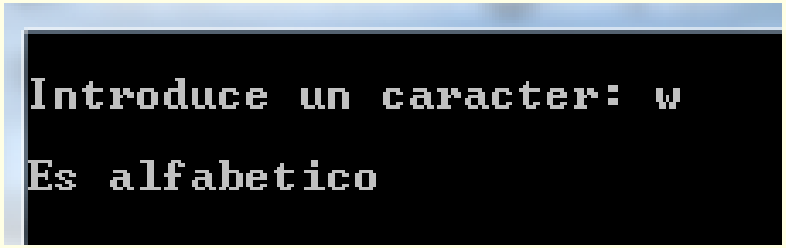
```
#include <stdio.h>
int esLetra (char c);
int main ()
{
  char car;
  printf ("\nIntroduce un caracter: ");
  scanf("%c", &car);
  printf("\nEs caracter alfabetico (1=si, 0=no) %d", esLetra(car));
  return 0;
}
int esLetra (char c)
{
  return((c>='A'&& c<= 'Z')|| (c>= 'a'&& c<= 'z'));
}
```



Funciones

Escribir un **programa** que lea un carácter y nos indique si es un carácter alfabético o no lo es. Se tendrá que utilizar una función **esLetra,** que reciba un carácter y escriba un mensaje en pantalla indicando si es una letra o no lo es

```
#include <stdio.h>
void esLetra (char c);
void main ()
{
  char car;
  printf ("\nIntroduce un caracter: ");
  scanf("%c", &car);
  esLetra(car);
}
void esLetra (char c)
{
  if ((c>='A'&& c<= 'Z')|| (c>= 'a'&& c<= 'z'))
    printf ("\nEs alfabetico");
  else  printf ("\nNo es alfabetico");
}
```



Variables locales y globales

- Variables **locales**
  - Declaración: interior de la función.
  - Ámbito: interior de la función.
  - Tiempo de vida: ejecución de la función.

- Variables **globales**
  - Declaración: fuera de cualquier función.
  - Ámbito: interior de todas las funciones.
  - Tiempo de vida: ejecución del programa.
  - **Problema**: produce efectos colaterales no deseados.

Índice

- Funciones
- Declaración de funciones. Prototipo
- Definición de funciones
- Llamada a la función
- **Parámetros**
- Recursividad

Parámetros de las funciones

- Existen dos formas de paso de argumentos a las funciones.
  - **Parámetros por valor (de entrada)**
  - **Parámetros referencia (de salida o de entrada/salida)**
- En el caso de utilizar la llamada por valor, el código de la función no puede alterar los argumentos usados en la llamada a la función.
- El convenio de paso por parámetros en C es la llamada por valor.

Resumen.
Parámetro de entrada y de entrada/salida

Parámetro de entrada (valor)

- Declaración

```
tipo_función  nombre_función (tipo1 param1, ... )
```

- Llamada

```
nombre_función (expresión1, ... )
```

Parámetro de salida o de entrada/salida

- Declaración

```
tipo_función  nombre_función (tipo1 *param1, ... )
```

- Llamada

```
nombre_función (&variable1, ... )
```

Parámetros de las funciones

Parámetros de entrada (por valor):

- En la llamada a la función los argumentos (parámetros actuales o reales) se evalúan y **se pasa una copia de los valores** a los parámetros (parámetros formales). En este caso, a los parámetros se les llama **parámetros valor.**

- Las modificaciones que la función realiza sobre los parámetros no se transmiten a los argumentos correspondientes. A esto se le llama **paso por valor.**

Ejemplo

```
#include <stdio.h>
int cuadrado (int y);

void main ()
{
    int   x;
    printf (" Introduce un valor entero: ");
    scanf("%d", &x);
    printf ("\n El cuadrado de %d es %d",x,cuadrado(x));
    return 0;
}

int cuadrado (int y)
{
   return  y * y;
}
```

```
Introduce un valor entero: 3

El cuadrado de 3 es 9
```

La variable x sigue teniendo su valor inicial 3

Parámetros de las funciones

Parámetros de salida o de entrada/salida:

- Cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado, se ha de utilizar el **paso de parámetros por referencia (dirección o variable).**
- El **argumento** (parámetro actual o real) será la **dirección de memoria de la variable** que queremos modificar.
- En C no existe el paso de parámetro por referencia, pero se pueden implementar los parámetros de salida con punteros.
- En C el **parámetro formal** debe declararse como **puntero (con un *)** y el **parámetro actual (argumento)** correspondiente debe **ir precedido del símbolo &** que indica la dirección de los valores pasados.

Resumen.
Parámetro de entrada y de entrada/salida

Parámetro de entrada (valor)

- Declaración

```
tipo_función  nombre_función (tipo1 param1, ... )
```

- Llamada

```
nombre_función (expresión1, ... )
```

Parámetro de salida o de entrada/salida

- Declaración

```
tipo_función  nombre_función (tipo1 *param1, ... )
```

- Llamada

```
nombre_función (&variable1, ... )
```

Resumen.
Diferencias: parámetros por valor y por variable

Parámetro de entrada (valor)

- Los **parámetros valor reciben copias** de los valores de los argumentos (parámetros actuales o reales) que se les pasan.
- **La asignación** a parámetros **por valor** de una función **nunca cambia el valor del parámetro actual** pasado como argumento.

Parámetro de salida o de entrada/salida

- Los **parámetros variables** (declarados con *, punteros) **reciben la dirección** de los argumentos pasados (parámetros reales); a estos les debe preceder el &
- En **las asignaciones** a parámetros **variables** de una función **cambia los valores de los argumentos** parámetros actuales.

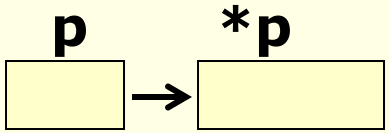
Paso de parámetros y punteros

- Un **puntero** es la dirección de memoria de una variable
- Cuando se declaran punteros hay que indicar el tipo de dato al que apuntan:

tipo_dato_apuntado *identificador_puntero

Ejemplo:

int *p; define una variable puntero p que apunta a una variable de tipo entero



p variable puntero: indica la dirección de memoria que señala a una variable anónima que contiene un valor de tipo entero

***p** variable anónima: contiene un valor entero, es la variable a la que señala la dirección de memoria p

Punteros. Operadores

El operador dirección (&):

- se aplica a cualquier variable y devuelve la dirección de memoria de la variable.
- no es propio de los punteros, se puede aplicar a todas las variables (por ejemplo, visto en la función scanf).

El operador indirección (*):

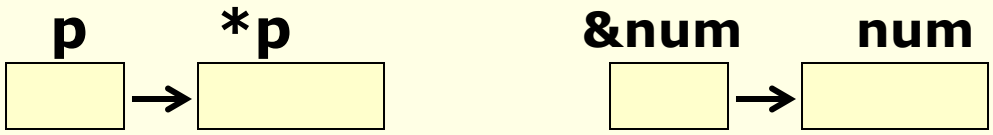
- devuelve el contenido de la posición de memoria "apuntada" por el puntero.
- sólo se aplica a los punteros.

*** es el operador complementario de &**

& y * tienen la **misma precedencia** que los **operadores monarios**

Ejemplo (operadores & y * representados gráficamente):

```
int  *p;
int num;
```



Ejemplos. Parámetros de entrada/salida

```
#include <stdio.h>
void intercambiar (int *x, int *y);

int main ()
{
   int n1 = 18, n2 = 3;
   printf ("\n n1= %d  n2= %d ", n1, n2 );
   intercambiar(&n1, &n2);
   printf ("\n n1= %d  n2= %d ", n1, n2 );
   return 0;
}

void intercambiar (int  *x, int  *y)
{
   int aux;
   aux = *x;
   *x = *y;
   *y = aux;
}
```

```
n1= 18   n2= 3
n1= 3    n2= 18
```

Ejemplos. Parámetros de entrada/salida

Dado el siguiente programa, indicar cuál sería la salida:

```
#include <stdio.h>
void valor (int num);
void  direc (int *num);
void main ()
{
   int  a;
   a = 4;
   direc( &a );
   printf ("a= %d \n", a );
   valor( a );
   printf ("a= %d \n", a );
}
```

```
void valor ( int num)
{
    num = num+1;
    printf (“num= %d ", num );
}

void direc ( int *num)
{
    *num = *num+1;
    printf (“num= %d ", *num);
}
```

Ejemplos. Parámetros de entrada/salida

Dado el siguiente programa, indicar cuál sería la salida:

```
#include <stdio.h>
void valor (int num);
void  direc (int *num);
void main ()
{
   int  a;
   a = 4;
   valor( a );
   printf ("a= %d \n", a );
   direc( &a );
   printf ("a= %d \n", a );
}
```

```
void valor ( int num)
{
    num = num+1;
    printf (“num= %d ", num );
}

void direc ( int *num)
{
    *num = *num+1;
    printf (“num= %d ", *num);
}
```

Ejemplos. Parámetros de entrada/salida

Las siguientes funciones calculan la potencia de dos números enteros positivos n^m y transmiten dicho valor a la función main. Escribir el cuerpo de las funciones, para cada caso:

```
int Potencia (int n, int m)
```

```
void Potencia (int n, int m, int *pot)
```

Ejemplos. Parámetros de entrada/salida

- Si tuviésemos declaradas las siguientes variables en la función principal:

  int a, b, c;

  escribe, para cada una de las funciones anteriores, una sentencia correcta de llamada a la función, desde la función principal:

int Potencia (int n, int m)

void Potencia (int n, int m, int *pot)

Paso de parámetros (con punteros)

```
#include <stdio.h>
void intercambiar (int *x, int *y);
void main ()
{  int a=3, b=5;
   printf("a= %d, b= %d\n", a, b);
   intercambiar (&a, &b);
   printf ("a= %d, b= %d\n", a, b);
}
void intercambiar (int *x, int *y)
{  int aux;
   printf ("x= %d, y= %d\n", *x, *y);
   aux = *x;
   *x = *y;
   *y = aux;
   printf ("x= %d y= %d\n", *x, *y);
}
```

Resultado en pantalla:

Índice

- Funciones
- Declaración de funciones. Prototipo
- Definición de funciones
- Llamada a la función
- Parámetros
- **Recursividad**

Recursividad

- Un subprograma es recursivo cuando se llama a sí mismo.

(cuando se define en términos de él mismo)

Recursividad

Todo subprograma recursivo debe constar de :

- Un **caso base** (límite), en el que se realiza la ejecución del subprograma sin llamarse a sí mismo.

- Un **paso recursivo**, en el que se realiza una llamada a sí mismo, de tal forma que se aproxime al caso base.

Recursividad. Ejemplo1

- Definición recursiva (en términos de ella misma) de la función factorial para valores positivos:

$$\text{factorial}(n) = \begin{cases} 1 & \text{si } n = 1 \\ n * \text{factorial}(n - 1) & \text{si } n > 1 \end{cases}$$

Recursividad. Ejemplo1

```
/* función factorial recursiva: */

int factorial (int  n)
{
   if  (n==1)
      return 1;
   else
      return n * factorial(n-1);
                  /* llamada recursiva*/
}
```

Recursividad. Ejemplo1

Caso base: n = 1

valor de la función = 1

Paso recursivo: n > 1

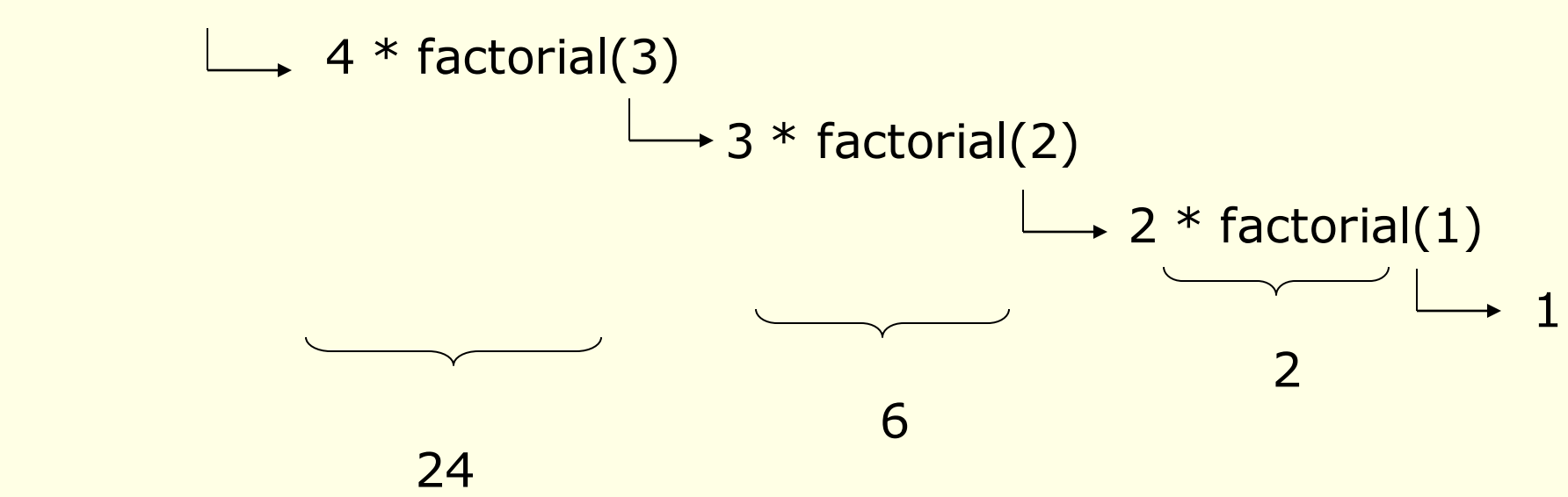
valor de la función = n * **factorial (n-1)**

Llamada recursiva a la función: factorial (n-1)

Recursividad. Ejemplo1

Llamada desde el programa : factorial(4)

Ejecución:



Recursividad

- Efecto de las llamadas recursivas:
  - Se reserva espacio de memoria para almacenar los objetos locales (parámetros, variables).
  - Se reciben los parámetros y se comienza la ejecución de las sentencias del cuerpo del subprograma.
  - Cuando termina el ejecución, se libera el espacio reservado, los identificadores locales dejan de tener vigencia y se ejecuta la siguiente instrucción a la llamada.
- Este proceso se repite, hasta que se llegue a un caso base.

Recursividad

- Efecto de las llamadas recursivas: