PROGRAMACIÓN 1

Grado en Ingeniería Informática e I2ADE

Tema 4

Programación modular

Dept. de Ciència de la Computació i Intel·ligència Artificial
Dpto. de Ciencia de la Computación e Inteligencia Artificial

Universitat d'Alacant
Universidad de Alicante

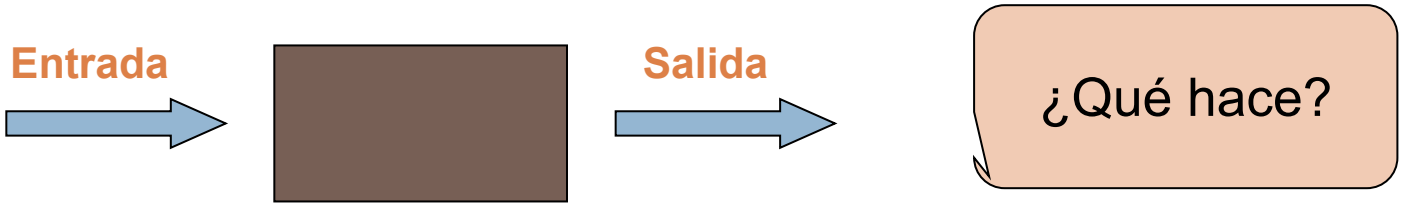
Índice

1. Concepto de módulo

- Cuando un programa es grande y complejo no es conveniente que todo el código esté dentro del programa principal (función `main()` en lenguaje C).
- Un módulo …
  - es un **fragmento de código** que se escribe de forma **independiente** al resto del programa.
  - se encarga de **realizar una tarea concreta** que resuelve un problema parcial del problema principal.
  - puede ser ***invocado*** (llamado) desde el programa principal o **desde otros módulos**
  - permite **ocultar los detalles** de la solución de un problema parcial (*caja negra*)

1. Concepto de módulo

- Cada módulo es una ***caja negra*** para el programa principal y para el resto de módulos.
- Para utilizar un módulo desde el programa principal o desde otros módulos …
  - Necesitamos conocer qué hace y su interfaz, es decir, sus entradas y salidas

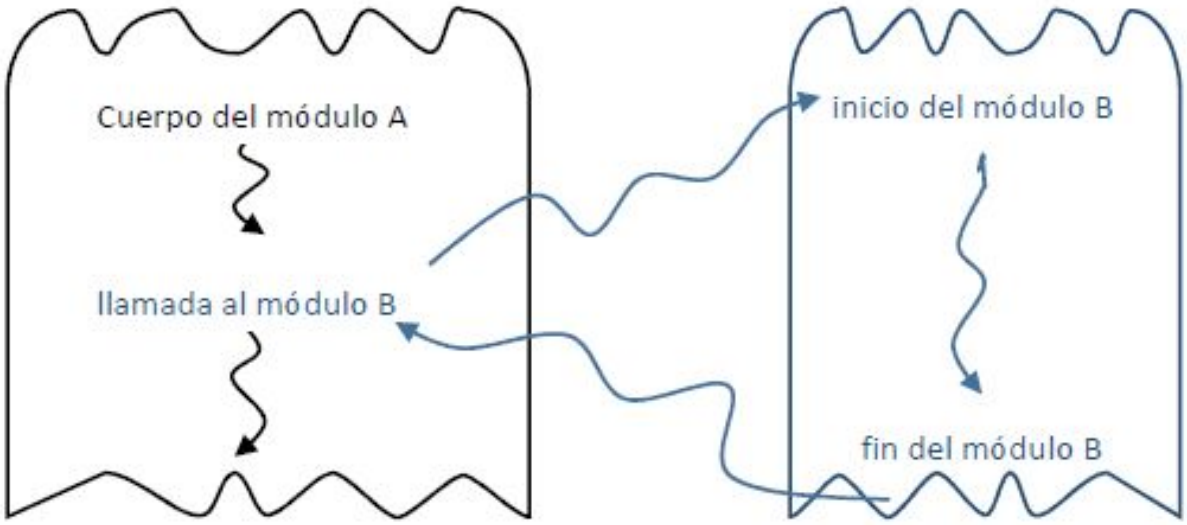


  - No necesitamos conocer los detalles internos de funcionamiento

1. Concepto de módulo

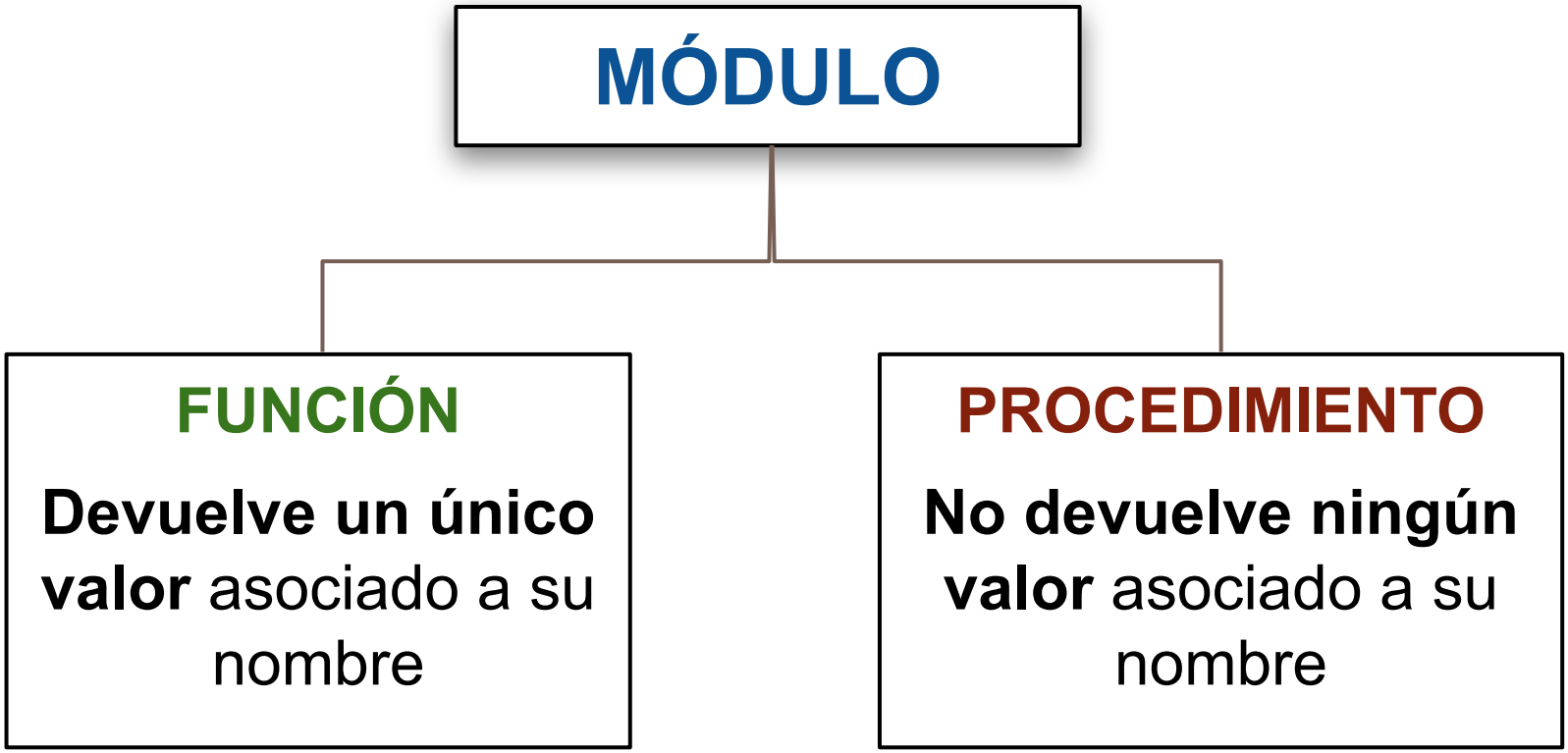
Transferencia del flujo de control

- Cuando un módulo A llama (invoca) a otro módulo B, el flujo de control (flujo de ejecución) pasa al módulo B
- Cuando termina de ejecutarse el módulo B, el flujo de control continúa en el módulo A, a partir de la sentencia siguiente a la llamada al módulo B



2. Tipos de módulos

Un módulo puede ser de dos tipos **en función de si devuelve o no un valor**:



3. Funciones en C

- Una **función** es un módulo que **devuelve un único resultado** asociado a su nombre.
- Puede recibir cero o más datos (parámetros) de entrada y en base a ellos, genera y devuelve un resultado mediante la instrucción **`return`**.

  Ejemplos: las funciones matemáticas

Función	Entrada	Salida
Raíz cuadrada (9)	9	3
Potencia (4, 3)	4, 3	64
Log (4)	4	0.60206
Cos (π)	π	-1

3. Funciones en C

Sintaxis

```
tipo_ret nombre( tipo_par1 par1, tipo_par2 par2, ...  ) {
  secuencia de sentencias
  return valor_de_retorno;
}
```

donde:

- `tipo_ret`: Tipo de dato del valor a devolver por la función.
- `nombre`: Identificador, nombre de la función.
- `tipo_parN`: Tipo de dato del parámetro (valor) de entrada en la posición *N*.
- `parN`: Nombre con el que se recoge el parámetro de entrada en la posición *N* para luego poder utilizarlo en la función.

3. Funciones en C

Cómo se usan las funciones

- Para ejecutar una función hay que **realizar una llamada** desde el `main` o desde otro módulo.
- Para llamar a una función **se usa su nombre** poniendo entre paréntesis los parámetros de entrada ordenados y separados por comas.
- Si la función no tiene parámetros de entrada se ponen los paréntesis vacíos.
- Cuando se llama a una función, se obtiene un valor como resultado. **Es muy importante que la llamada se incluya en una expresión que permita aprovechar el resultado**.

3. Funciones en C

Ejemplo:

Función en C que recibe dos números y devuelve el más grande de los dos.

Parámetros (datos) de entrada

```
int max (int a , int b){
   int m; //variable local
   if (a > b)
      m = a;
   else
      m = b;
   return (m);
}
```

Tipo de dato del valor de retorno

Las variables x e y en main() se recogen en la función max() como a y b, respectivamente.

```
int main (){
   int x,y,mayor; //variables locales
   printf("Escribe dos números: ");
   scanf("%d %d", &x, &y);
   mayor = max(x, y);
   printf("El mayor es: %d\n", mayor);
   return 0;
}
```

Dato de salida a devolver mediante la sentencia **return**. En el main() lo recogerá la variable **mayor**, que debe tener el mismo tipo de dato.

Sentencia `return`

- **Finaliza** la ejecución del cuerpo de la función.
- Se encarga de **devolver el valor de retorno** de la función, después de evaluar su *expresión* asociada.
- Es recomendable usar **una sola sentencia `return`** dentro del cuerpo de una función. De esta forma se mejora la legibilidad y el mantenimiento del código.
- Debería ser la **última sentencia** del cuerpo de la función.

```
return (expresión);
```

4. Procedimientos en C

- Un **procedimiento** es un módulo que realiza una tarea específica.
- Puede recibir cero o más valores (parámetros) de entrada para realizar dicha tarea.
- Puede devolver cero o más valores a través de los mismos parámetros que recibe, **NUNCA** mediante `return`.
- La diferencia con las funciones es, precisamente, que los procedimientos no utilizan `return`, por lo que su tipo de dato de retorno es **`void`** (vacío).

4. Procedimientos en C

Sintaxis

```
void nombre( tipo_par1 par1, tipo_par2 par2, ...  ) {
  secuencia de sentencias
}
```

donde:

- `void`: Tipo de dato vacío, ya que no devuelve ningún dato asociado a su nombre.
- `nombre`: Identificador, nombre de la función.
- `tipo_parN`: Tipo de dato del parámetro (valor) de entrada en la posición *N*.
- **par*N***: Nombre con el que se recoge el parámetro de entrada en la posición *N* para luego poder utilizarlo en la función.

4. Procedimientos en C

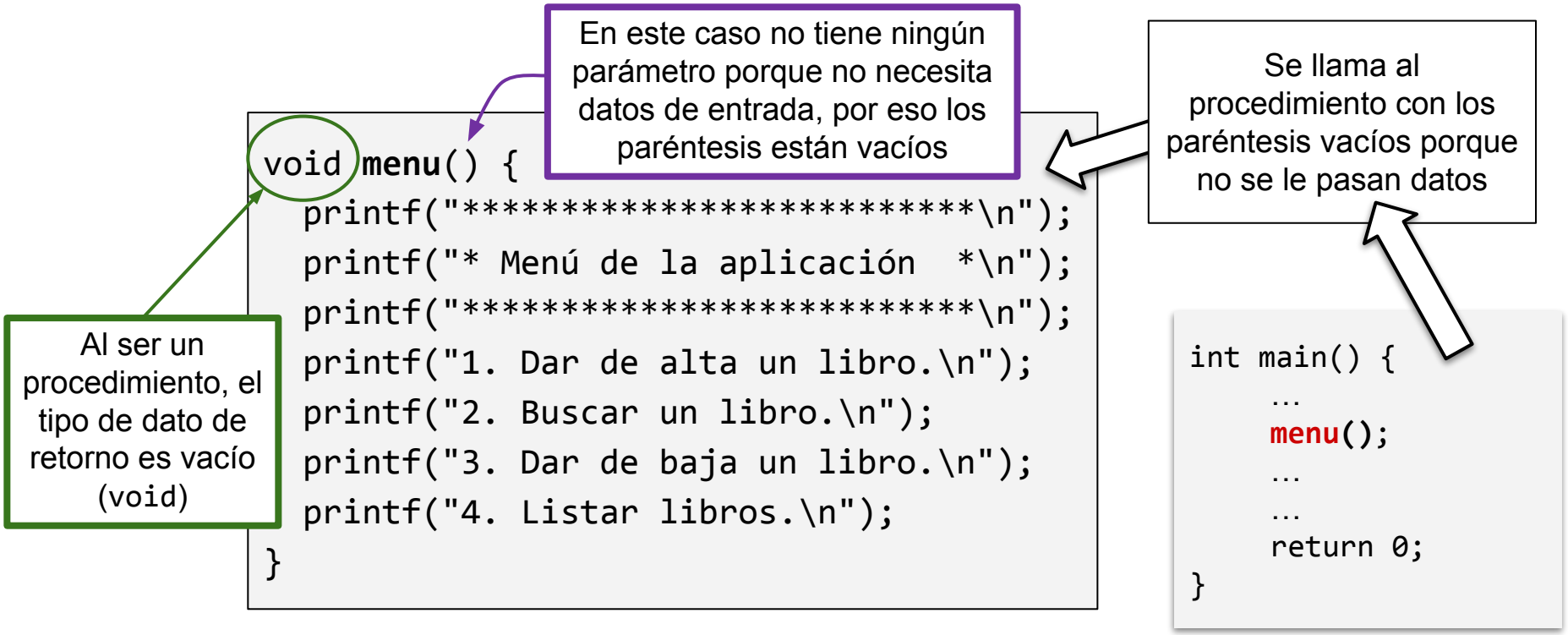
Cómo se usan los procedimientos

- Para ejecutar un procedimiento hay que **realizar una llamada** desde el main o desde otro módulo.
- Para llamar a un procedimiento **se usa su nombre** poniendo entre paréntesis los parámetros separados por comas.
- Si el procedimiento no tiene parámetros se ponen los paréntesis vacíos.
- La llamada a un procedimiento no se incluye en ninguna expresión, siempre será una única sentencia de código.

4. Procedimientos en C

Ejemplo:

Procedimiento en C que no recibe ningún parámetro y muestra por pantalla todas las opciones del menú de una aplicación.



Al ser un procedimiento, no tiene ninguna instrucción `return`.

5. Declaración y definición de módulos en C

- **La declaración de un módulo** consiste en asociar el nombre del módulo con el tipo de dato de retorno y los tipos de datos de los parámetros de entrada. Es lo que se llama **prototipo del módulo**. Se escriben antes del módulo principal `main()`.
- **La definición de un módulo** es la implementación en sí del módulo. Es decir, el conjunto de sentencias que forman el módulo. Se escriben después del módulo principal `main()`.
- Aunque no es obligatorio declarar los módulos antes de definirlos, en cuyo caso la definición iría antes del módulo principal `main()`, es muy conveniente puesto que se evitan posibles errores en la fase de compilado/enlazado del programa.

5. Declaración y definición de módulos en C

Ejemplo de ubicación de la declaración y la definición de módulos

Prototipos

```
#directivas del preprocesador
Declaración de constantes
Declaración de procedimientos y funciones
main() {
   Declaración de variables (de tipos simples)
   Cuerpo principal
     sentencias de control
     llamadas a procedimientos y funciones
}
Definición de procedimientos y funciones
```

```
#include <stdio.h>
#include <stdbool.h>

// Declaración de módulos (Prototipos)
bool esPar(int numero);

// Módulo principal main()
int main(){
  ...

  if( esPar(n) )
    ...

  return 0;
}

// Definición de módulos
bool esPar(int numero){
  bool valRet;

  if( numero % 2 == 0)
      valRet = true;
  else
      valRet = false;

  return valRet;
}
```

6. Parámetros de un módulo

- Los parámetros se utilizan para la transferencia de información entre módulos. Este proceso se llama **paso de parámetros o argumentos**.
- Un parámetro **permite pasar información desde un módulo a otro** y viceversa.
- Un parámetro en un módulo puede ser considerado como una variable cuyo valor debe ser proporcionado por otro módulo o bien ser devuelto desde el propio módulo.
- Se pueden identificar **tres tipos de parámetros**:
  - De entrada: sus valores son proporcionados por el módulo que llama.
  - De salida: sus valores se calculan en el subprograma (módulo) y se devuelven al módulo que ha llamado.
  - De entrada/salida: proporcionan valores al módulo y éstos además pueden ser modificados y ser devueltos al módulo que ha llamado.

6. Parámetros de un módulo

Parámetros formales y actuales

- Los **parámetros formales** son los que aparecen en la declaración del módulo.
- Los **parámetros actuales** o reales son los que aparecen en la llamada al módulo.
- La correspondencia entre los parámetros formales y actuales es la siguiente:

Debe coincidir	No es necesario que coincida
● Número de parámetros ● Tipo de dato de los parámetros ● Orden de los parámetros	● Nombre de los parámetros

6. Parámetros de un módulo

Paso de parámetros

- Paso de parámetros **por valor**. Se pasa una copia del valor original del parámetro actual o real. En este caso, si se modifica el valor del parámetro en el módulo, el valor original del parámetro real no se ve modificado en el punto del programa en el que se hizo la llamada al módulo.
- Paso de parámetros **por referencia**. Se pasa el valor original del parámetro actual o real. Es decir, se pasa una *referencia* a la dirección de memoria en la que se almacena dicho valor. En este caso, si se modifica el valor del parámetro en el módulo, el valor original del parámetro real también se ve modificado en el punto del programa en el que se hizo la llamada al módulo. Para ello, se escribe **&** delante del parámetro real en la llamada y ***** delante del parámetro formal en la declaración del módulo.

6. Parámetros de un módulo

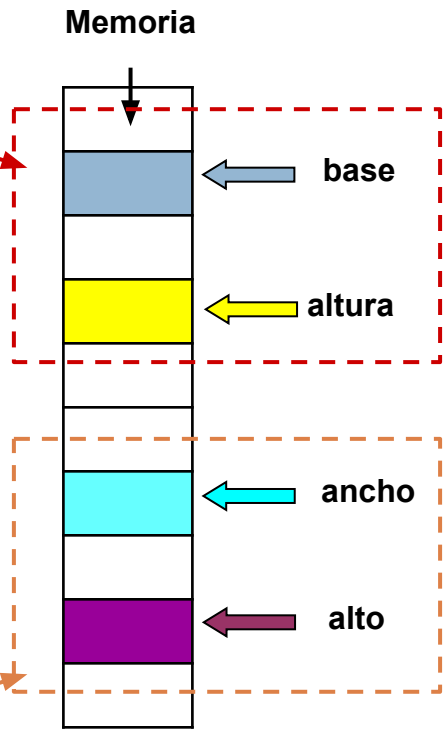
Paso de parámetros por valor

```
int main() {
  int base, altura, area, perimetro;

  printf("Dime la base del rectángulo: ");
  scanf("%d", &base);
  printf("Dime su altura: ");
  scanf("%d", &altura);

  rectangulo(base, altura, &area, &perimetro);

  printf("Área: %d\n", area);
  printf("Perímetro: %d\n", perimetro);

  return 0;
}
```

Parámetros `base` y `altura` originales

Paso de los parámetros `base` y `altura` **por valor** en la llamada al módulo

Copia de los parámetros `base` y `altura` originales recibidos en el módulo como `ancho` y `alto`




```
void rectangulo( int ancho, int alto, int *area_rect, int *perim ) {
   *area_rect = ancho * alto;
   *perim = 2 * (ancho + alto);
}
```

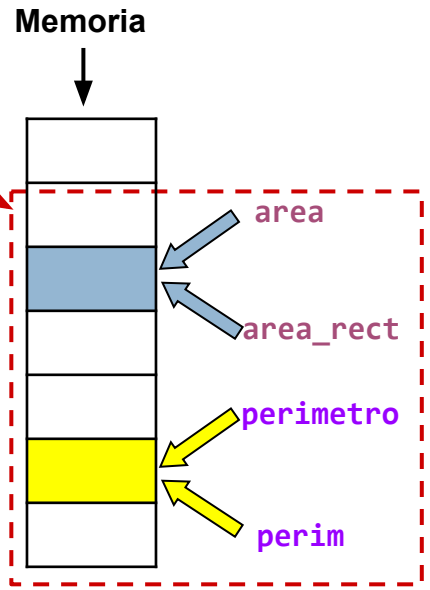
Recepción **por valor** de los parámetros `base` y `altura` en el punto de la llamada

6. Parámetros de un módulo

Paso de parámetros por referencia

```
int main() {
  int base, altura, area, perimetro;

  printf("Dime la base del rectángulo: ");
  scanf("%d", &base);
  printf("Dime su altura: ");
  scanf("%d", &altura);

  rectangulo(base, altura, &area, &perimetro);

  printf("Área: %d\n", area);
  printf("Perímetro: %d\n", perimetro);

  return 0;
}
```

Parámetros `area` y `perimetro` originales, que son los mismos que `area_rect` y `perim` en el módulo



Paso de los parámetros `area` y `perimetro` **por referencia** en la llamada al módulo. Se escribe **&** delante de cada parámetro a pasar por referencia.

```
void rectangulo( int ancho, int alto, int *area_rect, int *perim ) {
  *area_rect = ancho * alto;
  *perim = 2 * (ancho + alto);
}
```

Recepción como parámetros **por referencia** de las variables `area` y `perimetro` en el punto de la llamada. Se utiliza * delante de los parámetros formales en la declaración.

Al recibir los parámetros **por referencia** se utiliza * delante de ellos cada vez que se usen en el módulo.

7. Ámbito de una variable

El ámbito de una variable define la visibilidad de la misma, es decir, **la parte del programa donde la variable es accesible**.

```
#include<stdio.h>
#include<stdbool.h>

bool es_primo(int);

int main() {
  int n; // número introducido por
         // teclado (dato de entrada)

  printf("Introduce un número entero: ");
  scanf("%d", &n);
  if ( es_primo(n) )
    printf("El número es primo\n");
  else
    printf("El número no es primo\n");

  return 0;
}
```

ámbito de **n**

ámbito de **num**, **cont**, **primo**

```
// Este módulo comprueba si un número
// es primo o no
bool es_primo(int num) {
  int cont; // contador (dato auxiliar)
  bool primo; // es primo o no
              // (dato de salida)

  primo = true;
  cont = 2;
  while ( (cont < num) && primo) {
     // comprobar si es divisible por
     // otro número
     primo = ! (num % cont == 0);
     cont = cont + 1;
   }
  return primo;
}
```

7. Ámbito de una variable

Variables locales y variables globales

- Variable **local**
  - Su ámbito es el cuerpo del módulo en el que se declara.
  - Se crea cuando se declara y se destruye cuando finaliza la ejecución del módulo.
- Variable **global**
  - Su ámbito es todo el programa (todos sus módulos).
  - Se crea cuando se declara y se destruye cuando finaliza la ejecución del programa.

7. Ámbito de una variable

Comunicación entre módulos

- Debe realizarse a través de parámetros y **NO de variables globales**.
- El uso de variables globales no es aconsejable porque:
  - Dificulta la legibilidad del código.
  - Puede producir efectos colaterales al cambiar el valor de una variable global en un módulo.
  - Es contrario a uno de los conceptos más importantes de la programación: la **modularidad**

Nota: En la asignatura de Programación 1 **no se permite el uso de variables globales**.

7. Ámbito de una variable

Ventajas de la programación modular

- Facilita el diseño descendente y la programación estructurada
- Reduce el tiempo de programación
  - *Reusabilidad*: estructuración en *librerías* específicas (biblioteca de módulos)
  - *División* de la tarea de programación entre un equipo de programadores
- Disminuye el tamaño total del programa
  - Un *módulo* sólo está escrito una vez y puede ser utilizado varias veces desde distintas partes del programa
- Facilita la detección y corrección de errores
  - Mediante la comprobación individual de los módulos
- Facilita el mantenimiento del programa
  - Los programas son más fáciles de modificar
  - Los programas son más fáciles de entender (más legibles)

8. Bibliotecas del lenguaje C

- La mayoría de lenguajes de programación proporcionan una colección de procedimientos y funciones de uso común (**bibliotecas** o **librerías**)
- En el lenguaje C, para hacer uso de los módulos incluidos en una biblioteca se utiliza la directiva del compilador **`#include`**
- Existe una gran variedad de bibliotecas disponibles:
  - Funciones matemáticas
  - Manejo de caracteres y de cadenas de caracteres
  - Manejo de entrada y salida de datos
  - Manejo del tiempo (fecha, hora, …)
  - etc.

8. Bibliotecas del lenguaje C

Ejemplos de librerías del lenguaje C

Librería	Función	Descripción
`<math.h>`	double cos(double x)	Devuelve el coseno de x
	double sin(double x)	Devuelve el seno de x
	double exp(double x)	Devuelve e^x
	double fabs(double x)	Devuelve el valor absoluto de x
	double pow(double x, double y)	Devuelve x^y
	double round(double x)	Devuelve el valor de x redondeado
	double sqrt(double x)	Devuelve la raíz cuadrada de x
`<ctype.h>`	int isalnum(int c)	Devuelve verdadero si el parámetro es una letra o un dígito
	int isdigit(int c)	Devuelve verdadero si el parámetro es un dígito
	int toupper(int c)	Devuelve el carácter en mayúsculas
`<stdlib.h>`	void srand(unsigned int s)	Inicializa al valor que se pasa como argumento la semilla de la secuencia de números aleatorios a generar
	int rand(void)	Devuelve un número aleatorio entre 0 y RAND_MAX

9. Ejercicios

1. Hacer una función que devuelva la letra que le corresponde a un número de DNI que se pasa como parámetro mediante el siguiente algoritmo:
   a) Calcular el resto de la división del DNI entre 23
   b) En función del valor del resto, asociar la letra correspondiente según la siguiente tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

2. Diseña un módulo que reciba como parámetro un número n y dibuje en pantalla un cuadrado de tamaño n formado por asteriscos.
3. Mejora el ejercicio 2 añadiendo otro parámetro que permita que el cuadrado se dibuje con el carácter enviado como parámetro.
4. Diseña un módulo que reciba dos variables como parámetro e intercambie los valores de las mismas.
5. Diseña un módulo que permita leer y validar un dato de entrada de manera que su valor sea mayor que 0 y menor que 100 y devuelva la suma y la cuenta de los números impares entre 1 y dicho valor.

9. Ejercicios (*y II*)

6. Módulo que comprueba si un número que recibe como argumento es negativo, devolviendo `true` o `false`.
7. Módulo que devuelve cuántos divisores tiene un número que recibe como argumento.
8. Módulo que divide dos números enteros que se le pasan como argumento y devuelve el cociente y el resto. Debe comprobar que el divisor no es 0.
9. Módulo que recibe un importe bruto y el porcentaje de impuesto añadido y debe devolver el importe neto y la parte correspondiente al impuesto.
10. Implementa un programa que pida al usuario un número entero y le diga si es primo o no y le devuelva el siguiente número primo mayor al número introducido. Utilizar un módulo para saber si un número es primo; otro para calcular el siguiente número primo a un número dado; y otro módulo (el módulo `main()`) que haga uso de los dos anteriores.