Facultad de Informática

PROGRAMACIÓN I
Sesiones teóricas

Departamento de Ciencias de la Computación
y Tecnologías de la Información

SUBPROGRAMAS

Funciones (Procedimientos)

ARQUITECTURA DE UN PROGRAMA
Técnica Divide y Vencerás

- La **programación estructurada original** se basa en la técnica "**Divide y vencerás**"
  - Descomposición de un problema complejo en **subproblemas** más simples y **manejables**
  - Dividir el problema en **tareas** y cada tarea en una serie de **subprogramas** que incluirán los **algoritmos** más adecuados para cada tarea → **Técnica TOP-DOWN o de Refinamiento Sucesivo**
  - Un **subprograma** o **subrutina** es un conjunto de **instrucciones** que permiten la **ejecución de algún proceso** determinado y **lógico** desde el punto de vista humano
  - Los **esfuerzos** del programador deben concentrarse en que cada subprograma presente:
    - **Interfaz** consistente (comunicación con el resto del programa)
    - Descripción correcta de los **argumentos** de entrada y salida
    - **Independencia** del resto

ARQUITECTURA DE UN PROGRAMA

Subprogramas

- Los subprogramas se escriben **una única vez**, luego es posible hacer referencia a ellos ("invocarlos o **llamarlos**") desde diferentes puntos del código
  - Conjunto de sentencias con un **nombre asignado**
  - Permiten **reutilización** y evitan la duplicación de código
- Los subprogramas son **independientes** entre sí
  - Es posible **codificar** y **verificar** cada módulo o subprograma de **forma separada** sin tener en cuenta los demás módulos
  - Facilitan la **localización de errores** y la **modificación** del código
- Diseño **modular**
  - Programas **legibles** (se recomienda no sobrepasar 25 líneas físicas) y **fiables**
  - **Adaptación** natural al **trabajo colaborativo** o en equipo

ARQUITECTURA DE UN PROGRAMA

Subprogramas

- Permiten **modularidad** (diseño modular top-down)
- Proporcionan **reutilización del código**, se escriben una única vez y se invocan siempre que es necesario
- Proporcionan **abstracción**, se comportan como una caja negra
- Permiten **distribuir el trabajo**, pues dividen el problema en subproblemas
- Mejoran la **depuración de código**, ya que simplifican la localización de errores

ARQUITECTURA DE UN PROGRAMA

SUBPROGRAMAS. Concepto

- Conjunto de sentencias que tienen asociado un identificador y que se ejecutan como un grupo cuando se invocan desde la sección ejecutable del programa para que realicen una tarea

- Un procedimiento es una **UNIDAD** desde el punto de vista:
  - **Físico**: todas las sentencias están **agrupadas físicamente** en un mismo lugar
  - **Lógico**: entre todas las sentencias realizan **una única tarea**
  - **Referencial**: existe un **identificador** para identificar y referirse al conjunto de instrucciones

ARQUITECTURA DE UN PROGRAMA
SUBPROGRAMAS. Tipos.

- Procedimientos/Funciones estándar o **predeterminadas**
  - *SQRT*(dato): calcula la raíz cuadrada de dato.
  - CLOCK( ): proporciona el número de segundos transcurridos en el sistema
  - …
  - Entrada/Salida: printf, scanf, puts,…
  - Gestión memoria: malloc, calloc, free… etc.
- Procedimientos/Funciones **definidas por el usuario**

ARQUITECTURA DE UN PROGRAMA
Funciones en C

- Una función es una serie de declaraciones que se han agrupado y se les ha dado un nombre.
- Cada función es esencialmente un programa pequeño, con sus propias declaraciones y sentencias.
- Ventajas de las funciones:
  - Un programa se puede dividir en partes pequeñas que son más fáciles de entender, modificar y mantener.
  - Podemos evitar duplicar el código que se usa más de una vez.
  - Una función que originalmente era parte de un programa puede ser reutilizada en otros programas.

ARQUITECTURA DE UN PROGRAMA
Funciones

Declaración de una función*:*

return-type function-name (*parametros*)
{
    declaraciones
    sentencias
}

El tipo de retorno de una función es el tipo de valor que la función devuelve.

Reglas que rigen el tipo de devolución:

- Las funciones no pueden devolver matrices.
- Especificar que el tipo de retorno (*return-type*) es "void" indica que la función no devuelve un valor.
- Si se omite el tipo de retorno en C89, se supone que la función devuelve un valor de tipo int.
- En C99, omitir el tipo de retorno es ilegal.

ARQUITECTURA DE UN PROGRAMA
Funciones

C no requiere que la definición de una función preceda a sus llamadas.

PERO: cuando el compilador encuentra la primera llamada a la función en main( ) y no tiene información sobre la misma, en lugar de generar un mensaje de error, el compilador asume que la función devuelve un valor int (genera un mensaje de *warning*)

Decimos que el compilador ha creado una declaración implícita de la función. Esto puede dar muchos problemas!!!

Solución evidente: organizar el programa de modo que la definición de cada función preceda a todas sus llamadas.

return-type function-name (*parameters*) ; `antes del main()`

ARQUITECTURA DE UN PROGRAMA
Funciones

Ejemplo de función:
Esta función calcula el promedio de dos números reales:

```
double media (double a, double b)
      {
         return (a + b)/2;
      }
```

La palabra double al principio de la función es el tipo de datos que devuelve (retorna). Los identificadores a y b (los parámetros de la función) representan los números que se suministrarán cuando se llame a la función media.

```
double media (double a, double b)
      {
          double aux;
          aux= (a+b)/2;  /*código mas claro*/
          return(aux);
       }
```

ARQUITECTURA DE UN PROGRAMA
Funciones: llamadas

- La llamada a una función consiste en el nombre de la función seguido por la lista de ***arguments.***
  - `media(x, y)` es una llamada la función media.
- Los argumentos se utilizan para suministrar información a una función.
- La llamada anterior **media (x, y)** hace que los valores de x e y se copien en los parámetros a y b de la función definida anteriormente:

```
double media (double a, double b)
{
....
}
```

- Un argumento no tiene que ser una variable; cualquier expresión de un tipo compatible servirá.

```
media (5.1, 8.9);
media (x/2,  y/3);
```

Funciones: argumentos

En la definición de funciones que hemos dado, los argumentos se pasan por valor: cuando se llama a una función, cada argumento se evalúa y su valor se asigna al parámetro correspondiente.

Dado que el parámetro contiene una **copia** del valor del argumento, cualquier cambio realizado en el parámetro durante la ejecución de la función no afectará al argumento.

Con un paso de argumentos por valor, el parámetro puede modificarse sin afectar el argumento correspondiente, podemos usar parámetros como variables dentro de la función, reduciendo así el número de variables necesarias.

También se puede realizar el paso de parámetros por referencia: punteros.

Funciones. Ejemplo

media.C

```
#include <stdio.h>

double media(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf(“Introduzca tres números: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf(“La media de %g and %g: %g\n", x, y, media(x, y));
  printf(“La media de %g and %g: %g\n", y, z, media(y, z));
  printf(“La media de %g and %g: %g\n", x, z, media(x, z));

  return 0;
}
```

Dado que la función media(x,y) devuelve un dato (double) podemos usarla directamente, como ejemplos:

```
printf(%g\n”, media(x,y));

auxiliar=media(x,y);

If (media(2,4)> 0) printf(“La media es positiva\n”);
```

Funciones. Retorno

Si una función no es definida como **void** debe usar la declaración de **return** para especificar qué valor devolverá.

return *expresión*; (return 0);

o expresiones más complejas:

return n> = 0? n: 0;

OJO Si el tipo de expresión en una declaración de retorno no coincide con el tipo de retorno de la función, la expresión se convertirá **implícitamente** al tipo de retorno.

Ejecutar una declaración **return** en main es una forma de finalizar, otra forma es llamar a la función **exit**, que pertenece a <stdlib.h>.

Funciones. Retorno

return expression;

Es equivalente a:

exit(expression);

Importante:

La diferencia entre el **exit** y **return** es que **exit** provoca la terminación del **programa** independientemente de la función que lo llame.

La declaración de **return** provoca la finalización del programa solo cuando aparece en la función principal main( ).

Funciones. Retorno.

Además de 0, C nos permite pasar **EXIT_SUCCESS** (el efecto es el mismo):

exit (EXIT_SUCCESS);

Pasar **EXIT_FAILURE** indica terminación anormal:

exit (EXIT_FAILURE);

EXIT_SUCCESS y EXIT_FAILURE son macros definidas en <stdlib.h>.

Los valores de EXIT_SUCCESS y EXIT_FAILURE están definidos por la implementación. Los valores típicos son 0 y 1, respectivamente.

Ejemplos. Número Primo

Numero_primo.c

```
/* Analiza si un número es primo */

#include <stdbool.h>   /* OJO C99 */
#include <stdio.h>

bool es_primo(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n;
   divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

Ejemplos. Número Primo

```
int main(void)
{
  int n;

  printf("Entroduzca un número: ");
  scanf("%d", &n);
  if (es_primo(n))
    printf("Primo\n");
  else
    printf("No es primo\n");
  return 0;
}
```

Ejemplos: Triángulo

El programa como entrada solicita una letra y un número.
Como salida crea/dibuja un Triangulo

```
#include <stdio.h>

void dibujaTriangulo (char letra, int numero)
  {
   int i;
   printf(“\n”);
   while (numero > 0)
     {
       for (i=0; i<numero; i++) printf("%c", letra);
       printf("\n");
       numero--;
     }
    return;
  }

int main()
 {
   char letra;
   int numero;

   printf("Escriba una letra: ");
   scanf("%c", &letra);
   printf("Escriba un numero: ");
   scanf("%d", &numero);
   dibujaTriangulo(letra, numero);
   return 0;
 }
```

Ejemplos: Triángulo

Salida por pantalla es:

```
Escriba una letra: A
Escriba un numero: 6

AAAAAA

AAAAA

AAAA

AAA

AA

A

Process finished with exit code 0
```

RECURSIVIDAD

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD.

- Algo es **recursivo** cuando está definido en términos de sí mismo
- Ejemplos:
  - Los números naturales:

    1 es un número natural

    El siguiente de cualquier número natural también lo es

  - La función factorial:

    Cuando n = 0 $0! = 1$

    Para n>0 $n! = n*(n-1)!$

  - Estructuras árbol

  - La potencia de un número:

    Cuando n = 0 $x_0 = 1$

    Para n>0 $x_n = x * x_n\text{-}1$

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Naturaleza de la recursividad

- Hay un **caso base** en el que el resultado es inmediato
- Hay un **caso general** que para resolverlo nos acercamos al caso base en sucesivas veces

- *La Potencia de la recursión está en poder definir un número infinito de objetos utilizando un enunciado finito*

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Naturaleza de la recursividad

- Recursividad es la capacidad de un rutina de llamarse a sí misma
- Los algoritmos recursivos *suelen* ser apropiados cuando se define de forma recursiva:
  - La tarea a resolver
  - La función a calcular
  - La estructura de datos a procesar
- PERO, en la mayoría de los casos la solución recursiva no es la mejor porque pueden obtenerse soluciones simples por iteración.
- Sin embargo permite resolver de forma natural problemas de tipo recursivo difíciles de abordar

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD.

La siguiente función calcula *n!* recursivamente, usando la ecuación;

$n! = n \times (n - 1)!$:

```
int fact(int n)
  {
      if (n <= 1)
      return 1;
      else
      return n * fact(n - 1);
  }
```

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Naturaleza de la recursividad

- En muchos casos la solución recursiva no es la mejor porque pueden obtenerse soluciones simples por iteración

- La clase de funciones que tienen definiciones de la forma:

$$F_n(x) = G(x) \text{ si } n=0 \text{ y } F_n(x) = H(F_{n-1}(x)) \text{ si } n>0$$

siempre puede expresarse iterativamente, y por tanto una solución recursiva es innecesaria.

```
int fact(int n)
  {
      int aux;
      for(i=1;i>=n;i++) aux=aux*i;
      return aux;
  }
```

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD.

```
int fact(int n)
 {
   if (n <= 1)
   return 1;
   else
   return n * fact(n - 1);
 }
```

Para ver cómo funciona la recursión

```
i = fact(3);
```

tenemos : fact(3) encuentra que 3 no es menor o igual que 1, en nces llama fact(2), que encuentra que 2 no es igual o menor que 1, entonces llama a fact(1), que encuentra que 1 es menor o igual que 1, por lo que devuelve 1, lo que causa que fact(2) devuelva 2 × 1 = 2, causando

fact(3) devualva 3 × 2 = 6.

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Recursión infinita.

- Si un procedimiento o función no termina nunca de llamarse equivale a un bucle infinito
- Si un algoritmo recursivo no tiene punto de salida entonces está mal definido

⟶ hay que considerar la terminación de la recursión tal que:

- **Exista una salida no recursiva para el caso base**
- **Cada invocación recursiva debe referirse a un caso mas pequeño del que fue invocado**

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD.

La siguiente función recursiva calcula x^n, usando la ecuación:

$$x^n = x * x^{n-1}$$

```
int power(int x, int n)
  {
    if (n == 0)
    return 1;
    else
    return x * power(x, n - 1);
  }
```

Otra forma:

```
int power(int x, int n){return n == 0 ? 1 : x * power(x, n - 1);} NO!
```

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Ejemplo. Fibonacci

Programa que calcula un término de la serie Fibonacci.

Los números de Fibonacci quedan definidos por las ecuaciones:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

```
#include <stdio.h>
long serieFibonacci (int);

int main()
{
    int termino=0;
    while (termino >= 0)
        {
            printf("Escriba el termino de la serie que quiere encontrar: ");
            scanf("%d", &termino);
            printf("El termino buscado es: %d\n\n", serieFibonacci(termino));
        }
   return 0;
}
```

ARQUITECTURA DE UN PROGRAMA
RECURSIVIDAD. Ejemplo. Fibonacci

La función sería:

```
long serieFibonacci (int n)
 {
      if ((n == 0) || (n == 1))
          return n;
      else
          return  serieFibonacci(n - 1) + serieFibonacci(n - 2);
  }
```