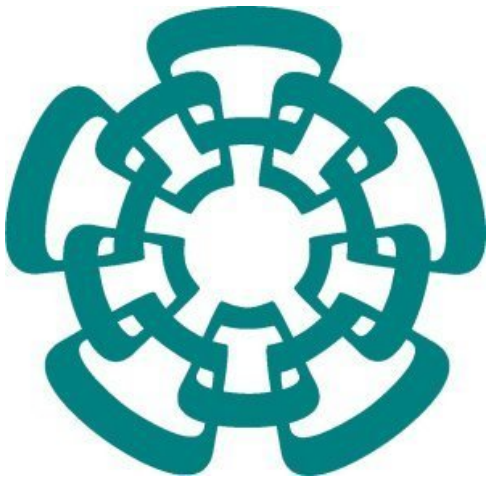
# Fundamentos de Programación - Parte 1

## Proceso de Admisión 2024



Dr. Mario Garza Fabre
Cinvestav Unidad Tamaulipas
mario.garza@cinvestav.mx

# Fundamentos de Programación

## Cinvestav Unidad Tamaulipas / Proceso de Admisión 2024

**Objetivo:**

- Estudiar los principios básicos y generales de programación, que aplican a todos los lenguajes.
- Se usará el Lenguaje C para desarrollar la lógica de programación requerida para la codificación de programas que sean interpretados correctamente por una computadora.

## Contenido:

1. Conceptos y fundamentos
    - Resolución de problemas, algoritmo, programa
    - Lenguajes de programación, Lenguaje C, estructura de un programa
    - Tipos de datos, constantes, variables, operadores
    - Bibliotecas estándar, entrada y salida
2. Arreglos y estructuras de control
    - Arreglos, estructuras de control, estructuras anidadas
3. Programación modular
    - Concepto de modularidad
    - Definición, declaración e invocación de funciones
    - Recursividad

## Prerrequisitos

Equipo de cómputo con entorno preparado para trabajar desde el inicio:

- Editor de texto básico (Notepad, Sublime, Editpad, Vi, Emacs)
- Compilador: GCC (Linux, macOS) o MinGW (Windows)
- Terminal / Línea de comandos

# Introducción

**¿Para qué programamos?**

- Resolver problemas (diferentes tipos, contextos y propósitos)

**¿Algoritmo vs Programa?**

- Algoritmo: secuencia de pasos para resolver el problema
- Programa: implementación en un *lenguaje de programación*

**Enfoque computacional para la resolución de problemas:**

- Análisis y modelado del problema
- Diseño del algoritmo
  - Pseudocódigo
  - Representación gráfica
- Implementación / codificación del programa

# Introducción

## Problema

¿Cuál es el puntaje más alto obtenido en el examen de admisión?

## Modelo (abstracción del problema)

Encontrar el valor máximo en una secuencia finita de enteros.

**Algoritmo (descripción textual):**

1. Definir primer entero de la secuencia como máximo temporal.
2. Comparar siguiente entero de la secuencia con el máximo temporal. Si es mayor, reemplazar máximo temporal con dicho entero.
3. Repetir Paso 2 si hay más enteros en la secuencia.
4. Detener cuando ya se han considerado todos los elementos de la secuencia. En este punto, el máximo temporal es el resultado.

# Introducción

## Problema

¿Cuál es el puntaje más alto obtenido en el examen de admisión?

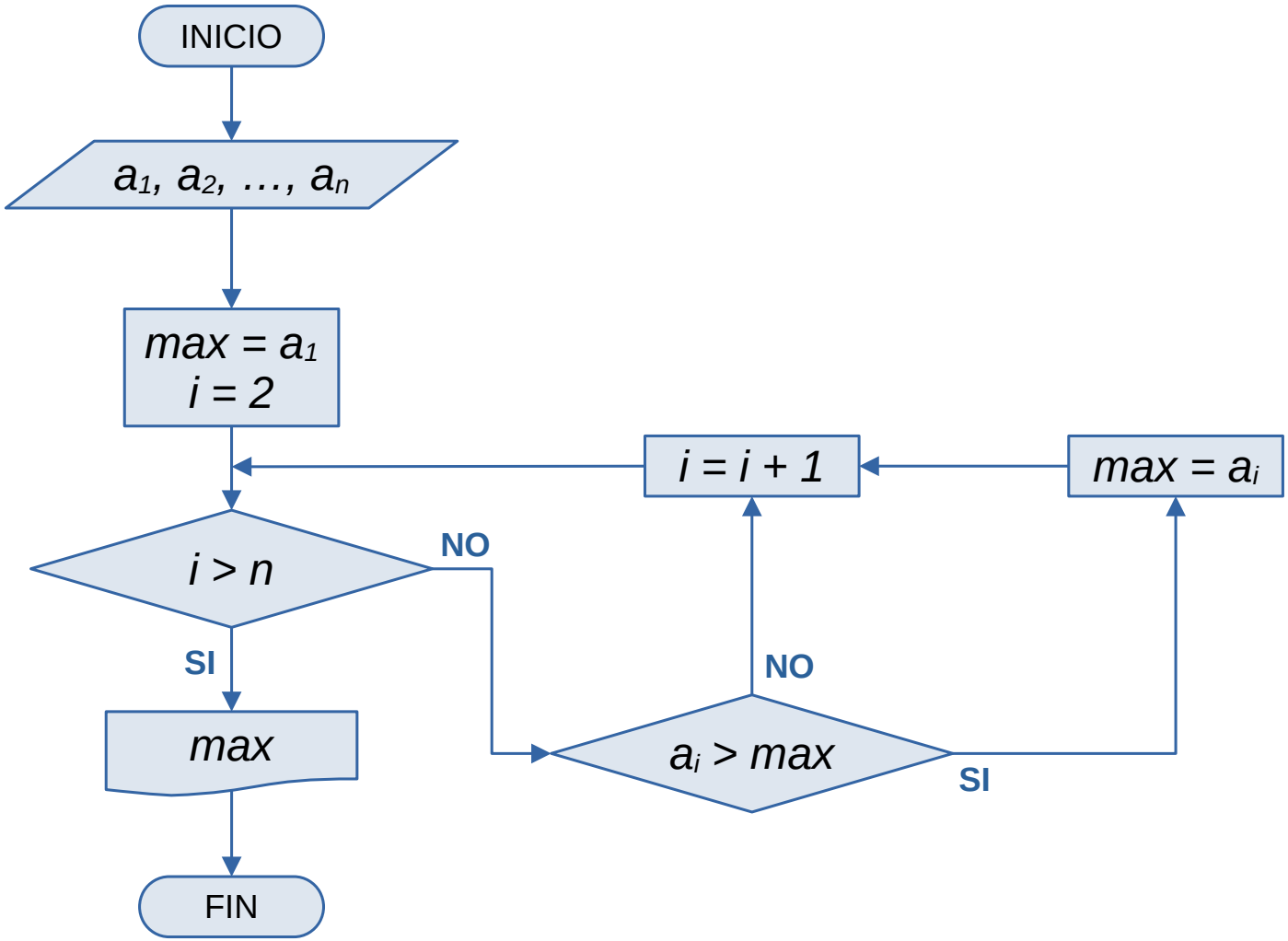
## Modelo (abstracción del problema)

Encontrar el valor máximo en una secuencia finita de enteros.

**Algoritmo (pseudocódigo):**

1. **procedure** $max(a_1, a_2, \cdots, a_n :$ integers$)$
2. $temp\_max := a_1$
3. **for** $i := 2$ **to** $n$
4. **if** $temp\_max < a_i$ **then** $temp\_max := a_i$
5. $\{temp\_max$ is the largest element$\}$

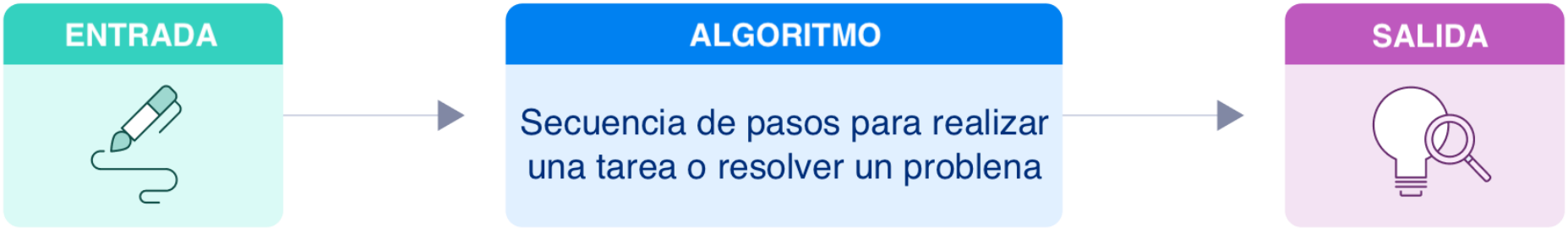
## Algoritmo (representación gráfica, diagrama de flujo):



## Simbología básica de los diagramas de flujo

| SÍMBOLO | NOMBRE | FORMA | DESCRIPCIÓN |
|---|---|---|---|
|  | Línea de flujo | Línea apuntando a otro símbolo | Representa el orden de operación de los procesos. Indicar el sentido de las operaciones dentro del flujo. |
|  | Terminal | Óvalo | Representar el inicio o el fin de un algoritmo. También puede representar una parada o una interrupción programada que sea necesaria realizar en un programa. |
|  | Proceso | Rectángulo | Representar una instrucción, o cualquier tipo de operación que origine un cambio de valor, forma o ubicación de datos. |
|  | Decisión | Rombo | Representar una instrucción, o cualquier tipo de operación que origine un cambio de valor, forma o ubicación de datos. |
|  | Entrada | Paralelogramo | Indica la entrada o salida de información. El proceso de hacer entrar datos en la forma de ingresar datos. |
|  | Salida | Hoja de papel impresa | Representa el proceso de hacer salir datos, en forma de mostrar los resultados. |

# Introducción




**Características de un buen algoritmo:**

- **Correctitud:** para cada entrada produce la salida deseada (correcta)
- **Precisión:** sus pasos están claramente definidos (no ambiguos)
- **Unicidad:** resultado de cada paso está únicamente definido por la entrada y resultados de pasos anteriores
- **Finitud:** termina después de un número finito de pasos
- **Generalidad:** aplicable a todas las posibles entradas (instancias)
- **Eficiencia:** complejidad baja (temporal y espacial), de ser posible

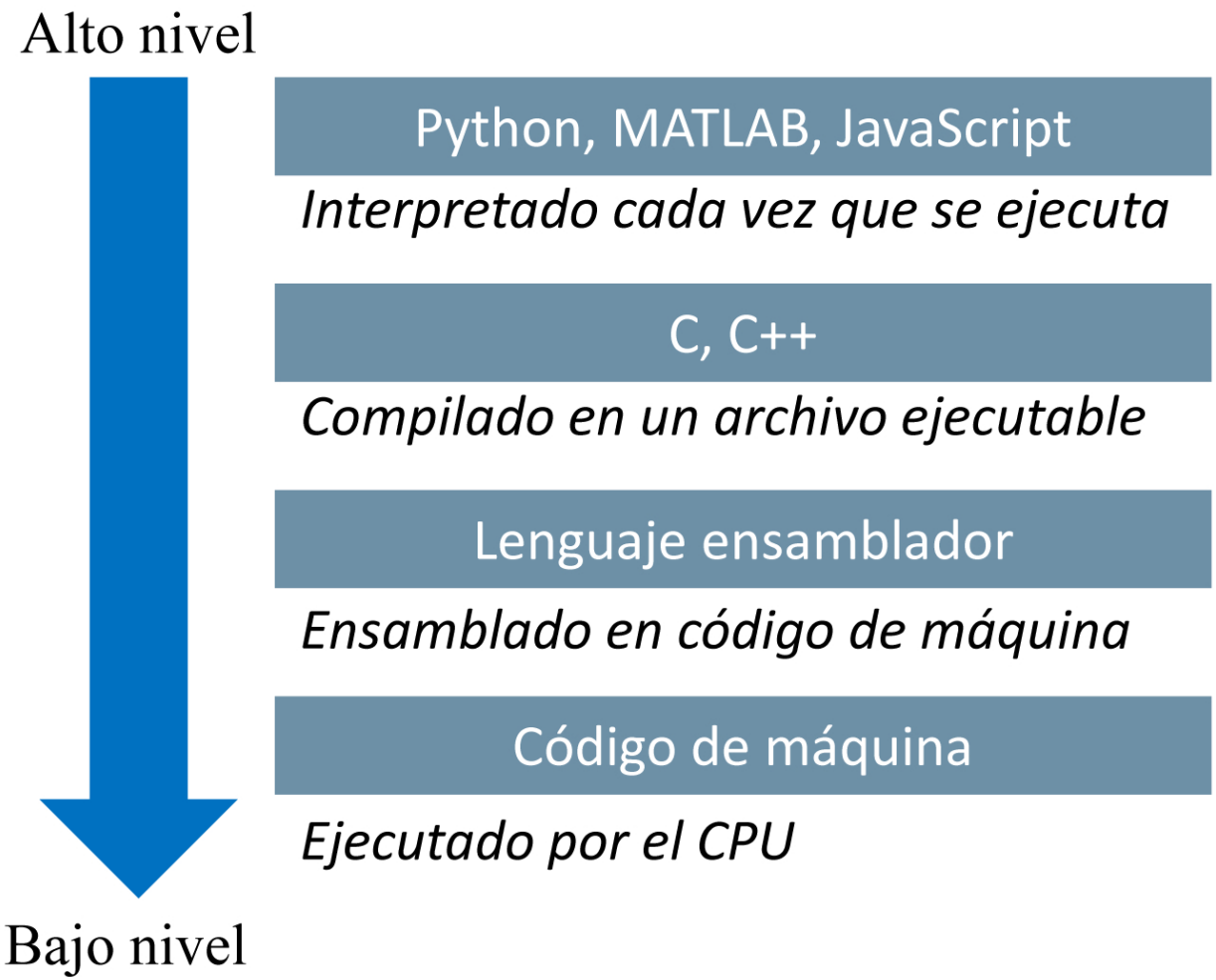
# Introducción

## Lenguaje de programación

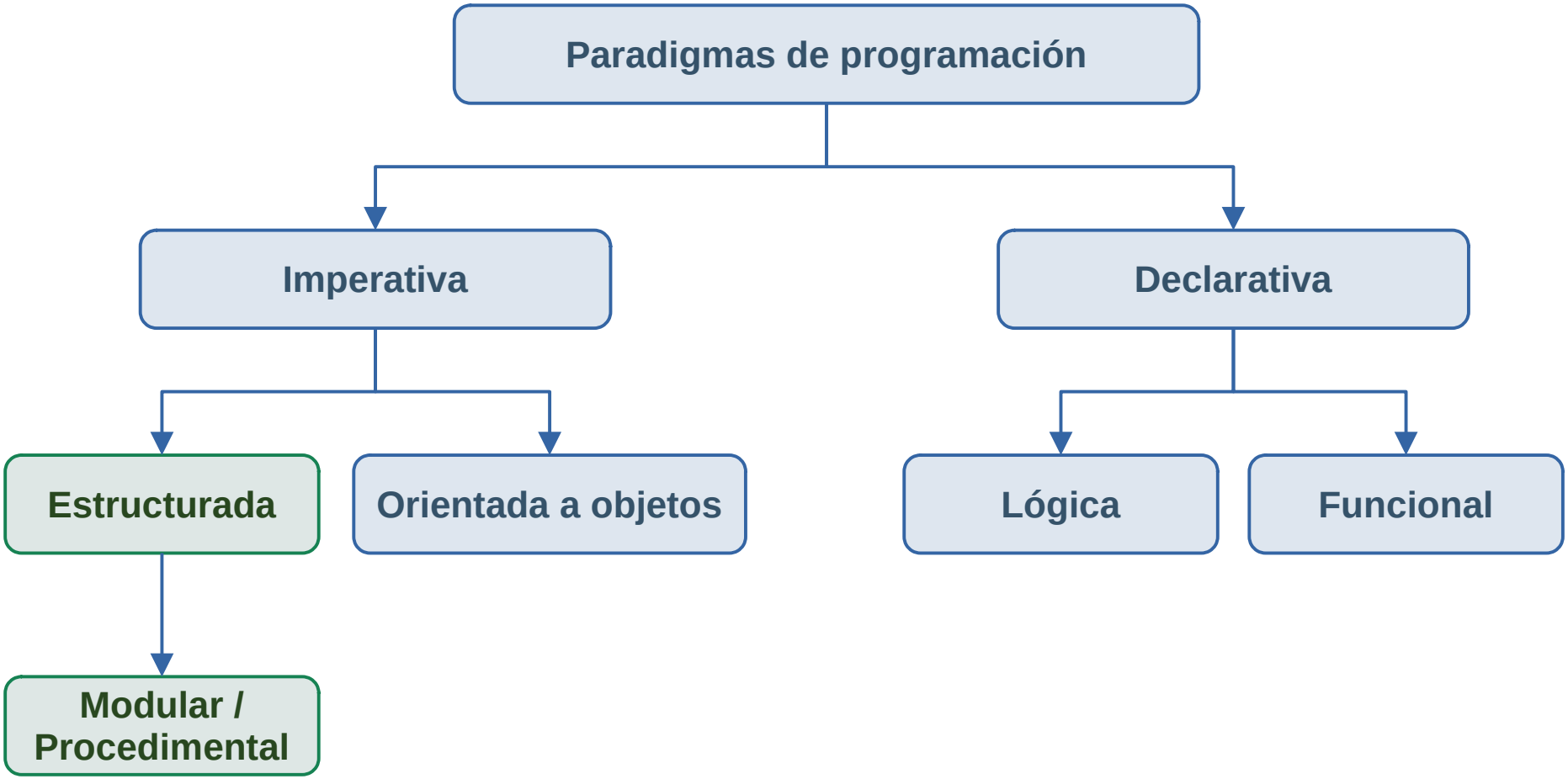
Lenguaje formal (con reglas bien definidas) para la escritura de instrucciones (algoritmos) en un formato que facilite su ejecución por una computadora (no necesariamente de manera directa).

# Introducción

## Nivel de abstracción de los lenguajes de programación

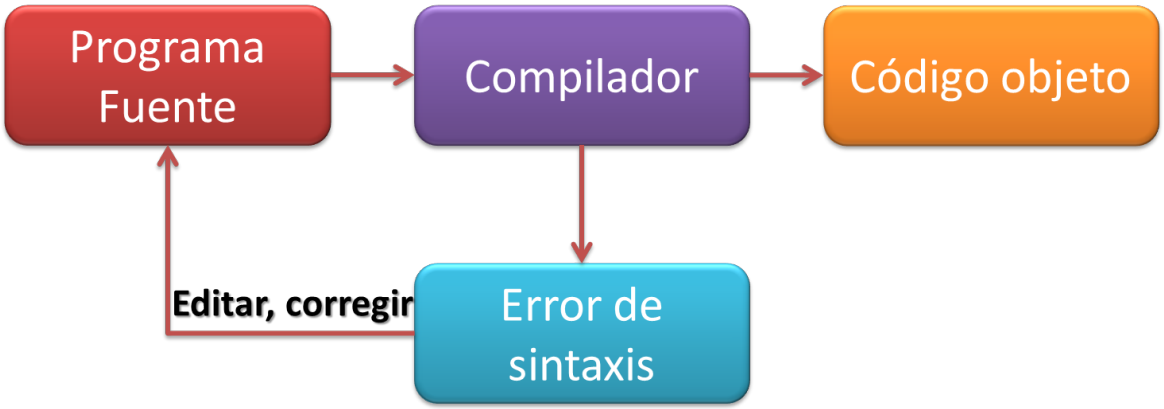


## Paradigmas de programación



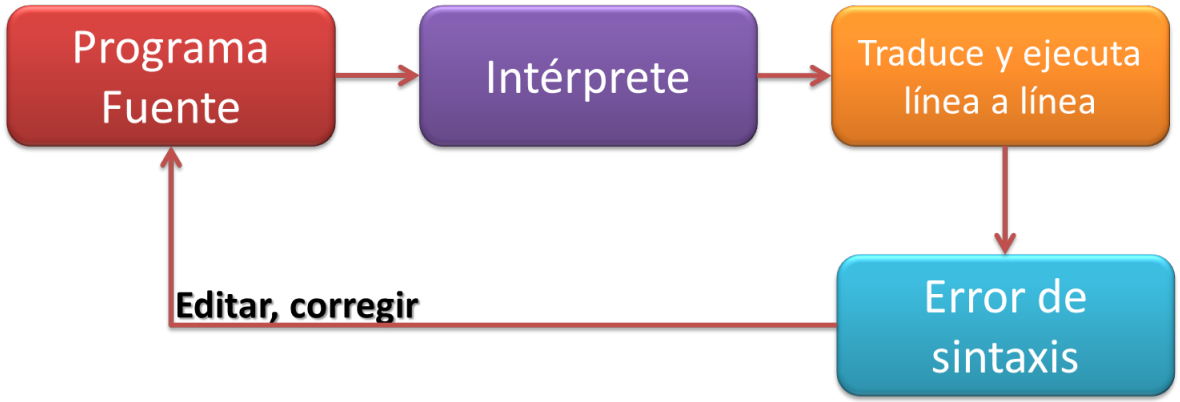
# Introducción

## Lenguajes de programación compilados e interpretados

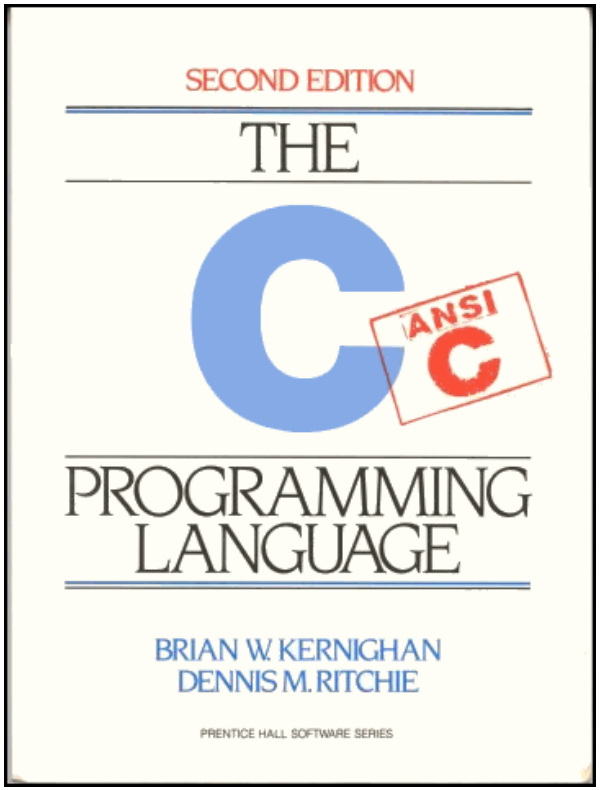


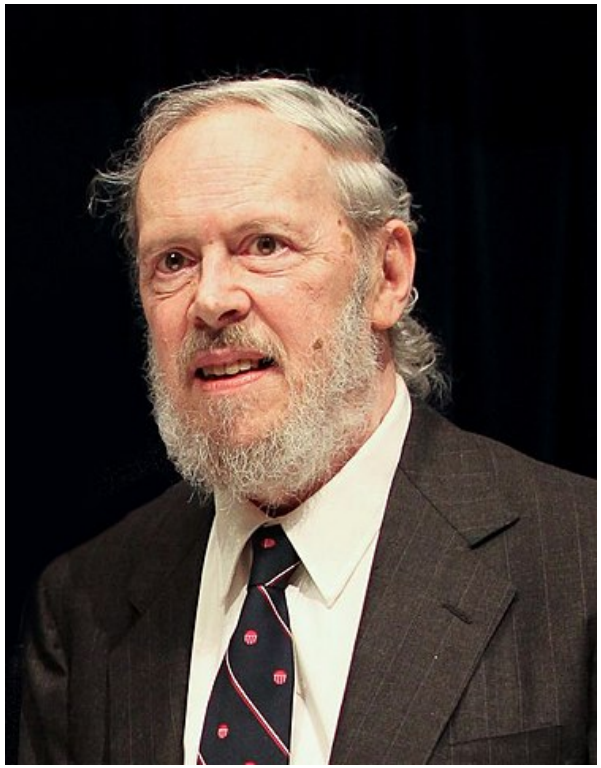

- C / C++
- Java
- Go
- ...

- Python
- Ruby
- Matlab
- ...



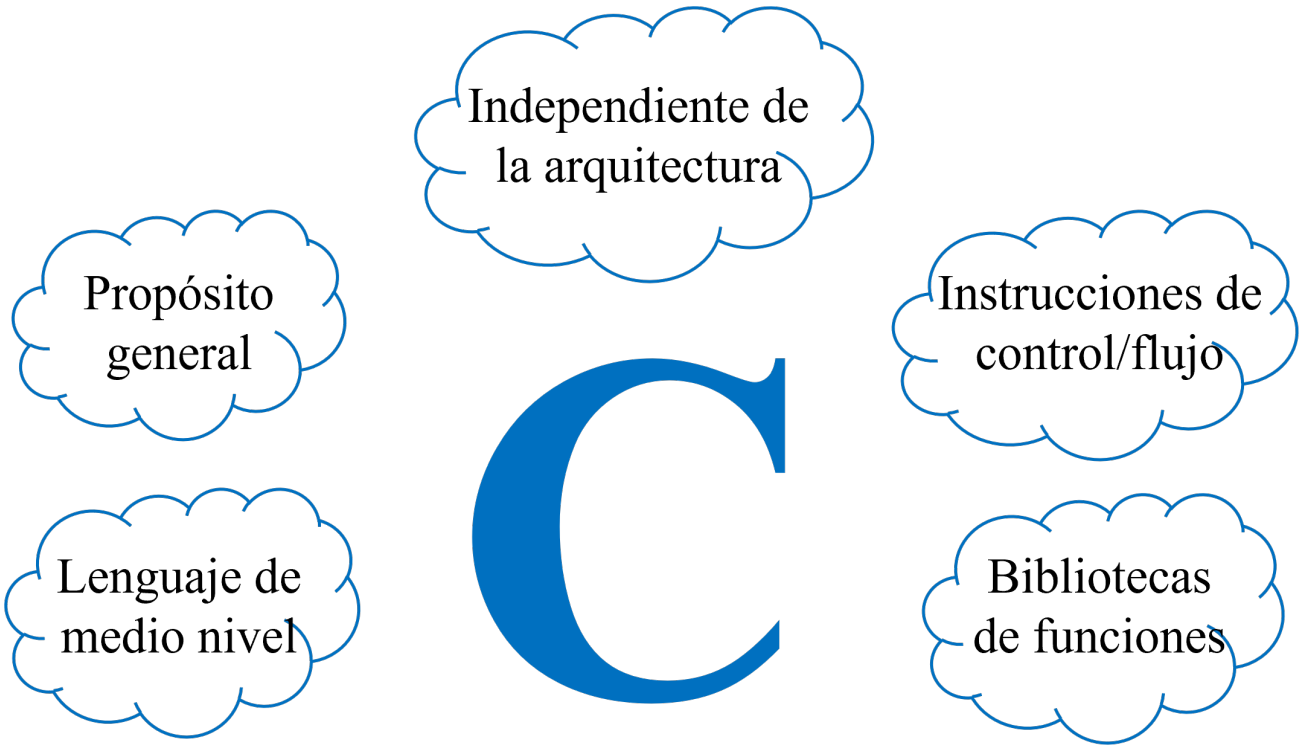
# Lenguaje C (ANSI C)



El **Lenguaje C** fue desarrollado por **Dennis Ritchie** (1941-2011) en los Laboratorios Bell entre 1972 y 1973. De acuerdo con el índice TIOBE (`https://www.tiobe.com/tiobe-index`), **en mayo de 2024**, el Lenguaje C ocupa el segundo lugar de los lenguajes de programación más populares.

# Lenguaje C (ANSI C)




## ANSI C

Estándar publicado por el *American National Standards Institute* (ANSI).

# Estructura de un programa en C

```c
/* Programa que calcula el area de un circulo
e imprime el resultado en pantalla. */

// Directivas del preprocesador
#include <stdio.h>  // Biblioteca estandar
#define PI 3.141592 // Definition ("Constant")

// Declaracion global
float radius = 0.8;

// Funcion principal
int main()
{
  // Declaracion local / Instruccion
  float area = PI * radius * radius;

  // Salida (impresion de resultados)
  printf("Area del circulo: %.4f\n", area);

  return 0;
}
```

# Estructura de un programa en C

- **Documentación (comentarios):**
  - Una sola línea: `//` y Bloque de líneas: `/* ... */`
- **Directivas del preprocesador (`#include`, `#define`, ...):**
  - Cabeceras, inclusión de bibliotecas, definiciones, ...
  - Procesamiento del código, previo a la compilación
- **Declaraciones globales:**
  - Elementos accesibles desde cualquier parte del programa (variables, constantes, funciones, ...)
- **Función principal:**
  - Función obligatoria, donde la ejecución comienza

# Estructura de un programa en C

- **Declaraciones locales:**
  - Alcance / ámbito limitado (variables, constantes, . . . )
- **Instrucciones / sentencias / estructuras de control:**
  - Cálculos, invocación de funciones / procedimientos, . . .
  - Finalizar una instrucción/sentencia con “;” (punto y coma)
  - Estructuras selectivas y repetitivas
- **Entrada y salida:**
  - Entrada de datos (por parte del usuario, . . . )
  - Salida de resultados (impresión a pantalla, . . . )

# Estructura de un programa en C

## Compilación y ejecución

- El archivo de código fuente tiene **extensión .c**
- Compilación con **gcc** desde línea de comandos:

```
gcc archivo_fuente.c -o ejecutable
```

- **Ejecución (Linux / MacOS):**

```
./ejecutable
```

- **Ejecución (Windows):**

```
ejecutable.exe
```

# Variables, constantes y tipos de datos

## Variable

**Espacio de memoria** cuyo tipo especifica la cantidad de memoria necesaria para guardar información.

- Cada variable debe declararse antes de ser utilizada
- Su valor puede cambiar a lo largo de la ejecución de un programa
- Características:
  - Nombres formados por letras y dígitos
  - El primer símbolo del nombre siempre es una letra o guión bajo '_'
  - Sensible a capitalización:
    - Letras mayúsculas y minúsculas son diferentes
    - `x` y `X` son nombres de variables diferentes

## Palabras reservadas

Identificadores predefinidos que no se pueden usar como nombres de variables:

| | | | |
|---|---|---|---|
| `auto` | `double` | `int` | `struct` |
| `break` | `else` | `long` | `switch` |
| `case` | `enum` | `register` | `typedef` |
| `char` | `extern` | `return` | `union` |
| `const` | `float` | `short` | `unsigned` |
| `continue` | `for` | `signed` | `void` |
| `default` | `goto` | `sizeof` | `volatile` |
| `do` | `if` | `static` | `while` |

# Variables, constantes y tipos de datos

## Declaración de variables

- Enunciar el nombre de la variable y asociarle un tipo:

```
[tipo_C] identificador;
```

donde

  - `[tipo_C]` determina el tipo de datos de la variable
  - `identificador` indica el nombre de la variable

- Una variable puede declararse en diferentes lugares:

  1. Fuera de todas las funciones (variable global)
  2. Dentro de una función (variable local)
  3. En la definición de parámetros de una función (variable local)

# Variables, constantes y tipos de datos

¿Cuál es el resultado esperado para este programa?

```c
#include <stdio.h>

int x = 9;

void funcion( int x )
{
    int y = 5;
    x *= 2;
    printf("x = %d, y = %d\n", x, y);
}

int main()
{
    int y = 1;
    funcion( x );
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

# Variables, constantes y tipos de datos

**Constante**

Valor que, una vez fijado por el compilador, no cambia durante la ejecución del programa.

- Puede ser un número, un carácter o una cadena de caracteres
- Se pueden definir "constantes" de dos formas:
  1. Usando la palabra clave `const`
  2. Usando la directiva `#define`
- **¿Cuál es la diferencia?**

# Variables, constantes y tipos de datos

```c
#include <stdio.h>

const int val = 10;
const float floatVal = 4.5;
const char charVal = 'G';
const char stringVal[10] = "ABC";

int main()
{
    printf("Constante entera: %d\n",val);
    printf("Constante de punto flotante: %.1f\n",floatVal);
    printf("Constante caracter: %c\n",charVal);
    printf("Cadena constante: %s\n",stringVal);

    return 0;
}
```

```
$ gcc 04_const_const.c -o 04_const_const
$ ./04_const_const
Constante entera: 10
Constante de punto flotante: 4.5
Constante caracter: G
Cadena constante: ABC
```

# Variables, constantes y tipos de datos

```c
#include <stdio.h>

#define val  10
#define floatVal  4.5
#define charVal 'G' // Comillas simples
#define stringVal "ABC" // Comillas dobles

int main()
{
    printf("Constante entera: %d\n",val);
    printf("Constante de punto flotante: %.1f\n",floatVal);
    printf("Constante caracter: %c\n",charVal);
    printf("Cadena constante: %s\n",stringVal);
    printf("Size of int: %lu\n",sizeof(int));

    return 0;
}
```

```
$ gcc 05_const_define.c -o 05_const_define
$ ./05_const_define
Constante entera: 10
Constante de punto flotante: 4.5
Constante caracter: G
Cadena constante: ABC
```

# Variables, constantes y tipos de datos

## Tipos de datos básicos

- `char` – un sólo byte capaz de almacenar un carácter
- `int` – un entero, típicamente refleja el tamaño natural de los enteros en el sistema huésped
- `float` – punto flotante de precisión simple, no tiene más de siete dígitos significativos
- `double` – punto flotante de doble precisión, puede tener hasta 16 dígitos significativos
- `void` – se utiliza para declarar funciones que no retornan un valor

# Variables, constantes y tipos de datos

## Tipos de datos básicos y modificadores

| Tipo | Modificador | | Rango | | Formato | Bytes |
|---|---|---|---|---|---|---|
| | | | Mínimo | Máximo | | |
| `char` | | `unsigned` | $0$ | $255$ | `%c` | $1$ |
| | | `signed` | $-128$ | $127$ | `%c` | $1$ |
| `int` | `short` | `unsigned` | $0$ | $65,535$ | `%u` | $2$ |
| | | `signed` | $-32,768$ | $32,767$ | `%d` | $2$ |
| | `long` | `unsigned` | $0$ | $4,294,967,295$ | `%lu` | $4$ |
| | | `signed` | $-2,147,483,648$ | $2,147,483,647$ | `%ld` | $4$ |
| `float` | | | $-3.40 \times 10^{38}$ | $3.40 \times 10^{38}$ | `%f` | $4$ |
| `double` | | | $-1.79 \times 10^{308}$ | $1.79 \times 10^{308}$ | `%lf` | $8$ |
| | `long` | | $-1.18 \times 10^{4932}$ | $1.18 \times 10^{4932}$ | `%Lf` | $16$ |

Los **modificadores** opcionales que pueden acompañar a los tipos básicos son: `signed`, `unsigned`, `long` y `short`, respectivamente.

La función `sizeof()` permite conocer el tamaño (en bytes) de cada tipo.

# Variables, constantes y tipos de datos

```c
#include <stdio.h>

int main()
{

    printf("Size of int: %lu bytes\n", sizeof(int));

    return 0;
}
```

```
$ gcc 06_sizeof.c -o 06_sizeof
$ ./06_sizeof
Size of int: 4 bytes
```

## Ejercicio

Extender este programa para investigar y confirmar el tamaño de todos los tipos de datos, con y sin modificadores (tabla en diapositiva anterior).

# Funciones de entrada/salida

- `printf` – escribe bytes en el `stdout` (pantalla), usa formatos específicos:

```
printf("Hola mundo \n");
printf("Imprimir %c %d %f \n",'a',28,3.0e+8);
```

- `scanf`– ingresa bytes (caracteres ASCII) desde el `stdin` (teclado), usa formatos y argumentos específicos:

```
scanf("%f",&flotante);
scanf("%c\n",&caracter);
scanf("%f %d %c",&flotante, &entero, &caracter);
scanf("%s",cadena); //No lleva &
scanf("%ld",&entero_largo);
```

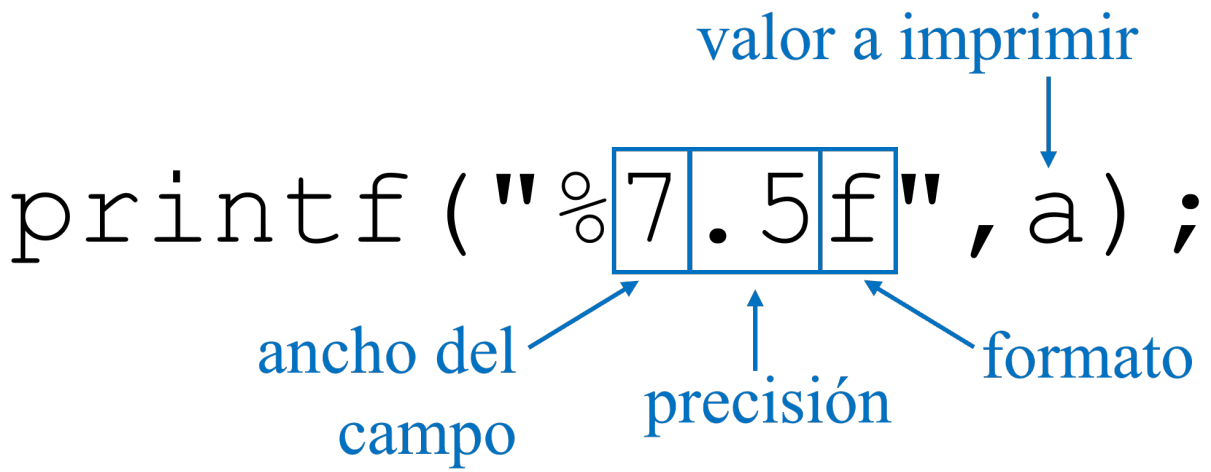
# Funciones de entrada/salida

| Tipo | Formato | Salida |
|---|---|---|
| Numérico | %d | (int) entero con signo base 10 |
| | %i | (int) entero con signo base 10 |
| | %u | (int) entero sin signo base 10 |
| | %o | (int) entero sin signo base 8 |
| | %x | (int) entero sin signo base 16 (0,1,...,9,a,b,...,f) |
| | %X | (int) entero sin signo base 16 (0,1,...,9,A,B,...,F) |
| | %f | (double) valor con signo de la forma [−]dddd.dddd |
| | %e | (double) valor con signo de la forma [−]d.ddde[±]ddd |
| | %E | (double) valor con signo de la forma [−]d.dddE[±]ddd |
| | %g | (double) valor con signo en formato f ó e |
| | %G | (double) igual que G, pero introduce E en vez de e |
| Carácter | %c | carácter simple |
| | %s | cadena de caracteres |
| | %% | imprime el carácter de porcentaje ( %) |

Especificadores de formato para la función `printf`

# Funciones de entrada/salida

**Especificador de precisión para** `printf` – combina un punto y un entero para indicar el número de posiciones decimales para los números de punto flotante y se puede colocar justo después del **especificador del ancho mínimo del campo**




Por ejemplo, `%7.5f` asegura que la longitud del ancho mínimo del campo sea de 7 caracteres con 5 posiciones decimales

# Funciones de entrada/salida

- Con el especificador %s, `scanf` leerá caracteres hasta encontrar un espacio, tabulador, o un retorno de carro (*enter*)
- La cadena de caracteres se guarda en un arreglo que debe ser lo suficientemente grande como para almacenarla
- Al final añade automáticamente el carácter nulo '\0'

```
scanf("%s",a);
```

a=

| 'h' | 'o' | 'l' | 'a' | '\0' |
|---|---|---|---|---|

# Operadores y expresiones


Los **operadores** son símbolos que indican la forma en cómo se manipulan los datos. Se aplican a variables u otros objetos denominados **operandos**.

# Operadores aritméticos

| Operador | Operación | Descripción | Ejemplo |
|---|---|---|---|
| + | Suma | Los operandos pueden ser enteros o flotantes | 5+2→7 |
| - | Resta | | 5-2→3 |
| * | Multiplicación | | 5*2→10 |
| / | División | | 5/2→2.5 |
| % | Módulo | Los operandos deben ser enteros | 5 %2→1 |

```c
#include <stdio.h>

int main()
{
  int a = 10, b = 3, c;
  float w = 2.0, x = 3.77, y, z;

  c = w - x; // Resultado es -1 (entero)
  y = a + b; // Resultado es 13.0 (flotante)
  z = a * x; // Resultado es 37.7 (flotante)
  printf("c = %d, y = %.2f, z = %.2f\n", c, y, z);

  return 0;
}
```

# Operadores lógicos y relacionales

| Tipo | Operador | Operación |
|---|---|---|
| Unario | ! | Negación (NOT) |
| Binario | && | Conjunción (AND) |
| | \|\| | Disyunción (OR) |

| Operador | Operando | Ejemplo a=1 y b=2 |
|---|---|---|
| > | mayor que | a>b→0 |
| >= | mayor o igual | a>=b→0 |
| < | menor que | a<b→1 |
| <= | menor o igual que | (a+1)<=b→1 |
| == | igual que | a==b→0 |
| != | diferente que | (a+b)!=3→0 |

- Los operadores relacionales se utilizan para formar expresiones lógicas
- Los operadores lógicos actúan sobre operandos lógicos
- El resultado de estos operadores son los valores lógicos/enteros `FALSO` (**0**) y `VERDADERO` (**1**)

# Ejercicio

**Ejercicio 1 (20 minutos)**

# Operadores para manejo de bits

| Operador | Descripción |
|---|---|
| & | Operación AND a nivel de bits |
| \| | Operación OR a nivel de bits |
| ^ | Operación XOR a nivel de bits |
| << | Desplazamiento a la izquierda |
| >> | Desplazamiento a la derecha |
| ~ | Complemento a uno (unario) |

| p | q | p&q | p\|q | p^q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Los operadores para manejo de bits ejecutan operaciones bit-a-bit. Solo pueden ser aplicados a datos de tipo entero (`char`, `int`, `short`, `long`).

# Operadores para manejo de bits

```c
#include <stdio.h>

int main()
{
    unsigned char a = 5; // a = 00000101
    unsigned char b = 9; // b = 00001001

    printf("a = %u, b = %X\n", a, b);
    printf("a&b = %u, a&b = %X\n", a&b, a&b); // 00000001 (1, 0x1)
    printf("a|b = %u, a|b = %X\n", a|b, a|b); // 00001101 (13, 0xD)
    printf("a^b = %u, a^b = %X\n", a^b, a^b); // 00001100 (12, 0xC)
    a = ~a;
    printf("~a = %u, ~a = %X\n", a, a); // 11111010 (250, 0xFA)
    printf("b<<1 = %u, b<<1 = %X\n", b<<1, b<<1); // 00010010 (18, 0
    x12)
    printf("b>>1 = %u, b>>1 = %X\n", b>>1, b>>1); // 00000100 (4, 0x4
    )

    return 0;
}
```

# Operadores de asignación

| Operador | Descripción | Equivalente |
|---|---|---|
| `x=y` | Asignación | `x=y` |
| `x+=y` | Suma y asignación | `x=x+y` |
| `x-=y` | Resta y asignación | `x=x-y` |
| `x*=y` | Multiplicación y asignación | `x=x*y` |
| `x/=y` | División y asignación | `x=x/y` |
| `x%=y` | Módulo y asignación | `x=x%y` |
| `x<<=y` | Desplazamiento a la izquierda y asignación | `x=x<<y` |
| `x>>=y` | Desplazamiento a la derecha y asignación | `x=x>>y` |
| `x&=y` | AND bit-a-bit y asignación | `x=x&y` |
| `x\|=y` | OR bit-a-bit y asignación | `x=x\|y` |
| `x^=y` | XOR bit-a-bit y asignación | `x=x^y` |

# Operadores de incremento y decremento

- El operador de incremento '++' suma 1 a su operando, mientras que el operador de decremento '--' sustrae 1
- Pueden ser prefijos (antes de la variable, `++x`) o sufijos (después de la variable, `x++`)
- **¿Mismo resultado si usamos `++x` o `x++`?**
- Por ejemplo, si `n=1`, entonces

```
x = n++; // asigna e incrementa (x=1 y n=2)
```

y en el caso

```
x = ++n; // incrementa y asigna (x=2 y n=2)
```

# Precedencia y asociatividad

| Nivel | Precedencia | Asociatividad |
|---|---|---|
| 1 | `()` | izquierda a derecha |
| 2 | `* / %` | izquierda a derecha |
| 3 | `+ -` | izquierda a derecha |
| 4 | `<< >>` | izquierda a derecha |
| 5 | `< <= > >=` | izquierda a derecha |
| 6 | `== !=` | izquierda a derecha |
| 7 | `&` | izquierda a derecha |
| 8 | `^` | izquierda a derecha |
| 9 | `\|` | izquierda a derecha |
| 10 | `&&` | izquierda a derecha |
| 11 | `\|\|` | izquierda a derecha |
| 12 | `= += -= *= /= %= &= ^= \|= <<= >>=` | derecha a izquierda |

**Precedencia:** prioridad de operaciones en expresiones que contienen más de un operador. **Asociatividad:** con operadores de misma prioridad, especifica si un operando se agrupa con el de su izquierda/derecha.

# Precedencia y asociatividad

- Las operaciones matemáticas tienen un orden de precedencia. Usar paréntesis en lugares diferentes puede alterar los resultados.
- Por ejemplo:

**Desarrollo 1**:

```
a = 12/3+2*2-1
  = 4+4-1
  = 8-1
  = 7
```

**Desarrollo 2**:

```
a = 12/(3+2)*2-1
  = 12/5*2-1
  = 2.4*2-1
  = 4.8-1
  = 3.8
```

**Desarrollo 3**:

```
a = (12/3)+2*(2-1)
  = 4+2*1
  = 4+2
  = 6
```

# Estructuras y sentencias de control

Las *estructuras de control* permiten modificar el flujo de ejecución de las instrucciones de un programa. Existen tres tipos de estructuras de control:

1. **Secuenciales** – ejecución sucesiva de dos o más operaciones
2. **Selectivas** – se realiza una u otra operación dependiendo de una condición
3. **Iterativas** – repetición de una operación mientras no se cumpla una condición

# Estructuras selectivas: `if-else`

- Se toma una decisión referente a la acción a ejecutar en un programa, basándose en el resultado lógico (`VERDADERO` o `FALSO`) de una expresión:

```
if (expresión)
        sentencia1;
else
        sentencia2;
```

- Si `expresión` es `VERDADERO`, entonces ejecutar `sentencia1`
- Si `expresión` es `FALSO`, entonces ejecutar `sentencia2`
- La cláusula `else` es opcional. Si `expresión` es `FALSO` y `else` se ha omitido, entonces ignorar `sentencia1`

# Estructuras selectivas: `if-else`

Es posible anidar las sentencias `if-else`, esto es, escribir una secuencia `if-else` dentro de otra secuencia `if-else`:

```
if (expresión1)
  if (expresión2)
    sentencia1;
  else
    sentencia2;
```

| expr1 | expr2 | sent1 | sent2 |
|---|---|---|---|
| F | F | no | no |
| F | V | no | no |
| V | F | no | si |
| V | V | si | no |

---

```
if (expresión1)
{
  if (expresión2)
    sentencia1;
}
else
  sentencia2;
```

| expr1 | expr2 | sent1 | sent2 |
|---|---|---|---|
| F | F | no | si |
| F | V | no | si |
| V | F | no | no |
| V | V | si | no |

# Estructuras selectivas: `else if`

- La construcción `else if` es utilizada para escribir programas con múltiples opciones de decisión
- Las `expresiones` son evaluadas en orden; si cualquiera de ellas es verdadera, la `sentencia` asociada se ejecuta y esto termina toda la cadena de opciones
- El último `else` es para indicar 'ninguna de las anteriores se satisfizo'

```
if (expresión1)
  sentencia1;
else if (expresión2)
  sentencia2;
else if (expresión3)
  sentencia3;
.
.
.
else
  sentenciaN;
```

# Estructuras selectivas: else if

```c
#include <stdio.h>

int main(void)
{
   char ar[20];
   int cc;
   float pu;

   printf("Nombre del articulo: "); scanf("%s", ar);
   printf("Cantidad comprada: "); scanf("%d", &cc);
   printf("Precio unitario: "); scanf("%f", &pu);

   printf("\n\n%20s\t%10s\t%10s\t%10s\t%10s\n",
          "Articulo","Cantidad","P.U.","%","Total");
   printf("%20s\t%10d\t%10.2f", ar, cc, pu);

   if (cc > 100)
     printf("\t%9d%%\t%10.2f\n\n", 40, cc*pu*0.6);
   else if (cc >= 25)
     printf("\t%9d%%\t%10.2f\n\n", 20, cc*pu*0.8);
   else if (cc >= 10)
     printf("\t%9d%%\t%10.2f\n\n", 10, cc*pu*0.9);
   else
     printf("\t%10s\t%10.2f\n\n", "--", cc*pu);

   return 0;
}
```

# Estructuras selectivas: `switch`

- La sentencia `switch` trata con múltiples opciones para probar si una expresión coincide con alguno de varios valores constantes
- Cada `case` está etiquetado con un valor entero constante o una expresión constante
- El `default` es opcional; si se omitió y ninguno de los casos coincide, ninguna acción se ejecuta
- Después que el código de un `case` se realizó, se debe indicar una acción explícita de escape con `break` o `return`

```
switch (expresion) {
  case const-expr1:
    setencia1;
  case const-expr2:
    setencia2;
  .
  .
  .
  default:
    setenciaN;
}
```

# Estructuras selectivas: switch

```c
#include <stdio.h>
int main( )
{
  char letra;
  printf("Introduzca una letra: "); scanf("%c",&letra);

  switch(letra) {
    case 'a':
    case 'A':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'e':
    case 'E':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'i':
    case 'I':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'o':
    case 'O':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'u':
    case 'U':
      printf("La letra %c es vocal\n",letra);
      break;
    default:
      printf("La letra %c es consonante\n",letra);
      break;
  }

  return 0;
}
```

# Ejercicio

**Ejercicio 2 (20 minutos)**

# Expresión condicional

La **expresión condicional**, provee una alternativa para escribir construcciones condicionales de forma compacta. Por ejemplo:

```
if (a > b)
        x = a;
else
        x = b;
```

puede escribirse alternativamente como:

```
x = (a > b) ? a : b; /* x = max(a, b) */
```

# Expresión condicional

```c
#include <stdio.h>
#define MIN(x, y) ((x < y) ? x : y) // Define funcion

int main()
{
  float a, b, c, menor;

  printf("Introduce tres numeros a, b y c: ");
  scanf("%f %f %f", &a, &b, &c);

  if (a < b)
    menor = MIN(a,c);
  else
    menor = MIN(b,c);

  printf("El menor es %f\n", menor);

  return 0;
}
```

# Estructuras iterativas: `for`

La estructura **for** permite definir ciclos controlados por contador:

```
for (inicialización; condición; progresión)
    sentencia;
```

- **inicialización** define el estado inicial de un contador
- **condición** es una expresión lógica que define un criterio para realizar las repeticiones; si es `VERDADERA`, ejecuta la `sentencia`, se evalúa la expresión en **progresión**, y vuelve a evaluarse la **condición**; si es `FALSA`, finalizan las repeticiones y continua con la siguiente sentencia del programa
- **progresión** es una expresión que se utiliza para modificar el valor del contador en forma incremental o decremental

# Estructuras iterativas: `for`

En el siguiente ejemplo se imprimen los números del 1 al 100.

```c
#include <stdio.h>

int main()
{

    int n;
    for (n = 1; n <= 100; n++)
        printf("%3d\n", n);

    return 0;
}
```

Literalmente dice: desde `n` igual a `1`, mientras `n` sea menor o igual que 100, con incrementos de `1`, imprime el valor de `n`.

# Estructuras iterativas: `while`

- La sentencia **while** se ejecutará mientras el valor de una condición sea verdadera:

```
while (condición)
    sentencia;
```

- **condición** puede ser una expresión relacional o lógica. Si es `VERDADERA`, se ejecuta la sentencia y se vuelve a evaluar la **condición**; si es `FALSA`, la sentencia no se ejecuta y el programa continua
- La **condición** se evalúa primero, de modo que la `sentencia` puede nunca ejecutarse si la `condición` es `FALSA`
- Se debe incluir algún elemento que altere el valor de `condición`, proporcionando la salida del ciclo

# Estructuras iterativas: `while`

En el siguiente ejemplo se imprimen los números del 1 al 100.

```c
#include <stdio.h>

int main()
{

    int n = 1;
    while (n <= 100) {
        printf("%3d\n", n);
        n++;
    }

    return 0;
}
```

Literalmente dice: mientras `n` sea menor o igual que 100, imprime el valor de `n` e incrementa en 1 el valor de `n`.

# Estructuras iterativas: `do-while`

- La sentencia `do-while` es similar al `while` excepto que la condición se evalúa al final del ciclo y no al inicio, lo que **obliga a que se ejecute la** `sentencia` **por lo menos una vez**:

```
do
   sentencia;
while (condición);
```

- Se debe incluir algún elemento que altere el valor de `condición` para proporcionar eventualmente la salida del ciclo

# Estructuras iterativas: `do-while`

En el siguiente ejemplo se imprimen los números del 1 al 100.

```c
#include <stdio.h>

int main()
{

    int n = 1;
    do {
        printf("%3d\n", n);
        n++;
    } while(n<=100);

    return 0;
}
```

Literalmente dice: imprime el valor de `n` e incrementa su valor en 1 mientras `n` sea menor o igual que 100.

# Ciclos anidados

Con las sentencias `for`, `while` y `do-while` se pueden formar **ciclos anidados:** un ciclo interno se ejecutará totalmente en cada iteración del ciclo que lo contiene.

```
while(condición) {

  while(condición) {
    sentencia(s);
  }
  sentencia(s);
}
```

```
do {
  sentencia(s);

  do {
    sentencia(s);
  }while(condición);

}while(condición);
```

```
for (init;cond;incr) {

  for (init;cond;incr)
    sentencia(s);
  }
  sentencia(s);
}
```

**¡Los ciclos externos e internos pueden ser de diferente tipo!**

# Sentencias `break` y `continue`

```
while(expresión){
   sentencia(s);
   if (condición){
      break;
   }
   sentencia(s);
}
sentencia(s);
```

```
do{
   sentencia(s);
   if (condición){
      break;
   }
   sentencia(s);
}while(expresión);
sentencia(s);
```

```
for(init;condición;progreso){
   sentencia(s);
   if (condición){
      break;
   }
   sentencia(s);
}
sentencia(s);
```

La sentencia **break** provee una salida prematura de las estructuras `for`, `while` y `do-while`. Se usa casi siempre con una sentencia `if-else` dentro del ciclo.

# Sentencias break y continue

```c
#include <stdio.h>

int main()
{
   int i=1;
   float num, suma = 0.0;
   do {

      printf("Ingresa n%d: ",i);
      scanf("%f",&num);
      i++;
      if (num < 0)
        break;
      suma += num ;

   } while(i<=10);
   printf("Suma = %.3f\n",suma);

   return 0;
}
```

# Sentencias `break` y `continue`

```
while(expresión){
    sentencia(s);
    if (condición){
        continue;
    }
    sentencia(s);
}
```

```
do{
    sentencia(s);
    if (condición){
        continue;
    }
    sentencia(s);
}while(expresión);
```

```
for(init;condición;progreso){
    sentencia(s);
    if (condición){
        continue;
    }
    sentencia(s);
}
```

La sentencia **continue**, estando dentro de las estructuras `while`, `do-while` y `for`, pasa el control para que se ejecute la siguiente iteración, ignorando todas las sentencias dentro del ciclo que se encuentran después de la llamada al `continue`. Se usa casi siempre con una sentencia `if-else` dentro del ciclo.

# Sentencias break y continue

```c
#include <stdio.h>

int main()
{
    int i;
    float num, suma = 0.0;

    for(i=1; i <= 10; ++i){

        printf("Ingresa n%d: ",i);
        scanf("%f", &num);
        if (num < 0)
            continue;
        suma += num;

    }
    printf("Suma = %.3f\n", suma);

    return 0;
}
```

# Ejercicio

Ejercicios 4, 5 y 6 (60 minutos)

# Arreglos

## Arreglos (*arrays*)

Representan el concepto matemático de vector y matriz.

Un arreglo es una secuencia de posiciones que tiene las siguientes características:

- Ordenado: el enésimo elemento puede ser identificado
- Homogéneo: contiene datos del mismo tipo
- Selección de posiciones mediante índices
- Unidimensionales y multidimensionales

# Arreglos unidimensionales

- **Arreglo unidimensional** o vector consta de $n$ elementos, cada uno con una posición específica designada por un *índice*:

```
datos[0], datos[1], . . . , datos[i], . . . , datos[n-1]
```

- Los índices son enteros consecutivos y comienzan en 0.
- En cada posición del vector se almacena un valor:

| X[0] | X[1] | X[2] | X[3] | X[4] | X[5] | X[6] | X[7] |
|---|---|---|---|---|---|---|---|
| 14.0 | 12.0 | 8.0 | 7.0 | 6.41 | 5.23 | 6.15 | 7.25 |

Un vector llamado `X` con ocho elementos

# Arreglos unidimensionales

- Declaración de un arreglo unidimensional:

```
tipo_C identificador[tamaño];
```

  donde

  - `tipo_C` indica el tipo de los elementos del arreglo
  - `identificador` es el nombre del arreglo
  - `tamaño` es una constante que indica el número de elementos

- Por ejemplo:

```
int lista[100];
char nombre[40];
```

  se crean dos arreglos, el primero llamado `lista` con 100 elementos (del 0 al 99) de tipo `int`, y el segundo llamado `nombre` con 40 elementos (del 0 al 39) de tipo `char`

# Arreglos unidimensionales

```c
#include <stdio.h>
#define TAM 10

int main()
{
   int a[TAM], n = 0;

   do{
      printf("a[%d] = ", n);
      scanf("%d", &a[n]);
      n++;
   }while (n < TAM);

   return 0;
}
```

# Arreglos multidimensionales

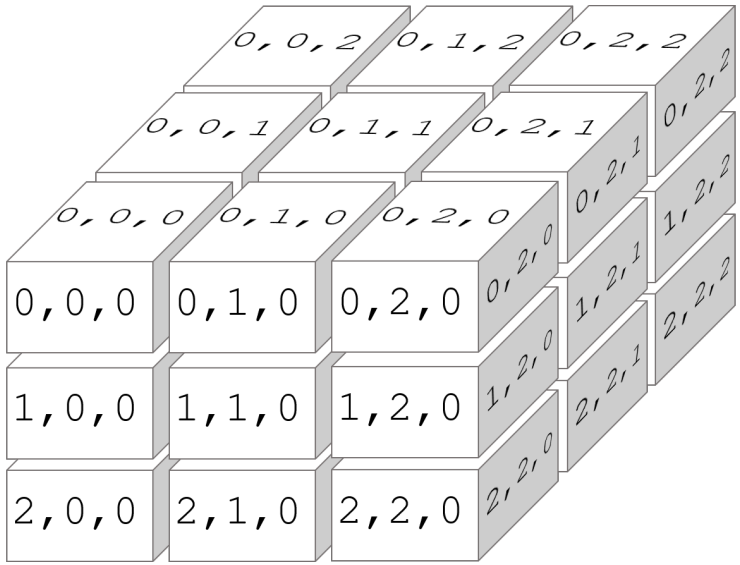
- Se pueden definir arreglos de varias dimensiones:
  - **Bidimensional:** dos dimensiones (matrices)
  - **Multidimensional:** tres o más dimensiones (hipermatrices)
- Declaración de un arreglo multidimensional:

```
tipo_C identificador[tam1][tam2]...[tamN];
```

  donde
  - `tipo_C` indica el tipo de los elementos del arreglo
  - `identificador` es el nombre del arreglo
  - `tam1`, `tam2`, . . . , `tamN` son constantes que indican el número de elementos en cada dimensión

# Arreglos multidimensionales

| | | | |
|---|---|---|---|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |




Por ejemplo:

```
char array2d[3][4];
int array3d[3][3][3];
```

se declaran dos arreglos multidimensionales, el primero llamado `array2d` de dos dimensiones con 12 elementos (tres filas y cuatro columnas) todos de tipo `char` (izquierda), el segundo llamado `array3d` de tres dimensiones con 27 elementos todos de tipo `int` (derecha)

# Inicialización de arreglos

Se pueden crear arreglos con valores preestablecidos desde su declaración:

```
int a[5] = {1,2,3,4,5};
int b[2][3] = {1,2,3,4,5,6};
```

se crean dos arreglos, el primero llamado 'a' es un vector con cinco elementos, mientras que el segundo es una matriz llamada 'b' con seis elementos. Debido a que los arreglos son almacenados por filas, el arreglo 'b' está organizado de la siguiente manera:

$$\texttt{b} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

# Inicialización de arreglos

```c
#include <stdio.h>
#define FILAS 3
#define COLS 3

int main()
{
  int t[FILAS][COLS];
  int f, c;

  for(f=0; f<FILAS; f++)
  {
    printf("\nDatos para la fila %d\n",f);
    for(c=0; c<COLS; c++)
      scanf("%d", &t[f][c]);
  }
  return 0;
}
```

# Cadenas de caracteres

- **Cadena de caracteres** – arreglo unidimensional en el cual todos sus elementos son de tipo `char`:

  ```
  char nombre[40];
  ```

  declara un arreglo llamado '`nombre`' con longitud máxima de 40 caracteres
- En la longitud máxima se considera un carácter nulo '`\0`', con el cual C finaliza todas las cadenas
- Si se desea introducir una cadena de hasta $n$ caracteres, se debe declarar el arreglo considerando el carácter nulo $(n + 1)$:

  ```
  char nombre[41];
  ```

# Cadenas de caracteres

- Una cadena puede ser inicializada asignándole un literal:

  ```
  char cadena[] = "abcde";
  ```

  declara una cadena con seis elementos (`cadena[0]` a `cadena[5]`), donde el sexto elemento es el carácter nulo

```c
#include <stdio.h>

int main()
{
   char saludo[] = "Hola ";
   char nombre[20];

   printf("Introduce tu nombre: ");
   scanf("%s",nombre);
   printf("%s%s\n",saludo,nombre);

   return 0;
}
```

# Cadenas de caracteres: `fgets` y `fputs`

- Para introducir cadenas de caracteres que incluyan espacios, se utiliza la función `fgets` definida en la biblioteca `stdio.h`:

```
char* fgets(char *str, int n, stdin);
```

  - `str` – puntero a un arreglo de caracteres donde se almacena la cadena leída
  - `n` – número máximo de caracteres que se leerán (incluido '`\0`')
  - `stdin` – entrada estándar
- Se pueden introducir todos los caracteres del teclado
- El carácter nulo '`\0`' se coloca automáticamente
- La función `fgets` devuelve un puntero al valor leído. Un valor nulo indica un error o una condición de fin de archivo (`EOF`, *end of file*)

# Cadenas de caracteres: `fgets` y `fputs`

- La función `fputs` escribe una cadena al monitor (`stdout`), y reemplaza el carácter nulo '`\0`' por el carácter de nueva línea '`\n`':

`int fputs(char *str, stdout)`

```
1  #include <stdio.h>
2  #define N 50
3
4  int main()
5  {
6    char saludo[] = "Hola ";
7    char nombre[N];
8
9    printf("Introduce tu nombre completo: ");
10   fgets(nombre, N, stdin);
11   printf("%s",saludo);
12   fputs(nombre, stdout);
13
14   return 0;
15 }
```

# Cadenas de caracteres: `fgets` y `fputs`

- La función `strlen` permite conocer la longitud de una cadena (número de caracteres antes del carácter nulo '\0'):

```
size_t strlen(const char *str)
```

- Para usar esta función es necesario incluir la biblioteca **string.h**

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char a[20]="Program";
    char b[20]={'P','r','o','g','r','a','m','\0'};

    printf("Length of string a = %lu\n",strlen(a));
    printf("Length of string b = %lu\n",strlen(b));

    return 0;
}
```

# Ejercicio

**Ejercicio 7 (45 minutos)**

# Ejercicio

## Ejercicio 8 (60 minutos)

# Ejercicio

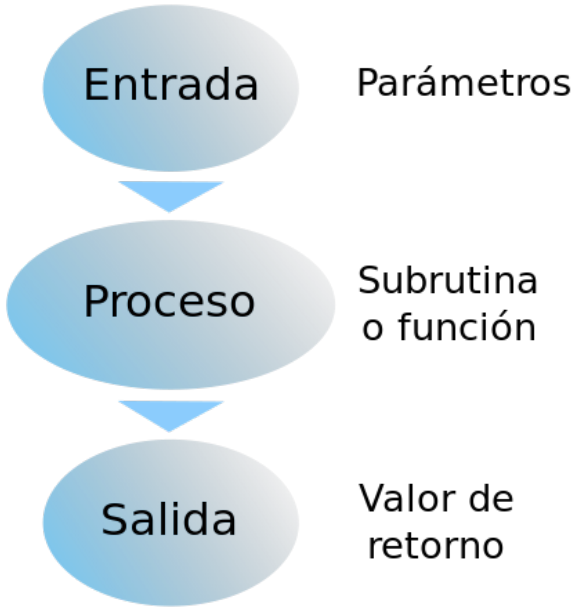
Ejercicio 9 (60 minutos)

# Programación modular / procedimental

**Términos usados indistintamente (no necesariamente sinónimos)**

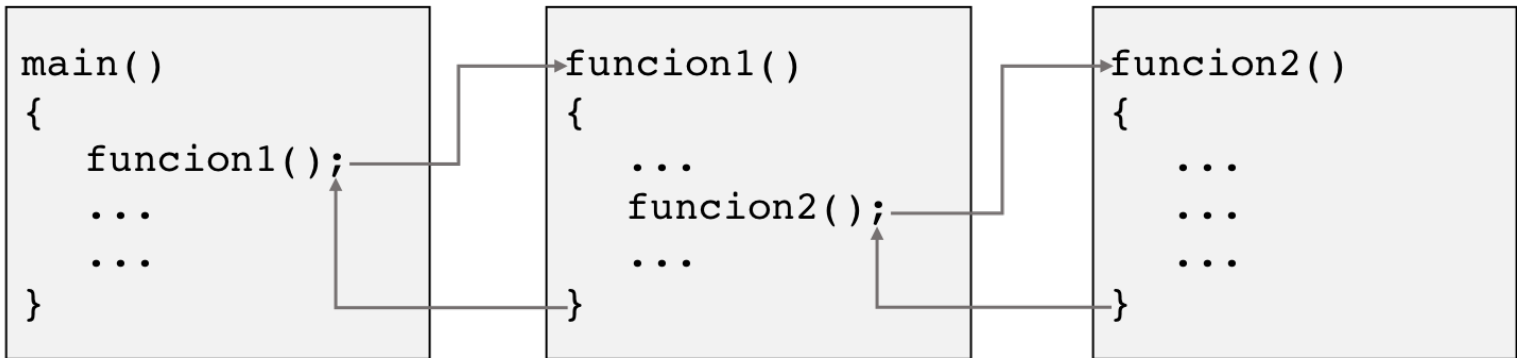
**módulo ⇔ función ⇔ procedimiento ⇔ subrograma ⇔ subrutina**

- División de un programa en módulos con el fin de hacerlo más legible y manejable
- Un problema complejo puede ser dividido en subproblemas más simples
- Un módulo tiene una tarea bien definida: **resolver uno de los subproblemas!**
- Colaboración / dependencia entre módulos
- **Reutilización de código:** escribir código una única vez e invocarlo tantas veces como se requiera!



# Funciones

- Una **función** es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica
- Todo programa en C consta de al menos una función, la función `main`, que es de donde comienza la ejecución del programa




Cuando se llama a una función, el control se pasa a la misma para su ejecución; y cuando finaliza, el control es devuelto a la función que la llamó para continuar con la ejecución a partir de la sentencia que efectuó la llamada.

# Definición de una función

```
tipo identificador(tipo id1, . . . , tipo idN)
{
   declaraciones;
   sentencias;
}
```

- **tipo** indica el tipo de valor devuelto por la función. Una función no puede retornar un arreglo o función, aunque si puede retornar un puntero a un arreglo o a una función
- **tipo id1, . . . , tipo idN** son los parámetros de la función. Estos **parámetros son opcionales**, y permiten el **paso de argumentos** a la función

# Variables locales y sentencia `return`

- El cuerpo de una función esta formado por todas las sentencias y declaraciones que ejecuta la función
- Tanto los **parámetros** de la función como las **variables** que se declaren dentro de de ella, son **locales: su ámbito/alcance está limitado a la función**
- La sentencia **return** devuelve un valor del tipo especificado en la cabecera de la función, retornando de esta manera el control a la función que hizo la llamada:

```
return (expresión);
```

- Si en la sentencia `return` no se especifica una `expresión`, la función no devuelve un valor

# Definición vs prototipo de una función

- El **prototipo o declaración** de una función permite especificar el nombre, el tipo del resultado, los parámetros formales y (opcionalmente) sus nombres
- No define el cuerpo de la función
- Un prototipo tiene la misma sintaxis que la definición de una función, pero no tiene cuerpo y termina con punto y coma
- Una función no puede ser llamada si previamente no está definida
- Si la función es definida antes de la función `main`, no es necesario incluir su prototipo. Sin embargo, la función **sólo podrá ser invocada desde funciones definidas posteriormente**

# Paso de argumentos

Existen dos formas de pasar los argumentos a los correspondientes parámetros de la función:

1. **Paso de argumentos por valor**
2. **Paso de argumentos por referencia**

Por ahora, nos concentraremos en el **paso de argumentos por valor**:

- Los parámetros de la función reciben una copia de los valores de los argumentos
- Los argumentos originales no son alterados por la función
- Se pueden pasar como argumentos variables, constantes y expresiones

# Paso de argumentos

```c
#include <stdio.h>

// Definicion de la funcion
float suma(float a, float b)
{
  return a+b;
}

int main()
{
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

  // Llamada a la funcion
  c = suma(a, b);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}
```

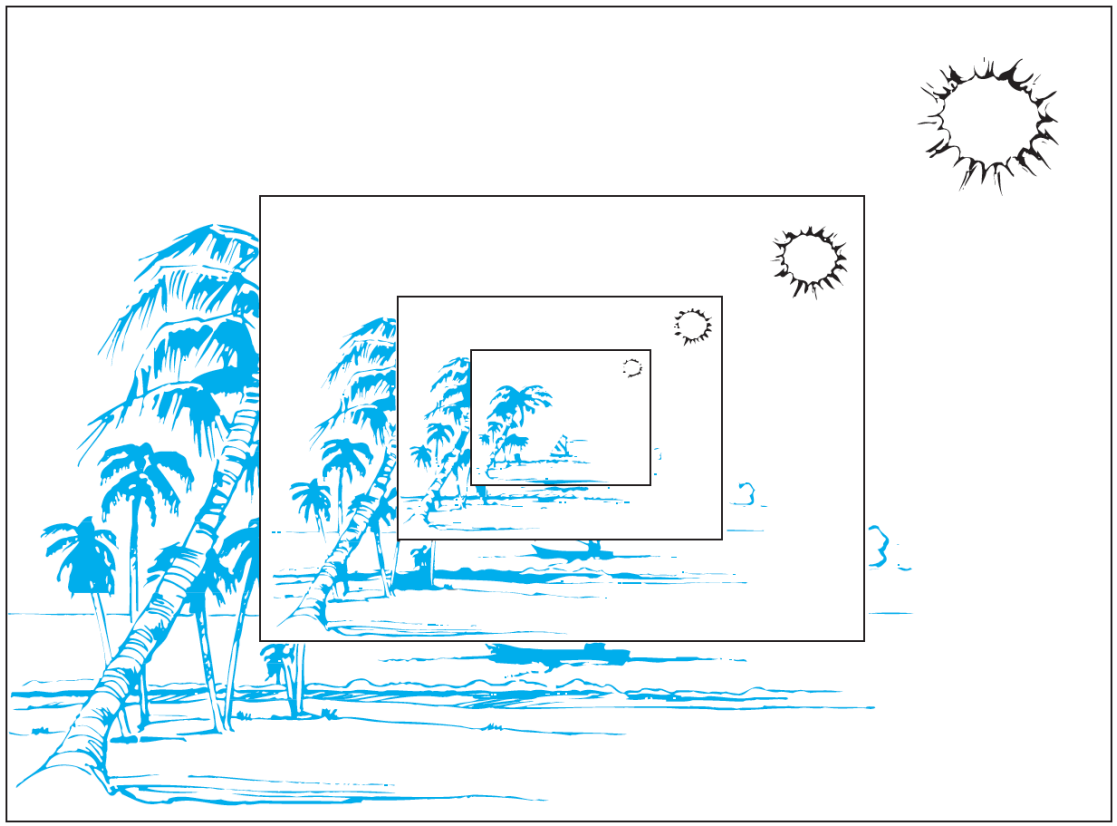
```c
#include <stdio.h>

// Prototipo de funcion
float suma(float, float);

int main()
{
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

  // Llamada a la funcion
  c = suma(a, b);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}

// Definicion de la funcion
float suma(float a, float b)
{
  return a+b;
}
```

# Ejercicio

**Ejercicio 10 (60 minutos)**

# Recursividad

A veces es más fácil describir un objeto o proceso en términos de sí mismo. Esto se llama **recursión**.



# Recursividad

## Algoritmo recursivo

Método que resuelve un problema mediante la **solución de instancias más pequeñas** del mismo problema. Para su definición se requiere:

- **Caso base:** la solución puede ser expresada directamente (sin recursión, no depende de instancias más pequeñas del problema)
- **Caso general (recursivo):** la solución depende de las soluciones de instancias más pequeñas

- Para todo algoritmo iterativo hay un algoritmo recursivo equivalente y viceversa
- Una solución recursiva genera un ciclo, para el que se cumple una condición de salida cuando se llega al caso base

# Recursividad: Factorial

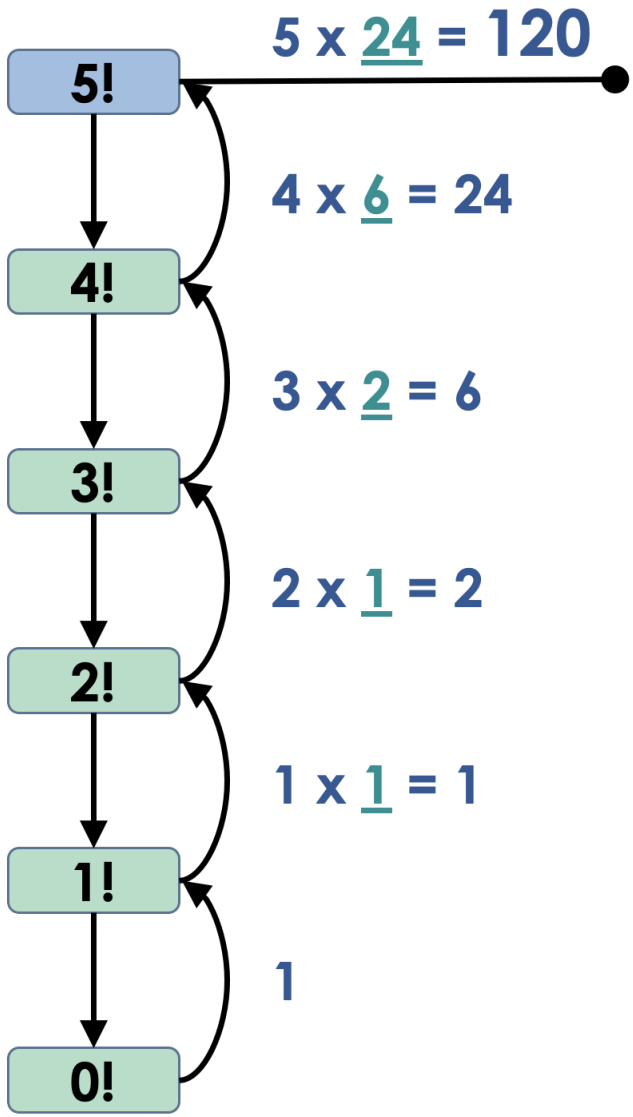
**Factorial de un número $n$:**

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{i=1}^{n} i$$

**Definición recursiva:**

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1)!, & n > 0 \end{cases}$$

Recursivamente, el problema se **resuelve a través de la solución de una instancia más pequeña**.



# Recursividad: Factorial

```c
#include <stdio.h>

// Version iterativa
int factorial_iter( int n ){
  int fac = 1, i;
  for(i=1; i<=n; i++)
    fac *= i;
  return fac;
}

// Version recursiva
int factorial_recu( int n ){
  if( n <= 1 )
    return n;
  return n * factorial_recu( n-1 );
}

int main(){
  int n;
  printf("Introduce un entero n: ");
  scanf("%d", &n);
  printf("Factorial iterativo: %d\n", factorial_iter(n));
  printf("Factorial recursivo: %d\n", factorial_recu(n));
  return 0;
}
```

# Ejercicio

Ejercicios 11, 12 y 13 (120 minutos)