# Programación I

## Clase 2
## Funciones y Procedimientos

*Universidad Tecnológica Nacional - Cuch Sede Chivilcoy*

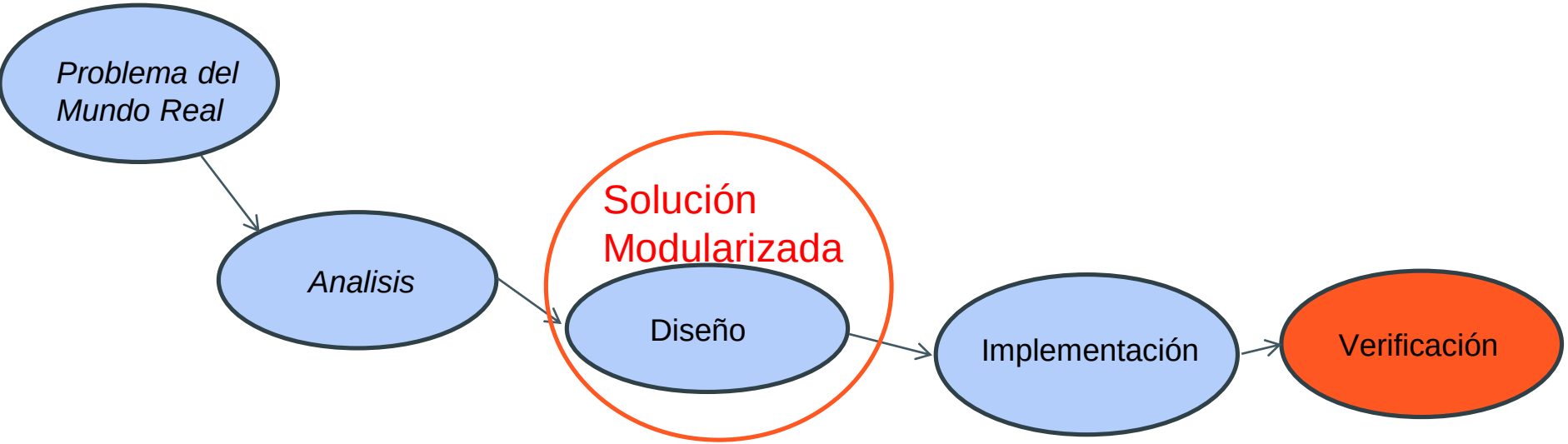
*Profesores*

*Luciana Denicio*

*Matias Garro*

# Etapas de resolución de un problema por computadora


Complejos
Extensos
Modificables



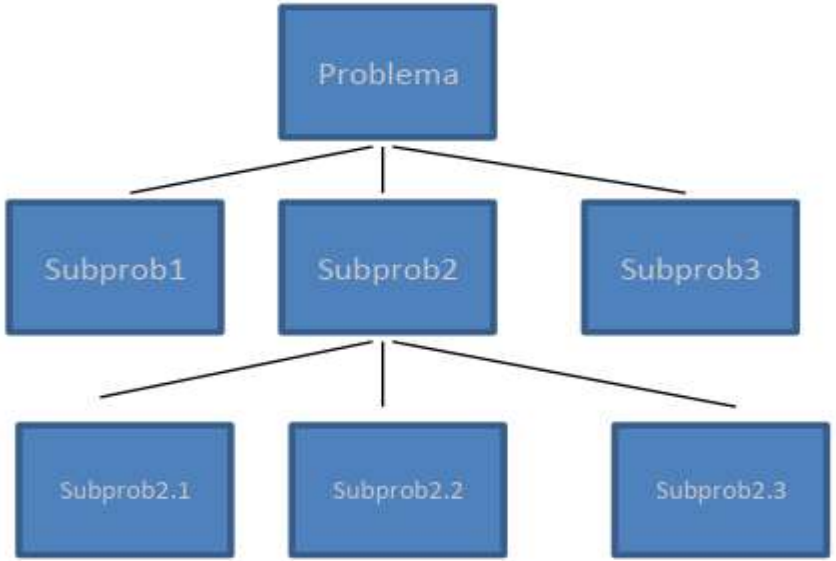
# Metodologia de diseño TOP DOWN

Principio de “Divide y vencerás”

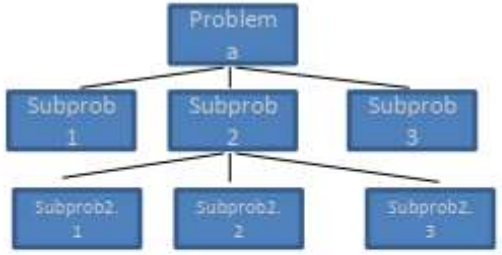
Descomponer el problema en partes (subproblemas) mas simples

Al descomponer un problema se debe tener en cuenta:

- Que cada subproblema resuelva una parte “bien” simple.
- Que cada subproblema pueda resolverse independientemente
- Que las soluciones a los subproblemas deben combinarse para resolver el problema original




**¿Cuándo se detiene la descomposición del problema?**




Permite distribuir el trabajo

Favorece el mantenimiento correctivo

## Ventajas de la descomposición del problema

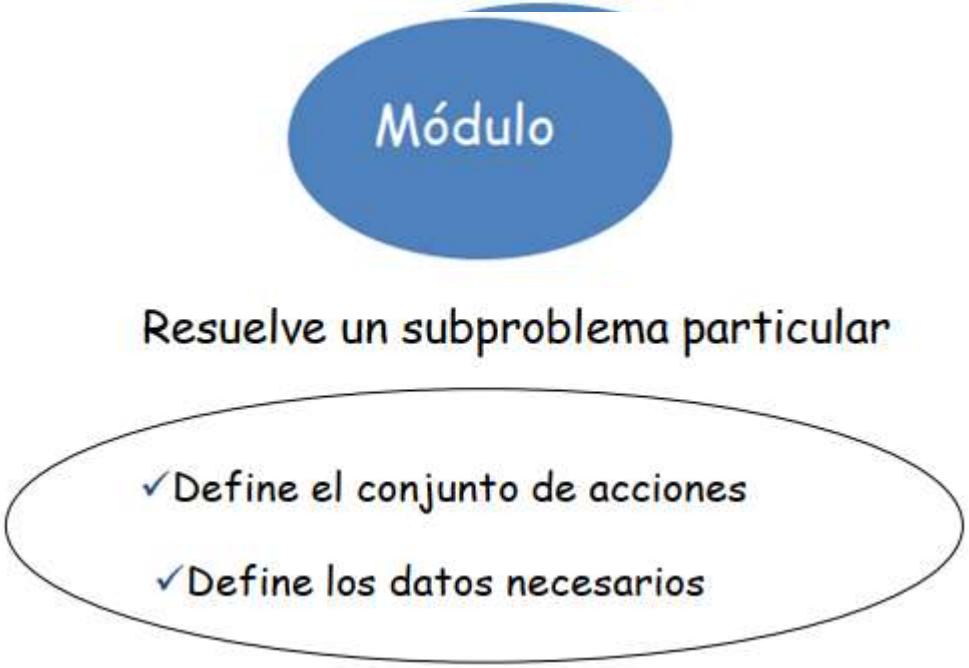
Facilita la reutilización dentro del mismo problema o en otro similar

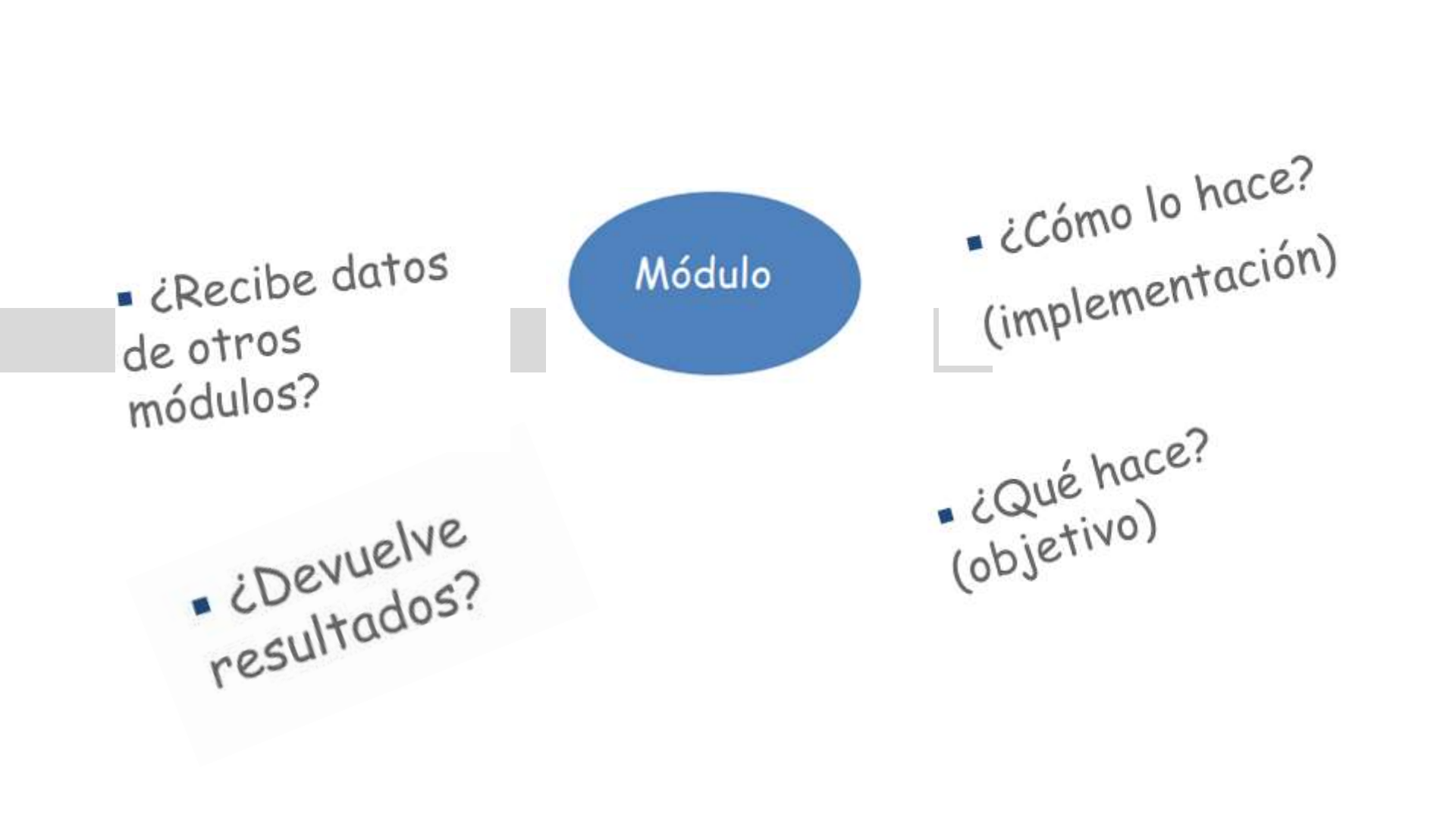
Facilita el crecimiento de los sistemas

Aumenta la legibilidad

# Modularizacion

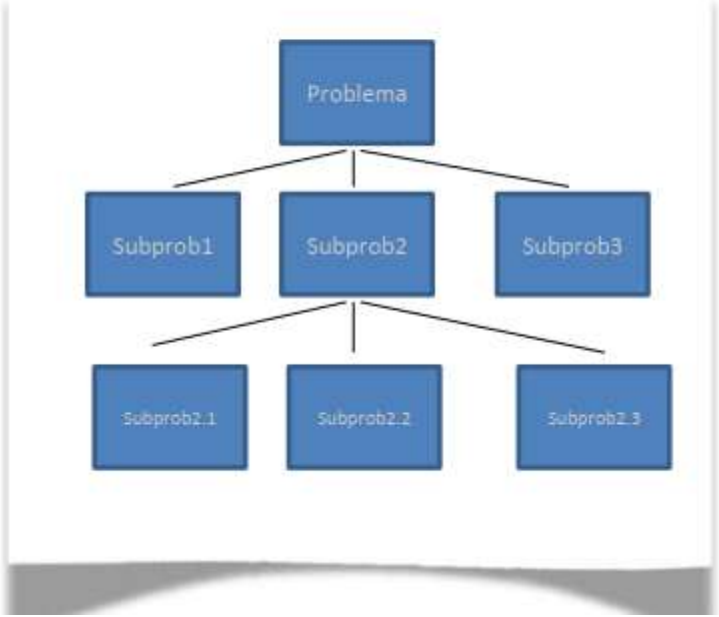
La tarea de Modularizar implica dividir un problema en partes. Se busca que cada parte realice una tarea simple y pueda resolverse de manera independiente a las otras tareas.




Módulo
¿Recibe datos de otros módulos?
¿Cómo lo hace? (implementación)
¿Devuelve resultados?
¿Qué hace? (objetivo)

# Como resultado de la etapa de Diseño se tiene:

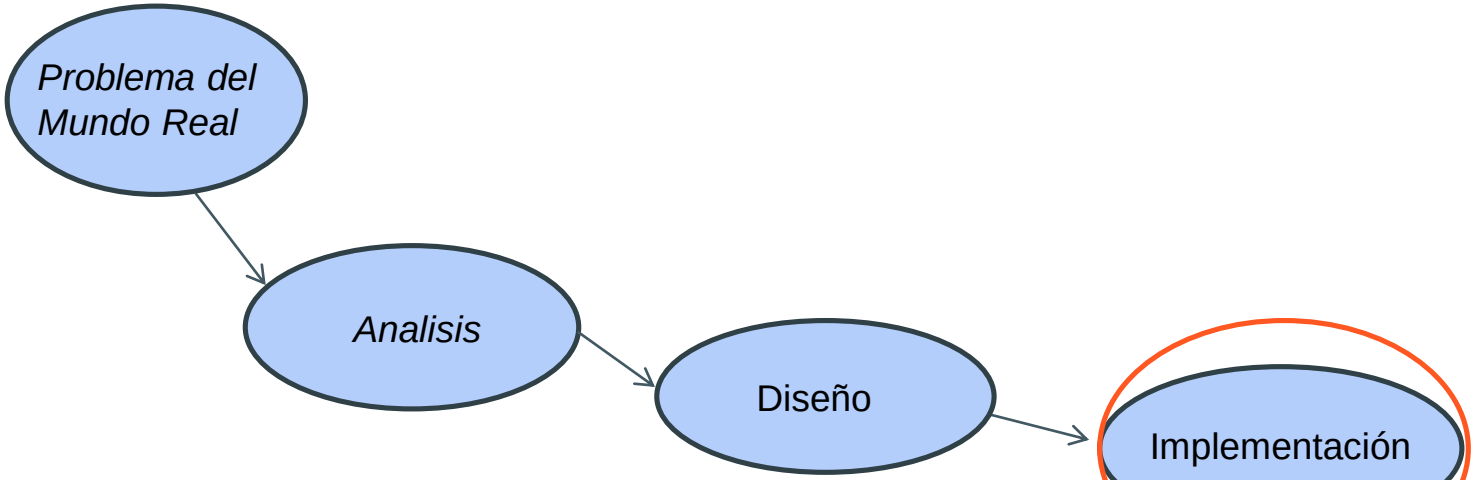
- Cuales son los módulos




Esta etapa no depende del lenguaje de Programación que se use…

- Cual es el objetivo de cada uno.
- Cuales son los datos propios
- Cuales son los datos compartidos con Otros módulos.
- Cuales son los datos compartidos con Otros módulos.
- Cuales es el conjunto de acciones para alcanzar ese objetivo

# Avanzamos a la etapa de Implementación




**Modelo**

**Solución Modularizada**

Módulos

Datos propios

Datos compartidos

**¿Como escribir el programa y los modulos?**

- Se debe elegir el lenguaje de programación para escribir los algoritmos de cada módulo y la declaración de sus datos
- Los lenguajes de programación ofrecen diversas opciones

Definición del módulo
¿Qué hace el módulo cuando se ejecuta?

- Encabezamiento (Interface)
  - Tipo de módulo
  - Identificación
  - Datos de comunicación
- Declaración de tipos
- Declaración de variables
- Sección de instrucciones ejecutables

Invocación del módulo
¿Cómo se hace cuando se quiere usar el módulo?

- Se debe conocer de qué manera se invoca o se llama al módulo para que ejecute sus acciones

La invocación puede hacerse mas de una vez

¿Qué ocurre con el flujo de control del programa?

# Modularizar en Lenguaje C

## ¿Funciones, métodos o procedimientos?

En el mundo de la programación, muchos acostumbramos hablar indistintamente de estos tres términos sin embargo poseen deferencias fundamentales.

## Funciones:

Las funciones son un conjunto de procedimiento encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente retornan un valor. Esta definición proviene de la definición de función matemática la cual posee un dominio y un rango, es decir un conjunto de valores que puede tomar y un conjunto de valores que puede retornar luego de cualquier operación.

# Como declarar un Función en C

**tipo_de_retorno nombre_de_funcion(tipo1 parametro1, tipo2 parametro2, ...);**

**tipo_de_retorno**: Es el tipo de valor que la función devuelve. Si no devuelve ningún valor, se usa void.
**nombre_de_funcion**: Es el nombre que se usará para invocar la función.
**tipo1, tipo2, ...**: Son los tipos de los parámetros que la función recibirá.
**parametro1, parametro2, ...**: Son los nombres de los parámetros que la función tomará.

**Ejemplo de una Declaración de Función**

Supongamos que queremos declarar e implementar una función que **suma dos números**:

```
int sumar(int a, int b) {
    return a + b;  // La función devuelve la suma de a y b
}

#include <stdio.h>

int sumar(int a, int b);  // Declaramos la función aquí

int main() {
    int resultado = sumar(3, 5);  // Llamamos a la función con 3 y 5 como argumentos
    printf("El resultado de la suma es: %d\n", resultado);
    return 0;
}
```

## Acerca de los argumentos o parámetros

Hay algunos detalles respecto a los argumentos de una función, veamos:

Una función o procedimiento pueden tener una cantidad cualquier de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros.

Aunque habitualmente no suelen tener más de 4 o 5.

Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.

Los argumentos de una función también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo de la función

Recordemos que una función siempre retorna algo, por lo tanto es obligatorio declararle un tipo (el primer componente de la sintaxis anterior), luego debemos darle un nombre a dicha función, para poder identificarla y llamarla durante la ejecución, después al interior de paréntesis, podemos poner los argumentos o parámetros. Luego de la definición de la "firma" de la función, se define su funcionamiento entre llaves; todo lo que esté dentro de las llaves es parte del cuerpo de la función y éste se ejecuta hasta llegar a la instrucción *return*.

**Consejos acerca de *return***

Debes tener en cuenta dos cosas importantes con la sentencia return:

- Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada

Es común encontrar funciones con múltiples sentencias return al interior de condicionales, pero una vez que el código ejecuta una sentencia return lo que haya de allí hacia abajo no se ejecutará.

- El tipo del valor que se retorna en una función debe coincidir con el del tipo declarado a la función, es decir si se declara int, el valor retornado debe ser un número entero.

# Procedimientos

**Hablemos un poco de los procedimientos**
Los procedimientos son similares a las funciones, aunque más resumidos. Debido a que los procedimientos no retornan valores, no hacen uso de la sentencia return para devolver valores y no tienen tipo específico, solo *void*. Veamos un ejemplo:

```
#include <stdio.h>

// Implementación del procedimiento

void imprimirMensaje(char* mensaje) {
    printf("%s\n", mensaje);  // Imprime el mensaje recibido como argumento
}
```

**void**: Indica que la función no devuelve ningún valor.
**nombreDelProcedimiento**: Es el nombre que le damos al procedimiento.
**tipoParametro1, tipoParametro2, ...**: Son los tipos de los parámetros que recibe el procedimiento (si los tiene).
**parametro1, parametro2, ...**: Son los nombres de los parámetros.

**Ejemplo de Declaración y Uso de un Procedimiento**
Vamos a crear un procedimiento que imprima un mensaje personalizado. Este procedimiento no devuelve ningún valor, solo realiza una acción (imprimir el mensaje).

**Paso 1: Declaración del Procedimiento**
Primero, declaramos el procedimiento (también conocido como **función** en C) que acepta un parámetro de tipo char* (cadena de caracteres) y no devuelve ningún valor.

```c
#include <stdio.h>

// Declaración del procedimiento
void imprimirMensaje(char* mensaje);

// Implementación del procedimiento
void imprimirMensaje(char* mensaje) {
    printf("%s\n", mensaje);  // Imprime el mensaje recibido
}

int main() {
    // Llamamos al procedimiento e imprimimos un mensaje
    imprimirMensaje("¡Hola, mundo!");

    return 0;
}
```

- La declaración del procedimiento imprimirMensaje, que toma un parámetro de tipo char* (una cadena de caracteres) y no devuelve nada (void).
- En el cuerpo del procedimiento, usamos printf para imprimir el mensaje que se pasa como parámetro
- En la función main(), llamamos al procedimiento imprimirMensaje, pasando el texto "¡Hola, mundo!" como argumento.

**Resumen:**

•Un **procedimiento** en C es simplemente una función que no devuelve ningún valor, y su tipo de retorno es **void**.

•Se declara de la misma manera que una función, pero con void en lugar de un tipo de retorno específico.

•Puede aceptar parámetros como cualquier otra función.

•El uso de procedimientos en C ayuda a estructurar el código, hacer que sea más modular y fácil de mantener.

En C, la declaración **char* mensaje** hace referencia a un **puntero a un carácter** (es decir, una cadena de caracteres). Vamos a desglosarlo para entenderlo mejor:

**1. char: Tipo de dato**
**char** es un tipo de dato en C que representa un **carácter**. Normalmente, un char ocupa 1 byte de memoria y almacena un solo carácter (por ejemplo, 'A', '1', '#').
**2. * (Asterisco): Puntero**
El ***** en la declaración char* indica que estamos declarando un **puntero**. Un puntero es una variable que almacena la **dirección de memoria** de otra variable en lugar de almacenar directamente un valor.

Entonces, **char* mensaje** es un puntero que va a almacenar la dirección de memoria de un **carácter** o de la primera posición de una **cadena de caracteres**.

**3. mensaje: Nombre de la variable**

**mensaje** es el nombre del puntero que declaramos. Este puntero va a almacenar la dirección de memoria de un carácter o el inicio de una cadena de caracteres.

**Combinado: char* mensaje**

**char* mensaje** es una declaración que significa "un puntero a un tipo char". Usualmente, este tipo de declaración se utiliza para trabajar con **cadenas de caracteres** en C, ya que las cadenas en C son arrays de caracteres terminados en un carácter nulo (\0).

**¿Cómo se usa este puntero?**

Cuando escribimos **char* mensaje**, estamos declarando una variable mensaje que puede almacenar la dirección de memoria de un **array de caracteres** (una cadena). En C, las cadenas se representan como arrays de caracteres, y la primera posición de un array de caracteres es un puntero a char.

## ¿Por qué es útil char* mensaje?

**Cadenas de caracteres**: Usar **char*** es la forma estándar en C para manejar **cadenas de caracteres**. Dado que las cadenas son arrays de caracteres, un puntero char* se utiliza para referirse al primer carácter de la cadena.
**Eficiencia**: Usar punteros para pasar cadenas de caracteres a funciones es eficiente, ya que no es necesario copiar toda la cadena, solo se pasa la dirección de memoria del primer carácter.

**Resumen:**

•**char* mensaje** es un **puntero a un carácter**.

•Se usa comúnmente para **trabajar con cadenas de caracteres**, ya que una cadena en C es simplemente un array de caracteres.

•Al declarar **char* mensaje**, estamos creando una variable que puede almacenar la dirección de memoria de una cadena de caracteres.

En el lenguaje C, existen dos formas principales de pasar parámetros a las funciones:

**por valor** y **por referencia**. Estas dos técnicas afectan cómo las variables se pasan a las funciones y cómo se modifican dentro de ellas. A continuación te explico las diferencias entre ambas, paso a paso.

**1. Paso por Valor**

Cuando pasamos un parámetro **por valor**, estamos enviando una **copia** del valor de la variable a la función. Cualquier cambio realizado en el parámetro dentro de la función **no afectará** al valor original de la variable que fue pasada.

**Pasos:**

Se crea una copia del valor de la variable que se pasa a la función.
La función opera sobre esa copia (no sobre la variable original).
Cualquier modificación dentro de la función afecta únicamente a la copia, no a la variable original.

Ejemplo

```
#include <stdio.h>

void cambiarValor(int x) {
    x = 20;  // Cambia solo la copia local de x
}

int main() {
    int numero = 10;
    cambiarValor(numero);  // Se pasa por valor
    printf("Valor de numero: %d\n", numero);  // Se imprime 10, no 20
    return 0;
}
```

- La variable numero tiene el valor 10.
- Cuando se llama a la función cambiarValor(numero), el valor de numero (10) se **copia** a la variable x en la función.
- Dentro de cambiarValor, se cambia x a 20, pero esto **no afecta** al valor de numero en main(), ya que solo se modificó la copia de x.

## Paso por Referencia

Cuando pasamos un parámetro **por referencia**, estamos pasando la **dirección de memoria** de la variable a la función. Esto significa que la función tiene acceso directo a la variable original y cualquier cambio realizado en el parámetro dentro de la función **afectará** al valor de la variable original.

- Se pasa la **dirección de memoria** de la variable a la función (es decir, un puntero a la variable).
- La función opera sobre la **variable original** a través de su dirección de memoria.
- Cualquier cambio realizado dentro de la función afectará directamente a la variable original.

```
#include <stdio.h>

void cambiarValor(int* x) {
    *x = 20;  // Modifica el valor de la variable original usando su dirección
}

int main() {
    int numero = 10;
    cambiarValor(&numero);  // Se pasa la dirección de numero
    printf("Valor de numero: %d\n", numero);  // Se imprime 20
    return 0;
}
```

- La variable numero tiene el valor 10.
- Cuando se llama a la función cambiarValor(&numero), se pasa la **dirección de memoria** de numero (usando el operador &).
- Dentro de la función, x es un puntero que apunta a la dirección de memoria de numero, por lo que al hacer *x = 20; estamos cambiando directamente el valor de numero.

Valor de numero: 20

## Diferencias Clave entre Paso por Valor y Paso por Referencia

| Característica | Paso por Valor | Paso por Referencia |
|---|---|---|
| **Qué se pasa** | Una **copia** del valor de la variable. | La **dirección de memoria** de la variable (un puntero). |
| **Modificación de la variable original** | No afecta a la variable original. | Afecta directamente a la variable original. |
| **Uso de memoria** | No requiere punteros, solo la copia de los valores. | Requiere punteros para manejar direcciones de memoria. |
| **Efecto en la variable** | La variable original **no cambia**. | La variable original **se modifica**. |
| **Costos de ejecución** | Normalmente es más eficiente para tipos pequeños (como `int`, `char`), pero para tipos grandes como arrays o estructuras puede ser ineficiente. | Puede ser más eficiente cuando se manejan grandes estructuras o arrays, ya que no se hace una copia de los datos, solo se pasa la dirección. |
| **Sintaxis** | Usualmente no se necesitan punteros. | Necesita punteros (y operadores como `&` para obtener la dirección y `*` para acceder al valor). |

**Cuándo usar cada uno:**

**Paso por Valor**:
Es útil cuando **no necesitas modificar** el valor de la variable original.
Es más sencillo y seguro, ya que no manipulas directamente las direcciones de memoria.
Usado comúnmente con tipos de datos simples (como enteros, flotantes, etc.), donde el costo de copiar el valor es bajo.

**Paso por Referencia**:
Es útil cuando **quieres modificar** el valor de la variable original dentro de la función.
Es más eficiente para trabajar con grandes estructuras de datos (como arrays o estructuras), ya que no es necesario copiar los datos completos.
Usado cuando se desea que la función pueda afectar el estado de la variable original o cuando se necesita modificar múltiples valores dentro de una función.

**Resumen:**

•**Paso por valor**: Se pasa una copia de la variable. Los cambios dentro de la función **no afectan** a la variable original.

•**Paso por referencia**: Se pasa la dirección de memoria de la variable. Los cambios dentro de la función **sí afectan** a la variable original.
Ambos enfoques tienen sus casos de uso dependiendo de lo que necesites lograr en tu programa.