Laboratorio 11

Programación de Computadores (2025-2)

Temas del Laboratorio 11

- Funciones
- Parámetros por valor y referencia
- Recursividad
- Implementaciones Iterativas vs Recursivas
- Ejercicios

Motivación

Al escribir código, muchas veces reescribimos ciertas secciones de código varias veces

Esto tiene diversas desventajas:

- Si hay un error en esta sección de código, debemos corregirlo varias veces
- Un código largo es un código más difícil de entender
- Se tarda más tiempo en depurar los errores

A través de las funciones vamos a **modularizar** el código

- Tomar un problema complejo, y dividirlo en varios pasos o partes más simples

Funciones

Una función es un bloque de código que realiza una tarea específica.

Las funciones permiten dividir un programa en partes más pequeñas y manejables, lo que facilita la **comprensión**, la **depuración** y el **mantenimiento** del código.

Además, las funciones promueven la reutilización del código, ya que se puede llamar a una función desde diferentes partes del programa.

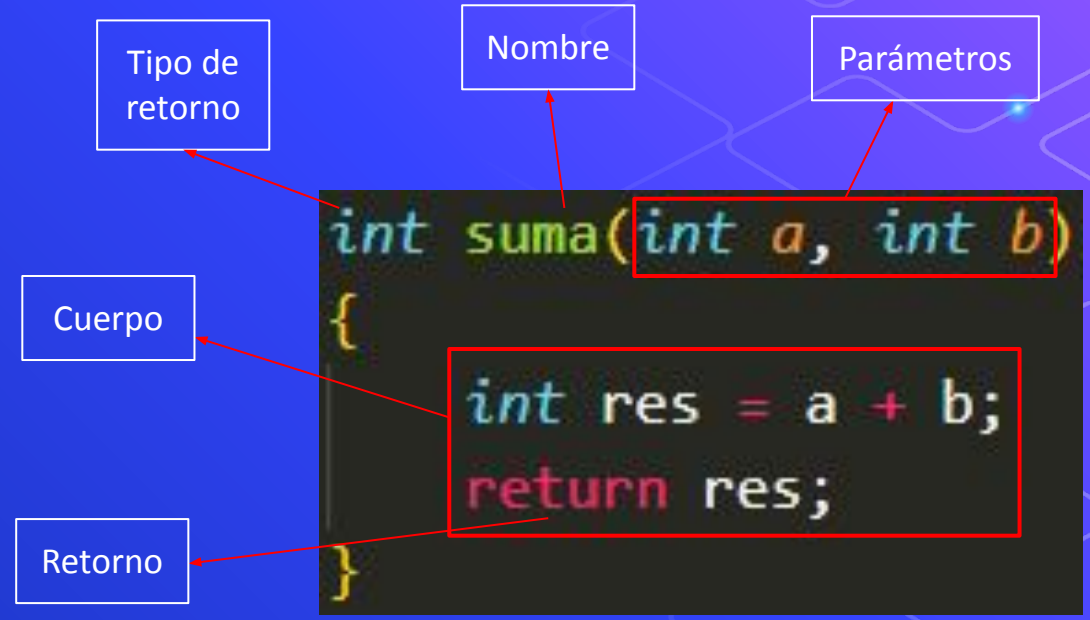
Las funciones pueden llamar a otras funciones.

```
tipo_retorno nombre_funcion(parametros) {
    // Cuerpo de la función
    // ...declaraciones y código...
    return valor_retorno;
}
```

Funciones

La declaración de una función consta de las siguientes partes:

- Tipo de retorno
- Nombre de la función
- Parámetros (entrada)
- Cuerpo de la función
- Retorno



Funciones

```
tipo_retorno nombre_funcion(parametros) {
    // Cuerpo de la función
    // ...declaraciones y código...
    return valor_retorno;
}
```

tipo_retorno: Especifica el tipo de dato que la función devolverá como resultado. Puede ser cualquier tipo de dato válido en C, como int, float, char, etc. Si la función no devuelve ningún valor, se usa **void.**

parametros: Son variables que la función recibe como entrada. Se pueden especificar cero o más parámetros separados por comas.

return valor_retorno: *return* se utiliza para devolver un valor de la función. El tipo de valor que se devuelve debe coincidir con el tipo de retorno especificado al principio de la función. Si la función no devuelve ningún valor se omite el *return*.

Declaración de Funciones

```c
int suma(int a, int b){
    return a+b;
}

int main(){

    int a = 5;
    int b = 7;
    int c = suma(a,b);
    printf("La suma de %d y %d es %d\n", a, b, c);
    return 0;
}
```

Declaración y código de la función antes del *main*

```c
int suma(int a, int b);

int main(){

    int a = 5;
    int b = 7;
    int c = suma(a,b);
    printf("La suma de %d y %d es %d\n", a, b, c);
    return 0;
}

int suma(int a, int b){
    return a + b;
}
```

Declaración antes del *main* y código después del *main*

Uso de parámetros por referencia

La función recibe direcciones de memoria de las variables originales.

Los cambios realizados a través de punteros dentro de la función afectan a las variables originales.

Se utiliza cuando se desea que la función pueda modificar directamente los valores de las variables originales.

```c
void restar(int *a, int *b, int *c) {
    *c = *a - *b;
}

int main() {
    int a = 5;
    int b = 3;
    int c;
    restar(&a, &b, &c);
    printf("%d - %d = %d\n", a, b, c);
    return 0;
}
```

Ejercicio 1

Crear un programa para recibir *n* números desde la terminal e imprimir los resultados, usando las siguientes funciones:

crearArreglo(&n): Recibe el número *n* desde la terminal, correspondiente a la cantidad de elementos y crear un arreglo de tamaño *n* en memoria dinámica. La función debe devolver el puntero al arreglo.

llenarArreglo(array, n): Le pide al usuario los n números y los guarda en el arreglo *array*

imprimirArreglo(array,n): Imprime en consola los *n* elementos del arreglo.

```c
int main() {

    int* miArreglo;
    int n;
    // Crear un arreglo dinámico
    miArreglo = crearArreglo(&n);
    // Llenar el arreglo con valores ingresados por el usuario
    llenarArreglo(miArreglo, n);
    // Imprimir el arreglo
    imprimirArreglo(miArreglo, n);
    // Liberar la memoria al final del programa
    free(miArreglo);

    return 0;
}
```

Main del ejercicio 1

Ejercicio 2

Crear las funciones para completar el programa:

concatenarFrases: Recibe las dos frases y crea en memoria dinámica la concatenación de las dos frases, entregando el puntero al bloque de memoria asignado.

invertirFrase: Recibe la primera frase ingresada e invierte los caracteres. La frase invertida debe crearse en memoria dinámica. La función devuelve el puntero al bloque de memoria que contiene la frase invertida.

liberarMemoria: Función que recibe el puntero a los bloques de memoria usados en las funciones anteriores para liberar la memoria asignada.

```c
int main() {
    char frase1[200], frase2[200];

    // Solicitar al usuario que ingrese dos frases
    printf("Ingrese la primera frase: ");
    // Leer hasta encontrar un salto de línea
    scanf("%[^\n]", frase1);
    getchar(); // Limpiar el buffer del teclado (tecla Enter)
    printf("Ingrese la segunda frase: ");
    scanf("%[^\n]", frase2);

    // Concatenar las dos frases e imprimir el resultado
    char* concatenada = concatenarFrases(frase1, frase2);
    printf("Frases concatenadas: %s\n", concatenada);

    // Invertir la primera frase e imprimir el resultado
    char* invertida = invertirFrase(frase1);
    printf("Primera frase invertida: %s\n", invertida);

    // Liberar memoria asignada para las frases
    liberarMemoria(concatenada);
    liberarMemoria(invertida);

    return 0;
}
```

Main del ejercicio 2

Recursividad

En computación y otras ciencias, se emplea la recursividad como una técnica en la cual una función se llama a sí misma para resolver una subparte de un problema.

Esta estrategia implica dividir el problema en partes más pequeñas sucesivamente hasta alcanzar un caso base, que representa el problema mínimo.

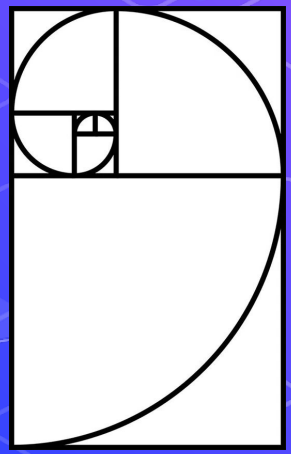
Fibonacci

La i-ésima posición de la secuencia de Fibonacci se define como:

$$f(i) = \begin{cases} i = 0 & 0 \\ i = 1 & 1 \\ i > 1 & f(i-1) + f(i-2) \end{cases}$$

Casos base

Paso recursivo



f = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,...}

Recursividad

¿Que pasa si...?

```c
void mi_funcion() {
    printf("Hola\n");
    mi_funcion();
}
```

Resultado:

```
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hol^C
```

Condición de salida

o “caso base”

```c
void mi_funcion2(int n) {

    if (n == 0) {
        return;
    }

    printf("Hola\n");
    mi_funcion2(n - 1);
}
```

* todos los códigos están disponibles en Canvas

Implementaciones Iterativas vs Recursivas

Toda función que utilice recursividad, se puede implementar de manera iterativa usando ciclos.

```
// Función que retorna la n-ésima potencia de 2
```

```
// implementación recursiva
int potencia2_r(int n) {
    if (n == 0) {
        return 1;
    } else {
        return 2 * potencia2_r(n - 1);
    }
}
```

```
//implementación iterativa
int potencia2_i(int n) {
    int potencia = 1;
    for (int i = 0; i < n; i++) {
        potencia *= 2;
    }
    return potencia;
}
```

Ejercicio 1

Crear una función que retorne la N posición de la secuencia de Fibonacci.

Ejercicio 2

En base a la siguiente implementación de binary search, crear una implementación usando recursividad

```c
// Binary search

// implementación iterativa
// retorna el índice del elemento x en el arreglo arr de tamaño n
int binary_search_i(int arr[], int n, int x) {
    int low = 0;
    int high = n - 1;
    int mid;

    while (low <= high) {
        mid = (low + high) / 2;

        if (x < arr[mid]) {         // si el elemento está en la primera mitad
            high = mid - 1;         // se continua la búsqueda en la primera mitad

        } else if (x > arr[mid]) { // si el elemento está en la segunda mitad
            low = mid + 1;          // se continua la búsqueda en la segunda mitad

        } else {                    // si el elemento está en la mitad
            return mid;             // se retorna el índice
        }
    }

    return -1;
}
```

* todos los códigos están disponibles en Canvas

Ejercicio extra 1

Imaginemos dos arreglos concatenados de igual tamaño, ambos están ordenados de menor a mayor.

Crear una función que junte ambos arreglos en un solo arreglo ordenado.

Recibir sólo los siguientes parámetros:

```
void merge(int *arreglo, int tamano){}
```

* Pista:
El segundo arreglo empieza en la posición tamano/2

Ejercicio extra 2

Con el código anterior, hacer una función recursiva que reciba un arreglo desordenado, lo divida en dos, se llame recursivamente a sí misma con cada mitad del arreglo y luego aplique la función merge().

Considerar que, en el caso base en el que tamano es == 1, no es necesario dividir ni hacer nada.

```
void mergeSort(int *arr, int n){
```