UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente:** Américo Chulluncuy Reynoso

2025-II

Sesión 1:

Recursividad e Iteración

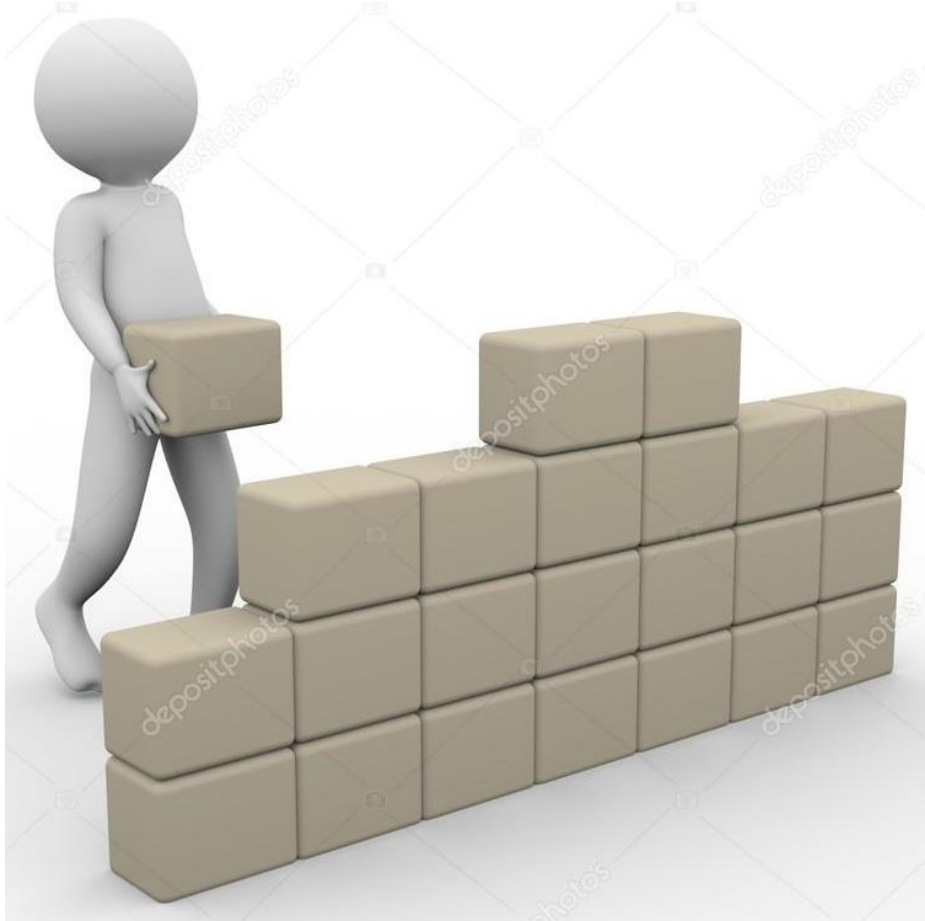
Contenido

1. Funciones en C++: Definición
2. Declaración (prototipo) vs definición
3. Llamada de una función
4. Función main. La pila de ejecución
5. Paso de parámetros por valor y por referencia
6. Paso de parámetros de tipo arreglos
7. Variables y alcance en funciones
8. Sobrecarga y Plantilla de funciones
9. Recursividad

Introducción

- Un programa de 1000 líneas dentro de main() es difícil de leer, mantener y escalar. En proyectos reales (como software de ingeniería, videojuegos, sistemas embebidos, etc.) es insostenible.

- Las funciones permiten:
  - ✓ Dividir el trabajo en tareas pequeñas y manejables.
  - ✓ Reutilizar código.
  - ✓ Organizar el código en bloques con responsabilidad única.
  - ✓ Hacer pruebas más fáciles y más precisas (unit testing).
  - ✓ Fomentar trabajo colaborativo



1. Funciones en C++: Definición

- Una función en C++ es un bloque de código con nombre que:
  - ✓ Puede tomar parámetros (entradas)
  - ✓ Realiza un proceso
  - ✓ Puede devolver un resultado (salida)

- **Sintaxis básica**

```
tipo_retorno nombre_funcion(tipo_1 arg_1, ..., tipo_n arg_n) {
    // cuerpo
    return valor_retorno;  // opcional
}
```

- **Ejemplo:**

```
double areaRectangulo(double base, double altura) {
    return base * altura;
}
```

2. Declaración (prototipo) vs definición

- **Declaración o prototipo**. Se escribe antes de main() para decirle al compilador que la función existe.

```
double areaRectangulo(double base, double altura);
```

  - ✓ Permite organizar el código por secciones (main arriba, funciones abajo).
  - ✓ Mejora la legibilidad y el orden.
  - ✓ Necesario cuando la función es definida después de ser llamada.
  - ✓ `areaRectangulo(double, double);` firma de la función (util en overloading)

prototipo llamada definición

nombre de la f ← parámetros

Sobrecarga de funciones

- **Definición**. Es la implementación completa

```
double areaRectangulo(double base, double altura) {
    return base * altura;
}
```

3. Llamada de una función

- Cuando una función es invocada:
  - ✓ Se evalúan los argumentos que se pasan.
  - ✓ El control del programa se transfiere a la función.
  - ✓ Se ejecuta el cuerpo de la función.
  - ✓ Se ejecuta una instrucción return (explícita o implícita).
  - ✓ El control retorna al punto de llamada en el program

se ejecuta si es llamada

Las funciones pueden ser llamadas dentro de otras funciones o incluso dentro de expresiones.

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

int main() {
   int x = 5;
   int y = 10;
   int resultado = sumar(x, y);  // ← llamada
   cout << "Resultado: " << resultado << endl;
   return 0;
}
```

implícito

Funciones(verbo funcionar) del return:
1.Finaliza ejecución de la función
2.Regresa el control a donde la función fue llamada
En C++,solo las funciones void y main() pueden terminar sin escribir return.

- main() se ejecuta.
- sumar(x, y) es llamada → a = 5, b = 10.
- Se ejecuta return a + b → devuelve 15.
- Ese valor es asignado a resultado.
- cout imprime el resultado.

4. main() como función especial

empieza ejecutando el main.

- main() es una función especial: es el punto de entrada del programa.

```
int main() {
    // ...
    return 0;
}
```

```
int main(int argc, char* argv[]) {
    //para argumentos desde la línea de comandos
}
```

Estructura de memoria para recordar que funciones están activas y dónde debe regresar al terminar la función

- **La pila de ejecución (**call stack**):** Cada vez que se llama una función, se crea un marco **(frame)** en la pila. Cuando la función termina, ese marco se destruye y se retorna al anterior.

```
int doble(int x) {
    return x * 2;
}

int sumarDobles(int a, int b) {
    return doble(a) + doble(b);
}

int main() {
    cout << sumarDobles(3, 4) << endl;
    return 0;
}
```

Stack de llamadas (flujo):

1. main() llama sumarDobles(3, 4)
2. sumarDobles llama doble(3)
3. doble(3) devuelve 6
4. sumarDobles llama doble(4)
5. doble(4) devuelve 8
6. sumarDobles devuelve 14 a main()

5. Paso de parámetro por valor y por fererencia

¿Qué es el paso de parámetros?

Cuando se llama a una función con argumentos, esos valores deben ser recibidos por parámetros. Hay dos formas comunes de hacerlo en C++:

Forma	¿Copia datos?	¿Puede modificar los originales?
Por valor	SI	NO
Por referencia	NO	SI

- **Paso por Valor** (int x) Se pasa una copia del valor. Lo que ocurra dentro de la función no afecta al original.
- **Paso por Referencia** (int& x) Se pasa una referencia al valor original. Lo que ocurra dentro de la función afecta al original.
- Y si no quieres modificar, pero pasas por referencia? Puedes usar const para seguridad contra modificaciones (const int& x)

Ahorra memoria y tiempo ← impide modificaciones.

Ejemplo: Paso por valor (duplicar el valor de un entero)

```
Antes de llamar a la función: 5
Dentro de la función: 10
Después de llamar a la función: 5
```

```cpp
#include <iostream>
using namespace std;

void duplicarValor(int x) {
    x = x * 2;
    cout << "Dentro de la función: " << x << endl;
}

int main() {
    int numero = 5;
    cout << "Antes de llamar a la función: " << numero << endl;
    duplicarValor(numero);
    cout << "Después de llamar a la función: " << numero << endl;
    return 0;
}
```

Ejemplo: Paso por referencia (duplicar el valor de un entero)

```
Antes de llamar a la función: 5
Dentro de la función: 10
Después de llamar a la función: 10
```

```cpp
#include <iostream>
using namespace std;

void duplicarValor(int& x) {
    x = x * 2;
    cout << "Dentro de la función: " << x << endl;
}

int main() {
    int numero = 5;
    cout << "Antes de llamar a la función: " << numero << endl;
    duplicarValor(numero);
    cout << "Después de llamar a la función: " << numero << endl;
    return 0;
}
```

6. Parámetros tipo arreglos.

- En C++, cuando pasamos un arreglo a una función:
  No se copia el arreglo → se pasa la dirección del primer elemento (puntero ??)
  (los arreglos de pasan por referencia automáticamente)
  Por lo tanto, la función puede modificar los elementos originales del arreglo.
  Se pierde el tamaño del arreglo, a menos que lo pases como parámetro adicional.
  - Dentro de la función ya no se sabe cuántos elementos tiene el arreglo

el arreglo se convierte en un puntero al primer
<> int* arr (no es un arreglo)

- **Prototipo típico:**

```
void mostrar(int arr[], int tam);
```

no se conoce el tamaño del arreglo

sol pasar el tamaño por valor.

```
#include <vector>
int main(){
 std::vector<int> v;
 return 0;
}
```

- Alternativa moderna: std::vector — contenedor dinámico de la biblioteca <vector> se comporta como un arreglo
  - Guarda internamente los elementostt
  - ✓Tiene tamaño interno (.size())
  - ✓Paso flexible, por valor o referencia, pero más seguro
  - ✓Más flexible (puede crecer dinámicamente)

Los arreglos tienen tamaño fijo, en cambio los vectores pueden crecer o reducirse dinámicamente

v.plush_back(num);

no es necesario, el compilador lo asume automáticamente

tiempo de ejecución

paso por referencia cte es recomendado

```
#include <vector>
using namespace std;
int main(){
vector<int> v;
return 0;
}
```

Ejemplo: duplicar los elementos de un arreglo

```cpp
#include <iostream>
using namespace std;
// Definición de la función
void modificarArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {arr[i] *= 2;}
}
int main() {
    int miArray[] = {1, 2, 3, 4, 5};
    int tamano = sizeof(miArray) / sizeof(miArray[0]);
    modificarArray(miArray, tamano); // llamada

    cout << "Array modificado: ";
    for (int i = 0; i < tamano; i++) {
        cout << miArray[i] << " ";
    }
    cout << endl;
}
```

operador que devuelve el tamaño en bytes

arreglo

devuelve el tamaño en bytes del primer elemento.

1byte=8bits
Byte: Unidad de información de memoria
Bit: Unidad mínima de información
Solo puede ser 0 o 1.

```
Array modificado: 2 4 6 8 10
```

Parámetros tipo matriz

- Se debe especificar el tamaño de las columnas. El compilador necesita saber cuántas columnas hay para poder hacer el cálculo de desplazamiento correcto: mat[i][j] ↔ *(mat + i × columnas + j)

aritmética de punteros

```
void imprimirMatriz(int matriz[][4]);
```

- En **la llamada** solo se especifica el nombre de la matriz

```
imprimirMatriz(miMatriz);
```

Ejemplo: imprimir una matriz

```cpp
#include <iostream>
using namespace std;
const int FILAS = 3; //Variable global
const int COLUMNAS = 3; //Variable global

// Función para imprime una matriz
void imprimirMatriz(int matriz[][COLUMNAS]) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int miMatriz[FILAS][COLUMNAS] = {{1, 2, 3},{4, 5, 6},{7, 8, 9}};
    imprimirMatriz(miMatriz); //llamada

    return 0;
}
```

La matriz no requiere del número de filas porque la aritmética de punteros solo depende de las columnas.
*(matriz + i x columnas + j)

Si se quiere recorrer la matriz, se necesita número de filas y columnas.

7. Variables y alcance en funciones

- El alcance (scope) de una variable se refiere a dónde puede ser accedida o utilizada en el código. En C++, hay principalmente tres tipos de alcance:

Tipo de variable	¿Dónde vive?	Acceso
Local	Dentro de un bloque {}	Sólo ese bloque
Global	Fuera de cualquier función	Todo el archivo
Estática (static)	Dentro de la función	Sólo la función (mantiene su valor entre llamadas)

- **Variables Locales**: declarada dentro de una función o bloque, se crean al entrar en el bloque y se destruyen al salir.
- **Variables Globales**: declarada fuera de cualquier función. Se pueden usar en todo el archivo fuente.
- **Variable staticas**: conservan su valor entre llamadas

Tiempo de vida de las variables

- Peligros de las variables globales:
  - ✓ Difícil de rastrear errores.
  - ✓ Rompe el encapsulamiento.
  - ✓ Reduce la reutilización de funciones

Cualquier función puede modificarla
Cuando cambia de valor, no se sabe dónde ocurrió.

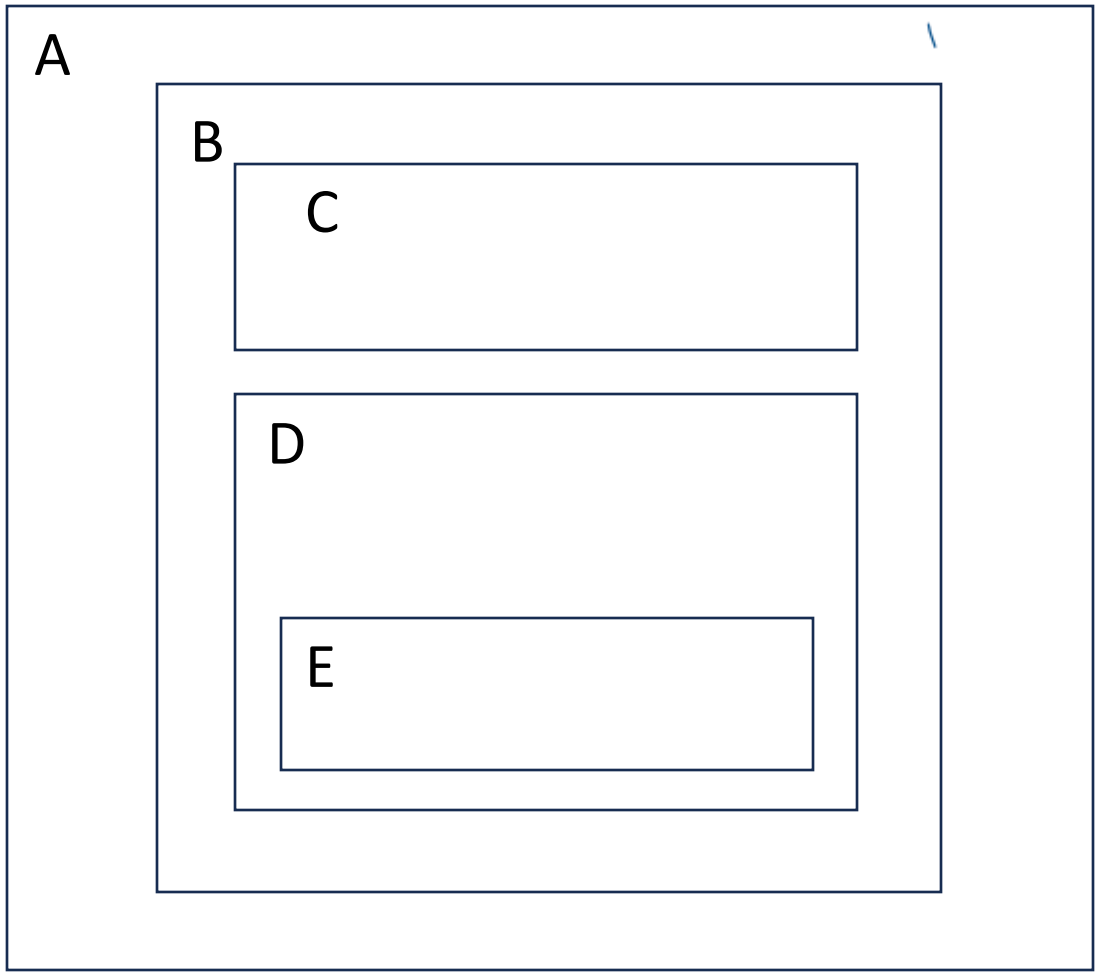
Las funciones dependen de algo externo, no es autónoma.

No se puede reutilizar directamente, se necesita que exista la variable global.

- Reglas de Alcance en C++:
  - ✓ Una variable solo vive en el bloque donde fue declarada.
  - ✓ No se puede acceder a una variable antes de declararla.
  - ✓ Se pueden tener variables con el mismo nombre en bloques distintos.

- Tiempo de vida de las variables:
  - ✓ Las variables locales existen solo mientras el bloque se ejecuta.
  - ✓ Las globales existen durante toda la ejecución del programa.
  - ✓ Las static locales tienen alcance limitado, pero vida extendida

Identificando variables locales y globales



Variables definidas en:	Son accesibles desde:
A	A, B, C, D, E
B	B, C, D, E
C	C
D	D, E
E	E

8. Sobrecarga de funciones (overloading)

- Permite declarar varias funciones con el mismo nombre siempre que tengan diferentes parámetros (en cantidad o tipo) . Ejemplo:

```
int mul(int,int);
float mul(float,int);

int main(){
    int r1 = mul(2,3);
    float r2 = mul(0.2,4);
    return 0;
}

int mul(int a,int b){ return a*b; }
float mul(double x, int y){ return x*y;}
```

```
int sumar(int a, int b);
int sumar(int a, int b, int c);
int multiplicar(int a, int b);
int multiplicar(int a, int b, int c);

int main() {
    cout << sumar(5, 3) << endl;
    cout << sumar(5, 3, 2) << endl;
    cout << multiplicar(5, 3) << endl;
    cout << multiplicar(5, 3, 2) << endl;

    return 0;
}
// implemente las funciones aquí!!
```

mismo parámetro

`int f(int x); double f(int x);` define una sobrecarga?

No es una sobrecarga porque ambas funciones tienen la misma cantidad de parámetros y el mismo tipo de parámetro.

Plantilla de Funciones (templates)

- C++ permite crear funciones genéricas, es decir, una función que puede tomar cualquier tipo de dato como argumento.

- **Sintáxis**:

```
template <typename T>
T duplicar(T valor) {
    return valor * 2;
}
```

Una función template puede ser de cualquier tipo

- **Ejemplo:**

```
template <typename T>
void imprimirGenerico(T dato) {
    cout << "Dato: " << dato << endl;
}
```

```
imprimirGenerico(5);          // int
imprimirGenerico(3.14);       // double
imprimirGenerico("Hola");     // const char
```

9. Recursividad

- Es una técnica de programación en la que una función se llama a sí misma para resolver un problema, dividiéndolo en subproblemas más pequeños. Se compone de:
  - ✓ **Caso base**: define cuándo detenerse. Es decir, ya no se llama así misma.
  - ✓ **Caso recursivo**: define cómo dividir el problema y llamarse a sí misma.

Hace que el problema avance hacia el caso base.

- ¿Como funciona la recursión?
  Cada llamada a la función se apila en la memoria (stack) y se resuelve cuando se alcanza el caso base.

pila de ejecución

Estructura de memoria

La pila funciona como una torre de platos, el último plato que se pone es el primero que se saca.

Cada vez que una función se llama, se crea un marco de activación (guarda variables locales, parámetros, punto al que debe regresar)

En recursión, cada llamada se apila encima de la anterior.

- Importancia del caso base: Si no se define correctamente el caso base, se produce un ciclo infinito y eventualmente un desbordamiento de pila.

Es como poner los platos sin retirar ninguno.

Ejemplo: Factorial recursivo

```
int factorialRecursivo(int n) {
    if (n == 0)
        return 1;
    return n * factorialRecursivo(n - 1);
}
```

¿Como se ejecuta la recursión?

Ejemplo para n = 3:

factorial(3)
→ 3 * factorial(2)
→ 2 * factorial(1)
→ 1 * factorial(0)
→ 1

Otros ejemplos comunes de recursividad:

1. Fibonacci
2. Potencias
3. Inversión de una cadena
4. Suma de elementos en un arreglo
5. mcm, mcd
6. Palabra palíndroma
7. Imprimir números en forma ascendente
8. Sumar los dígitos de un número
9. Contar los dígitos de un número
10. Convertir un número a base binaria

Ejemplo: Factorial con recursividad mutua

En una recursividad mutua (indirecta), una función llama a otra, que a su vez llama de regreso a la primera, creando un ciclo recursivo.

```
// Declaración anticipada
int factorialB(int n);

int factorialA(int n) {
    if (n <= 1)
        return 1;
    return n * factorialB(n - 1);
}

int factorialB(int n) {
    if (n <= 1)
        return 1;
    return n * factorialA(n - 1);
}
```

¿Como se ejecuta la recursión? Ejemplo para n = 5:
Llamando a factorialA(5);

factorialA(5)
→ 5 * factorialB(4)
→ 4 * factorialA(3)
→ 3 * factorialB(2)
→ 2 * factorialA(1)
→ retorna 1 (caso base)

Recursión vs iteración

Repetición con bucles
while
do while
for

Característica	Recursividad	Iteración
Concepto	Función se llama a sí misma	Uso de bucles (for, while)
Uso de memoria	Más consumo (por el stack)	Menor consumo
Código	Más compacto	Más explícito
Velocidad	Generalmente más lenta	Más rápido
Casos ideales	Problemas de naturaleza recursiva	Procesos repetitivos simples

Situaciones donde es más conveniente usar una técnica que otra

No necesita dividirse en subproblemas

Se puede dividir en subproblemas del mismo tipo

Ejemplo: Recursión vs iteración

```
int factorialIterativo(int n) {
    int resultado = 1;
    for (int i = 2; i <= n; ++i) {
        resultado *= i;
    }
    return resultado;
}
```

Toda función recursiva puede transformarse en una versión iterativa.

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

Ejemplo: Factorial con recursividad mutua

En una recursividad mutua (indirecta) Es, una función llama a otra, que a su vez llama de regreso a la primera, creando un ciclo recursivo.

```
// Declaración anticipada
int factorialB(int n);

int factorialA(int n) {
    if (n <= 1)
        return 1;
    return n * factorialB(n - 1);
}

int factorialB(int n) {
    if (n <= 1)
        return 1;
    return n * factorialA(n - 1);
}
```

¿Como se ejecuta la recursión? Ejemplo para n = 5:
Llamando a factorialA(5);

factorialA(5)
→ 5 * factorialB(4)
→ 4 * factorialA(3)
→ 3 * factorialB(2)
→ 2 * factorialA(1)
→ retorna 1 (caso base)