**👨🏽‍💻 C++ 101 – Session 10 Notes**

**Topics Covered:**

* Default Parameters
* Function Overloading
* Function Scope
* Function Recursion
* Assignment (with explanations and sample code)

**🔹 1. Default Parameters**

**❓ What are Default Parameters?**

**Default parameters** allow you to assign **default values** to function arguments **in case the caller doesn't provide them**.

This makes functions more flexible and reduces the need for multiple overloaded versions.

**✅ Syntax:**

void greet(string name = "Guest") {

    cout << "Hello, " << name << "!" << endl;

}

**✅ Usage:**

greet("Ngambo");  // Output: Hello, Ngambo!

greet();          // Output: Hello, Guest!

If no value is provided, the default "Guest" is used.

**📌 Notes:**

* Default values must be given **from right to left**.
* You **cannot** skip a parameter in the middle.

✅ Valid:

void greet(string name = "Guest", string greeting = "Hi") { ... }

❌ Invalid:

void greet(string greeting = "Hi", string name) { ... } // ❌ won't compile

**✅ Another Example with Numbers:**

int multiply(int x, int y = 2) {

    return x \* y;

}

int main() {

    cout << multiply(5) << endl;     // Output: 10 (5 \* 2)

    cout << multiply(5, 3) << endl;  // Output: 15

}

Adding **default parameters** helps simplify your code and reduces the need to overload functions for common use cases.

**🔹 2. Function Overloading**

**❓ What is Function Overloading?**

Function overloading means creating **multiple functions with the same name**, but with **different parameter types or counts**.

The compiler uses the **number and type of arguments** to determine which version of the function to execute.

This makes code easier to read and write, and is a form of **polymorphism** in C++.

**✅ Example:**

int add(int a, int b) {

// Function to sum two integers

int sum(int a, int b) {

    return a + b;

}

// Function to sum two doubles

double sum(double x, double y) {

    return x + y;

}

// Function to sum three integers

int sum(int a, int b, int c) {

    return a + b + c;

}

Now you can use sum() with both integers and doubles:

int main() {

    cout << sum(3, 4) << endl;         // Calls sum(int, int) → 7

    cout << sum(2.5, 4.3) << endl;     // Calls sum(double, double) → 6.8

    cout << sum(1, 2, 3) << endl;      // Calls sum(int, int, int) → 6

    return 0;

}

**⚠️ Rules for Overloading:**

* Functions **must differ** by **number or types** of parameters.
* Return type **alone is NOT enough** to distinguish functions.

// ❌ Invalid - return type alone does not overload a function

int show() {}

string show() {}  // ERROR!

**🔹 3. Function Scope**

**❓ What is Scope?**

**Scope** defines where a variable can be **accessed or used** in your program.

There are **two main types** of scope in C++:

**🔸 Local Scope**

Variables declared **inside** a function or block {} can only be accessed **within** that function.

void printAge() {

    int age = 25;  // local variable

    cout << age;

}

Trying to access age outside printAge() will cause an error.

**🔸 Global Scope**

Variables declared **outside all functions** are **global** and accessible from **any function** in the file.

int age = 30;

void showAge() {

    cout << age; // OK

}

int main() {

    cout << age; // OK

}

**⚠️ Best Practice:**

Avoid using global variables when possible. Use local scope to prevent bugs and make code easier to understand.

**🔹 4. Function Recursion**

**❓ What is Recursion?**

A **recursive function** is a function that **calls itself** to solve a smaller version of a problem.

**🔁 Two Key Parts:**

1. **Base Case** – When to **stop** the recursion
2. **Recursive Case** – When to **call the function again**

**✅ Example: Sum of numbers from 1 to n**

int sum(int a) {

    if (a == 0) {

        return 0;  // base case

    } else {

        return a + sum(a - 1);  // recursive call

    }

}

So, sum(5) becomes:

5 + sum(4)

→ 5 + 4 + sum(3)

→ 5 + 4 + 3 + sum(2)

→ ...

→ 5 + 4 + 3 + 2 + 1 + 0 = 15

**🧠 How Recursion Works:**

When a function calls itself:

* A **new frame** is added to the **call stack**
* The program **waits** for the innermost call to finish
* Then it **"unwinds"** and combines the results

⚠️ Without a **base case**, recursion will lead to **infinite loops** or **stack overflow errors**.

**📌 Assignment**

**🔸 1. Why does the following code give an error?**

#include <iostream>

using namespace std;

int printArray(int myArr[5]) {

    for (int i = 0; i < 5; i++) {

        cout << myArr[i] << " ";

    }

    cout << endl;

    return 0;

}

int main() {

    printArray({1, 2, 3, 4, 5});  // ❌ Error!

    return 0;

}

### 🔹 ****2. Recursive Factorial Function****

Write a C++ program that:

* Prompts the user to enter a number
* Uses a **recursive function** to calculate the factorial of that number
* Displays the result to the user

📌 **Reminder:**  
Factorial of 5 is 5 \* 4 \* 3 \* 2 \* 1 = 120

**🧠 Summary**

| **Concept** | **Key Takeaway** |
| --- | --- |
| **Function Overloading** | Same function name, different parameter types or counts |
| **Scope** | Controls where variables are accessible (local vs global) |
| **Recursion** | A function that calls itself to solve smaller problems |
| **Default Parameters** | Let you assign default values to parameters so function calls can be shorter or more flexible |