Lecture16 – Master Functions | Declaration, Definition, Call , Parameter and Return Types

1. Introduction to Functions

Problem Statement

- **Issue**: Repetitive code is bulky, unreadable, and buggy
- **Example**: Printing "welcome to world of beautiful flowers" multiple times requires writing the same code repeatedly
- **Solution**: Functions allow us to write code once and call it multiple times

Why Functions over Loops?

- **Loops**: Good for repetitive tasks in sequence
- **Functions**: Better when you need to perform the same task at different points in your program
- **Example**: Print star pattern → print "good morning" → print star pattern again

What is a Function?

- A **reusable block of code** that performs a specific task
- Takes input (parameters) and may return output
- Makes code **modular, readable, and maintainable**

Basic Syntax

```
returnType functionName(parameter1, parameter2, ...) {
   // Function body
   return value;  // if return type is not void
}
```

Simple Example

```
#include<iostream>
using namespace std;
void greet() {
    cout << "Hello, welcome to Khanam Coding!" << endl;
}
int main() {
    greet(); // Function call
    return 0;
}
```

2. Understanding Void Data Type

Void Characteristics

- **Stores**: Nothing
- **Memory size**: 0 bytes
- **Usage**: Primarily for function return types
- **Meaning**: Function doesn't return any value

Example

```
void greet() {
    cout << "Hello, World!";
    // No return statement needed
}
```

3. Types of Functions

A. Library Functions (Predefined)

- **Definition**: Functions already defined in C++ libraries

- **Requirement**: Must include appropriate header files
- **Examples**:
  - cout, cin → from <iostream>
  - sqrt(), pow() → from <cmath>
  - strlen(), strcpy() → from <cstring>

```
#include <cmath>
int num = 16;
cout << sqrt(num); // Output: 4
```

B. User-defined Functions

Functions created by programmers for specific tasks.

Types of User-defined Functions:

Type	Example	Description
No arguments, no return	void greet()	Simple task execution
With arguments, no return	void printSum(int a, int b)	Process input, no output
With arguments, with return	int add(int a, int b)	Process input, return result
With default arguments	int add(int a, int b = 5)	Optional parameters
Inline functions	inline int square(int x)	Performance optimization
Recursive functions	Function calls itself	Complex problem solving

C. Built-in Functions

- **Definition**: Functions defined within the C++ compiler
- **Requirement**: No header files needed
- **Examples**: Operators (+, -, *, /), ternary operator

4. Function Declaration, Definition & Call

Three Key Concepts

1. **Declaration**: Function prototype (tells compiler about function)
2. **Definition**: Actual function code implementation
3. **Call**: Using the function in your program

Example

```
#include<iostream>
using namespace std;
void showMessage(); // Declaration

int main() {
   showMessage(); // Call
   return 0;
}

void showMessage() { // Definition
   cout << "This is a user-defined function!" << endl;
}
```

Important Notes

- **Main function**: return 0 is optional in C++ (automatic)
- **Other functions**: Must manually return values if return type is not void
- **Function placement**: Declaration must come before usage

5. Function Parameters and Return Types

Function with Parameters (No Return)

```cpp
void add(int a, int b) {
    cout << "Sum: " << a + b << endl;
}

int main() {
    int x = 5, y = 15;
    add(x, y);  // Pass by value
    return 0;
}
```

Function with Return Value

```cpp
int multiply(int a, int b) {
    return a * b;
}
int main() {
    int result = multiply(4, 5);
    cout << "Multiplication: " << result << endl;
    return 0;
}
```

Practical Example: Even/Odd Check

```cpp
bool isEven(int num) {
    return (num % 2 == 0);
}
```

6. Function Overloading

Multiple functions with the same name but different parameters.

```
void print(int i) {
    cout << "Integer: " << i << endl;
}
void print(double d) {
    cout << "Double: " << d << endl;
}
void print(string s) {
    cout << "String: " << s << endl;
}
int main() {
    print(10);      // Calls int version
    print(5.5);     // Calls double version
    print("Hello"); // Calls string version
    return 0;
}
```

7. Function Call Stack

Concept

- **Memory structure** that tracks function calls
- **LIFO (Last In, First Out)** principle
- **Push**: Add function to stack when called
- **Pop**: Remove function when completed

Example Flow

```
void funC() { cout << "Inside funC" << endl; }
void funB() { cout << "Inside funB" << endl; funC(); }
void funA() { cout << "Inside funA" << endl; funB(); }
```

```
int main() {
    cout << "Inside main" << endl;
    funA();
    return 0;
}
```

Stack Execution:

1. **main()** → Stack: [main]
2. **funA()** called → Stack: [main, funA]
3. **funB()** called → Stack: [main, funA, funB]
4. **funC()** called → Stack: [main, funA, funB, funC]
5. **funC()** completes → Stack: [main, funA, funB]
6. **funB()** completes → Stack: [main, funA]
7. **funA()** completes → Stack: [main]
8. **main()** completes → Stack: []

8. Pass by Value

Concept

- Function receives **copies** of arguments
- Original variables remain **unchanged**
- Changes inside function don't affect original values

Example: Value Change

```
void changeValue(int x) {
    x = x + 10;
    cout << "Inside function, x = " << x << endl; // 15
```

```
}
int main() {
    int a = 5;
    changeValue(a);
    cout << "Inside main, a = " << a << endl; // Still 5
    return 0;
}
```

Example: Failed Swap

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
    // Only copies are swapped, not originals
}
```

9. Variable Scoping

Types of Scope

1. Local Scope

- Variables declared inside functions/blocks
- Only accessible within that function/block

```
void show() {
    int a = 10;  // Local variable
    cout << a;   // Works
}
// cout << a; // Error - not accessible here
```

2. Global Scope

- Variables declared outside all functions
- Accessible throughout the entire program

```
int x = 100;  // Global variable

void fun() {
    cout << x;  // Accessible
}
int main() {
    cout << x;  // Also accessible
    return 0;
}
```

3. Function Scope

- Function parameters have scope limited to that function

```
void greet(string name) {  // 'name' only accessible in greet()
    cout << "Hello, " << name << endl;
}
```

4. Block Scope

- Variables in if, while, for blocks

```
int main() {
    if (true) {
        int x = 50;  // Block scope
        cout << x;   // Works here
    }
    // cout << x;  // Error - not accessible here
```

```
    return 0;
}
```

Scope Rules

- **Same block**: Cannot have variables with same name
- **Different blocks**: Can reuse variable names

10. Inline Functions

Purpose

- **Optimization technique** to reduce function call overhead
- Compiler replaces function calls with actual function code
- Improves performance for small, frequently called functions

How It Works

1. Write function with inline keyword
2. During compilation, function calls are replaced with function body
3. No actual function call happens at runtime

Example

```
inline int getMax(int a, int b) {
    return (a > b) ? a : b;
}
int main() {
    int x = 5, y = 10;
    int result = getMax(x, y);  // This gets replaced with: (x > y) ? x : y
    return 0;
}
```

Important Notes

- **Best for**: Single-line or very small functions
- **Compiler decision**: Compiler may ignore inline request for large functions
- **Trade-off**: Reduces function call overhead but may increase code size

When Compiler Accepts Inline

- ✅ **Single line functions**
- ✅ **2-3 line simple functions**
- ❌ **Large functions (3+ lines of complex code)**

11. Default Arguments

Concept

- Provide default values for function parameters
- If argument not passed, default value is used
- **Rule**: Default arguments must be rightmost parameters

Example

```
void welcome(string name = "Guest") {
    cout << "Welcome, " << name << endl;
}
int main() {
    welcome("Khaiser");  // Output: Welcome, Khaiser
    welcome();           // Output: Welcome, Guest
    return 0;
}
```

Multiple Default Arguments

```
void printInfo(string name, int age = 18, string city = "Unknown") {
```

```
    cout << name << ", " << age << ", " << city << endl;
}

// Valid calls:
printInfo("Khanam");                    // Khanam, 18, Unknown
printInfo("Reema", 25);                 // Reema, 25, Unknown
printInfo("Seema", 30, "New York");     // Seema, 30, New York
```

Invalid Default Argument Placement

```
// ❌ Wrong: Non-default parameter after default parameter
void func(int a = 5, int b);  // Error!

// ✅ Correct: Default parameters at the end
void func(int b, int a = 5);  // OK
```

12. Key Benefits of Functions

1. Code Reusability

- Write once, use multiple times
- Reduces code duplication

2. Modularity

- Break large problems into smaller parts
- Easier to debug and maintain

3. Readability

- Code becomes more organized
- Function names describe what code does

4. Maintainability

- Changes needed in only one place

- Easier to update and fix bugs

5. Testing

- Individual functions can be tested separately
- Easier to identify issues

13. Best Practices

Function Design

1. **Single Responsibility**: Each function should do one thing well
2. **Meaningful Names**: Function names should describe their purpose
3. **Parameter Limit**: Avoid too many parameters (max 3-4 recommended)
4. **Return Values**: Use return values instead of global variables when possible

Code Organization

1. **Declare Before Use**: Function declarations should come before usage
2. **Group Related Functions**: Keep similar functions together
3. **Comment Complex Functions**: Add comments for complex logic

Performance Considerations

1. **Use inline for small functions**: Optimize frequently called small functions
2. **Pass by reference for large objects**: Avoid copying large data
3. **Avoid deep recursion**: Can cause stack overflow

Summary

✅ **Functions are reusable code blocks**
✅ **Improve code modularity and readability**
✅ **Support parameters and return types**
✅ **Function overloading enables multiple versions**

- ✅ **Pass-by-value creates copies of arguments**
- ✅ **Variable scoping determines accessibility**
- ✅ **Inline functions optimize performance**
- ✅ **Default arguments provide flexibility**