SWEG2102 Fall 2026

Fundamentals of Programming II

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

Lecture 02

Modular Programming (Part I)
Introduction to Function

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

01 Overview of Modular Programming

02 Basic Concepts of Function

03 Function Declaration vs Definition

04 Function Call and Return

05 Pitfalls & Best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Explain modular programming and its benefits
- Describe functions and identify their components
- Distinguish between function declaration vs definition
- Write and call functions with proper return types
- Build a small modular program using functions

Overview of Modular Programming

Overview of Modular Programming

Core Concept

Modular programming is a software design technique where a large, complex **program** is divided into smaller, independent, focused and reusable units called **modules**.

- 👉 **Modules** are a self-contained unit of code that performs a **specific task**.
- 👉 E.g. function, class, file, library.

👉 **How do you solve a big/complex problem?**

*E.g. You are tasked to organize a **full tech festival** with in two weeks.*

- ➢ *Where do you start?*
- ➢ *Can you handle alone at once?*

Cont'd

Scenario: tech festival organization

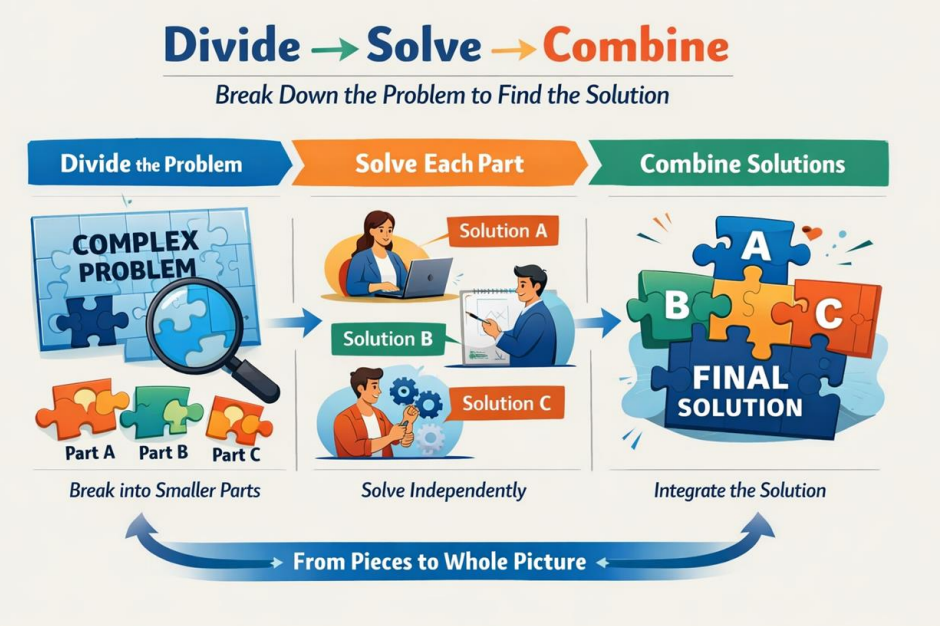
👉 **Most likely not**

👉 **You will not handle it alone**

- ***Organizing a full tech festival involves several tasks and requires coordination, planning, and teamwork***

👉 **So what should you do?**

- Break down tasks into **small, focused and manageable** activities?
- Assign responsible person / team for each task.



Source: AI-generated image (ChatGPT)

Cont’d

Why Modular Programming?

Problem: **Monolithic Program** (all code in one block)

- Hard to understand
- Difficult to debug
- Challenging to maintain
- Low reusability
- Poor collaboration support

Solution: **Modular Programming**

👉 We solve **complex programming problems** by dividing a **large monolithic program** into smaller, independent units called **modules**.

Cont'd

Analogy: Computer System

A Monolithic Architecture

- Computer system where everything is fused into one single chip:
  - ✓ CPU, memory, storage, and input/output are all tightly combined
  - ✓ If one part fails, the entire system is affected
  - ✓ Upgrading one function requires redesigning everything

Monolithic program is like building a computer with monolithic architecture

Modular Architecture

- A real computer system built from **separate components.**
- Each component (CPU, Memory, Disk, GPU, I/O devices etc.):
  - *Has a specific role*
  - *Can be replaced or upgraded independently*
  - *Works together through defined interfaces*

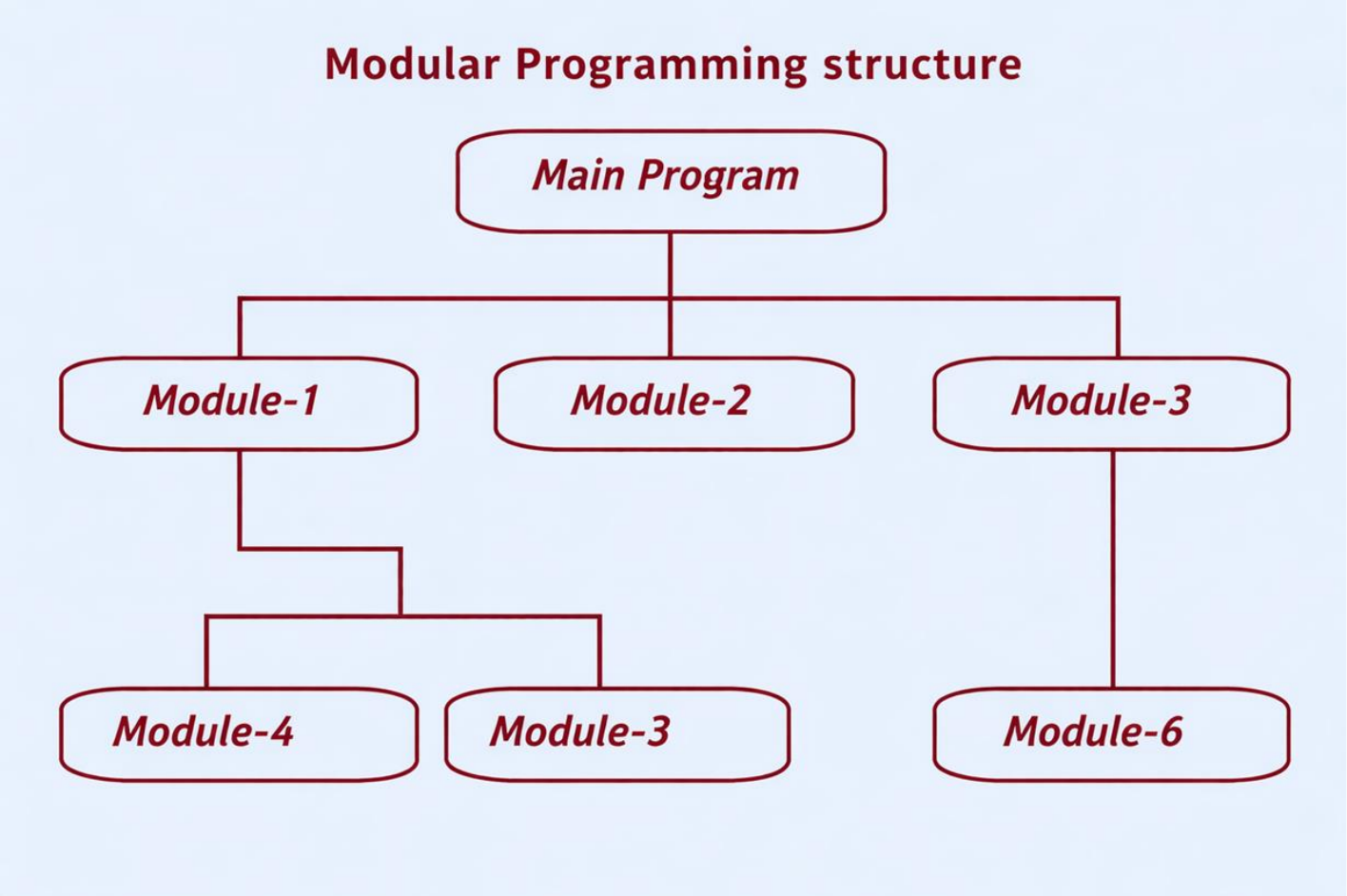
- **The same is true for Modular programming**

Cont’d

Key Benefits Modular Programming

- **Readability:** Clear separation of concerns makes code easier to understand.
- **Reusability:** Call the same function from multiple places; write once, use often.
- **Testability:** Verify individual functions in isolation (unit testing) to catch bugs early.
- **Maintainability:** Change implementation details of one module without touching others.
- **Scalability:** Systems can grow easily by adding new features without redesign
- **Collaboration**: Multiple developers can work on different modules simultaneously.

Cont’d



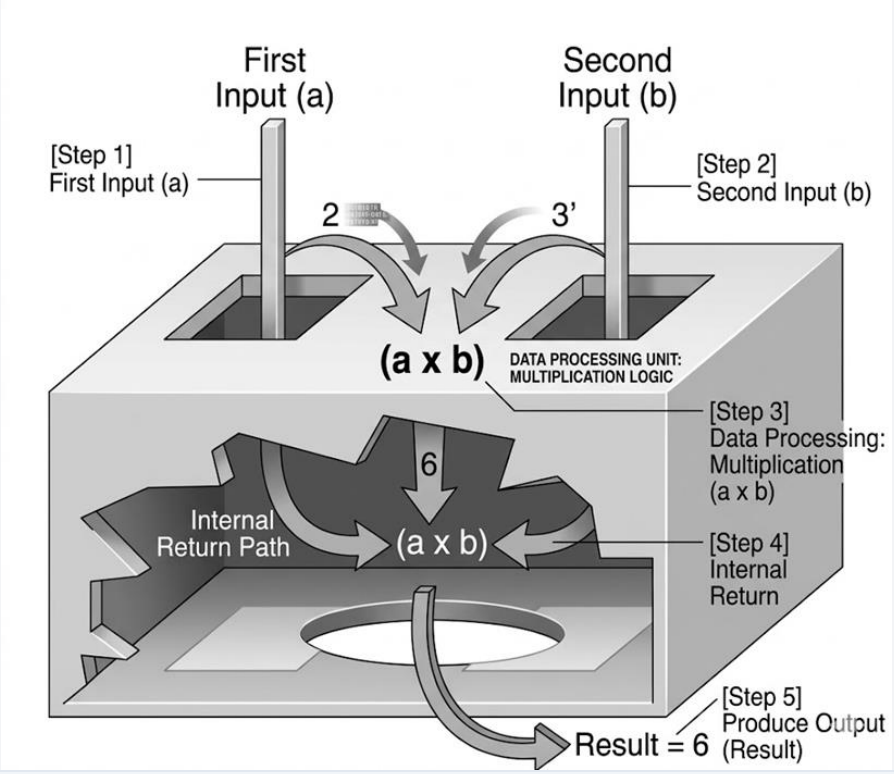
Source: *https://www.sankalandtech.com/Tutorials/C/modular-programming-c.html*

Basic Concepts of Functions

1. What is a Function?

- A **function** is a **named**, **self-contained** block of reusable code
- It performs a specific task.
- Each function can be **called** in turn when it needed

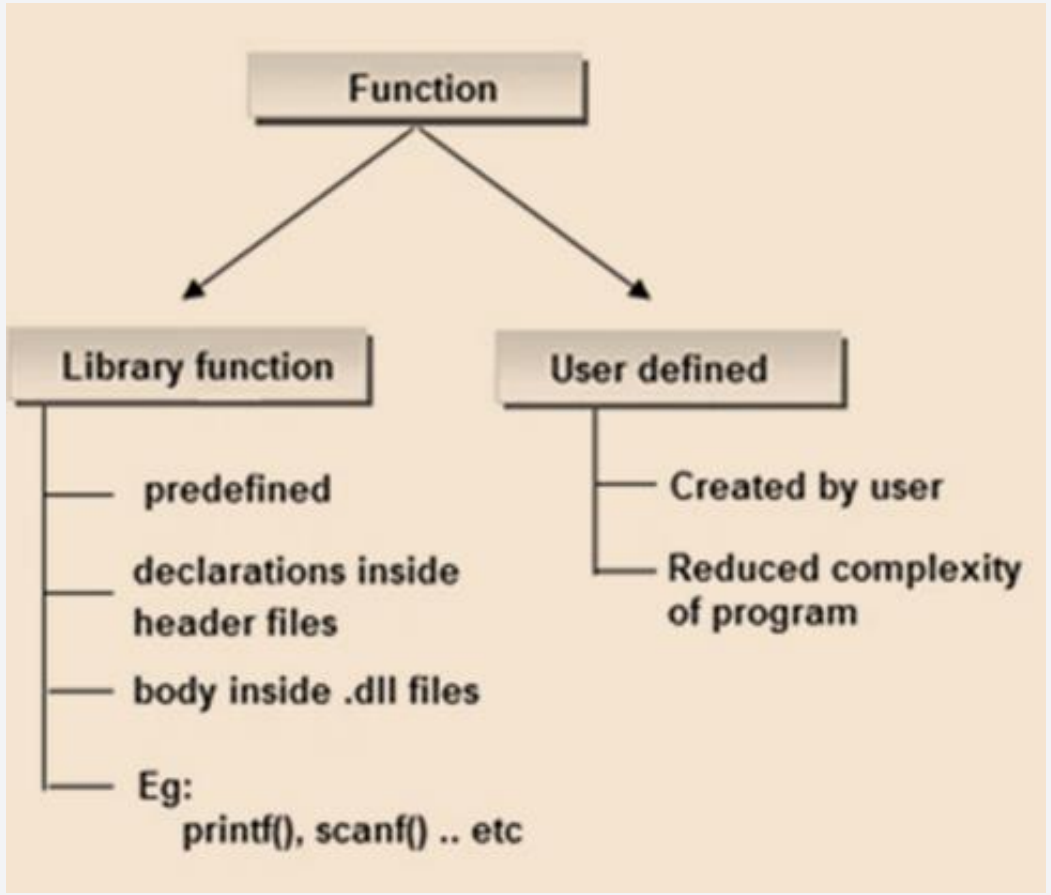
A function can accepts an ***input, act on data (process the input)*** and ***return a value*** (produces an output).



Source:
AI-generated image (Gemini 3)

Cont'd

Types of Functions



Note:

- Most **programming language** allow programmer to **define their own function**.
- So, programmer can **groups code** to perform a specific task and given a name (identifier).

Source: *https://learnfrenzy.com/programming/c-programming/c-functions/*

Cont'd

Anatomy of a Function

General Syntax

```
return_type funcName (param-1, param-2, param-n) {
        // function body statements
}
```

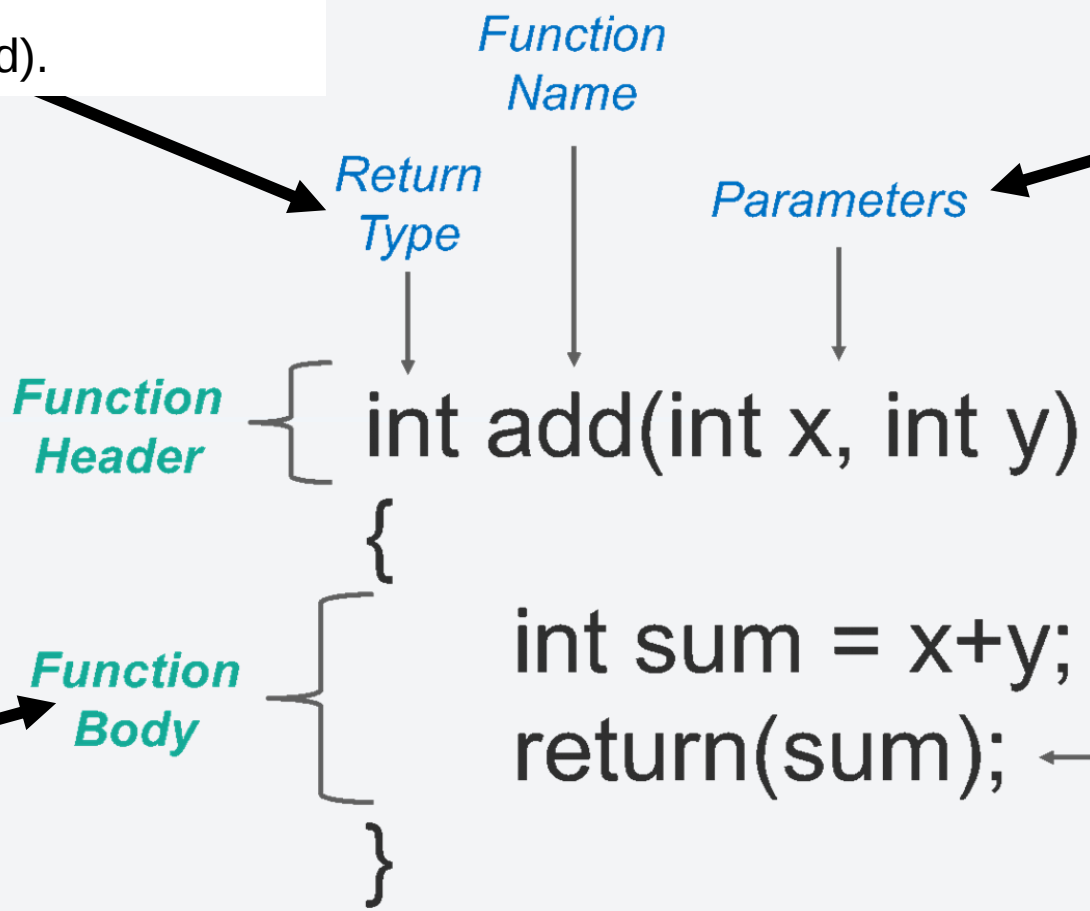
1. **Name:** *A unique, descriptive identifier/name used to invoke the function.*
2. **Parameters:** *Optional input values (type + name) passed to the function when called*
3. **Return Type:** *The data type of the output value sent back (or void if none).*
4. **Body:** *The sequence of statements enclosed in { } that executes the logic.*

Cont'd

- A function may return a value.
- It refers to the data type of the value the function returns.
- It is optional (void).

- The name of function it is decided by programmer
- Should be meaningful valid identifier

- A value which is pass in function at the time of calling of function
- It is like a placeholder.
- It is optional.
- Parameter identifier is also optional



- Value returned by the function
- Single literal or expression

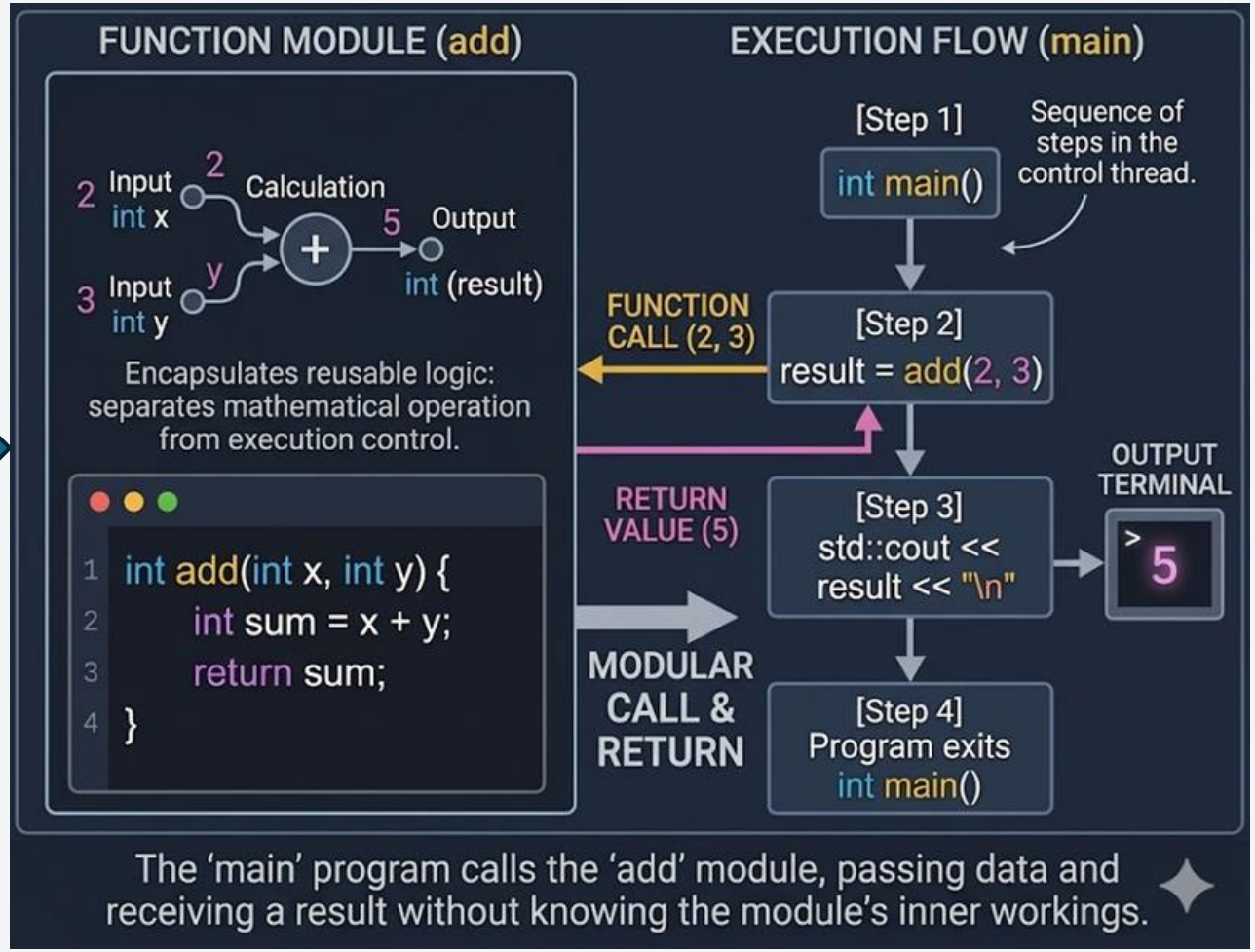
The collection of statements

Cont’d

C++ Example

example.cpp

```cpp
#include <iostream>

int add (int x, int y) {
        int sum = x + y;
        return sum;
}

int main() {
        int result = add(2, 3);
        std::cout << result << "\n";
        // Output: 5
}
```



Source: *AI-generated image (Gemini 3)*

2. Function: Declaration Vs. Definition

Declaration (Prototype)

- Tells the compiler about a function's name, return type, and parameters.
- It has **no body** and ends with a semicolon.
- Often placed in header files.

Definition (Implementation)

- Provides the actual **body** of the function enclosed in braces { ... }.
- This is where the **code logic resides** and memory is allocated.

C++ Example

square.cpp

```cpp
#include <iostream>
// 1. Declaration (Prototype)
int square (int n); // Semicolon!

int main() {
        return square(5); // Call
 }

// 2. Definition (Implementation)
int square (int n) {
        return n * n;
 }
}
```

Cont'd

Three ways to Declare Function

👉Using a Separate Header File:

- *Place function prototypes in a separate file (commonly a header file like .h), and include it in your program using the #include directive.*

👉 Writing Prototypes Inside the Program:

- *Function prototypes can also be written at the top of the same program file, before main()*

👉 Combining Declaration with Definition:

- *If the function is defined before it is called, then a separate prototype is not needed.*
- *A function definition also acts as its declaration*

Cont’d

C++ Example 1

Function Declaration in a separate Header file

header.h

```
#include <iostream>

// Declaration (Prototype)
double calcAverage(int data[], int n);
```

averageCalc.cpp

```
#include <iostream>
#include"header.h"

int main() {
    double avg = calcAverage(data, 5);
    std::cout << avg; // output
}

double calcAverage(int data [], int n) {
    double sum = 0;
    for(int i = 0; i < count; i++) {
        sum += data[i];
     }
return sum / count;
}
```

Cont'd

C++ Example 2: Function Declaration

- Writing Prototypes Inside the Program
- Combining Declaration with Definition

example.cpp

```cpp
#include <iostream>

int product (int x, int y);  // Function prototype

// definition + declaration
int add (int a, int b) {
        return a + b;
}
int main() {
        std:: cout << add (2, 3);
        std:: cout<<product(5, 7);
}

// Function definition
int product (int x, int y) {
        return x * y;
}
```

Cont’d

🧩 Prototype Purpose & Usage

Function prototypes enable the **separation** of interface and implementation:

👉 **Forward Declaration:**

- *Allows calling a function before its full definition appears in the code.*

👉 **Separate Compilation:**

- *Enables placing declarations in **header files (.h)** and definitions in source files.*

👉 **Modular Design:**

- *Promotes code organization and reusability across multiple files.*

3. Function Calls

Core Concept

- A **function call** is the act of invoking a function to execute its defined task.
- When the function called, the program control transfers from the caller (**e.g., main()**) to the called function.

Syntax

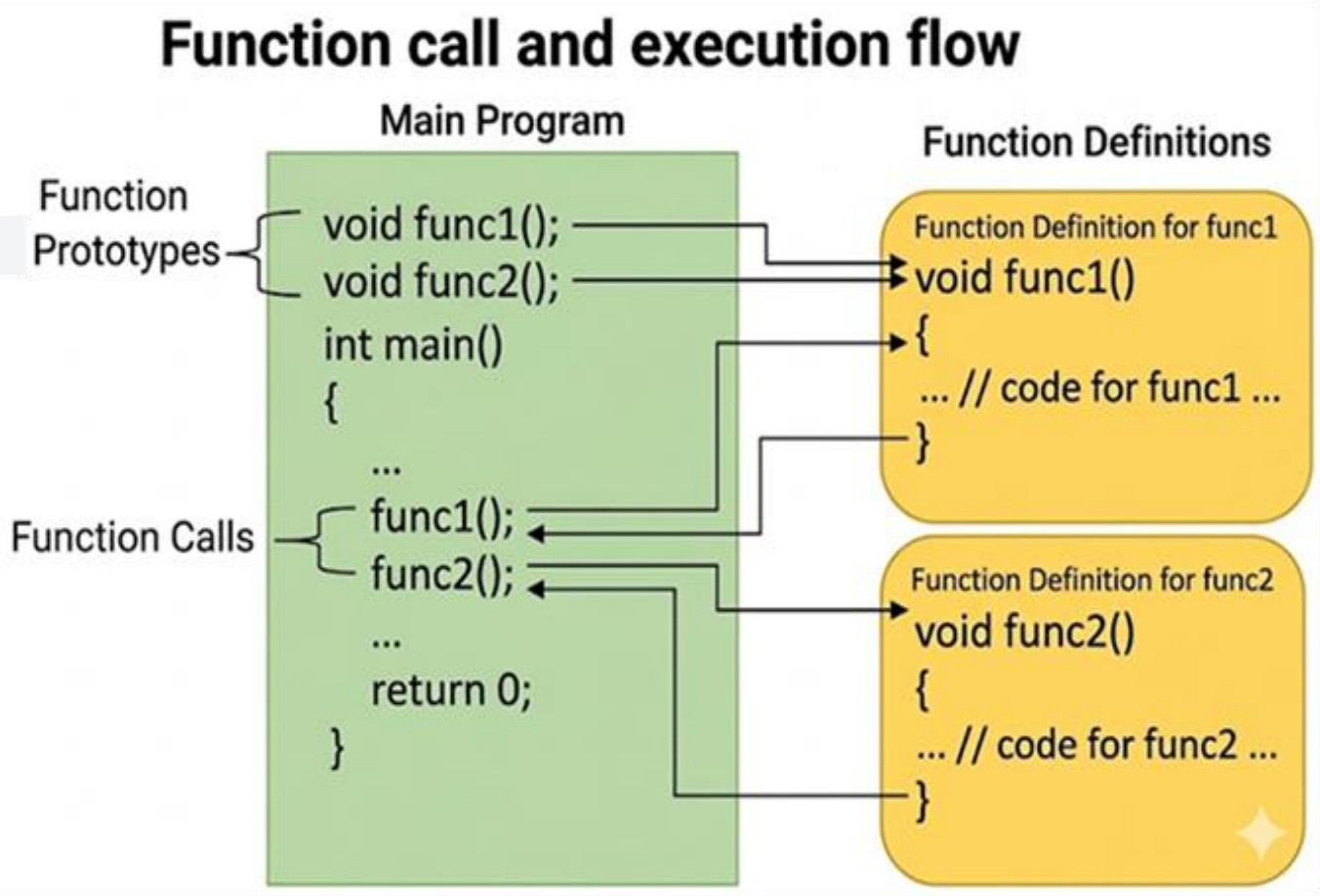
```
function_name (arguments);
```

Optional

Cont'd

When a function is called

1. The program pauses execution at the call point
2. Program control transfers to the called function
3. If any, arguments are passed to parameters
4. Function body executes
5. A value (if any) is returned
6. Control goes back to the caller, to the point execution paused.



Source: *Re-created using AI (ChatGPT)*

Cont'd

Types of Function Calls

Depending on how it is defined, Function calls can be

- Function Call Without Parameters
- Function Call With Parameters
- Function Call in Expression
- Nested Function Calls

Important Rules

- *Function must be declared before calling*
- *Number and type of arguments must match parameters*
- *Parentheses () are mandatory during function call*

example.cpp

```cpp
#include <iostream>

void greeting () { // Function without parameters
   cout << "Welcome to Functions!" << endl;
}

int add (int a, int b) { // Function with parameters
    return a + b;
}

int square (int x) {  // Function used in expression
    return x * x;
}

int main() {

    greeting();  // Function Call Without Parameters

    int sum = add (5, 10); // Function Call With Parameters

    cout << "Sum = " << sum << endl;

    // Function Call in Expression
     cout << "Square of 4 = " << square(4) << endl;

    // Nested Function Calls
    cout << " Result = " << add (square(2), square(3));

    return 0;

 }
```

4. Return Types & Return Statements

👉 Return Types

- Specifies the **data type** of the **value** a function returns to the caller.
- A function's return type can be:
  - ✓ Primitive data types (e.g., int, float, etc.)
  - ✓ Complex types (e.g., string, arrays via pointers, structures)
  - ✓ void (no value is returned)

👉 Return Statements

- Sends a value back to the caller
- Immediately terminates function execution
- It can also be **used early** (conditionally) to exit a function before reaching the end.
- A return statement is:
  - ✓ Mandatory in non-void functions
  - ✓ Optional in void functions

Golden Rule:

- ➢ Returned value must match return type.
- ➢ A function can return exactly **one** value

Guideline:

- ➢ Choose the **data type** that best represents the ***single piece*** of data your function computes or produces.

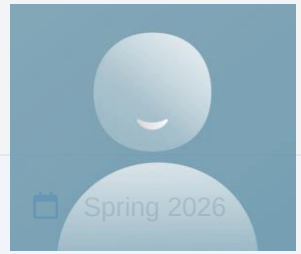
Cont'd

Example

example.cpp

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

// Function with int return type
int computeSum (int a, int b) {
    return a + b;      // return statement
}

// Function demonstrating early return
int safeDivide (int a, int b) {
    if (b == 0) {
        cout << "Error: Division by zero!" << endl;
        return -1;      // early return
    }
    return a / b;      // integer division
}

// Void function (no return value)
void displayResult (int value) {
    cout << "Result = " << value << endl;
    return;            // optional in void function
}

int main() {
    int x = 10, y = 5;

    // Function call with return type int
    int sum = computeSum(x, y);
    displayResult (sum);

    // Function call using expression
    displayResult(computeSum(7, 3));

    // Early return demonstration
    int result = safeDivide(10, 0);
    cout << "Safe Divide (10/2): " << result << endl;

    return 0;
}
```

Practical Exercises



Practice Exercise

Exercise #1 — Temperature Converter

Task Description

- Build a modular temperature converter system by implementing
  - ✓ double cToF(double c) & double fToC(double f)
- Write an output function to format results cleanly.

Hints

- Celsius to Fahrenheit: F = C * 9.0/5.0 + 32.0
- Fahrenheit to Celsius: C = (F - 32.0) * 5.0/9.0
- Use 9.0 instead of 9 to avoid integer division.
- Maintain single responsibility: **compute** vs **print**.

Cont’d

Solution #1: Temp Converter

temp_converter.cpp

```
#include <iostream>

//Function Signatures/Declaration
double cToF(double c);
double fToC(double f);
void printTemp(double temp, char unit);

int main() {

double c = 25.0;
double f = cToF(c);
printTemp(f, 'F');   //Expected Output: 77.0 F
double f2 = 100.0;
double c2 = fToC(f2);
printTemp(c2, 'C'); //Expected Output: 37.77 C
return 0;
}

// Pure function: no I/O inside
double cToF(double c) {
        return (c * 9.0 / 5.0) + 32.0;
}

// Decoupled logic
double fToC (double f) {
        return (f - 32.0) * 5.0 / 9.0;
}
```

Cont’d

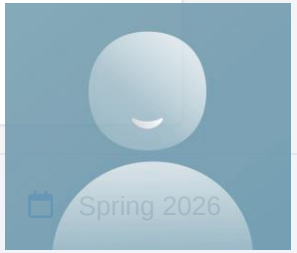
Exercise #2— Prime Checker

Task Description

- Write a function bool isPrime(int n) that determines whether a given integer is a prime number.
- The function returns **true** if prime, **false** otherwise.
- *Assume $n \geq 2$.*

Hints

- A prime number is only divisible by 1 and itself.
- Use a loop to test divisibility with the modulo operator (%).
- **Optimization:** You only need to check for divisors up to sqrt(n).
- Return **false** immediately if any divisor is found (early exit).



Cont’d

Solution #2: Prime Checker

temp_converter.cpp

```cpp
#include <iostream>

bool isPrime(int n); //Function Declaration

int main() {
cout << "isPrime(7): " << isPrime(7) << endl;
// Expected Output: 1 (true)

cout << "isPrime(8): " << isPrime(8) << endl;
// Expected Output: 0 (false)

cout << "isPrime(25): " << isPrime(25) << endl;
// Expected Output: 0 (false)

return 0;
}
```

```cpp
// Pure function: no I/O inside
bool isPrime (int n) {

if (n < 2)  return false; // Handle edge cases

// Optimized trial division
for (inti = 2; i * i <= n; i++) {
        if (n % i == 0) {
                return false;
        }
}
return true;
}
```

Common Pitfalls and Best Practices

Common Pitfalls

- ✗ Over modularization
- ✗ Poor function naming
- ✗ Prototype mismatch, calling function with wrong arguments
- ✗ Calling before prototype / missing declaration
- ✗ Returning wrong type
- ✗ Missing return in non-void function and trying to return value in void function

Best Practices

- ✓ Avoid over-modularization
- ✓ Use descriptive/ meaningful function names
- ✓ Keep functions **small** and follow **Single Responsibility Principle (SRP)**
- ✓ Always declare function before use (or use header/prototype properly)
- ✓ Definition must exactly match declaration
- ✓ Avoid mismatch between arguments and parameters
- ✓ Ensure return value matches return type

Quick Check

1. **Why do we use functions instead of writing everything in main()?**
   A. Faster execution
   B. Better hardware usage
   C. Better structure and reuse
   D. Required by compiler

Answer: C (modularity)

2. **When a function is called, what happens to main()?**
   A. It terminates
   B. It pauses temporarily
   C. It runs in parallel
   D. It is deleted

Answer: B (Control temporarily transfers)

3. **Given `int f() { return 3.9; }`, What happens?**
   A. Returns 3.9 exactly
   B. Compiler error
   C. Returns 3 (truncated)
   D. Runtime crash

Answer: C
(implicit conversion)

References

Test Books

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia