1. Introduction of Programming

Syeda Jannatul Naim
Lecturer | Dept. of CSE
World University of Bangladesh (WUB)

January, 2025

Content

1. History of Programming
2. Structure/Procedure Oriented Programming and Object Oriented Programming
3. Program Development Process
    a. Stage-1: program design
        i. Problem analysis
        ii. Outline the program structure
        iii. Algorithm development
        iv. Selection of control structures
    b. Stage-2: program coding
    c. Stage-3: program testing and debugging
        i. Types of errors
        ii. Program testing: unit and integration testing
        iii. Program debugging
    d. Stage-4: Program Efficiency (during algorithm development)
        i. Execution time (time complexity)
        ii. Memory requirements (space complexity)
4. History of C++ language
5. Characteristics and Importance of C++ language
6. Where should use C++ language and where should not

1. History of Programming

Programming is the process of instructing computers to perform specific tasks by writing sets of instructions called programs or code. It's a form of problem-solving where we translate human logic into a language computers understand.

1. Machine Language (1st Generation)

- Written in **binary (0s and 1s)**
- Very fast but **extremely difficult** for humans

Example:

```
10101010 00010101
```

2. Assembly Language (2nd Generation)

- Uses mnemonics like `ADD`, `MOV`
- Still machine-dependent

Example:

```
MOV A, B
ADD A, C
```

3. High-Level Languages (3rd Generation)

- Easy to read and write
- Portable across machines

Examples:

- **C** (1972)
- FORTRAN
- COBOL
- Pascal

4. Object-Oriented Languages (4th Generation)

- Uses objects and classes
- Better for large software

Examples:

- C++
- Java
- Python

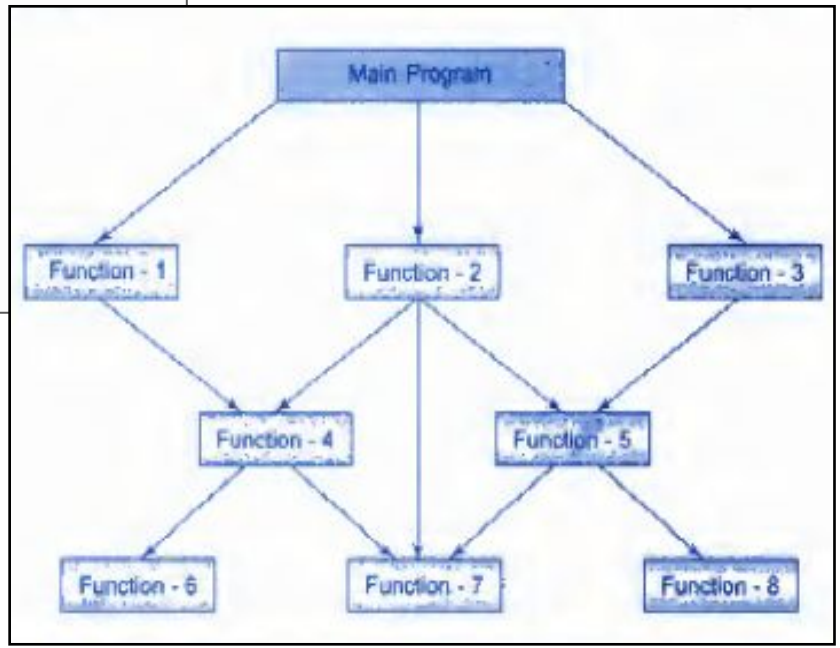
2. Procedure Oriented Programming and Object Oriented Programming

Structured/Procedural Programming

Philosophy: "Divide and conquer" - break problems into functions/procedures

Characteristics:

- Top-down approach
- Functions operate on data
- Global and local variables
- Examples: C, Pascal, FORTRAN



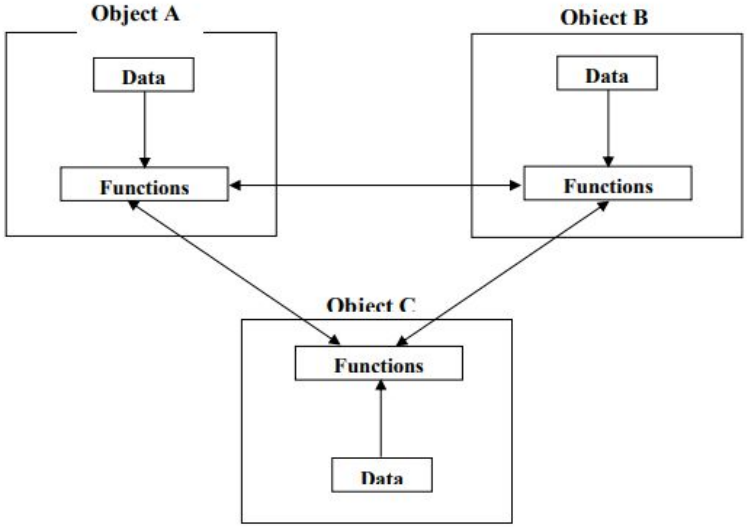
2. Procedure Oriented Programming and Object Oriented Programming

Object-Oriented Programming (OOP)

Philosophy: Model real-world objects with attributes and behaviors

Four Pillars:

1. **Encapsulation:** Bundling data and methods
2. **Inheritance:** Creating new classes from existing ones
3. **Polymorphism:** Same interface, different implementations
4. **Abstraction:** Hiding complex implementation



2. Procedure Oriented Programming and Object Oriented Programming

Aspect	Procedural	Object-Oriented
Approach	Top-down	Bottom-up
Focus	Functions/Procedures	Objects/Classes
Data Security	Less secure (global data)	More secure (encapsulation)
Reusability	Function libraries	Class inheritance
Best For	Small projects, system programming	Large projects, GUI applications
Example	Operating systems, utilities	Enterprise software, games

3. Program Development Process

a. **Stage-1: program design**
   i. Problem analysis
   ii. Outline the program structure
   iii. Algorithm development
   iv. Selection of control structures
b. **Stage-2: program coding**
c. **Stage-3: program testing and debugging**
   i. Types of errors
   ii. Program testing: unit and integration testing
   iii. Program debugging
d. **Stage-4: Program Efficiency (during algorithm development)**
   i. Execution time (time complexity)
   ii. Memory requirements (space complexity)

Stage-1: Program Design

1. Problem Analysis

Understand:

- What is input?
- What is output?
- Constraints?

Example

> Find the sum of N numbers
> Input: N numbers
> Output: Sum

2. Outline the Program Structure

Break problem into parts (functions)

Example:

- Input numbers
- Calculate sum
- Display result

3. Algorithm Development

Step-by-step solution

Algorithm: Sum of N numbers

1. Start
2. Read N
3. Initialize sum = 0
4. Repeat N times:
   - Read number
   - sum = sum + number
5. Print sum
6. End

4. Selection of Control Structures

- **Sequence**: step by step
- **Selection**: `if`, `switch`
- **Iteration**: `for`, `while`

Stage-2: Program Coding

Convert algorithm into code:

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

int main() {
    int n, sum = 0, x;

    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> x;
        sum += x;
    }

    cout << "Sum = " << sum;

    return 0;
}
```

Stage-3: Program Testing and Debugging

1. Types of Errors

(a) Syntax Errors

- Grammar mistakes

```c
printf("Hello")   // missing semicolon
```

(b) Runtime Errors

- Occur during execution

```c
int x = 5 / 0;    // divide by zero
```

(c) Logical Errors

- Wrong logic, wrong output

```c
avg = sum * n;    // should be sum / n
```

2. Program Testing

Unit Testing

- Test each function separately

Integration Testing

- Test combined modules

3. Program Debugging

Finding and fixing errors.

Methods:

- Print variable values
- Step-by-step execution (debugger)
- Remove errors one by one

Stage-4: Program Efficiency (during algorithm development)

- After designing an algorithm, we must check **how efficient it is**. Efficient programs:
  - Run **faster,** Use **less memory,** Scale well for **large inputs**
- Program efficiency mainly depends on:
  1. Execution Time (Time Complexity)
  2. Memory Requirements (Space Complexity)

What is Time Complexity?

Time complexity measures **how the running time of an algorithm grows** as the input size increases.

- It does **NOT** measure actual time in seconds.
- It measures **number of operations** performed.
- We use **Big-O notation** to represent time complexity.

What is Space Complexity?

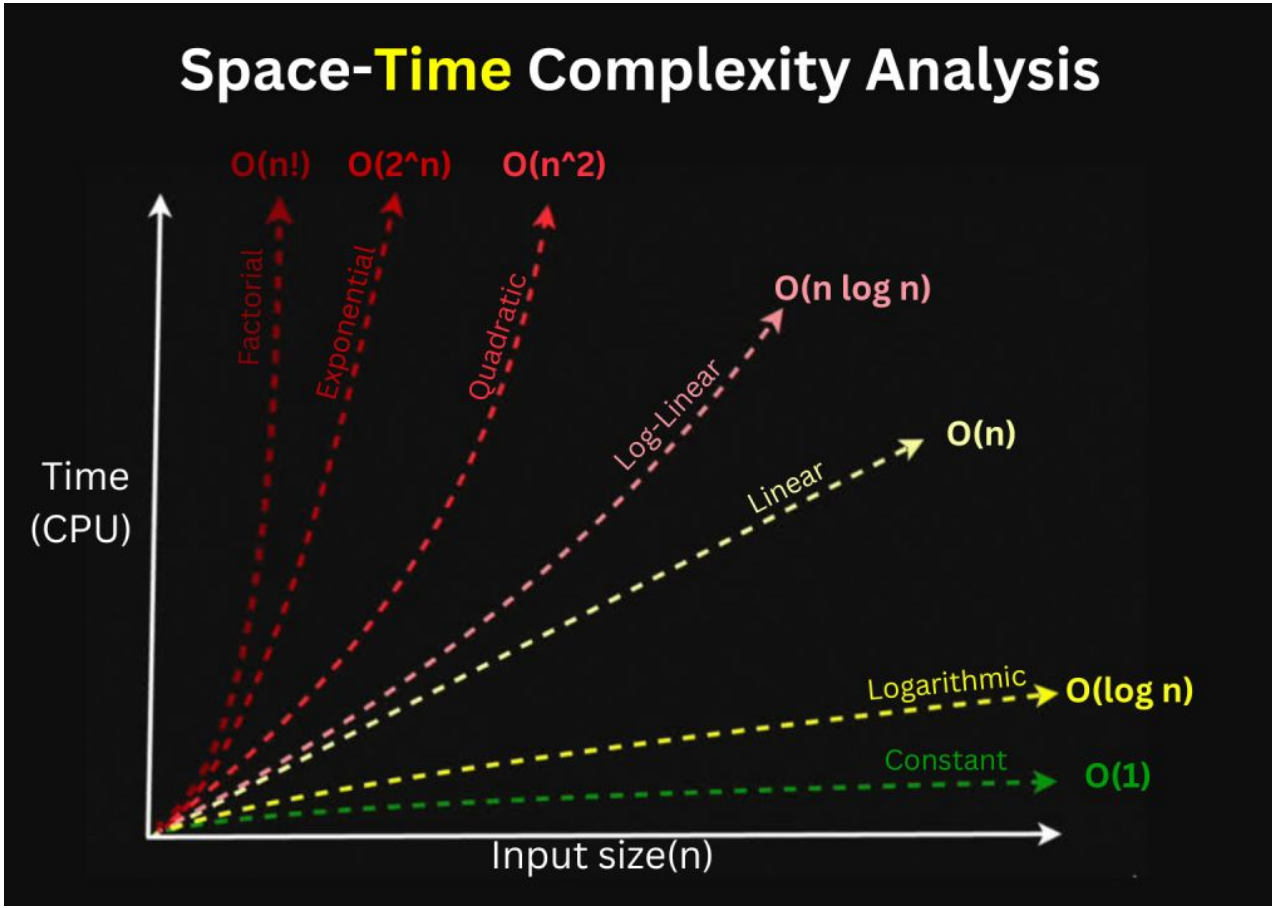
Space complexity measures the **amount of extra memory** used by an algorithm. Includes:

- Variables
- Arrays
- Dynamic memory
- Recursive stack

Stage-4: Program Efficiency (during algorithm development)

Complexity	Name	Operations for n=1000	Growth Rate
O(1)	Constant	1	Excellent
O(log n)	Logarithmic	~10	Excellent
O(n)	Linear	1,000	Good
O(n log n)	Linearithmic	~10,000	Fair
$O(n^2)$	Quadratic	1,000,000	Poor
$O(n^3)$	Cubic	1,000,000,000	Very Poor
O(2^n)	Exponential	1.07×10^{301}	Terrible
O(n!)	Factorial	4×10^{2567}	Catastrophic

Stage-4: Program Efficiency (during algorithm development)



Stage-4: Program Efficiency (during algorithm development)

❖ Time Limit Exceeded (TLE)

TLE (Time Limit Exceeded) means our program **did not finish within the allowed time** set by the system (online judge, compiler, or runtime environment). The logic may be correct, but the algorithm is **too slow**.

❖ Why TLE Happens:

1. Inefficient time complexity
   Using $O(n^2)$ or $O(2^n)$ when n is large.
2. Unnecessary nested loops
3. Repeated calculations
4. Slow input/output
5. Infinite or very long loops
6. Using recursion without optimization
7. Brute-force approach instead of optimized algorithms

Stage-4: Program Efficiency (during algorithm development)

- **Operations Per Second (Approximate):**
  - **Modern CPU:** 1 to 5 × 10^8 operations/second
  - **Contest/Online Judge:** Usually 10^7-10^8 operations/second
    - 1 second limit: ~10^8 operations
    - 2 second limit: ~2×10^8 operations
    - 5 second limit: ~5×10^8 operations
  - **TLE on Languages: C/C++ < Java < Python = 1 : 2 : 4**
  - **How to Avoid TLE:**
    - Analyze **time complexity before coding**
    - Avoid nested loops where possible
    - Use **hashing**, **binary search**, **two pointers**
    - Replace recursion with DP or memoization
    - Use **fast I/O** (scanf/printf in C/C++, buffered I/O)
    - Break early when condition is met
    - Precompute results if reused

4. History of C++ language

Origins (1979-1985)

- **Creator**: Bjarne Stroustrup at Bell Labs (AT&T)
- **Initial Motivation**: Enhance C for systems programming while adding **Simula-like object-oriented features**
- **Original Name**: "C with Classes" (1979)
- **Key Early Features**:
  - Classes, inheritance, inlining
  - Default arguments, stronger type checking
  - Initially a preprocessor (`cfront`) translating to C
- **First Name Change**: "C++" (1983) – the `++` operator signifying evolution beyond C

Formalization & Early Standardization (1985-1998)

- **First Commercial Release** (1985)
- **Reference Book**: *The C++ Programming Language* (1st edition, 1985) by Stroustrup
- **Key Developments**:
  - Virtual functions, operator overloading, references (C++ 2.0, 1989)
  - Templates, exception handling, namespaces (early 1990s)
  - Standard Template Library (STL) by Alexander Stepanov (1994)
- **ANSI/ISO Standardization** begins (1990)

Standard C++ (1998-2011)

- **ISO/IEC 14882:1998** – First International Standard ("C++98")
- Included the STL as part of the standard library
- **Minor Revision**: "C++03" (2003) – mainly bug fixes
- **TR1** (Technical Report 2006): Preview of C++11 features (smart pointers, regular expressions, etc.)

Modern C++ Renaissance (2011-Present)

- **C++11** ("C++0x"): Major transformation (2011)
  - Auto type, lambda expressions, move semantics (`&&`), `constexpr` , threading support
  - Widely seen as a new language
- **C++14** (2014): Refinement of C++11 features
  - Generic lambdas, variable templates, improved `constexpr`
- **C++17** (2017): Significant additions
  - Structured bindings, `std::variant` , `std::optional` , filesystem library, parallel algorithms
- **C++20** (2020): Major evolution
  - Concepts (constrained templates), modules, coroutines, ranges, three-way comparison (`<=>`)
- **C++23** (2023): Incremental improvements
  - `std::print` , stacktraces, more constexpr enhancements
- **C++26** (Upcoming): Under development
  - Expected features: pattern matching, reflection, contract support

5. Characteristics of C++ language

1. Emphasis is on data rather than procedure
2. Programs are divided into what are known as objects
3. Data structures are designed such that they characterize the objects
4. Functions are operate on the data of an object are tied together in the data structure.
5. Data is hidden and can not be accessed by external functions
6. Object may communicate with each other through functions
7. New data and functions can be easily added whenever necessary
8. Follows bottom-up approach in program design

5. Importance of C++ language

- Through inheritance we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to save development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program,
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques, for communication between objects makes the interface descriptions with external systems are much simpler.

6. WHERE TO USE C++

1. System Programming

Use C++ when you need **low-level control** and **high performance**.

- Operating systems
- Device drivers
- Compilers
- Database engines

Examples:

- Linux kernel parts
- Windows system components

2. Game Development

C++ is the **industry standard** for high-performance games.

- Game engines
- Real-time graphics
- Physics engines

Examples:

- Unreal Engine
- Unity (core engine)

3. Embedded Systems

Best when:

- Limited memory
- Real-time constraints
- Hardware interaction

Examples:

- Microcontrollers
- Automotive software
- Robotics

4. High-Performance Applications

C++ excels where **speed and efficiency** matter.

- Scientific simulations
- Image/video processing
- Financial trading systems

Examples:

- MATLAB core libraries
- Adobe Photoshop engine

5. Large-Scale Applications

Good for **long-term, complex projects**.

- Browsers
- Desktop applications

Examples:

- Google Chrome
- Microsoft Office components

6. Real-Time Systems

Use C++ where **timing is critical**.

- Flight control systems
- Medical devices
- Telecom systems

6. WHERE NOT TO USE C++

1. Simple Web Backends/REST APIs

- Better alternatives: Python (Django/Flask), JavaScript/TypeScript (Node.js), Go, Java
- Why: Slower development, more boilerplate, overkill for CRUD apps
- Exception: High-performance microservices needing extreme throughput

2. Rapid Prototyping & Scripting

- Better alternatives: Python, Ruby, JavaScript
- Why: C++ compilation slows iteration; lacks REPL environment
- Exception: Prototyping performance-critical algorithm kernels

3. Simple Desktop/Mobile GUIs

- Better alternatives:
  - Desktop: Electron, JavaFX, C# (WPF/WinForms)
  - Mobile: Swift (iOS), Kotlin (Android), Flutter/Dart
- Why: Steep learning curve for GUI frameworks; verbose UI code
- Exception: Professional creative tools (Adobe suite, DAWs) needing GPU access

4. Data Science & Machine Learning

- Better alternatives: Python (NumPy, Pandas, Scikit-learn, PyTorch)
- Why: Ecosystem is smaller; development too slow for experimentation
- Exception: Deploying trained models in production for inference optimization

5. Simple DevOps/Admin Scripts

- Better alternatives: Bash, Python, PowerShell
- Why: Compilation overhead; harder to write and maintain
- Exception: Performance-critical system tools (like `grep`, `sort` optimizations)

6. Quick Automation Tasks

- Better alternatives: Python, Perl, AutoHotkey
- Why: Not worth the development time for one-off tasks

END
of
Chapter 1

***Reference**:* E. Balaguruswamy; Object Oriented Programming with C++; *chapter-1, chapter-2, chapter-17;*