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 01

Revision of Basic Constructs

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

Outline

Topics Covered

01 Variables & Constants

02 Operators & Expressions

03 Conditional Statements

04 Loop Constructs

05 Arrays Concepts

04 Pointers & References

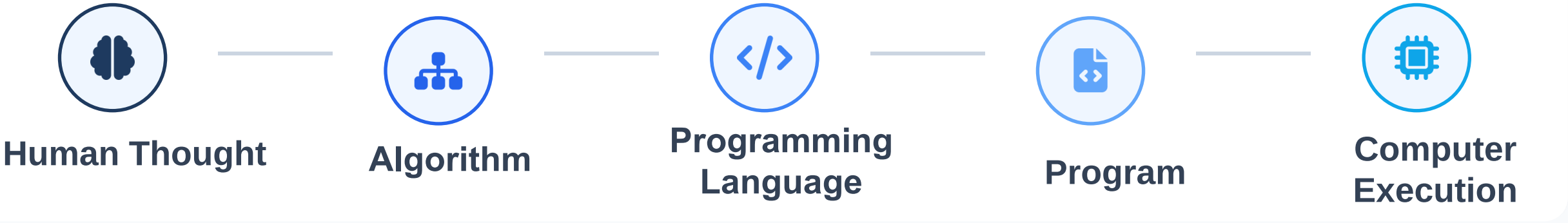
Learning Objectives

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

- Refresh your understanding of core programming constructs
- Identify common pitfalls in foundational programming and apply industry best practices
- Build a strong conceptual foundation for upcoming Programming II topics

1. What is a Program?

> “A **program** is a sequence of precise **instructions** written in a **programming language** that directs a computer to perform a specific task by transforming **input** data into meaningful **output** through automated processing” (*Webopedia, 2024).*



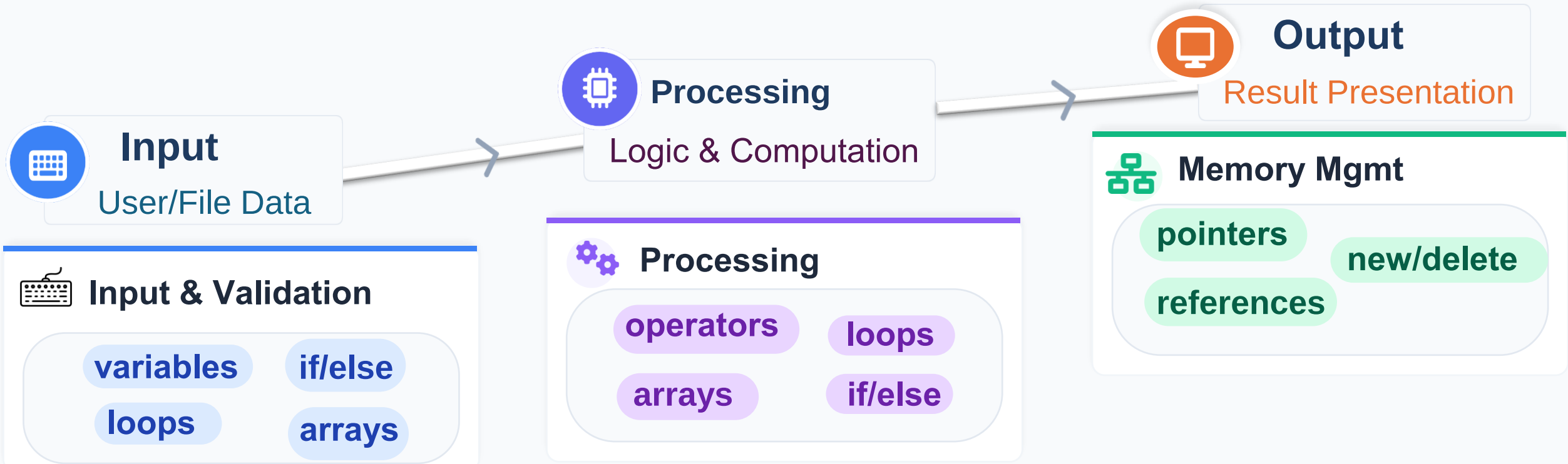
Algorithmic Thinking:

- Only **10%** is syntax in Programming, **90% logic**

2. Basic Structure of a Program

Input–Process–Output (IPO) Model

The IPO model explains how a program transforms **input** data into meaningful **output** through **processing** logic (Bjarne Stroustrup, 2013).

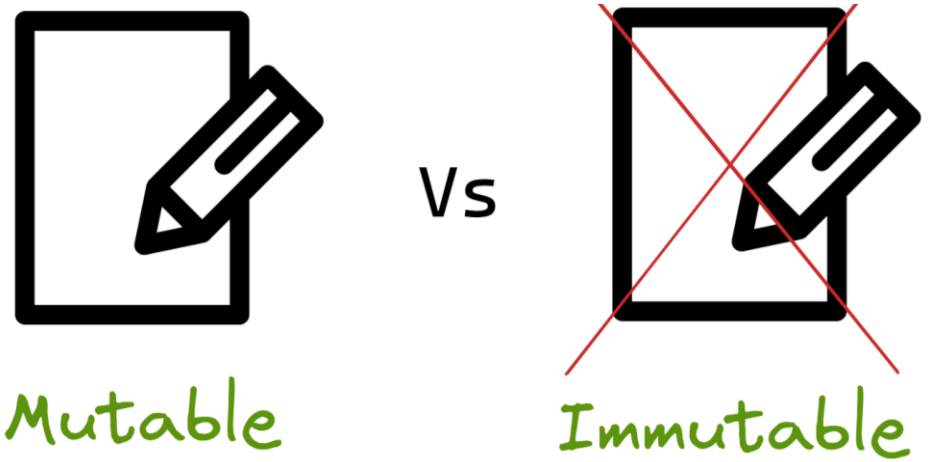


3. Variables & Constants

Core Concept

Named Storage Location

- **Variables and constants** act as human-readable **symbolic names** for specific memory addresses.
- Allow programmers to **store, retrieve, and manipulate data** without managing raw memory locations directly.



Variables

value can be change

Constants

value cannot be change

Cont'd

Characteristics

- **Identifiers:** Both variables and constants need names, often called identifiers
- **Naming Conventions:** Variables often use **camelCase**, while constants are frequently written fully in uppercase
- **Scope:** where variables and constants are accessible (local or global).
- **Lifetime:** How long the variable exists in memory
- **Data Types:** determines what kind of data stores, the **size and layout**

Declaration Syntax

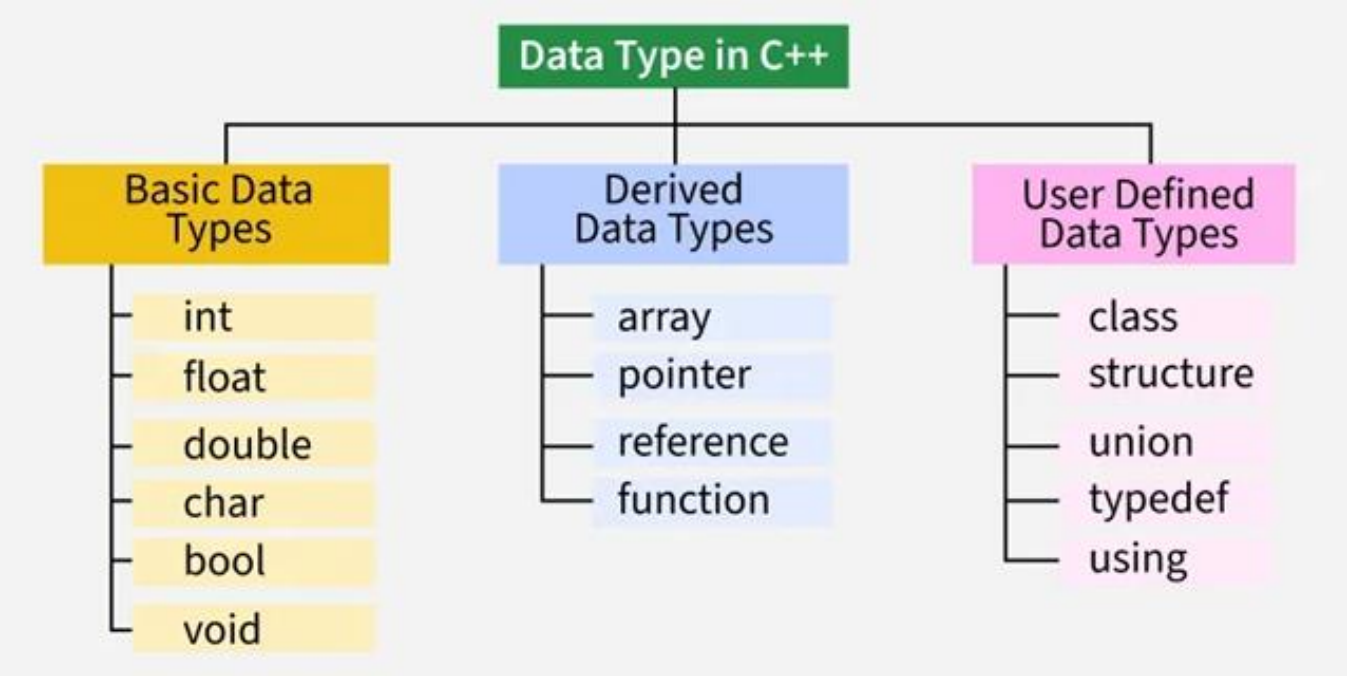
```
type variable_name;
or
type variable_name = value;
or
type variable_name {value};
```

```
const type const_name = value;
or
#define const_name value;
```

Cont'd

Data Types

- Describes the **property** of the data and the **size** of the reserved memory.
- Established when the variable is defined



Signed vs Unsigned

- **Unsigned**
  - a **positive integer** only
- **Signed**
  - it is either negative or positive

Source: *https://www.geeksforgeeks.org/cpp/cpp-data-types/*

Cont'd

Primitive Data Types

- Every variable has a **fixed data type** determined at compile time.
- The **sizes** may vary on the compiler and **architecture of the computer system**, but relative sizes are guaranteed.

Table 1: Primitive data types built into the C++ language.

TYPE	TYPICAL SIZE	EXAMPLE DECLARATION
int	4 bytes (32-bit)	**int**count = 0;
double	8 bytes (64-bit)	**double**pi = 3.14159;
char	1 byte	**char**grade = 'A';
bool	1 byte	**bool**isValid = true;
float	4 bytes	**float**ratio = 2.5f;
void	0 bytes	Used for functions in no return

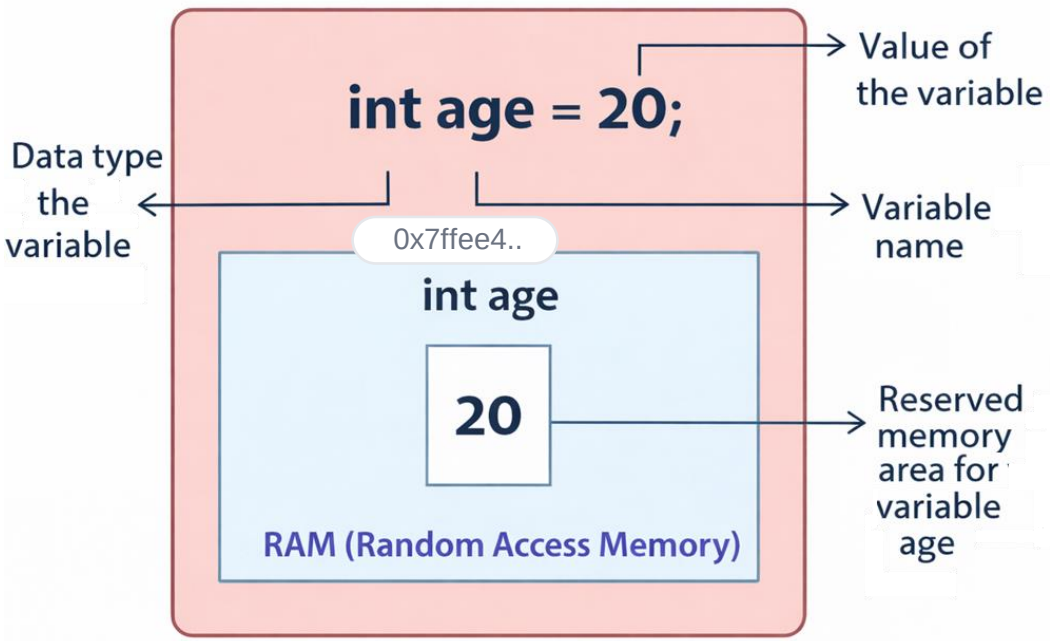
Note: Use sizeof() to determine the exact size on your system

Cont'd

Example of Declaration and Initialization

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

int main() {

  // 1. Basic Initialization (C-style)
  int count = 10;
  char grade = 'A';

  // 2. Constructor Style (C++)
  int score(85);

  // 3. Multiple Declaration
  int width = 5, height = 10;

  // 4. Constants & Inference (Modern C++)
  const double PI = 3.14159;
  auto isReady = true; // Deducts bool

  return 0;
}
```

Memory Abstraction

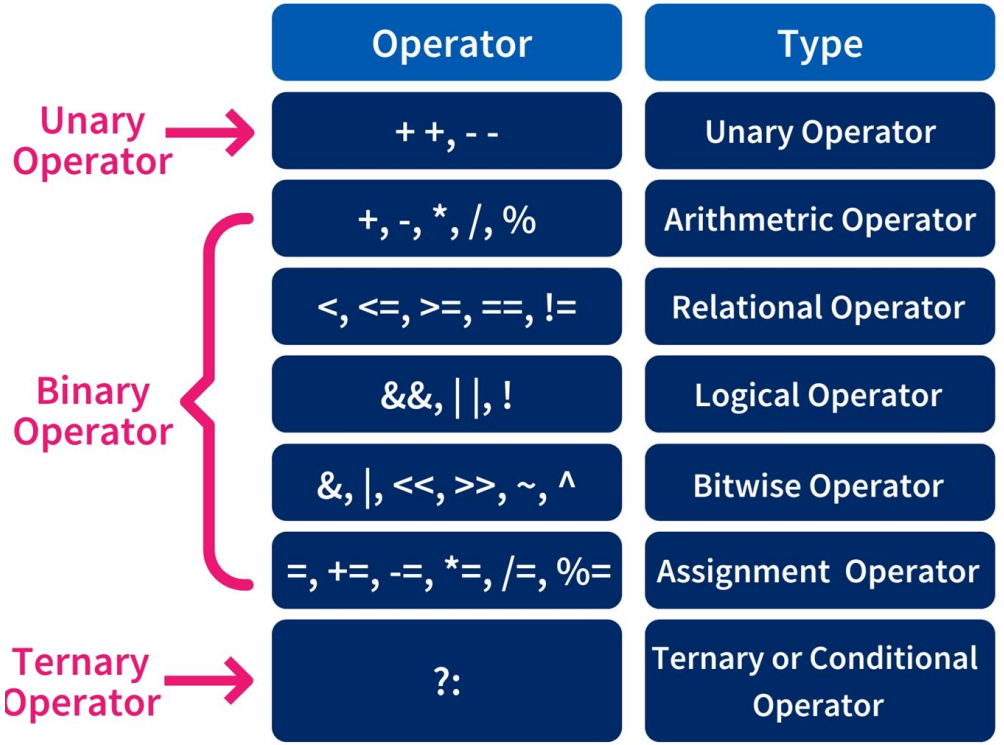


***Source**: Generate by ChatGPT 4.0*

4. Operators and Expressions

Core Concept

- **Operators** are **special symbols** or **keywords** that instruct a computer what operations to perform (**mathematical / logical / data manipulation**).
- Fundamental building blocks for expressions and logic in programming.
- An **expression** is a combination of *operands (variables, constants, or values) and operators* that is evaluated to produce a single value.



Source: https://www.scaler.com/topics/cpp/operators-in-cpp/

Cont'd

Operator Precedence & Associativity

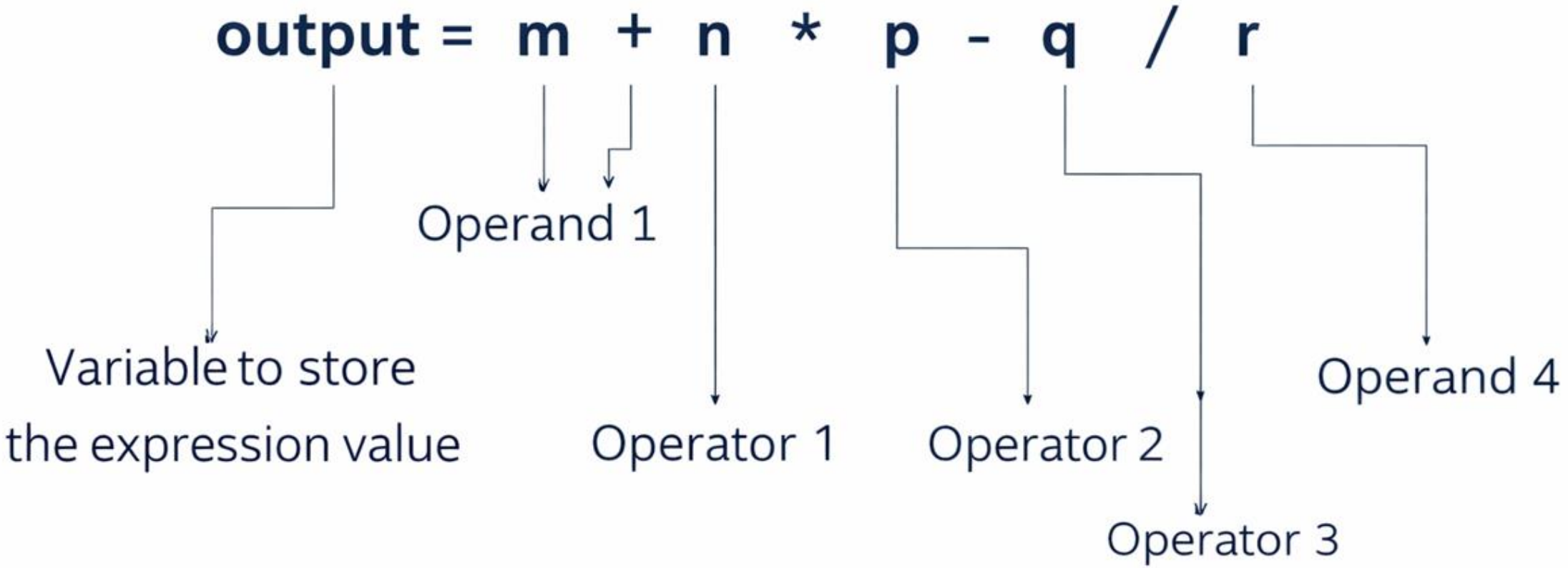
- ✓ Precedence refers to the order in which **operators** are evaluated.
- ✓ When operators have the same precedence, **associativity** rules apply

Sample Precedence Table (High to Low)

LEVEL	OPERATORS	ASSOCIATIVITY
1 (Highest)	::	Left to Right
2	++ -- () [] . ->	Left to Right
3	++ -- + - ! ~ * & (unary)	Right to Left
5	* / %	Left to Right
6	+ -	Left to Right
7	&&	Left to Right
8	||	Left to Right
9 (Lowest)	= += -= *= /=	Right to Left

Cont'd

Example of an Expression



Cont’d

Challenge Exercise:

Expression Evaluation

Instructions

- ✓ Calculate the result of each expression using operator precedence rules.
- ✓ Don't just guess—break it down step-by-step!

Tip: For equal precedence, check associativity!
Most are Left-to-Right.

1: Mixed Arithmetic — Easy

```
10 + 20 * 3 - 5
```

→ ?

2: Integer Division & Modulo — Medium

```
100 + 10 % 3
```

→ ?

3: Precedence Trap — Tricky

```
2 + 3 << 1
```

→ ?

4; Assignment Side Effects — Hard

```
// a,b=0 initially
(a=5) +(b=2) * 4
```

Result:?
Final a, b:?

Cont'd

Solutions

1: Mixed Arithmetic — Easy

`10 + 20 * 3 - 5` → `10 + (20 * 3) - 5` → `10 + 60 - 5` → `70- 5` → **65**

2: Integer Division & Modulo — Medium

`100 + 10 % 3` → `100 + (10 % 3)` → `100 + 1` → **101**

3: Precedence Trap — Tricky

`2 + 3 << 1` → `(2 + 3) << 1` → `5 << 1` → **10**

4; Assignment Side Effects — Hard

`(a=5) + (b=2) * 4` → `5 + 2 * 4` → `5 + (2 * 4)` → `5 + 8` → **Result: 13**
Final a=5, b=2

5. Flow Controls

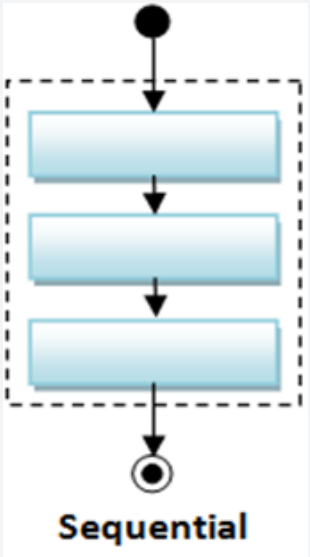
What is Flow Control?

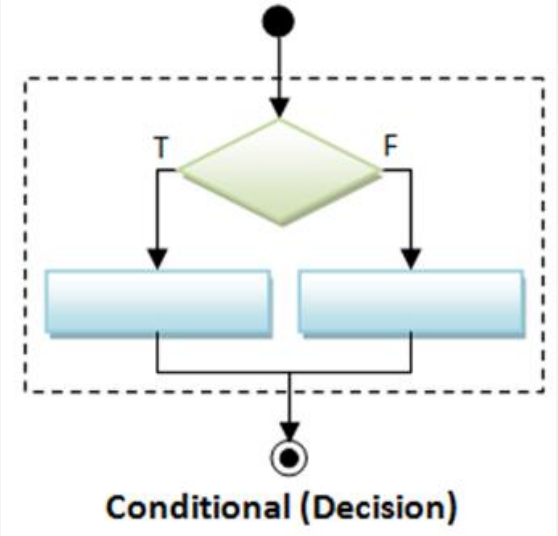
It refers to the **order** in which **statements or instructions** in a program are executed.

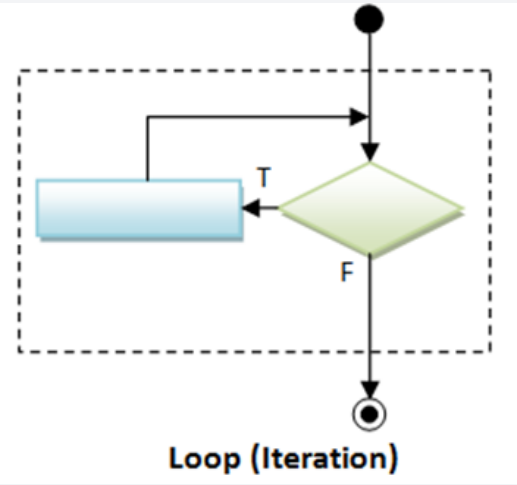
It determines **how** the program **moves from one instruction to another** during execution.

By default, a program executes **sequentially**, but flow control structures allow the program to:

- **Make decisions** (selection)
- **Repeat actions** (iteration)
- **Change the execution path**



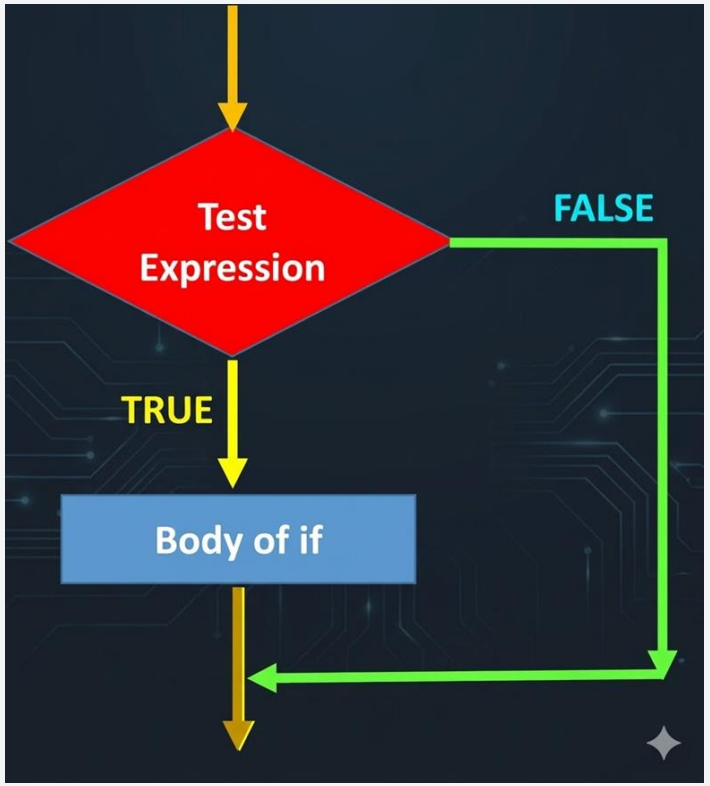




5.1. Selection Statements

Overview

- Allow programs to **make decisions** and handle different scenarios dynamically
- Allow programs to **execute different code blocks** based on boolean expressions.
- This **breaks** the linear execution of code.



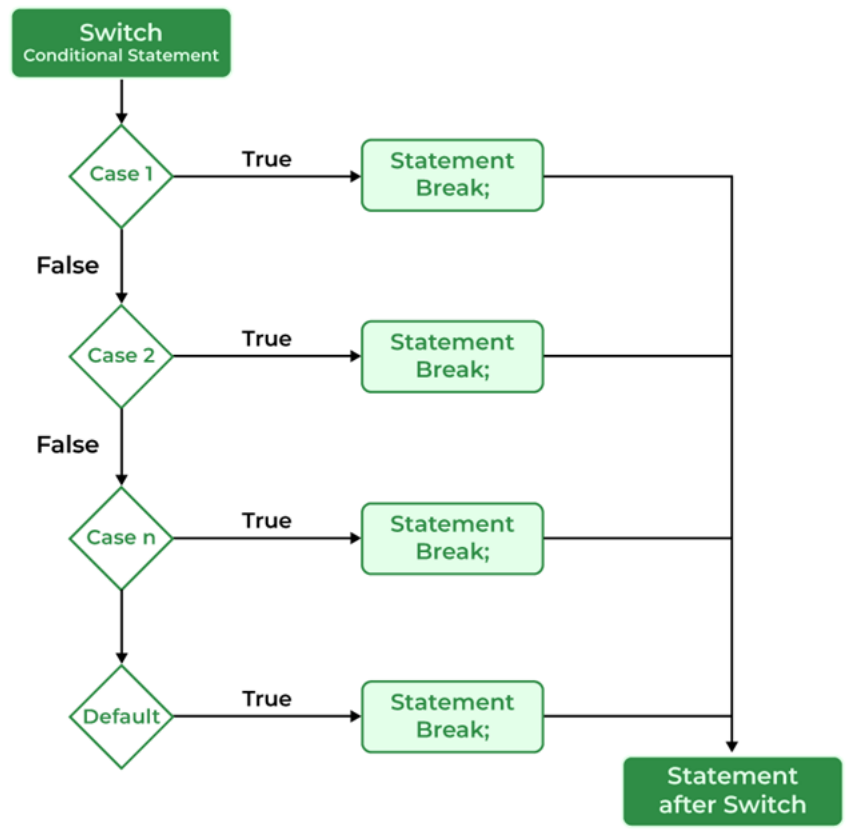
***Source**: Generate by Gimini 3*

Cont'd

Common Patterns

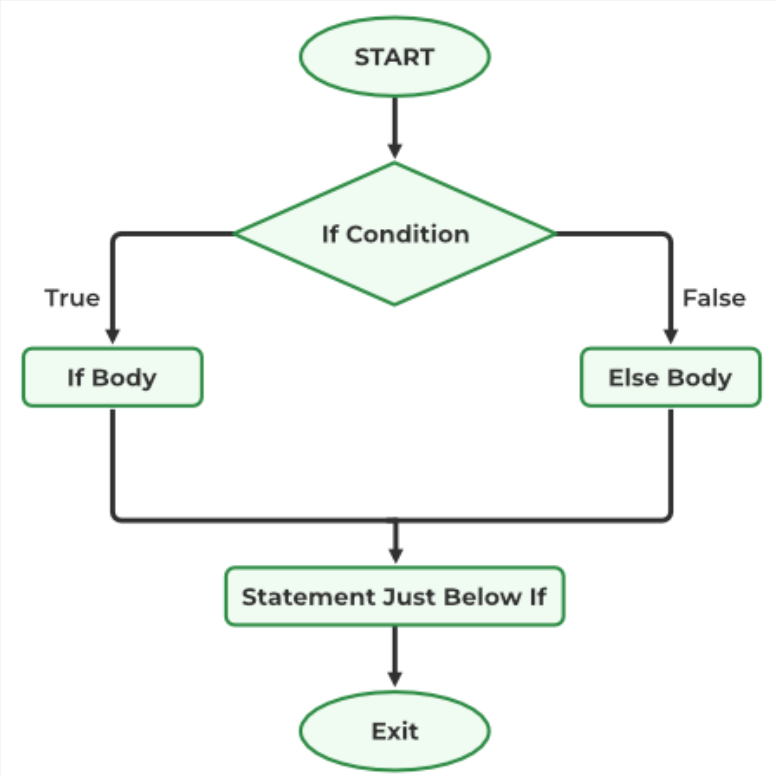
1. switch:-

- selecting one path from multiple constant



2. if / else:

- basic **binary** decisions and chaining conditions



***Source**:*
https://www.geeksforgeeks.org/ cpp

Cont’d

Example 1: if – else if ladder

Execution Flow Analysis

Check 1. score >= 90 (78 >= 90) → **false**

Check 2:score >= 80 (78 >= 80) → **false**

Check 3:score >= 70 (78 >= 70) → **true**

↳ Executes body: prints "Grade: C“

Remaining else block is skipped.

grading.cpp

```cpp
#include <iostream>

int main() {
    int score{78};

    // If-else-if ladder example
    if (score >= 90) {
        std::cout << "Grade: A\n";
    }
    else if (score >= 80) {
        std::cout << "Grade: B\n";
    }
    else if (score >= 70) {
        std::cout << "Grade: C\n";
    }
    else {
        std::cout << "Grade: D/F\n";
    }

    return 0;
}
```

>_ PROGRAM OUTPUT

```
Grade: C_
```

Cont'd

Example 2: switch case

Assume use input:

- num1 = 212, num2 = 43
- Operator (oper): /

Output?

- 4.93

```
int main() {
    char oper;
    float num1, num2;
    cout << "Enter an operator (+, -, *, /): ";
    cin >> oper;
    cout << "Enter two numbers: " << endl;
    cin >> num1 >> num2;

    switch (oper) {
        case '+':
            cout <<"Addition Result: "<< num1 + num2;
            break;
        case '-':
            cout <<"Subtraction Result: "<< num1 - num2;
            break;
        case '*':
            cout <<"Multiplication Result: "<< num1 * num2;
            break;
        case '/':
            cout <<"Division Result: "<< num1 / num2;
            break;
        default:
            cout << "Incorrec operator!";
            break;
    }
    return 0;
}
```

5.2. Loops

Overview

Loops enable repeated execution of code blocks until a specific condition is met.

- **Repetition:**
  - *Allow execution of a block of code multiple times.*
- **Efficiency:**
  - *Avoid code duplication (DRY principle).*

Key Components

1. **Initialization:** Setup loop variable.
2. **Update:** Change loop variable.
3. **Body:** Code to execute

Cont’d

Loop Types & Usage

1) while loop:

- ✓ Repeat until condition becomes false (pre-check)

```
while (condition) { ... }
```

2) for loop:

- ✓ Counted iteration (known number of times)
- ✓ Iterating over arrays/vectors (index-based)

```
for (int i=0; i<10; ++i) { ... }
```

3) do-while loop:

- ✓ Execute at least once, then check condition (post-check)

```
do { ... } while (condition);
```

4) Range-based for (C++11):

- ✓ Cleanest syntax for iterating over containers

Cont'd

Example:

- for loop
- while loop

```cpp
#include <iostream>
int main() {
    int n;
    cout << "Enter a number (N): ";
    cin >> n;

    cout << "\nFor Loop (1 to " << n << "): ";
    for (int i = 1; i <= n; ++i) {
        cout << i << " ";
    }

    cout << "\nWhile Loop (" << n << " down to 1): ";
    int temp = n; // Initialize counter
    while (temp >= 1) {
        std::cout << temp << " ";
        temp--; // Decrement to break the condition
    }

    cout << "\n";
    return 0;
}
```

Output

```
Enter a number (N): 5

For Loop (1 to 5): 1 2 3 4 5

While Loop (5 down to 1): 5 4 3 2 1
```

6. Arrays - Concept and Declaration

What is an Array?

A fixed-size collection of elements of the **same data type**, stored in **contiguous memory locations**.

- **Homogeneous Data:** *All elements must be of the same type*
- **Fixed Size:** *Size must be known at compile-time.*
- **Zero-Based Indexing:** *Access elements using indices from 0 to N-1.*
- **Contiguous Memory:** Elements are stored side-by-side in RAM.

Cont'd

Declaration Syntax

General Form:

```
// 1D array
type arrayName[arraySize];

//2D array
type arrayName[rowSize][colSize];
```

Example 1: int data[5] = {10, 20, 14}

Memory Visualization

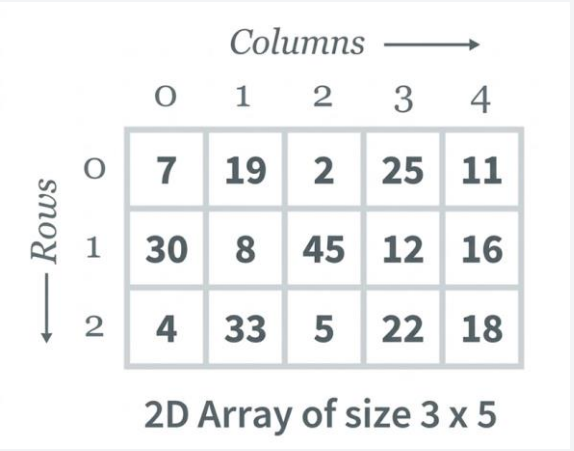
data

10	20	14	0	0
0	1	2	3	4

Since the size of int is 4 bytes, each array element takes 4 bytes

Example 2:

```
int matrix[3][5]  =
        {{7, 19, 2, 25, 11},
        {30, 8, 45, 12, 16},
        {4, 33, 5, 22, 18}}
```



2D Array of size 3 x 5

Nested braces make initialization readable and structured

Cont’d

Array Initialization:

```
// --- 1D ARRAY INITIALIZATION ---

// 1. Full Initialization
int nums[5] = {10, 20, 30, 40, 50};

// 2. Partial Initialization
// (Remaining elements become 0)
int partial[5] = {1, 2};

// 3. Universal Zero Initialization
int zeros[10] = {}; // Cleaner than {0}

// 4. Size Inference
// Compiler sets size to 3
double data[] = {1.5, 2.5, 3.5};
```

```
// --- 2D ARRAY INITIALIZATION ---

// 5. Explicit Row/Column Initialization
// Format: array[rows][columns]
int matrix[2][3] = {
    {1, 2, 3}, // Row 0
    {4, 5, 6}  // Row 1
};

// 6. 2D Size Inference
// (Only the first dimension can be omitted)
int grid[][2] = {
    {10, 20},
    {30, 40},
    {50, 60}
}; // Compiler infers 3 rows
```

Cont'd

Accessing Arrays Elements

```
// 1D Array
#include <iostream>
using namespace std;
int main() {
    int scores[5] = {85, 92, 78, 90, 88};

    // Assume the first element is the largest
    int maxVal = scores[0];

    for (int i = 1; i < 5; i++) {
        if (scores[i] > maxVal) {
          // Update maxVal if a larger one is found
            maxVal = scores[i];
        }
    }

    cout << "The highest score is: " << maxVal << endl;
    return 0;
}
```

```
The highest score is: 92
```

```
// 2D Array
#include <iostream>
using namespace std;
int main() {
int matrix[2][3] = {{1, 2, 3},  {4, 5, 6}};
int totalSum = 0;

for (int i = 0; i < 2; i++) {  // contorl row
    for (int j = 0; j < 3; j++) { // control column
        // Accessing [row][column]
        totalSum += matrix[i][j];
    }
}
cout << "Total: "<<totalSum<<endl;
}
```

```
Total: 21
```

Cont'd

Exercise: Finding the resulting array after the below code executed.

```
#include <iostream>

int main() {
    double prices[5] = {45.0, 120.0, 30.0, 65.5, 10.0};
    int size = 5;

    for (int i = 0; i < size; i++) {
        if (prices[i] > 50.0) {
            prices[i] *= 0.90;
        }
    }
    return 0;
}
```

Resulting Array:

{45.0, 108.0, 30.0, 58.95, 10.0}

7. Pointers - Fundamental Concept

What is a Pointer?

A variable that stores the **memory address** of another object, rather than holding a value directly.

```
// Declaration
int* ptr;

// Initialization
ptr = &variable;
```

Memory Visualization


```
int num=42;
int*ptr=&num;

// num is variable at memory location 0x104
// ptr stores address of num (0x104)
// *ptr access value at 0x104 (which is 42)
```

Cont'd

Key Operators

& **Address-of Operator**

"Get the address of this variable"

&x -> 0x7ffd...

***** **Dereference Operator**

- **Dereferencing** - *access the value stored at the memory address held by the pointer*
- ***Indirect modification*** - *directly modifies the variable which the pointers points to*

dereference.cpp

```cpp
#include <iostream>
int main() {
    int score = 100;
    int* pScore = &score;

    // 1. Read value via pointer
    std::cout << "Original: " << *pScore << "\n";

    // 2. Modify value via pointer
    *pScore = 200; ✓ Changes 'score'

    std::cout << "New Score: " << score << "\n";

    // 3. Double Pointer
    int** ppScore = &pScore;
    **ppScore = 500;
```

TERMINAL OUTPUT

```
Original: 100
New Score: 200
Final Score: 500
```

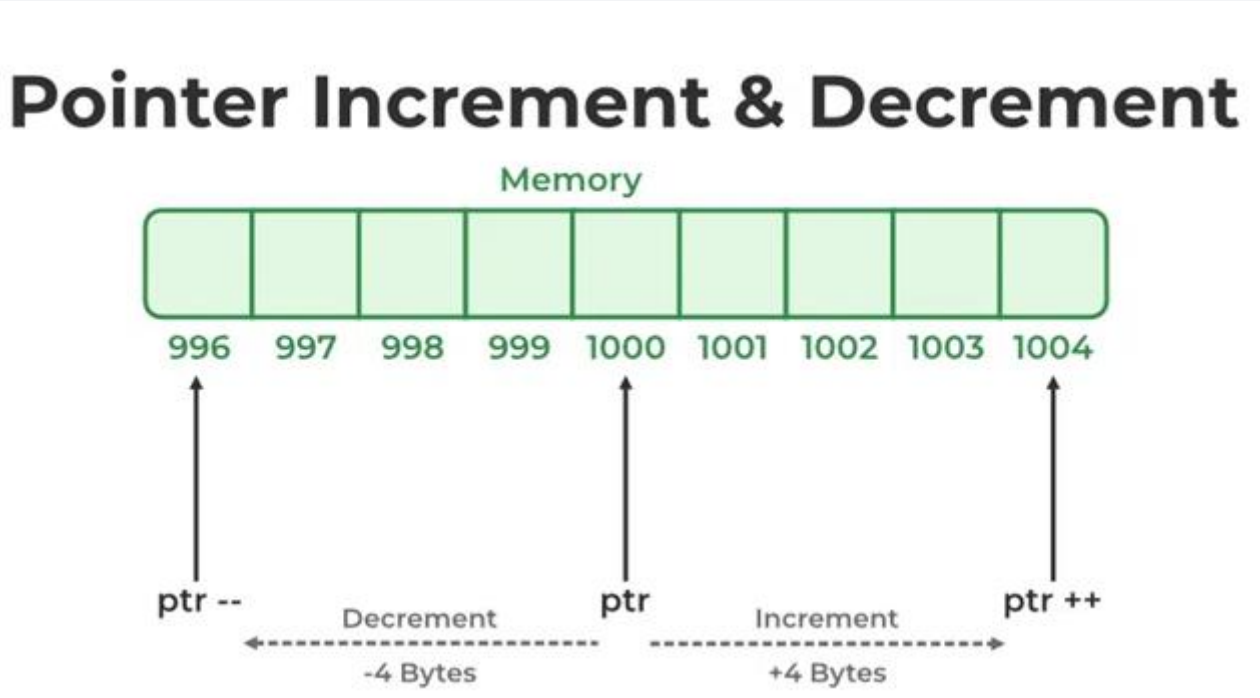
7.1 Pointer Arithmetic

Incrementing (++Ptr / ptr++)

- Incrementing a pointer moves it to the next memory address based on the data type size.
- For **int***, adds **4 bytes** to move to the next integer memory location.

Decrementing (--ptr / ptr--)

- Decreases address by data type size., moves it to the previous memory address.
- Useful for traversing arrays in reverse order.



Source: https://www.geeksforgeeks.org/cpp/cpp-pointer-arithmetic/

8. Common Programming Pitfalls

Initialization & Memory Issues

- ✕ Uninitialized arrays
- ✕ Uninitialized variables
- ✕ Missing null terminator a

Logic & Expression Errors

- ✕ Assignment (=) vs Equality (==)
- ✕ Incorrect use of modulo (%)
- ✕ Type mismatches

Control Flow Mistakes

- ✕ Infinite loops
- ✕ Missing break in switch
- ✕ Off-by-one errors
- ✕ Empty statements

Design Issues

- ✕ Accidental variable shadowing
- ✕ Overuse of global variables
- ✕ Modifying loop body incorrectly

9. Programming Best Practices

📦 Variables & Data

- ✓ Initialize variables at declaration
- ✓ Use meaningful, descriptive names
- ✓ Prefer **const** for immutable values
- ✓ Declare variables at point of first use

👓 Code Clarity

- ✓ Keep expressions simple & readable
- ✓ Use parentheses for explicit precedence
- ✓ Keep loop bodies small & focused
- ✓ Avoid deep nesting (early returns)

🔀 Control Structures

- ✓ Always use braces **{}** for blocks
- ✓ Always add **break** in switch cases
- ✓ Choose the right loop construct
- ✓ Handle all switch cases (use default)

🛡️ Memory & Safety

- ✓ Use proper initialization for arrays
- ✓ Validate indices to avoid out-of-bounds
- ✓ Check pointers for nullptr before use
- ✓ Match new/delete correctly

Summary/Takeaway

Program Logic

Storage

Variables

Data Types & Scope

Defines **what** data exists and where it lives (scope).

Computation

Operators

Arithmetic & Logic

Transforms data states through calculation and **comparison**.

Decision

Conditionals

Control Flow

*Determines **which path** execution takes based on data state.*

Iteration

Loops

Repetition

Executes tasks **repeatedly** to process collections or await conditions.

Arrays

Data Collections

Groups related data elements linearly for efficient **indexed access**.

Memory

Pointers

Direct Access

*Accesses memory **directly**, enabling dynamic structures and efficiency.*

Practice Questions

Predict Output

1. What is the output of this expression? (Apply operator precedence)

```
int a{2}, b{3}, c{4};
cout << a + b * c;
```

Answer: 14

2. Pointer Arithmetic Challenge:

   What does this print?

```
int arr[] {10, 20, 30};
int* p = arr;
cout << *(p + 2);
```

Answer: 30

3. Spot the bug! Why is this unsafe?

```
int* p = new int(5);
delete p;
*p = 10;
```

Answer: Why This is Unsafe?

- Deleting an uninitialized pointer, result in undefined behavior
- Memory management must follow strict order: **Allocate, Use, Deallocate**

References

Text 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

- **Object-Oriented Programming in C++ (4th Edition)** by Robert Lafore, Sams Publishing, 2022.
- **Principles of Object-Oriented Programming**, by Stephen W. & Dung N., OpenStax CNX, 2022.

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, modify and compile 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