LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

UNIT 2

Basic Computer Engineering (BT-205)

Algorithm

- Algorithm is sequence of activities to be processed for getting desired output from a given input.
- An algorithm is a set of steps to solve a problem.
- It takes input and gives the desired output.
- The steps must be clear and easy to follow.
- Every algorithm should have a definite end.
- There can be many algorithms to solve the same problem.
- An algorithm must always produce the intended result.
- It must finish its job in a finite amount of time.
- Every step must be clear and have only one meaning.

Properties of algorithm

- **Finiteness**: An algorithm must stop after a limited number of steps.
- **Definiteness**:Every single step must be clear and have only one meaning.
- **Well-Defined Input & Output:** An algorithm must have well-defined inputs to work with and produce specified output.
- **Effectiveness:** Algorithms to be developed/written using basic operations. Every step must be simple enough to be done with pencil and paper.
- **Clarity**: Language Independent, Clear & Unambiguous. An algorithm must be clear and have only one meaning, regardless of various languages.

Check if a given number is even or odd.

- **Algorithm Steps (in Pseudocode):**

1) START
2) READ the number n
3) CALCULATE the remainder when n is divided by 2 (n % 2)
4) IF the remainder is equal to 0:
5) THEN PRINT "The number is EVEN"
6) ELSE:
7) PRINT "The number is ODD"
8) STOP

FLOWCHART

- The flowchart is a diagram which visually presents the flow of data through processing systems. This means by seeing a flow chart one can know the operations performed and the sequence of these operations in a system. Algorithms are nothing but sequence of steps for solving problems. So a flow chart can be used for representing an algorithm.

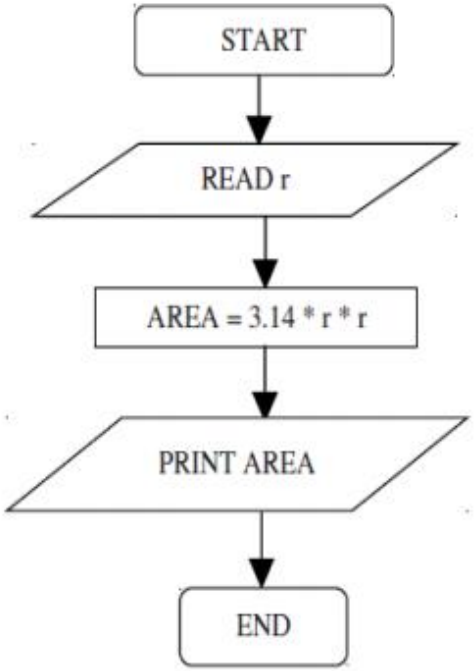
- For example suppose you are going for a picnic with your friends then you plan for the activities you will do there. If you have a plan of activities then you know clearly when you will do what activity. Similarly when you have a problem to solve using computer or in other word you need to write a computer program for a problem then it will be good to draw a flowchart prior to writing a computer program. Flowchart is drawn according to defined rules.

FLOWCHART

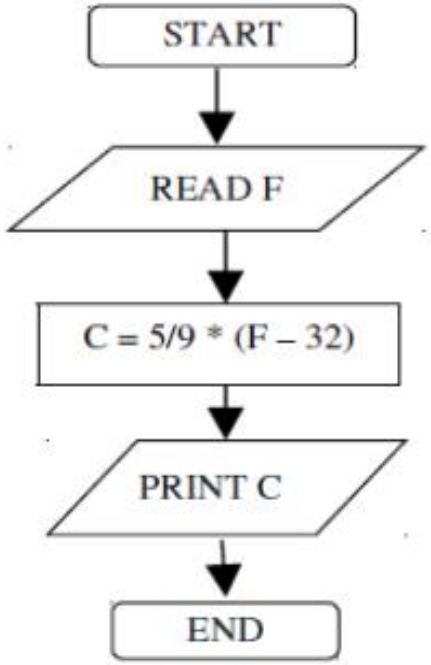
Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Example of Flowchart

Problem1: Find the area of a circle of radius r.



Problem 2: Convert temperature Fahrenheit to Celsius.



General Rules for flowcharting

- All boxes of the flowchart are connected with Arrows. (Not lines)
- Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
- The Decision symbol has two exit points; these can be on the sides or the bottom and one side.
- Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
- Connectors are used to connect breaks in the flowchart.
- Examples are: (i) From one page to another page.
- (ii) From the bottom of the page to the top of the same page.
- (iii) An upward flow of more then 3 symbols

General Rules for flowcharting

- Subroutines and Interrupt programs have their own and independent flowcharts.
- All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
- All flowcharts end with a terminal or a contentious loop.
- Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed.

Complexity

- Algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n.
- If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n
- While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of space, which translates to the algorithm's memory requirements.
- Algorithms are of two types
- 1. Space Complexity 2. Time Complexity

Complexity

1. **Space Complexity**: It is the amount of memory which is needed by the algorithm (program) to run to completion. We can measure the space by finding out that how much memory will be consumed by the instructions and by the variables used.

Suppose we want to add two integer numbers. To solve this problem we have following two algorithms:

Algorithm 1:	Algorithm 2:
Step 1- Input A.	Step 1- Input A.
Step 2- Input B.	Step 2- Input B.
Step 3- Set C: = A+ B.	Step 3- Write: 'Sum is ', A+B.
Step 4- Write: 'Sum is ', C.	Step 4- Exit.
Step 5- Exit.	

Complexity

Both algorithms will produce the same result. But first takes 6 bytes and second takes 4 bytes (2 bytes for each integer). And first has more instructions than the second one. So we will choose the second one as it takes less space than the first one.

Time Complexity

- It is the amount of time which is needed by the algorithm (program) to run to completion.
- We can measure the time by finding out the compilation time and run time. The compilation time is the time which is taken by the compiler to compile the program.
- This time is not under the control of programmer. It depends on the compiler and differs from compiler to compiler.
- One compiler can take less time than other compiler to compile the same program. So we ignore the compilation time and only consider the run time.
- The run time is the time which is taken to execute the program. We can measure the run time on the basis of number of instructions in the algorithm.

Time Complexity

E.g.

Suppose we want to add two integer numbers. To solve this problem we have following two algorithms:

Algorithm 1:

Step 1- Input A.

Step 2- Input B.

Step 3- Set C: = A+ B.

Step 4- Write: 'Sum is ', C.

Step 5- Exit.

Algorithm 2:

Step 1- Input A.

Step 2- Input B.

Step 3- Write: 'Sum is ', A+B.

Step 4- Exit.

Suppose 1 second is required to execute one instruction. So the first algorithm will take 4 seconds and the second algorithm will take 3 seconds for execution. So we will choose the second one as it will take less time.

Program

- A program is a set of instructions given to a computer to perform a specific operation.
- While executing the program, raw data is processed into a desired output format. These computer programs are written in a programming language which are high level languages.
- The computer only understands binary language (the language of 0's and 1's) also called machine-understandable language or low-level language but the programs we are going to write are in a high-level language which is almost similar to human language.
- Like we have different languages to communicate with each other, likewise, we have different languages like C, C++, C#, Java, python, etc to communicate with the computers.

Programing Languages

- **Types of Programming Languages**

There are two **types of programming languages**, which can be categorized into the following ways:

1. Low level language

a) Machine language (1GL)

b) Assembly language (2GL)

2. High level language

a) Procedural-Oriented language (3GL)

b) Problem-Oriented language (4GL)

c) Natural language (5GL)

Programing Languages

- **Low level language**: This language is the most understandable language used by computer to perform its operations. It can be further categorized into:
- **Machine Language (1GL)**: Machine language consists of strings of binary numbers (i.e. 0s and 1s) and it is the only one language, the processor directly understands. Machine language has an Merits of very fast execution speed and efficient use of primary memory.
- **Merits:** It is directly understood by the processor so has faster execution time since the programs written in this language need not to be translated. It doesn't need larger memory.
- **Demerits:** It is very difficult to program using 1GL since all the instructions are to be represented by 0s and 1s.
- Use of this language makes programming time consuming.
- It is difficult to find error and to debug.
- It can be used by experts only.

Programing Languages

- **Assembly Language:** Assembly language is also known as low-level language because to design a program programmer requires detailed knowledge of hardware specification. This language uses mnemonics code (symbolic operation code like 'ADD' for addition) in place of 0s and 1s. The program is converted into machine code by assembler. The resulting program is reffered to as an object code.
- **Merits:** It is makes programming easier than 1GL since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes programming process faster.
- Error can be identified much easily compared to 1GL.
- It is easier to debug than machine language.

Programing Languages

- **Demerits:** Programs written in this language is not directly understandable by computer so translators should be used.
- It is hardware dependent language so programmers are forced to think in terms of computer's architecture rather than to the problem being solved.
- Being machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

High level language

- A high-level language is a programming language that is designed to be easily understood and written by humans.
- Its syntax and structure are closer to human language (like English) and mathematical notation, rather than the machine language (binary 1s and 0s) that a computer's CPU understands directly.
- **Key Characteristics:**
- **Readability:** The code is in-built and uses familiar words (e.g., if, while, print).
- **Abstraction:** It abstracts away the complex, low-level details of the computer's hardware, such as memory management and register allocation. You don't need to think about where in memory a variable is stored.

High level language

- **Portability:** Code written in a high-level language can, in theory, run on different types of computers with minimal changes. This is achieved by using compilers or interpreters specific to each machine.
- **Ease of Use:** It requires less code to perform complex operations, significantly increasing developer productivity.
- **Translation Process:** Since a CPU can only execute machine code, code written in a high-level language must be translated. This is done by one of two programs:
- **Compiler:** Translates the entire source code into machine code (an executable file) before the program is run (e.g., C++, Rust).
- **Interpreter:** Translates and executes the source code line-by-line at runtime (e.g., Python, JavaScript). (Note: Modern languages often use a mix, like compiling to bytecode first).

High level language Types

- High-level languages are often categorized by their **programming paradigm**—the fundamental style and approach to structuring the code and solving problems. Many modern languages support multiple paradigms.
- **Procedural Programming:** This paradigm organizes code into procedures (also known as functions or subroutines). The program follows a sequential step-by-step list of instructions, calling procedures as needed.
- Focus: "What are the steps to solve the problem?"
- Key Concepts: Functions, procedures, sequential execution.
- Examples: C, Pascal, Fortran, Go.
- Analogy: A recipe for baking a cake. It's a list of instructions to follow in order.

High level language Types

- **Object-Oriented Programming (OOP)**
- This paradigm organizes code around "objects," which are instances of "classes." These objects contain data (attributes) and methods (functions) that operate on the data. It's excellent for modeling real-world systems.
- Focus: "What are the objects involved and how do they interact?"
- Key Concepts: Classes, Objects, Inheritance, Encapsulation, Polymorphism.
- Examples: Java, C++, Python, C#, Ruby.
- Analogy: Building a car. You create objects like Engine, Wheel, and Door. Each object has its own properties and functions, and they interact with each other.

High level language Types

- **Scripting Languages:** These are often a subset of procedural languages designed for writing short, fast scripts to automate tasks, glue together components, or for web development. They are typically interpreted.
- Focus: Automating tasks and rapid development.
- Key Concepts: Dynamically typed, interpreted, high-level abstractions for specific tasks (e.g., web, system admin)
- Examples: Python, JavaScript, PHP, Ruby, Perl, Bash.

High level language Types

- **Multi-Paradigm Languages:** Most modern languages are designed to support more than one paradigm, giving developers flexibility.
- Examples:
- Python: Supports procedural, object-oriented, and functional programming.
- C++: Supports procedural, object-oriented, and generic programming.
- JavaScript: Supports procedural, object-oriented (prototype-based), and functional programming.
- Scala: Combines object-oriented and functional programming seamlessly.

OOP Vs POP

Feature	Object-Oriented Programming (OOP)	Procedural-Oriented Programming (POP)
Core Approach	Programs are built around **objects** and **data**.	Programs are built around **functions** and **procedures**.
Program Structure	**Bottom-Up**: Design the objects first, then how they interact.	**Top-Down**: Break the main problem into smaller sub-procedures.
Data & Function Relationship	**Tightly Coupled**: Data and functions (methods) are bound together in objects.	**Loosely Coupled**: Data and functions are separate entities.
Core Principles	**Abstraction, Encapsulation, Inheritance, Polymorphism.**	Not formally defined. Relies on sequencing, selection, and iteration.
Data Security	**High (via Encapsulation)**: Data is often private and can only be accessed through methods.	**Low**: Data is typically global and accessible throughout the program, leading to potential insecurity.

OOP Vs POP

Feature	Object-Oriented Programming (OOP)	Procedural-Oriented Programming (POP)
Code Reusability	**High (via Inheritance & Composition)**: New classes can inherit properties and methods from existing ones.	**Moderate**: Reusability is achieved by calling functions from different parts of the program.
Ease of Modification	**Easy**: Changes can be isolated within objects. Adding new features often doesn't break existing code.	**Difficult**: A change in a data structure may require modifying all the functions that access it.
Suitable For	Large, complex, and frequently updated systems (e.g., GUI apps, simulation games, enterprise software).	Smaller, simpler, or performance-critical tasks (e.g., system utilities, mathematical calculations, scripts).
Problem Solving View	Models the **"real-world"** entities and their interactions.	Models the **"procedure"** or steps to solve the problem.
Example Languages	Java, C++, Python, C#, Ruby	C, Pascal, Fortran, BASIC, Go

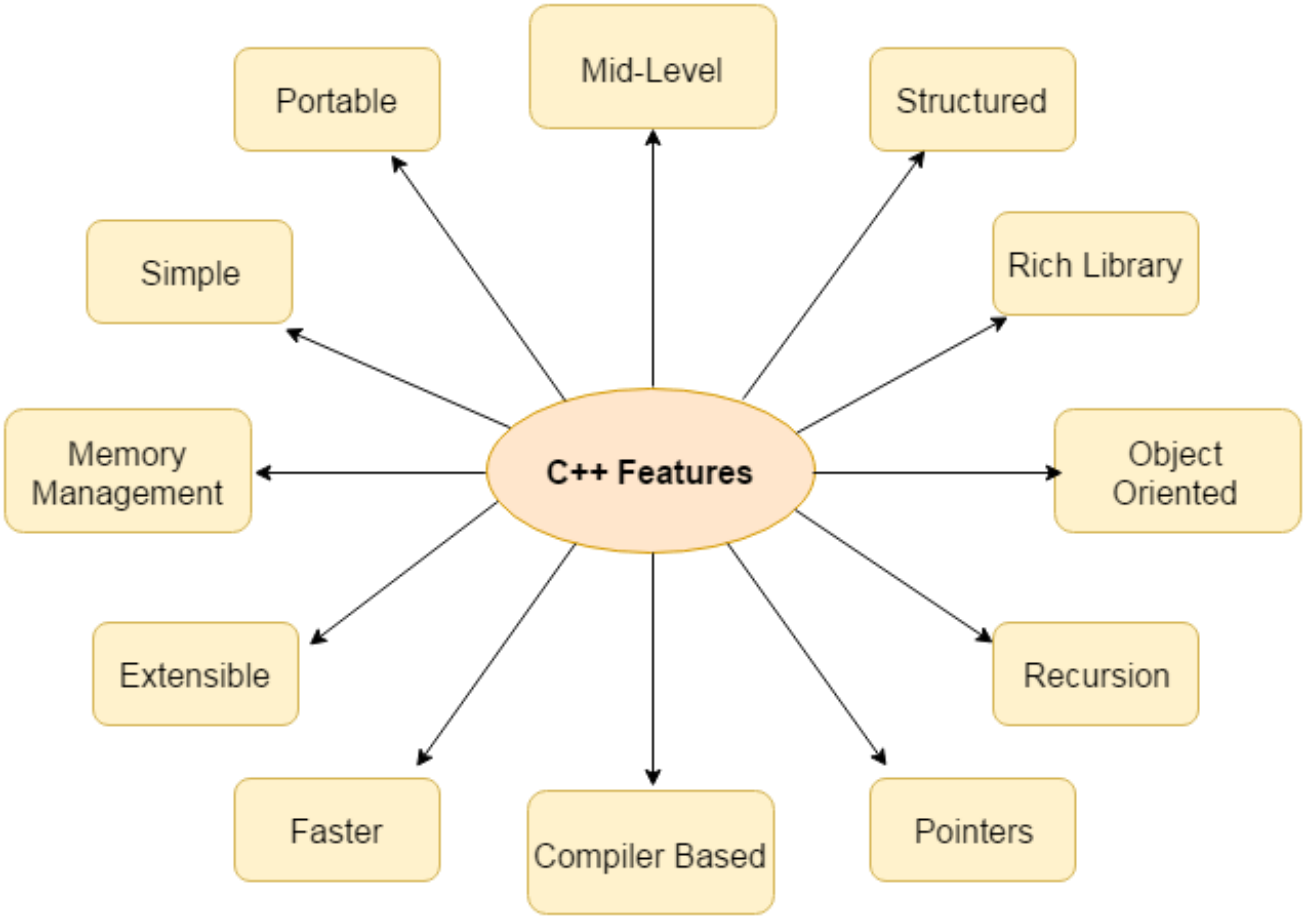
Introduction to C++

- C++ is a general-purpose, object-oriented programming language developed by Bjarne Stroustrup in 1979 as an extension of the C language.
- C++ is a middle-level programming language that combines the features of procedural programming (from C) and object-oriented programming (OOP).
- It allows developers to write efficient, low-level system programs as well as high-level applications with concepts like classes, inheritance, polymorphism, and abstraction.
- C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs.
- C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- C++ is portable and can be used to develop applications that can be adapted

Introduction to C++

- Simple: C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.
- Machine Independent or Portable: Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.
- Mid-level programming language: C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.
- Structured programming language: C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.
- Rich Library: C++ provides a lot of inbuilt functions that makes the

Introduction to C++



Introduction to C++

- Memory Management: It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.
- Speed: The compilation and execution time of C++ language is fast.
- Pointer: C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.
- Recursion: In C++, we can call the function within the function. It provides code reusability for every function.
- Extensible: C++ language is extensible because it can easily adopt new features.
- Object Oriented: C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
- Compiler based: C++ is a compiler based programming language, it means without compilation no C++ program can be executed.

C	C++
C follows the **procedural style programming.**	C++ is multi-paradigm. It supports both **procedural and object oriented.**
Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
C follows the **top-down approach.**	C++ follows the **bottom-up approach.**
C does not support function overloading.	C++ supports function overloading.
In C, you can't use functions in structure.	In C++, you can use functions in structure.
C does not support reference variables.	C++ supports reference variables.
In C, **scanf() and printf()** are mainly used for input/output.	C++ mainly uses stream **cin and cout** to perform input and output operations.
Operator overloading is not possible in C.	Operator overloading is possible in C++.
C programs are divided into **procedures and modules**	C++ programs are divided into **functions and classes.**
C does not provide the feature of namespace.	C++ supports the feature of namespace.
Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
C does not support the inheritance.	C++ supports inheritance.

C++ Character Set

- **Analogy:** Just like the English language uses letters (A-Z), digits (0-9), and symbols (., !, ?), C++ has its own set of valid characters that it understands.
- These are the characters you are allowed to use to write your C++ code:
- **Letters:** Both uppercase (A-Z) and lowercase (a-z). C++ is case-sensitive (age is different from Age).
- **Digits:** 0 to 9.

C++ Character Set

- **Special Symbols:** These are the punctuation marks of C++. The most common ones you'll see are:

  + - * / = % & # ! ? ^ ~ \ | < > ( ) [ ] { } : ; . , _ " '

- **White Spaces:** Spaces, tabs (\t), newlines (\n). The compiler mostly ignores these, but they are crucial for making your code readable.

- **Key Takeaway:** You must write your code using only these characters. Using a curly quote “ ” instead of a straight quote " or a symbol like $ in the wrong place will cause an error.

C++ Character Set

- **Special Symbols:** These are the punctuation marks of C++. The most common ones you'll see are:

  + - * / = % & # ! ? ^ ~ \ | < > ( ) [ ] { } : ; . , _ " '

- **White Spaces:** Spaces, tabs (\t), newlines (\n). The compiler mostly ignores these, but they are crucial for making your code readable.

- **Key Takeaway:** You must write your code using only these characters. Using a curly quote “ ” instead of a straight quote " or a symbol like $ in the wrong place will cause an error.

Tokens

- Tokens are the individual words and punctuation marks.
- They are the smallest meaningful elements that the compiler can understand.
- When you write a line of code, the compiler breaks it down into tokens before analysing. The main types of tokens are:

1) Keywords

- It is predefined reserve words.
- These are reserved words that have a special, fixed meaning in C++. You cannot use them as variable names. Examples:
- int, float, double, char, void, if, else, for, while, do, return, class, public, private, namespace etc.

Tokens

2) Identifiers

- These are names given by the programmer to various program elements like variables, functions, classes, etc.
- Rules for naming identifiers:
- Must start with a letter (a-z, A-Z) or an underscore (_).
- Cannot be a keyword.
- Case-sensitive.
- Good identifiers: totalSum, _count, player2, calculateArea
- Bad identifiers: 2player (starts with digit), float (is a keyword), total sum (has a space between)

Tokens

3) Literals/Constants

- These are fixed values that don't change during execution. They are literally what you write.
- Literals are the actual values that are directly written in the code to represent specific data. They are used to provide initial values for variables.
- **Integer Literals**: 10, -5, 0
- **Floating-point Literals**: 3.14, -0.5, 2.0
- **Character Literals:** 'A', 'z', '$' (must be in single quotes)
- **String Literals:** "Hello World" (must be in double quotes)

Tokens

4) Operators

- Symbols that perform an operation on one or more operands (values).
- Arithmetic: +, -, *, /, % (modulus - gives remainder)
- Relational: >, <, ==, !=, >=, <= (check conditions)
- Assignment: = (assigns a value)

Tokens

5) Punctuators/Separators

- Special Symbols used to structure the code.
- Semicolon ;: Acts like a full stop. It marks the end of a statement.
- int age = 25; // Statement ends here
- Curly Braces { }: Used to group a block of code (e.g., for a function or a loop).
- Parentheses ( ): Used after function names and in expressions.

Comment

- Can use C form of comments **/* A Comment */**
- Can also use // form:
- when // encountered, remainder of line ignored
- It works only on that line.
- **Examples:**
- **int I; // One Line Comment**
- **char C; /* Multiline comment */**

Precedence and Associativity

- Remember BODMAS/BIDMAS/PEMDAS from math?
- 5 + 2 * 3 is 11, not 21, because multiplication (*) has a higher precedence than addition (+). C++ has the same rules!
- **Precedence:** Which operator is evaluated first in an expression with multiple different operators.
- **Associativity:** If two operators have the same precedence, which one is evaluated first? Left-to-right (most common) or right-to-left (e.g., assignment =).

Precedence and Associativity

- **Simple Example:**
- int result = 5 + 2 * 3;
- * has higher precedence than +, so 2 * 3 is done first -> 6.
- Then 5 + 6 is done -> 11.
- Finally, = assigns 11 to result.
- Pro Tip: When in doubt, use parentheses ( ) to force the order you want! (5 + 2) * 3 will give 21. It makes your intention clear and your code safer.

Data Types

- A data type specifies the type of data that a variable can store such as integer, floating, character etc.
- Data types are like different types of containers. You wouldn't store water in a cardboard box. Similarly, you use different data types to store different kinds of data efficiently.

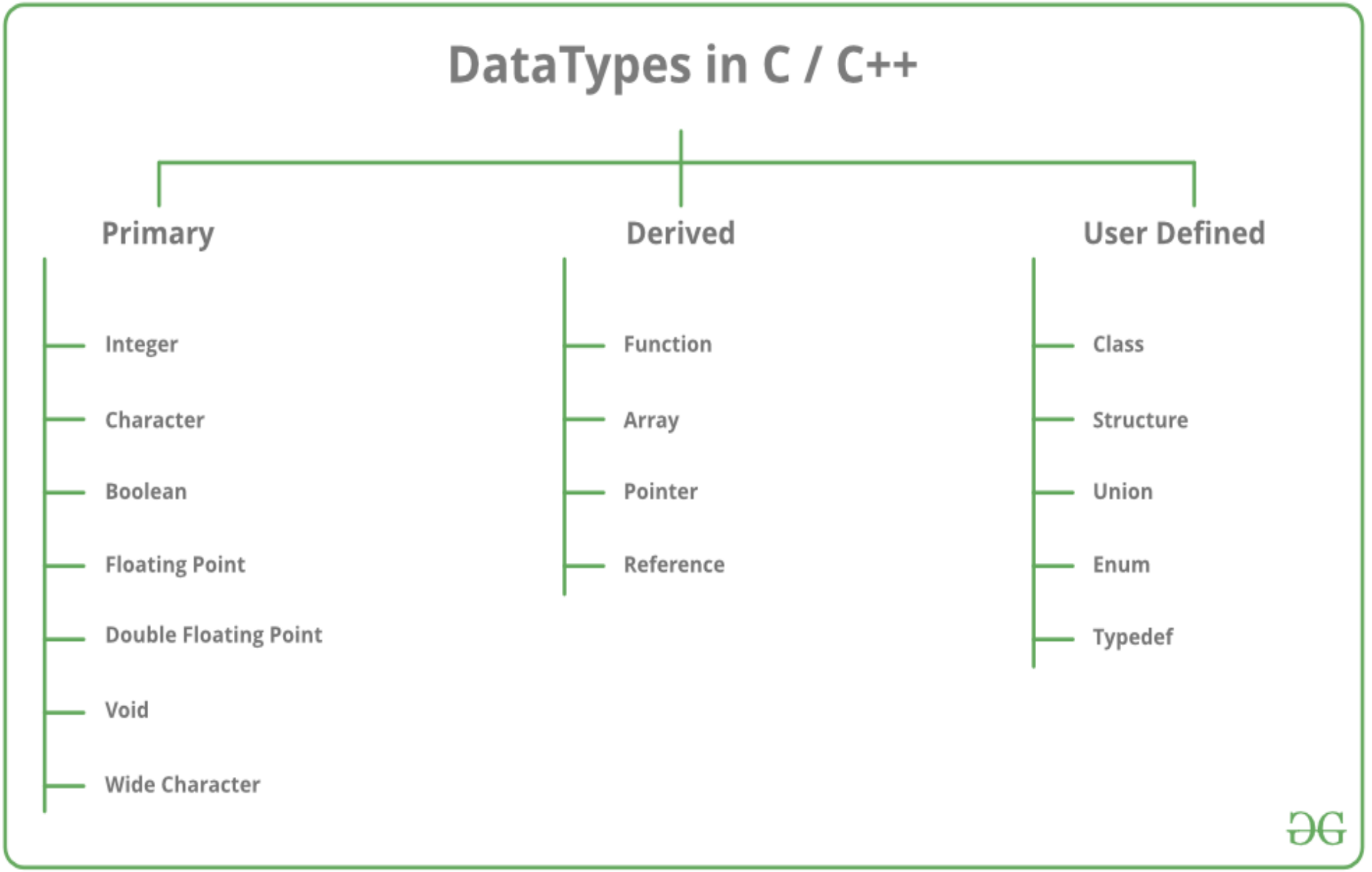
Data Type	What it stores	Example	Size (approx.)
int	Integers (whole numbers)	10, -5, 0	4 bytes (Range – 32768 to 32767)
float	Floating-point numbers (decimals)	3.14, -10.5	4 bytes
double	Double-precision decimals (more precise)	3.1415926535	8 bytes

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Data Types

Data Type	What it stores	Example	Size (approx.)
char	A single character	'A', '!', '5'	1 byte (Range -128 to 127)
bool	Boolean values (true/false)	true, false	1 byte
void	Represents "no type"	Used for functions that return nothing	N/A

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"



Variables

- A variable is like a named box (the variable name) where you can store a piece of data (the value). You can put things in the box, look at what's inside, and change it.
- **Declaration:** Telling the compiler, "I need a box of this type and here's its name."
- int age;
- **Assignment:** Putting a value into the box.
- age = 25;
- **Initialization:** Declaration and assignment in one step. (This is best practice!)
- int score = 100; // or int score {100}; (modern C++)

Program Structure

- Think of writing a C++ program like building a house. You need to follow a specific plan.
- **1. The Foundation:** The main() Function
- Every single C++ program must have one special room where everything starts. This room is called the main() function.
- It's the front door. When you run your program, the computer always looks for main() and starts executing the code inside it first.

```
int main() {
    // Your instructions go here!
    return 0;
}
```

Program Structure

- int main(): This is the name and type of the function. Just memorize this for now.
- { }: The curly braces mark the beginning and end of the main() room. All the action happens between them.
- return 0;: This is the "all done!" signal to the computer. 0 means everything finished successfully.
- **2. Getting Your Tools:** #include Directives
- Before you can build anything, you need to get your tools out of the toolbox.
- #include is like opening a toolbox. It tells the computer, "Go get a set of ready-made tools I need for this job."
- *These tools are stored in header files (like <iostream>).*

Program Structure

- These tools are stored in header files (like <iostream>).
- **3. Giving Instructions:** Statements
- Inside the main() function, you write the instructions, step-by-step. Each instruction is called a statement.
- Every statement must end with a semicolon ;
- **4. The Grand Finale:** Putting It All Together
- Let's build our simple "Hello Class" house using all these parts.

Program Structure

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, Class!";
    return 0;
}
```

Operators

- Operators are special symbols that perform operations on variables and values. Think of them like the basic math symbols you already know (+, -, ×, ÷) but with more capabilities.
- **1. Arithmetic Operators (For basic math)**

Operator	Example	Same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2

Operators

- **Arithmetic Operators:**
- Used for mathematical calculations.
- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo - remainder of division)
- ++ (Increment)
- -- (Decrement)

Operators

- Operators are symbols that instruct the compiler to perform specific mathematical, relational, logical, or bitwise operations on operands.
- **Assignment Operators (Giving values to variables)**

Operator	Example	Same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2

Operators

- **Assignment Operators:**
- Used to assign values to variables.
- = (Simple assignment)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulo and assign)

Operators

- Operators are fundamental to C++ programming, enabling data manipulation, control flow, and complex computations.

3. Comparison Operators (Compare values - return TRUE or FALSE)

Operator	Name	Example	Result
==	Equal to	5 == 3	false
!=	Not equal to	5 != 3	true
>	Greater than	5 > 3	true
<	Less than	5 < 3	false
>=	Greater than or equal	5 >= 5	true
<=	Less than or equal	5 <= 3	false

Operators

- **Relational (Comparison) Operators:**
- Used to compare two operands and determine the relationship between them, resulting in a boolean (true/false) value.
- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

Operators

- **Logical Operators:** Used to combine or modify boolean expressions.
- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)
- **Bitwise Operators:** Used to perform operations on individual bits of integer operands.
- & (Bitwise AND)
- | (Bitwise OR)
- << (Left Shift)
- >> (Right Shift)

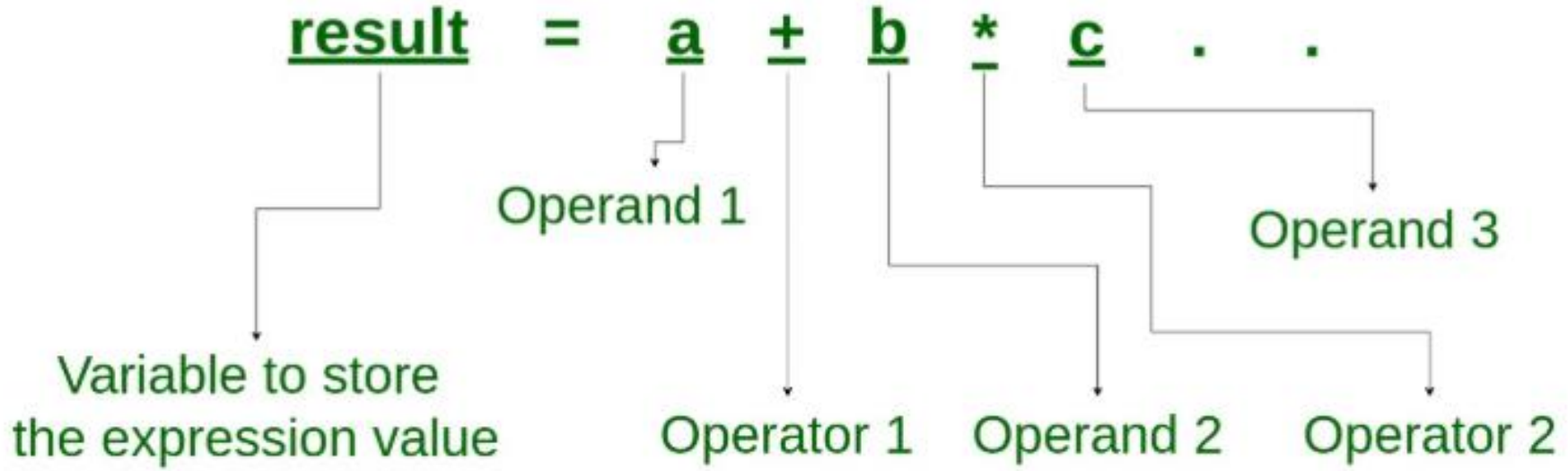
Ternary /Conditional Operator

Expression1 ? Expression2 : Expression3

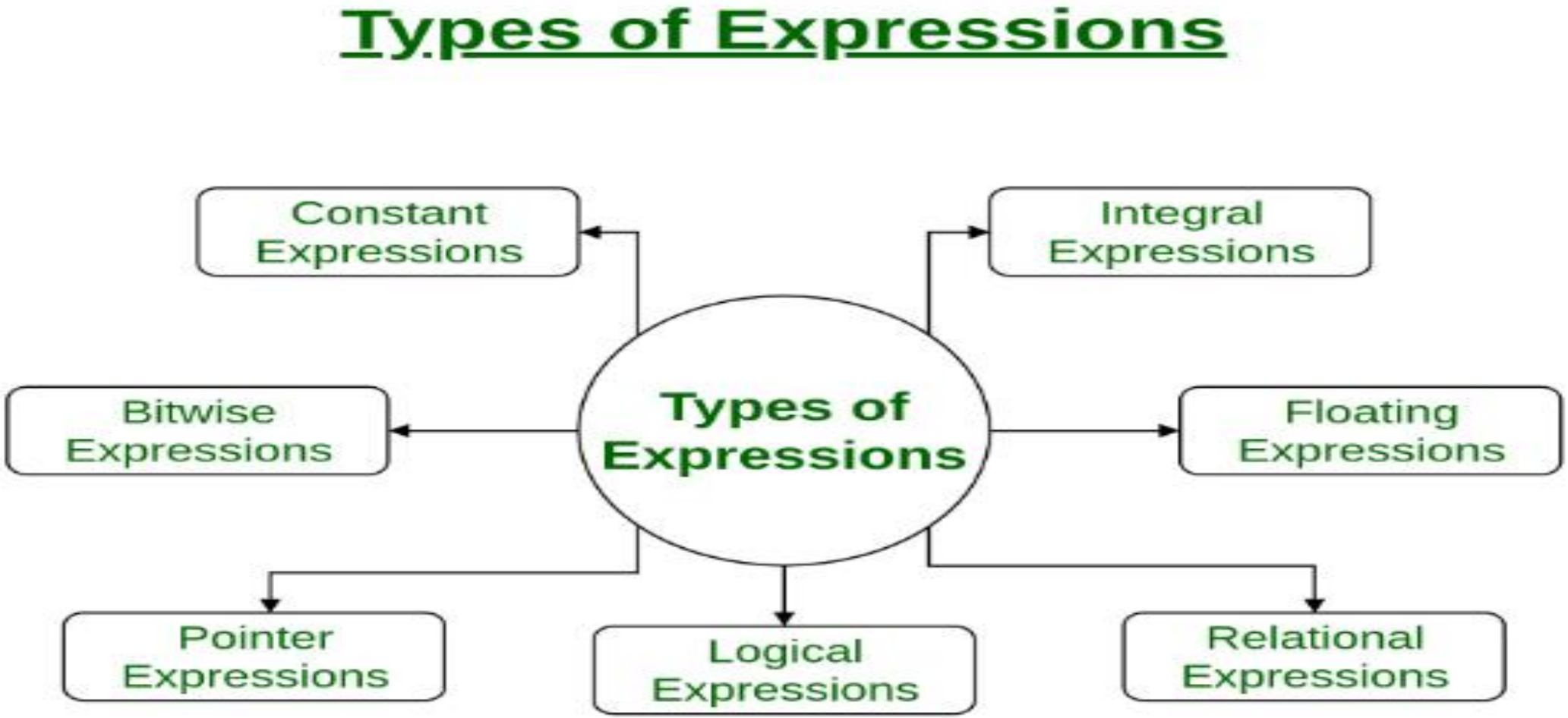
```
#include <iostream>
using namespace std;
int main() {
    int a = 3, b = 4;
    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number "
         "is " << result;
    return 0;
}
```

Expression

- An expression is a combination of operators, operands to evaluates a single value.
- An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.



Expression


Types of Expressions
Constant Expressions
Integral Expressions
Bitwise Expressions
Types of Expressions
Floating Expressions
Pointer Expressions
Logical Expressions
Relational Expressions

Expression

- **Constant expressions**: Constant Expressions consists of only constant values. A constant value is one that doesn't change. Expressions whose values can be determined at compile-time.
- Examples: 5, 'x'
- **Integral expressions:** Integral Expressions are those which produce integer results after implementing all type conversions.
- Examples: x, x * y, where x and y are integer variables.
- **Floating expressions:** Float Expressions are which produce floating point results after implementing all type conversions.
- Examples: x + y where x and y are floating point variables, 10.75

Expression

- **Relational expressions:** Use relational operators (==, !=, >, <) to compare two operands and produce a boolean result (true or false).
- Examples: x <= y, x + y > 2
- **Logical expressions:** Combine two or more relational expressions using logical operators (&& for AND, || for OR, ! for NOT) to produce a boolean result.
- Examples: x > y && x == 10, x == 10 || y == 5
- **Pointer expressions:** Pointer Expressions produce address values. Examples: &x, ptr, ptr++

Expression

- **Bitwise expressions:** Perform operations on data at the bit level using bitwise operators (&, |,<<, >>).
- Bitwise Expressions are used to manipulate data at bit level. They are basically used for **testing or shifting** bits.
- Examples:
- x << 3 (it shifts three bit position to left.)
- int shifted_value = 5 << 1;.
- shifts one bit position to right.

Characteristics of Expression

➢Expressions are formed by combining:

1. **Operators:** Symbols that perform operations (e.g., +, -, *, /, =, ==, &&, ||).
2. **Operands:** The values or variables on which operators act. These can be constants (e.g., 5, 3.14, 'a'), variables (e.g., x, myVariable), or the return values of function calls (e.g., sqrt(9)).
3. **Evaluation:** Expressions are evaluated to produce a result, which can be of various data types (e.g., int, double, bool).
4. **Side Effects:** Some expressions can also cause "side effects," which are actions beyond simply computing a value, such as modifying the value of a variable (e.g., x = 5; or i++;).
5. **Hierarchy:** Expressions can be nested, meaning one expression can be part of a larger expression. During evaluation, inner expressions are typically computed first.

Statements

- A statement is a single instruction that performs an action.
- such as declaring a variable, assigning a value, or calling a function.
- Most statements end with a semicolon (;).
- Example:
- int x = 10; // Declaration and assignment statement
- cout << "Hello"; // Output statement
- return 0; // Return statement

Control Structure

- Control structures manage the flow of execution within a program.
- It enable decision-making, repetition, and jumping between different parts of the code.

1. **Selection (Decision-Making) Statements:** These execute specific blocks of code based on conditions.
2. **Iteration (Looping) Statements:** These repeatedly execute a block of code.
3. **Jump Statements:** These alter the normal flow of control unconditionally.

Selection (Decision-Making) Statements

- These execute specific blocks of code based on conditions.

1. **if statement:** Executes a block of code if a condition is true.

Example:

```
if (age >= 18)
{
    cout << "Eligible to vote.";
}
```

Selection (Decision-Making) Statements

- These execute specific blocks of code based on conditions.

2. **if-else statement:** Executes if block (first block) when a condition is true, and executes else block (another block) when condition is false.

```
if (score > 90)
 {
   cout << "Excellent!";
 }
 else {
        cout << "Keep practicing.";
      }
```

Selection (Decision-Making) Statements

- These execute specific blocks of code based on conditions.

3. **if-else if-else ladder:** Checks multiple conditions sequentially.

Example:

```
if (grade == 'A') { /* ... */ }
else if (grade == 'B') { /* ... */ }
else { /* ... */ }
```

Selection (Decision-Making) Statements

- These execute specific blocks of code based on conditions.

3. **switch statement:** Selects one of many code blocks to execute based on the value of an expression.

Example: int day =1;

```
switch (day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    default: cout << "Invalid day";
}
```

Iteration (Looping) Statements

- Control structures manage the flow of execution within a program.
- These repeatedly execute a block of code.

1. **for loop:** Repeats a block of code a specific number of times, with initialization, condition, and increment/decrement.

Example:

```
for (int i = 0; i < 5; i++)
{
   cout << i << " ";
 }
```

Iteration (Looping) Statements

2. **while loop:** Repeats a block of code as long as a condition remains true (entry-controlled).

Example:

```
int count = 0;
while (count < 5)
{
   cout << "Looping...";
   count++;
 }
```

Iteration (Looping) Statements

3. **do-while loop:** Repeats a block of code at least once, then continues as long as a condition remains true (exit-controlled).

Example:

```
int i = 0;
do {
        cout << i;
        i++;
    }
while (i < 0);
// Executes once even if condition is false
```

Jump Statements

- It manage the flow of execution within a program.
- These alter the normal flow of control unconditionally.

1. **break statement:** Terminates the innermost loop or switch statement.
2. **continue statement:** Skips the rest of the current iteration of a loop and proceeds to the next iteration.
3. **goto statement:** Transfers control to a labeled statement within the same function (generally discouraged due to potential for unstructured code).
4. **return statement:** Exits a function and optionally returns a value.

I/O Operations

- C++ utilizes a stream-based approach for Input/Output (I/O) operations, treating data as a sequence of bytes flowing between a program and external devices like the keyboard, screen, or files.
- The core of C++ I/O lies in the <iostream> header file and its associated classes.
- Standard I/O Streams: (cin & cout)
- Input Stream: If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
- Output Stream: If the direction of flow of bytes is opposite, i.e. from main memory to device (display screen) then this process is called output.

Standard I/O Streams

- **cin (Standard Input Stream):** This object, an instance of the istream class.
- It is used to read input from the standard input device, typically the keyboard.
- The extraction operator (>>) is used with cin to extract data and store it in variables.
- C++ utilizes a stream-based approach for Input/Output (I/O) operations, treating data as a sequence of bytes flowing between a program and external devices like the keyboard, screen, or files.
- The core of C++ I/O lies in the <iostream> header file and its associated classes.

Standard I Stream- cin

1. **cin (Standard Input Stream):**

```
#include <iostream>
using namespace std;
int main() {
    int num;
    cout << "Enter an integer: ";
    cin >> num; // Reads an integer from the keyboard
    return 0;
}
```

Standard O Streams- cout

- **cout (Standard Output Stream):** This object, an instance of the ostream class.
- It is used to display output to the standard output device, typically the console screen.
- The insertion operator (<<) is used with cout to insert data into the stream for display.

```
#include <iostream>
int main()
{
        std::cout << “Your Name " << std::endl;
        return 0;
 }
```

cerr

- **cerr (Standard Error Stream):** An unbuffered ostream used for displaying error messages.
- **Un-buffered Standard Error Stream - cerr.**
- It is also an instance of the iostream class.
- it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.

```
#include <iostream>
using namespace std;
int main() {
    cerr << "An error occurred";
    return 0;
}
```

clog (Standard Log Stream)

- **clog (Standard Log Stream):** A buffered ostream used for displaying log messages.
- This is also an instance of ostream class.
- It is used to display errors.
- It is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.

```
#include <iostream>
using namespace std;
int main() {
    clog << "An error occurred";
    return 0;
}
```

Array

- An array is a collection of same type elements in contiguous memory locations.
- It allows you to store multiple values under a single name and access them using an index.
- An array is a data structure used to store std::vector (dynamic size) and std::array (fixed-size)
- Elements of Array are accessed using an index, starting from 0.
- **Fixed Size:** arrays have a fixed size determined at compile time.
- **Zero-based indexing:** Array indices always start from 0.
- **Bounds Checking:** bounds checking means accessing an index outside the declared range can lead to undefined behavior (e.g., crashes or corrupted data).

Array Declaration and Initialization

- Arrays must be declared with a specific data type, a name, and a size (number of elements).
- The size must be a constant expression, meaning it can be an integer literal or a const variable.
- // Declare an array of 5 integers

```
int num[5];
```

- // Declare and initialize an array with values

```
int score[] = {85, 90, 78, 92, 88};
```

- // Declare and initialize an array with a specified size

```
char letter[3] = {'a', 'b', 'c'};
```

Array Accessing Elements

- Elements in an array are accessed using their index within square brackets [].
- The first element is at index 0, the second at index 1, and so on.

```
int Num[3] = {10, 20, 30};
```

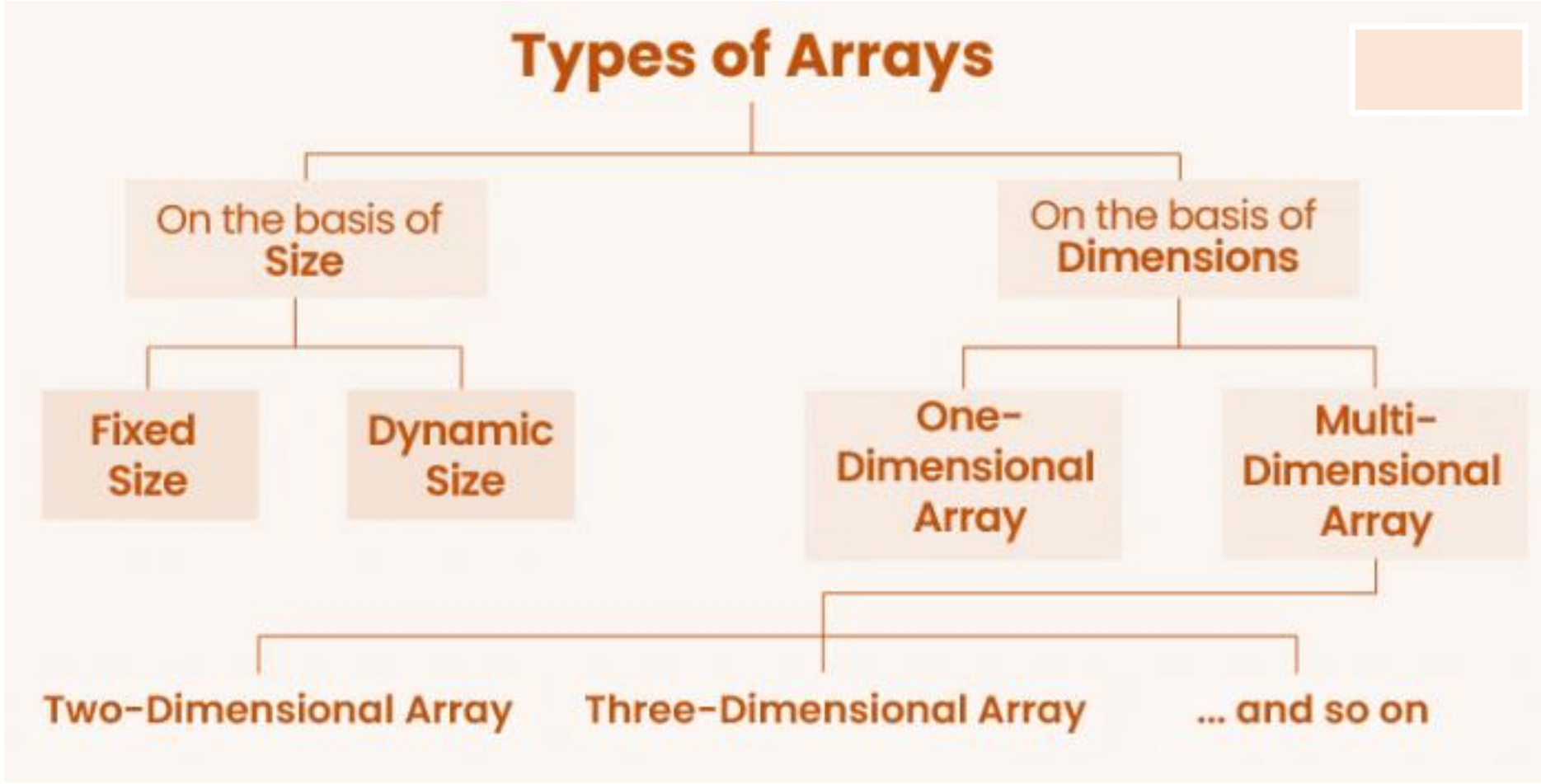
- // Accessing the **first** element (value 10) - index no. – [0]

```
int first = Num[0];
```

- // Modifying the **second** element - index no. – [1]

```
Num[1] = 25;
```

Types of Arrays

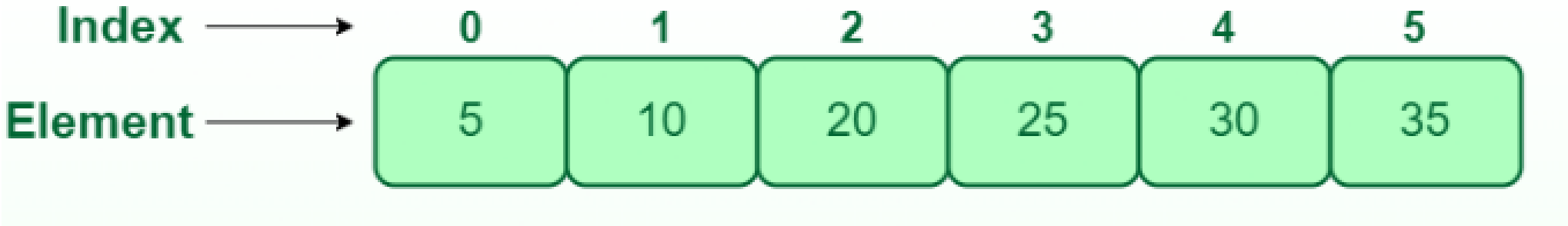


Types of Arrays

Arrays can be broadly categorized by their number of dimensions.

1. **One-dimensional arrays(1-D)-**Linear data
2. **Multidimensional arrays –(N-D)-**cubes or block-like structures
   (**Two-dimensional arrays- (2-D)-**Row-Column Matrix)

1. **Fixed Size Array -** Memory is allocated at compile time
2. **Dynamic Sized Array -** Memory is allocated dynamically,



One-Dimensional Arrays (1D Arrays)

- A linear collection of elements, often conceptualized as a single row or column.
- It is linear collection of elements, like a list.
- Each element is accessed by a single index.
- one-dimensional arrays used to store linear data.
- Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
                                        // An array of 5 integers
int thirdElement = numbers[2];
                                // Accessing the third element (index 2)
cout<< thirdElement;
                        // thirdElement will be 30
```

Two-Dimensional Arrays (2D Arrays)

- An array of arrays, often visualized as a grid or matrix with rows and columns. Elements are accessed using two indices: one for the row and one for the column.
- 2D array used to hold data of row-column matrix.
- An array of arrays, commonly used for representing grids or tables.
- **Example:**

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
// A 2x3 matrix
int element = matrix[1][2];
// Accessing the element in the second row (index 1) and third column (index 2)
cout<<element;
// element will be 6
```

Multi-Dimensional Arrays (2D Arrays)

- Arrays with more than two dimensions (e.g., 3D, 4D).
- Used for representing complex data structures like cubes or higher-dimensional spaces.
- An array composed of two-dimensional arrays, often visualized as a cube-like structure.

Example 3D Array:

```
int cube[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
// A 2x2x2 cube
// Accessing an element within the cube
int value = cube[1][0][1];
// value will be 6
```

Fixed Size Array

- These arrays have a predetermined size that cannot be changed after declaration.
- Fixed-Size Arrays have a size that is determined at the time of declaration and remains constant throughout the program's execution.
- Memory is allocated at compile time, and their size cannot be changed later.
- Example:

```
// declares an array arr that can hold exactly 5 integer elements
int arr[5];
// Another way (creation and initialization both)
int arr2[5] = {1, 2, 3, 4, 5};
```

Dynamic Size Array

- These arrays can change their size during program execution(runtime), allowing for more flexible storage.
- Memory is allocated dynamically, allowing elements to be added or removed as needed.
- The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need.
- The memory is dynamically allocated and de-allocated in these arrays.
- std::vector are examples of dynamic arrays that can automatically resize.
- #include<vector> // Dynamic Integer Array
- vector<int> v;

Dynamic Size Array

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
        int n;
cout << "Enter array size: ";
cin >> n;
vector<int> dynamicArray(n);
for (int i = 0; i < n; ++i)
{
 dynamicArray[i] = i * 10;
 }
cout << "Elements: ";
 for (int val : dynamicArray)
{
  cout << val << " ";
 }
  cout << endl;    return 0;}
```

Function

- A function is a block of organized, reusable code that designed to performs a specific task.
- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions used to perform certain actions
- It is reusable blocks of code. Define the code once in function, and use it many times.
- Functions can take inputs (parameters), execute a block of statements, and optionally return a result.
- Function improve code readability, modularity, and reusability.

Function Syntax

```
Function_Return_Type  Function_Name( Function_Parameters)
{
  // code to be executed
}
```

- *// Declaration*- void add( );
- //Definition- void add( ) { **cout<<**a + b; }
- //call- add( );

1. **Function Declaration(**function prototype) introduces the function to the compiler. informs the compiler about the function's existence.

   // Declaration- void add( );

Function

2. **Function Definition** provides the actual implementation. its name, and the types and order of its parameters.

   Definition- void add( ) { **cout<<**a + b; }

3. **Parameters(variable) and Arguments(values):**

- Parameters are placeholders in the function definition. Ex. int a
- Arguments are actual values passed during function calls.

Types of Functions

1. Library Functions: These are built-in functions provided by C++ standard libraries, such as cout, sqrt().

➢You can use them by including appropriate headers like <iostream>, <cmath>, or <string>.

```
#include <iostream> // Required for input/output
#include <cmath> // Required for mathematical function sqrt
using namespace std;
int main() {
float number = 25.0;
float SR = sqrt(number); // Calculates square root
cout << "Square root of " << number << " is: " << SR;
return 0;
}
```

Library Functions-Ex-2

```
#include <iostream>
#include <cmath> // Required for mathematical function pow
using namespace std;
int main() {
float base = 2.0;
float exponent = 3.0;
float p = pow(base, exponent); // Calculates power (base^exponent)
cout << base << " raised to the power of " << exponent << " is: " << p;
return 0;
}
```

Library Functions-Ex-3

```
#include <iostream>
#include <algorithm> // Required for min, max functions
using namespace std;
int main() {
int a = 10;
int b = 20;
int minimum = min(a, b); // Finds the smaller of two values
int maximum = max(a, b); // Finds the larger of two values
cout << "Minimum of " << a << " and " << b << " is: " << minimum << endl;
cout << "Maximum of " << a << " and " << b << " is: " << maximum << endl;
return 0;
}
```

Library Functions-Ex-4

```
#include <iostream>
#include <string>
// Required for string functions
using namespace std;
int main() {
    string str1 = "Hello, ";
    string str2 = "your name";
        // 1. Concatenation using + operator
    string result1 = str1 + str2;
    cout << "Concatenation using +: " << result1 << endl;
// 2. append() function
    string result2 = str1;
    result2.append(str2);
    cout << "Concatenation using append(): " << result2 << endl;

// 3. length() or size() function
    cout << "Length of str2: " << str2.length() << endl; // or str3.size()
    return 0;
}
```

User Defined Functions

User defined functions are based on input and return type.

1. **No parameters, no return value:** The function performs a task but does not take input or return anything.
2. **Parameters, no return value:** The function takes input but does not return a result.
3. **No parameters, return value:** The function returns a result but does not take any input.
4. **Parameters and return value:** The function takes input and returns a result.

Function-1. No parameters, no return value

```
#include <iostream>
Using namespace std;
// Create a function
void Fun( )
{
  cout << “Function Executed!";
}
int main( )
{
  Fun( ); // call the function
  return 0;
}
```

Function-1. No parameters, no return value

➢Example 2:

```
#include <iostream>
using namespace std;
// Function
void Hello( )
{
cout << “Hello class!“;
cout<<endl;
}

int main( )
{
Hello( ); // Calling the function
return 0;
}
```

Function-2. parameters, no return value

```
#include <iostream>
using namespace std;
void Sum(int n1, int n2)
{
 int sum = n1 + n2;
cout << "The sum is: " << sum << endl;
}
int main()
{
int a = 60;
int b = 55;
Sum(a,  b);          // fun call with different values
   Sum(20, 7);
return 0;
}
```

Function-3. no parameters, return value

```
#include <iostream>
using namespace std;
// Function definition: no parameters, return value
int get()
{
   // This function returns a  integer value
    return 42;
}
```

```
int main()
{
        // Call the function
int result = get();
         // Display the returned value
cout << "The number returned by the function is: " << result;
    return 0;
}
```

Function-4. parameters, return value

```
#include <iostream>
using namespace std;
// Function definition: parameters, return value
int add(int a, int b)
{
 int sum = a + b;
return sum;      // Return
}
```

```
int main()
 {
int n1 = 75;
int n2 = 57;
int sum = add(n1, n2);
cout << "The sum of " << n1 << " and "
<< n2 << " is: " << sum;
return 0;
 }
```

More than one Function

➢Example:

```
#include <iostream>
using namespace std;
// Function1
void Hello()
{
cout << “Hello Your Name!" << endl;
}
// Function2
int square(int num)
{
    return num * num;
}
```

```
int main()
{
Hello();    // Calling function1


int result = square(5);
// Calling the square function2
cout << "Square of 5 is: " << result << endl;
return 0;
}
```

Function Syntax

➢Example:

```
#include <iostream>
using namespace std;
// Function declaration
        (prototype)
        int add(int a, int b);
        int main()
        {
        int num1 = 10;
        int num2 = 20;
// Calling function
        int sum = add(num1, num2);
cout << "The sum is: " << sum << endl;
return 0;
}
// Function definition
int add(int a, int b)
{
return a + b; // Returns the sum of a and
b
}
```

Ex. functions

```
#include <iostream>
using namespace std;
// Defining function that prints given
   number
void print(int n)
{
    cout << n << endl;
}
```

```
int main()
 {
    int num1 = 10;
    int num2 = 99;
        // Calling print and passing both
    // num1 and num2 to it one by
   one
    print(num1);
    print(num2);
    return 0;
}
```

Function Overloading

Example: Function Overloading

```
#include <iostream>
using namespace std;
// Overloaded functions
int add(int a, int b)
{
return a + b;
}
double add(double a, double b)
{
return a + b;
}
```

```
int main()
{
cout << "Integer addition: " ;
cout<< add(3, 4) << endl;
cout << "Double addition: “;
cout<< add(3.5, 4.5) << endl;
return 0;
}
```

Allows two add functions with the **same name** but **different parameter** lists integer & double.