DSA With C++ and Java

L-1 Basic of Languages

L-1.1 Input and Output

When you embark on your C++ programming journey, it's perfectly okay not to dive too deep into the intricacies of the language right from the start. In fact, it's advisable to initially focus on grasping the big picture and building a strong foundation. In this guide, we'll walk you through the basic skeleton of a C++ program and the essential components you need to know to get started.

Including Libraries

C++ is a versatile language, and it relies on libraries to access various functionalities. To perform tasks like input and output, we include specific libraries at the beginning of our code. For instance, **#include<iostream>** is used for input and output operations, while **#include<math.h>** allows us to use mathematical functions. Simply put, libraries provide pre-built functions and tools for us to use in our code.

The Generic Skeleton

The generic skeleton of a C++ program consists of two main components: the **library inclusion** and the **main function**. After including the necessary libraries, you declare the main function using int main() { /* Your code here */ return 0; }. This serves as the entry point for your program.

```
#include<iostream>


int main() {

    // Your code here

    return 0;

}
```

Output with cout

To display output in C++, you'll commonly use the cout function from the iostream library. However, you need to specify that it belongs to the std (standard) namespace. For instance, **std::cout << "Hey, Striver!";** will print "Hey, Striver!" to the console. You enclose the text you want to display within double quotation marks.

Code:

```
#include<iostream>


int main() {

    std::cout << "Hey, Striver!";

    return 0;
```

```
}
```

Output: Hey, Striver!

If we want to print **Hey, Striver! twice in 2 lines** and we write **std::cout << "Hey, Striver!";** again and again then it will print it consecutively on the same line.

Code:

```
#include<iostream>


int main() {
    std::cout << "Hey,
Striver!";
    std::cout << "Hey,
Striver!";
    return 0;
}
```

Output: Hey, Striver!Hey, Striver!

You can use the **newline character \n** to insert a **line break** within a single std::cout statement. Here's the code and its corresponding terminal output:

Code:

```
#include<iostream>


int main() {
    std::cout << "Hey,
Striver!" << "\n";
    std::cout << "Hey,
Striver!";
    return 0;
}
```

Output:

```
Hey, Striver!
Hey, Striver!
```

As you can see, the newline character \n inserts a line break, but the second "Hey, Striver!" is still on the same line as the first one.
You can also use **std::endl** to insert a **newline character** and **flush the output buffer**. Here's the code and its corresponding terminal output:

Code:

```
#include<iostream>
```

```
int main() {
    std::cout << "Hey,
Striver!" << std::endl;
    std::cout << "Hey,
Striver!";
    return 0;
}
```

Output:

```
Hey, Striver!
Hey, Striver!
```

Using \n for line breaks in C++ is a common and efficient way to achieve the desired output. It's a simple and straightforward approach, and it's **typically faster** than other methods for adding line breaks, such as using std::endl.

The reason for this is that \n is a simple escape sequence that inserts a newline character, which is a **low-level operation** that directly m**oves the cursor to the beginning of the next line in the output**. On the other hand, std::endl not only adds a newline character but also **flushes the output buffer**. Flushing the buffer can be a more costly operation in terms of performance, especially when you're printing a large amount of text.

using namespace std

By adding using **namespace std;** at the beginning of your program, you're telling the compiler that you want to use all the **names from the std namespace** without explicitly specifying std:: each time. This can make your code cleaner and more concise.

Code:

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

int main() {
    cout << "Hey,
Striver!" << :endl;
    cout << "Hey,
Striver!";
    return 0;
}
```

Output:

```
Hey, Striver!
```

Hey, Striver!

"using namespace std;" is a useful shortcut for simplifying your C++ code, especially when you're learning the language and writing smaller programs. It helps **reduce clutter** and makes your **code more readable**. However, as your programming projects grow in complexity, consider using it sparingly to avoid potential naming conflicts. It's a tool that can make your C++ journey smoother, so use it wisely.

Taking User Input using cin

One of the fundamental aspects of programming is taking input from the user. In C++, this is achieved with the help of the **cin stream**, which allows you to **receive input** from the user via the terminal or console.

Code:

```
#include<iostream>

using namespace std;


int main() {

    int x;

    cin >> x;

    cout << "Value of x: " << x;

    return 0;

}
```

Input: 10

Output: Value of x: 10

When you run this program and enter a value (e.g., 10) in the terminal, cin captures that value, stores it in the variable x, and then displays it using cout. Here's how it works:

1. The program waits for user input.
2. You enter a value (e.g., 10) and press Enter.
3. cin reads the entered value and stores it in the variable x.
4. cout then displays the value of x.

To accept multiple inputs, we can simply use the **>> operator** with cin for each variable we want to receive input for. Let's demonstrate this by taking two variables, x, and y, as input and displaying their values:

Code:

```
#include<iostream>

using namespace std;


int main() {

    int x;
```

```
    int y;
    cin >> x >> y;
    cout << "Value of x: " << x << " and y: " <<y;
    return 0;
}
```

Input: 10 20

Output: Value of x: 10 and y: 20

When you run this program, it waits for two separate inputs from the user, which should be entered one after the other, separated by spaces or Enter key presses. Here's how it works:

1. The program waits for the first input (for x).
2. You enter a value (e.g., 10) and press Enter.
3. cin reads and stores the value in x.
4. The program then waits for the second input (for y).
5. You enter another value (e.g., 20) and press Enter.
6. cin reads and stores this value in y.

You might be wondering about the meaning of "int x" and "int y." Let's now dive into the topic of data types in C++.

Note:

To make the process more convenient, there's a shortcut that allows you to include almost all standard libraries at once using **#include<bits/stdc++.h>**.

The bits/stdc++.h header is a shortcut that includes a **vast number of standard C++ libraries**, making it easier to access a wide range of functions and classes without specifying each library individually. It's a **time-saving approach** for programmers, especially when you need several standard libraries in your code.

However, it's important to be aware of potential compatibility issues and consider the impact on compile time, especially in large projects. When used judiciously, it can be a valuable asset in streamlining your C++ development process.

Code:

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    int x;
    int y;
    cin >> x >> y;
    cout << "Value of x: " << x << " and y: " <<y;
```

```
    return 0;
}
```

Input: 10 20

Output: Value of x: 10 and y: 20

L-1.2 Data Types

What Are Data Types?

At the core of any programming language lies the concept of data types a way for the language to understand what kind of value a variable can hold. Data types define not only the nature of the data but also the operations you can perform on it and the amount of memory it will consume. If you think of a variable as a box, then the data type is the label on that box telling the computer what's inside and how to handle it. In some languages (like C++ and Java), you must explicitly declare the data type before using a variable. In others (like Python and JavaScript), the type is determined automatically at runtime based on the assigned value.

Data Types in C++

C++ is a statically typed language, meaning you must specify the type of every variable at compile time. This ensures type safety and better performance, but it also means the compiler will throw errors if you try to assign incompatible types.

Primitive Types:

int for integers (whole numbers).

float and double for decimal values, with double providing more precision.

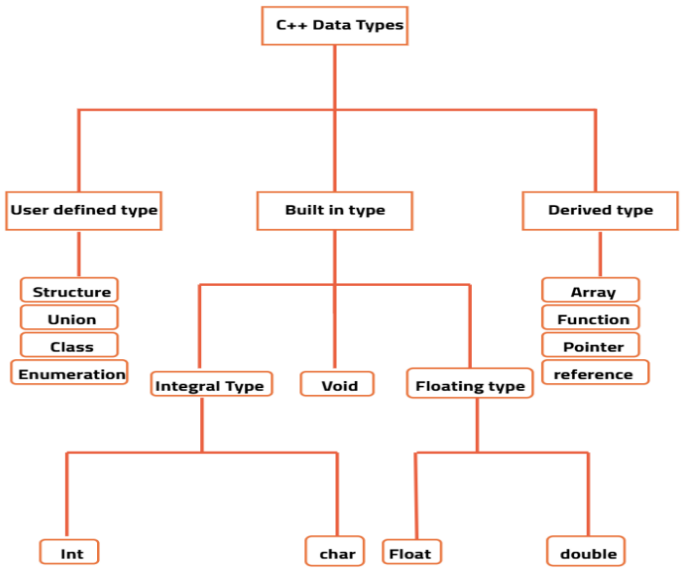
char for single characters.

bool for Boolean values (true or false).

Derived Types:

Arrays, pointers, references, and function types fall under this.

User-defined Types:



Structures (struct), classes (class), and enumerations (enum).

C++ also supports modifiers like unsigned, short, and long to tweak the size and range of numeric types. For example, unsigned int only stores non-negative integers but allows larger maximum values than a signed int.

Data Types in Java

Java is also statically typed, but unlike C++, it runs on the JVM (Java Virtual Machine) and offers automatic memory management via garbage collection. Data types in Java are split into two broad categories:

Primitive Types (fixed in size and stored directly in memory):

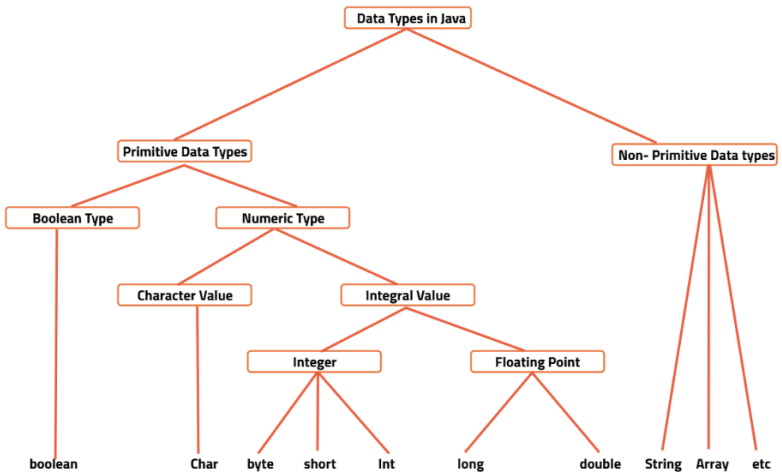
byte, short, int, long for integers of increasing size.

float, double for floating-point numbers.

char for Unicode characters.

boolean for logical true/false values.

Non-Primitive (Reference) Types:



These include objects like String, arrays, user-defined classes, and interfaces. A reference type variable holds a memory address pointing to the actual object in the heap.

One key difference from C++ is that Java's boolean type can only be true or false you can't treat it as a number.

Data Types in Python

Python is dynamically typed, meaning you don't need to declare the data type the interpreter figures it out at runtime. This makes coding faster but can also lead to subtle bugs if you're not careful.

Numeric Types:

int for integers (unlimited size).

float for decimal numbers (double precision).

complex for complex numbers (3+5j).

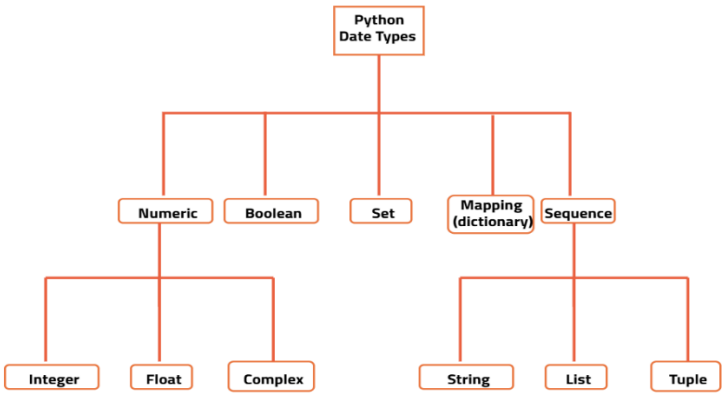
Sequence Types: list, tuple, and range for ordered collections.

Text Type: str for strings.

Mapping Type: dict for key-value pairs.

Set Types: set and frozenset for unordered collections of unique elements.

Boolean Type: bool for logical operations.



Python is flexible you can assign an integer to a variable, then later assign a string to the same variable without errors. But this flexibility means you have to be more cautious with type-related logic.

Data Types in JavaScript

JavaScript is also dynamically typed but is often considered more "loose" than Python when it comes to type conversions. It has fewer built-in types, but its type coercion rules can cause unexpected behavior if you're not careful.

Primitive Types:

number for both integers and floating-point numbers.

number for both integers and floating-point numbers.

string for text.

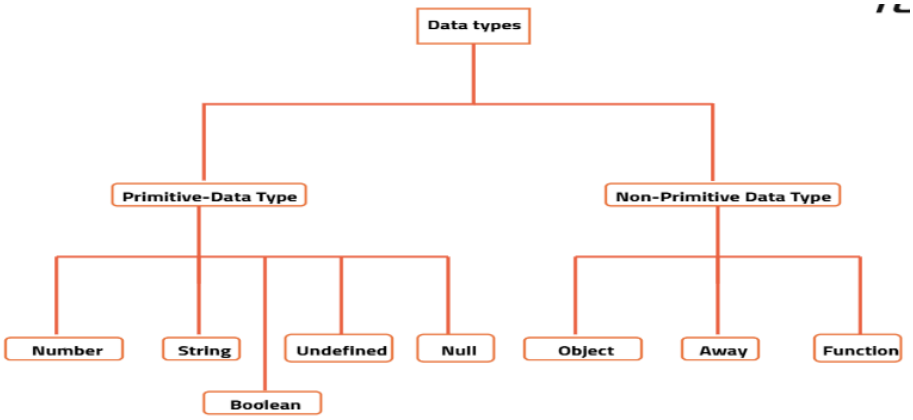
boolean for true/false.

undefined for variables declared but not assigned a value.

null for intentional absence of a value.

symbol for unique identifiers.

Objects: Arrays, functions, and all other non-primitive structures in JS are objects.



One important quirk: In JavaScript, typeof null returns "object", which is a long-standing bug from the early days of the language.

Why Data Types Matter

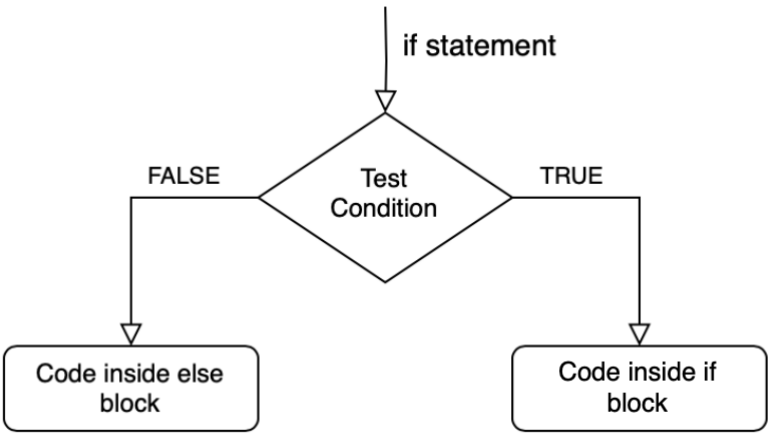
Regardless of the language, understanding data types is crucial for writing efficient, bug-free programs. In statically typed languages like C++ and Java, they help catch errors before running the code and allow for optimizations at compile time. In dynamically typed languages like Python and JavaScript, they offer flexibility but require extra care to avoid type mismatch errors. Data types also determine how much memory your variables use, how they're stored, and what operations can be performed on them.

Language	Core Types (Numeric, Text, Boolean)	Collections	Special Values	Key Features
C++	**int, long, float, double, char, bool, std::string**	**std::vector, std::map, std::set, std::array**	**nullptr**	**Static typing;** value semantics; manual memory management.
Java	**int, long** (primitives), **double, char, boolean, String**	**ArrayList, HashMap, HashSet,** arrays	**null**	**Static typing;** distinction between primitives and objects; String is immutable.
Python	**int** (unlimited), **float, str, bool**	**list, tuple, dict, set**	**None**	**Dynamic typing;** everything is an object; clear distinction between mutable and immutable types.
JavaScript	**number, bigint, string, boolean**	**Array,** Object ({}), **Map, Set**	**null, undefined**	**Dynamic typing;** numbers are floating-point by default; has type coercion.

L-1.3 Conditional Statements (If-Else)

Conditional statements are a fundamental concept in programming that allows you to make decisions based on certain conditions. These statements enable your code to **execute different blocks of cod**e depending on whether **specific conditions** are met or not. In this blog post, we'll delve into the basics of **conditional statements**, starting with the ubiquitous **if-else** statement and gradually exploring more complex scenarios.

The `if-else` Statement



'if statement' is used to execute a block of code only if a certain condition is met. It allows us to conditionally execute code based on whether the specified condition is true.

'else statement', on the other hand, is an optional companion to the if statement. It specifies what code to execute if the condition in the if statement is not met (i.e. if it is false).

Let's break down the flow of control:

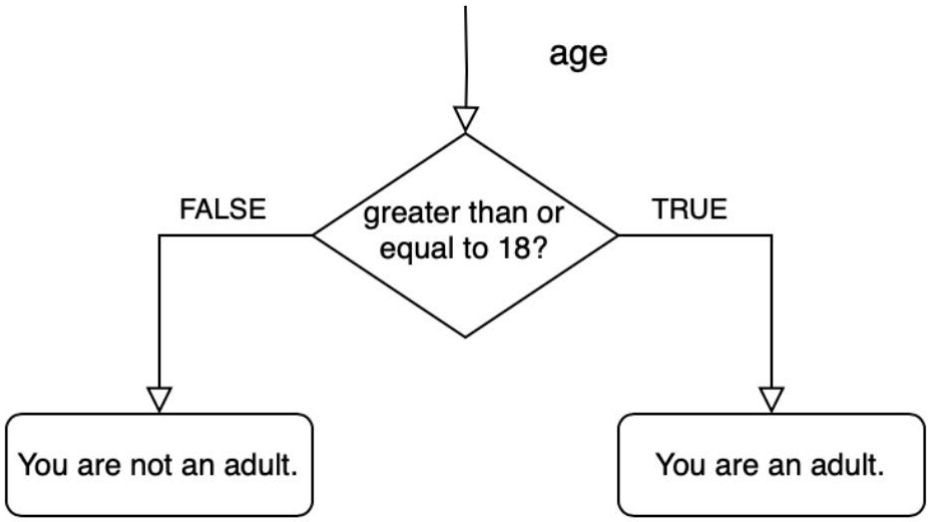
- If the test condition in the if statement is true, a block of code inside the if block will be executed.
- If the test condition is false, the code inside the else block (if present) will be executed.

Code:

C++Java

```
#include<bits/stdc++.h>

using namespace std;


int main() {

int age=10;



if (age >= 18) {

    cout << "You are an adult." << endl;

} else {

    cout << "You are not an adult." << endl;

}


}
```

Output: You are not an adult.



In this example, we take the **user's age as input** and use an **if statement** to check if the age is greater than or equal to 18. If the condition is true, it prints "You are an adult," and if it's false ie. the '**else**' condition is satisfied, it prints "You are not an adult."

Simplifying Code with "else if"

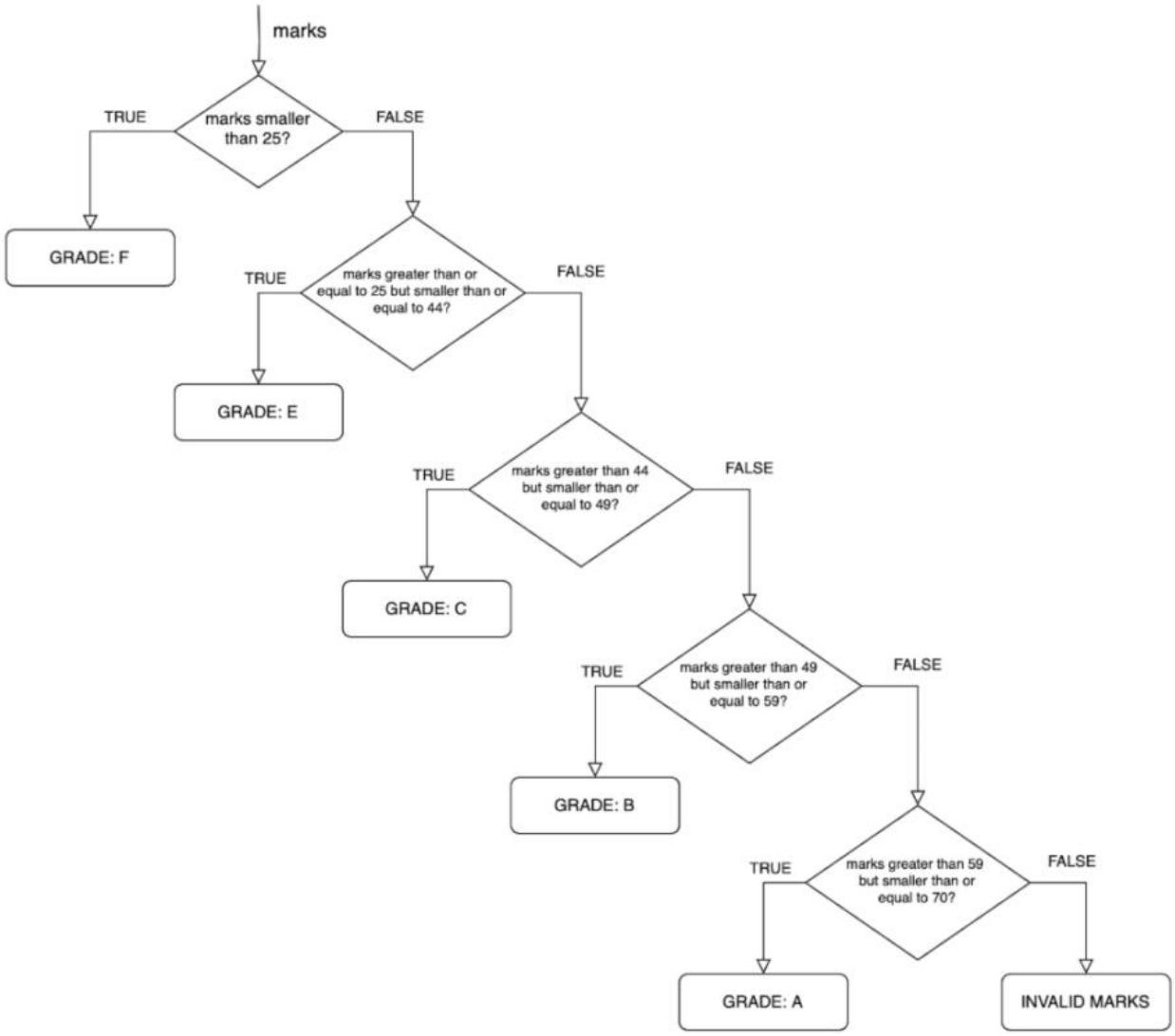
As your code becomes more complex, you'll often encounter scenarios where you need to check multiple conditions. Instead of writing multiple independent if statements, you can streamline your code using else if statements.

Let's say we want to grade students based on their marks within specific ranges:

Code:

```
#include <iostream>

using namespace std;


int main() {
    int marks = 54;



    if (marks < 25) {
        cout << "Grade: F" << endl;
    } else if (marks >= 25 && marks <= 44) {
        cout << "Grade: E" << endl;
    } else if (marks >= 45 && marks <= 49) {
        cout << "Grade: D" << endl;
    } else if (marks >= 50 && marks <= 59) {
        cout << "Grade: C" << endl;
    } else if (marks >= 60 && marks <= 69) {
        cout << "Grade: B" << endl;
    } else if (marks >= 70) {
        cout << "Grade: A" << endl;
    } else {
        cout << "Invalid marks entered." << endl;
    }


    return 0;
}
```

Output: Grade: C



- If marks are less than 25, it prints "Grade: F."
- If marks are between 25 and 44 (inclusive), it prints "Grade: E."
- If marks are between 45 and 49 (inclusive), it prints "Grade: D."
- If marks are between 50 and 59 (inclusive), it prints "Grade: C."
- If marks are between 60 and 69 (inclusive), it prints "Grade: B."
- If marks are 70 or higher, it prints "Grade: A."
- If marks are outside the valid range, it prints "Invalid marks entered."

The provided code for grading based on marks is functional, but it **can be simplified** for better readability and maintainability. In the current code, there are **several redundant comparisons** of marks with specific values. For example, when checking for grades E, C, B, and A, you have to **repeatedly check** marks >= X && marks <= Y, which can be error-prone and hard to maintain as the grade ranges change.

We can refactor the code, we can **remove the lower bounds** and check only the **upper bounds** for each grade as the code flow is such that we move along the conditions only after satisfying the previous one.

Code:

```
#include <iostream>

int main() {
    int marks = 54;
    char grade;

    if (marks < 25) {
        grade = 'F';
    } else if (marks <= 44) {
        grade = 'E';
    } else if (marks <= 49) {
        grade = 'D';
    } else if (marks <= 59) {
        grade = 'C';
    } else if (marks <= 69) {
        grade = 'B';
    } else if (marks >= 70) {
        grade = 'A';
    } else {
        grade = 'X'; // Use 'X' to indicate invalid marks
    }

    std::cout << "Grade: " << grade << std::endl;

    return 0;
}
```

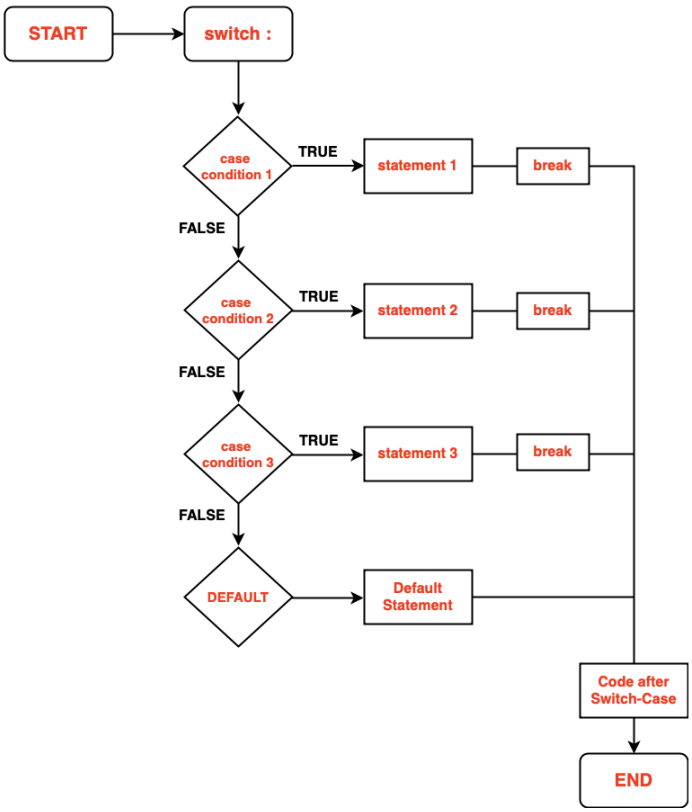
Output: Grade: C

Conditional statements are indispensable tools for **controlling the flow of your program.** Whether you're making simple decisions or handling complex logic, if-else statements and their variants empower you to write code that **responds dynamically to changing conditions.** Mastering these fundamentals is essential for any aspiring programmer.

L-1.4 Conditional Statements (Switch Case)

If-else statements are like the Swiss Army knife of decision-making. They offer flexibility and can handle a wide range of conditions and branching logic. You can use them when you need to **evaluate complex conditions** or when **conditions aren't based on simple equality checks.** If-else statements are often the preferred choice for scenarios where the **conditions are not easily enumerable** or where you need to execute different blocks of code based on various conditions.
On the other hand, **switch statements shine when you have a single variable to compare against multiple distinct values.** They are concise, making the code cleaner and more structured. So, if-else statements and switch statements can complement each other, with if-else statements handling complex conditions and switch statements simplifying cases with multiple exact matches.

The 'Switch' Statement



To illustrate the switch statement, let's consider a common problem: given a number between 1 and 7, print the corresponding day of the week. Here's how we can use the switch statement for this task:

Code:

```
#include <iostream>

using namespace std;


int main() {
```

```
    int day = 4;

    switch (day) {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
            cout << "Wednesday";
            break;
        case 4:
            cout << "Thursday";
            break;
        case 5:
            cout << "Friday";
            break;
        case 6:
            cout << "Saturday";
            break;
        case 7:
            cout << "Sunday";
            break;
        default:
            cout << "Invalid";
    }

    return 0;
}
```

Output: Thursday

In this example, if you set the **day** variable to 4, it will print **"Thursday"** since it matches the case 4 statement. The **break statement is**

crucial; it ensures that the switch statement exits after the matching case is executed, preventing the execution of subsequent cases.

The Default Case: The default case serves as a **safety net**. If **none of the cases match** the expression, the code inside the **default block will execute**. In our example, if you input a number outside the range of 1 to 7, it will print "Invalid."

Key Considerations for Switch Case Statements:

- **Requirement for a Constant Expression**
  A switch statement necessitates that its expression results in a constant value. This can include constants and arithmetic operations.

Code:

```
#include <iostream>

using namespace std;


int main() {

    const int x = 10;

    const int y = 5;


    switch (x + y) {

        case 15:

            cout << "Result is 15." << endl;

            break;

        case 20:

            cout << "Result is 20." << endl;

            break;

        default:

            cout << "No match found." << endl;

    }


    return 0;

}
```

Output: Result is 15.

- **Limited to Integer or Character Types**
  Switch statements are exclusively designed to handle integer or

character values. Ensure that the expression provides values of type int or char.

Code:

```
#include <iostream>

using namespace std;


int main() {

    char grade = 'B';


    switch (grade) {

        case 'A':

            cout << "Excellent!" << endl;

            break;

        case 'B':

            cout << "Good!" << endl;

            break;

        default:

            cout << "Not specified." << endl;

    }


    return 0;

}
```

Output: Good!

- **Significance of the 'Break' Keyword**
  The 'break' keyword holds significant importance within switch cases. It serves as an exit mechanism from the switch statement. Its omission implies the execution of all subsequent cases.
- **Optional Default Case**
  Optionally, you can include a 'default' case, which executes when none of the case values match. It's not obligatory and can be excluded if not needed.

3. **Prohibition of Duplicate Case Values**
   Within a switch statement in C++, duplicates of case values are disallowed. Each case value must be distinct.

Code:

```
#include <iostream>
```

```
using namespace std;

int main() {
    int day = 2;

    switch (day) {
        case 1:
            cout << "Monday." << endl;
            break;
        case 2:
            cout << "Tuesday." << endl;
            break;
        case 2:  // Duplicate case, not allowed
            cout << "Duplicate case." << endl;
            break;
        default:
            cout << "Invalid day." << endl;
    }
    return 0;
}
```

- **Potential for Nested Switch Statements**
  While it's possible to nest one switch statement inside another in C++, this practice is generally discouraged due to its potential to introduce complexity and hinder code readability.

Code:

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

int main() {
    int x = 2;
    int y = 3;

    switch (x) {
        case 1:
```

```
        cout << "x is 1." << endl;
        switch (y) {
            case 1:
                cout << "y is 1." << endl;
                break;
            default:
                cout << "y is not 1." << endl;
```

L-1.5 What are arrays, strings?

An array is a linear data structure in which we store data and perform any operation, we can randomly access data in an array (With the help of its *index values).*

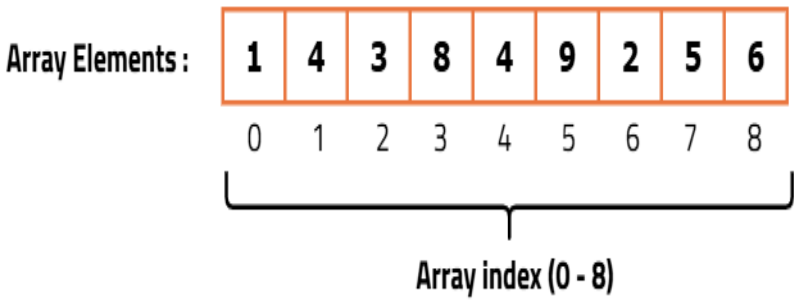
In other words, an array is a collection of similar types of elements (Homogeneous elements) that have contiguous memory locations i.e One after another.

Arrays store the related information in adjacent memory blocks.

If you need to access one or more than one element, the process of finding or performing operations on that element becomes very fast because your computer very well knows where the value is located in the memory.

Let's Visualize arrays..!

To understand how the array works, Visualize your computer's memory as a continuous block or grid. Each piece of block or grid contains information/data that is stored in it.



Why are Arrays '0' indexed?

Counting arrays from the index value 0 simplifies the computation for the memory. Though it simplifies the computation. But it adds an extra step of an unnecessary subtraction of 1 i.e (n-1) for each access.

The array of length n can be indexed by the integers from 0 to n-1.

Most programming languages have been designed in this way only, **so indexing from 0** is pretty much inherent to the language.

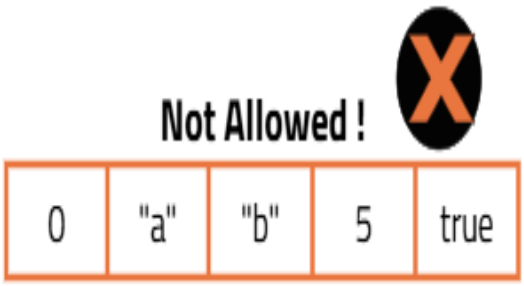
Defining an array:

As we already specified, the array contains homogeneous data so the above one is an integer array as it contains integer data.

Arrays can store numbers, strings, boolean values, characters, objects, etc. But once you define the type of values that your array will store, all its elements must be of that same type. You cannot "insert" different types of data in a single array.

Arrays cannot store heterogeneous data, Now what it means.

Let's understand it through some visuals.



Creating an array :

Assign it to a variable *(Name of an array)*.

Define the type of elements that it will store *(integer, string, boolean)*.

Define its size *(the maximum number of elements it will store)*.

Syntax : Data_type array_name [Array_size] ;

E.g: **int myarray [8];**

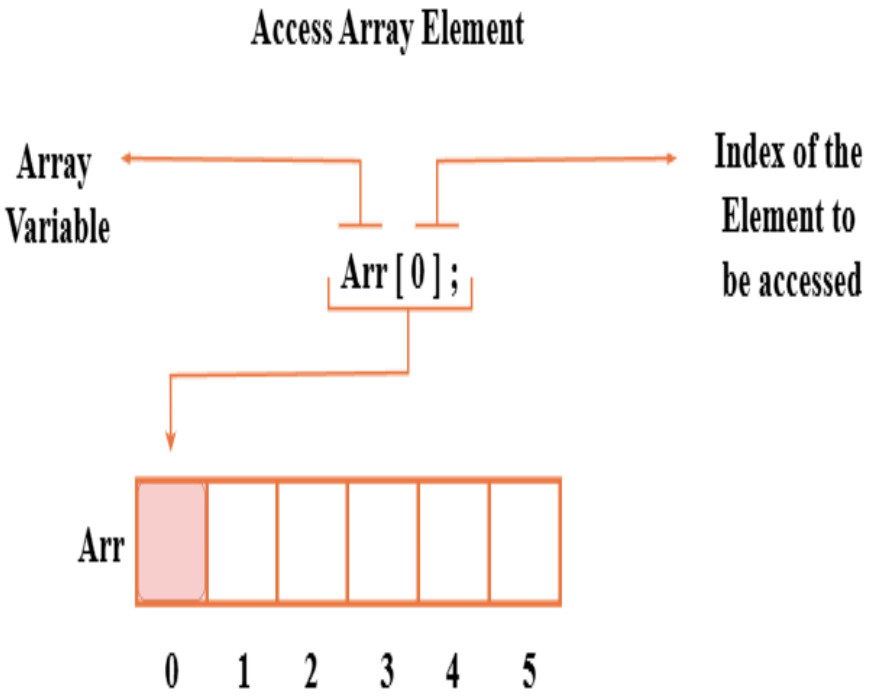
- Int => Datatype.
- myArray => Name of an array.
- [8] => Size of an array.

When you define the size of the array, all of that space in memory is **"reserved"**

Finding an Element in an array

You have three options to find an element in an array:

- **If we know where the element is located,** We can use the index values to access it.
- **If you don't know where it's located,** you can use algorithms to optimize your search, such as Linear, Binary Search, etc.
- If you need faster repeated lookups, you can store the array elements in a hash-based data structure (like a hash set or hash map).



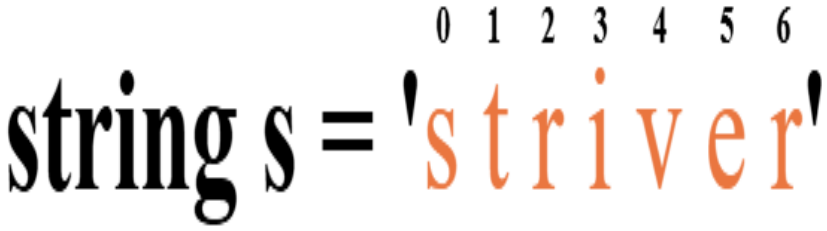
Summary of the Arrays:

- **Memory is allocated instantly:** After the array is created the memory is allocated instantly and the array is empty until you assign the values.
- **Elements are located in contiguous manner:** So the elements can be accessed very efficiently (direct access, O(1) = constant time) using it's index values.
- **Arrays are powerful data structures:** The type of elements and the size of the array are fixed and defined when you create it. Arrays store elements of the same type (homogeneous).
- **Inserting elements:** Reach to the end of the array and insert the element. The insertion at the end of the array. Insertion takes constant time O(1). We can also insert the elements in the middle or the start of the array but it's a bit of a complex process.
- **Removing elements:** Reach to the index and remove the element. The removal operation takes constant time O(1).

What are Strings?

Strings are like a **series of characters** stored in a **specific order**. Each character in a string is assigned an index, starting from 0. This means the first character is at index 0, the second character at index 1, and so on.

Let's take an example. Suppose we have a string s containing the word "striver." If we want to access the character 's', we can do so by using s[0], which corresponds to the first character. Similarly, s[1] will give us 't', and s[2] will give us 'r'. This **zero-based indexing** is a fundamental concept in working with strings.



Finding the Length of a String

To determine the length of a string, we can use the **size** or **length function**. These functions return the number of characters in the string. For example, if we have s as our string, you can find its length like this:

Code

```
#include <bits/stdc++.h>

using namespace std;


// Class containing the method to find string length

class Solution {

public:

    // Function to return length of a string

    int findLength(string s) {

        // Return length using built-in function

        return s.length();

    }

};


// Driver code

int main() {

    // Create object of Solution class
```

```
    Solution obj;
    // Input string
    string s = "Hello World";
    // Call function and print result
    cout << obj.findLength(s) << endl;
    return 0;
}
```

Accessing Individual Characters

You can access individual characters within a string using **square brackets**, just like an array. Strings also follow 0-based indexing. For instance:

Code

```
#include <bits/stdc++.h>
using namespace std;

// Class containing the method to access characters
class Solution {
public:
    // Function to print each character of a string
    void accessCharacters(string s) {
        // Loop through each index
        for (int i = 0; i < s.length(); i++) {
            // Print the character at index i
            cout << s[i] << endl;
        }
    }
};

// Driver code
int main() {
    // Create object of Solution class
    Solution obj;
    // Input string
    string s = "Hello";
```

```
    // Call the function
    obj.accessCharacters(s);
    return 0;
}
```

To achieve the functionality of concatenation of strings in Java, we use the StringBuilder class.

Passing, Returning, and Assigning Strings

Strings in C++ can be **assigned** and **passed** like **primitive type**s. Assigning one string to another makes a **deep copy** of the character sequence:

In C++, strings can be **seamlessly passed between functions**. When you pass a string as an **argument** to a function, you're essentially making a copy of the string. Any changes made to the string within the function won't affect the original string outside of it.

Copying strings is not merely a superficial process, it involves creating a new string with an identical character sequence. Whether you're assigning a string to another or passing it to a function, you're essentially **creating a fresh copy**.

Code

```
#include <bits/stdc++.h>
using namespace std;

// Solution class containing modifyString function
class Solution {
public:
    // Function to modify the string
    string modifyString(string str) {
        // Assign str to a new variable
        string newStr = str;

        // Modify the new string
        newStr[0] = 'H';

        // Return the modified string
        return newStr;
    }
};
```

```
int main() {
    // Original string
    string original = "hello";

    // Create object of Solution class
    Solution sol;

    // Call modifyString and store the result
    string modified = sol.modifyString(original);

    // Print both strings
    cout << "Original String: " << original << endl;
    cout << "Modified String: " << modified << endl;

    return 0;
}
```

String Comparison

The == known as the **equality operator** is used for comparing two values to check if they are equal. In programming, it's commonly used to **compare variables**, such as numbers or strings, to determine if they have the same value. For example, x == y will **return true** if x is equal to y, and false otherwise. The != known as the **inequality operator** is used to check if two **values are not equal**. It's the opposite of the equality operator. If the values being compared are not equal, != returns true; if they are equal, it returns false. We can check if two strings are equal or not and at the same time we can also check whether particular characters of two strings are equal or not.

Code

```
#include <bits/stdc++.h>
using namespace std;

// Class containing the compareStrings function
class Solution {
public:
    // Function to compare two strings
```

```
    bool compareStrings(string str1, string str2) {
        // Return true if strings are equal
        return str1 == str2;
    }
};

// Main function
int main() {
    // Create object of Solution class
    Solution obj;

    // Input first string
    string str1;
    cin >> str1;

    // Input second string
    string str2;
    cin >> str2;

    // Compare strings and print result
    if (obj.compareStrings(str1, str2))
        cout << "Strings are equal";
    else
        cout << "Strings are not equal";

    return 0;
}
```

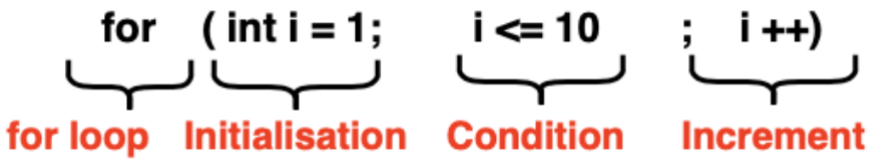
L-1.6 For Loops

What is a For Loop and Why is it Used?

A for loop is a **control structure** in programming that allows you to **execute a specific block of code repeatedly.** It's especially useful when you want to **perform the same task multiple times without duplicating your code.** Let's break down the essential components of a for loop:

1. **Initialization:** You declare and initialize a variable that serves as a counter. This step only happens once at the beginning.
2. **Condition:** You specify a condition that determines when the loop should stop executing.

Increment/Decrement: You define how the counter variable changes after each iteration.



Code:

C++Java

```
#include <iostream>

using namespace std;


int main() {
    for (int i = 1; i <= 10; i++) {
        cout << "Hey, Striver, this is the " << i << "'th iteration" << endl;
    }
    return 0;
}
```

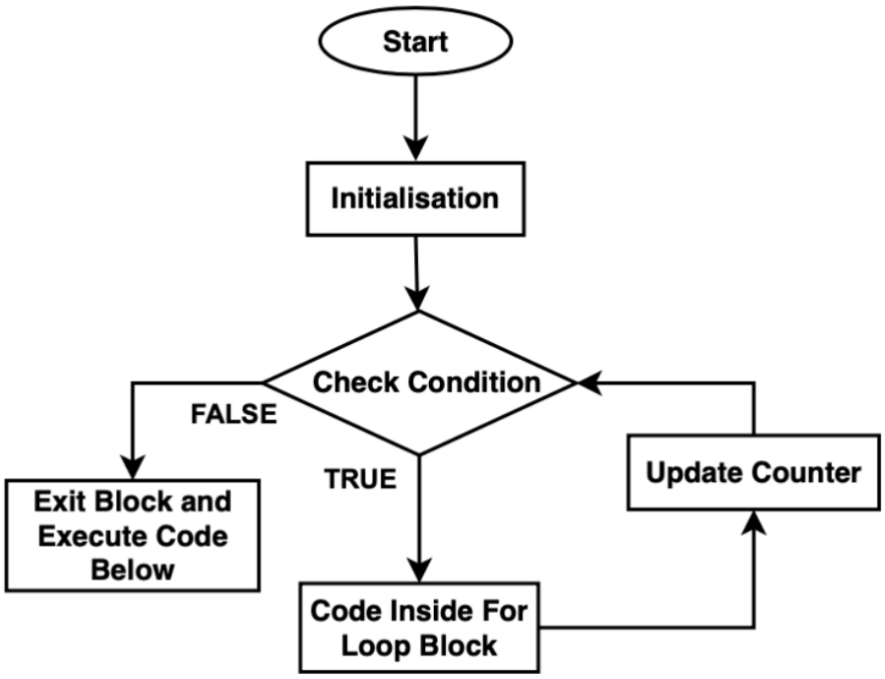
Output:

```
Hey, Striver, this is the 1'th iteration
Hey, Striver, this is the 2'th iteration
Hey, Striver, this is the 3'th iteration
Hey, Striver, this is the 4'th iteration
Hey, Striver, this is the 5'th iteration
Hey, Striver, this is the 6'th iteration
Hey, Striver, this is the 7'th iteration
Hey, Striver, this is the 8'th iteration
Hey, Striver, this is the 9'th iteration
Hey, Striver, this is the 10'th iteration
```

In this example, the loop will run ten times because it starts with i equal to 1, and the condition is met until i becomes 11 then the loop breaks. The variable i is incremented by 1 in each iteration.

```
for (int i = 1; i <= 10; i++) {
    cout << "Hey, Striver, this is the " << i << "'th iteration" << endl;
}
```

i initialised at i = 1	condition i <= 10 check	printed at terminal	increment
i = 1	true	Hey, Striver, this is the 1'st iteration	i = 1 + 1 = 2
i = 2	true	Hey, Striver, this is the 2'nd iteration	i = 2 + 1 = 3
i = 3	true	Hey, Striver, this is the 3'rd iteration	i = 3 + 1 = 4
		... and so on	
i = 9	true	Hey, Striver, this is the 9'th iteration	i = 9 + 1 = 10
i = 10	true	Hey, Striver, this is the 10'th iteration	i = 10 + 1 = 11
i = 11	false	FOR LOOP BREAKS AS CONDITION FAILS	

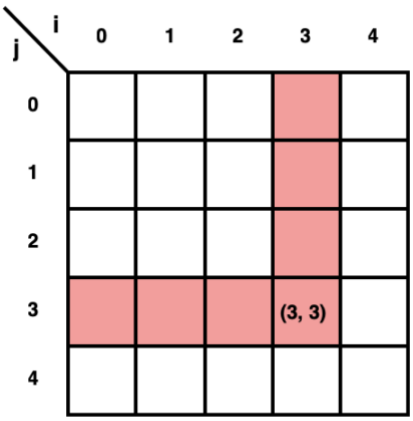


1. **Start:** The process begins at the "start" point, indicating the beginning of the loop.
2. **Initialize counter:** In this step, the loop control variable is initialised. This typically involves assigning an initial value to the counter. Usually denoted by the variable 'i'
3. **Condition check:** The condition for the loop is checked in this stage. If the condition is evaluated as "true," the loop continues to execute; otherwise, it terminates.
4. If the **condition is "true,"** the loop proceeds to execute the loop body and increment/decrement the counter.

5. After completing an **iteration**, the process returns to the "Condition check" step to re-evaluate the condition.
6. After executing the loop body, the **counter** is incremented or decremented. This step is crucial for **controlling the loop's termination.**
7. The loop continues to execute as long as the **condition remains "true."** Once the condition becomes "false," the loop exits.
8. After **exit from the loop**, the code below the for-loop is executed and the program moves on.

Nested For Loops

Just like for loops, you can nest one for loop inside another. This concept becomes incredibly useful when you're working with **multi-dimensional data structures** like a 2-D Array or need to solve complex problems involving **multiple iterations.**



Code:

C++Java

```
#include <iostream>

using namespace std;


int main() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << "i = " << i << ", j = " << j << endl;
            // Nested loop code
        }
    }
    return 0;
}
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
i = 2, j = 0
i = 2, j = 1
i = 2, j = 2
```

Conditionals Inside For Loops

For loops are versatile; you can include **conditional statements** (if, else if, else) within them. This allows you to execute **different code blocks based on certain conditions** during each iteration.

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        // Code for even numbers
    } else {
        // Code for odd numbers
    }
}
```

Customising For Loops

You have flexibility in how you structure your for loop. For instance, you can customise the **increment step to achieve specific patterns** or **iterate a certain number of times.**

Code:

C++Java

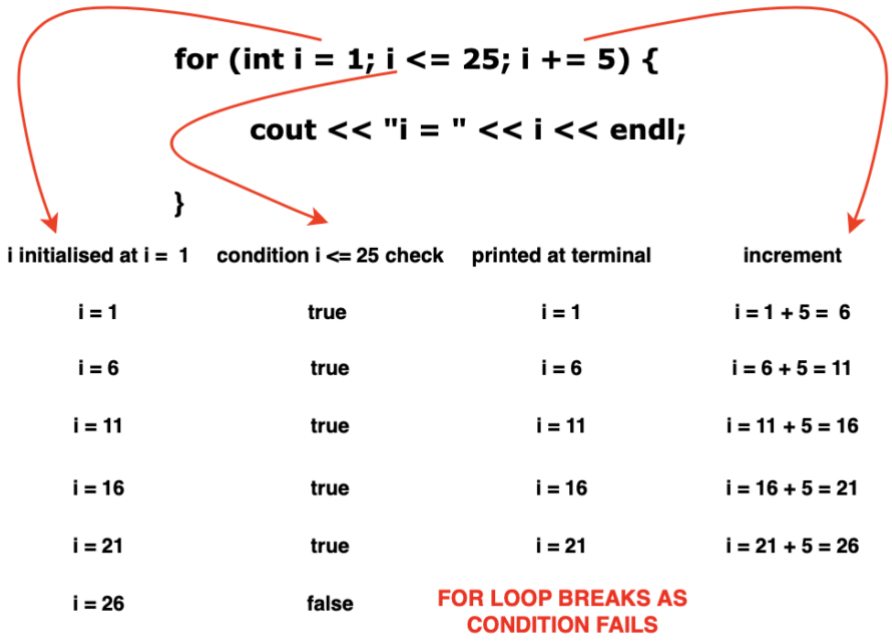
```
#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 25; i += 5) {
        cout << "i = " << i << endl;
    }
    return 0;
}
```

Output:

```
i = 1
i = 6
i = 11
```

```
i = 16
i = 21
```

In this example, i starts at 1 and increases by 5 in each iteration, resulting in five iterations.



L-1.7 While Loops

In the world of computer programming, loops are invaluable tools that allow us to execute a block of **code repeatedly** until a **certain condition is met**. One such loop is the "while" loop.

A while loop follows a simple sequence of steps:

1. **Evaluation of Test Expression:** The loop begins by evaluating a test expression.
2. **Condition Check:** If the test expression is true, the code inside the loop's body is executed.
3. **Re-evaluation:** After executing the code, the test expression is evaluated again.

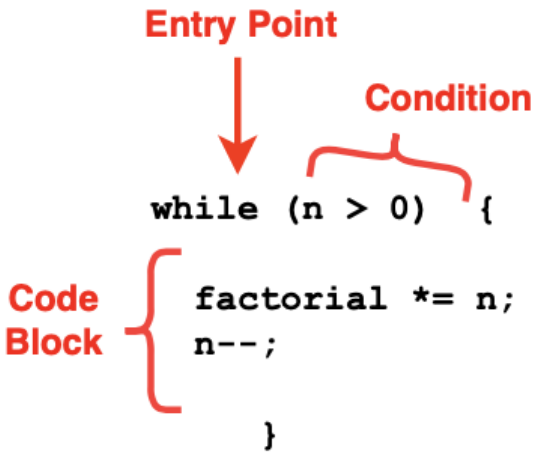
Continuation or Termination: This process continues until the test expression becomes false, at which point the while loop terminates.

Syntax

```
while (condition) {

    // body of the loop

}
```

A **while loop** is a control structure in programming that **repeatedly executes a block** of code as long as a specified **condition remains true**. A "while" loop

will not execute the code at all if the **initial condition** is false. It relies on the condition being true from the outset for any code execution to occur.



While loops can be best illustrated with the practical example of finding the **Factorial**. Factorial of a number 'n' is the product of all positive integers from 1 to 'n'. To compute this using a while loop, we initialise a **factorial variable to 1** and repeatedly **multiply it by 'n'** while **decrementing 'n' until 'n' becomes 0**. This ensures that we calculate the factorial correctly.

Code:

```
#include <iostream>

using namespace std;


int main() {
    int n = 5;
    int factorial = 1;


    while (n > 0) {
        factorial *= n;
        n--;
    }


    cout << "Factorial of 5 is: " << factorial << endl;


    return 0;
}
```

Output:

```
Factorial of 5 is: 120
```

While loops are particularly useful when you need to ensure that a block of code executes **only when the condition is satisfied** as it terminates as soon as that **condition becomes false**. This can be vital for tasks like **validating user input** or **processing data until a specific condition is met**. By checking the condition at the beginning of the loop, you can control whether the loop body is executed or not.

Termination Conditions:

In algorithm design, it's essential to define clear and well-defined termination conditions for while loops. The termination condition specifies when the loop should stop executing. Without proper termination conditions, a while loop can run indefinitely, leading to what's known as an "infinite loop." Infinite loops can crash programs and consume excessive system resources, making them a critical issue to avoid.

It's crucial to define clear termination conditions for while loops in algorithms to prevent infinite loops. Termination conditions ensure that the loop will eventually exit, making the algorithm correct and efficient.

Optimisation:

Optimising while loops involves making them more efficient by minimising unnecessary iterations. One common optimization technique is to use loop control statements like break or continue within the loop body.

break: It allows you to exit the loop prematurely, even before the termination condition is met. For example, if you're searching for a value in an array, once you find it, you can break out of the loop instead of continuing to search through the remaining elements.

continue: It skips the current iteration of the loop and moves to the next one. This can be useful when you want to skip certain elements or avoid executing some code under specific conditions.

By using these control statements judiciously, you can reduce the number of iterations, which can significantly improve the efficiency of your algorithm.

Code:

```
#include <iostream>

using namespace std;


int main() {
    int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int target = 6;

    // Using 'break' to exit the loop early when the target is found
    for (int num : numbers) {
        if (num == target) {
            cout << "Target found: " << target << endl;
            break; // Exit the loop immediately
```

```
        }
        cout << "Checking: " << num << endl;
    }

    // Using 'continue' to skip specific elements
    for (int num : numbers) {
        if (num % 2 == 0) {
            continue; // Skip even numbers
        }
        cout << "Odd number: " << num << endl;
    }

    return 0;
}
```

Output:

```
Checking: 1
Checking: 2
Checking: 3
Checking: 4
Checking: 5
Target found: 6
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

- The first loop uses **break** to **exit the loop immediately** when the target value is found, **preventing unnecessary iterations.**
- The second loop uses **continue** to **skip even numbers**, so only odd numbers are printed, **avoiding the execution of code** for even numbers.

L-1.8 Functions (Passed by reference) in

In computer programming, functions are one of the most fundamental building blocks. They allow us to organize and modularize code, making programs more readable, reusable, and efficient. A function is essentially a block of code designed to perform a specific task, and it can be called multiple times within a program.

A function in C++ follows a simple sequence of steps:

1. **Declaration**: The function is declared with its return type, name, and parameters.

2. **Definition**: The function body is written, which contains the actual code to be executed.
3. **Calling**: The function is invoked from the main() function or another function in the program.

Syntax

```
returnType functionName(parameters) {
    // body of the function
    return value; // optional, only if returnType is not void
}
```

Functions improve program structure by allowing code to be divided into smaller, manageable sections. This also makes debugging and testing easier.

Types of Functions

1. **Built-in Functions**: Provided by C++ libraries (e.g., sqrt(), pow(), cin, cout).
2. **User-defined Functions**: Created by the programmer to solve specific problems.

Functions can be further categorized based on parameter passing:

- **Pass by Value**: A copy of the variable is passed, so changes inside the function do not affect the original variable.
- **Pass by Reference**: The reference (alias) of the variable is passed, so changes affect the original variable.
- **Pass by Pointer**: The memory address of the variable is passed, and the function works directly with the variable in memory.

Practical Example: Addition, Increment, and Swap

Consider the following example where we use different types of functions to add two numbers, increment a number, and swap two numbers.

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

// Function Declaration
int add(int a, int b);        // Adds two numbers (Pass by Value)
void increment(int& x);       // Increments number (Pass by Reference)
void swap(int& a, int& b);    // Swaps two numbers (Pass by Reference)

// Function Definitions
int add(int a, int b) {
```

```
    return a + b;
}

void increment(int& x) {
    x++;
}

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int num1 = 5, num2 = 10;

    // Function call - Pass by Value
    cout << "Sum of " << num1 << " and " << num2 << " = " << add(num1, num2) << endl;

    // Function call - Pass by Reference (increment)
    cout << "Before increment: num1 = " << num1 << endl;
    increment(num1);
    cout << "After increment: num1 = " << num1 << endl;

    // Function call - Pass by Reference (swap)
    cout << "Before swap: num1 = " << num1 << ", num2 = " << num2 << endl;
    swap(num1, num2);
    cout << "After swap: num1 = " << num1 << ", num2 = " << num2 << endl;

    return 0;
}
```

Output

```
Sum of 5 and 10 = 15
```

```
Before increment: num1 = 5

After increment: num1 = 6

Before swap: num1 = 6, num2 = 10

After swap: num1 = 10, num2 = 6
```

Function Termination

Every function in C++ either returns a value of its declared type or terminates without returning anything if it is declared with a void return type. A return statement specifies the value to be returned and also marks the end of function execution. If no return value is required, the function simply completes its task and returns control to the caller.

Optimisation

Optimisation of functions involves making them more efficient and reusable. Some techniques include:

- **Function Overloading**: Defining multiple functions with the same name but different parameter lists to handle different types of inputs.
- **Inline Functions**: Using the inline keyword to reduce the overhead of function calls for small, frequently used functions.
- **Default Arguments**: Providing default values for parameters so that the function can be called with fewer arguments.

Functions in C++ are vital for creating structured, maintainable, and efficient programs. By defining clear function responsibilities and using appropriate parameter passing methods, programmers can build powerful applications while keeping the code simple and organized.

L-1.9 Time Complexity and Space Complexity

What is Time Complexity?

We can solve a problem using different logic and different codes. Time complexity basically helps to judge different codes and also helps to decide which code is better. In an interview, an interviewer generally judges a code by its time complexity.

Now, the term, time complexity, seems that it is referring to the time taken by a machine to execute a particular code. But in real life, ***Time complexity does not refer to the time taken by the machine to execute a particular code***.

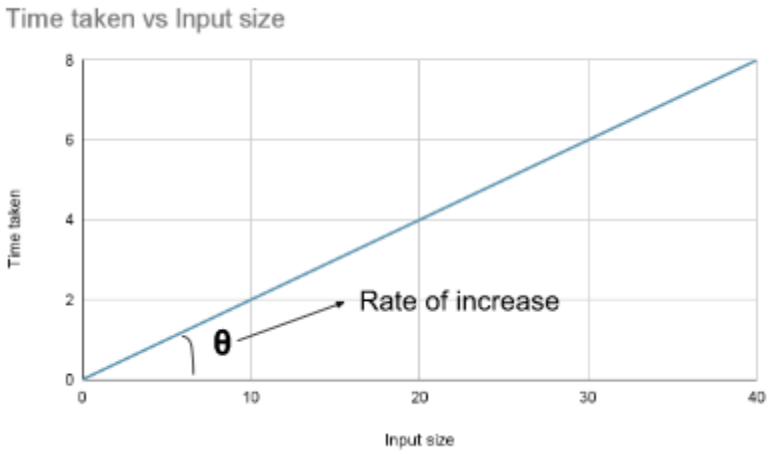
Let's understand why we should not judge any code on the basis of the time taken by a machine.

If we run the same code in a low-end machine(e.g. old windows machine) and in a high-end machine(e.g. Latest MacBook), we will observe that two different machines take different amounts of time for the same code. The high-end machine will take lesser time as compared to the low-end machine.

So, the time taken by a machine can be changed depending on the configuration. ***That is why we should not compare the two different codes on the basis of the time taken by a machine as the time is dependent on it.***

Definition:

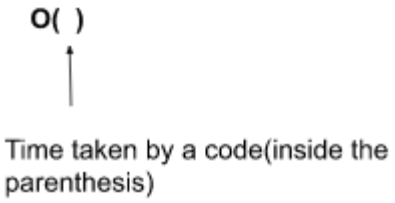
The rate at which the time, required to run a code, changes with respect to the input size, is considered the time complexity. **Basically, the time complexity of a particular code depends on the given input size, not on the machine used to run the code.**
Let's understand this using the following diagram:



Now, the next question that comes to our mind is how we will represent the time complexity of a code as we are not going to use the standard units like minutes or seconds. Let's discuss it below:

How we will represent the time complexity of any code:

To represent the time complexity, we generally use the Big O notation. The Big O notation looks like the following:



Let's understand this using the following example:

Given Code:

```
for(int i = 1; i <= 5; i++){
    cout << "Raj";
}
```

The time complexity for this code will be nothing but the number of steps, this code will take to be executed. So, if we write this in terms of Big O notation, it will be like O(no. of steps).

Let's observe the steps for this code:

1. First, the assigning step(i = 1) will be done.
2. The second step will be the comparison i.e. i <= 5.
3. The third step will be the print statement (i.e. cout << "Raj";).
4. The fourth step will be the increment(i.e. i++).
5. In the fifth step, the updated value of i will be again checked i.e. the comparison(i <= 5).
6. In the sixth step, the print statement will be executed and so on.

This flow will continue until the value of i becomes greater than 5(i.e. 6). In a broader sense, we can observe that the 'for loop' will run 5 times and for each time three steps will be surely executed i.e. checking/comparison, printing, and increment. So, the total steps will be 5*3 = 15. And the time complexity in terms of Big O notation will be O(15).

Now, if we write N instead of 5, the number of steps will be then N*3 = 3N and the time complexity will be O(3*N).

But this manual counting process is not feasible for any code. As the 'for loop' might run a billion or million times and inside that 'for loop', there might be a large no. of operations or some other 'for loops' as well. So, we have to find out a better approach to calculate the time complexity of any given code.

Here come the three rules, that we are going to follow while calculating the time complexity:

1. We will always ***calculate the time complexity for the worst-case scenario***.
2. We will ***avoid including the constant terms***.
3. We will also ***avoid the lower values.***

Let's discuss the rules individually:

1. **Calculate the time complexity for the worst-case scenario:**

Before discussing the point we need to understand the three terms i.e. Best Case, Worst Case, and Average Case.

Let's understand these three terms considering the following piece of code:

```
if(marks < 25)      cout << "grade D";
else if(marks < 45) cout << "grade C";
else if(marks < 65) cout << "grade B";
else                cout << "grade A";
```

1. **Best Case:** This term refers to the case where the code takes the least amount of time to get executed. For example, if the mark is 10(i.e. < 25), only the first line will be executed and the rest of the lines will be skipped. So, the least amount of steps i.e. only 2 steps are required in this case. This is an example of the best case.
2. **Worst Case:** This term refers to the case where the code takes the maximum amount of time to get executed. For example, if the mark is 70(i.e. >

65), the last line will be executed after checking all the above conditions. So, the maximum amount of steps i.e. 4 steps are required in this case. This is an example of the worst case.

3. **Average Case:** This term is pretty self-explanatory. This is basically the case between the best and the worst.

Now, as we always want that our system serves the maximum number of clients, we need to calculate the time complexity for the worst-case scenario. With this, we can actually judge the robustness of any code or any system.

2. Avoid including the constant terms:

Let's understand this rule considering the time complexity: $O(4N^3 + 3N^2 + 8)$. Now, if we consider the value of N as 10^5 the time complexity will be like this: $O(4*10^{15} + 3*10^{10} + 8)$. In this case, the constant term 8 is very less significant in terms of changing the time complexity with different values of N. That is why we should avoid the constant terms while calculating the time complexity.

If we want to think of this case in terms of code, we can consider the following code:

```
int x = 2;
for(int i = 1; i <= N; i++){
    cout << "raj";
}
```

Here, the first step (i.e. int x = 2) will be executed in unit time i.e. constant time. The precise time complexity is O(3N + 1) but in this case, the constant 1 is very less significant. So we will write the time complexity as O(3N) avoiding the constant term.

3. Avoid the lower values:

Now, in the previous example, the given time complexity is $O(4N^3 + 3N^2 + 8)$ and we have reduced it to $O(4N^3 + 3N^2)$. Here, we can clearly observe if the value of N is a large number, the second term i.e. $3N^2$ will also be a less significant term. For example, if the value of N is 10^5 then the term $3*10^{10}$ becomes less significant with respect to $4*10^{15}$. So, we can also avoid the lower values and the final time complexity will be $O(4N^3)$

Note: *A point to remember is that we can actually ignore the constant coefficients as well. For example, considering the time complexity $O(4N^3)$ as $O(N^3)$ is also correct.*

Apart from the widely used Big O notation, there are several other notations. Among them, the two most common are the Theta notation(θ) and the Omega notation(Ω). The differences are shown in the below table:

Big O notation	Theta notation(θ)	Omega notation(Ω)
Represents the worst-case time complexity i.e. the upper bound.	Represents the average-case time complexity.	Represents the best-case time complexity i.e. the lower bound.

These concepts are not much important from the interview perspective and so here we are not going to discuss these in detail. Please follow any standard textbook if you want the details and the mathematical derivations.

Let's quickly discuss some questions to make the concepts clear:

Question 1:

Given the following code:

```
for(int i = 0; i < N; i++){
    for(int j = 0; j < N; j++){

        // Block of code
        // that runs in constant time.
    }
}
```

In order to calculate the time complexity of the code, we need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1 i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to N-1 i.e. N times. The following illustration depicts the process:

```
for i = 0, j = [0, 1, 2, 3, ....., N-1]
for i = 1, j = [0, 1, 2, 3, ....., N-1]
for i = 2, j = [0, 1, 2, 3, ....., N-1]
for i = 3, j = [0, 1, 2, 3, ....., N-1]
for i = 4, j = [0, 1, 2, 3, ....., N-1]
for i = 5, j = [0, 1, 2, 3, ....., N-1]
.
.
.
for i = N-1, j = [0, 1, 2, 3, ....., N-1]
```

Now, we can clearly observe the total number of steps i.e. N+N+N+N+.....+N times = N*N = N^2. So, **the time complexity will be $O(N^2)$**. We can ignore other constant steps as well as the innermost block of code as it runs in constant time.

Question 2:

Given the following code:

```
for(int i = 0; i < N; i++){
    for(int j = 0; j <= i; j++){

        // Block of code
        // that runs in constant time.
    }
}
```

In order to calculate the time complexity of the code, we again need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1

i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to i i.e. (i+1) times. The following illustration depicts the process:

```
for i = 0, j = [0] // 1 time
for i = 1, j = [0, 1] // 2 times
for i = 2, j = [0, 1, 2] // 3 times
for i = 3, j = [0, 1, 2, 3] // 4 times
for i = 4, j = [0, 1, 2, 3, 4] // 5 times
for i = 5, j = [0, 1, 2, 3, 4, 5] // 6 times
.
.
.
for i = N-1, j = [0, 1, 2, 3, ……, N-1] // N times
```

Now, we can clearly observe the total number of steps i.e. 1+2+3+4+.....+N. Now we know the formula of the summation of the first N natural numbers i.e. $(N*(N+1))/2 = N^2/2 + N/2$. So, the precise time complexity will be $O(N^2/2 + N/2)$. Now, we should ignore the lower values. So, **the time complexity will be $O(N^2/2)$**. It can be also written as **$O(N^2)$** avoiding the coefficient 1/2.

These are the basics of time complexity. Now, let's move on to the space complexity part.

What is Space Complexity?

The term space complexity generally refers to the memory space that a code uses while being executed. Again space complexity is also dependent on the machine and so we are going to **represent the space complexity using the Big O notation** instead of using the standard units of memory like MB, GB, etc.

Definition:

Space complexity generally represents the summation of auxiliary space and the input space. Auxiliary space refers to the space that we use additionally to solve a problem. And input space refers to the space that we use to store the inputs.

Let's understand this using the following example:

```
Input(a)//1 Input space
Input(b)//1 Input space
c = a+b
//c-> 1 auxiliary space
```

The variables a and b are used to store the inputs but c refers to the space we are using to solve the problem and c is the auxiliary space. Here the space complexity will be O(3).

Similarly, if we use an array of size n, the space complexity will be O(N).

Good coding practice:

If a question of adding two numbers like a and b is asked, one of the possible methods will be

b = a+b. In this case, the space complexity is definitely reduced as we are not using any extra variable but this is not a good practice to manipulate the given inputs or data. In an interview, we must be careful that we will not manipulate the given data even if the space complexity becomes 2N instead of N. If the interviewer specifically instructs us to manipulate, then only we should attempt this method.

Note: *A company may use the same data for different purposes. That is why we should not attempt to manipulate the given data for reducing the space complexity. So, we will never manipulate the given data i.e. the inputs until the interviewer specifically says so.*

We are now pretty much done with our concepts of time complexity and space complexity. Now, we will briefly discuss some points about competitive programming or the online judge.

Points to remember:

In competitive programming or in the platforms like Leetcode and GeeksforGeeks, we generally run our codes on online servers. Most of these servers execute roughly 10^8 operations in approximately 1 second i.e. 1s. We must be careful that if the time limit is given as 2s the operations in our code must be roughly $2*10^8$, not 10^{16}. Similarly, 5s refers to $5*10^8$. Simply, if we want our code to be run in 1s, the time complexity of our code must be around $O(10^8)$ avoiding the constants and the lower values.