**Examination Answer Book**

**UNIVERSITY EXAMS**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| REGISTRATION NUMBER | | | | | | | | | VU-BIT-2403-1462-EVE | | | | | | |
| Title of The Program (eg BBA, BSC, BPH, BSWA) | | | | | | | | | | | | | BIT | | |
| Bachelor of Information Technology | | | | | | | | | | | | | | | |
| Department | | | | Other Depts in Faculty of Science and Technology | | | | | | | | | | | |
| Faculty | Faculty of Science and Technology | | | | | | | | | | | | | | |
| Year Of study (YrI , YrII, YrIII, or YrIV) | | | | | | | | | | | 1 | | | | |
| Module Code and Name | | | | | | | 1203 ST | | | | | | | | |
| Programming Fundamentals | | | | | | | | | | | | | | | |
| Semester | | | 2 | | | | | | | | | | | | |
| (Copy from the heading to the Examination Paper) | | | | | | | | | | | | | | | |
| Retake: | | Yes | | |  | | | No | |  | | (Tick whichever is applicable) | | | |
| Date of examination | | | | | | Sun Sep 22 2024 08:00:00 GMT+0300 (East Africa Time) | | | | | | | | | |
|  | | | | | |  | | | | | | | | | |
| **DIRECTIONS TO CANDIDATES (Turn to page ii for more instructions).** | | | | | | | | | | | | | **FOR USE BY EXAMINERS ONLY** | | |
| **Question Number** | **Internal Examiner** | **External Examiner** |
| 1. Leave margin blank. 2. Begin each answer on a fresh page. 3. Write the number of each question and theCandidate's Number at the top of each page. 4. Write the numbers of the questionswhich you have attempted, with subsections where necessary, in the spacesprovided below | | | | | | | | | | | | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **TOTAL** |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NUMBER OF QUESTIONS** you have answered in the order in which you have written them | | | | | | | | |
| QN.1 | QN.2 | QN.4 | QN.5 |  |  |  |  |  |

**How and where should I submit my examination script?**

Every student will be required to attend their examination via [VClass Students Portal](https://vclass.ac/) E.g. you go to [www.vclass.ac](http://www.vclass.ac) and login, to your account, then on the left sidebar menu **click on Examinations**.

Under examinations you will see the following: -

1. Instructions for that particular examination with time required to finish your examination as per instructions,
2. A student will be required to download the question paper and the answer sheet provided by the university within the same module examination, or a student can be required to attempt structured questions within the system depending on how the examination was set.
3. Submission of answered questions is done,
4. Student is required to click to **consent** to show that the answered exam belongs to them.
5. **Note** that if an examination is for download, a student will be required to download the question paper and answer sheet, write their examination within the given stipulated time.
6. Required to scan and upload back the answered booklet through the same portal as per format available.
7. Examinations uploaded will directly be received by the Registry department.
8. Students here are required to use [VClass e-Learning system](https://vclass.ac)for all examinations and for any failure they can contact the Registry department for guidance.
9. No late submission will be accepted.

**Avoid any examination malpractice because this will attract severe penalties such as invalidating the exams answered script whose consequences will attract retakes.**

**QUESTION ONE (A)**

A leap year is an essential concept in the Gregorian calendar system, designed to keep our calendar year aligned with the Earth's revolutions around the Sun. By adding an extra day to the calendar every four years, we can account for the additional 0.25 days in each year that aren't covered by the usual 365 days. This adjustment prevents our calendar from drifting over time and ensures that seasonal events occur at the same time each year.

**Below is the source code.**

#include <iostream>

using namespace std;

bool is\_leap\_year (unsigned int Y) {

// Check if the year is a century year

if (Y % 100 == 0) {

// Century year is a leap year only if it is divisible by 400

return (Y % 400 == 0);

}

// Non-century year is a leap year if it is divisible by 4

return (Y % 4 == 0);

}

int main() {

unsigned int year;

cout << "Enter a year: ";

cin >> year;

if (is\_leap\_year(year)) {

cout << year << " is a leap year." << endl;

} else {

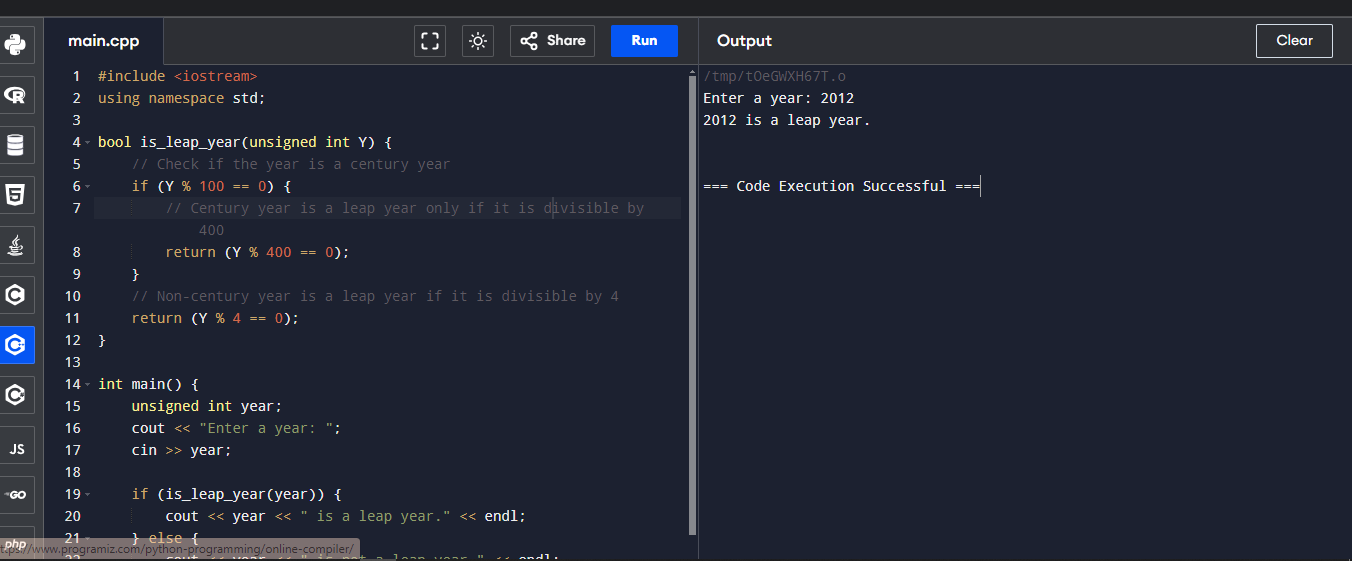
cout << year << " is not a leap year." << endl;

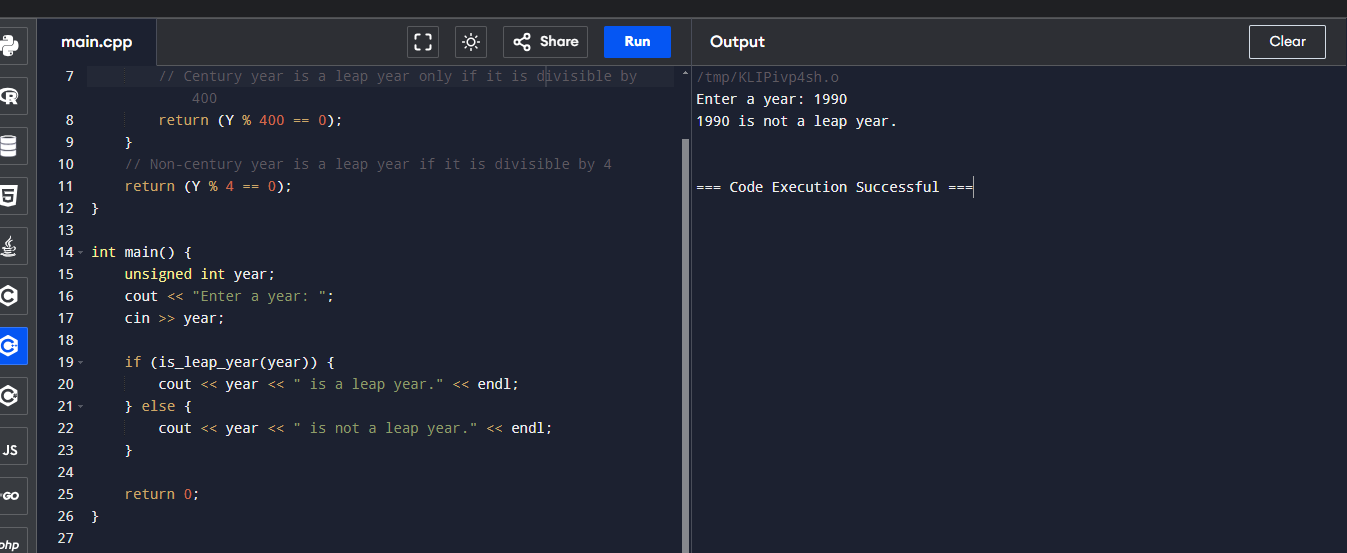
}

return 0;

}

**Screenshot of my ouput**

****

****

**QUESTION FOUR (B)**

To sum this series using C++, am going to use a loop to iterate over the numerators and denominators of the series and calculate the sum.

#include <iostream>

using namespace std;

int main() {

double sum = 0.0; // Initialize sum as a double for precision

int numerator = 1; // Starting numerator

int denominator = 7; // Starting denominator

// Loop through the series

while (numerator <= 95 && denominator <= 97) {

sum += static\_cast<double>(numerator) / denominator; // Add the term to the sum

numerator += 2; // Increment numerator by 2

denominator += 2; // Increment denominator by 2

}

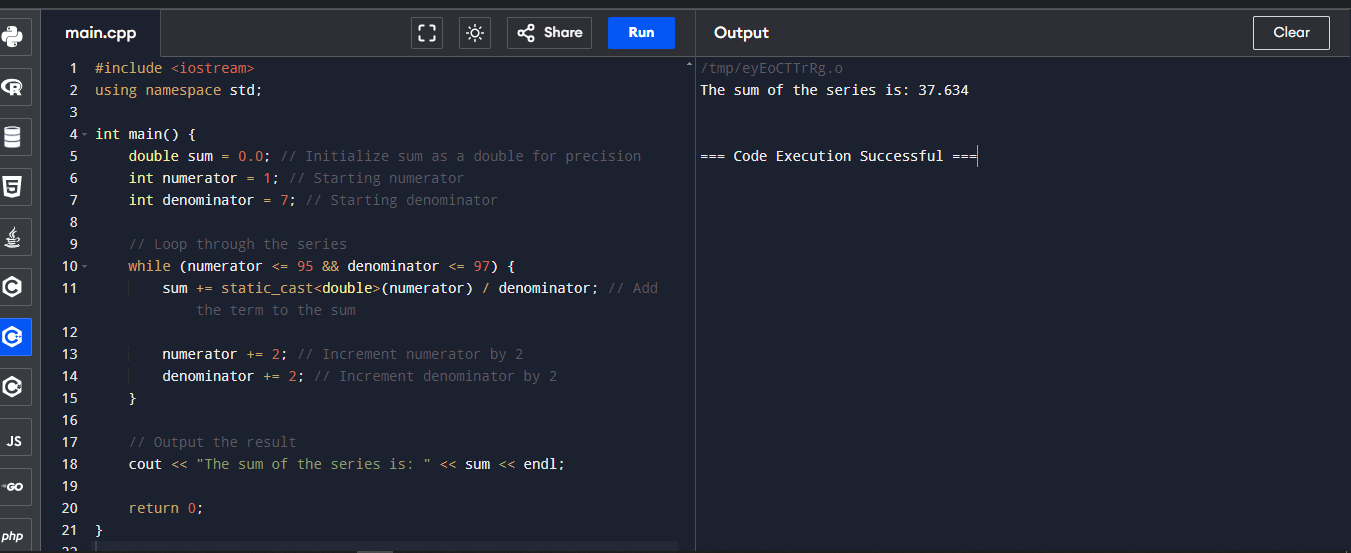
// Output the result

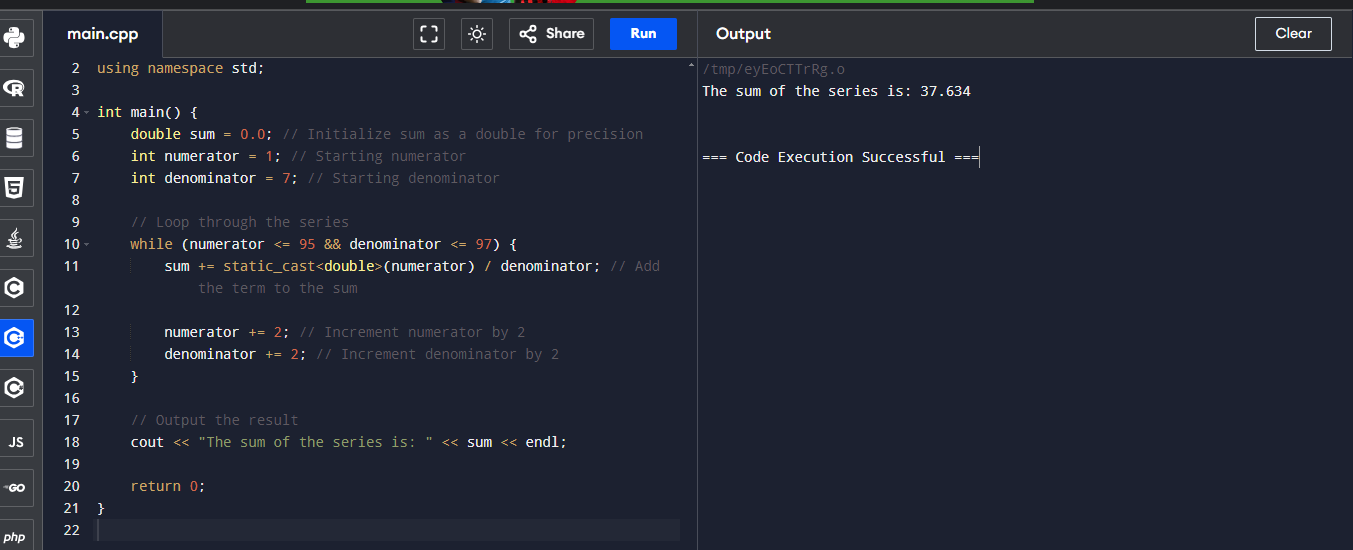
cout << "The sum of the series is: " << sum << endl;

return 0;

}

**Screenshot of my output**





**QUESTION TWO (A)**

**Function in C++**

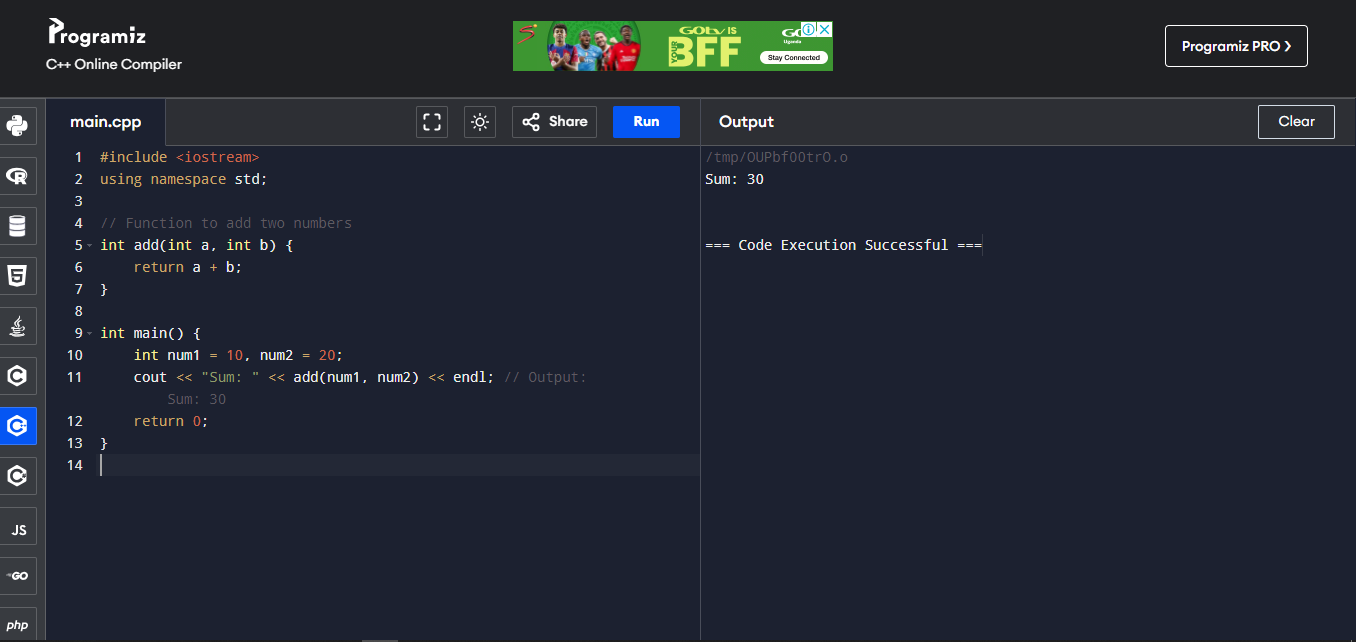
A function in C++ is a block of code designed to perform a specific task. Functions help in organizing code, improving readability, and facilitating reusability. Functions can be of two types: predefined (library functions) and user-defined functions. They can also have different return types and parameters.

**These are the Key Concepts:**

* **Return Type:** Specifies the type of value the function will return.
* **Function Name:** Identifies the function uniquely.
* **Parameters:** Variables passed to the function for processing.
* **Function Body:** Contains the code to be executed when the function is called.
* **Function Call:** Invokes the function to execute its code.

**Basic Implementation:**

**Screenshot**



**Code**

#include <iostream>

using namespace std;

// Function to add two numbers

int add(int a, int b) {

return a + b;

}

int main() {

int num1 = 10, num2 = 20;

cout << "Sum: " << add(num1, num2) << endl; // Output: Sum: 30

return 0;

}

**(ii) Arrays**

An **array** is a collection of elements of the same data type, stored in contiguous memory locations. It allows you to store multiple items using a single variable name and access them through an index. Arrays can be either single-dimensional or multi-dimensional.

1. **Single-dimensional Array:**

* A single-dimensional array is a linear data structure used to store a collection of elements of the same type. It is indexed, starting from zero, which allows quick access and manipulation of elements.
* **Use Case:** Ideal for storing a list of items like marks of students, scores, etc.

**Basic Implementation: code and screenshot of my output.**

#include <iostream>

using namespace std;

int main() {

int arr[5] = {1, 2, 3, 4, 5}; // Single-dimensional array

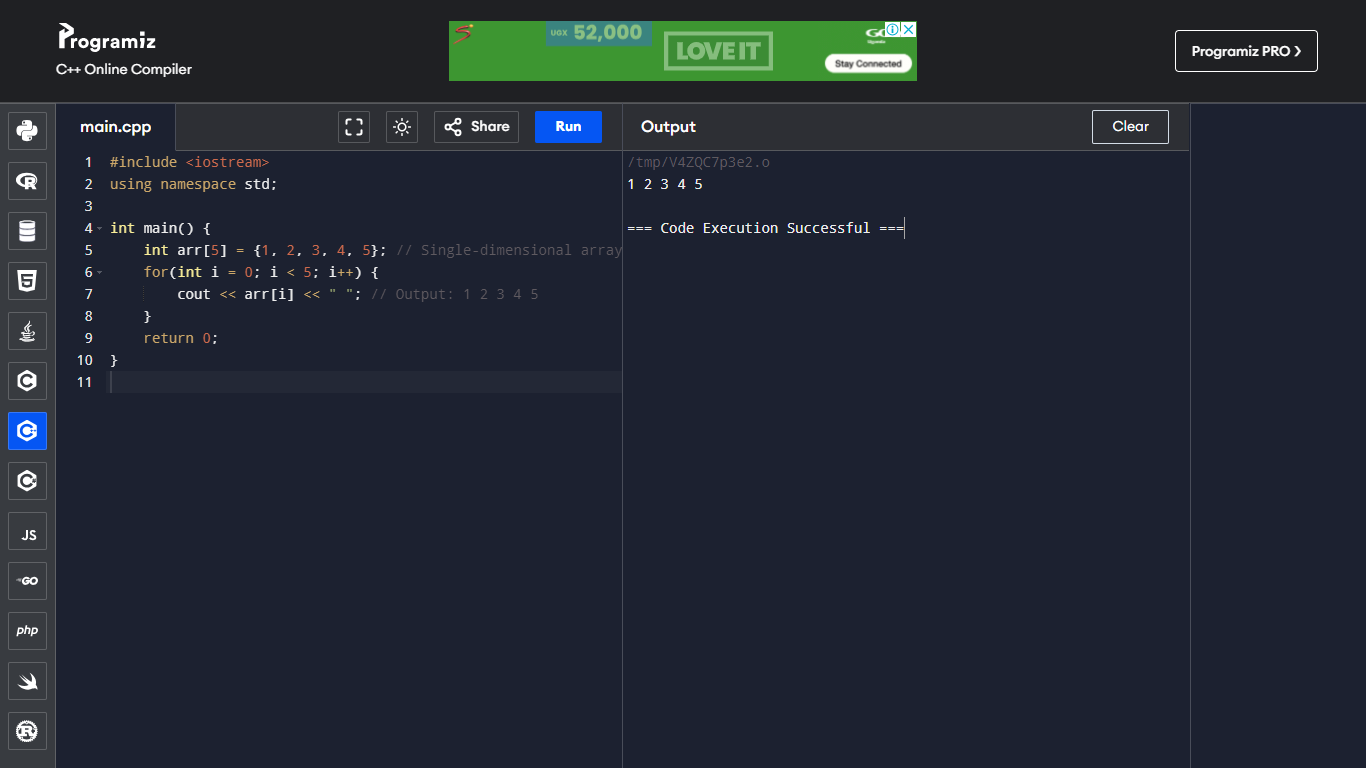
for(int i = 0; i < 5; i++) {

cout << arr[i] << " "; // Output: 1 2 3 4 5

}

return 0;

}



1. **Multi-dimensional Array:** A multi-dimensional array is an array of arrays. The most common multi-dimensional array is the two-dimensional array, often used to represent a matrix or table.

**Basic Implementation code and screenshot.**

#include <iostream>

using namespace std;

int main() {

int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; // 2D array

for(int i = 0; i < 2; i++) {

for(int j = 0; j < 3; j++) {

cout << matrix[i][j] << " "; // Output: 1 2 3 4 5 6

}

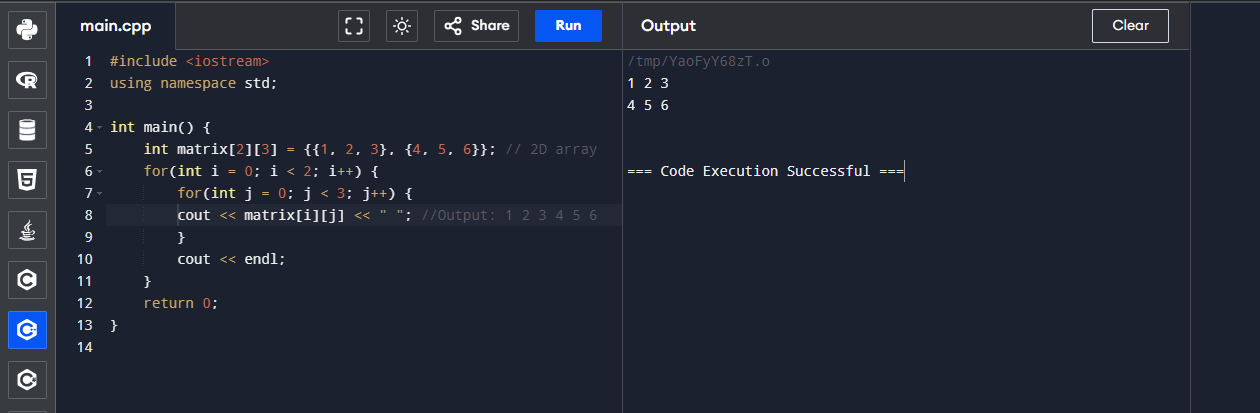
cout << endl;

}

return 0;

}

screenshot

****

**(iii) Loops**

Loops are used to repeatedly execute a block of code as long as a given condition is true. Common loops in C++ include for, while, and do-while loops.

**For Loop Example: code**

#include <iostream>

using namespace std;

int main() {

for(int i = 1; i <= 5; i++) { // Loop from 1 to 5

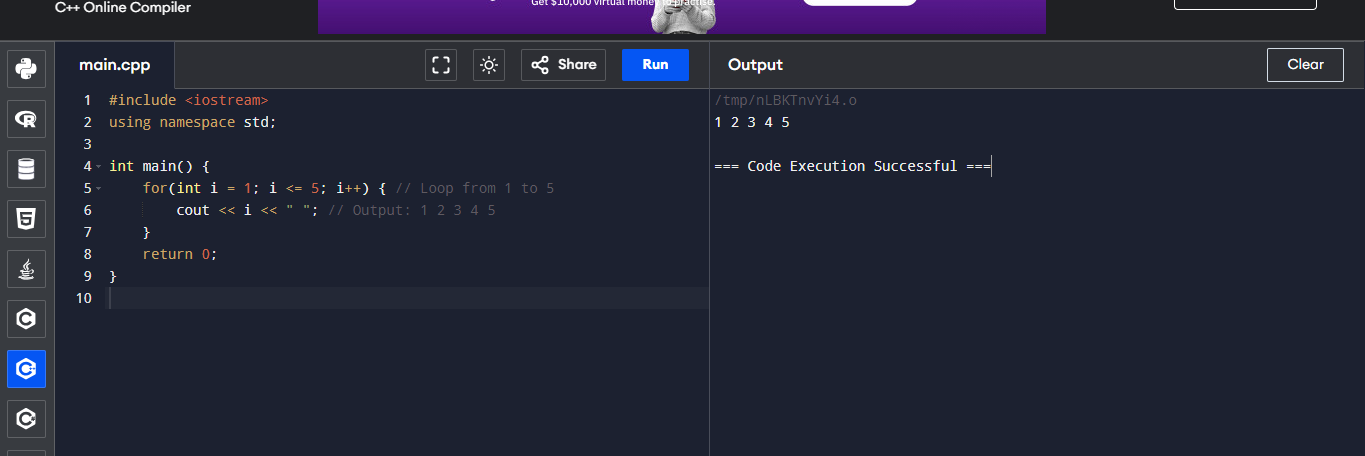
cout << i << " "; // Output: 1 2 3 4 5

}

return 0;

}

**For Loop Example: screenshot**



**While Loop Example: Code**

#include <iostream>

using namespace std;

int main() {

int i = 1;

while(i <= 5) { // Loop while i is less than or equal to 5

cout << i << " "; // Output: 1 2 3 4 5

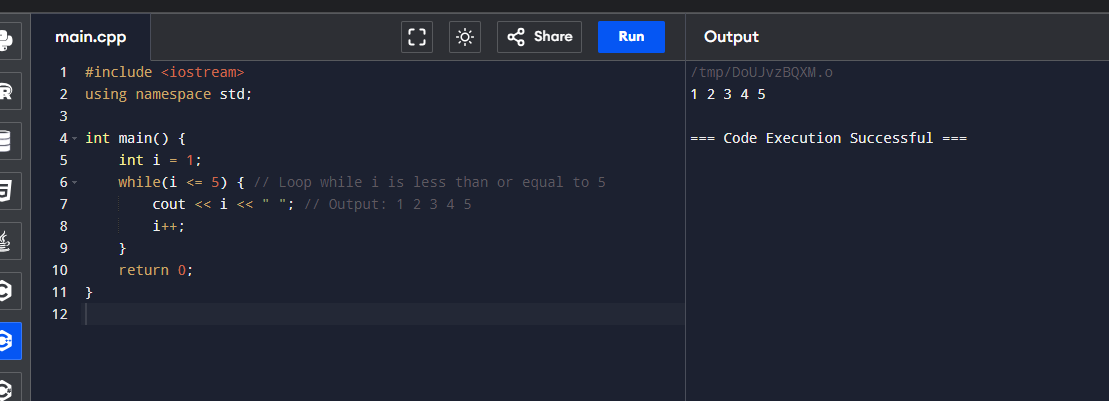
i++;

}

return 0;

}

**While Loop Example: screenshot**



**(iv) Control Statements**

Control statements guide the flow of execution in a program based on conditions. These include decision-making statements (if, if-else, switch) and looping constructs (for, while, do-while). They allow programs to make decisions, execute different code paths, and repeat actions.

**Types:**

1. **If-Else:** Executes a block of code based on a condition.
2. **Switch:** Allows a variable to be tested for equality against a list of values.
3. **Break:** Exits a loop or switch statement.
4. **Continue:** Skips the current iteration of a loop and continues with the next.

**Basic Implementation**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

// If-Else Ladder Example

if(age < 18) {

cout << "You are a minor." << endl;

} else if(age >= 18 && age <= 60) {

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

} else {

cout << "You are a senior citizen." << endl;

}

char grade;

cout << "Enter your grade (A, B, C, D, F): ";

cin >> grade;

// Switch Statement Example

switch(grade) {

case 'A':

cout << "Excellent!" << endl;

break;

case 'B':

cout << "Well done!" << endl;

break;

case 'C':

cout << "Good job!" << endl;

break;

case 'D':

cout << "You passed." << endl;

break;

case 'F':

cout << "Better luck next time." << endl;

break;

default:

cout << "Invalid grade." << endl;

}

// Break and Continue Example

for(int i = 1; i <= 10; i++) {

if(i == 5) {

continue; // Skip iteration when i is 5

}

if(i == 8) {

break; // Exit loop when i is 8

}

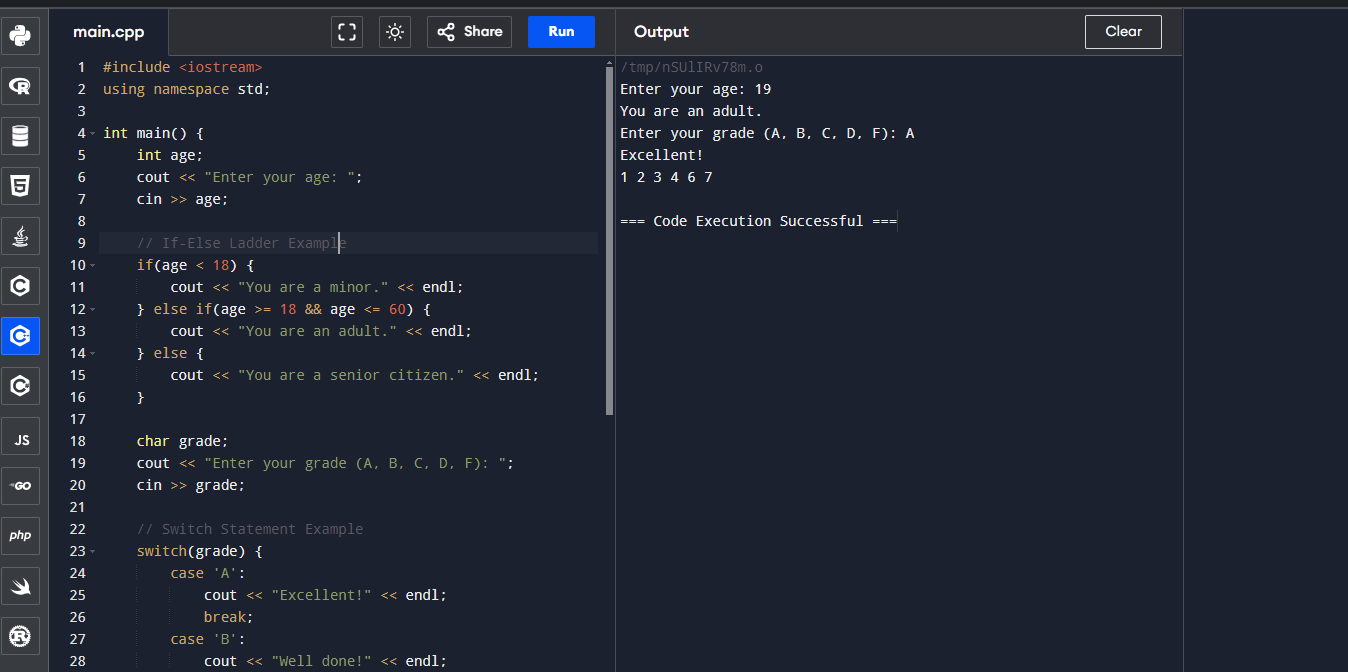
cout << i << " "; // Output: 1 2 3 4 6 7

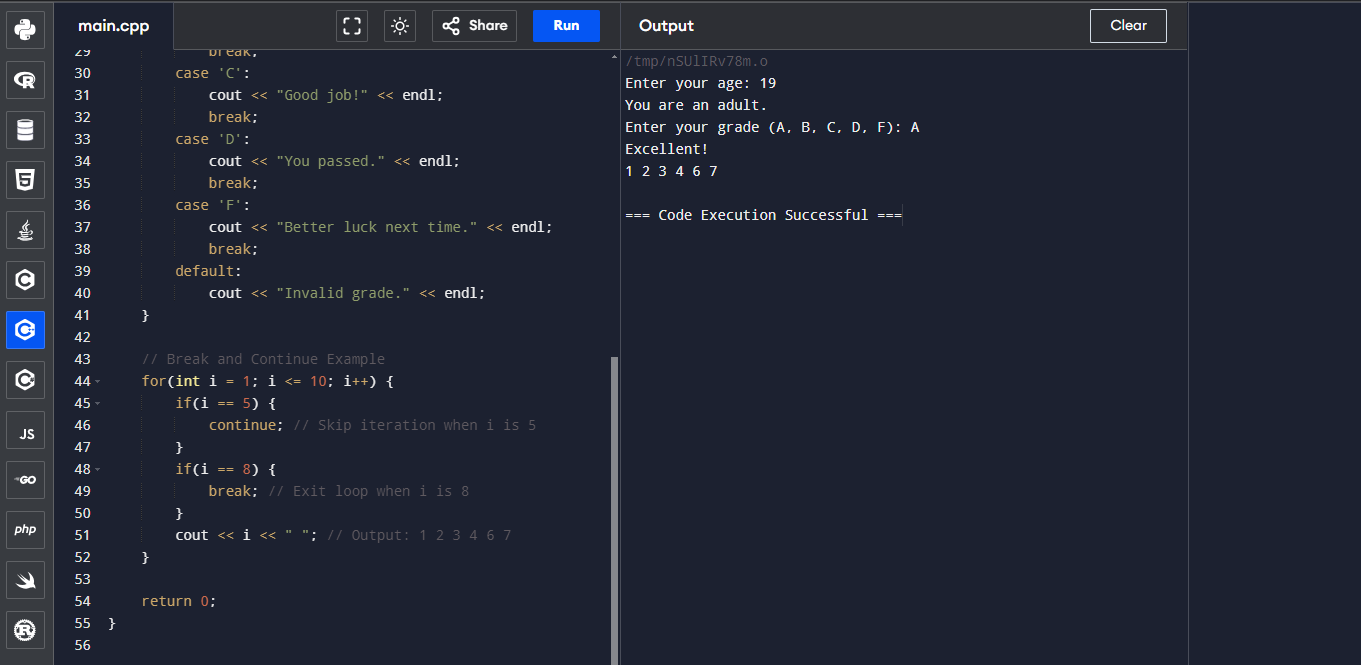
}

return 0;

}

**Basic Implementation screenshot**

****

****

**QUESTION TWO (B)**

**A C++ function that finds a triangle's third angle, taking the other two angles as input. Note: the sum of the angles in a triangle is always 180**

**Source code**

#include <iostream>

using namespace std;

// Function to find the third angle of a triangle

int findThirdAngle(int angle1, int angle2) {

// The sum of all angles in a triangle is always 180 degrees

int thirdAngle = 180 - (angle1 + angle2);

return thirdAngle;

}

int main() {

int angle1, angle2;

cout << "Enter the first angle: ";

cin >> angle1;

cout << "Enter the second angle: ";

cin >> angle2;

// Find and display the third angle

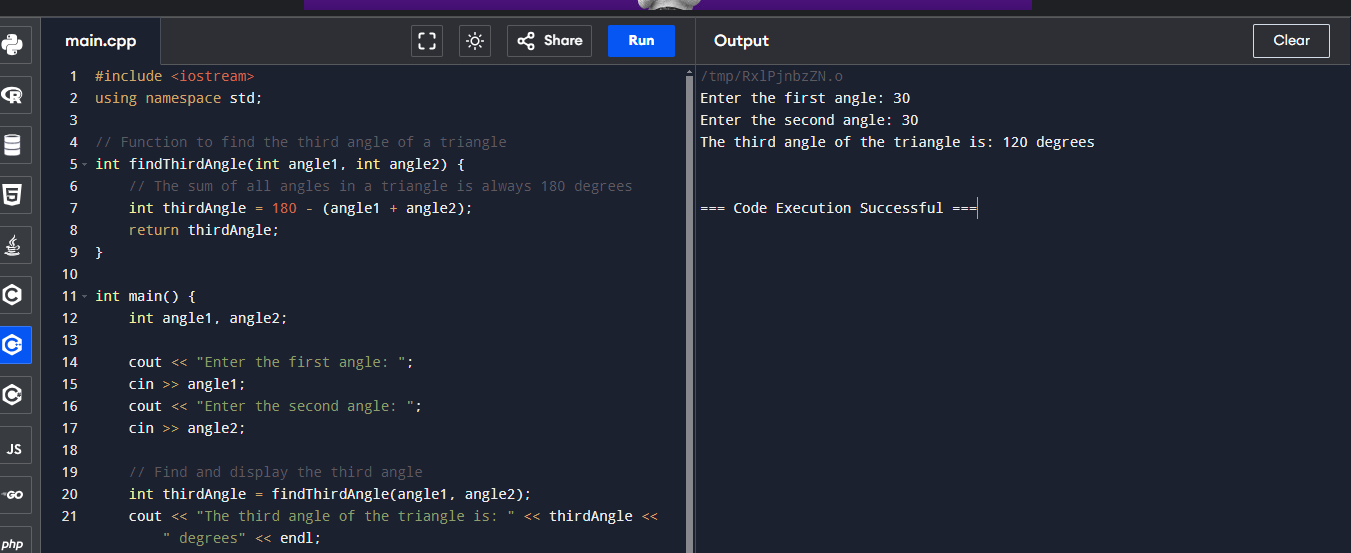
int thirdAngle = findThirdAngle(angle1, angle2);

cout << "The third angle of the triangle is: " << thirdAngle << " degrees" << endl;

return 0;

}

**Below is a screen shot**



**QUESTION TWO (C) (i)**

**QUESTION FOUR (A)**

In programming, there are three primary types of errors that programmers need to be aware of: syntax errors, runtime errors, and logic errors. Each type has distinct characteristics and requires different approaches for detection and resolution.

**1. Syntax Errors**

These occur when the code violates the rules of the programming language. It's similar to grammar errors in human languages.

**Examples:**

**Missing semicolons** in languages like JavaScript or Java.

Incorrectly using keywords or functions, such as misspelling print() as prnt() in Python.

**Detection:**

Syntax errors are usually caught by the compiler or interpreter when the code is being compiled or executed. Most integrated development environments (IDEs) highlight these errors as you type, making them easier to identify and fix.

**Resolution:**

Reviewing the error message provided by the compiler or interpreter, which typically indicates the location and nature of the error. Correct the syntax according to the language's rules.

**2. Runtime Errors**

These errors occur while the program is running, often due to unexpected conditions that the program cannot handle. They prevent the program from continuing its execution.

**Examples:**

* Division by zero, which causes a crash or an exception.
* Trying to access an index in an array that is out of bounds.
* File not found errors when attempting to open a non-existent file.

**Detection:** Runtime errors are not detected until the program is executed. They often cause the program to terminate unexpectedly or throw exceptions.

**Resolution:**

To handle runtime errors, programmers can use exception handling mechanisms like try...catch blocks (in languages like Java or C#) or try...except (in Python). It’s also essential to validate inputs and conditions that could lead to such errors.

**3. Logic Errors**

Logic errors occur when the code runs without crashing but produces incorrect or unintended results. These errors are often the most challenging to identify because the code is syntactically correct and runs as expected.

**Examples:**

Using the wrong formula in a calculation, such as area = length + width instead of area = length \* width.

Incorrectly using a loop condition, which may result in the loop running too many or too few times.

Logical conditions that never trigger certain branches of code, leading to unexpected behavior.

**Detection:**

Logic errors are typically identified through testing and debugging. Programmers must carefully analyze the output of their code and compare it with expected results.

**Resolution:**

To fix logic errors, carefully review the algorithm and logic used in the code. Debugging tools can help trace the flow of execution and the values of variables at different stages, making it easier to identify where the logic goes wrong.

**Summary**

**Syntax Errors:** Caught during compilation or interpretation; involve incorrect code structure.

**Runtime Errors:** Occur during program execution; often result from invalid operations or unexpected conditions.

**Logic Errors:** Produce incorrect results without crashing; require thorough testing and debugging to identify and correct.

In conclusion, paying attention to these errors and employing proper coding practices like code reviews, unit testing, and using debugging tools can help minimize their impact and improve code quality.

**QUESTION FOUR (B)**

The given C++ program structure has a slight mistake in the cout statement, particularly in the use of the division and assignment operator (/=).

T**he Statement: cout << "a / b = \t" << a/=b << endl;**

1. **Initialization**: If i assume the values a = 10 and b = 3 before this statement:
   * a /= b is equivalent to a = a / b.
   * In integer division, 10 / 3 equals 3 because the fractional part is discarded.
2. **Assignment**: The statement a /= b divides a by b and then assigns the result back to a. After this operation:
   * a becomes 3.
3. **Output**: The value of a after the operation is 3. Therefore, the statement cout << a /= b outputs the updated value of a (which is 3).

**Final Output**

The corrected program and output would be:

**Source Code**

#include <iostream>

using namespace std;

int main() {

int a = 10;

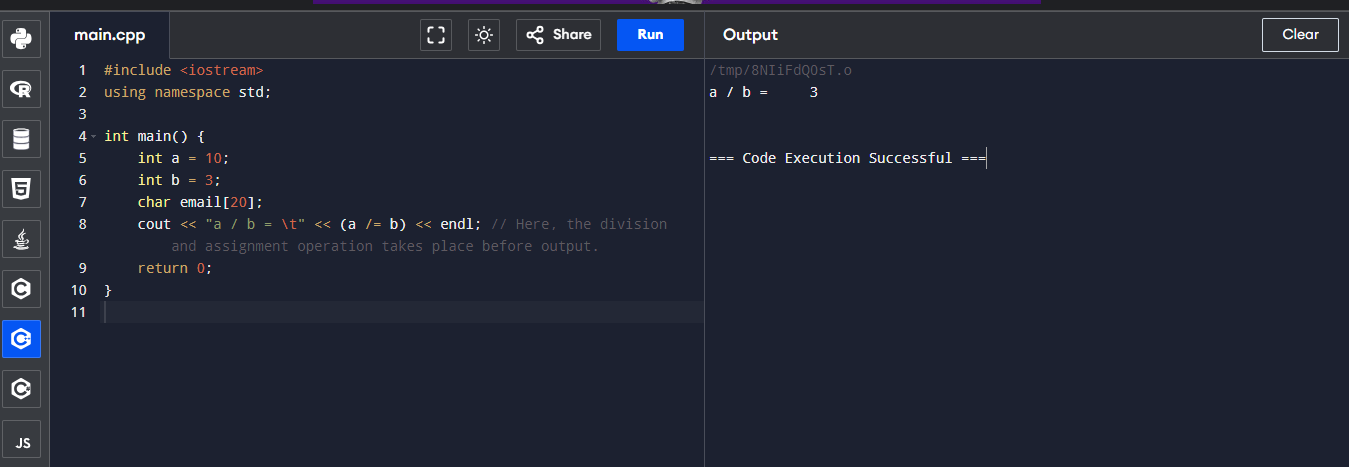
int b = 3;

char email[20];

cout << "a / b = \t" << (a /= b) << endl; // Here, the division and assignment operation takes place before output.

return 0;

}

**Screenshot. **

**QUESTION FOUR (B) (ii)**

The variable email is defined as an array of characters with a size of 20:

char email[20];

* The **size of the array** email is 20, which means it can hold up to 20 characters.
* This includes the null terminator \0, which is required to mark the end of a C-style string in C++.

**Length of Characters Stored**

* **About the capacity**: The array can store a string of up to 19 visible characters, because the 20th position is reserved for the null terminator \0.
* **About the actual content**: The length of the stored characters depends on what values are assigned to email within the program. In this snippet, since no value is assigned to email, it contains uninitialized garbage values or is empty.

In summary, the **capacity** of the email array is 20 characters, but the effective length of the stored string would be one less than the capacity when using C-style strings due to the null terminator requirement.

**QUESTION FOUR (C)(iii)**

**(iii) Assign an email and print it**

To assign an email to the email array and then print it to the screen, you can use the strcpy function from the <cstring> library and cout for printing.

#include <iostream>

#include <cstring> // Include this for strcpy

using namespace std;

int main() {

char email[20];

// Assign an email of choice

strcpy(email, "stephenlubanga@mail.com");

// Print the assigned email

cout << "Assigned Email: " << email << endl;

return 0;

}

**Explanation**

 strcpy(email, "stephenlubanga@mail.com"); assigns the string "example@mail.com" to the email variable.

 cout << "Assigned Email: " << email << endl; prints the assigned email.

**QUESTION FOUR (iv)**

**Get input from the user and print it**

To get input from the user, use the cin function. Since cin stops reading at the first whitespace, cin.get() or cin.getline() can be used for more flexibility with strings.

#include <iostream>

using namespace std;

int main() {

char email[20];

// Prompt the user for input

cout << "Enter your email: ";

// Get input from user and store it in the email variable

cin.getline(email, 20); // Use getline to allow spaces and restrict to 19 characters (19 + null terminator)

// Print the user-input email

cout << "Entered Email: " << email << endl;

return 0;

}

**Explanation**

 cin.getline(email, 20); reads up to 19 characters of input from the user and stores it in email. This prevents buffer overflow and includes a null terminator.

 cout << "Entered Email: " << email << endl; prints the user-entered email to the screen.

**QUESTION FOUR (C)**

This C++ program that allows the user to enter a grade between 0 and 100. The program validates the input using a do-while loop to ensure that the entered grade is within the acceptable range.

#include <iostream>

using namespace std;

int main() {

int grade;

// Use a do-while loop to validate the input

do {

cout << "Enter the grade scored in class (0-100): ";

cin >> grade;

// Check if the entered grade is out of range

if (grade < 0 || grade > 100) {

cout << "Invalid input. Please enter a grade between 0 and 100." << endl;

}

} while (grade < 0 || grade > 100); // Repeat the loop if the input is invalid

// Print the valid grade

cout << "You entered a valid grade: " << grade << endl;

return 0;

}

**Explanation:**

1. **Variable Declaration**: int grade; is used to store the user-entered grade.
2. **do-while Loop**:
   * The do block allows the user to enter a grade.
   * The if statement checks if the grade is outside the valid range (0-100). If so, an error message is displayed.
   * The loop continues (while condition) if the grade is not within the valid range.
3. **Valid Input**: Once a valid grade is entered, the loop exits and the grade is printed.

This program effectively ensures that the user only enters a valid grade between 0 and 100.

**QUESTION FIVE (A)**

**Definitions of Key C++ Programming Terms**

**(i) Data Type**

A **data type** in C++ specifies the kind of data a variable can hold and determines the memory allocated for that data. It dictates the range of values that can be stored and the operations that can be performed on those values. Data types are categorized into primitive types (e.g., int, float, char, bool), derived types (e.g., arrays, pointers), and user-defined types (e.g., structures, unions, classes). Understanding data types is essential for efficient memory management and error prevention in programming. For example, using an int instead of a double for decimal numbers can result in data loss due to type constraints.

**(ii) Variable**

A **variable** in C++ is a symbolic name for a memory location used to store data that can change during the program's execution. Variables have a type, name, and optionally, an initial value. They can be accessed and modified throughout the program, making them essential for dynamic and interactive applications. For instance, a variable age of type int can be used to store and update a person's age. The scope (local, global) and lifetime (duration of existence) of a variable are important concepts that define where and how long the variable can be accessed in the code.

**(iii) Algorithm**

An **algorithm** is a set of step-by-step instructions designed to solve a specific problem or perform a task. In C++, algorithms are implemented using loops, conditionals, and other control structures. Characteristics of a good algorithm include being well-defined, finite, and effective. For example, the algorithm for finding the greatest common divisor (GCD) of two numbers can be implemented using the Euclidean method. Understanding algorithms is crucial for writing efficient and optimized code, as they form the logic behind how data is processed and manipulated.

**(iv) Function**

A **function** in C++ is a reusable block of code that performs a specific task. It consists of a return type, name, parameters, and a body of code. Functions help break down complex programs into smaller, manageable parts, making them easier to understand, test, and maintain. For instance, a function calculateArea() can be used to compute the area of different shapes based on input parameters. Functions promote code modularity, reuse, and separation of concerns, which are fundamental principles in structured and object-oriented programming.

In conclusion, combining these concepts, C++will enables the development of complex, efficient, and maintainable software. Data types define the nature of data, variables store and manage this data, algorithms provide a methodical way to solve problems, and functions allow for modular and organized code. Together, they form the foundation of effective C++ programming.

**QUESTION FIVE B(i)**

**(i) Print Out Your Name**

This program will print out my name.

#include <iostream> //

int main() {

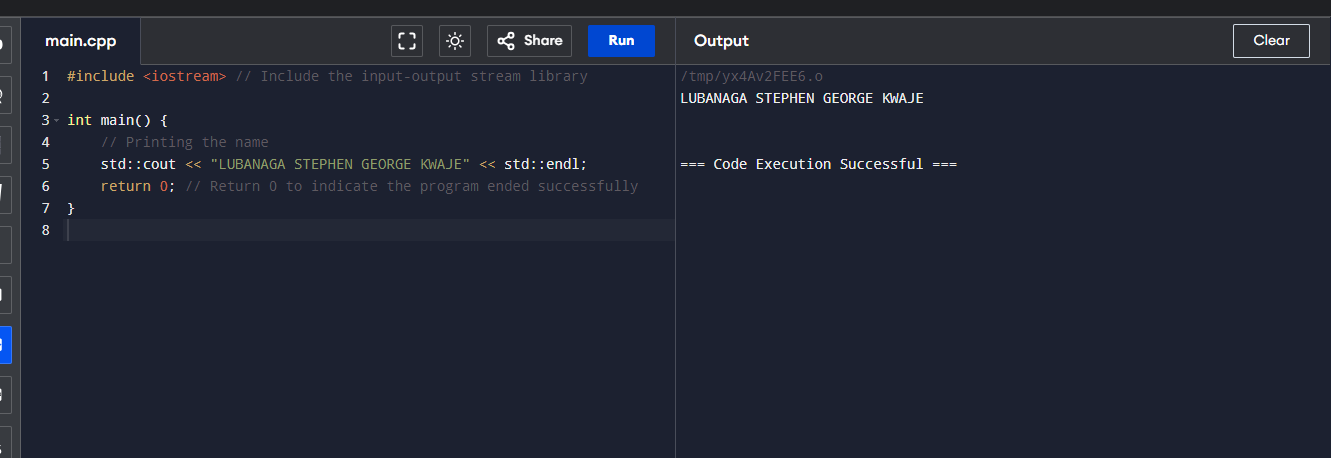
// Printing the name

std::cout << "LUBANAGA STEPHEN GEORGE KWAJE" << std::endl;

return 0; // Return 0 to indicate the program ended successfully

}

**screenshot**



This is My explanation of the **Print Out of My Name:**

* The program uses std::cout to print the name "LUBANAGA STEPHEN GEORGE KWAJE" to the console.
* std::endl is used to insert a new line after printing the name.

**(ii) Print Multiples of 3 from 1 to 20**

This program prints out all the multiples of 3 from 1 to 20, each on a new line.

#include <iostream> //

int main() {

// Loop through numbers from 1 to 20

for (int i = 1; i <= 20; ++i) {

// Check if the number is a multiple of 3

if (i % 3 == 0) {

std::cout << i << std::endl; // Print the number followed by a new line

}

}

return 0; // Return 0 to indicate the program ended successfully

}

**This is my explanation of the Print Multiples of 3:**

* A for loop iterates from 1 to 20.
* The if statement checks if a number is divisible by 3 using the modulus operator (%).
* If the condition is true, the number is printed using std::cout and std::endl to move to the next line.

**QUESTION FIVE (C)**

Comments in C++ are non-executable parts of the code that are used to describe or annotate the program. They do not affect the program's execution and are ignored by the compiler. Comments are crucial for making the code more readable and maintainable by providing explanations, documentation, or notes within the source code. They can also be used to temporarily disable parts of the code during debugging or development.

There are two main types of comments in C++:

1. **Single-line Comments**
2. **Multi-line Comments**

**(i) Single-line Comments**

**Single-line comments** in C++ are used to annotate or explain specific lines of code. They begin with two forward slashes (//) and extend to the end of the line. Anything following // on that line is ignored by the compiler. These comments are typically used for brief explanations, code documentation, or to temporarily disable a single line of code during testing or debugging.

**Why to Use:**

* To provide brief explanations or annotations for specific lines of code, making it easier for others (or yourself) to understand what the code does.
* To temporarily disable (comment out) a single line of code during debugging or testing without deleting it.

**How to Use:**

* Single-line comments in C++ are written using two forward slashes (//). Everything following // on that line is treated as a comment and is ignored by the compiler.

**When to Use:**

* When you need to explain the purpose of a specific line of code, variable, or function call.
* To annotate code changes or fixes, especially when working in collaborative projects.
* To add to-do notes or reminders within the code.

**This is the source code.**

int age = 25; // This variable stores the age of the user

// age = age

**(ii) Multi-line comments**

**Multi-line comments** in C++ are a way to include descriptive text or annotations that span multiple lines within the source code. They begin with /\* and end with \*/, and everything between these markers is ignored by the compiler. Multi-line comments are used to provide detailed documentation, explain complex code sections, or temporarily disable large blocks of code during debugging. They are particularly useful when the explanation or notes are too lengthy to fit into a single line.

**Why to Use:**

* To provide detailed explanations or documentation for a section of code, function, or class.
* To describe complex logic, algorithms, or the purpose of a program module.
* To temporarily comment out multiple lines of code during testing or debugging without deleting them.

**How to Use:**

* Multi-line comments in C++ begin with /\* and end with \*/. Everything between these symbols is treated as a comment, regardless of how many lines it spans.

**When to Use:**

* When you need to add a comprehensive explanation or documentation that spans several lines.
* To comment out large blocks of code during development or testing to isolate issues.
* To include detailed information about the code’s functionality, such as author notes, version history, or specific implementation details.

**For example.**

#include <iostream>

int main() {

std::cout << "Hello, World!" << std::endl;

return 0;

}