1.) #include<iostream>

using namespace std;

int main() {

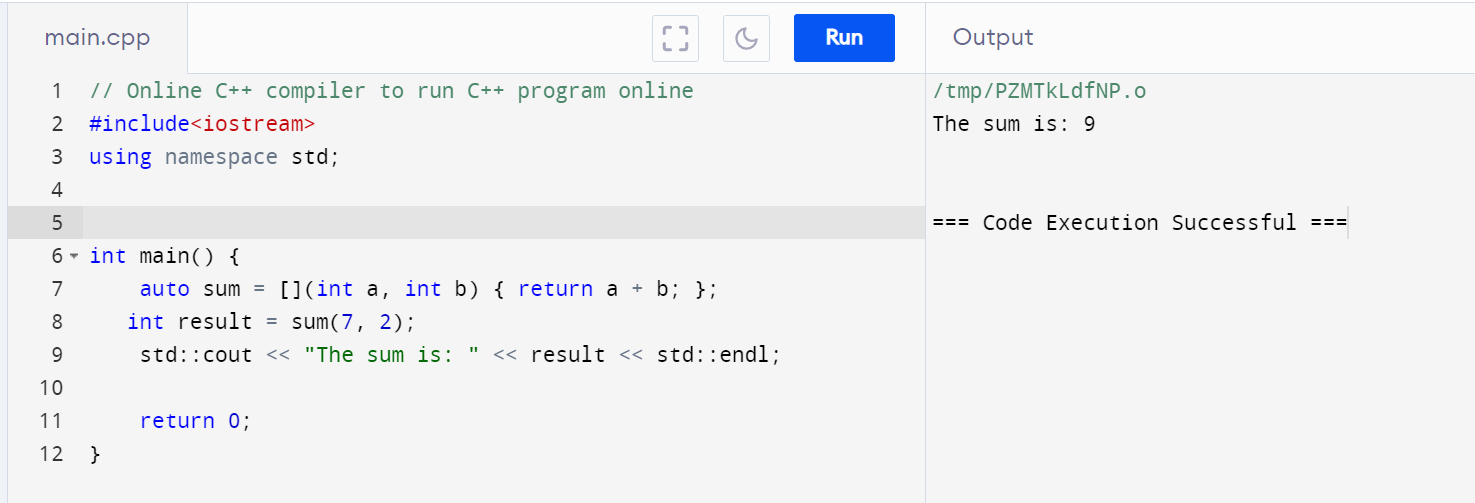
auto sum = [](int a, int b) { return a + b; };

int result = sum(7, 2);

std::cout << "The sum is: " << result << std::endl;

return 0;

}



2.) #include <iostream>

using namespace std;

int main() {

int num = 3;

// Lambda capturing 'num' by value and squaring it

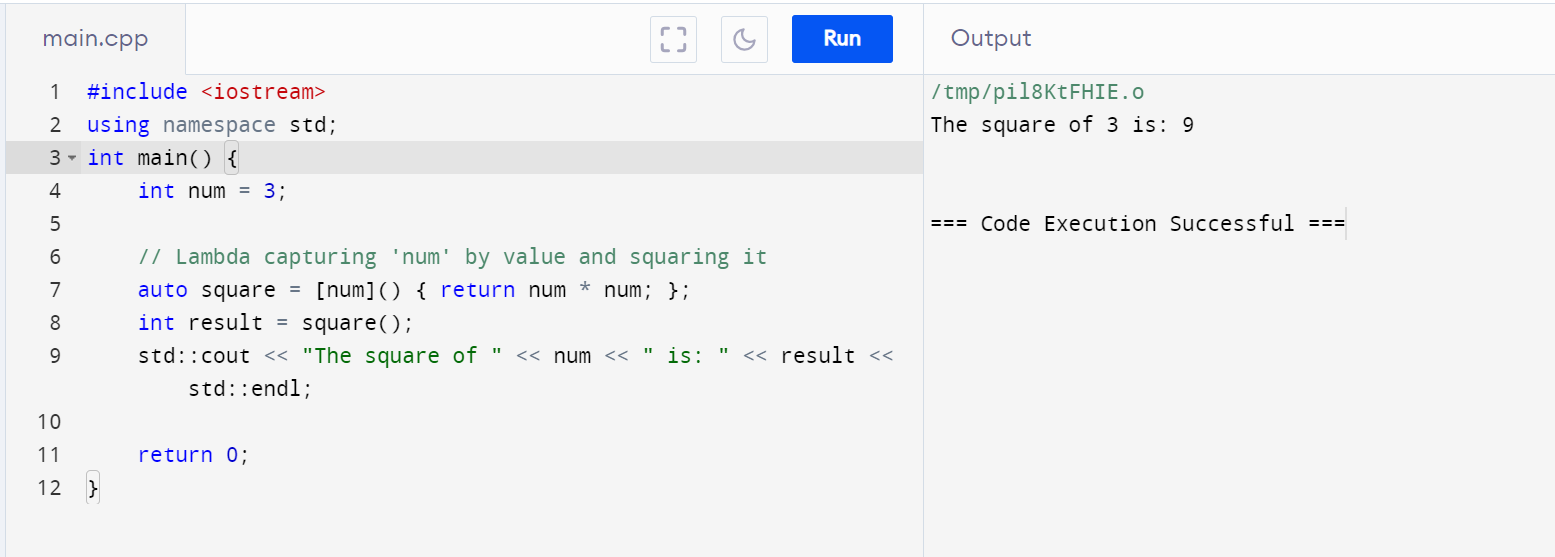
auto square = [num]() { return num \* num; };

int result = square();

std::cout << "The square of " << num << " is: " << result << std::endl;

return 0;

}



3.) #include <iostream>

using namespace std;

int main() {

std::string text = "world!";

auto addPrefix = [&text]() -> std::string {

std::string prefix = "Hello, ";

text = prefix + text;

return text;

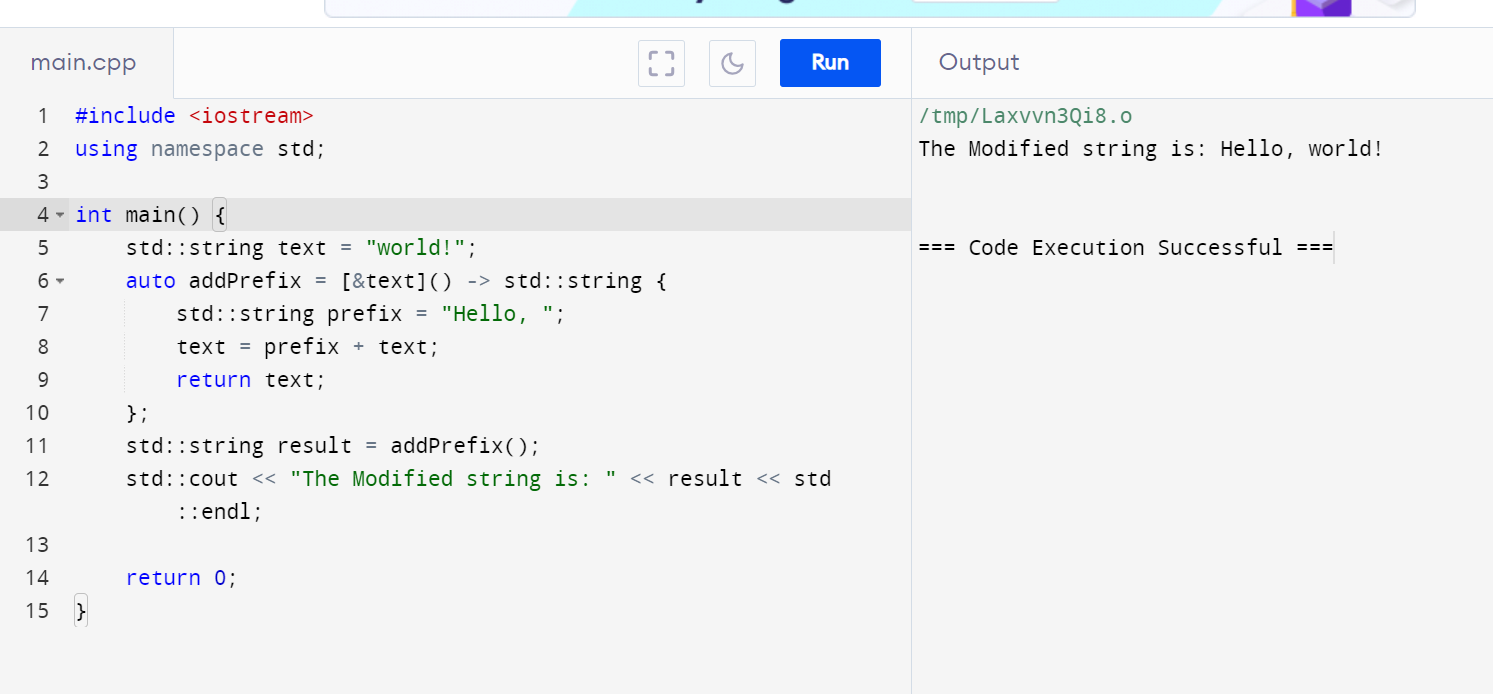
};

std::string result = addPrefix();

std::cout << "The Modified string is: " << result << std::endl;

return 0;

}



4.) #include <iostream>

int main() {

int number = 6;

bool shouldDouble = true;

// Lambda capturing 'number' and 'shouldDouble' by value

auto conditionalOperation = [number, shouldDouble]() -> int {

if (shouldDouble) {

return number \* 2;

} else {

return number;

}

};

// Example usage

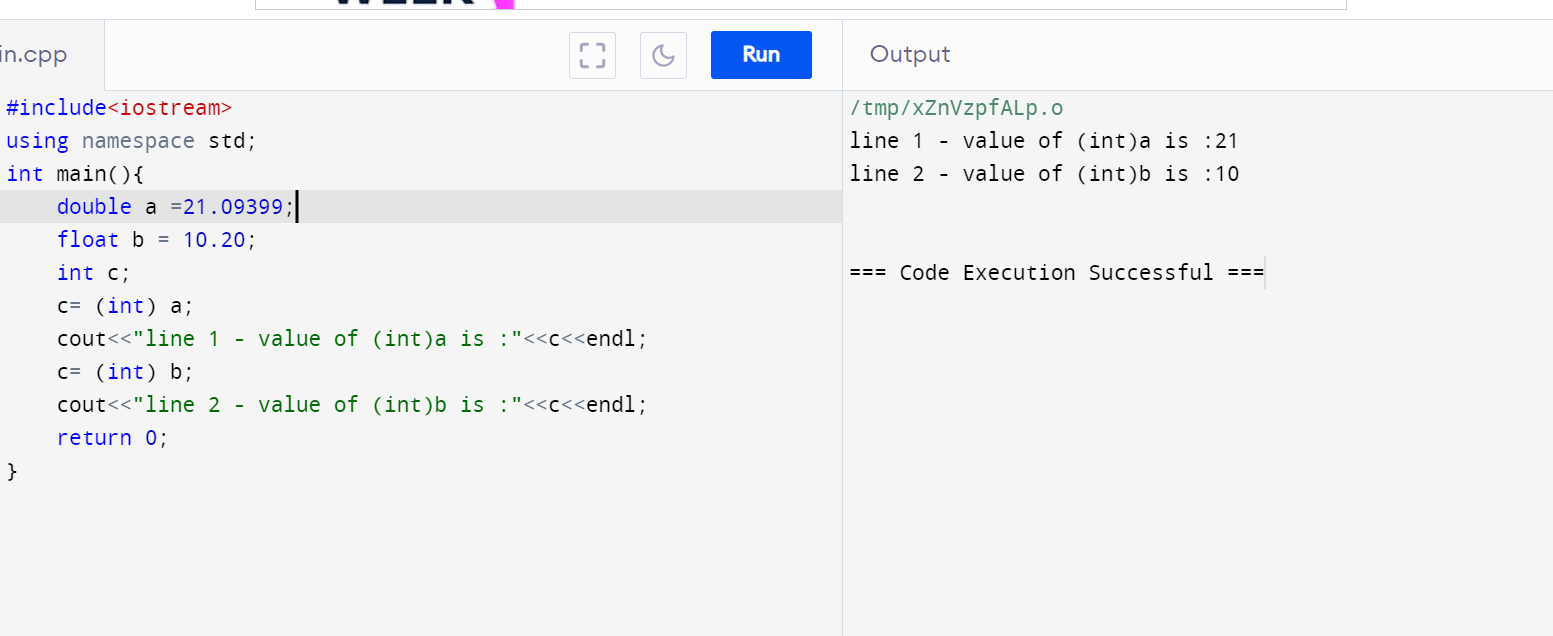
int result = conditionalOperation();

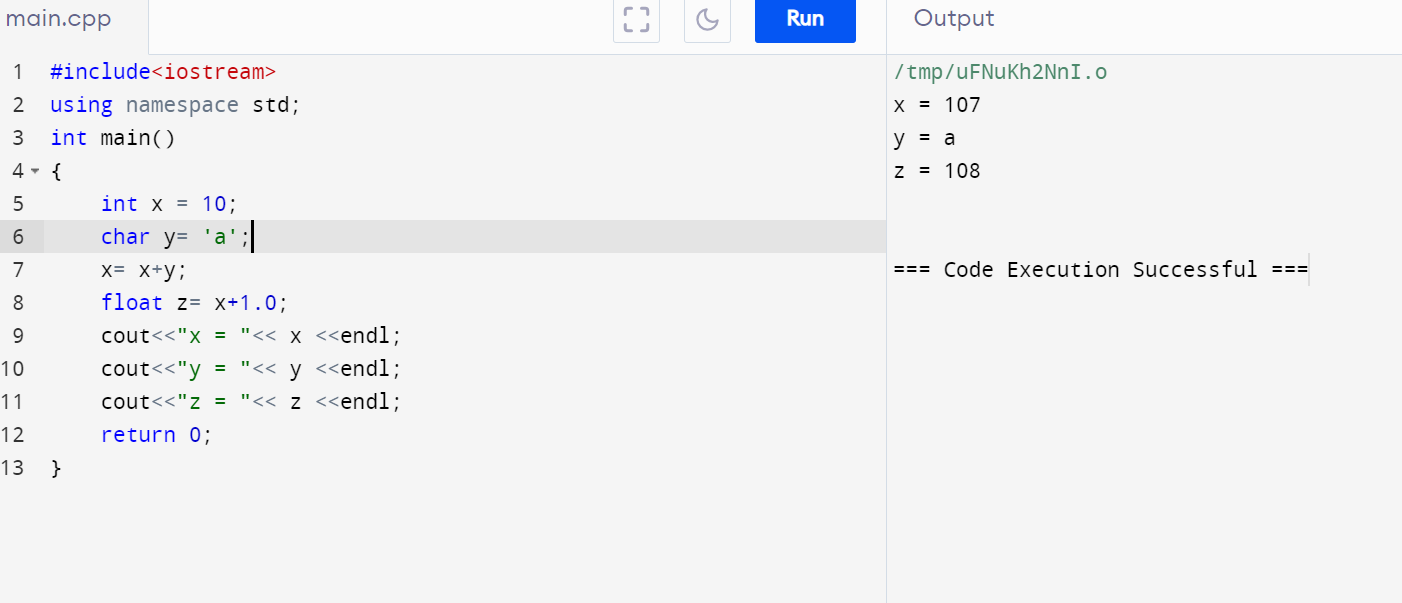
std::cout << "The result is: " << result << std::endl;

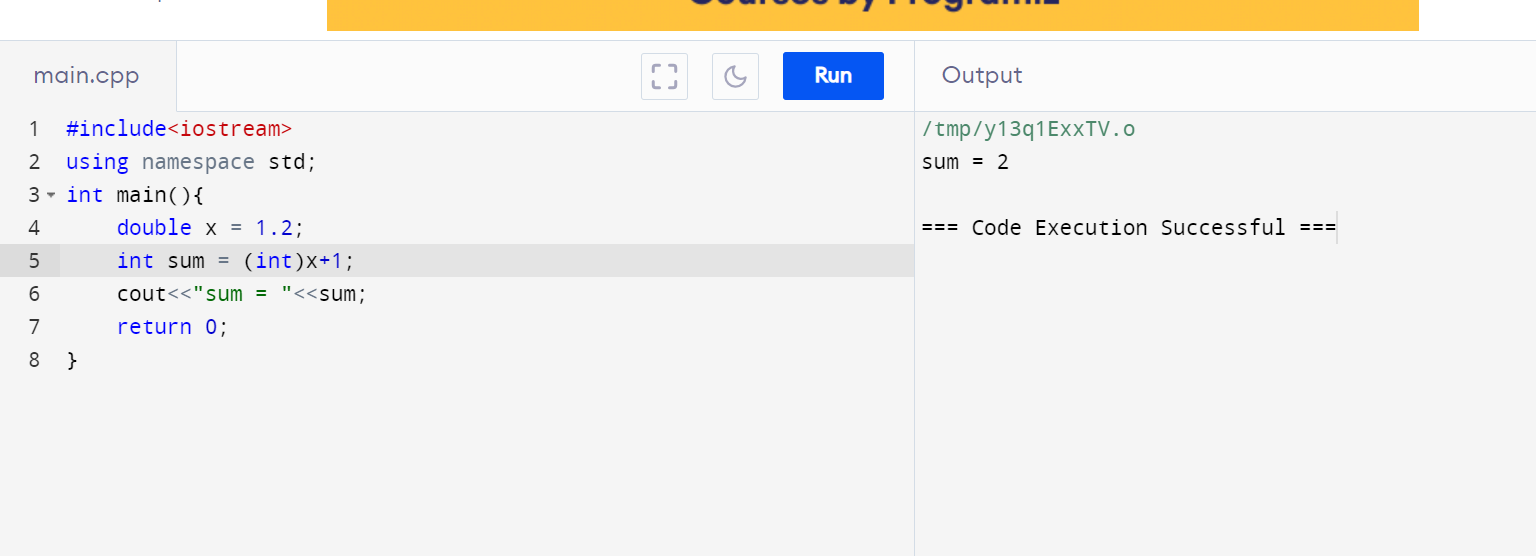
return 0;

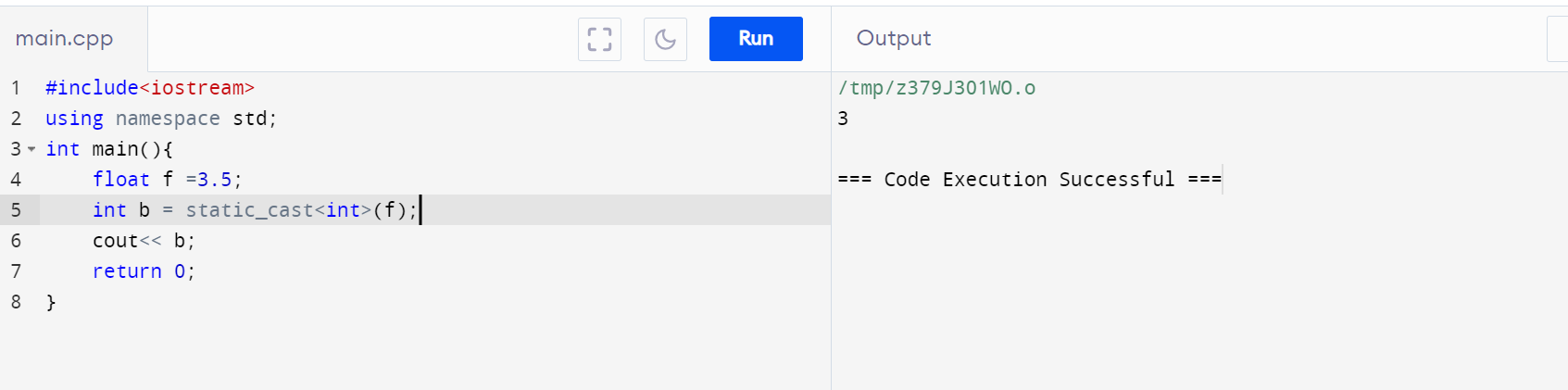
}



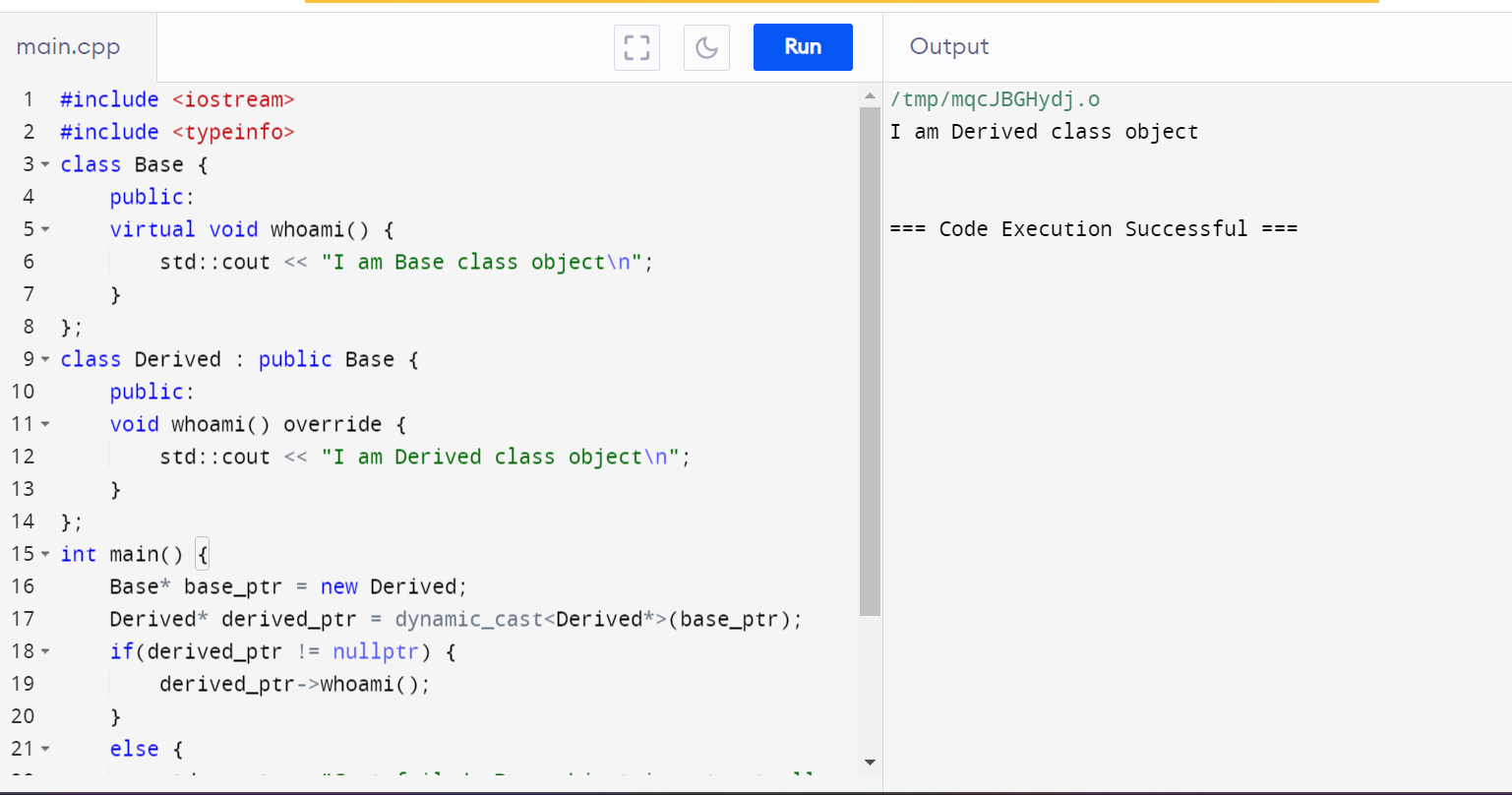




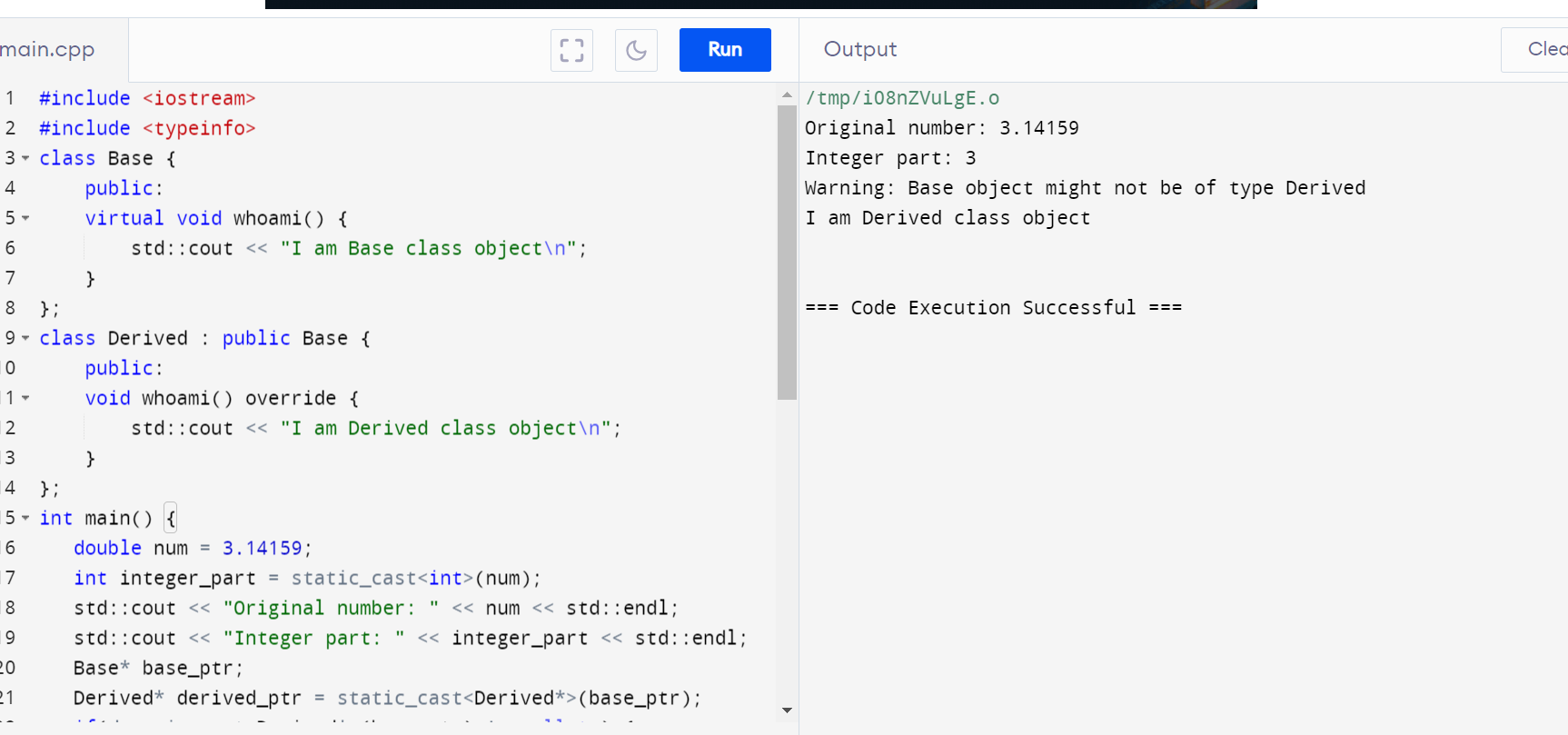


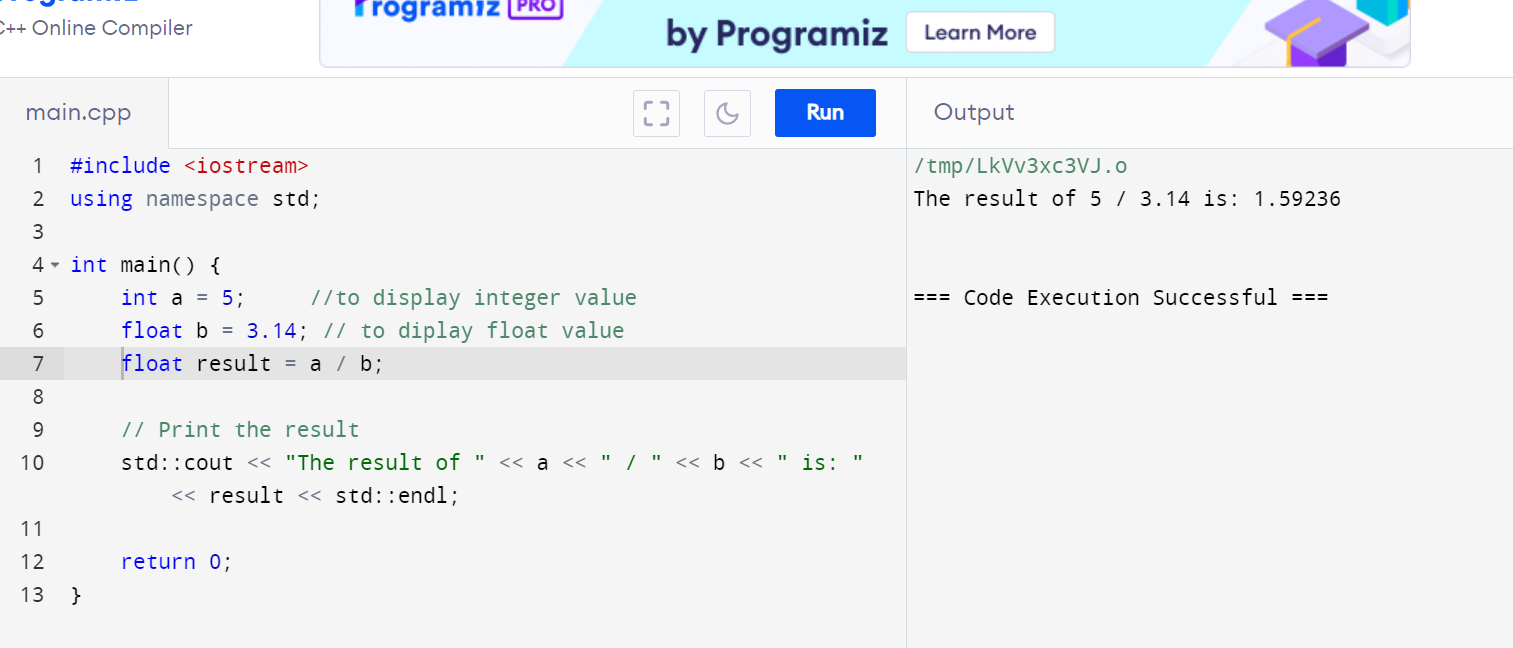












**Implicit Casting :**

When performing the division a / b:

* a is of type int.
* b is of type float.

In C++, when an operation involves operands of different types, implicit casting (or type promotion) occurs to convert the operands to a common type. In this case, the int a is implicitly cast to a float before the division, because b is a float. This ensures that the division operation is performed using floating-point arithmetic, which can handle the decimal part of the division result.

2.) #include <iostream>

int main() {

int x = 256;

char y = static\_cast<char>(x); // Explicit casting from int to char

std::cout << "Value of y: " << static\_cast<int>(y) << std::endl; // Print value of y

return 0;

}

Understanding the Data Loss:

Value Assignment: int x = 256; assigns the value 256 to x, which is well within the range of int.

Explicit Casting: char y = static\_cast<char>(x); casts the value of x (256) to a char. In C++, a char typically ranges from -128 to 127 (if signed) or 0 to 255 (if unsigned).

Data Loss: Since char is usually signed by default in most implementations, the value 256 (which is 0x100 in hexadecimal) exceeds the maximum value representable by a signed char (which is 127). Therefore, when 256 is cast to char, it undergoes what's called integer overflow and wraps around due to the limited range of char. The result of wrapping around depends on the platform's implementation, but typically it becomes 0 because of the two's complement representation.

Output: std::cout << static\_cast<int>(y); will output the integer value corresponding to y. If y is 0 due to overflow, std::cout will print 0.

Avoiding Data Loss:

Check the Range: Before performing the cast, check if the value of x lies within the range of char. For a char, this would typically mean checking if x is between -128 and 127 (for signed char) or 0 to 255 (for unsigned char).

Use Conditional Logic: If x exceeds the range of char, you should handle this situation appropriately in your code. This might involve clamping the value or deciding on a default behavior based on your application's requirements.

Consider Alternative Types: If possible, reconsider whether char is the appropriate type for your needs. If you anticipate needing values outside its range, consider using a larger data type such as int or short.

3.) #include <iostream>

int main() {

double d = 123.456;

int i = static\_cast<int>(d);

std::cout << "Converted value: " << i << std::endl;

return 0;

}

Explanation of behavior:

Conversion Process: When you cast double d to int i using static\_cast<int>(d), the compiler truncates the fractional part of d (123.456) and assigns the integer part to i.

Resulting Value: In this case, i will be assigned the value 123. This is because converting from double to int truncates towards zero; it simply drops the decimal part of the number.

Range Consideration:

Double (double): Typically holds a range of values much larger than int, and can represent both fractional and whole numbers with greater precision.

Integer (int): Has a smaller range compared to double, usually from -2147483648 to 2147483647 on most systems (32-bit integers).

Behavior with Larger to Smaller Conversion:

If the double value exceeds the range that can be represented by an int, the behavior is undefined in C++. This could result in unexpected values or errors depending on the compiler and system.

For values within the range of int, the conversion is straightforward, and the result is predictable

4.) #include <iostream>

int main() {

int num = 10;

int \*ptr = &num; // Pointer to int, pointing to num

float fPtr = reinterpret\_cast<float>(ptr); // Explicitly cast ptr to a float pointer

std::cout << "Value of num (interpreted as float): " << \*fPtr << std::endl; // Access the float pointer and print its value

return 0;

}

No, this casting is not safe.:

* **Different Types**: An int and a float have different internal representations and sizes in memory. For example, an int typically uses 4 bytes to store an integer value, while a float typically uses 4 bytes to store a floating-point value, but the bit patterns for these types are interpreted differently by the hardware.
* **Pointer Interpretation**: When you cast an int\* to a float\*, the pointer itself does not change, but the interpretation of the memory content changes. Dereferencing fPtr will interpret the bits at the memory location as a float rather than an int, leading to undefined behavior and potentially incorrect or nonsensical values.

**5**.) #include <iostream>

int main() {

int num = 20;

float fval = 3.14;

int \*intPtr = &num;

float \*floatPtr = &fval;

std::cout << "Value through floatPtr: " << \*floatPtr << std::endl;

std::cout << "Value through intPtr: " << \*intPtr << std::endl;

\*intPtr = 30;

std::cout << "Modified value through floatPtr (undefined behavior): " << \*floatPtr << std::endl;

return 0;

}

6.) #include <iostream>

int main() {

int x = 42;

int& refX = x;

try {

float& refF = reinterpret\_cast<float&>(refX);

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

std::cout << "Value of refX: " << refX << std::endl;

std::cout << "Value of refF (interpreted as float): " << refF << std::endl;

} catch (...) {

std::cerr << "An error occurred during the cast." << std::endl;

}

return 0;

}

7.) #include <iostream>

int main() {

int x = 42;

float f = 3.14f;

int& refX = x;

try {

float& refF = reinterpret\_cast<float&>(refX);

// Print the values and addresses

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

std::cout << "Value of f: " << f << std::endl;

std::cout << "Value of refX: " << refX << std::endl;

std::cout << "Value of refF (interpreted as float): " << refF << std::endl;

} catch (...) {

std::cerr << "An error occurred during the cast." << std::endl;

}

return 0;

}

8.) #include <iostream>

int area(int width, int height) {

return width \* height;

}

double area(double width, double height) {

return width \* height;

}

int main() {

int intWidth = 5;

int intHeight = 10;

int intArea = area(intWidth, intHeight);

std::cout << "Area of rectangle (int): " << intArea << std::endl;

double doubleWidth = 5.5;

double doubleHeight = 10.5;

double doubleArea = area(doubleWidth, doubleHeight);

std::cout << "Area of rectangle (double): " << doubleArea << std::endl;

double implicitCastArea = area(intWidth, intHeight);

std::cout << "Area of rectangle (implicit cast to double): " << implicitCastArea << std::endl;

int explicitCastArea = area(static\_cast<int>(doubleWidth), static\_cast<int>(doubleHeight));

std::cout << "Area of rectangle (explicit cast to int): " << explicitCastArea << std::endl;

return 0;

}

9.) #include <iostream>

#include <cmath>

double celsiusToFahrenheit(double celsius) {

return (celsius \* 9.0 / 5.0) + 32.0;

}

int main() {

double celsius;

std::cout << "Enter temperature in Celsius: ";

std::cin >> celsius;

double fahrenheit = celsiusToFahrenheit(celsius);

int roundedFahrenheit = static\_cast<int>(std::round(fahrenheit));

std::cout << "Temperature in Fahrenheit: " << roundedFahrenheit << std::endl;

return 0;

}

10.) #include <iostream>

void safePointerArithmetic() {

int arr[5] = {1, 2, 3, 4, 5};

int\* ptr = arr;

std::cout << "Safe Pointer Arithmetic:" << std::endl;

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

std::cout << "Value at arr[" << i << "] = " << \*ptr << std::endl;

++ptr; // Safe pointer arithmetic

}

}

void unsafePointerArithmetic() {

int arr[5] = {1, 2, 3, 4, 5};

int\* ptr = arr;

std::cout << "Unsafe Pointer Arithmetic:" << std::endl;

// Performing unsafe pointer arithmetic by casting to char\* and manipulating

char\* charPtr = reinterpret\_cast<char\*>(ptr);

for (int i = 0; i < sizeof(arr); ++i) {

std::cout << "Byte at position " << i << " = " << static\_cast<int>(\*charPtr) << std::endl;

++charPtr; // Unsafe pointer arithmetic

}

}

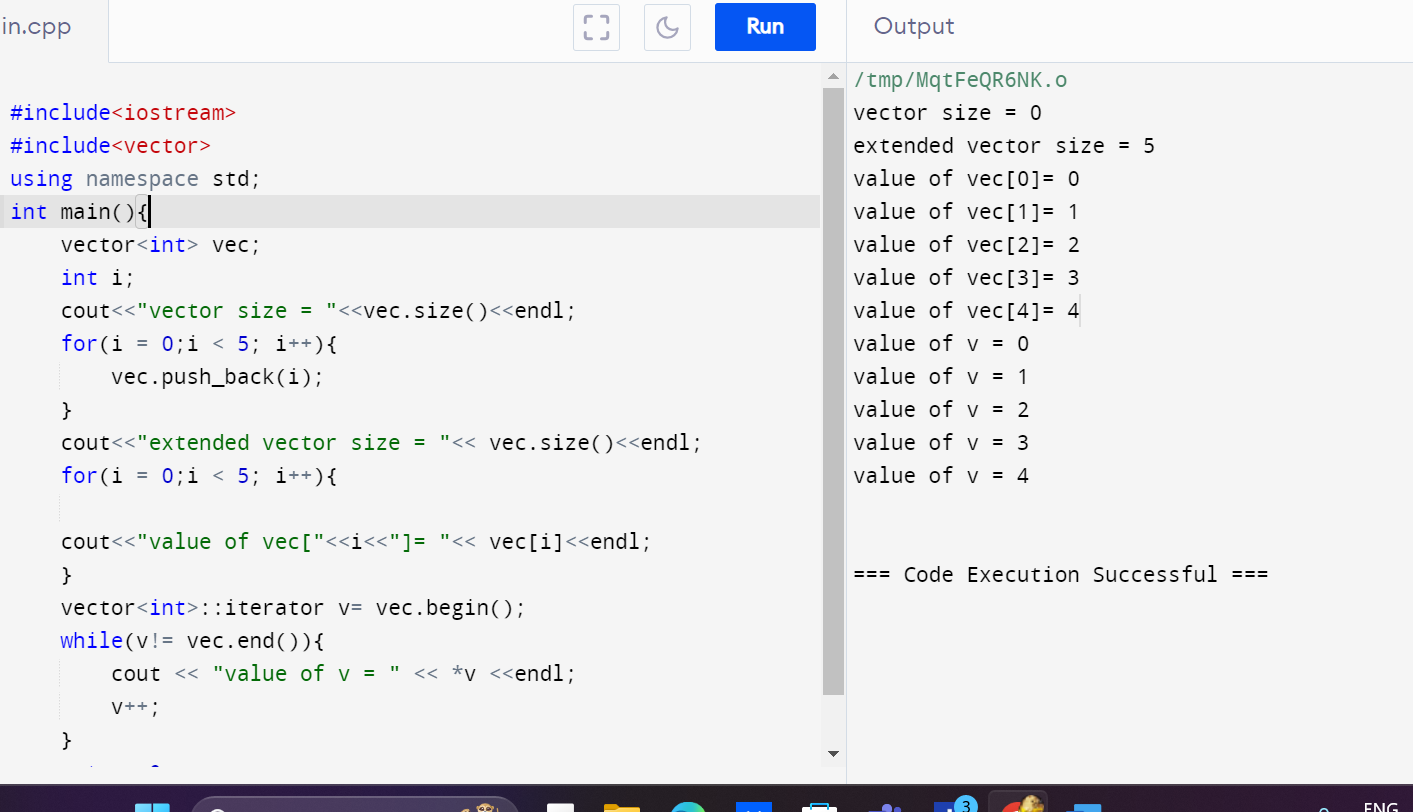
int main() {

safePointerArithmetic();

unsafePointerArithmetic();

return 0;

}



#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> numbers;

int num;

std::cout << "Enter integers (type 'done' to finish):" << std::endl;

// Read integers from the user

while (std::cin >> num) {

numbers.push\_back(num);

}

// Clear the error flag and ignore remaining input

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

// Sort the numbers

std::sort(numbers.begin(), numbers.end());

// Print the sorted numbers

std::cout << "Sorted integers:" << std::endl;

for (const int& n : numbers) {

std::cout << n << " ";

}

std::cout << std::endl;

return 0;

}