TASK – 1

1. Basic Lambda: Define a lambda expression that takes two integers as arguments and returns their sum. Use auto to infer the return type.

#include <iostream>

using namespace std;

int main() {

auto sum = [](int a, int b) {

return a + b;

};

int x = 9;

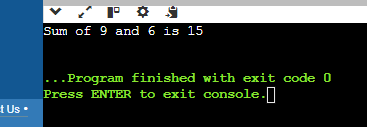
int y = 6;

cout << "Sum of " << x << " and " << y << " is " << sum(x, y) << endl;

return 0;

}

OUTPUT:



2. Capture by Value: Write a lambda that captures an integer by value from the enclosing scope, squares it, and returns the result.

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter an integer: ";

cin >> num;

auto square = [num]() {

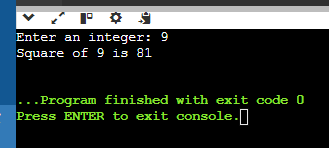
return num \* num; };

cout << "Square of " << num << " is " << square() << endl;

return 0;

}

OUTPUT:



3. Capture by Reference: Create a lambda that captures a string by reference, appends a fixed prefix, and returns the modified string.

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "world";

auto addPrefix = [&str]() {

str = "Hello, " + str;

return str;

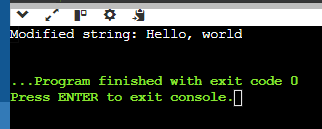
};

cout << "Modified string: " << addPrefix() << endl;

return 0;

}

OUTPUT:



4. Multiple Captures: Construct a lambda that captures two variables (an integer and a boolean) by value and performs a conditional operation based on the boolean value.

#include <iostream>

using namespace std;

int main() {

int num = 10;

bool flag = true;

auto conditionalOperation = [num, flag]() -> int {

if (flag) {

return num \* 2; // If flag is true, double the number

} else {

return num / 2; // If flag is false, halve the number }

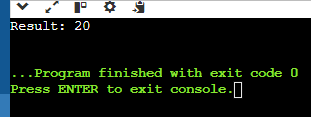
};

cout << "Result: " << conditionalOperation() << endl;

return 0;

}

OUTPUT:



TASK – 2

1. Implicit Casting: Write a program that declares an int variable a with the value 10 and a float variable b with the value 3.14. Then, perform the division a / b and print the result. Explain how implicit casting works in this scenario.

#include <iostream>

using namespace std;

int main() {

int a = 9;

float b = 5.67;

float result = static\_cast<float>(a) / b;

cout << "Result of " << a << " / " << b << " is " << result << endl;

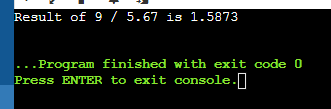
return 0;

}

Explicit Casting and Division Operation:

* Before performing the division, we explicitly cast a to a float using static\_cast<float>(a).
* This ensures that the division operation involves two floating-point numbers: 10.0 (cast from a) and 3.14 (value of b).
* The division 10.0 / 3.14 is performed using floating-point arithmetic.

OUTPUT:



2. Explicit Casting - Data Loss: Declare an int variable x with the value 256 and a char variable y. Assign the value of x to y using explicit casting. Print the value of y. Discuss the data loss that might occur and how to avoid it if necessary.

#include <iostream>

using namespace std;

int main() {

int x = 256;

char y = static\_cast<char>(x);

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

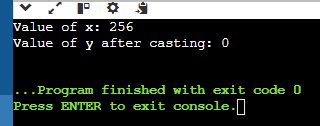
cout << "Value of y after casting: " << static\_cast<int>(y) << endl;

return 0;

}

* To avoid data loss, ensure that the value being cast fits within the range of the target type. In this case, ensure that x is within the range of char (-128 to 127 for signed char, 0 to 255 for unsigned char). If the value exceeds this range, handle it accordingly, for example, by using a different data type or handling the overflow explicitly.

OUTPUT:



3. Explicit Casting - Range Conversion: Declare a double variable d with the value 123.456. Use explicit casting to convert d to an int variable i and print i. Explain the behavior when converting from a larger range to a smaller one.

#include <iostream>

using namespace std;

int main() {

double a = 93.506;

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

cout << "Value of d: " << a << endl;

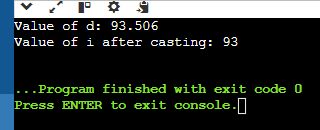
cout << "Value of i after casting: " << i << endl;

return 0; }

Range and Precision: double can represent a wide range of values, including fractional (decimal) values and int can only represent whole numbers (integers). When converting a double to an int, the conversion cannot preserve the fractional part because int does not support it.

Truncation: The conversion process truncates the fractional part of the double value.

OUTPUT:



4. Casting Pointers - Same Type: Declare an int variable num and an int pointer ptr initialized with the address of num. Cast ptr to a float pointer fPtr using explicit casting. Is this casting safe? Why or why not?

#include <iostream>

using namespace std;

int main() {

int num = 42;

int\* ptr = &num;

float\* fPtr = reinterpret\_cast<float\*>(ptr);

cout << "Value of num: " << num << endl;

cout << "Address of num: " << ptr << endl;

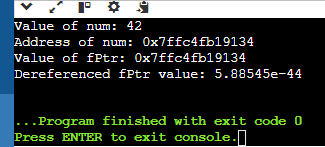
cout << "Value of fPtr: " << fPtr << endl;

cout << "Dereferenced fPtr value: " << \*fPtr << endl;

return 0; }

Casting pointers to different types using reinterpret\_cast is generally not safe. Pointers to different types have different memory layouts, and treating one type as another can lead to undefined behavior.

OUTPUT:



5. Casting Pointers - Different Types: Declare an int variable num and a float variable fval. Initialize an int pointer intPtr with the address of num and a float pointer floatPtr with the address of fval. Can you safely cast intPtr to floatPtr? Explain.

#include <iostream>

using namespace std;

int main() {

int num = 39;

float fval = 9.67;

int\* intPtr = &num;

float\* floatPtr = &fval;

float\* unsafeFloatPtr = reinterpret\_cast<float\*>(intPtr);

cout << "Value of num: " << num << endl;

cout << "Value of fval: " << fval << endl;

cout << "Address of num (intPtr): " << intPtr << endl;

cout << "Address of fval (floatPtr): " << floatPtr << endl;

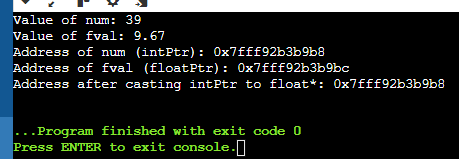
cout << "Address after casting intPtr to float\*: " << unsafeFloatPtr << endl;

return 0;

}

float\* unsafeFloatPtr = reinterpret\_cast<float\*>(intPtr); performs an unsafe cast from an int pointer to a float pointer using reinterpret\_cast. reinterpret\_cast allows the conversion, but it does not ensure that the resulting pointer will be valid or meaningful when dereferenced.

OUTPUT:



6. Casting References - Same Type: Declare an int variable x and an int reference refX assigned to x. Cast refX to a float reference refF. What happens in this case?

#include <iostream>

using namespace std;

int main() {

int x = 25;

int& refX = x;

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

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

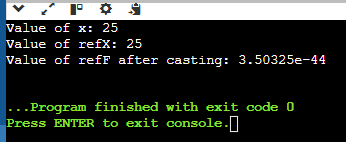
cout << "Value of refX: " << refX << endl;

cout << "Value of refF after casting: " << refF << endl;

return 0;

}

OUTPUT:



7. Casting References - Different Types: Declare an int variable x and a float variable f. Initialize an int reference refX with x. Can you cast refX to refer to f? Why or why not?

No, you cannot cast an int reference refX to refer to a float variable f.

References are not convertible between different types, and the types of the variables being referenced are different (int vs float). Casting refX to refer to f would be like trying to make a square peg fit into a round hole - the memory layouts and types are incompatible.

The compiler will error, something like: "cannot bind 'float' lvalue to 'int' reference".

To reference the float variable f, you need to create a new float reference, like: float& refF = f;. #include <iostream>

using namespace std;

int main() {

int x = 5;

float f = 3.14;

int& refX = x; // refX references x

// Error: cannot bind 'float' lvalue to 'int' reference

// float& refF = (float&) refX; // Don't do this!

// Instead, create a new float reference that references f

float& refF = f; // refF references f

return 0; }

8. Challenge: Area Calculation (Implicit vs. Explicit): Write two functions to calculate the area of a rectangle. One function should take two int arguments for width and height and return an int area. The other function should take two double arguments and return a double area. Discuss the implications of using implicit and explicit casting in these functions.

Implicit Casting

#include <iostream>

int area\_int(int w, int h) {

return w \* h; // implicit casting to int }

double area\_double(double w, double h) {

return w \* h; // implicit casting to double }

int main() {

int width = 5;

int height = 3;

std::cout << "Area (int): " << area\_int(width, height) << std::endl;

double width\_double = 5.5;

double height\_double = 3.3;

std::cout << "Area (double): " << area\_double(width\_double, height\_double) << std::endl;

return 0; }

Explicit Casting

#include <iostream>

int area\_int\_explicit(int w, int h) {

return static\_cast<int>(w \* h); // explicit casting to int

}

double area\_double\_explicit(double w, double h) {

return static\_cast<double>(w \* h); // explicit casting to double }

int main() {

int width = 5;

int height = 3;

std::cout << "Area (int): " << area\_int\_explicit(width, height) << std::endl;

double width\_double = 5.5;

double height\_double = 3.3;

std::cout << "Area (double): " << area\_double\_explicit(width\_double, height\_double) << std::endl;

return 0;

} The implications of using implicit casting are:

- In the area\_int function, if the result of w \* h exceeds the range of int, it will be truncated, leading to incorrect results.

- In the area\_double function, the result of w \* h will be promoted to double automatically, which may lead to precision loss if the original values were integers.

The implications of using explicit casting are:

- In the area\_int\_explicit function, the result of w \* h is explicitly cast to int, which ensures that the result is truncated to an integer value, avoiding potential precision issues.

- In the area\_double\_explicit function, the result of w \* h is explicitly cast to double, which ensures that the result is promoted to a double value, preserving precision.

9. Challenge: Temperature Conversion (Casting and Rounding): Create a program that takes a temperature in Celsius as input from the user. Use explicit casting and appropriate rounding techniques to convert it to Fahrenheit and print the result.

#include <iostream>

#include <cmath>

int main() {

double celsius;

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

std::cin >> celsius;

double fahrenheit = (celsius \* 9.0 / 5.0) + 32.0;

fahrenheit = std::round(fahrenheit \* 100.0) / 100.0; // Round to two decimal places

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

return 0;

}

10. Challenge: Pointer Arithmetic with Casting (Safe vs. Unsafe): Demonstrate safe and unsafe pointer arithmetic with casting. Explain the potential consequences of unsafe pointer manipulation.

Unsafe Pointer Arithmetic with Casting

Unsafe pointer arithmetic involves casting pointers to different types and manipulating them without proper bounds checking. Here's an example:

#include <iostream>

using namespace std;

int main() {

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

int \*ptr = arr;

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

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

cout << "Element " << i << ": " << static\_cast<int>(\*charPtr) << endl;

charPtr++; }

return 0;

}

OUTPUT:

Element 0: 1

Element 1: 0

Element 2: 0

Element 3: 0

Element 4: 2

Explanation of Unsafe Example:

In this example, ptr is a pointer to an array of integers.

charPtr is created by casting ptr to char\*, effectively treating the memory as an array of characters rather than integers.

The loop iterates over the memory locations, interpreting them as characters (char), but printing them as integers.

Safe Pointer Arithmetic

Safe pointer arithmetic involves using pointers within their intended types and ensuring bounds checking. Here’s an example of safe pointer arithmetic:

#include <iostream>

using namespace std;

int main() {

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

int \*ptr = arr;

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

std::cout << ptr[i] << " "; }

cout << endl;

return 0;

}

OUTPUT:

1 2 3 4 5

Explanation of Safe Example:

ptr is a pointer to an array of integers (int\*).

The loop iterates over the array using safe pointer arithmetic (ptr[i]), accessing each element of the array.

Potential Consequences of Unsafe

Undefined Behavior: This code demonstrates undefined behavior because it accesses memory in a manner inconsistent with its original type (int accessed as char). This violates strict aliasing rules and can lead to unpredictable results.

Memory Corruption: Incorrect casting and pointer arithmetic can corrupt memory if the program writes to memory locations not intended for the type it assumes.

Portability Issues: Reliance on such unsafe practices can lead to code that behaves differently on different platforms or compilers.