**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 result = sum(5, 3);

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

return 0;

}

**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 = 5;

auto square = [num]() {

return num \* num;

};

int result = square();

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

return 0;

}

**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>

int main()

{

std::string str = "World";

auto addPrefix = [&str]()

{

str = "Hello, " + str;

return str;

};

std::string result = addPrefix();

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

return 0;

}

**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 number = 42;

bool flag = true;

auto conditionalOperation = [number, flag]()

{

if (flag)

{

return number \* 2;

}

else

{

return number / 2;

}

};

int result = conditionalOperation();

std::cout << "The result of the conditional operation is: " << result << std::endl;

return 0;

}

**Type casting:**

#include <iostream>

using namespace std;

int main()

{

double a = 21.09399;

float b = 10.20;

int c ;

c=(int)b;

cout<<c<<endl;

}

**Implicit Type conversion :**

#include<iostream>

using namespace std;

int main()

{

int x = 10;

char y= 'a';

x= x+y;

float z= x+1.0;

cout<<"x = "<< x <<endl;

cout<<"y = "<< y <<endl;

cout<<"z = "<< z <<endl;

return 0;

}

**Explicit type conversion:**

#include<iostream>

using namespace std;

int main()

{

double x = 1.2;

int sum = (int)x+1;

cout<<"sum = "<<sum;

return 0;

}

**CONVERSION USING CAST OPERATOR :**

#include<iostream>

using namespace std;

int main()

{

float f =3.5;

int b = static\_cast<int>(f);

cout<< b;

return 0;

}

**CONST CAST:**

#include<iostream>

int main()

{

const int value= 30;

int\* writable\_value = const\_cast<int\*>(&value);

\*writable\_value = 20;

std::cout<< value <<std::endl;

return 0;}

**Dynamic\_cast (expr)**

#include<iostream>

#include<typeinfo>

class base

{

public:

virtual void whoami{

std::cout<<"I am Base class object\n";

}

};

class derived:public base

{

public:

void whoami() override

{

std:: cout<<"I am a Derived class object\n";

}

};

int main()

{

Base\* base\_ptr = new derived;

Derived\* derived\_ptr = dynamic\_cast<"Derived".>(base\_ptr);

if(derived\_ptr!= nullptr){

derived\_ptr=>whoami();

}

else

{

std::cout<<"cast failed: Base object is not actually derived\n";

}

delete base\_ptr;

return 0;

}

**REINTERPRET CAST (EXP):**

#include<iostream>

using namespace std;

int main()

{

int value = 10;

float float\_ptr = reinterpret\_cast<float>(&value);

std::cout << \*float\_ptr << std::endl;

}

**DYNAMIC CAST:**

#include<iostream>

#include<typeinfo>

using namespace std;

class Base

{

public:

virtual void whoamI(){

std::cout << "I am Base class object\n" << std::endl;

}

};

class Derived : public Base{

public :

void whoamI() override {

std::cout << "I am Derived class object" << std::endl;

}

};

int main(){

double num = 3.124434;

int integer\_part = static\_cast<int>(num);

std::cout << "Original number : " << num<< std::endl;

std::cout << "Integer part : " << integer\_part << std::endl;

Base \*base\_ptr;

Derived\* derived\_ptr = static\_cast<Derived\*>(base\_ptr);

if(dynamic\_cast<Derived\*>(base\_ptr) != nullptr){

derived\_ptr = static\_cast<Derived\*>(base\_ptr);

derived\_ptr->whoamI();

}else{

std::cout << "Cast Failed : Base class object is not actually Derived" << std::endl;

}

Base\* actual\_derived\_ptr = new Derived;

derived\_ptr = dynamic\_cast<Derived\*>(actual\_derived\_ptr);

if(derived\_ptr != nullptr){

derived\_ptr -> whoamI();

}else{

std::cout << "Cast Failed : Base class object is not actually Derived" << std::endl;

}

delete actual\_derived\_ptr;

int value = 10;

float\* float\_ptr = reinterpret\_cast<float\*>(&value);

return 0;

}

10 QUESTIONS/TASKS:

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>

int main() {

int a = 10;

float b = 3.14;

// Perform the division

float result = a / b;

// Print the result

std::cout << "The result of the division is: " << result << std::endl;

return 0;

}

1. 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>

int main() {

int x = 256;

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

// Print the value of y

std::cout << "The value of y is: " << static\_cast<int>(y) << std::endl;

return 0;

}

1. 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>

int main() {

double d = 123.456;

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

// Print the value of i

std::cout << "The value of i is: " << i << std::endl;

return 0;

}

1. 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>

int main() {

int num = 42;

int\* ptr = &num;

// Cast the int pointer to a float pointer

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

// Print the values

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

std::cout << "Value at ptr (int\*): " << \*ptr << std::endl;

std::cout << "Value at fPtr (float\*): " << \*fPtr << std::endl;

return 0;

}

1. 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

EXAMPLE1: #include <iostream>

int main() {

int num = 42;

float fval = 3.14f;

int\* intPtr = &num;

float\* floatPtr = &fval;

// Attempt to cast intPtr to floatPtr

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

// Print values

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

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

std::cout << "Value at intPtr (int\*): " << \*intPtr << std::endl;

std::cout << "Value at castedPtr (float\*): " << \*castedPtr << std::endl;

return 0;

}

EXAMPLE 2:

#include <iostream>

int main() {

int num = 42;

float fval = 3.14;

int\* intPtr = &num;

float\* floatPtr = &fval;

// Attempt to cast intPtr to floatPtr

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

// Print the original and casted pointer values

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

std::cout << "Value at intPtr (int\*): " << \*intPtr << std::endl;

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

std::cout << "Value at floatPtr (float\*): " << \*floatPtr << std::endl;

std::cout << "Value at castedPtr (float\*): " << \*castedPtr << std::endl;

return 0;

}

1. 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>

int main() {

int x = 42;

int& refX = x;

// Attempt to cast refX to a float reference

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

// Print the values

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

std::cout << "Value through refX (int&): " << refX << std::endl;

std::cout << "Value through refF (float&): " << refF << std::endl;

return 0;

}

1. 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?

#include <iostream>

int main() {

int x = 42;

float f = 3.14;

int& refX = x;

// Attempt to cast refX to refer to f

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

// Print the values

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

std::cout << "Value through refX (int&): " << refX << std::endl;

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

std::cout << "Value through refF (float&): " << refF << std::endl;

return 0;

}

1. 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.

#include <iostream>

// Function to calculate area with int arguments

int calculateAreaInt(int width, int height) {

return width \* height;

}

// Function to calculate area with double arguments

double calculateAreaDouble(double width, double height) {

return width \* height;

}

int main() {

int widthInt = 5;

int heightInt = 10;

double widthDouble = 5.5;

double heightDouble = 10.1;

// Calculate area using int arguments

int areaInt = calculateAreaInt(widthInt, heightInt);

std::cout << "Area (int): " << areaInt << std::endl;

// Calculate area using double arguments

double areaDouble = calculateAreaDouble(widthDouble, heightDouble);

std::cout << "Area (double): " << areaDouble << std::endl;

// Implicit casting: using int function with double values

int areaImplicitCast = calculateAreaInt(widthDouble, heightDouble);

std::cout << "Area (implicit cast to int): " << areaImplicitCast << std::endl;

// Explicit casting: using double function with int values

double areaExplicitCast = calculateAreaDouble(static\_cast<double>(widthInt), static\_cast<double>(heightInt));

std::cout << "Area (explicit cast to double): " << areaExplicitCast << std::endl;

return 0;

}

1. 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> // For rounding functions

int main() {

// Declare a variable to store the temperature in Celsius

double tempCelsius;

// Prompt the user for input

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

std::cin >> tempCelsius;

// Convert Celsius to Fahrenheit

double tempFahrenheit = (tempCelsius \* 9.0 / 5.0) + 32.0;

// Round the Fahrenheit temperature to the nearest whole number

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

// Print the result

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

return 0;

}

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

#include <iostream>

int main() {

// Safe Pointer Arithmetic

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

int\* pInt = arr;

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

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

std::cout << "Value at pInt + " << i << ": " << \*(pInt + i) << std::endl;

}

// Unsafe Pointer Arithmetic with Casting

char\* pChar = reinterpret\_cast<char\*>(pInt);

std::cout << "\nUnsafe Pointer Arithmetic with Casting:" << std::endl;

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

std::cout << "Value at pChar + " << i << ": " << static\_cast<int>(\*(pChar + i)) << std::endl;

}

return 0;

}

STL:

#include<iostream>

#include<vector>

using namespace std;

int main(){

vector<int> vec;

int i ;

std::cout << "vector size" << vec.size() << std::endl;

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

vec.push\_back(i);

}

std::cout << "Extented vector size : "<<vec.size() << std::endl;

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

std::cout << vec[i] << std::endl;

}

vector<int>:: iterator v = vec.begin() ;

while(v != vec.end()){

std::cout << "Value of v is : " << \*v << std::endl;

v++;

}

     return 0;

}

API:

1.VECTOR:

write a program that read integers from the user, sort them and prints the result

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

// Vector to store integers

std::vector<int> numbers;

// Prompt the user to enter integers

std::cout << "Enter integers (enter non-integer to end):\n";

// Read integers from user input

int num;

while (std::cin >> num) {

numbers.push\_back(num);

}

// Sort the numbers

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

// Print the sorted numbers

std::cout << "Sorted numbers:\n";

for (int num : numbers) {

std::cout << num << " ";

}

std::cout << "\n";

return 0;

}