**Date:08/07/2024**

**Day\_11\_Assignments:**

Q.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;

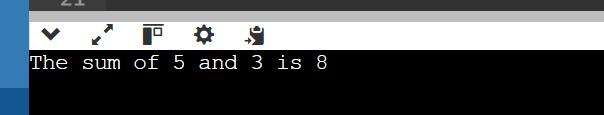
};

int x = 5,y = 3;

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

return 0;

}



Q.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 = 4;

auto square = [num]()

{

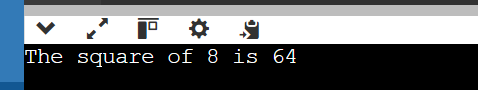
return num \* num;

};

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

return 0;

}



Q.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 = "Programming";

auto addPrefix = [&str]()

{

string prefix = "C++ ";

str = prefix + str;

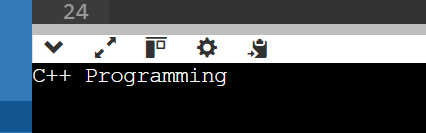
return str;

};

cout << addPrefix() <<endl;

return 0;

}



Q.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 condition = true;

auto conditionalOperation = [num, condition]()

{

if (condition) {

return num \* 8;

} else {

return num / 7;

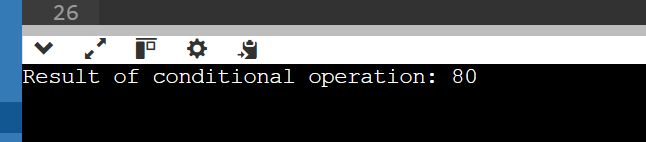
}

};

cout << "Result of conditional operation: " <<conditionalOperation()<<endl;

return 0;

}



Q.5

#include <iostream>

using namespace std;

main()

{

double a =21.09399;

float b=10.20;

int c;

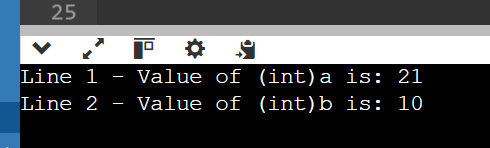
c=(int) a;

cout<<"Line 1 - Value of (int)a is: "<<c<<endl;

c=(int)b;

cout<<"Line 2 - Value of (int)b is: "<<c<<endl;

}



Q.6

#include<iostream>

using namespace std;

int main()

{

double x=1.2;

int sum =(int)x + 1;

cout<<"Sum = "<<sum;

return 0;

}

Q.7

#include <iostream>

using namespace std;

int main()

{

float f=3.5;

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

cout<<b;

}

Q.8

#include <iostream>

using namespace std;

int main()

{

const int value=10;

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

\*writable\_value=20;

cout<<value<<endl;

return 0;

}

Q.9

#include <iostream>

#include<typeinfo>

using namespace std;

class Base{

public:

virtual void whoami(){

cout<<"I am a base class object\n";

}

};

class Derived:public Base{

public:

void whoami() override {

cout<<"I am 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{

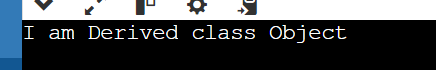
cout<<"cast failed:Base object is not actually derived\n";

}

delete base\_ptr;

return 0;

}



Q.10

#include <iostream>

using namespace std;

int main()

{

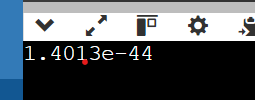
int value=10;

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

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

return 0;

}



Q.11

#include <iostream>

#include<typeinfo>

using namespace std;

class base{

public:

virtual void whoami(){

cout<<"i am a base class object\n";

}

};

class Derived:public base{

public:

void whoami() override{

cout<<"I am a derived class object \n";

}

};

int main()

{

double num=3.14159;

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

cout<<"original number:"<<num<<endl;

cout<<"Integer part:"<<integer\_part<<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{

cout<<"warning: Base object might not be of type derived"<<endl;

}

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

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

if(derived\_ptr != nullptr){

derived\_ptr->whoami();

}else{

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

}

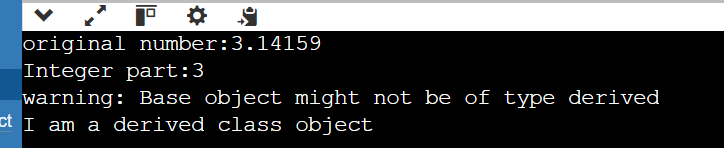
delete actual\_derived\_ptr;

int value=10;

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

return 0;

}



Q.12 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 = 10;

float b = 3.14;

float result = a / b;

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

return 0;

}



Q.13 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() {

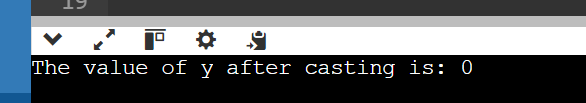
int x = 256;

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

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

return 0;

}



Q.14 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() {

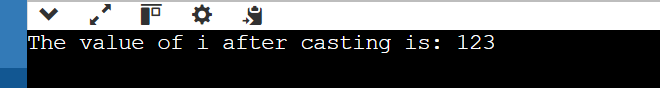
double d = 123.456;

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

cout << "The value of i after casting is: "<< i <<endl;

return 0;

}



Q.15 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;

// Explicitly casting int pointer to float pointer

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

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

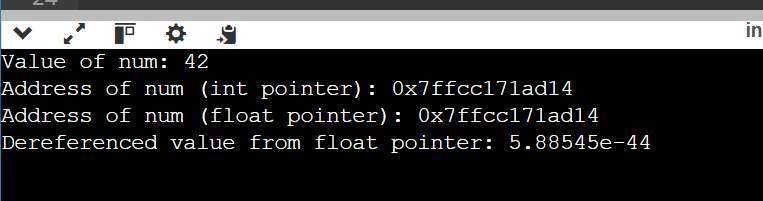
cout << "Address of num (int pointer): " << ptr <<endl;

cout << "Address of num (float pointer): " << fPtr <<endl;

cout << "Dereferenced value from float pointer: " << \*fPtr <<endl;

return 0;

}



Q.16 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 = 42;

float fval = 3.14f;

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 (int pointer): " << intPtr <<endl;

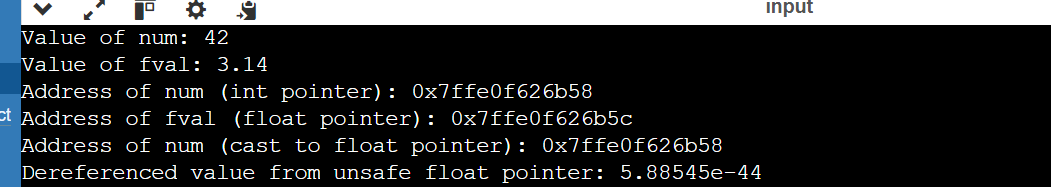
cout << "Address of fval (float pointer): " << floatPtr <<endl;

cout << "Address of num (cast to float pointer): " << unsafeFloatPtr <<endl;

cout << "Dereferenced value from unsafe float pointer: " << \*unsafeFloatPtr <<endl;

return 0;

}



Q.17 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 = 42;

int& refX = x;

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

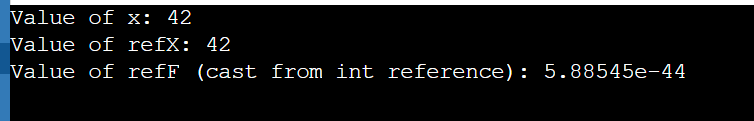
cout << "Value of x: " << x <<endl;

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

cout << "Value of refF (cast from int reference): " << refF <<endl;

return 0;

}



Q.18 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>

using namespace std;

int main() {

int x = 42;

float f = 3.14f;

int& refX = x;

float& refF = f;

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

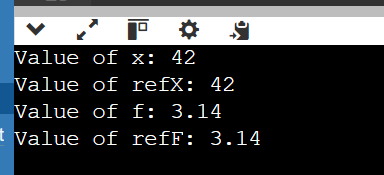
cout << "Value of refX: " << refX <<endl;

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

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

return 0;

}



Q.19 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>

using namespace std;

int calculateArea(int width, int height) {

return width \* height;

}

double calculateArea(double width, double height) {

return width \* height;

}

int main() {

int widthInt = 5;

int heightInt = 10;

double widthDouble = 5.5;

double heightDouble = 10.5;

int areaInt = calculateArea(widthInt, heightInt);

double areaDouble = calculateArea(widthDouble, heightDouble);

cout << "Area with int arguments: " << areaInt << std::endl;

cout << "Area with double arguments: " << areaDouble <<endl;

double implicitCastArea = calculateArea(widthInt, heightInt);

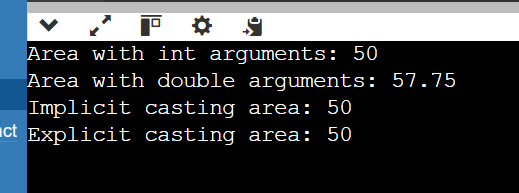
cout << "Implicit casting area: " << implicitCastArea <<endl;

int explicitCastArea = calculateArea(static\_cast<int>(widthDouble), static\_cast<int>(heightDouble));

cout << "Explicit casting area: " << explicitCastArea <<endl;

return 0;

}



Q.20

#include <iostream>

#include <vector>

#include <algorithm> // for std::sort

int main() {

std::vector<int> numbers;

int num;

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

while (std::cin >> num) {

numbers.push\_back(num);

}

// Clear the input stream in case of non-integer input

std::cin.clear();

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

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

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

for (const int& n : numbers) {

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

}

std::cout << std::endl;

return 0;

}