DAY-11 Assignment and LABs

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

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

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

3. 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. */

/* Note - The 'auto' keyword is used to let the compiler automatically

deduce the type of the lambda function.

1. auto x = 10; // The compiler deduces that x is of type int

2. auto y = 3.14; // The compiler deduces that y is of type double

3. auto sum = [](int a, int b) { return a + b; }; // The compiler

deduces the type of the lambda expression. */

```
#include <iostream>

#include <string>

using namespace std;


/*1.  int main() {

    // Basic Lambda fun. sum

    auto sum = [](int a, int b) {  // Define a lambda function that takes two integers, a and b

        return a + b; // Return the sum of a and b


    };
```

```
    // Call the lambda function with arguments 3 and 4, store the result in the variable result
    int result = sum(6, 4);  // result will be 7
    cout << "Sum is : " << result << endl;
    return 0;
}  */

/*2.  int main(){
    // Capture by Value
    int num = 16;
    auto square = [num]() {
        return num * num;

    };
    int Result = square();  // Result will be 256
    cout << "Square of num. ( 16 ) is : " << Result <<endl;
    return 0;

} */

/* 3. int main(){
    // Capture by Reference

    string str = "I am fine.";
    auto appendPrefix = [&str]() {
        str = "Hey, how are you ? \n " + str;
        return str;
```

```
    };
    string modifiedStr = appendPrefix();  // modifiedStr will be "Hello World"
    cout << "Modified String is : " << modifiedStr << endl;

    return 0;
}  */

/*4.  int main(){

    // Multiple Captures
    int num1 = 20;
    bool flag = false;
    auto conditionalOperation = [num1, flag]() {

        return flag ? num1 * 2 : num1 / 2;   // If flag is true, it returns num1 * 2.
                                             // If flag is false, it returns num1 / 2.
    };
    int conditionalResult = conditionalOperation();  // conditionalResult will be 40 because flag is true
    cout << "Conditional Operation Result is : " << conditionalResult << endl;

    return 0;
} */
```

// Topic : Type Casting in C++ .

// **Code - 1 : Type casting Problem**

```
/*int 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 1 - Value of (int)b is : "<<c<<endl;
    return 0;
} */
```

// Type - 1 Implicit Type casting

// bool -> char -> short int -> long ->long long ->float ->double

/* Small data type -> Big data Type => Implicit Typecasting

Big data type -> Small data Type => Emplicit Typecasting */

// Example.1

```
/* int x = 10;  // Integer x
char y = 'a'; // Character y
//Implicit typecasting
x = x+y;
float z = x +1.0;
cout<<"x = "<<x<<endl<<"y = "<<y<<"z = "<<z<<endl;
```

```
return 0;

} */
```

// Type 2 - Explicit typecasting

//Example.2

```
/* int main(){
    double x = 1.2;
    // Explicit conversion from double to int
    int sum = (int)x + 1;
    cout<<"Sum = "<<sum;
    return 0;
} */
```

//Conversion using cast operator

/* Types of cast operator.

1. const_cast<type>(exper.)

2. dynamic_cast<type>(exper.)

3. reinterpret_cast<type>(exper.)

4. static_cast<type>(exper.) */

//Example.3

```
/* int main()
   {
   float f =4.5;
   //using cast operator
   int b = static_cast<int>(f);
   cout <<b<<endl;
   } */
```

// Code - 1 const_cast (expression) -

```
/* int main(){
   const int value = 10; // Constant variable can't be modified
   // Attempt to modify a const variable ( usually discouraged)
   int* writable_value = const_cast<int*>(&value);
   *writable_value = 20;  // Modifying the value through the pointer
   cout<<value<<endl;// Still prints 10 ( undefined behaviour)
   return 0;
   } */
```

// Code - 2 dynamic_cast (expression) -

```
/* 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(){
    Base* base_ptr = new Derived;  // Base pointer pointing to Derived object

    Derived* derived_ptr = dynamic_cast<Derived*>(base_ptr);
    if ( derived_ptr != nullptr) {
        derived_ptr -> whoami();   // Calls Derived class's whoami
    }else{
        cout<<" Cast failed: Base object is not actually Derived\n";
    }
     delete base_ptr;
     return 0;

} */
```

// Code - 3 reinterpret_cast<type>(expression) -

```
/* int main(){
```

```
    int value = 10;

    float* float_ptr = reinterpret_cast<float*>(&value);

    // Accessing the memory as a float ( low- level and potentially

    // risky)

    cout<<*float_ptr<<endl; // Might print garabage depending on memory layout

    return 0;

} */
```

// Code- 4 For dynamic_cast (expression)

```
/* 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() {

    // **static_cast Example ( truncating double to int)**
```

```
    double num = 3.14159;
    int integer_part = static_cast<int>(num); // truncates the decimal
    cout<<"Orginal number is:"<<num<<endl;
    cout<<"Integer part is:"<<integer_part<<endl;
    //**Incorrecting upcasting (assuming Derived objects but nor gurantee)**
    //This could lead to undefined behaviour if base_ptr doesn't point to derived
    Base* base_ptr; // Pointer to Base class ( unknown actual type)
    Derived* derived_ptr = static_cast<Derived*>(base_ptr);
    //Safer approach: check the actual type before casting
    if( dynamic_cast<Derived*>(base_ptr)!=nullptr){
        derived_ptr = static_cast<Derived*>(base_ptr); // Dowmcast only if safe
        derived_ptr->whoami(); // Call Derived class's whoami ( assuming valid downcast)
    }else{
        cout<<"Warning: Base object might not be of type Derived.\n";
    }
    // **dynamic_cast example ( safe downcasting with runtime check)**
Base* actual_derived_ptr = new Derived;
derived_ptr = dynamic_cast<Derived*>(actual_derived_ptr);
if(derived_ptr!=nullptr){
    derived_ptr->whoami();// Safe to call Derived's whoami
}else{
    cout<<"Cast failed: Base object is not actually derived\n";
}

delete actual_derived_ptr; // Release memory
// **reinterpret_cast example ( low_level casting, use with cautions)**
```

```
int value = 10;

float* float_ptr = reinterpret_cast<float*>(&value);

// Accessing memory as a float (undefined behaviour if not careful)

// cout << *float_ptr << endl;  // Not recommended, might print garabage.

return 0;

} */
```

// ** Assignment - 2 | Time - 2:30 p.m.**

/* Code- 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. */

```
/* int main ()

{

  int a = 10;

  float b = 3.14;

  float value = a * b;                    // Implicit casting from int to float

  cout << "The Value of a * b: " << value << endl;

  return 0;

} */
```

/* Code- 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. */

```
/* int main() {

    int x = 256;

    char y = static_cast<char>(x); // Explicit casting from int to char
```

```
    cout << "Value of y: " << static_cast<int>(y) << endl;

    return 0;

} */
```

/* Code- 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. */

```
/* int main() {

    double d = 234.56;

    int i = static_cast<int>(d); // Explicit casting from double to int

    cout << "Value of 'i' is: " << i << endl;

    return 0;

} */
```

/* 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? */

```
 /* int main() {

    int num = 80;

    int* ptr = &num;

    float* fPtr = reinterpret_cast<float*>(ptr); // Explicit casting from int* to float*

    cout << "Value of *fPtr: " << *fPtr << endl; // Unsafe operation

    return 0;

} */
```

/* 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. */

```
/* int main() {
    int num = 42;
    float fval = 3.14f;
    int* intPtr = &num;
    float* floatPtr = &fval;

    // Unsafe: Casting intPtr to floatPtr
    floatPtr = reinterpret_cast<float*>(intPtr);
    cout << "Value of *floatPtr: " << *floatPtr << endl; // Unsafe operation
    return 0;
} */
```

/* 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? */

```
/* int main() {
    int x = 42;
    int& refX = x;

    // Unsafe and invalid: Casting int& to float&
    // float& refF = reinterpret_cast<float&>(refX);
     return 0;
} */
```

/* 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? */

```
/* int main() {
    int x = 42;
    float f = 3.14f;
    int& ref = x;

    // Unsafe and invalid: Casting int& to refer to float
    // int& ref = reinterpret_cast<int&>(f); // Compilation error
    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.*/

```
// Function to calculate area with int arguments
/* int area(int width1, int height1) {
    return width1 * height1;
}

// Function to calculate area with double arguments
double area(double width2, double height2) {
    return width2 * height2;
}
```

```
int main() {
    int width1 = 5, height1 = 10;
    double width2 = 7.5, height2 = 11.5;

    cout << "Area of figure in (int) conversion is: " << area(width1, height1) << endl; // Output: 50
    cout << "Area of figure in (double) conversion is: " << area(width2, height2) << endl; // Output: 57.75

    return 0;
} */
```

/* 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 <cmath>

int main() {
    double celsius;
    cout << "Enter Temperature in Celsius ? ";
    cin >> celsius;

    double fahrenheit = celsius * 9.0 / 5.0 + 32.0;
    int roundedFahrenheit = static_cast<int>(round(fahrenheit));

    cout << "Temperature in Fahrenheit (rounded) is : " << roundedFahrenheit << 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. */

```
/* int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int* intPtr = arr;

    // Safe pointer arithmetic
    cout << "Safe pointer arithmetic:" << endl;
    for (int i = 0; i < 5; ++i) {
        cout << *(intPtr + i) << " ";
    }
    cout << endl;

    // Unsafe pointer arithmetic with casting
    cout << "Unsafe pointer arithmetic:" << endl;
    char* charPtr = reinterpret_cast<char*>(intPtr);
    for (int i = 0; i < 5 * sizeof(int); ++i) {
        cout << *(charPtr + i) << " ";
    }
    cout << endl;

    return 0;
} */
```

// **Code - 5 Program on Concept of Vector**

```
#include <vector>


/* int main() {
    // Create a vector to store list
    vector<int> vec;
    int i;


    // Display the original size of vec
    cout << "Vector size = " << vec.size() << endl;


    // Push 5 values into the vector
    for(i = 0; i < 5; i++) {
        vec.push_back(i);
    }


    // Display extended size of vec
    cout << "Extended vector size = " << vec.size() << endl;


    // Access and display 5 values from the vector
    for (i = 0; i < 5; i++) {
        cout << "Value of vec[" << i << "] = " << vec[i] << endl;
    }
    // Use iterator to access the values
    cout << "Using iterator to access the values:" << endl;
    vector<int>::iterator v = vec.begin();
```

```
    while(v != vec.end()) {
        cout << "Value of vec = " << *v << endl;
        v++;
    }
    return 0;
} */
```

// Code - 6 Program on concept of Lists

```
#include <list>
// Function to display the elements of a list
void showlist(list<int> g) {
    for (list<int>::iterator it = g.begin(); it != g.end(); ++it)
    {
        cout << '\t' << *it;
    }
    cout << '\n';
}
int main() {
    list<int> gqlist1, gqlist2;
    for (int i = 0; i < 10; i++) {
        gqlist1.push_back(i * 2);
        gqlist2.push_back(i * 3);

    }
```

```
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front(): " << gqlist1.front();
    cout << "\ngqlist1.back(): " << gqlist1.back();

    cout << "\ngqlist1.pop_front(): ";
    gqlist1.pop_front();
    showlist(gqlist1);

    cout << "\ngqlist2.pop_back(): ";
    gqlist2.pop_back();
    showlist(gqlist2);

    cout << "\ngqlist1.reverse(): ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);
    return 0;
}
```