Day-10:

File Processing: Design a base class File with a virtual function readData() that has an empty body. Create derived classes like TextFile and ImageFile inheriting from File and overriding readData() with their specific reading procedures. Implement a function that takes a pointer to File as input, attempts to read the data using the readData() function, and handles potential errors based on the actual derived class type (e.g., different file formats).

#include <iostream>

#include <stdexcept>

class File {

public:

virtual void readData() = 0;

virtual ~File() {}

};

class TextFile : public File {

public:

void readData() override {

std::cout << "Reading text file...\n";

throw std::runtime\_error("Text file reading error");

}

};

class ImageFile : public File {

public:

void readData() override {

std::cout << "Reading image file...\n";

throw std::runtime\_error("Image file reading error");

}

};

void processFile(File \*file) {

try {

file->readData();

}

catch (const std::runtime\_error& e) {

std::cerr << "Error reading file: " << e.what() << std::endl;

}

catch (...) {

std::cerr << "Unknown error occurred while reading file." << std::endl;

}

}

int main() {

TextFile textFile;

ImageFile imageFile;

std::cout << "Processing text file...\n";

processFile(&textFile);

std::cout << "\n";

std::cout << "Processing image file...\n";

processFile(&imageFile);

return 0;

}

Output:

Processing text file...

Reading text file...

Error reading file: Text file reading error

Processing image file...

Reading image file...

Error reading file: Image file reading error

Design an abstract factory class hierarchy to create different families of products (e.g., furniture). Use pointers and runtime polymorphism. Define an abstract base class FurnitureFactory with a virtual function createChair(). Create derived classes like ModernFurnitureFactory and ClassicFurnitureFactory that override createChair() to return pointers to concrete chair objects specific to their style. Utilize the factory pattern with runtime polymorphism to allow for flexible furniture creation based on user choice

#include <iostream>

class Chair {

public:

virtual void sitOn() const = 0;

virtual ~Chair() {}

};

class ModernChair : public Chair {

public:

void sitOn() const override {

std::cout << "Sitting on a modern chair.\n";

}

};

class ClassicChair : public Chair {

public:

void sitOn() const override {

std::cout << "Sitting on a classic chair.\n";

}

};

class FurnitureFactory {

public:

virtual Chair\* createChair() = 0;

virtual ~FurnitureFactory() {}

};

class ModernFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ModernChair();

}

};

class ClassicFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ClassicChair();

}

};

int main() {

FurnitureFactory \*factory = nullptr;

Chair \*chair = nullptr;

bool isModern = true;

if (isModern) {

factory = new ModernFurnitureFactory();

} else {

factory = new ClassicFurnitureFactory();

}

chair = factory->createChair();

chair->sitOn();

delete chair;

delete factory;

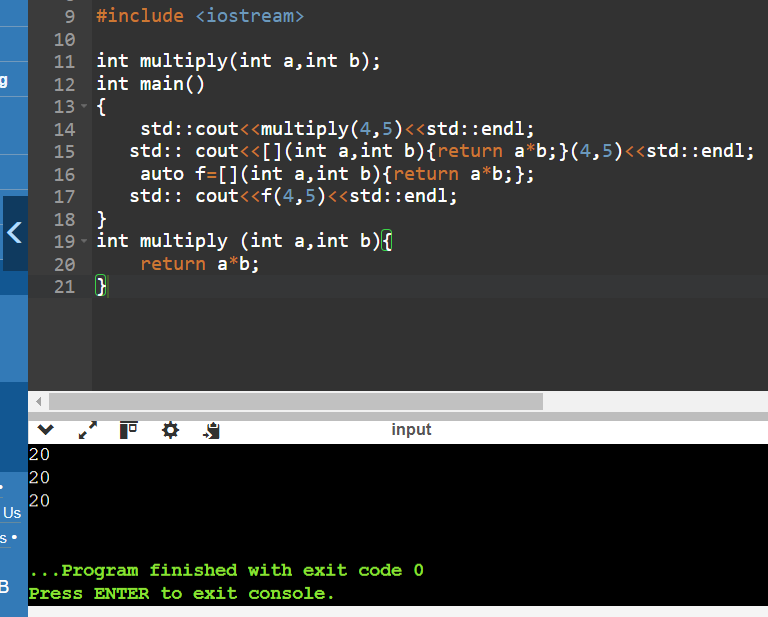
return 0;

}

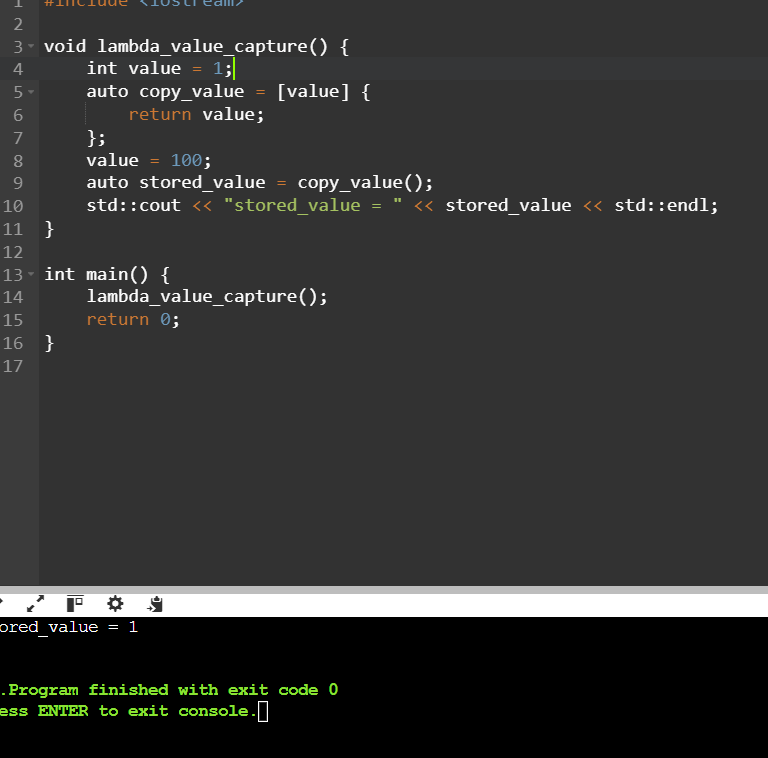
Output:

Sitting on a modern chair.

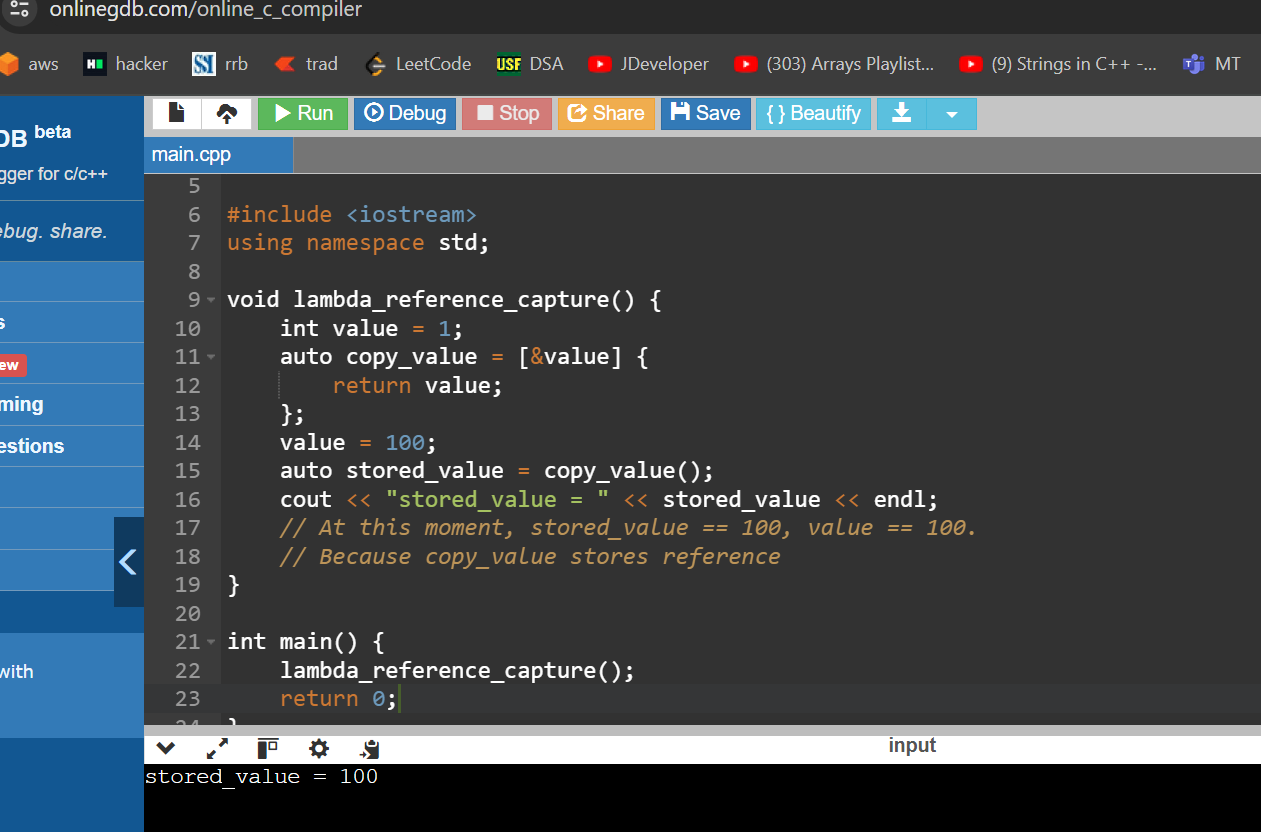
LAMBDA EXPRESSION EXAMPLE:



CAPTURE BY VALUE:



CAPTURE BY REFERENCE:



#include<iostream>

using namespace std;

int main()

{

int m=0;

int n=0;

[&,n](int a) mutable {m=++n +a;}(4);

cout<<m<<endl<<n<<endl;

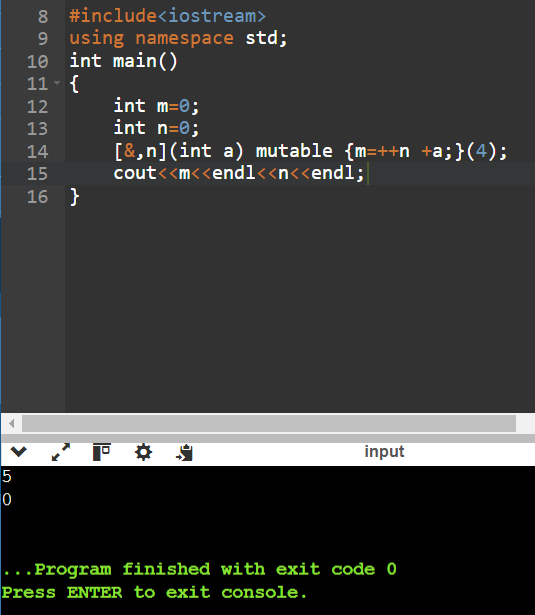
}

Output:

5

0

INLINE FUNCTION BY BOTH(Refernce and Value):



Practice Problem Statement:

Scenario: You're working on a data analysis project where you need to filter a list of integers based on whether they are even or odd. You want to use a lambda expression to achieve this filtering.

Task:

Define a function named filter\_even\_odds that takes two arguments:

const std::vector<int>& numbers: The vector containing the integer values.

bool is\_even: A flag indicating whether to filter even (true) or odd (false) numbers.

Inside the function, use a lambda expression to iterate through the numbers vector.

Within the lambda, check if the current number is even using the modulo operator (%).

If the even/odd condition matches the is\_even flag, add the number to a new filtered vector.

Return the filtered vector from the filter\_even\_odds function.

#include <iostream>

#include <vector>

std::vector<int> filter\_even\_odds(const std::vector<int>& numbers, bool is\_even) {

std::vector<int> filtered;

auto lambda = [&](int num) {

if ((num % 2 == 0 && is\_even) || (num % 2 != 0 && !is\_even)) {

filtered.push\_back(num);

}

};

for (int num : numbers) {

lambda(num);

}

return filtered;

}

int main() {

std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> even\_numbers = filter\_even\_odds(numbers, true);

std::cout << "Even Numbers: ";

for (int num : even\_numbers) {

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

}

std::cout << "\n";

std::vector<int> odd\_numbers = filter\_even\_odds(numbers, false);

std::cout << "Odd Numbers: ";

for (int num : odd\_numbers) {

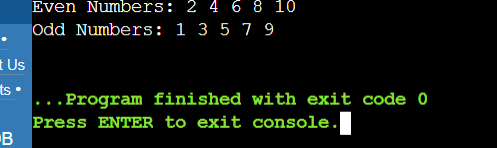
std::cout << num << " ";

}

std::cout << "\n";

return 0;

}



2. Finding Maximum Value:

Scenario: You have a list of objects and want to find the object with the highest value based on a specific criterion.

Task:

Define a function named find\_max that takes two arguments:

const std::vector<T>& objects: The vector containing the objects (can be any type T).

std::function<bool(const T& a, const T& b)> compare: A function object (e.g., a lambda) that defines the comparison logic for finding the maximum.

Inside the function, use a std::accumulate with a lambda expression to iterate through the objects vector.

Within the inner lambda, compare the current element with the current maximum using the provided compare function.

If the current element is greater (based on the comparison logic), return it as the new maximum.