SUNNY KUMAR
8 JULY TASK

THIS DOC FILE CONSIST OF LAMBDA FUNCTION AND TYPE CASTING (IMPLICIT AND EXPLICIT) PROGRAM.

/*
Basic Lambda: Define a lambda expression that takes two integers as arguments and returns their sum. Use auto to infer the return type.
Capture by Value: Write a lambda that captures an integer by value from the enclosing scope, squares it, and returns the result.
Capture by Reference: Create a lambda that captures a string by reference, appends a fixed prefix, and returns the modified string.
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>
#include <string>
using namespace std;
int main() {
    auto sum = [](int a, int b) {
        return a + b;
    };
    int result1 = sum(3, 4); // result1 will be 7
    cout << "Sum is: " << result1 << endl;

    int x = 5;
    auto square = [x]() {
        return x * x;
    };
    int squared = square(); // squared will be 25
    cout << "Square: " << squared << endl;
    string str = "rohit";
    auto appendPrefix = [&str]() {
        str = "sunny, " + str;
        return str;
    };
    string modifiedStr = appendPrefix();
    cout << "Modified String: " << modifiedStr << endl;

    int number = 10;
    bool flag = true;
    auto conditionalOperation = [number, flag]() {
        if (flag) {
            return number * 2;
        } else {
            return number / 2;
        }
    };
    int result2 = conditionalOperation(); // result2 will be 20 if flag is true, otherwise
5
    cout << "Conditional Operation Result: " << result2 << endl;

    return 0;
}
```

TYPE CASTING

```
/******************************************************************************

Welcome to GDB Online.
GDB online is an online compiler and debugger tool for C, C++, Python, Java, PHP, Ruby, Perl,
C#, OCaml, VB, Swift, Pascal, Fortran, Haskell, Objective-C, Assembly, HTML, CSS, JS, SQLite, Prolog.
Code, Compile, Run and Debug online from anywhere in world.
```

```
*******************************************************************************/

#include <iostream>
using namespace std;
int main()
{
   int a;
   float b=7.89;
   double d=456.678;
   a=(int)b;
   cout<<"value of b is"<<a<<endl;
   a=(int) d;
   cout<<"value of d is"<<a;

    return 0;
}
```

[2]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
using namespace std;
int main()
```

```
{
   int x=10;
   char c='a';
   x=x+c;
   float f=x+1.0;
   cout<<"x is"<<x<<endl<<"y is"<<c<<" f is"<<f<<endl;

   return 0;
}
```

[3]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
using namespace std;
int main()
{
   double x=1.3;
   int t=(int)x+1;

   cout<<"t is"<<t;

   return 0;
}
```

[4]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
using namespace std;
int main()
{
    float f=3.5;
    int b =static_cast<int> f;
    cout<<"b is "<<b;

    return 0;
}
```

[5]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>

int main()
{
    const int value =10; //constatnt variable
    int* writable_value=const_cast<int*>(&value);//attemp to modify a const
variable
    *writable_value=20;//modify the value through pointer

    std::cout<<value;//still print 10

    return 0;
}
```

[6]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
using namespace std;

class base{
    public:
    virtual void whoami(){
        cout<<"i am a base class object"<<endl;
    }
};
class derived : public base{
    public:
    void whoami() override
    {
        cout<<"i am a derived class object"<<endl;
    }
};
```

```
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 class not derived"<<endl;
   }
   delete base_ptr;


   return 0;
}
```

[7]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>

int main()
{
   int value=10;
   float* float_ptr=reinterpret_cast<float*>(&value);
   std::cout<<float_ptr;
```

```
    return 0;
}
```

[8]

```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
#include <typeinfo> // For dynamic_cast

class Base {
public:
    virtual void whoami() {
        std::cout << "I am a Base class object.\n";
    }
};

class Derived : public Base {
public:
    void whoami() override {
        std::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
    std::cout << "Original number: " << num << std::endl;
    std::cout << "Integer part: " << integer_part << std::endl;

    // Incorrect upcasting (assuming Derived object but not guaranteed)
    // This could lead to undefined behavior if base_ptr doesn't point to a Derived
    Base* base_ptr; // Pointer to a Base class (unknown actual type)
    Derived* derived_ptr = static_cast<Derived*>(base_ptr);

    // Safer approach: check the actual type before downcasting
    if (dynamic_cast<Derived*>(base_ptr) != nullptr) {
        derived_ptr = static_cast<Derived*>(base_ptr); // Downcast only if safe
        derived_ptr->whoami(); // Call Derived class's whoami (assuming valid
downcast)
    } else {
        std::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 {
        std::cout << "Cast failed: Base object is not actually Derived\n";
    }

    delete actual_derived_ptr; // Releasing memory

    // reinterpret_cast example (low-level casting, use with caution)
    int value = 10;
    float* float_ptr = reinterpret_cast<float*>(&value);
    // Accessing memory as a float (undefined behavior if not careful)
    std::cout << "*float_ptr = " << *float_ptr << std::endl; // Not recommended,
might print garbage

    return 0;
}
```

/***Purpose: Casts away the const or volatile qualifier from an expression. This allows modifying a supposedly constant variable, but be cautious as it can break code that relies on const-correctness.**
Use Cases: This is generally discouraged as it can lead to unexpected behavior. However, it might be necessary in rare cases when working with legacy code or APIs that don't handle const correctly.*/

```
#include <iostream>

void modifyConstVariable(const int* ptr) {
   int* nonConstPtr = const_cast<int*>(ptr);
   *nonConstPtr = 42; // Modifying the supposedly constant variable
}

int main() {
   const int x = 10;
   std::cout << "Before modification: " << x << std::endl;

   modifyConstVariable(&x);

   std::cout << "After modification: " << x << std::endl; // The result is undefined
behavior
   return 0;
}
```

/***Purpose: Performs a runtime check to see if a pointer or reference to a base class can be safely cast to a derived class type. If the cast fails (i.e., the object isn't actually of the derived type), it returns nullptr.**
Use Cases: This is particularly useful for working with polymorphism in inheritance hierarchies. It ensures type safety and avoids potential errors from incorrect casting.
*/

```
#include <iostream>

using namespace std;
class Base {
public:
    virtual ~Base() = default;
    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";
    }
};
void identify(Base* base) {
    Derived* derived = dynamic_cast<Derived*>(base);
    if (derived) {
        derived->whoami();
    } else {
        cout << "Base object is not actually Derived.\n";
    }
}
int main() {
    Base* b1 = new Base();
    Base* b2 = new Derived();
    identify(b1);
    identify(b2);
    delete b1;
    delete b2;
    return 0;
}
```

/***

Purpose: Reinterprets the bit pattern of an expression as a different type. This allows casting pointers to different pointer types, converting pointers

to integers and vice versa (low-level operations). However, it's very powerful and can lead to undefined behavior if not used carefully.
Use Cases: This is for advanced scenarios like memory manipulation or interfacing with low-level hardware. Use it with extreme caution as it bypasses type checking.

```
*******************************************************************************/

#include <iostream>

using namespace std;

int main() {
    int value = 10;
    cout << "Original value: " << value << endl;

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

    char* char_ptr = reinterpret_cast<char*>(&value);
    cout << "Bytes of value: ";
    for (int i = 0; i < sizeof(value); ++i) {
        cout << static_cast<int>(char_ptr[i]) << " ";
    }
    cout << endl;

    return 0;
}

/*******************************************************************************
```

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 << "Result of a / b: " << result << endl;
   return 0;
}

/*****************************************************************************
```

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 y: " << static_cast<int>(y) << endl; // Printing as int to see
numeric value
   return 0;
}

/*****************************************************************************
```

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 << "Value of i: " << i << endl;
    return 0;
}

/******************************************************************************
```

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 << "Reinterpreted float pointer address: " << fPtr << endl;
    cout << "Value at float pointer (undefined behavior): " << *fPtr << endl;

    return 0;
}
```

```
/*************************
 * 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 = 3.14;

   int* intPtr = &num;
   float* floatPtr = &fval;

   // Attempt to cast intPtr to floatPtr
   float* castedPtr = reinterpret_cast<float*>(intPtr);

   cout << "Value of num: " << num << endl;
   cout << "Address of num: " << intPtr << endl;
   cout << "Reinterpreted float pointer address: " << castedPtr << endl;
   cout << "Value at reinterpreted float pointer (undefined behavior): " <<
*castedPtr << endl;

   return 0;
}

/*****************************************************************************
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 = 10;
    int &refX = x;

    // Attempting to cast refX to a float reference (which is not allowed)
    float &refF = static_cast<float&>(refX);

    // Outputting the values
    cout << "x: " << x << endl;
    cout << "refF: " << refF << endl;

    return 0;
}

/*****************************************************************************
```

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 = 10;
    float f = 3.14;

    int &refX = x;  // refX is a reference to x
    // Attempting to cast refX to refer to f (not allowed)
    // float &refF = static_cast<float&>(refX);  // This line would cause a
compilation error

    // Outputting the values
    cout << "x: " << x << endl;
    cout << "f: " << f << endl;

    return 0;
}
```

/***

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 static_cast<double>(width) * height;
}

int main() {
    int widthInt = 5, heightInt = 3;
    double widthDouble = 4.5, heightDouble = 2.5;

    int areaInt = calculateArea(widthInt, heightInt);
    double areaDouble = calculateArea(widthDouble, heightDouble);

    cout << "Area using int arguments: " << areaInt << endl;
    cout << "Area using double arguments: " << areaDouble << endl;

    return 0;
}

/*****************************************************************************
```

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>

using namespace std;

double celsiusToFahrenheit(double celsius) {
    return static_cast<double>(celsius) * 9 / 5 + 32;
}

int main() {
    double celsius;

    cout << "Enter temperature in Celsius: ";
    cin >> celsius;

    double fahrenheit = celsiusToFahrenheit(celsius);

    cout << "Temperature in Fahrenheit: " << round(fahrenheit * 100) / 100 <<
endl;

    return 0;
}

/***************************************************************************
```

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

In this question there are two program

1.for safe
2.for unsafe

```
***************************************************************************/

/*
#include <iostream>

using namespace std;

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr;  // Pointer to the beginning of the array

    // Safe pointer arithmetic using array indices
    for (int i = 0; i < 5; ++i) {
        cout << "Element " << i << ": " << *(ptr + i) << endl;
    }

    return 0;
}

*/
```

```
#include <iostream>

using namespace std;

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr;  // Pointer to the beginning of the array

    // Unsafe pointer arithmetic using casting and manipulation
    for (int i = 0; i < 5; ++i) {
        int *unsafePtr = reinterpret_cast<int *>(reinterpret_cast<char *>(ptr) + i *
sizeof(int));
        cout << "Element " << i << ": " << *unsafePtr << endl;
    }

    return 0;
}
```

/*vector in cpp*/

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Create a vector of integers
    vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Accessing elements using index
    cout << "Elements in the vector:" << endl;
    for (size_t i = 0; i < numbers.size(); ++i) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // Using iterators to access elements
    cout << "Elements in the vector using iterators:" << endl;
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // Accessing elements using range-based for loop (C++11 and later)
    cout << "Elements in the vector using range-based for loop:" << endl;
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;

    // Size of the vector
    cout << "Size of the vector: " << numbers.size() << endl;

    return 0;
}
```

/***

list in cpp(STL)

***/

```
#include <iostream>
#include <list>
```

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

using namespace std;

// Function to print elements of a list
void printList(const list<int>& lst) {
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    list<int> myList;

    // Adding elements to the list
    myList.push_back(10);
    myList.push_back(20);
    myList.push_back(30);
    myList.push_back(40);
    myList.push_back(50);

    // Printing elements of the list
    cout << "Original list: ";
    printList(myList);

    // Accessing front and back elements
    cout << "Front element: " << myList.front() << endl;
    cout << "Back element: " << myList.back() << endl;

    // Removing front element
    myList.pop_front();
    cout << "List after pop_front: ";
    printList(myList);

    // Accessing front and back elements again
    cout << "Front element: " << myList.front() << endl;
    cout << "Back element: " << myList.back() << endl;

    // Reversing the list
    myList.reverse();
    cout << "List after reversing: ";
    printList(myList);

    // Sorting the list
    myList.sort();
    cout << "List after sorting: ";
    printList(myList);

    return 0;
}



/******************************************************************************
array sort
*******************************************************************************/

#include <iostream>

using namespace std;
void sort(int arr[], int size)
{
    int temp;
    for(int i=0;i<size;i++)
    {
        for(int j=i+1;j<size;j++)
        {
            if(arr[i]>arr[j])
            {
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
    }
    for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<" ";
    }
}

int main() {
    int size;
    cout<<"enter the array size: "<<endl;
    cin>>size;
    int arr[size];
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
    cout<<"array elements are before sorting: "<<endl;
    for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<" ";
    }
    sort(arr,size);
    return 0;
}
```