**MOUNIKA DAY 11**

**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(3, 4);

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

return 0;

}

OUTPUT:  
The sum is: 7

**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 x = 5;

auto square = [x]() {

return x \* x;

};

int result = square();

cout << "The square of " << x << " is: " << result << endl;

return 0;

}

OUTPUT:  
The square of 5 is: 25

**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 text = "World";

auto appendPrefix = [&text]() {

string prefix = "Hello, ";

text = prefix + text;

return text;

};

string result = appendPrefix();

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

return 0;

}

OUTPUT:

The modified string is: Hello, World

**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 = 10;

bool condition = true;

auto conditionalOperation = [number, condition]() {

if (condition) {

return number \* 2;

} else {

return number / 2;

} };

int result = conditionalOperation();

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

return 0;

}

OUTPUT:

The result of the conditional operation is: 20

**TYPE CASTING:**

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1) Implicit Type

Conversion

1. Explicit Type Conversion.

#include <iostream>

using namespace std;

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

return 0;

}

OUTPUT:

Line 1 - Value of (int)a is :21

Line 2 - Value of (int)b is :10

**IMPLICIT TYPE CONVERSION**

Implicit Type Conversion Also known as 'automatic type conversion'.

Done by the compiler on its own, without any external trigger from the user.

Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.

All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long-> float -> double -> long double

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

EX:

#include <iostream>

using namespace std;

int main() {

int x = 10; // integer x

char y = 'a'; // character y

// y is implicitly converted to int. ASCII value of 'a' is 97

x = x + y;

// x is implicitly converted to float

float z = x + 1.0;

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

<< "y = " << y << endl

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

return 0;

}

OUTPUT:

x = 107

y = a

z = 108

**EXPLICIT TYPE CONVERSION**

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This

can be also considered as forceful casting. Syntax:(type) expression

where type indicates the data type to which the final result is converted.

EX:

#include <iostream>

using namespace std;

int main() {

double x = 1.2;

// Explicit conversion from double to int

int sum = (int)x + 1;

cout << "Sum = " << sum;

return 0;

}

OUTPUT:  
Sum = 2

**CONVERSION USING CAST OPERATOR**

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator.

The most general cast supported by most of the C++ compilers is as follows -

(type) expression

Where type is the desired data type. There are other casting operators supported by C++, they are listed below-

·

**->**const\_cast<type> (expr) - The const\_cast operator is used to explicitly override const and/or volatile in a cast. The target type must be the same as the source type except for the alteration of its const or volatile attributes. This type of casting manipulates the const attribute of the passed object, either to be set or removed.

**->**dynamic\_cast<type> (expr) - The dynamic\_cast performs a runtime cast that verifies the validity of the cast. If the cast cannot be made, the cast fails and the expression evaluates to null. A dynamic\_cast performs casts on polymorphic types and can cast a A\* pointer into a B\* pointer only if the object being pointed to actually is a B object.

**->**reinterpret\_cast<type> (expr) - The reinterpret\_cast operator changes a pointer to any other type of pointer. It also allows casting from pointer to an integer type and vice versa.

**->**static\_cast<type> (expr) - The static\_cast operator performs a nonpolymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer.

All of the above-mentioned casting operators will be used while working with classes and objects. For now, try the following example to understand a simple cast operators available in C++. Copy and paste the

following C++ program in test.cpp file and compile and run this program.

#include<iostream>

Using namespace std;

{

Float f=3.5;

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

Cout<<b;

}

OUTPUT:

3

**EX:1**

#include <iostream>

int main() {

const int value = 10; // Constant variable

// Attempt to modify a const variable (usually discouraged)

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

\*writable\_value = 20; // Modifying the value through the pointer

std::cout << value << std::endl; // Still prints 10 (undefined behavior)

return 0;

}

OUTPUT:

10

**EX:2**

#include <iostream>

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() {

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;

}

OUTPUT:

I am a Derived class object

**EX:3**

#include<iostream>

using namespace std;

int main() {

int value = 10;

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

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

return 0;

}

OUTPUT:

1.4013e-44

**EX:4**

#include <iostream>

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;

// \*\*dynamic\_cast example (assuming Derived object assigned to Base pointer)\*\*

Base\* base\_ptr1 = new Derived;

Derived\* derived\_ptr1 = dynamic\_cast<Derived\*>(base\_ptr1);

if (derived\_ptr1 != nullptr) {

derived\_ptr1->whoami(); // Safe to call Derived's method

} else {

std::cout << "Cast failed: Base object is not of type Derived\n";

}

delete base\_ptr1;

// \*\*dynamic\_cast example (Safe downcasting with runtime check)\*\*

Base\* base\_ptr2 = new Derived;

Derived\* derived\_ptr2 = dynamic\_cast<Derived\*>(base\_ptr2);

if (derived\_ptr2 != nullptr) {

derived\_ptr2->whoami(); // Safe to call Derived's method

} else {

std::cout << "Cast failed: Base object is not actually Derived\n";

}

delete base\_ptr2;

// \*\*reinterpret\_cast example (Low-level casting, use with caution)\*\*

int value = 10;

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

std::cout << "Float value: " << \*float\_ptr << " (Not recommended, might print garbage)\n";

return 0;

}

OUTPUT:

Original number: 3.14159

Integer part: 3

I am a Derived class object

I am a Derived class object

Float value: 1.4013e-44 (Not recommended, might print garbage)

**PRACTICE QUESTONS**

**1. const\_cast (expr)**

**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**

It seems like you're describing the const\_cast operator in C++. This operator is indeed used to cast away the const or volatile qualifiers from pointers or references. It's typically discouraged because it can lead to undefined behavior if misused, especially when attempting to modify a variable that was originally declared as const.

Here's a summary based on your description:

* **Purpose:** Casts away the const or volatile qualifier from an expression.
* **Use Cases:** It's generally discouraged due to potential undefined behavior. However, it might be necessary in scenarios such as interacting with legacy code or APIs that were not designed to handle const correctly.

#include <iostream>

void printString(char\* str) {

std::cout << str << std::endl;

}

int main() {

const char\* constStr = "Hello, World!";

printString(const\_cast<char\*>(constStr));

return 0;

}

OUTPUT:

Hello, World!

**2. dynamic\_cast (expr)**

**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>

class Base {

public:

virtual ~Base() {}

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";

}

};

class AnotherDerived : public Base {

public:

void whoami() override {

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

}

};

void identifyObject(Base\* basePtr) {

if (Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr)) {

derivedPtr->whoami(); // Safe to call Derived's method

} else if (AnotherDerived\* anotherDerivedPtr = dynamic\_cast<AnotherDerived\*>(basePtr)) {

anotherDerivedPtr->whoami(); // Safe to call AnotherDerived's method

} else {

std::cout << "Unknown object type\n";

}

}

int main() {

Base\* base1 = new Derived;

Base\* base2 = new AnotherDerived;

Base\* base3 = new Base;

identifyObject(base1);

identifyObject(base2);

identifyObject(base3);

delete base1;

delete base2;

delete base3;

return 0;

}

OUTPUT:

I am a Derived class object

I am AnotherDerived class object

Unknown object type

**3. reinterpret\_cast (expr)**

**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>

#include <cstdint> // For uintptr\_t

void manipulateMemory(uintptr\_t address) {

int\* intPtr = reinterpret\_cast<int\*>(address);

\*intPtr = 42;

}

int main() {

int value = 10;

std::cout << "Before manipulation: " << value << std::endl;

uintptr\_t address = reinterpret\_cast<uintptr\_t>(&value);

manipulateMemory(address);

std::cout << "After manipulation: " << value << std::endl;

return 0;

}

OUTPUT:

Before manipulation: 10

After manipulation: 42

**4. static\_cast (expr)**

**Purpose: Performs a basic type conversion between compatible types. It's similar to implicit conversions but allows explicit control.**

**Use Cases: This is commonly used for converting between related data types like int to float or casting a base class pointer to a derived class pointer (upcasting). It's generally safe as long as the conversion is valid**

#include <iostream>

using namespace std;

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() {

// Numeric conversion

int intValue = 42;

float floatValue = static\_cast<float>(intValue);

std::cout << "Integer: " << intValue << ", Float: " << floatValue << std::endl;

Derived derivedObj;

Base\* basePtr = static\_cast<Base\*>(&derivedObj);

basePtr->whoami();

Derived& derivedRef = derivedObj;

Base& baseRef = static\_cast<Base&>(derivedRef);

baseRef.whoami();

enum class Color { RED, GREEN, BLUE };

Color color = Color::GREEN;

int colorValue = static\_cast<int>(color);

std::cout << "Enum Color value: " << colorValue << std::endl;

return 0;

}

OUTPUT:

Integer: 42, Float: 42

I am a Derived class object

I am a Derived class object

Enum Color value: 1

1. **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 output of " << a << " / " << b << " is: " << output << endl;

return 0;

}

OUTPUT:

The output of 10 / 3.14 is: 3.18471

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>

using namespace std;

int main() {

int x = 256;

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

cout << "The value of y after casting x (256) to char is: " << static\_cast<int>(y) << endl;

return 0;

}

OUTPUT:

The value of y after casting x (256) to char is: 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>

using namespace std;

int main() {

double d = 123.456;

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

cout << "The value of i after casting d (123.456) to int is: " << i << endl;

return 0;

}

OUTPUT:

The value of i after casting d (123.456) to int is: 123

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?**

int num = 10;

int \*ptr = &num;

float \*fPtr = (float \*)ptr;

Is this casting safe?

No, this casting is generally not safe

Directly casting a pointer from int\* to float\* is unsafe due to potential differences in size, representation, and alignment between int and float types. It can lead to undefined behavior, which might cause your program to crash, produce incorrect results, or behave unpredictably. Using language features like unions or copying values properly are safer alternatives for achieving similar functionality without risking pointer-related issues

.

· **Type Size Difference**:

· **Representation Difference**:

·**Alignment and Access Issues**:

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.**

int num = 10;

float fval = 3.14;

int \*intPtr = &num;

float \*floatPtr = &fval;

**Can you safely cast intPtr to floatPtr?**

No, it is generally not safe to cast intPtr to floatPtr

Here are the reasons why:

1. Type Size and Representation:
2. Alignment Requirements:
3. Pointer Dereferencing:
4. **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;

// Function to calculate area with int arguments

int Area(int width, int height) {

return width \* height;

}

// Function to calculate area with double arguments

double Area(double width, double height) {

return width \* height;

}

int main() {

int intWidth = 3;

int intHeight = 2;

double doubleWidth = 3.5;

double doubleHeight = 2.5;

// Calculate area using int arguments

int intArea = Area(intWidth, intHeight);

cout << "Area with int arguments: " << intArea << endl;

// Calculate area using double arguments

double doubleArea = Area(doubleWidth, doubleHeight);

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

return 0;

}

OUTPUT:

Area with int arguments: 6

Area with double arguments: 8.75

**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 <iostream>

#include <cmath> // For rounding functions

using namespace std;

int main() {

double celsius;

cout << "Enter temperature in Celsius: ";

cin >> celsius;

double fahrenheit = static\_cast<double>(celsius) \* 9.0 / 5.0 + 32.0;

fahrenheit = round(fahrenheit \* 100.0) / 100.0;

cout << "Temperature in Fahrenheit: " << fahrenheit << endl;

return 0;

}

OUTPUT:

Enter temperature in Celsius: 98

Temperature in Fahrenheit: 208.4

**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.**

**Unsafe Pointer Arithmetic with Casting**

Unsafe pointer arithmetic involves casting pointers to different types and manipulating them without proper bounds checking. Here's an example:

#include <iostream>

using namespace std;

int main() {

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

int \*ptr = arr;

char \*charPtr = reinterpret\_cast<char\*>(ptr);

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

cout << "Element " << i << ": " << static\_cast<int>(\*charPtr) << endl;

charPtr++;

}

return 0;

}

OUTPUT:

Element 0: 1

Element 1: 0

Element 2: 0

Element 3: 0

Element 4: 2

**Explanation of Unsafe Example:**

* In this example, ptr is a pointer to an array of integers.
* charPtr is created by casting ptr to char\*, effectively treating the memory as an array of characters rather than integers.
* The loop iterates over the memory locations, interpreting them as characters (char), but printing them as integers.

**Safe Pointer Arithmetic**

Safe pointer arithmetic involves using pointers within their intended types and ensuring bounds checking. Here’s an example of safe pointer arithmetic:

#include <iostream>

using namespace std;

int main() {

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

int \*ptr = arr;

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

std::cout << ptr[i] << " ";

}

cout << endl;

return 0;

}

OUTPUT:

1 2 3 4 5

**Explanation of Safe Example:**

* ptr is a pointer to an array of integers (int\*).
* The loop iterates over the array using safe pointer arithmetic (ptr[i]), accessing each element of the array.

**Potential Consequences of Unsafe**

* **Undefined Behavior:** This code demonstrates undefined behavior because it accesses memory in a manner inconsistent with its original type (int accessed as char). This violates strict aliasing rules and can lead to unpredictable results.
* **Memory Corruption:** Incorrect casting and pointer arithmetic can corrupt memory if the program writes to memory locations not intended for the type it assumes.
* **Portability Issues:** Reliance on such unsafe practices can lead to code that behaves differently on different platforms or compilers.