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; };

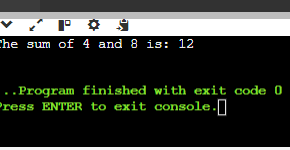
int result = sum(4,8);

cout << "The sum of 4 and 8 is: " << result << endl;

return 0;

}

Output:



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 a = 7;

auto square = [a]() { return a \* a; };

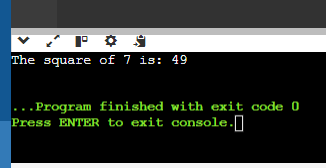
int result = square();

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

return 0;

}

Output:



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 text = "Dawn";

auto addPrefix = [&text]() -> std::string {

std::string prefix = "Souvik ";

text = prefix + text;

return text;

};

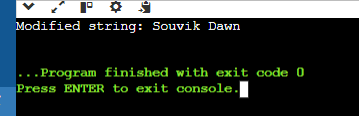
string result = addPrefix();

cout << "Modified string: " << result << endl;

return 0;

}

Output:



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 cond = true;

auto lambda = [num, cond]() mutable {

if (cond) {

num \*= 2;

} else {

num /= 2;

}

return num;

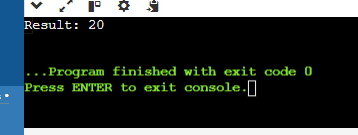
};

cout << "Result: " << lambda() << endl;

return 0;

}

Output:



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.

#include<iostream>

using namespace std;

void modifyConst(const int& val) {

int& nonConstVal = const\_cast<int&>(val);

nonConstVal = 10;

}

int main() {

int num = 5;

const int& constRef = num;

cout << "Before modification: " << constRef <<endl;

modifyConst(constRef);

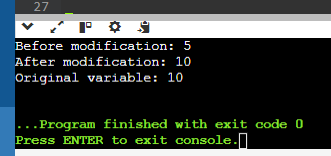
cout << "After modification: " << constRef <<endl;

cout << "Original variable: " << num << endl;

return 0;

}

Output:



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() {}

};

class Derived : public Base {

public:

void doSomething() { cout << "Derived class method called!" << endl; }

};

int main() {

Base\* basePtr = new Derived();

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

if (derivedPtr != nullptr) {

derivedPtr->doSomething();

} else {

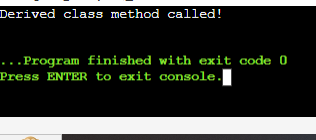
cout << "Cast failed! The object is not of type Derived." << endl;

}

return 0;

}

Output:



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>

using namespace std;

int main() {

int num = 42;

int\* numPtr = &num;

uintptr\_t intPtr = reinterpret\_cast<uintptr\_t>(numPtr);

cout << "Pointer as integer: " << intPtr << endl;

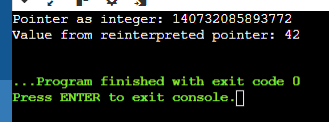
int\* newPtr = reinterpret\_cast<int\*>(intPtr);

cout << "Value from reinterpreted pointer: " << \*newPtr <<endl;

return 0;

}

Output:



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 show() {

cout << "Base class show function" << endl;

}

};

class Derived : public Base {

public:

void show() override {

cout << "Derived class show function" << endl;

}

};

int main() {

int i = 10;

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

cout << "Integer value: " << i << endl;

cout << "Double value: " << d << endl;

Derived derived;

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

basePtr->show();

Base base;

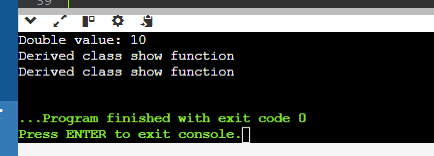
Derived\* derivedPtr = static\_cast<Derived\*>(basePtr);

derivedPtr->show();

return 0;

}

Output:



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;

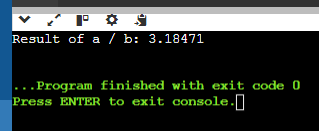
float result = a / b;

cout << "Result of a / b: " << result << endl;

return 0;

}

Output:



Explanation:

1. a is of type int and b is of type float.
2. Before performing the division operation, the compiler implicitly casts a from int to float.
3. The division is then performed between two float values , resulting in a float.
4. The result of the division is of type float, which is then assigned to the float variable result.

In this code the int variable a is implicitly cast to a float to match the type of b before performing the division, resulting in a float value for the result.

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;

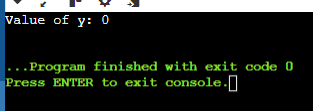
y = static\_cast<char>(x);

cout << "Value of y: " << static\_cast<int>(y) << endl;

return 0;

}

Output:



Data Loss – Yes we lost the original value of 256.The reason for this data loss is that the char type has a limited range of values (-128 to 127).when we cast the int value 256 to a char the value is fit within the char range for this reason it resulting in a value of 0.

How to avoid this – You need to store the value 256 in a char variable then consider using an unsigned char which has a range of 0 to 255.

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);

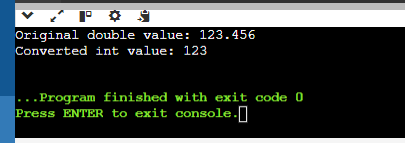
cout << "Original double value: " << d << endl;

cout << "Converted int value: " << i << endl;

return 0;

}

Output:



Behavior When Converting from a Larger Range to a Smaller One - When you convert a value from a larger range (like double) to a smaller range (like int), several things can happen like Explicit casting allows you to convert a value from one type to another, with full control over the conversion process. When converting from a double to an int, the fractional part is truncated, resulting in a loss of precision. This behavior is useful when you need only the integer part of a floating-point number, but it's essential to be aware of potential precision loss and overflow issues when working with large values.

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 << "Address of num: " << &num << endl;

cout << "Address stored in ptr: " << ptr << endl;

cout << "Address stored in fPtr: " << fPtr << endl;

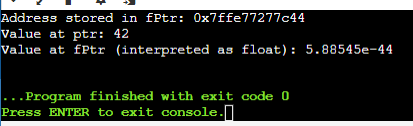
cout << "Value at ptr: " << \*ptr << endl;

cout << "Value at fPtr (interpreted as float): " << \*fPtr << endl;

return 0;

}

Output:



Is the casting safe or not – No the casting is not safe because

1. int and float have different internal representations. An int is typically stored as a binary integer, whereas a float is stored in IEEE 754 format, which includes a sign bit, exponent bits, and mantissa bits. Interpreting the bit pattern of an int as a float can lead to unexpected and meaningless values.
2. Different types may have different alignment requirements. Casting between incompatible pointer types can violate these requirements, potentially causing hardware traps or performance penalties on some architectures.

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 = 45;

float fval = 2.34;

int\* intPtr = &num;

float\* floatPtr = &fval;

float\* castedPtr = reinterpret\_cast<float\*>(intPtr);

cout << "Address of num: " << &num << endl;

cout << "Address stored in intPtr: " << intPtr << endl;

cout << "Address stored in castedPtr: " << castedPtr << endl;

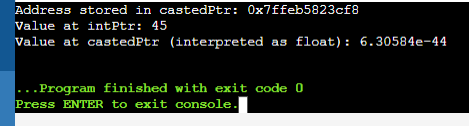
cout << "Value at intPtr: " << \*intPtr << endl;

cout << "Value at castedPtr (interpreted as float): " << \*castedPtr <<endl;

return 0;

}

Output:



Can you safely cast intPtr to floatPt - Casting an int\* to a float\* is not safe because of the different internal representations and potential for undefined behavior. The correct approach is to use pointers of the appropriate type for the data they point to, ensuring type safety and avoiding unpredictable behavior. If you need to store both integers and floats, consider using a union or a structure with explicit fields for each type. This ensures that you can safely access and manipulate the data without resorting to unsafe pointer casts.

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;

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

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

return 0;

}

In this case – the program will not compile.The reason is that we are trying to cast a reference to an int to a reference to a float which is a different type.For this reason the compiler will prvent this cast because it is a not valid convrsion.

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 x = 10;

float f = 3.14;

int& refX = x;

No you can not cast refx to f.In this above code refx is a reference to an int and it is bound to the memory location of x. You can not cast refx to refer to f because f is a float which is a different type from int.

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>

int calculateArea(int width, int height) {

return width \* height; }

double calculateArea(double width, double height) {

return width \* height;

}

int main() {

int width\_int = 5;

int height\_int = 3;

double width\_double = 5.5;

double height\_double = 3.5;

int area\_int = calculateArea(width\_int, height\_int);

std::cout << "Area with int arguments: " << area\_int << std::endl;

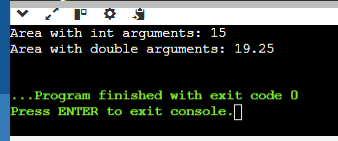
double area\_double = calculateArea(width\_double, height\_double);

std::cout << "Area with double arguments: " << area\_double << std::endl;

return 0;

}

Output:



Explanation:

Implicit Casting (int arguments):

Function: int calculateArea(int width, int height)

Implication: The multiplication width \* height is performed using int values, and the result is automatically converted to an int before returning. This can truncate decimal places if present in calculations.

Explicit Casting (double arguments):

Function: double calculateArea(double width, double height)

Implication: No casting is needed for the return type (double) because the multiplication width \* height is performed using double values. This ensures that the result maintains precision, crucial for calculations involving decimal values.

Usage:

Main Function: Demonstrates how to call each function with appropriate arguments (int and double) and prints the calculated area.

Choosing the Right Function:

Use the calculateArea(int, int) function when dealing exclusively with integer dimensions to maintain simplicity and avoid unnecessary precision.

Use the calculateArea(double, double) function when dealing with dimensions that may have decimal values to ensure accuracy in calculations.

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>

int main() {

double celsius;

std::cout << "Enter the temperature in Celsius: ";

std::cin >> celsius;

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

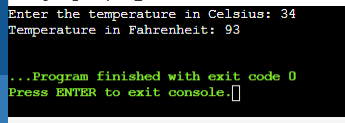
int rounded\_fahrenheit = static\_cast<int>(std::round(fahrenheit));

std::cout << "Temperature in Fahrenheit: " << rounded\_fahrenheit << std::endl;

return 0;

}

Output:



Explicit Casting: Used to ensure correct arithmetic operations by converting between int and double types as needed.

Rounding: Ensures that the converted temperature is displayed in a user-friendly format (whole number Fahrenheit).

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

#include <iostream>

using namespace std;

int main() {

int arr[5] = {10, 20, 30, 40, 50};

int\* ptr = arr;

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

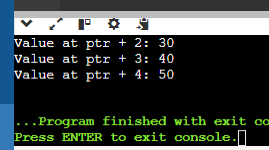
cout << "Value at ptr + " << i << ": " << \*(ptr + i) << endl;

}

return 0;

}

Output:



Memory Corruption: Accessing memory beyond allocated boundaries can corrupt data or affect program stability.

Undefined Behavior: Violating type rules or accessing uninitialized or freed memory can lead to unpredictable program behavior.

Security Vulnerabilities: Improper pointer manipulation can be exploited by attackers to execute arbitrary code or gain unauthorized access.

Debugging Challenges: Errors caused by pointer mismanagement can be difficult to debug and fix, especially in large codebases.

Safe Pointer Arithmetic: Ensures operations are within the defined boundaries of allocated memory, minimizing risks.

Unsafe Pointer Arithmetic with Casting: Can lead to severe consequences such as memory corruption, undefined behavior, and security vulnerabilities, due to incorrect memory access.