1.) What is type casting in C++ and what are the two main types?

Type casting refers to converting a variable from one data type to another. It can be done in two main ways:

1. **Implicit Casting (Automatic Conversion)**:

* This occurs automatically when the compiler changes a variable from one type to another. For example, assigning an int value to a float variable.

**2. Explicit Casting (Manual Conversion)**:

* This requires the programmer to specify the type conversion explicitly.
* There are several ways to perform explicit casting in C++:

2.) Explain the difference between implicit and explicit type casting.

### Key Differences

1. **Control**:
   * Implicit: Performed automatically by the compiler.
   * Explicit: Requires programmer intervention to specify the cast.
2. **Safety**:
   * Implicit: Generally safer, but can still lead to precision loss.
   * Explicit: Greater risk of data loss or runtime errors if not used carefully.
3. **Syntax**:
   * Implicit: No special syntax required.
   * Explicit: Requires cast operators or functions.
4. **Use Cases**:
   * Implicit: Common in type promotions and simple assignments.
   * Explicit: Used for more complex or unsafe conversions, such as converting pointers or handling specific type requirements.

### 3.) When would you use implicit type casting in C++?

### Implicit type casting in C++ (also known as implicit type conversion) occurs when the compiler automatically converts one type of data into another type without any explicit instructions from the programmer. This is done to make operations compatible or to ensure data is properly handled in expressions.

### Here are some scenarios where implicit type casting is used in C++:

### 1. \*Assignment Compatibility:\*

### - When assigning a value of one type to a variable of another type, if the destination type can represent all values of the source type, C++ allows implicit conversion.

### Example :-

### int a = 10;

### double b = a; // int to double (implicit conversion)

### 2. \*Arithmetic Operations:\*

### - During arithmetic operations involving operands of different types, C++ will perform implicit conversions to ensure compatibility.

### [9:47 am, 9/7/2024] Ameesha Wipro: Example :-

### int x = 5;

### double y = 2.5;

### double z = x + y; // int to double (implicit conversion)

### 3. \*Function Call Arguments:\*

### - When passing arguments to functions, if the function parameter type is different from the argument type, implicit conversion may occur if it's safe and makes sense.

### Example :-

### void printDouble(double d) {

### cout << d << endl;

### }

### int num = 100;

### printDouble(num); // int to double (implicit conversion)

### [9:48 am, 9/7/2024] Ameesha Wipro: 4. \*Return Statements:\*

### - When a function returns a value, if the return type is different from the actual returned expression's type, implicit conversion may occur.

### Example :-

### int getInteger() {

### return 3.14; // double to int (implicit conversion)

### }

### 5. \*Mixing Enumerations and Integers:\*

### - Enumerated types can be implicitly converted to integers and vice versa, depending on the context.

### Example :-

### enum Color { Red, Green, Blue };

### int num = Blue; // enum to int (implicit conversion)

### 6. \*Initialization of Objects:\*

### - When initializing objects with different types, implicit conversion might happen if the constructor supports it.

### Example :-

### double d = 3; // int to double (implicit conversion)

### 4.) How can you explicitly cast an integer to a float in C++?

In C++, explicitly casting an integer to a float can be done using three primary methods:

**1. C-Style Cast**

* Uses the C-style casting syntax, which places the target type in parentheses before the variable.
* Example: (float)integerVariable.

**2. Function-Style Cast**

* Uses the constructor-like syntax, where the target type is treated as a function.
* Example: float(integerVariable).

**3. C++ static\_cast Operator**

* Uses the static\_cast operator, which is specific to C++ and provides better type safety and readability.
* Example: static\_cast<float>(integerVariable).

Each of these methods will convert an integer to a float, but static\_cast is generally preferred in modern C++ for its clarity and type safety.

5.) What are the potential risks associated with explicit type casting?

Explicit type casting can introduce several potential risks, including:

1. **Data Loss**:
   * When casting from a larger data type to a smaller one, information can be lost. For example, casting a float or double to an int truncates the decimal part.
   * Example: int i = (int)3.14; // i becomes 3
2. **Precision Loss**:
   * When converting between floating-point types and integers, or even between different floating-point types, precision can be lost.
   * Example: float f = 123456789.123456789; double d = (double)f; // d may not hold the exact value of f
3. **Undefined Behavior**:
   * Casting pointers, especially using reinterpret\_cast, can lead to undefined behavior if the types are not compatible or if the cast violates type aliasing rules.
   * Example: int\* p = reinterpret\_cast<int\*>(0x1234); // arbitrary memory address
4. **Invalid Downcasting**:
   * In polymorphic inheritance, using static\_cast to downcast a base class pointer/reference to a derived class without proper type checking can lead to undefined behavior.
   * Example: Base\* b = new Derived(); Derived\* d = static\_cast<Derived\*>(b); // valid only if b is actually pointing to a Derived object
5. **Aliasing Issues**:
   * Using const\_cast to remove const qualifiers and then modifying the object can lead to undefined behavior if the original object was declared const.
   * Example: const int i = 42; int\* p = const\_cast<int\*>(&i); \*p = 43; // modifying a const object
6. **Memory Corruption**:
   * Incorrect casting of pointers, especially in low-level programming, can lead to memory corruption and hard-to-debug errors.
   * Example: int\* p = new int(10); char\* c = reinterpret\_cast<char\*>(p); \*c = 'a'; // potentially corrupts the memory
7. **Logical Errors**:
   * Misinterpreting the type you are casting to or from can lead to logical errors in the program, producing incorrect results.
   * Example: int i = 65; char c = static\_cast<char>(i); // c becomes 'A', but misinterpretation can occur

6.) Describe the four different types of explicit casting operators in C++.

There are four types of explicit casting operators, each with specific use cases and characteristics:

**1. static\_cast**

* **Purpose**: Used for standard type conversions, such as converting between numeric types (e.g., int to float), converting pointers to related types within an inheritance hierarchy (upcasting and downcasting), and for user-defined conversions.
* **Safety**: Provides compile-time type checking, ensuring that the conversion is safe and meaningful.
* **Usage Example**: float f = static\_cast<float>(3);

**2. dynamic\_cast**

* **Purpose**: Primarily used for safe downcasting in an inheritance hierarchy. It checks at runtime whether the cast is valid, and if not, it returns nullptr for pointers or throws a std::bad\_cast exception for references.
* **Safety**: Ensures that the cast is valid at runtime, providing a way to safely navigate polymorphic class hierarchies.
* **Usage Example**: Derived\* d = dynamic\_cast<Derived\*>(basePointer);

**3. const\_cast**

* **Purpose**: Used to add or remove const or volatile qualifiers from a variable. It is the only cast that can be used to change the constness of a variable.
* **Safety**: Generally safe if used correctly, but removing const can lead to undefined behavior if the underlying object was originally defined as const.
* **Usage Example**: const int\* p = &i; int\* q = const\_cast<int\*>(p);

**4. reinterpret\_cast**

* **Purpose**: Used for low-level reinterpreting of the bit pattern of an object. It allows casting between any pointer types, even unrelated ones, and between pointer and integer types.
* **Safety**: The most dangerous cast as it can easily lead to undefined behavior if not used carefully. It should be used sparingly and with a deep understanding of what is being done.
* **Usage Example**: int\* p = reinterpret\_cast<int\*>(0x1234);

8.) When should you use static\_cast for type casting?

static\_cast should be used for type casting in C++ when you need a safe and explicit type conversion that is checked at compile-time. Here are specific scenarios where static\_cast is appropriate:

1. **Basic Data Type Conversions**:
   * Converting between fundamental data types like int, float, double, etc.
   * Example: float f = static\_cast<float>(3);
2. **Pointer Conversions in Inheritance Hierarchies**:
   * Upcasting: Converting a derived class pointer to a base class pointer.
   * Downcasting: Converting a base class pointer to a derived class pointer, but only when you are certain of the type.
   * Example (upcasting): Base\* b = static\_cast<Base\*>(d); where d is a Derived\*.
   * Example (downcasting): Derived\* d = static\_cast<Derived\*>(b); where b is actually pointing to a Derived object.
3. **Removing const or volatile Qualifiers**:
   * While const\_cast is specifically for changing const or volatile qualifiers, static\_cast can be used in situations where removing such qualifiers is part of a larger cast.
   * Example: const int\* ci = &i; int\* i = static\_cast<int\*>(ci); (although const\_cast is more appropriate here).
4. **Converting Between Related Classes**:
   * When you have classes related by inheritance and need to convert pointers or references between them.
   * Example: If class B inherits from class A, you can use static\_cast to convert A\* to B\* when you are sure the object is of type B.
5. **User-Defined Conversions**:
   * For invoking user-defined conversion operators.
   * Example: Converting an object of a class to another type using a conversion operator defined in the class.
6. **Avoiding Implicit Conversions**:
   * When you want to make an implicit conversion explicit for clarity and maintainability.
   * Example: int i = 42; double d = static\_cast<double>(i); instead of double d = i;.

9.) In what scenario would you use dynamic\_cast for type casting?

In C++, dynamic\_cast is used primarily in scenarios involving polymorphic types, specifically when working with inheritance hierarchies where polymorphism is employed through virtual functions. Here are the key scenarios where dynamic\_cast is appropriate:

1. **Downcasting in Polymorphic Hierarchies**:
   * When you have a base class pointer or reference and you need to safely cast it to a derived class pointer or reference.
2. **Runtime Type Identification (RTTI)**:

* When you need to check the type of an object at runtime within an inheritance hierarchy.

1. **Handling Polymorphic Base Classes**:

* When working with interfaces or polymorphic base classes, dynamic\_cast allows you to check and safely cast to derived types.

4 **Avoiding Undefined Behavior**:

* dynamic\_cast performs runtime checks to ensure that the cast is valid. If the cast is not valid (e.g., trying to cast to a type that is not related in the inheritance hierarchy), dynamic\_cast returns nullptr for pointers or throws a std::bad\_cast exception for references.
* This helps avoid undefined behavior that might occur with static\_cast when downcasting to an incorrect type.

5. **Safe Handling of Pointers and References**:

* dynamic\_cast is particularly useful when dealing with pointers or references to polymorphic types where you need to ensure type safety during runtime.

10.) Explain the purpose of const\_cast and when it might be necessary.

const\_cast in C++ is used to add or remove const or volatile qualifiers from a variable. Its primary purpose is to modify the constness or volatility of an object, allowing otherwise restricted operations. Here's when const\_cast might be necessary:

1. **Modifying const Variables**:
   * When you have a pointer or reference to a const-qualified object and you need to modify the underlying object.

**Interfacing with Legacy Code**:

* When dealing with legacy code or third-party libraries that use const\_cast to work around constness restrictions.

**Implementing Non-const Interface Methods**:

* When implementing interface methods that are declared as non-const, but you need to modify members that are logically const within the implementation.

**Working with Mutable Data Members**:

* When working with mutable data members within const member functions to update internal state without violating const-correctness.

11.) What are the dangers of using reinterpret\_cast and why should it be used with caution?

reinterpret\_cast in C++ is a powerful casting operator that allows for low-level reinterpretation of the bit pattern of an object. However, it comes with significant risks and should be used with extreme caution due to the following dangers:

1. **Undefined Behavior**:
   * reinterpret\_cast can lead to undefined behavior if used incorrectly, especially when casting between unrelated types or when the cast violates type aliasing rules.

**Type Safety Violations**:

1. Unlike other casts (static\_cast, dynamic\_cast, etc.), reinterpret\_cast performs no type checking during compilation. It assumes the programmer knows exactly what they are doing, which can lead to inadvertent errors.

**Platform Dependency**:

1. The behavior of reinterpret\_cast can vary across different platforms and compilers, making code less portable and harder to maintain.

**Debugging Complexity**:

1. Code involving reinterpret\_cast can be difficult to debug because it operates at a low level of abstraction, making it challenging to trace errors and understand the program flow.

**Security Risks**:

1. Misuse of reinterpret\_cast can lead to security vulnerabilities, such as buffer overflows or memory corruption, especially when dealing with pointers and memory management.

12.) Can you cast a pointer to a different data type using explicit casting?

Yes, you can cast a pointer to a different data type using explicit casting in C++. This is typically done using the reinterpret\_cast operator when you need to reinterpret the bit pattern of the pointer as if it were pointing to a different type of object.

13.) What happens when casting a larger data type to a smaller one? How can data loss occur?

When casting a larger data type to a smaller one in C++, data loss can occur due to truncation or loss of precision. Here's an explanation of what happens and how data loss can occur:

**1. Truncation of Data:**

* **Example**: Casting from a double to an int.
* **Behavior**: The value is truncated to fit within the range of the smaller data type.
* **Data Loss**: If the original value is outside the range representable by the smaller type, truncation occurs, and the excess bits are discarded.

**Loss of Precision:**

* **Example**: Casting from a float to an int.
* **Behavior**: The floating-point value is converted to an integer by truncating the fractional part.
* **Data Loss**: Floating-point numbers have a greater range and precision than integers. When casting to an integer, the fractional part is discarded, leading to potential loss of precision.

**Range Limitations:**

* **Example**: Casting from a larger integer type (long long) to a smaller one (int).
* **Behavior**: The value is reduced to fit within the range of the smaller type.
* **Data Loss**: If the original value exceeds the maximum representable value of the smaller type, it wraps around or is truncated, leading to incorrect results.

13.)How can you check if a type casting operation is successful with dynamic\_cast?

In C++, dynamic\_cast is used primarily for type-safe downcasting within inheritance hierarchies that involve polymorphism. Here's how you can check if a type casting operation using dynamic\_cast is successful:

1. **Returns Null Pointer (Pointers)**:
   * When casting pointers, if the target type is not a valid pointer of the derived type, dynamic\_cast returns a null pointer (nullptr).
   * This indicates that the cast was unsuccessful because the object being pointed to is not of the derived type.
2. **Throws std::bad\_cast (References)**:
   * When casting references, if the target type is not a valid reference to the derived type, dynamic\_cast throws a std::bad\_cast exception.
   * This exception indicates that the cast was unsuccessful because the referenced object is not of the derived type.
3. **Runtime Type Identification (RTTI)**:
   * dynamic\_cast performs a runtime check to verify the validity of the cast.
   * It uses the runtime type information (RTTI) to determine if the cast can be safely performed.
   * If the cast is not valid, it results in a null pointer (for pointers) or throws an exception (for references).
4. **Safety Mechanism**:
   * The primary purpose of dynamic\_cast is to provide a safe mechanism for downcasting in polymorphic hierarchies.
   * It ensures that the cast is performed only if the object is actually of the derived type, preventing undefined behavior that could occur with static\_cast.

14.) Is there a way to perform type casting without using any casting operators?

In C++, type casting without using any casting operators (such as static\_cast, dynamic\_cast, reinterpret\_cast, const\_cast) directly is not possible. The casting operators are intrinsic to the language's syntax and semantics for converting one type to another. They provide explicit mechanisms for the programmer to indicate the desired type conversion and allow the compiler to perform necessary checks and adjustments.

However, in some cases, type conversion can occur implicitly or through other mechanisms that are not explicitly called "casting operators," but they still involve type conversion. These include:

1. **Implicit Conversions**:
   * These are conversions that happen automatically by the compiler when compatible types are used in expressions, assignments, and function calls.
   * Example: int to float, float to double, etc.
2. **Constructor-Based Conversions**:
   * In the context of classes, constructor functions can serve as conversion operators when used to construct objects of one class type from another.

**User-Defined Conversion Operators**:

* Classes can define conversion operators (operator type()) to specify how objects of one class type can be converted to another type.

15.) What are some best practices for using type casting effectively in C++ code?

Using type casting effectively in C++ is crucial for maintaining code clarity, correctness, and safety. Here are some best practices to follow:

1. **Prefer Static Casting (static\_cast)**:
   * Use static\_cast for most type conversions where the conversion is well-defined and known at compile-time.
   * It provides compile-time type checking and is explicit about the intent of the conversion.
2. **Use const\_cast Sparingly**:
   * Avoid const\_cast whenever possible and only use it when you need to remove const or volatile qualifiers for legitimate reasons, such as interfacing with legacy APIs.
3. **Limit reinterpret\_cast Usage**:
   * reinterpret\_cast should be used with extreme caution due to its potential for causing undefined behavior.
   * Use it only when absolutely necessary for low-level memory manipulations or interfacing with hardware.
4. **Avoid C-Style Casts**:
   * Avoid using C-style casts ((type)value) in favor of C++ casting operators (static\_cast, dynamic\_cast, etc.).
   * C++ casting operators provide better readability and type safety.
5. **Use dynamic\_cast for Polymorphic Types**:
   * Use dynamic\_cast when dealing with polymorphic types and downcasting within inheritance hierarchies.
   * It provides runtime type checking and helps avoid undefined behavior associated with static\_cast for downcasting.
6. **Understand Type Conversion Costs**:
   * Be aware of the potential performance costs associated with type conversions, especially between numeric types or when using casting in tight loops.
   * Consider whether the conversion is necessary and whether it can be optimized or avoided.
7. **Document Non-Obvious Conversions**:
   * Document explicit type casts, especially when the conversion may not be immediately obvious from the code.
   * Provide comments explaining why the cast is necessary and what guarantees or assumptions are made.
8. **Validate Pointers after Casting**:
   * After casting pointers using reinterpret\_cast or static\_cast, validate the resulting pointer to ensure it points to a valid object or memory location.
   * Avoid accessing memory through invalid pointers resulting from incorrect casting.
9. **Consider Alternative Design Patterns**:
   * Sometimes, using polymorphism, virtual functions, or templates may offer more type-safe and maintainable solutions than explicit type casting.
   * Evaluate whether a design change could eliminate the need for explicit type casting altogether.
10. **Code Reviews and Testing**:
    * Include type casting operations in code reviews to ensure correctness and adherence to best practices.
    * Test edge cases and scenarios where type casting is used to verify behavior and correctness under different conditions.

16.) Create a code example that demonstrates the use of static\_cast for performing a calculation.

#include <iostream>

int main() {

// Calculate the area of a circle using static\_cast for type conversion

double radius = 5.0;

double area = static\_cast<double>(radius \* radius \* 3.14159);

std::cout << "Radius: " << radius << std::endl;

std::cout << "Area of the circle: " << area << std::endl;

return 0;

}

17.) Write a program that showcases the difference between implicit and explicit casting of integers to floats.

#include <iostream>

int main() {

int intVal1 = 5;

int intVal2 = 2;

// Implicit casting

float implicitResult = intVal1 / intVal2; // Division of two integers

// Explicit casting

float explicitResult = static\_cast<float>(intVal1) / static\_cast<float>(intVal2); // Division of two floats

std::cout << "Implicit casting result (int / int): " << implicitResult << std::endl;

std::cout << "Explicit casting result (float / float): " << explicitResult << std::endl;

return 0;

}

**Explanation:**

* **Implicit Casting**:
  + implicitResult is assigned the result of intVal1 / intVal2, which is integer division.
  + Since both intVal1 and intVal2 are integers, the division operation results in an integer. In this case, 5 / 2 results in 2.
  + The result is then implicitly cast to float, so implicitResult becomes 2.0.
* **Explicit Casting**:
  + explicitResult is assigned the result of static\_cast<float>(intVal1) / static\_cast<float>(intVal2), which converts both integers to floats before performing the division.
  + Since intVal1 and intVal2 are explicitly cast to float, the division operation results in floating-point division. In this case, 5.0 / 2.0 results in 2.5.
  + explicitResult correctly captures the floating-point result 2.5.

18.) Simulate a scenario where dynamic\_cast is used for checking inheritance relationships between classes.

#include <iostream>

#include <vector>

#include <typeinfo>

// Base class

class Animal {

public:

virtual ~Animal() {}

virtual void makeSound() const {

std::cout << "Some generic animal sound" << std::endl;

}

};

// Derived class

class Dog : public Animal {

public:

void makeSound() const override {

std::cout << "Bark" << std::endl;

}

};

// Another derived class

class Cat : public Animal {

public:

void makeSound() const override {

std::cout << "Meow" << std::endl;

}

};

// Function to identify the type of Animal using dynamic\_cast

void identifyAnimal(Animal\* animal) {

if (Dog\* dog = dynamic\_cast<Dog\*>(animal)) {

std::cout << "This is a dog." << std::endl;

dog->makeSound();

} else if (Cat\* cat = dynamic\_cast<Cat\*>(animal)) {

std::cout << "This is a cat." << std::endl;

cat->makeSound();

} else {

std::cout << "This is some other type of animal." << std::endl;

animal->makeSound();

}

}

int main() {

// Create a vector of Animal pointers

std::vector<Animal\*> animals;

// Add different types of animals to the vector

animals.push\_back(new Animal());

animals.push\_back(new Dog());

animals.push\_back(new Cat());

// Identify each animal in the vector

for (Animal\* animal : animals) {

identifyAnimal(animal);

std::cout << "-------------------" << std::endl;

}

// Clean up

for (Animal\* animal : animals) {

delete animal;

}

return 0;

}

Using reinterpret\_cast in C++ is generally discouraged due to the risks it entails, such as breaking type safety and causing undefined behavior. However, there are specific situations where reinterpret\_cast might be justified. Here are some scenarios, along with a discussion of potential risks:

### 1. Interfacing with Hardware or Low-Level APIs

#### Scenario:

* **Justification**: When dealing with hardware registers, memory-mapped I/O, or certain low-level APIs, you might need to interpret raw memory addresses as specific types.
* **Example**: Accessing a hardware register via its memory address.

#### Risks:

* **Undefined Behavior**: If the casted pointer is dereferenced and the memory doesn't actually represent the intended type, this can cause undefined behavior.
* **Portability Issues**: Code relying on specific memory addresses and casting might not be portable across different hardware or compilers.

### 2. Type-Punning

#### Scenario:

* **Justification**: Type-punning is sometimes used in low-level code to reinterpret the bit pattern of a value as another type.
* **Example**: Interpreting a float's bits as an int for bit manipulation.

#### Risks:

* **Strict Aliasing Rule**: Violating the strict aliasing rule can lead to undefined behavior, as the compiler may make assumptions about types that don't alias each other.
* **Portability Issues**: Different platforms might represent floating-point numbers differently, leading to inconsistent results.

### 3. Serialization/Deserialization

#### Scenario:

* **Justification**: When serializing data to a binary format or deserializing it back, you might need to reinterpret raw data bytes as specific types.
* **Example**: Reading raw bytes from a file and interpreting them as a custom structure.

#### Risks:

* **Alignment Issues**: The casted pointer might not be correctly aligned for the target type, causing undefined behavior when accessed.
* **Endianness Issues**: The byte order might differ between systems, leading to incorrect interpretation of data.

### 4. Implementing Polymorphic Behavior

#### Scenario:

* **Justification**: In some cases, when implementing polymorphism manually or interfacing with C libraries, you might need to cast between different pointer types.
* **Example**: Casting a base class pointer to a derived class pointer when you know the actual object type.

#### Risks:

* **Type Safety**: If the cast is incorrect and the object is not of the intended type, this can lead to accessing invalid memory and undefined behavior.
* **Maintainability**: Using reinterpret\_cast for polymorphism can make the code harder to understand and maintain, as it bypasses the type system's safety checks.

20.) Compare and contrast type casting with type conversion in

Type casting and type conversion are both mechanisms in programming languages like C++ to convert a value from one type to another. While they are related concepts, they have distinct differences in their usage, implications, and underlying mechanisms.

**Type Casting**

**Definition**: Type casting is a way of explicitly converting a value from one type to another. It can be done using casting operators in C++.

**Types of Type Casting**:

1. **Implicit Casting**: Also known as automatic or coercion, it is performed by the compiler without any explicit instruction from the programmer.
   * Example: Converting an int to a float when assigning to a float variable.

**Explicit Casting**: Done manually by the programmer using casting operators.

* **C-style cast**: (type)value
* **C++ casting operators**: static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast

**Type Conversion**

**Definition**: Type conversion is a process by which a value of one type is transformed into a value of another type. This can be done automatically by the compiler (implicit conversion) or manually by the programmer (explicit conversion).

**Types of Type Conversion**:

1. **Implicit Conversion**: Similar to implicit casting, where the compiler automatically converts types when required.
   * Example: Converting a smaller integer type to a larger integer type.

**Explicit Conversion**: Conversion that requires explicit instructions by the programmer.

* This can involve constructor calls or assignment operators.

**Comparison**

1. **Control**:
   * **Type Casting**: Offers more control and precision. The programmer explicitly specifies the desired type.
   * **Type Conversion**: Can be implicit or explicit. Implicit conversion offers less control, while explicit conversion offers control through functions or constructors.
2. **Usage Context**:
   * **Type Casting**: Commonly used in polymorphism, pointer conversions, and when working with low-level data.
   * **Type Conversion**: Often used in converting between numeric types, strings, and custom types through constructors or conversion functions.
3. **Syntax**:
   * **Type Casting**: Uses casting operators like static\_cast, dynamic\_cast, etc.
   * **Type Conversion**: Can be implicit in assignments or done through conversion functions (e.g., std::stoi).
4. **Error Handling**:
   * **Type Casting**: Explicit casts can lead to runtime errors if not used correctly, especially with reinterpret\_cast or dynamic\_cast.
   * **Type Conversion**: Conversion functions can throw exceptions (e.g., std::stoi throws std::invalid\_argument).
5. **Performance**:
   * **Type Casting**: Can be more efficient in terms of execution since it’s more direct.
   * **Type Conversion**: May involve more overhead, especially if complex logic is involved in the conversion function.