TASK – 1

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

A: Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion. They are two main types:

**Implicit Type Casting or Implicit Type Conversion:** It is known as the automatic type casting. It automatically converted from one data type to another without any external intervention such as programmer or user. It means the compiler automatically converts one data type to another. All data type is automatically upgraded to the largest type without losing any information. It can only apply in a program if both variables are compatible with each other.

**Explicit Type Casting or Explicit Type Conversion**: It is also known as the manual type casting in a program. It is manually cast by the programmer or user to change from one data type to another type in a program. It means a user can easily cast one data to another according to the requirement in a program. It does not require checking the compatibility of the variables. In this casting, we can upgrade or downgrade the data type of one variable to another in a program. It uses the cast () operator to change the type of a variable.

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

|  |  |
| --- | --- |
| Implicit Type Conversion | Explicit Type Conversion |
| An implicit type conversion is automatically performed by the compiler when differing data types are intermixed in an expression. | An explicit type conversion is user-defined conversion that forces an expression to be of specific type. |
| An implicit type conversion is performed without programmer's intervention. | An Explicit type conversion is performed with programmers intervention |
| Example: a, b = 5, 25.5 c = a + b | Example: a, b = 5, 25.5 c = int(a + b) |

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

A: Implicit type casting, also known as automatic type conversion or coercion, in C++ occurs when the compiler automatically converts one data type to another without explicit instructions from the programmer. Here are some scenarios where implicit type casting is commonly used: **Arithmetic Operations, Function Calls**, Assignment Operations, Mixed Expressions and Return Values.

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

A: In C++, you can explicitly cast an integer to a float using several methods. Here are the common ways to perform explicit type casting:

1. **C-Style Cast**:

int i = 42;

float f = (float)i;

1. **C++ static\_cast**:

int i = 42;

float f = static\_cast<float>(i);

1. **Functional Cast**:

int i = 42; float f = float(i);

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

A: Potential for Overflow, Underflow, and Data Loss: Be mindful of potential risks such as overflow, underflow, and data loss when performing explicit conversions, especially with numeric data types.

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

A: Casting operators are used for type casting in C++. They are used to convert one data type to another. C++ supports four types of casts:

**static\_cast -** It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.

**dynamic\_cast -** The operator is mainly used to perform down casting (converting a pointer/reference of a base class to a derived class). It ensures type safety by performing a runtime check to verify the validity of the conversion.

**const\_cast -** The operator is used to modify the const or volatile qualifier of a variable. It allows programmers to temporarily remove the constancy of an object and make modifications.

**reinterpret\_cast – It** is used to convert the pointer to any other type of pointer. It does not perform any check whether the pointer converted is of the same type or not

**7. When should you use static\_cast for type casting?**

A: In general you use **static\_cast** when you want to convert numeric data types such as enums to ints or ints to floats, and you are certain of the data types involved in the conversion. **static\_cast** conversions are not as safe as **dynamic\_cast** conversions, because **static\_cast** does no run-time type check, while **dynamic\_cast** does.

**8. In what scenario would you use dynamic\_cast for type casting?**

A: Dynamic\_cast is used for type casting in scenarios where runtime type safety is essential, specifically within a polymorphic class hierarchy. The main scenarios include:

1. **Safe Down casting**: When you need to convert a base class pointer/reference to a derived class pointer/reference and want to ensure the conversion is valid at runtime. This prevents unsafe type conversions that could lead to undefined behavior.
2. **Type Identification**: When you need to determine the actual derived type of an object at runtime, enabling you to perform operations specific to that derived type. This is useful in scenarios where you have a collection of base class pointers but need to handle derived class-specific logic.
3. **Handling Multiple Inheritances**: When working with complex inheritance hierarchies, especially those involving multiple inheritance, dynamic\_cast ensures that the conversion respects the actual runtime type of the object, accounting for the intricacies of multiple inheritance.

**9. Explain the purpose of const\_cast and when it might be necessary.**

A: const\_cast is used in C++ to add or remove const (or volatile) qualifiers from a variable. The primary purpose of const\_cast is to allow modifications to an object that was originally defined as const. It enables you to override the const correctness of the type system in specific situations where you know that the modification is safe or necessary. Purposes of const\_cast: Removing const for Legacy Code or Libraries, Adding const for API Compatibility, Calling Non-const Functions on Const Objects.

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

A: Reinterpret\_cast is a powerful and potentially dangerous casting operator in C++. It is used for low-level type conversions that involve reinterpreting the underlying bit pattern of an object. While it can be useful in specific scenarios, it should be used with extreme caution due to the following dangers:

**Dangers of reinterpret\_cast:**

1. **Undefined Behavior**:
   * reinterpret\_cast can easily lead to undefined behavior if the resulting casted type is not compatible with the original type. This can happen when the casted type does not correctly interpret the bit pattern of the original type.
2. **Type Safety Violation**:
   * It bypasses the type safety provided by the C++ type system. This can lead to difficult-to-diagnose bugs and crashes because the compiler cannot guarantee the validity of the cast.
3. **Alignment Issues**:
   * Some types have strict alignment requirements. Reinterpreting a pointer to a type with different alignment requirements can cause alignment faults, leading to crashes or incorrect behavior on some architectures.
4. **Memory Access Errors**:
   * Reinterpreting a pointer to an incompatible type and then dereferencing it can result in accessing invalid memory locations, which can corrupt data or crash the program.
5. **Loss of Portability**:
   * Code that relies on reinterpret\_cast is often not portable across different compilers or platforms because it depends on the specific memory layout and alignment of the types involved.
6. **Hard-to-Debug Issues**:
   * Bugs resulting from improper use of reinterpret\_cast can be very hard to trace and debug because they may not manifest immediately and can cause subtle corruption of data.

**11. Can you cast a pointer to a different data type using explicit casting?**

A: Yes, you can cast a pointer to a different data type using explicit casting in C++. This is done using the reinterpret\_cast operator. However, it's important to be aware that this can lead to undefined behavior if not done carefully.

The reinterpret\_cast operator essentially tells the compiler to treat the memory pointed to by the pointer as a different data type, without actually changing the underlying data.

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

A: When casting a larger data type to a smaller one, the extra bits of the larger type are truncated or discarded, potentially resulting in data loss.

**Data loss can occur in several ways:**

**1. Truncation:** Discarding higher-order bits or precision bits can result in a loss of significant digits or a reduction in the value's range.

**2. Overflow:** Casting a large value to a smaller type can cause overflow, resulting in an incorrect value (e.g., casting a large positive int to a smaller unsigned type like uint8\_t).

**3. Underflow:** Casting a small value to a smaller type can cause underflow, resulting in an incorrect value (e.g., casting a small negative int to an unsigned type like uint8\_t).

4. Rounding errors: Casting a floating-point value to a smaller type can result in rounding errors, leading to a loss of precision.

**To avoid data loss, it's essential to:**

1. Use appropriate casting operators (e.g., static\_cast or reinterpret\_cast).

2. Ensure the destination type can represent the original value's range and precision.

3. Use wider types when possible to avoid truncation and overflow.

4. Be aware of the potential risks and consequences.

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

A: With dynamic\_cast, you can check if a type casting operation is successful by checking if the result is a null pointer. Here's an example:

Base\* base = new Derived(); // assume Derived is a subclass of Base

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

if (derived != nullptr) {

// The cast was successful, and derived points to a valid Derived object

} else {

// The cast failed, and derived is a null pointer

}

In this example, dynamic\_cast attempts to cast the base pointer to a Derived\* pointer. If the cast is successful, derived will point to the Derived object. If the cast fails (e.g., if base is not a Derived object), derived will be a null pointer.

By checking if derived is not null, you can determine if the cast was successful. This is a common way to ensure that the cast was successful and to avoid null pointer dereferences.

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

A: Yes, there are a few ways to perform type casting without using casting operators in C++:

**1. Constructor casting:** Some classes have constructors that take an object of a different type and create a new object of their own type. For example, you can create a std::string object from a const char\* using the std::string constructor.

**2. Assignment:** You can assign a value of one type to a variable of another type, and the compiler will perform the necessary conversion. For example, you can assign an int value to a float variable.

**3. Function calls:** Some functions take arguments of a different type and perform the necessary conversion. For example, the std::sqrt function takes a float argument, but you can pass an int value to it and the compiler will perform the conversion.

**4. User-defined conversions:** You can define your own conversion functions or constructors in your classes to perform type casting without using casting operators.

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

A: **1. Avoid C-style casts:** Instead of using C-style casts (e.g., (Type)variable), use C++ casting operators (e.g., static\_cast<Type>(variable)).

**2.Use casting for conversion, not for forcing:** Casting should be used for legitimate conversions, not to force a type mismatch.

**3. Be aware of object slicing:** When casting a derived class to a base class, be aware of object slicing and potential loss of information.

**4.Use dynamic\_cast with caution:** Dynamic casting can be slow and may fail. Use it only when necessary and check for null pointers.

**5. Document casting:** Clearly document the reasoning behind casting, especially for complex or potentially risky casts.

**6. Avoid casting between unrelated types:** Casting between unrelated types can lead to undefined behavior. Avoid it whenever possible.

**7. Use const\_cast with care:** Const casting can be dangerous, as it bypasses const correctness. Use it only when necessary and with caution.

**8. Consider alternative designs:** Before casting, consider alternative designs that avoid the need for casting, such as using templates or polymorphism**.**

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

A: #include <iostream>

Using namespace std;

int main() {

// Define a double variable

double d = 3.7;

// Use static\_cast to convert double to int

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

// Perform a calculation

int result = i \* 2;

// Print the result

cout << "Result: " << result << endl;

return 0;

}

This code example demonstrates the following:

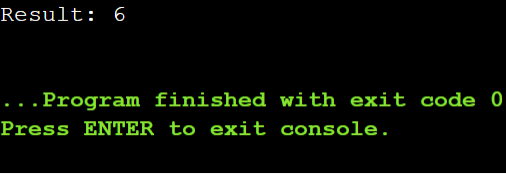
1. Define a double variable d with the value 3.7.

2. Use static\_cast to convert the double value to an int, which truncates the decimal part, resulting in i being assigned the value 3.

3. Perform a calculation by multiplying i by 2, resulting in result being assigned the value 6.

4. Print the result to the console using cout.

**Output:**



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

A: #include <iostream>

int main() {

int x = 5;

// Implicit casting (automatic conversion)

float y = x;

std::cout << "Implicit casting: " << std::fixed << y << std::endl;

// Explicit casting (using static\_cast)

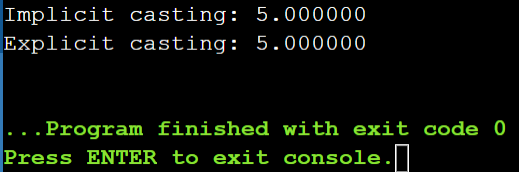
float z = static\_cast<float>(x);

std::cout << "Explicit casting: " << std::fixed << z << std::endl;

return 0;

}

**Output:**



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

A: #include <iostream>

class Animal {

public:

virtual void sound() = 0;

};

class Dog : public Animal {

public:

void sound() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void sound() override {

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

}

};

int main() {

Animal\* animal = new Dog(); // Create a Dog object through Animal pointer

// Check if animal is a Dog

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

if (dog != nullptr) {

std::cout << "animal is a Dog" << std::endl;

dog->sound(); // Outputs: Woof!

} else {

std::cout << "animal is not a Dog" << std::endl;

}

animal = new Cat(); // Create a Cat object through Animal pointer

// Check if animal is a Cat

Cat\* cat = dynamic\_cast<Cat\*>(animal);

if (cat != nullptr) {

std::cout << "animal is a Cat" << std::endl;

cat->sound(); // Outputs: Meow!

} else {

std::cout << "animal is not a Cat" << std::endl;

}

return 0;

}

In this scenario:

We have an Animal base class with a virtual sound() method.

Dog and Cat classes inherit from Animal and implement the sound() method.

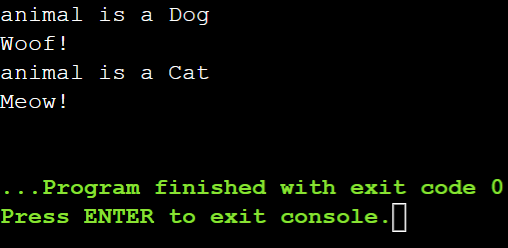
In main(), we create objects of Dog and Cat through an Animal pointer.

We use dynamic\_cast to check if the animal pointer is pointing to a Dog or Cat object.

If the cast is successful (i.e., the pointer is indeed pointing to a Dog or Cat object), we call the sound() method on the casted pointer.

If the cast fails (i.e., the pointer is not pointing to a Dog or Cat object), we print an error message.

**Output:**



**19. Discuss situations where using reinterpret\_cast might be justified, considering its potential risks.**

A: using reinterpret\_cast might be justified, along with considerations of its potential risks:

**1. Low-level programming:** When working with low-level programming, such as device drivers, embedded systems, or memory-mapped I/O, reinterpret\_cast might be necessary to cast between pointer types and integer types.

**2. Serialization and deserialization:** When serializing and deserializing data, reinterpret\_cast can be used to cast between pointer types and byte arrays.

**3. Legacy code:** When working with legacy code that uses outdated casting practices, reinterpret\_cast might be necessary to maintain compatibility.

**4. Performance-critical code:** In performance-critical code, reinterpret\_cast can be used to avoid overhead of virtual functions or runtime type checking.

Risks:

**1. Undefined behavior:** Using reinterpret\_cast can lead to undefined behavior if the cast is not valid.

**2. Type punning**: reinterpret\_cast can be used to perform type punning, which can lead to unexpected behavior.

**3. Loss of type safety**: reinterpret\_cast bypasses type safety checks, which can lead to errors.

**4. Maintenance and debugging:** Code using reinterpret\_cast can be harder to maintain and debug.

**20. Compare and contrast type casting with type conversion in?**

A: Type casting and type conversion are related but distinct concepts in programming:

**Type Casting:**

* + Explicitly forces a value of one type to be treated as another type
  + Does not change the underlying value or memory representation
  + Only changes the interpretation of the value
  + Typically uses casting operators (e.g., static\_cast, dynamic\_cast, reinterpret\_cast)

**Type Conversion:**

* + Changes the underlying value or memory representation from one type to another
  + Creates a new value or object with a different type
  + Can involve data loss or transformation (e.g., truncation, rounding)
  + Often implicit (automatic) or explicit (using conversion functions or VECTORS constructors).

VECTORS

Example:

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

// 1. Construction

std::vector<int> vec1; // Default constructor

std::vector<int> vec2(10, 5); // Fill constructor (10 elements with value 5)

std::vector<int> vec3{1, 2, 3, 4, 5}; // Initializer list constructor

std::vector<int> vec4(vec3.begin(), vec3.end()); // Range constructor

std::vector<int> vec5(vec3); // Copy constructor

std::vector<int> vec6(std::move(vec5)); // Move constructor

// 2. Assignment

vec1 = vec2; // Copy assignment

vec1 = std::move(vec2); // Move assignment

vec1 = {10, 20, 30}; // Initializer list assignment

// 3. Element Access

std::cout << "Element at index 1: " << vec1[1] << std::endl; // Operator[]

std::cout << "Element at index 2: " << vec1.at(2) << std::endl; // at()

std::cout << "First element: " << vec1.front() << std::endl; // front()

std::cout << "Last element: " << vec1.back() << std::endl; // back()

int\* data = vec1.data(); // data()

std::cout << "Element via data pointer: " << data[0] << std::endl;

// 4. Iterators

std::cout << "Elements in vec1: ";

for (auto it = vec1.begin(); it != vec1.end(); ++it) { // begin() and end()

std::cout << \*it << " ";

}

std::cout << std::endl;

std::cout << "Elements in reverse: ";

for (auto it = vec1.rbegin(); it != vec1.rend(); ++it) { // rbegin() and rend()

std::cout << \*it << " ";

}

std::cout << std::endl;

// 5. Capacity

std::cout << "Size: " << vec1.size() << std::endl; // size()

std::cout << "Capacity: " << vec1.capacity() << std::endl; // capacity()

std::cout << "Is empty: " << vec1.empty() << std::endl; // empty()

vec1.resize(5); // resize()

std::cout << "Resized vec1 size: " << vec1.size() << std::endl;

vec1.reserve(20); // reserve()

std::cout << "Reserved capacity: " << vec1.capacity() << std::endl;

// 6. Modifiers

vec1.assign(7, 100); // assign()

vec1.push\_back(200); // push\_back()

vec1.pop\_back(); // pop\_back()

vec1.insert(vec1.begin() + 1, 300); // insert()

vec1.erase(vec1.begin() + 2); // erase()

vec1.emplace(vec1.begin(), 400); // emplace()

vec1.emplace\_back(500); // emplace\_back()

vec1.swap(vec3); // swap()

vec1.clear(); // clear()

// 7. Non-member Functions

std::cout << "Is vec1 == vec3? " << (vec1 == vec3) << std::endl; // operator==

std::swap(vec1, vec3); // swap()

std::cout << "Elements after swap: ";

for (const auto& elem : vec1) {

std::cout << elem << " ";

}

std::cout << std::endl;

// 8. Algorithms

std::sort(vec1.begin(), vec1.end()); // sort()

std::cout << "Sorted elements: ";

for (const auto& elem : vec1) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

1.Imagine you're building a program to manage a list of tasks. Each task is represented by a Task object containing details like description, priority, and due date. You want to add tasks to a vector that stores these Task objects.

Challenge:

You have two options for adding new tasks:

Pre-created Tasks: You might have a pre-defined Task object with all its details set.

Creating Tasks on the Fly: You might need to create a new Task object on the fly while adding it to the vector, specifying the details during insertion.

Understanding the Difference:

insert: Use this if you already have a complete Task object ready to be inserted. insert takes the existing Task object and places it at a specific position in the vector. This might involve copying the object's data.

emplace: Use this if you need to create a new Task object with specific details while adding it to the vector. emplace calls the Task constructor directly within the vector's memory, initializing the new object with the provided values. This avoids unnecessary copying.

#include <iostream>

#include <vector>

#include <string>

#include <chrono>

#include <iomanip>

using namespace std;

struct Task {

string description;

int priority;

string dueDate;

// Constructor

Task(const string& desc, int prio, const string& due)

: description(desc), priority(prio), dueDate(due) {}

};

void printTask(const Task& task) {

cout << "Description: " << task.description

<< ", Priority: " << task.priority

<< ", Due Date: " << task.dueDate << endl;

}

void printTasks(const vector<Task>& tasks) {

for (const auto& task : tasks) {

printTask(task);

}

}

int main() {

vector<Task> taskList;

// Adding pre-created tasks using insert

Task preCreatedTask("Complete homework", 1, "2024-07-10");

taskList.insert(taskList.begin(), preCreatedTask);

// Adding tasks on the fly using emplace\_back

taskList.emplace\_back("Write project report", 2, "2024-07-15");

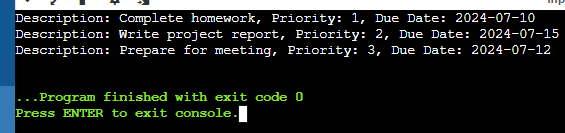
taskList.emplace\_back("Prepare for meeting", 3, "2024-07-12");

// Displaying all tasks

printTasks(taskList);

return 0;

}

OUTPUT:  


2. Design and implement a C++ program that utilizes vectors to efficiently store and manage student exam data. The program should allow for:

Adding new students with their names, IDs, and scores.

Finding a student by name or ID.

Calculating and displaying the average score for a specific student or for the entire class.

(Optional) Modifying existing student data (e.g., adding a new score).

#include <iostream>

#include <vector>

#include <string>

#include <numeric>

using namespace std;

struct Student {

string name;

int id;

vector<double> scores;

double averageScore() const {

if (scores.empty()) return 0.0;

return accumulate(scores.begin(), scores.end(), 0.0) / scores.size();

}

};

void addStudent(vector<Student>& students, const string& name, int id, const vector<double>& scores) {

students.push\_back({name, id, scores});

}

Student\* findStudentByName(vector<Student>& students, const string& name) {

for (auto& student : students) {

if (student.name == name) {

return &student;

}

}

return nullptr; }

Student\* findStudentById(vector<Student>& students, int id) {

for (auto& student : students) {

if (student.id == id) {

return &student;

} }

return nullptr;

}

void displayStudent(const Student& student) {

cout << "Name: " << student.name << ", ID: " << student.id << ", Scores: ";

for (double score : student.scores) {

cout << score << " "; }

cout << ", Average Score: " << student.averageScore() << endl;

}

void displayClassAverage(const vector<Student>& students) {

if (students.empty()) {

cout << "No students in the class." << endl;

return;

}

double totalAverage = 0.0;

for (const auto& student : students) {

totalAverage += student.averageScore();

}

cout << "Class Average Score: " << totalAverage / students.size() << endl;

}

int main() {

vector<Student> students;

// Adding new students

addStudent(students, "Harry", 101, {90.0, 85.5, 88.0});

addStudent(students, "Reeta", 102, {78.5, 82.0, 80.0});

string nameToFind = "Harry";

Student\* student = findStudentByName(students, nameToFind);

if (student) {

displayStudent(\*student);

} else {

cout << "Student with name " << nameToFind << " not found." << endl;

}

int idToFind = 102;

student = findStudentById(students, idToFind);

if (student) {

displayStudent(\*student);

} else {

cout << "Student with ID " << idToFind << " not found." << endl; }

displayClassAverage(students);

return 0;

}

OUTPUT:

