**Date:09/07/2024**

**Day\_12\_Assignments**

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

Type casting in C++ is the process of converting a variable from one data type to another. This is useful in various situations, such as when you need to perform operations on different types of variables or when interfacing with different parts of a program that expect specific data types.

There are two main types of type casting in C++:

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

* + This type of casting is done automatically by the compiler.
  + It occurs when data is assigned from a smaller to a larger data type, or from a derived class pointer/reference to a base class pointer/reference.

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

* This type of casting is done explicitly by the programmer using cast operators.
* C++ provides several ways to perform explicit type casting:
  + **C-Style Cast**: (type)value
  + **Functional Cast**: type(value)
  + **Static Cast**: static\_cast<type>(value)
  + **Dynamic Cast**: dynamic\_cast<type>(value) (used mainly with pointers and references to polymorphic types)
  + **Const Cast**: const\_cast<type>(value) (used to add or remove the const qualifier)
  + **Reinterpret Cast**: reinterpret\_cast<type>(value) (used for low-level reinterpreting of bit patterns)

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

**Implicit Type Casting (Automatic Type Conversion)**

1. **Automatic**:
   * The compiler automatically handles implicit type casting.
   * It occurs without any explicit instruction from the programmer.
2. **Safety and Predictability**:
   * Implicit conversions are usually safe and predictable, converting smaller types to larger types (e.g., int to float).

### Explicit Type Casting (Manual Type Conversion)

1. **Manual**:
   * The programmer explicitly specifies the type conversion.
   * It is done using cast operators such as (type), static\_cast<type>, dynamic\_cast<type>, const\_cast<type>, and reinterpret\_cast<type>.
2. **Control**:
   * The programmer has complete control over explicit type casting, making it possible to convert between incompatible types.

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

Implicit type casting, also known as automatic type conversion, in C++ occurs when the compiler automatically converts one data type to another. This is typically done to ensure compatibility between different data types during operations. Here are some common scenarios where implicit type casting is used:

1. **Arithmetic Operations:** When performing arithmetic operations involving different data types, C++ implicitly converts the operands to a common type to perform the operation.
2. **Assignment:** When assigning a value of one type to a variable of another type, implicit conversion may occur.
3. **Function Calls:** When passing arguments to a function, if the argument types do not match the parameter types, implicit conversion may occur to match the parameter types.
4. **Mixed-Type Expressions:** In expressions involving multiple types, C++ will perform implicit type conversions to ensure consistency in the operation.
5. **Boolean Context:** Any non-boolean type used in a boolean context (like if or while conditions) is implicitly converted to bool.

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

In C++, you can explicitly cast an integer to a float using various methods. Here are the most common approaches:

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. Function-Style Cast:

int i = 42;

float f = float(i);

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

**Loss of Precision:** When casting from a larger or more precise type to a smaller or less precise type, you may lose information.

cpp

Copy code

double d = 9.99;

int i = static\_cast<int>(d); // 'i' becomes 9, losing the fractional part.

**Data Overflow or Underflow:** Casting values that are outside the range of the target type can lead to overflow or underflow, resulting in unexpected values.

cpp

Copy code

long long largeValue = 9223372036854775807LL; // Max value for long long

int i = static\_cast<int>(largeValue); // Overflow, value becomes unpredictable

**Type Safety:** Casting can circumvent the type safety features of C++, potentially leading to undefined behavior.

cpp

Copy code

void\* ptr = &d;

int\* intPtr = static\_cast<int\*>(ptr); // Unsafe, type mismatch

**Undefined Behavior:** Casting between incompatible types can result in undefined behavior, which may cause crashes or other erratic behavior.

cpp

Copy code

int i = 42;

float\* fPtr = reinterpret\_cast<float\*>(&i); // Undefined behavior

**Hard-to-Debug Errors:** Explicit casts can obscure the original intent of the code, making it harder to understand and debug, especially in complex codebases.

**Compiler Warnings:** Explicit casts can suppress compiler warnings, potentially hiding issues that would otherwise be caught.

s **Platform-Dependent Behavior:** The result of some casts might be dependent on the platform or compiler, leading to non-portable code.

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

**static\_cast**:

* **Purpose**: Used for conversions between related types.
* **Use Cases**:
  + Converting between arithmetic types (e.g., int to float).
  + Converting pointers and references within an inheritance hierarchy.
  + Performing explicit user-defined conversions.
* **Safety**: Checked at compile-time; safer than C-style casts since it provides better type checking.

**dynamic\_cast**:

* **Purpose**: Used for safe downcasting in polymorphic hierarchies (classes with virtual functions).
* **Use Cases**:
  + Safely casting a base class pointer or reference to a derived class pointer or reference.
* **Safety**: Checked at runtime; returns nullptr for pointers or throws std::bad\_cast for references if the cast fails.

**const\_cast**:

* **Purpose**: Used to add or remove const or volatile qualifiers.
* **Use Cases**:
  + Removing const from a pointer or reference to modify a value that was originally constant.
  + Adding const to a pointer or reference to enforce immutability.
* **Safety**: Should be used with caution; modifying a value that was originally declared as const can lead to undefined behavior.

**reinterpret\_cast**:

* **Purpose**: Used for low-level reinterpreting of bit patterns.
* **Use Cases**:
  + Casting one pointer type to another unrelated pointer type.
  + Casting between pointer and integer types.
  + Dangerous and should be used sparingly, usually for system-level or hardware-specific code.
* **Safety**: Not type-safe; can lead to undefined behavior if used incorrectly.

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

**Arithmetic Conversions:** When converting between different arithmetic types, such as int to float or double to int.

**Pointer Upcasting and Downcasting in Inheritance Hierarchies:** When converting pointers or references within a class hierarchy, such as casting a base class pointer to a derived class pointer and vice versa. Note that downcasting from a base class to a derived class should be done only when you are sure of the type.

**Enum Conversions:** When converting between an enum and its underlying integral type.

**Explicit User-Defined Type Conversions:** When you have user-defined type conversions through constructors or type conversion operators.

**Removing const Qualifiers:** When you need to add or remove const qualifiers. However, this is more commonly done with const\_cast.

**Converting Pointers to Void and Back:** When converting a typed pointer to void\* and then back to the original type.

**Q.9 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. This can be necessary in specific scenarios where you need to modify or access a variable that is originally defined as const or volatile. Here are some common uses and purposes for const\_cast:

### 1. ****Removing**** const ****for Legacy Code or APIs****

Sometimes you need to pass a const variable to a function that does not accept const arguments, especially when dealing with legacy code or third-party libraries that do not use const correctness.

### ****2.Modifying a**** const ****Member Variable in a Member Function****

When you need to modify a member variable in a member function that is marked as const, const\_cast can be used. This is generally rare and often considered bad practice, but there are situations where it might be necessary.

### 3. ****Interfacing with APIs Expecting Non-const Parameters****

When dealing with APIs that require non-const parameters, but the original data is const.

### ****4.Removing Volatile Qualifiers****

const\_cast can also be used to add or remove volatile qualifiers.

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

reinterpret\_cast in C++ is a powerful tool but should be used with caution due to the following dangers:

1. **Undefined Behavior**: Using reinterpret\_cast can lead to undefined behavior if you cast between unrelated types or if the cast violates alignment requirements. For example, casting between pointers to unrelated types or casting away constness can result in unpredictable behavior.
2. **Type Safety Violation**: Unlike static\_cast or dynamic\_cast, reinterpret\_cast bypasses type checking during compilation. This means the compiler does not ensure the validity of the cast, which can lead to runtime errors if the casted types are not compatible.
3. **Platform Dependency**: The behavior of reinterpret\_cast can vary across different platforms and compilers. What works on one platform or compiler may not work correctly on another, leading to non-portable code.
4. **Code Readability**: Overuse of reinterpret\_cast can make code harder to understand and maintain because it doesn't clearly indicate the intent of the code (whether it's a low-level memory manipulation or a genuine type conversion).

**Q.11 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++. The explicit cast operators in C++ are:

1. **Static Cast**: This is the most commonly used cast and is used for conversions that are well-defined by the language, such as numeric conversions or casting between related classes in a class hierarchy.
2. **Dynamic Cast**: This cast is used for safe downcasting in class hierarchies with polymorphic types (types that have virtual functions). It performs a runtime check to ensure the validity of the cast.
3. **Const Cast**: This cast is used to add or remove const or volatile qualifiers from a pointer or reference.
4. **Reinterpret Cast**: This is the most powerful and dangerous cast, which allows casting between unrelated types, such as converting between pointers to different types or between pointer and integral types.

**Q.12 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++, such as casting from a larger integer type to a smaller integer type (like casting from long long to int), data loss can occur due to the following reasons:

1. **Overflow**: If the value of the larger data type exceeds the range that can be represented by the smaller data type, overflow occurs. For example, if you have a long long variable with a value greater than what can be represented by an int, casting it to int will truncate the higher-order bits, leading to a loss of data.
2. **Loss of Precision**: Casting from a floating-point type (like double) to an integer type (like int) can lead to a loss of fractional part. For example, casting 3.5 (a double) to int results in 3, losing the 0.5 fractional part.
3. **Truncation**: When casting from a floating-point type to an integer type, the decimal part is truncated. This means only the integer part of the floating-point value is retained, discarding any fractional part.

To handle potential data loss when casting from a larger to a smaller data type, it's crucial to:

* **Ensure Range Safety**: Check if the value of the larger type falls within the acceptable range of the smaller type before casting.
* **Consider Loss of Precision**: Be aware of the loss of fractional parts when casting from floating-point types to integer types.
* **Use Bounds Checking**: Consider using conditional checks or methods to ensure that the value being cast is within the allowable range of the smaller data type to prevent overflow.

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

When using dynamic\_cast in C++, you can check if a type casting operation is successful by examining the result of the cast. Here's how you can do it:

1. **For Pointers**: If the cast is successful, dynamic\_cast returns a pointer of the target type if the object pointed to is of the correct type (or a type derived from it). If the cast fails (i.e., the object isn't of the correct type), dynamic\_cast returns a null pointer (nullptr)
2. **For References**: When dynamic\_cast is used on references, if the cast is successful, it returns a reference to the target type.
3. If the cast fails, it throws a std::bad\_cast exception.In summary, to check if a dynamic\_cast operation is successful:

* For pointers, compare the result to nullptr.
* For references, handle the std::bad\_cast exception that might be thrown.

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

Implicit Type Casting. Implementing the implicit type casting is very easy in a program. We use it to convert the data type of any variable without losing the actual meaning that it holds. The implicit type casting occurs automatically.

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

static\_cast<T> is the first cast you should attempt to use. It is called static because the C++ standard requires compilers to validate static\_casts at compile-time. If the compiler cannot resolve the type conversion as valid, then it won't compile. The T is a type name such as int, a class name or struct.

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

#include <iostream>

int main() {

double pi = 3.14159;

int radius = 5;

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

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

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

return 0;

}

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

#include <iostream>

int main() {

int intValue = 42;

float floatValue;

floatValue = intValue; // Implicitly converts int to float

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

std::cout << "intValue: " << intValue << std::endl;

std::cout << "floatValue: " << floatValue << std::endl;

// Explicit casting

floatValue = static\_cast<float>(intValue); // Explicitly converts int to float

std::cout << "\nExplicit casting:" << std::endl;

std::cout << "intValue: " << intValue << std::endl;

std::cout << "floatValue: " << floatValue << std::endl;

return 0;

}

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

Let's create a scenario where dynamic\_cast is used to check inheritance relationships between classes in C++. Consider a base class Shape and two derived classes Circle and Rectangle. We'll use dynamic\_cast to verify if an object pointer is pointing to a Circle or Rectangle object.

#include <iostream>

// Base class

class Shape {

public:

virtual ~Shape() {} // Ensure a virtual destructor for polymorphic behavior

};

// Derived class Circle

class Circle : public Shape {

public:

void drawCircle() {

std::cout << "Drawing Circle..." << std::endl;

}

};

// Derived class Rectangle

class Rectangle : public Shape {

public:

void drawRectangle() {

std::cout << "Drawing Rectangle..." << std::endl;

}

};

int main() {

Circle circle;

Rectangle rectangle;

Shape\* shapePtr;

// Pointer to Circle object

shapePtr = &circle;

// Use dynamic\_cast to check if the object is a Circle

if (Circle\* circlePtr = dynamic\_cast<Circle\*>(shapePtr)) {

std::cout << "shapePtr points to a Circle." << std::endl;

circlePtr->drawCircle();

} else {

std::cout << "shapePtr does not point to a Circle." << std::endl;

}

// Pointer to Rectangle object

shapePtr = &rectangle;

// Use dynamic\_cast to check if the object is a Rectangle

if (Rectangle\* rectPtr = dynamic\_cast<Rectangle\*>(shapePtr)) {

std::cout << "shapePtr points to a Rectangle." << std::endl;

rectPtr->drawRectangle();

} else {

std::cout << "shapePtr does not point to a Rectangle." << std::endl;

}

return 0;

}

**Q.19 Compare and contrast type casting with type conversion in**

* **Control**: Type casting provides explicit control over how types are converted, allowing for conversions that are not automatically handled by the compiler. Type conversion, on the other hand, relies on compiler rules and happens automatically in many cases.
* **Safety**: Type casting can be unsafe if used incorrectly, especially with reinterpret\_cast or const\_cast. Type conversion is generally safer because it follows compiler-defined rules and ensures data integrity.
* **Use Cases**: Type casting is used when you need to perform specific conversions or bypass type safety checks (e.g., converting between unrelated types with reinterpret\_cast). Type conversion is used for everyday conversions handled automatically by the compile.

Q.20

#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;

}

Q.21

#include<iostream>

#include<vector>

#include<string>

struct Product{

std::string name;

double price;

}

int main()

{

std::vector<product>cart;

product apple={"Apple" 1.99};

cart.insert(cart.begin(),Apple);

cart.emplace\_back("banana",0.79);

for(const product& item : )

}

Q.22 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>

class Task {

public:

std::string description;

int priority;

std::string due\_date;

Task(const std::string& desc, int pri, const std::string& date)

: description(desc), priority(pri), due\_date(date) {}

void print() const {

std::cout << "Description: " << description

<< ", Priority: " << priority

<< ", Due Date: " << due\_date << std::endl;

}

};

int main() {

std::vector<Task> tasks;

// Pre-created Task

Task task1("Task 1", 1, "2024-07-15");

tasks.insert(tasks.begin(), task1); // Using insert with a pre-created Task object

// Creating Task on the Fly

tasks.emplace\_back("Task 2", 2, "2024-07-20"); // Using emplace to create and add Task

// Another pre-created Task

Task task3("Task 3", 3, "2024-07-25");

tasks.insert(tasks.end(), task3); // Insert at the end

// Creating Task on the Fly at a specific position

tasks.emplace(tasks.begin() + 1, "Task 4", 4, "2024-07-30"); // Emplace at position 1

// Display all tasks

std::cout << "Task List:" << std::endl;

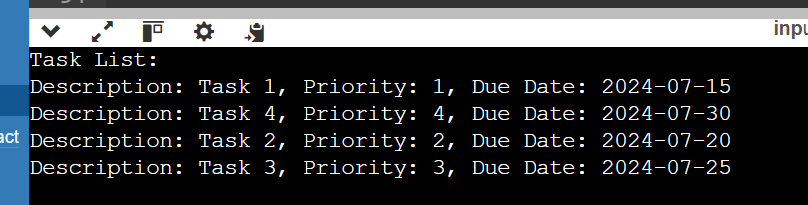
for (const auto& task : tasks) {

task.print();

}

return 0;

}



Q.23 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 <algorithm>

#include <numeric>

class Student {

public:

std::string name;

int id;

std::vector<int> scores;

Student(const std::string& name, int id, const std::vector<int>& scores)

: name(name), id(id), scores(scores) {}

void addScore(int score) {

scores.push\_back(score);

}

double getAverageScore() const {

if (scores.empty()) return 0.0;

int total = std::accumulate(scores.begin(), scores.end(), 0);

return static\_cast<double>(total) / scores.size();

}

void print() const {

std::cout << "Name: " << name << ", ID: " << id << ", Scores: ";

for (const auto& score : scores) {

std::cout << score << " ";

}

std::cout << "Average Score: " << getAverageScore() << std::endl;

}

};

class StudentManager {

private:

std::vector<Student> students;

public:

void addStudent(const std::string& name, int id, const std::vector<int>& scores) {

students.emplace\_back(name, id, scores);

}

Student\* findStudentByName(const std::string& name) {

auto it = std::find\_if(students.begin(), students.end(), [&name](const Student& s) {

return s.name == name;

});

return it != students.end() ? &(\*it) : nullptr;

}

Student\* findStudentById(int id) {

auto it = std::find\_if(students.begin(), students.end(), [id](const Student& s) {

return s.id == id;

});

return it != students.end() ? &(\*it) : nullptr;

}

double getClassAverage() const {

if (students.empty()) return 0.0;

double totalScore = 0.0;

int totalStudents = 0;

for (const auto& student : students) {

totalScore += student.getAverageScore();

totalStudents++;

}

return totalScore / totalStudents;

}

void printAllStudents() const {

for (const auto& student : students) {

student.print();

}

}

};

int main() {

StudentManager manager;

// Adding students

manager.addStudent("ram", 1, {90, 85, 88});

manager.addStudent("shayam", 2, {78, 82, 80});

manager.addStudent("dev", 3, {92, 91, 89});

// Finding and displaying a student by name

Student\* student = manager.findStudentByName("ram");

if (student) {

std::cout << "Found student by name 'ram':" << std::endl;

student->print();

} else {

std::cout << "Student 'ram' not found." << std::endl;

}

// Finding and displaying a student by ID

student = manager.findStudentById(2);

if (student) {

std::cout << "Found student by ID '2':" << std::endl;

student->print();

} else {

std::cout << "Student with ID '2' not found." << std::endl;

}

// Displaying the average score for the entire class

std::cout << "Class average score: " << manager.getClassAverage() << std::endl;

student = manager.findStudentByName("dev");

if (student) {

student->addScore(95);

std::cout << "Updated student 'dev' with a new score:" << std::endl;

student->print();

}

std::cout << "All students:" << std::endl;

manager.printAllStudents();

return 0;

}

