Day-12:

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

Type casting in C++ is the process of converting one data type to another. It can be done implicitly (automatically by the compiler) or explicitly (manually by the programmer using casting operators).

**Implicit Type Casting:**

* 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:**

* 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 Casting (Coercion):**

* **Definition:** Implicit type casting is performed automatically by the compiler when it encounters expressions where the types do not match but the conversion is considered safe and lossless.
* **Example:** Converting an int to a float in an arithmetic operation (int a = 5; float b = a / 2;).

**Explicit Type Casting (Type Casting):**

* **Definition:** Explicit type casting is done manually by the programmer using casting operators to convert a variable from one type to another.
* **Example:** Converting a double to an int using a cast operator (double d = 3.14; int i = (int)d;).

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

Implicit type casting in C++ is used automatically by the compiler in these situations:

1. **Arithmetic Operations**: When you mix different types like integers (int) and floating-point numbers (float, double) in calculations, the compiler converts smaller types to larger ones to avoid losing precision.
2. **Assignments**: When you assign a value of one type to a variable of another type, if the types are compatible, the compiler handles the conversion for you.
3. **Function Calls**: When you pass arguments to functions, the compiler may convert them to match the expected parameter types if it's safe to do so.
4. **Comparisons**: When you compare values of different types, the compiler may convert one type to match the other for the comparison.

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

Explicit type casting in C++ allows the programmer to convert one data type to another using casting operators. To convert an integer (int) to a float (float), you can use the functional notation casting operator

int integerValue = 10;

float floatValue = float(integerValue);

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

* **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++.

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:

1. **static\_cast**
2. **dynamic\_cast**
3. **const\_cast**
4. **reinterpret\_cast**

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

This is the simplest type of cast that can be used. It is a **compile-time cast**. It does things like implicit conversions between types (such as int to float, or pointer to void\*), and it can also call explicit conversion functions.

Convert between numeric types like int, float, double, etc., where no data loss or narrowing occurs.

 Example: static\_cast<double>(integerValue)

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

 It's used in polymorphic base classes and requires at least one [virtual function](https://unstop.com/blog/virtual-function-in-cpp) in the base class. Type casting in C++ with dynamic\_cast ensures the validity of the downcast and returns a valid pointer if the cast is successful

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

const\_cast is one of the type casting operators. It is used to change the constant value of any object or we can say it is used to remove the constant nature of any object.

const\_cast can be used in programs that have any object with some constant value which need to be changed occasionally at some point.

Syntax:

const\_cast<type name>(expression)

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

reinterpret\_cast is a powerful casting operator in C++ that allows for casting between unrelated types, but it can be dangerous if used improperly. Here are some dangers of using reinterpret\_cast and why it should be used with caution:

**1. Loss of type safety:** By casting between unrelated types, reinterpret\_cast bypasses the type system's safety checks, potentially leading to undefined behavior.

**2. Pointer aliasing:** reinterpret\_cast can create multiple aliases for the same memory location, leading to confusion and errors.

**3. Data corruption**: Casting between types with different sizes or representations can lead to data corruption or truncation.

**4. Undefined behavior:** Using reinterpret\_cast can lead to undefined behavior, especially when casting between types that are not related by a casting relationship.

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++. 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**?**

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?

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();

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

if (derived != nullptr)

} else {

}

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?

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?

**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.

**#include <iostream>**

**int main() {**

**int intValue = 10;**

**double doubleValue = 3.5;**

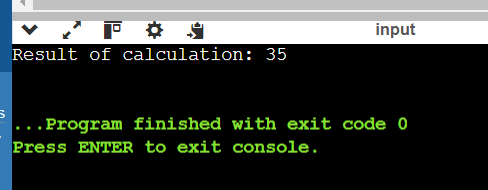
**double result = static\_cast<double>(intValue) \* doubleValue;**

**std::cout << "Result of calculation: " << result << std::endl;**

**return 0;**

**}**

**Output:**



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

#include <iostream>

int main() {

int x = 5;

float y = x;

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

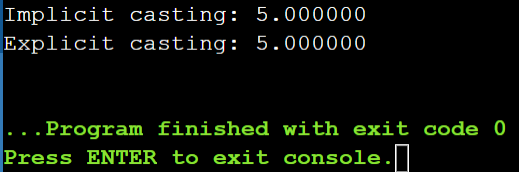
float z = static\_cast<float>(x);

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.

#include <iostream>

class Animal {

public:

virtual ~Animal() {}

};

class Dog : public Animal {

public:

void bark() {

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

}

};

class Cat : public Animal {

public:

void meow() {

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

}

};

int main() {

Animal\* animalPtr;

Dog myDog;

animalPtr = &myDog;

if (Dog\* dogPtr = dynamic\_cast<Dog\*>(animalPtr)) {

std::cout << "Animal is a Dog." << std::endl;

dogPtr->bark();

} else {

std::cout << "Animal is not a Dog." << std::endl;

}

Cat myCat;

animalPtr = &myCat;

if (Cat\* catPtr = dynamic\_cast<Cat\*>(animalPtr)) {

std::cout << "Animal is a Cat." << std::endl;

catPtr->meow();

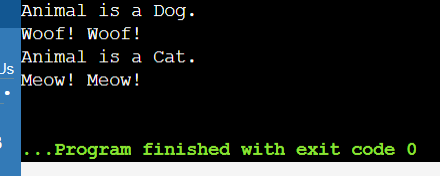
} else {

std::cout << "Animal is not a Cat." << std::endl;

}

return 0;

}



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

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

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 constructors)

#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();

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());

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

for (const auto& elem : vec1) {

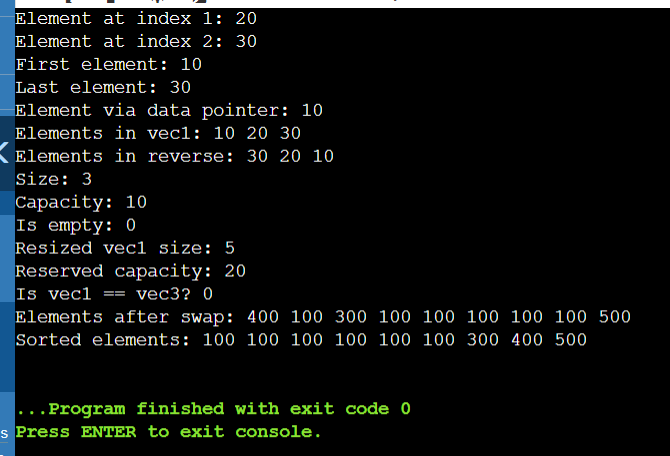
std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

} **output:**



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 {

private:

std::string description;

int priority;

std::string dueDate;

public:

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

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

void displayTask() const {

std::cout << "Description: " << description << ", Priority: " << priority

<< ", Due Date: " << dueDate << "\n";

}

};

class TaskManager {

private:

std::vector<Task> tasks;

public:

void insertTask(const Task& task) {

tasks.push\_back(task);

}

void emplaceTask(const std::string& desc, int prio, const std::string& date) {

tasks.emplace\_back(desc, prio, date);

}

void displayAllTasks() const {

for (const auto& task : tasks) {

task.displayTask();

}

}

};

int main() {

TaskManager manager;

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

manager.insertTask(task1);

manager.emplaceTask("Prepare presentation slides", 2, "2024-07-20");

manager.emplaceTask("Review meeting agenda", 3, "2024-07-18");

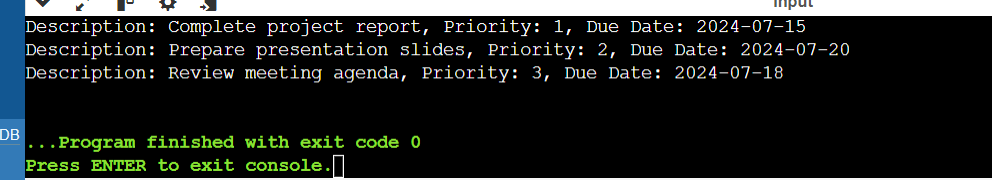
std::cout << "All tasks:\n";

manager.displayAllTasks();

return 0;

}

Output:



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>

struct Student {

std::string name;

int id;

std::vector<int> scores;

Student(const std::string& n, int i) : name(n), id(i) {}

};

class ExamManager {

private:

std::vector<Student> students;

public:

void addStudent(const std::string& name, int id, int score) {

Student newStudent(name, id);

newStudent.scores.push\_back(score);

students.push\_back(newStudent);

}

void findStudentByName(const std::string& name) {

for (const auto& student : students) {

if (student.name == name) {

std::cout << "Student found: " << student.name << " (ID: " << student.id << ")\n";

return;

}

}

std::cout << "Student not found.\n";

}

void findStudentByID(int id) {

for (const auto& student : students) {

if (student.id == id) {

std::cout << "Student found: " << student.name << " (ID: " << student.id << ")\n";

return;

}

}

std::cout << "Student not found.\n";

}

void calculateAverageScore(const std::string& name) {

for (const auto& student : students) {

if (student.name == name) {

int sum = 0;

for (int score : student.scores) {

sum += score;

}

double average = static\_cast<double>(sum) / student.scores.size();

std::cout << "Average score for " << student.name << ": " << average << "\n";

return;

}

}

std::cout << "Student not found.\n";

}

void calculateClassAverage() {

int totalSum = 0;

int totalScores = 0;

for (const auto& student : students) {

for (int score : student.scores) {

totalSum += score;

totalScores++;

}

}

double classAverage = static\_cast<double>(totalSum) / totalScores;

std::cout << "Class average score: " << classAverage << "\n";

}

void modifyStudentScore(const std::string& name, int newScore) {

for (auto& student : students) {

if (student.name == name) {

student.scores.push\_back(newScore);

std::cout << "Score updated for " << student.name << ".\n";

return;

}

}

std::cout << "Student not found.\n";

}

};

int main() {

ExamManager manager;

manager.addStudent("John Doe", 1, 80);

manager.addStudent("Jane Smith", 2, 90);

manager.addStudent("Bob Johnson", 3, 70);

manager.findStudentByName("Jane Smith");

manager.findStudentByID(2);

manager.calculateAverageScore("John Doe");

manager.calculateClassAverage();

manager.modifyStudentScore("Jane Smith", 95);

manager.calculateAverageScore("Jane Smith");

return 0;

}

Output:

