Q. What is type casting in C++ and what are the two main types?
Ans: type casting is the process of converting one data type to another data type .
There are two type of type casting:-

1. Implicit casting
2. Explicit casting

Q.Explain the difference between implicit and explicit type casting.
Ans: 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 the programmer's intervention.

Q.When would you use implicit type casting in C++?
Ans: Here are a few scenarios where implicit type casting is commonly used:

Promoting smaller data types to larger ones:
When performing arithmetic operations between different numeric types, the compiler automatically promotes the smaller type to the larger type to prevent data loss. For example, adding an integer to a double will automatically promote the integer to a double before performing the addition.
Converting between numeric types:
The compiler can implicitly convert between different numeric types like int, float, and double. For example, assigning an integer value to a float variable will automatically convert the integer to a float.
Converting between numeric and character types:
A character can be implicitly converted to its corresponding integer ASCII value.

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

Ans:
```
#include <iostream>

int main() {

  int num = 42;

  float fnum = static_cast<float>(num);
```

```
    std::cout << "Integer: " << num << std::endl;

    std::cout << "Float: " << fnum << std::endl;

    return 0;

}
```

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

Ans: Explicit type casting, which involves manually converting values between different data types, can introduce some risks to the code. These include:

<u>Data loss</u>
When converting a larger data type to a smaller one, some data might be lost because smaller types can't hold all the values of larger types.
<u>Overflow and underflow</u>
There's a potential for overflow, underflow, and data loss, particularly when dealing with numeric data types.
<u>Runtime errors</u>
Incorrect explicit type casting can lead to unexpected behavior or runtime errors.
<u>Increased complexity</u>
Explicit type casting can make the code more complex, especially when dealing with multiple data type conversions.
<u>Verbose code</u>
Explicit type casting requires additional syntax or function calls, which can make the code more verbose.

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

Ans: static cast, dynamic cast, reinterpret cast, and const cast

Q. When should you use static_cast for type casting?

Ans: Conversions between numeric types:

1. This is the most common use case, such as converting an `int` to a `float`, or a `double` to an `int`.
2. Converting between related pointer types:
3. Converting between pointer and integer types:
4. Converting from `void`:*
5. Calling explicit conversion functions:

Q. In what scenario would you use dynamic_cast for type casting?

Ans: 1. Downcasting in Polymorphic Hierarchies:

2. Checking the Object's Actual Type:

3. Handling Runtime Errors:

4. Working with Unknown Object Types:

Q. Explain the purpose of const_cast and when it might be necessary.

Ans: `const_cast` is a C++ type casting operator used to add or remove the `const` qualifier from a variable. This means it can be used to modify a variable that was originally declared as constant or to pass a constant variable to a function that expects a non-constant argument.

Q. What are the dangers of using reinterpret_cast and why should it be used with caution?

Ans: reinterpret_cast is a type casting operator that allows conversions between almost any pointer or integral type, even if the data types are different. However, it's considered a dangerous operator and should be used sparingly.

Q. Discuss situations where using reinterpret_cast might be justified, considering its potential risks.

Ans: it can be justified in specific situations where you need to perform low-level type conversions that other cast operators cannot achieve. However, it comes with significant risks, such as type safety violations and undefined behavior if misused.

Q.Simulate a scenario where dynamic_cast is used for checking inheritance relationships between classes.

Ans: #include <iostream>

```
#include <memory>  // for std::unique_ptr



// Base class

class Animal {

public:

    virtual ~Animal() {}  // Ensure the base class has a virtual destructor

    virtual void speak() const = 0;

};
```

```
// Derived class Dog
class Dog : public Animal {
public:
    void speak() const override {
        std::cout << "Woof!" << std::endl;
    }
    void fetch() const {
        std::cout << "Fetching!" << std::endl;
    }
};

// Derived class Cat
class Cat : public Animal {
public:
    void speak() const override {
        std::cout << "Meow!" << std::endl;
```

```
    }
    void purr() const {
        std::cout << "Purring!" << std::endl;
    }
};

void identifyAndInteract(Animal* animal) {
    if (Dog* dog = dynamic_cast<Dog*>(animal)) {
        std::cout << "This is a dog." << std::endl;
        dog->speak();
        dog->fetch();
    } else if (Cat* cat = dynamic_cast<Cat*>(animal)) {
        std::cout << "This is a cat." << std::endl;
        cat->speak();
        cat->purr();
    } else {
        std::cout << "Unknown animal type." << std::endl;
```

```
    }
}

int main() {
    std::unique_ptr<Animal> myDog = std::make_unique<Dog>();
    std::unique_ptr<Animal> myCat = std::make_unique<Cat>();

    identifyAndInteract(myDog.get());  // Should identify and interact with Dog
    identifyAndInteract(myCat.get());  // Should identify and interact with Cat

    return 0;
}
```

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

Ans: #include <iostream>

```
void implicitCasting() {
    int intVal1 = 42;
    int intVal2 = 7;

    // Implicit casting during assignment
    float floatVal1 = intVal1;
    std::cout << "Implicit casting during assignment: " << floatVal1 << std::endl;

    // Implicit casting during arithmetic operation
    float floatResult = intVal1 / intVal2;  // Division of integers, result is implicitly cast to float
    std::cout << "Implicit casting during arithmetic operation (int / int): " << floatResult << std::endl;
}

void explicitCasting() {
    int intVal1 = 42;
    int intVal2 = 7;
```

```
    // Explicit casting during assignment

    float floatVal1 = static_cast<float>(intVal1);

    std::cout << "Explicit casting during assignment: " << floatVal1 << std::endl;



    // Explicit casting during arithmetic operation

    float floatResult = static_cast<float>(intVal1) / intVal2;  // One operand is
explicitly cast to float

    std::cout << "Explicit casting during arithmetic operation (float / int): " <<
floatResult << std::endl;

}



int main() {

    std::cout << "Demonstrating Implicit Casting:" << std::endl;

    implicitCasting();

    std::cout << std::endl;



    std::cout << "Demonstrating Explicit Casting:" << std::endl;
```

```
    explicitCasting();

    return 0;
}
```

Q. Create a code example that demonstrates the use of static_cast for performing a calculation.

Ans:
```
#include <iostream>

void calculateAreaWithStaticCast() {
    int radiusInt = 5;
    const double PI = 3.14159;

    // Calculate area using integer radius (incorrect)
    double areaInt = PI * radiusInt * radiusInt;  // This works but shows integer
usage

    // Calculate area using static_cast to convert integer to double (correct)
```

```
    double radiusDouble = static_cast<double>(radiusInt);

    double areaDouble = PI * radiusDouble * radiusDouble;



    std::cout << "Calculating Area of a Circle" << std::endl;

    std::cout << "Using integer radius (incorrect but straightforward): " << areaInt
<< std::endl;

    std::cout << "Using static_cast to convert integer to double (correct): " <<
areaDouble << std::endl;

}



int main() {

    calculateAreaWithStaticCast();

    return 0;

}
```

Q. How can you check if a type casting operation is successful with dynamic_cast?

Ans: #include <iostream>

```
class Base {

public:

    virtual ~Base() {}  // Ensure the base class has a virtual destructor

};


class Derived : public Base {

public:

    void derivedFunction() {

        std::cout << "Derived function called!" << std::endl;

    }

};


class Unrelated {};


void checkPointerCasting(Base* basePtr) {

    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
```

```
    if (derivedPtr) {
        std::cout << "Pointer cast successful!" << std::endl;
        derivedPtr->derivedFunction();
    } else {
        std::cout << "Pointer cast failed!" << std::endl;
    }
}

int main() {
    Base base;
    Derived derived;
    Unrelated unrelated;

    Base* basePtr = &base;
    Base* derivedBasePtr = &derived;

    std::cout << "Casting Base* to Derived*:" << std::endl;
```

```
    checkPointerCasting(basePtr);

    std::cout << "Casting Derived* (as Base*) to Derived*:" << std::endl;
    checkPointerCasting(derivedBasePtr);

    return 0;
}
```

ALL VECTOR API AND METHOD:

```
#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;
```

```cpp
}

// 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;

}

/*
```

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 <vector>

#include <string>

#include <iostream>

Using namespace std;

class Task {
```

```
public:
    string description;
    int priority;
    string dueDate;

    Task(const string& desc, int prio, const string& date)
        : description(desc), priority(prio), dueDate(date) {}

    void display() const {
        cout << "Task: " << description << ", Priority: " << priority << ", Due Date: " <<
dueDate << endl;
    }
};

int main() {
    vector<Task> tasks;
```

```
    // Using insert with a pre-created Task object
    Task preCreatedTask("Finish report", 1, "2024-07-15");
    tasks.insert(tasks.end(), preCreatedTask);


    // Using emplace to create a Task object on the fly
    tasks.emplace_back("Prepare presentation", 2, "2024-07-10");


    // Display all tasks
    for (const auto& task : tasks) {
        task.display();
    }


    return 0;
}
/*
```

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 <bits/stdc++.h>


using namespace std;

class StudentManage {

private:

    vector<Student> students;


public:

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

        students.emplace_back(name, id);
```

```
}

Student* findStudentByName(const string& name) {
    auto it = 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 = find_if(students.begin(), students.end(),
        [id](const Student& s) { return s.id == id; });
    return (it != students.end()) ? &(*it) : nullptr;
}

double calculateAverageScore() const {
    if (students.empty()) return 0.0;
    double totalScore = 0;
```

```
        int totalEntries = 0;

        for (const auto& student : students) {

            totalScore += accumulate(student.scores.begin(), student.scores.end(),
0);

            totalEntries += student.scores.size();

        }

        return totalEntries > 0 ? totalScore / totalEntries : 0.0;

    }



    void displayStudent(const Student* student) const {

        if (student) {

            student->display();

        } else {

            cout << "Student not found." << endl;

        }

    }
```

```
    void displayAllStudents() const {
        for (const auto& student : students) {
            student.display();
        }
    }
};

int main() {
    StudentManage manager;

    manager.addStudent("Sunny", 1);
    manager.addStudent("rohit", 2);

    Student* Sunny = manager.findStudentById(1);
    if (Sunny) {
        Sunny->addScore(85);
        Sunny->addScore(90);
```

```
}

Student* rohit = manager.findStudentById(2);
if (rohit) {
    rohit->addScore(78);
    rohit->addScore(82);
}

Student* student = manager.findStudentByName("Sunny");
manager.displayStudent(student);
cout << "Class average score: " << manager.calculateAverageScore() << endl;

manager.displayAllStudents();

if (student) {
    student->addScore(95);
}
```

```
    manager.displayStudent(student);

    return 0;
}
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>
#include <vector>
#include <string>
using namespace std;
```

```
int main()

{

    vector<product> cart;

    product apple={"apple", 1.99};

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

    cart.emplace_back(){banana, 0.79};



    return 0;

}
```