Type Conversion

What is Type Conversion in C++?

Type conversion in C++ refers to changing a value from one data type to another. It can occur **implicitly (automatically)** or **explicitly (manually)**.

Types of Type Conversion

1. Implicit Type Conversion (Automatic Type Conversion)

- Also called **Type Promotion**.
- Done automatically by the compiler.
- Converts a smaller data type to a larger data type to prevent data loss.

Example: Implicit Conversion

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

int main() {
    int num = 10;
    double d = num;  // int is implicitly converted to double
    long value = num;
    cout << "num as double: " << d << endl;
    return 0;
}
```

Output:

```
num as double: 10.0
```

✔️ No data loss occurs because `int` (4 bytes) is safely converted to `double` (8 bytes).

---

2. Explicit Type Conversion (Type Casting)

- Performed manually by the programmer.
- Uses **type casting** methods:
    - **C-style casting:** `(type)value`
    - `static_cast<type>(value)`
    - `dynamic_cast<type>(value)` (for polymorphism)
    - `reinterpret_cast<type>(value)` (for low-level memory operations)
    - `const_cast<type>(value)` (removes `const` qualifier)

1️⃣ `static_cast` (Compile-time Casting)

🔹 What is `static_cast`?

`static_cast` is used for **compile-time conversions**. It is **faster** but does **not check safety** at runtime.

📌 When to Use `static_cast`?

✔ **Converting between fundamental types** (int ↔ float, char ↔ int).

✔ **Upcasting in class hierarchy** (Derived → Base).

✔ **Converting void pointers (`void* → specific type*`).**

✔ **Converting enum to int and vice versa.**

📝 Example 1: Converting Data Types

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

int main() {
    float f = 5.7;
    int x = static_cast<int>(f);  // Converts float to int
    cout << "Original float: " << f << ", After static_cast: " << x << endl;
    return 0;
}
```

🛠 Output:

```
Original float: 5.7, After static_cast: 5
```

📝 Example 2: Upcasting (Safe)

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

class Animal {
public:
    void sound() { cout << "Some animal sound\n"; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Woof! Woof!\n"; }
};

int main() {
    Dog d;
    Animal* a = static_cast<Animal*>(&d); // Upcasting Dog → Animal
    a->sound(); // ✅ Works fine
    return 0;
}
```

🛠️ Output:

```
Some animal sound
```

2️⃣ `dynamic_cast` (Runtime Type Checking)

🔷 What is `dynamic_cast`?

`dynamic_cast` is used for **safe downcasting** in **polymorphic** class hierarchies (**Base → Derived**) at runtime.

📌 When to Use `dynamic_cast`?

✔️ **Checking object type at runtime** (Run-Time Type Identification - RTTI).

✔️ **Downcasting in class hierarchy** (Base → Derived).

✔️ **Ensuring the object is of the correct type before accessing derived class members.**

1️⃣ Detecting the Type of an Object at Runtime (RTTI)

🔷 **Scenario:** You have a list of different animal types and need to check if an object is a `Dog` or a `Cat` at runtime.

✅ **Using `dynamic_cast` (Safe Approach)**

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

class Animal {
public:
    virtual void makeSound() { cout << "Some animal sound\n"; }
};

class Dog : public Animal {
public:
    void makeSound() override { cout << "Woof! Woof!\n"; }
    void fetch() { cout << "🐶 Dog is fetching the ball!\n"; }
};

class Cat : public Animal {
public:
    void makeSound() override { cout << "Meow!\n"; }
};

void identifyAnimal(Animal* animal) {
    Dog* dog = dynamic_cast<Dog*>(animal);
    if (dog) {
        cout << "✅ This is a Dog!\n";
        dog->fetch();
        return;
    }
```

```
    Cat* cat = dynamic_cast<Cat*>(animal);
    if (cat) {
        cout << "✅ This is a Cat!\n";
        return;
    }

    cout << "❓ Unknown Animal\n";
}

int main() {
    vector<Animal*> animals = {new Dog(), new Cat(), new Animal()};

    for (Animal* animal : animals) {
        identifyAnimal(animal);
    }

    for (Animal* animal : animals) {
        delete animal;
    }

    return 0;
}
```

🛠️ Output:

```
✅ This is a Dog!
🐶 Dog is fetching the ball!
✅ This is a Cat!
❓ Unknown Animal
```

🔷 **Why `dynamic_cast`?**

Without it, we would have to use a **custom type-checking system** (`enum`, `virtual isType()` functions, etc.), making code more **error-prone and less maintainable**.

2️⃣ Preventing Invalid Function Calls (Avoid Undefined Behavior)

🔷 **Scenario:** Calling a function that doesn't exist in a wrongly casted object.

🚨 Without `dynamic_cast` (Undefined Behavior)

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

class Animal {
public:
    virtual void makeSound() { cout << "Some animal sound\n"; }
};

class Dog : public Animal {
public:
    void wagTail() { cout << "🐶 Dog is wagging tail!\n"; }
};

int main() {
    Animal* animal = new Animal();

    // Unsafe cast: Compiler won't stop you, but behavior is unpredictable!
    Dog* dog = (Dog*)animal;  // ❌ Using C-style cast (Unsafe!)

    dog->wagTail();  // ❌ Undefined behavior! Calls non-existent function.

    delete animal;
    return 0;
}
```

🚨 Potential Problems:

1. **Crash:** If the memory layout is incompatible, calling `dog->wagTail()` can **segfault**.
2. **Unexpected behavior:** You might execute garbage code.
3. **Hard-to-debug issues:** The compiler won't warn you!

✅ Using `dynamic_cast` (Safe)

```
Dog* dog = dynamic_cast<Dog*>(animal);
if (dog) {
```

```
    dog→wagTail();
} else {
    cout << "❌ Downcasting failed! This is not a Dog.\n";
}
```

3️⃣ Using `dynamic_cast` with References (Throws Exception)

🔷 **Scenario:** When working with references instead of pointers.

✅ **Example:**

```
#include <iostream>
#include <stdexcept>
using namespace std;

class Animal {
public:
    virtual ~Animal() {}
};

class Dog : public Animal {};

int main() {
    Animal a;
    try {
        Dog& d = dynamic_cast<Dog&>(a);  // ❌ This will throw an exception
    } catch (bad_cast& e) {
        cout << "❌ Exception: " << e.what() << endl;
    }
    return 0;
}
```

🛠️ Output:

```
❌ Exception: std::bad_cast
```

🔷 **Why?**

Unlike pointers, which return `nullptr` on failure, `dynamic_cast` **on references throws** `std::bad_cast` if the onversion is invalid.

4️⃣ Using `dynamic_cast` with Smart Pointers (Best Practice)

📝 Example 1: Downcasting (Safe)

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

class Animal {
public:
    virtual void sound() { cout << "Some animal sound\n"; }  // ✅ Must be vi
rtual
};

class Dog : public Animal {
public:
    void bark() { cout << "Woof! Woof!\n"; }
};

int main() {
    Animal* a = new Dog(); // Base class pointer to derived class
    Dog* d = dynamic_cast<Dog*>(a);

    if (d) {
        d->bark(); // ✅ Safe downcast
    } else {
        cout << "❌ Downcasting failed!" << endl;
    }

    delete a;
    return 0;
}
```

🛠️ Output:

```
Woof! Woof!
```

📝 Example 2: Detecting Type at Runtime

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

class Animal {
public:
    virtual void sound() { cout << "Animal sound\n"; }
};

class Dog : public Animal {};
class Cat : public Animal {};

void identifyAnimal(Animal* animal) {
    if (dynamic_cast<Dog*>(animal)) {
        cout << "This is a Dog!\n";
    } else if (dynamic_cast<Cat*>(animal)) {
        cout << "This is a Cat!\n";
    } else {
        cout << "Unknown Animal\n";
    }
}

int main() {
    Dog dog;
    Cat cat;
    Animal* a1 = &dog;
    Animal* a2 = &cat;

    identifyAnimal(a1);
    identifyAnimal(a2);

    return 0;
}
```

🛠️ Output:

```
This is a Dog!
This is a Cat!
```

📚 Student Task 1: Convert a Character to ASCII and Back

📝 Problem Statement:

Write a C++ program where the user enters a character. Convert it to its ASCII value using `static_cast`, then convert it back to the character.

📝 Expected Code

▼ Solution

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

int main() {
    char ch;
    cout << "Enter a character: ";
    cin >> ch;

    int asciiValue = static_cast<int>(ch); // Convert char to int
    cout << "ASCII Value: " << asciiValue << endl;

    char originalChar = static_cast<char>(asciiValue); // Convert back
    cout << "Converted Back to Character: " << originalChar << endl;

    return 0;
}
```

🛠️ Example Input/Output

```
Enter a character: A
ASCII Value: 65
Converted Back to Character: A
```

📚 Student Task 2: Safe Downcasting

📝 Problem Statement:

Create a base class `Vehicle` with derived classes `Car` and `Bike`. Use `dynamic_cast` to check if an object is a `Car` or `Bike` at runtime.

📝 Expected Code

▼ Solution

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

class Vehicle {
public:
    virtual void showType() { cout << "Vehicle\n"; }
};

class Car : public Vehicle {
public:
    void drive() { cout << "🚗 Driving a Car!\n"; }
};

class Bike : public Vehicle {
public:
    void ride() { cout << "🏍️ Riding a Bike!\n"; }
};

void identifyVehicle(Vehicle* v) {
    if (Car* c = dynamic_cast<Car*>(v)) {
        cout << "This is a Car!" << endl;
        c->drive();
```

```
    } else if (Bike* b = dynamic_cast<Bike*>(v)) {
        cout << "This is a Bike!" << endl;
        b->ride();
    } else {
        cout << "Unknown Vehicle!" << endl;
    }
}

int main() {
    Car car;
    Bike bike;
    Vehicle* v1 = &car;
    Vehicle* v2 = &bike;

    identifyVehicle(v1);
    identifyVehicle(v2);

    return 0;
}
```

🛠️ Example Input/Output

```
This is a Car!
🚗 Driving a Car!
This is a Bike!
🏍️ Riding a Bike!
```

📌 `const_cast`

The `const_cast` operator **removes the `const` qualifier** from a variable. It allows modifying data that was originally declared as `const`—but **it should be used cautiously and only in valid scenarios**.

🚨 Important:

- `const_cast` **cannot** remove `const` from truly `const` variables (e.g., `const int x = 10;` stored in read-only memory).

- It is **useful when dealing with functions that do not modify data but accept non-const parameters**.

🔷 1. Correct Use Case – Modifying `const` Inside a Function (Safe)

Sometimes, a function receives a `const` parameter but we **know** it is actually modifiable.

✅ **Example: Legacy Function Modifying a `const` Parameter**

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

void modifyValue(const int* p) {
    int* ptr = const_cast<int*>(p);  // Removing const
    *ptr = 42;  // Modifying value safely
}

int main() {
    int x = 10;  // ✅ Not a truly `const` variable
    cout << "Before: " << x << endl;

    modifyValue(&x);  // Passing address to function
    cout << "After: " << x << endl;  // ✅ Successfully modified

    return 0;
}
```

🛠️ Output

```
Before: 10
After: 42
```

✔️ **Why does this work?**

- `x` is **not actually** `const`, only passed as `const int*`.
- `const_cast` removes `const` and modifies `x` safely.

🔷 2. Use in Class Methods (Mutable Member Variables)

If a class method is marked `const`, it **cannot modify any member variables**, but sometimes we need exceptions.

✅ **Example: Allowing Modification of `mutable` Variable Inside a `const` Method**

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

class Student {
    mutable int accessCount;  // `mutable` allows modification in `const` met
hods
    string name;

public:
    Student(string n) : name(n), accessCount(0) {}

    void display() const {
        // Modify `mutable` variable inside a `const` method
        const_cast<int&>(accessCount)++;
        cout << "Student Name: " << name << ", Access Count: " << accessC
ount << endl;
    }
};

int main() {
    Student s("John Doe");
    s.display();
    s.display();

    return 0;
}
```

🛠️ Output

```
Student Name: John Doe, Access Count: 1
```

```
Student Name: John Doe, Access Count: 2
```

✔️ **Why does this work?**

- The `mutable` keyword allows modifying `accessCount`, even inside a `const` method.
- `const_cast` removes `const` to update `accessCount`.

🔷 3. Modifying a `const` Class Object

A `const` class object prevents modification of its members, but `const_cast` can override this.

✅ **Example: Modifying a `const` Object**

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

class Test {
    int value;
public:
    Test(int v) : value(v) {}
    void setValue(int v) { value = v; }
    void display() const { cout << "Value: " << value << endl; }
};

int main() {
    const Test obj(10);  // `const` object
    cout << "Before modification: ";
    obj.display();

    Test& modifiableObj = const_cast<Test&>(obj);  // Remove `const`
    modifiableObj.setValue(42);  // Modify the object

    cout << "After modification: ";
    obj.display();

    return 0;
}
```

🛠️ Output

```
Before modification: Value: 10
After modification: Value: 42
```

✔️ Why does this work?

- `obj` is originally `const`, but `const_cast` allows modification.
- Be cautious: modifying `const` objects can lead to **undefined behavior** if they are truly immutable.

🔷 4. `const_cast` with Function Overloading (C-style APIs)

Some **old C libraries** do not use `const`, and passing a `const` variable to them causes errors.

✅ Example: Calling a Non-const Function with `const` Data

```
#include <iostream>
#include <cstring> // C-style string functions
using namespace std;

void modifyString(char* str) {  // C-style function (expects non-const)
    strcpy(str, "Modified!");
}

int main() {
    const char original[] = "Hello";  // `const` string
    char* modifiableStr = const_cast<char*>(original);  // Remove `const`

    modifyString(modifiableStr);  // Pass to function
    cout << "Modified string: " << original << endl;  // 🚨 Undefined Behavio
r

    return 0;
}
```

❌ **This causes undefined behavior** because `original` is **truly** `const`.

✔️ **Solution:** Instead, use a writable copy:

```
char temp[] = "Hello";  // Non-const copy
modifyString(temp);  // ✅ Works fine
cout << "Modified string: " << temp << endl;
```

📌 `reinterpret_cast`

The `reinterpret_cast` operator **converts one pointer type into another unrelated pointer type**. It is the most powerful and dangerous type of casting in C++.

🚨 **Important:**

- `reinterpret_cast` should be used **only when absolutely necessary**.
- It is **platform-dependent** and can cause **undefined behavior** if used incorrectly.
- It does **not** perform any safety checks—**use it carefully!**

🔷 1. Converting a Pointer to an Integer and Vice Versa

Sometimes, we may need to store a pointer as an integer or retrieve a pointer from an integer.

✅ **Example: Storing a Pointer as an Integer**

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

int main() {
    int a = 42;
    int* p = &a;

    uintptr_t address = reinterpret_cast<uintptr_t>(p);  // Convert pointer to in
teger
    cout << "Pointer as Integer: " << address << endl;

    int* newPtr = reinterpret_cast<int*>(address);  // Convert back to pointer
    cout << "Value at new pointer: " << *newPtr << endl;  // Should print 42
```

```
    return 0;
}
```

🛠️ Output

```
Pointer as Integer: 140732928183432  (some memory address)
Value at new pointer: 42
```

✔️ Why does this work?

- `reinterpret_cast` allows **storing a pointer as an integer**.
- It can then be **converted back to a pointer safely**.
- `uintptr_t` is used to store addresses safely in an integer.

  `uintptr_t` is an **unsigned integer type** that can safely store **a pointer's address** without loss of information.

  It is defined in `<cstdint>` as:

  ```
  #include <cstdint>
  typedef unsigned int uintptr_t;
  ```

  📌 **Key Points:**

  - It is an **integer type** that is **large enough** to hold a **pointer**.
  - Useful when **storing** or **manipulating** pointers as integers.
  - It is an **implementation-defined** type (depends on the system).

  ## 🔷 Why Use `uintptr_t`?

  1. **Safely Convert a Pointer to an Integer**
     - Helps in **pointer arithmetic** or storing pointers in **non-pointer contexts**.
  2. **Portability**
     - `uintptr_t` ensures that the conversion works correctly on **any platform** (32-bit or 64-bit).

3. **Interfacing with Low-Level Code**
    - Useful in **memory manipulation**, **bitwise operations**, and **hardware programming**.

🔷 2. Casting Between Unrelated Pointer Types

If we need to convert between two **completely different pointer types**, `reinterpret_cast` is used.

✅ Example: Converting `int*` to `char*`

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

int main() {
    int num = 65;  // ASCII value of 'A'
    int* intPtr = &num;

    char* charPtr = reinterpret_cast<char*>(intPtr);  // Convert int* to char*
    cout << "Interpreted as char: " << *charPtr << endl;  // Prints 'A'

    return 0;
}
```

🛠️ Output

```
Interpreted as char: A
```

✔️ Why does this work?

- The `int` variable stores `65`, which is ASCII `'A'`.
- `reinterpret_cast<char*>` makes it appear as a `char` instead of an `int`.

🚨 **Caution:** If we modify `charPtr`, it might corrupt the memory of `num`!

🔷 3. Converting a Function Pointer

Sometimes, we need to convert a function pointer to a `void*` for generic storage or retrieval.

✅ Example: Function Pointer Conversion

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

void myFunction() {
    cout << "Hello from myFunction!" << endl;
}

int main() {
    void (*funcPtr)() = myFunction;
    void* genericPtr = reinterpret_cast<void*>(funcPtr);  // Store function po
inter as void*

    void (*recoveredFunc)() = reinterpret_cast<void (*)()>(genericPtr);  // Co
nvert back
    recoveredFunc();  // Call the function

    return 0;
}
```

🛠️ Output

```
Hello from myFunction!
```

✔️ Why does this work?

- We **store a function pointer as a** `void*` and later retrieve it.
- This is useful for **callback functions** in low-level programming.

🔷 4. Using `reinterpret_cast` in Class Inheritance

If we have a **base class** and **derived class**, we can use `reinterpret_cast` to **force conversion**.

✅ Example: Casting Between Unrelated Objects

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

```
class A {
public:
    void show() { cout << "Class A" << endl; }
};

class B {
public:
    void display() { cout << "Class B" << endl; }
};

int main() {
    A objA;
    B* objB = reinterpret_cast<B*>(&objA);  // Convert A* to B*

    objB->display();  // 🚨 Undefined Behavior!

    return 0;
}
```

🚨 **Warning!** This may **crash or produce garbage output** because `A` and `B` are **unrelated classes**.

🔷 1. Writing and Reading an `int` to a Binary File

Let's **store an integer** in a file and then read it back.

✅ Writing an Integer

```
#include <iostream>
#include <fstream>  // For file handling
using namespace std;

int main() {
    int num = 12345;  // Some number to store

    ofstream outFile("data.bin", ios::binary);  // Open file in binary mode
    outFile.write(reinterpret_cast<char*>(&num), sizeof(num));  // Convert int
* to char*
```

```
    outFile.close();

    cout << "Number written to file!" << endl;
    return 0;
}
```

📌 **Explanation:**

1️⃣ Open the file in **binary mode** (`ios::binary`).

2️⃣ Convert `int*` to `char*` using `reinterpret_cast`.

3️⃣ Write the data using `.write()`.

4️⃣ Close the file.

✅ Reading the Integer Back

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    int num;

    ifstream inFile("data.bin", ios::binary);  // Open the file in binary mode
    inFile.read(reinterpret_cast<char*>(&num), sizeof(num));  // Read and convert back
    inFile.close();

    cout << "Number read from file: " << num << endl;  // Should print 12345
    return 0;
}
```

📌 **How it Works?**

1️⃣ Open file in **binary mode**.

2️⃣ Use `.read()` to load data and convert `char*` back to `int*`.

3️⃣ Print the retrieved number.

✔️ **Output**

```
Number read from file: 12345
```

🔷 2. Writing and Reading a `struct` (Multiple Values)

Let's **store multiple values** in a struct inside a binary file.

✅ Writing a Struct

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student {
    int id;
    char name[20];
    float marks;
};

int main() {
    Student s = {101, "Anand", 89.5};  // Create a student object

    ofstream outFile("student.bin", ios::binary);
    outFile.write(reinterpret_cast<char*>(&s), sizeof(s));  // Convert struct* t
o char*
    outFile.close();

    cout << "Student data written!" << endl;
    return 0;
}
```

📌 **Why `reinterpret_cast`?**

- The `write()` function needs `char*`, but we have a `Student*`.
- `reinterpret_cast` converts `Student* → char*` for proper storage.

✅ Reading the Struct Back

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student {
    int id;
    char name[20];
    float marks;
};

int main() {
    Student s;

    ifstream inFile("student.bin", ios::binary);
    inFile.read(reinterpret_cast<char*>(&s), sizeof(s));  // Convert back
    inFile.close();

    cout << "ID: " << s.id << endl;
    cout << "Name: " << s.name << endl;
    cout << "Marks: " << s.marks << endl;

    return 0;
}
```

✔️ **Output**

```
ID: 101
Name: Anand
Marks: 89.5
```

🔷 3. Writing and Reading an Array of Structures

Let's **store multiple students** in a binary file.

✅ Writing an Array of Students

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student {
    int id;
    char name[20];
    float marks;
};

int main() {
    Student students[3] = {
        {101, "Anand", 89.5},
        {102, "Rahul", 78.2},
        {103, "Priya", 92.1}
    };

    ofstream outFile("students.bin", ios::binary);
    outFile.write(reinterpret_cast<char*>(&students), sizeof(students));
    outFile.close();

    cout << "All student data written!" << endl;
    return 0;
}
```

📌 **Why is this efficient?**

- The entire array is written in **one step**, making it **faster**.

✅ Reading the Array Back

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student {
    int id;
    char name[20];
```

```
    float marks;
};

int main() {
    Student students[3];

    ifstream inFile("students.bin", ios::binary);
    inFile.read(reinterpret_cast<char*>(&students), sizeof(students));
    inFile.close();

    cout << "Students read from file:\n";
    for (int i = 0; i < 3; i++) {
        cout << "ID: " << students[i].id << ", Name: " << students[i].name
             << ", Marks: " << students[i].marks << endl;
    }

    return 0;
}
```

✔️ **Output**

```
Students read from file:
ID: 101, Name: Anand, Marks: 89.5
ID: 102, Name: Rahul, Marks: 78.2
ID: 103, Name: Priya, Marks: 92.1
```

🔷 4. Writing and Reading a Single Class Object

We'll create a **Student class**, write an object to a binary file, and then read it back.

✅ Writing a Student Object

```
#include <iostream>
#include <fstream>
using namespace std;

class Student {
```

```
public:
    int id;
    char name[20];
    float marks;

    // Constructor to initialize values
    Student(int i = 0, const char* n = "", float m = 0.0) {
        id = i;
        strcpy(name, n);
        marks = m;
    }

    // Function to display student details
    void display() {
        cout << "ID: " << id << ", Name: " << name << ", Marks: " << marks <
< endl;
    }
};

int main() {
    Student s(101, "Anand", 88.5);  // Creating an object

    ofstream outFile("student_class.bin", ios::binary);
    outFile.write(reinterpret_cast<char*>(&s), sizeof(s));  // Convert object* t
o char*
    outFile.close();

    cout << "Student object written to file!" << endl;
    return 0;
}
```

📌 **Explanation:**

- We define a `Student` **class** with `id`, `name`, and `marks`.
- The constructor initializes values.
- We create an object and **store it in a binary file** using `reinterpret_cast`.

✅ Reading the Student Object

```
#include <iostream>
#include <fstream>
using namespace std;

class Student {
public:
    int id;
    char name[20];
    float marks;

    void display() {
        cout << "ID: " << id << ", Name: " << name << ", Marks: " << marks <
< endl;
    }
};

int main() {
    Student s;

    ifstream inFile("student_class.bin", ios::binary);
    inFile.read(reinterpret_cast<char*>(&s), sizeof(s));  // Convert back
    inFile.close();

    cout << "Student object read from file:" << endl;
    s.display();  // Display the read values

    return 0;
}
```

✔ **Output**

```
Student object read from file:
ID: 101, Name: Anand, Marks: 88.5
```

🔷 5. Writing and Reading an Array of Student Objects

Now, let’s **store multiple student objects** in a binary file.

✅ Writing an Array of Student Objects

```
#include <iostream>
#include <fstream>
using namespace std;

class Student {
public:
    int id;
    char name[20];
    float marks;

    Student(int i = 0, const char* n = "", float m = 0.0) {
        id = i;
        strcpy(name, n);
        marks = m;
    }

    void display() {
        cout << "ID: " << id << ", Name: " << name << ", Marks: " << marks << endl;
    }
};

int main() {
    Student students[3] = {
        Student(101, "Anand", 88.5),
        Student(102, "Rahul", 76.2),
        Student(103, "Priya", 92.1)
    };

    ofstream outFile("students_class.bin", ios::binary);
    outFile.write(reinterpret_cast<char*>(&students), sizeof(students));
    outFile.close();

    cout << "Student objects written to file!" << endl;
```

```
    return 0;
}
```

📌 **Why `reinterpret_cast`?**

- The `.write()` function requires `char*`, but we have a `Student*`.
- `reinterpret_cast` allows us to **store the raw memory of the objects**.

✅ Reading the Array of Student Objects

```
#include <iostream>
#include <fstream>
using namespace std;

class Student {
public:
    int id;
    char name[20];
    float marks;

    void display() {
        cout << "ID: " << id << ", Name: " << name << ", Marks: " << marks <
< endl;
    }
};

int main() {
    Student students[3];

    ifstream inFile("students_class.bin", ios::binary);
    inFile.read(reinterpret_cast<char*>(&students), sizeof(students));
    inFile.close();

    cout << "Students read from file:\n";
    for (int i = 0; i < 3; i++) {
        students[i].display();
    }
```

```
    return 0;
}
```

✔ **Output**

```
Students read from file:
ID: 101, Name: Anand, Marks: 88.5
ID: 102, Name: Rahul, Marks: 76.2
ID: 103, Name: Priya, Marks: 92.1
```

Type Conversion in OOPS (Class Type Conversion)

Type conversion in **object-oriented programming (OOPs)** involves **class objects** and is categorized as:

1. **Basic Type to Class Type**
2. **Class Type to Basic Type**
3. **Class Type to Another Class Type**

---

1. Basic Type to Class Type (Using Constructor)

Converts a **basic data type** (like `int`, `float`) to a **class type**.

This is done using a **constructor** that takes the basic type as an argument.

Example

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

class Number {
    int value;
public:
    Number(int x) {  // Constructor for conversion
        value = x;
    }
    void display() {
        cout << "Value: " << value << endl;
```

```
    }
};

int main() {
    int num = 100;
    Number obj = num;  // Implicit conversion (int → Number)
    obj.display();
    return 0;
}
```

Output:

```
Value: 100
```

✔ The `Number` constructor is called when `num` is assigned to `obj`.

---

2. Class Type to Basic Type (Using Conversion Function)

Converts a **class object** to a **basic type** using a **conversion operator function**.

What is a Conversion Function?

A **conversion function** in C++ is a special **member function** used to convert an object of a class to another data type (either a basic data type or another class type).

It is defined inside a class using the **operator keyword**, followed by the type to which the object should be converted.

Syntax of a Conversion Function

```
operator typeName() {
    // Conversion logic
    return value;
}
```

- **No return type** is specified (not even `void`).
- It does not take any parameters.

- It is called **implicitly** when conversion is needed.
- A class can have n number of type conversion function.

Example: Class Type to Basic Type

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

class Number {
    int value;
public:
    Number(int x) { value = x; }  // Constructor
    operator int() { return value; }  // Conversion function
};

int main() {
    Number obj = 50;
    int num = obj;  // Implicit conversion (Number → int)
    cout << "Converted value: " << num << endl;
    return 0;
}
```

Output:

```
Converted value: 50
```

✔ The **operator function** `operator int()` allows `Number` to be used as an `int`.

---

3. Class Type to Another Class Type

This occurs when an object of one class is converted to an object of another class.

Method 1: Using a Conversion Constructor

The **destination class** has a constructor that takes an object of the **source class**.

Example

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

class Rectangle {
    int width, height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}
    int getWidth() { return width; }
    int getHeight() { return height; }
};

class Square {
    int side;
public:
    Square(Rectangle r) {  // Conversion constructor
        side = min(r.getWidth(), r.getHeight());
    }
    void display() { cout << "Side of Square: " << side << endl; }
};

int main() {
    Rectangle rect(8, 5);
    Square sq = rect;  // Implicit conversion (Rectangle → Square)
    sq.display();
    return 0;
}
```

Output:

```
Side of Square: 5
```

✔️ The `Square` constructor takes a `Rectangle` object and extracts the smallest dimension.

Method 2: Using Overloaded Type Conversion Operator

The **source class** defines a **conversion operator function** that returns an object of the **destination class**.

Example

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

class Square {
    int side;
public:
    Square(int s) { side = s; }
    int getSide() { return side; }
};

class Rectangle {
    int width, height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}

    operator Square() { // Conversion function
        return Square(min(width, height));
    }
};

int main() {
    Rectangle rect(10, 6);
    Square sq = rect;  // Implicit conversion (Rectangle → Square)
    cout << "Side of Square: " << sq.getSide() << endl;
    return 0;
}
```

Output:

```
Side of Square: 6
```

✔ The **operator function** `operator Square()` performs the conversion.

Summary

Type Conversion	Method Used

Implicit Type Conversion	Done by the compiler automatically
Explicit Type Conversion	Uses **type casting** (`(type)value` , `static_cast<>`)
Basic Type → Class Type	Uses a **constructor** in the class
Class Type → Basic Type	Uses a **conversion function** (`operator type()`)
Class Type → Another Class Type	- **Conversion constructor** in the destination class - **Overloaded type conversion operator** in the source class

---

Student Task:

This task will help students understand **object type conversion** in C++ through **three types of conversions**:

1. **Basic to Class Type Conversion**
2. **Class to Basic Type Conversion**
3. **Class to Class Type Conversion**

🎯 Task Overview

🔷 You need to create a **C++ program** that demonstrates all three types of conversions.

🔷 Implement a **Student** class that stores **marks** and convert it into different types.

🔷 Use **constructor overloading, type conversion functions, and operator overloading.**

📌 Task Breakdown

1️⃣ Basic to Class Type Conversion

👉 Convert an `int` (marks) into a `Student` object.

Requirements

- Use a **parameterized constructor** to accept an integer.

- Convert an integer to a `Student` object.

Example

```
Student s1 = 85;  // Convert int to Student object
s1.display();  // Should print: "Marks: 85"
```

2️⃣ Class to Basic Type Conversion

👉 Convert a `Student` object into an `int` (marks).

Requirements

- Use a **type conversion function** to return marks.

Example

```
Student s2(90);
int totalMarks = s2;  // Convert Student object to int
cout << "Total Marks: " << totalMarks;  // Should print: "Total Marks: 90"
```

3️⃣ Class to Class Type Conversion

👉 Convert a `Student` object into a `Grade` object.

Requirements

- Implement a `Grade` class that stores grades (`A`, `B`, `C` etc.).
- Define a **conversion operator** inside `Student` to convert it into `Grade`.

Example

```
Student s3(78);
Grade g = s3;  // Convert Student object to Grade object
g.display();  // Should print: "Grade: B"
```

🚀 Task: Write a Complete Program

Write a C++ program implementing all three conversions.

💡 Hints

1. Use **constructors** for basic-to-class conversion.
2. Use **overloaded type conversion functions** for class-to-basic conversion.
3. Use **conversion operators** for class-to-class conversion.

🎯 Expected Output

```
Marks: 85
Total Marks: 90
Grade: B
```

▼ **Solution**

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

// Forward declaration of class Grade
class Grade;

// Student class
class Student {
private:
    int marks;

public:
    // 1️⃣ Basic to Class Type Conversion: Constructor accepting int
    Student(int m) {
        marks = m;
    }

    // Function to display marks
    void display() {
        cout << "Marks: " << marks << endl;
    }

    // 2️⃣ Class to Basic Type Conversion: Overloading type conversion to ir
```

```
    operator int() {
        return marks;
    }

    // 3️⃣ Class to Class Type Conversion: Convert Student to Grade
    operator Grade();
};

// Grade class for storing grades
class Grade {
private:
    char grade;

public:
    // Constructor
    Grade(char g) {
        grade = g;
    }

    // Function to display grade
    void display() {
        cout << "Grade: " << grade << endl;
    }
};

// Defining conversion function from Student to Grade
Student::operator Grade() {
    char g;
    if (marks >= 90)
        g = 'A';
    else if (marks >= 80)
        g = 'B';
    else if (marks >= 70)
        g = 'C';
    else if (marks >= 60)
        g = 'D';
    else
        g = 'F';
```

```
    return Grade(g);
}

// Main function
int main() {
    // 1️⃣ Basic to Class Conversion
    Student s1 = 85;  // Convert int to Student
    s1.display();      // Output: Marks: 85

    // 2️⃣ Class to Basic Type Conversion
    Student s2(90);
    int totalMarks = s2;  // Convert Student to int
    cout << "Total Marks: " << totalMarks << endl;  // Output: Total Marks: 9

    // 3️⃣ Class to Class Conversion
    Student s3(78);
    Grade g = s3;  // Convert Student to Grade
    g.display();   // Output: Grade: C

    return 0;
}
```