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)

Example: Explicit Conversion

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

int main() {
    double num = 10.75;

    int intNum1 = (int)num;  // C-style cast
    int intNum2 = static_cast<int>(num);  // static_cast

    cout << "Using C-style cast: " << intNum1 << endl;
    cout << "Using static_cast: " << intNum2 << endl;

    return 0;
}
```

Output:

```
Using C-style cast: 10
Using static_cast: 10
```

✔️ The fractional part is **truncated**.

`dynamic_cast<type>(value)` - discuss later

`reinterpret_cast` - Used for Low-Level Memory Conversion

Think of `reinterpret_cast` as **"forcefully treating"** one type of data as another, even if they are **completely unrelated**.

🔷 **Why use it?**

- Used for **low-level memory operations** (e.g., converting pointers).
- Used when we **reinterpret** memory, even if it doesn't make logical sense in high-level code.
- Mainly used in **system programming, hardware interfacing, and bitwise operations**.

📌 **Example: Converting a pointer to an integer (and back)**

```
#include <iostream>

using namespace std;

int main() {
    int x = 42;
    int* ptr = &x;

    // Convert pointer to an integer (memory address)
    uintptr_t address = reinterpret_cast<uintptr_t>(ptr);
    cout << "Pointer as integer: " << address << endl;

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

    return 0;
}
```

✔️ **Use Case:** Storing a pointer as an integer and converting it back.

The
`uintptr_t` type is an **unsigned integer type** that is guaranteed to be able to store a **pointer** without loss of data. It is defined in the `<cstdint>` header.

Key Reasons to Use `uintptr_t`

1. **Storing Memory Addresses as Integers**
    - When you need to store a pointer value as an integer (for debugging, serialization, or hardware-related tasks).
2. **Performing Arithmetic on Pointers**
    - Useful when you need to perform pointer arithmetic in a portable way.
3. **Interfacing with Low-Level Code (Hardware & Embedded Systems)**
    - Often used in **system programming**, device drivers, and embedded systems where pointers need to be cast into integers.
4. **Portability & Safety**
    - Unlike `unsigned int` or `size_t`, `uintptr_t` is **guaranteed** to be the correct size to hold a pointer, making code more portable across different platforms (32-bit vs. 64-bit systems).

📌 **Example: Treating a float's memory as an integer (bitwise operation)**

```
#include <iostream>

using namespace std;

int main() {
    float pi = 3.14;

    // Treat the memory of 'pi' as an integer
    int intValue = reinterpret_cast<int&>(pi);

    cout << "Float value: " << pi << endl;
    cout << "Interpreted as integer: " << intValue << endl;

    return 0;
}
```

✔️ **Use Case:** Useful in graphics programming, bit manipulation, and memory hacks.

`const_cast` - Removing `const` to Modify a Read-Only Variable

Think of `const_cast` as **"removing the safety lock"** from a `const` variable so you can modify it.

🔷 Why use it?

- When working with **APIs** that take `const` pointers but we **need to modify** the data.
- When a function mistakenly declares a parameter as `const` but we know it's safe to change.

📌 Example: Removing `const` to Modify a Variable

```
#include <iostream>

using namespace std;

void modify(const int* ptr) {
    int* modifiablePtr = const_cast<int*>(ptr);
    *modifiablePtr = 100;  // Changing value
}

int main() {
    int x = 42;
    modify(&x);
    cout << "Modified value: " << x << endl;
    return 0;
}
```

✔️ **Use Case:** Used in functions where a `const` variable needs modification.

```
#include <iostream>

using namespace std;

class Demo {
public:
    mutable int data = 10;  // `mutable` allows modification even in `const` fu
nction
```

```
    void show() const {
        cout << "Before: " << data << endl;
        const_cast<Demo*>(this)->data = 20;  // Removing const
        cout << "After: " << data << endl;
    }
};

int main() {
    Demo obj;
    obj.show();
    return 0;
}
```

✔ **Use Case:** Modifying data inside a `const` function.

🚀 Quick Comparison

Type	Purpose	Example Use Case
`reinterpret_cast`	Treats one type as another (even if unrelated)	Converting a pointer to an integer or vice versa
`const_cast`	Removes `const` from a variable	Modifying a `const` object in special cases

---

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 in
    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;
}
```