🧮 1. Arithmetic Operators

Purpose: Perform basic mathematical operations.

Operator	Description	Example	Analogy
`+`	Addition	`a + b`	Combining two quantities
`-`	Subtraction	`a - b`	Removing one quantity from another
`*`	Multiplication	`a * b`	Scaling a quantity
`/`	Division	`a / b`	Splitting into equal parts
`%`	Modulus	`a % b`	Finding the remainder
`++`	Increment	`++a` or `a++`	Increasing by one
`--`	Decrement	`--a` or `a--`	Decreasing by one

Example:

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

int main() {
    int a = 10, b = 3;
    cout << "Addition: " << a + b << endl;
    cout << "Subtraction: " << a - b << endl;
    cout << "Multiplication: " << a * b << endl;
    cout << "Division: " << a / b << endl;
    cout << "Modulus: " << a % b << endl;
    return 0;
}
```

Output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3
Modulus: 1
```

🧾 2. Assignment Operators

Purpose: Assign values to variables.

Operator	Description	Example	Equivalent

Operator	Description	Example	Equivalent
`=`	Assign	`a = b`	Assign value of b to a
`+=`	Add and assign	`a += b`	`a = a + b`
`-=`	Subtract and assign	`a -= b`	`a = a - b`
`*=`	Multiply and assign	`a *= b`	`a = a * b`
`/=`	Divide and assign	`a /= b`	`a = a / b`
`%=`	Modulus and assign	`a %= b`	`a = a % b`

Example:

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

int main() {
    int a = 5;
    a += 3; // a = a + 3
    cout << "a after += 3: " << a << endl;
    return 0;
}
```

Output:

```
a after += 3: 8
```

🔗 3. Relational Operators

Purpose: Compare two values.

Operator	Description	Example	Result
`==`	Equal to	`a == b`	true if a equals b
`!=`	Not equal to	`a != b`	true if a is not equal to b
`>`	Greater than	`a > b`	true if a is greater than b
`<`	Less than	`a < b`	true if a is less than b
`>=`	Greater than or equal to	`a >= b`	true if a is greater than or equal to b
`<=`	Less than or equal to	`a <= b`	true if a is less than or equal to b

Example:

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

int main() {
    int a = 5, b = 10;
    cout << "Is a equal to b? " << (a == b) << endl;
    return 0;
}
```

Output:

```
Is a equal to b? 0
```

(Note: In C++, `0` represents `false`, and `1` represents `true`.)

⚙️ 4. Logical Operators

Purpose: Combine multiple conditions.

Operator	Description	Example	Result			
`&&`	Logical AND	`a && b`	true if both a and b are true			
`		`	Logical OR	`a	b`	true if either a or b is true
`!`	Logical NOT	`!a`	true if a is false			

Example:

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

int main() {
    bool a = true, b = false;
    cout << "a AND b: " << (a && b) << endl;
    cout << "a OR b: " << (a || b) << endl;
    cout << "NOT a: " << (!a) << endl;
    return 0;
}
```

Output:

```
```

```
a AND b: 0
a OR b: 1
NOT a: 0
```

🧠 5. Bitwise Operators

Purpose: Perform operations at the bit level.

Operator	Description	Example	Result		
`&`	Bitwise AND	`a & b`	Bits set in both a and b		
`	`	Bitwise OR	`a	b`	Bits set in either a or b
`^`	Bitwise XOR	`a ^ b`	Bits set in a or b but not both		
`~`	Bitwise NOT	`~a`	Inverts bits of a		
`<<`	Left shift	`a << 1`	Shifts bits of a left by 1		
`>>`	Right shift	`a >> 1`	Shifts bits of a right by 1		

Example:

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

int main() {
    int a = 5; // Binary: 0101
    int b = 9; // Binary: 1001
    cout << "a & b: " << (a & b) << endl;
    cout << "a | b: " << (a | b) << endl;
    cout << "a ^ b: " << (a ^ b) << endl;
    cout << "~a: " << (~a) << endl;
    cout << "b << 1: " << (b << 1) << endl;
    cout << "b >> 1: " << (b >> 1) << endl;
    return 0;
}
```

Output:

```
a & b: 1
a | b: 13
a ^ b: 12
~a: -6
b << 1: 18
b >> 1: 4
```

🎯 6. Conditional (Ternary) Operator

Purpose: Short-hand for `if-else`.

Syntax: `condition ? expression_if_true : expression_if_false;`

Example:

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

int main() {
    int a = 10, b = 20;
    int max = (a > b) ? a : b;
    cout << "The maximum is: " << max << endl;
    return 0;
}
```

Output:

```
The maximum is: 20
```

📌 7. Other Operators

a. Sizeof Operator

Purpose: Returns the size of a data type.

Example:

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

int main() {
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
    return 0;
}
```

Output:

```
Size of int: 4 bytes
```

(Note: Size may vary based on the system.)

b. Comma Operator

Purpose: Allows multiple expressions where only one is expected.

Example:

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

int main() {
    int a = (1, 2, 3);
    cout << "Value of a: " << a << endl;
    return 0;
}
```

Output:

```
Value of a: 3
```

(Only the last value is assigned.)

c. Pointer Operators

- `*` : Dereference operator
- `&` : Address-of operator

Example:

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

int main() {
    int a = 10;
    int *ptr = &a;
    cout << "Address of a: " << &a << endl;
    cout << "Value at ptr: " << *ptr << endl;
    return 0;
}
```

Output:

```
Address of a: 0x7ffee4bff5ac
Value at ptr: 10
```

🧠 Operator Precedence and Associativity

Operators have a defined precedence which determines the order of evaluation in expressions. For instance, multiplication and division have higher precedence than addition and subtraction.

Example:

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

int main() {
    int result = 10 + 20 * 3;
    cout << "Result: " << result << endl;
    return 0;
}
```

Output:

```
Result: 70
```

(Multiplication is performed before addition.)

For a detailed table of operator precedence and associativity, you can refer to resources like GeeksforGeeks or W3Schools.

📚 Summary

- **Arithmetic Operators:** Perform basic math operations.
- **Assignment Operators:** Assign values to variables.
- **Relational Operators:** Compare values.
- **Logical Operators:** Combine multiple conditions.
- **Bitwise Operators:** Perform operations at the bit level.
- **Conditional Operator:** Short-hand for `if-else`.
- **Other Operators:** Include `sizeof`, comma, and pointer operators.

🧠Bitwise Operators in C++

🧩 Purpose:

They **manipulate individual bits** within integers — often used for performance optimization, system programming, cryptography, etc.

🔧 Operator Summary:

Operator	Description	Example	Meaning	
`&`	Bitwise AND	`a & b`	Only 1 where **both** bits are 1	
`|`	Bitwise OR	`a	b`		1 where **either** bit is 1
`^`	Bitwise XOR	`a ^ b`	1 where bits are **different**	
`~`	Bitwise NOT	`~a`	Inverts each bit (1 → 0, 0 → 1)	
`<<`	Left Shift	`a << 1`	Shifts bits **left**, adds 0 on right	
`>>`	Right Shift	`a >> 1`	Shifts bits **right**, discards rightmost bit	

🔍 Example with Binary Representation:

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

int main() {
    int a = 5;  // Binary:  0101
    int b = 9;  // Binary:  1001

    cout << "a & b: " << (a & b) << endl;   // AND
    cout << "a | b: " << (a | b) << endl;   // OR
    cout << "a ^ b: " << (a ^ b) << endl;   // XOR
    cout << "~a: " << (~a) << endl;         // NOT
    cout << "b << 1: " << (b << 1) << endl; // Left Shift
    cout << "b >> 1: " << (b >> 1) << endl; // Right Shift

    return 0;
}
```

🧮 Step-by-step Binary Explanation:

Variables:

```
a = 5      → 0000 0101
b = 9      → 0000 1001
```

✅ Outputs with Binary Logic:

Expression	Binary Operation	Decimal Result	Explanation
`a & b`	0000 0101 & 0000 1001 = 0000 0001	`1`	Only bit 0 is set in both

Expression	Binary Operation	Decimal Result	Explanation	
`a	b`	0000 0101 | 0000 1001 = 0000 1101	`13`	Combines set bits from both
`a ^ b`	0000 0101 ^ 0000 1001 = 0000 1100	`12`	Only bits that are different	
`~a`	~0000 0101 = 1111 1010	`-6`	2’s complement of 6 (inverts bits)	
`b << 1`	0000 1001 << 1 = 0001 0010	`18`	Left shift: multiply by 2 (adds 0 on right)	
`b >> 1`	0000 1001 >> 1 = 0000 0100	`4`	Right shift: divides by 2 (drops least significant bit)	

💡 Important Notes:

- **Negative Results:** Bitwise NOT (`~`) returns negative because of how signed integers are stored (2's complement).
- **Left/Right Shifting**:
  - `x << n` → `x * 2^n`
  - `x >> n` → `x / 2^n` (floor division)

🛠 Use Cases:

- **Flags and Masks**
- **Efficient Multiplication/Division**
- **Compression and Encryption**
- **Low-level device control / embedded systems**

🔚 Summary Table

Operation	Decimal Result	Binary View	
`a & b`	1	`0000 0001`	
`a	b`	13	`0000 1101`
`a ^ b`	12	`0000 1100`	
`~a`	-6	`1111 1010`	
`b << 1`	18	`0001 0010`	
`b >> 1`	4	`0000 0100`	

🔢 How to Extract Digits from a Number in C++

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

int main() {
    int a = 123;

    // Extract last digit
    int lastDigit = a % 10; // 3
    cout << "Last digit: " << lastDigit << endl;

    // Remove last digit
    int remaining = a / 10; // 12
    cout << "Remaining number: " << remaining << endl;

    return 0;
}
```

💡 Explanation with Analogies:

Operation	Code	Meaning	Result
🍪 Get Last Digit	`a % 10`	Modulus gives remainder — last digit	`3`
✂️ Trim Last Digit	`a / 10`	Integer division removes the last digit	`12`

🔄 Full Digit Extraction (Loop)

To extract **all digits one by one from last to first**, use a loop:

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

int main() {
    int a = 123;

    cout << "Digits of " << a << " (reversed): ";
    while (a > 0) {
        int digit = a % 10;      // 🍪 Get last digit
        cout << digit << " ";
        a = a / 10;              // ✂️ Remove last digit
    }

    return 0;
}
```

🖨️ Output:

```
Digits of 123 (reversed): 3 2 1
```

🔁 Bonus: Digits in Original Order

If you want to print digits **in the correct order**, you can store them in an array or vector and then print them in reverse, or reverse the number first. Perfect! Let's now extract the **digits in the original order** (from left to right) instead of reversed.

✅ Extract Digits in Original Order (Left to Right)

🔄 Method 1: Using a `vector` to store and then print

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

int main() {
    int a = 123;
    vector<int> digits;

    // 🍪 Step 1: Extract digits in reverse
    while (a > 0) {
        digits.push_back(a % 10); // Get last digit
        a = a / 10;               // Remove last digit
    }

    // 🔁 Step 2: Reverse the vector to get original order
    cout << "Digits in correct order: ";
    for (int i = digits.size() - 1; i >= 0; i--) {
        cout << digits[i] << " ";
    }

    return 0;
}
```

🧠 Output:

```
Digits in correct order: 1 2 3
```

🎲 Analogy:

Imagine you're peeling off the last layer of an onion (the last digit) and storing it in a box. Once you're done peeling all layers, you open the box from the bottom to top to see the original structure! 🧅 📦

⚡ Bonus: Extract Digits Using a String (Quick Hack)

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

int main() {
    int a = 123;
    string s = to_string(a);

    cout << "Digits in original order: ";
    for (char ch : s) {
        cout << ch << " ";
    }

    return 0;
}
```

🧠 Output:

```
Digits in original order: 1 2 3
```

💡 Useful when you don't need to do calculations with the digits — just display them.

🔁 Reverse a Number in C++

🧠 Idea:

Take the last digit → append it to the result → remove it from the original → repeat until 0.

✅ Code Example:

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

int main() {
    int a = 123;
    int reversed = 0;

    while (a > 0) {
        int digit = a % 10;            // 🔍 Get last digit
        reversed = reversed * 10 + digit; // ⬅ Shift digits left & add new digit
        a = a / 10;                    // 🔄 Remove last digit
    }
```

```
    cout << "Reversed number: " << reversed << endl;
    return 0;
}
```

📤 Output:

```
Reversed number: 321
```

🎲 Analogy:

Imagine you're stacking blocks numbered 1, 2, and 3 vertically 🧱🧱🧱. Reversing means lifting the last one first and stacking it into a new tower from the base. Now it becomes 3, 2, 1 🏗.

💡 Bonus: Handle Reversed Number as a String

If you're okay treating numbers as text (e.g., for display), here's a shortcut using string:

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

int main() {
    int a = 123;
    string s = to_string(a);
    reverse(s.begin(), s.end());

    cout << "Reversed string version: " << s << endl;
    return 0;
}
```

🧠 Output:

```
Reversed string version: 321
```

🔁 Reverse an Integer Safely in C++

🧠 Objective:

- Reverse digits of an integer
- Support **negative numbers**

- Avoid **integer overflow**

✅ Final Code with Comments:

```
#include <iostream>
#include <climits> // For INT_MAX and INT_MIN
using namespace std;

int reverseInteger(int x) {
    int reversed = 0;

    while (x != 0) {
        int digit = x % 10;  // 🔍 Extract last digit

        // 🛑 Check overflow/underflow before adding digit
        if (reversed > INT_MAX / 10 || (reversed == INT_MAX / 10 && digit > 7))
            return 0; // Overflow
        if (reversed < INT_MIN / 10 || (reversed == INT_MIN / 10 && digit < -8))
            return 0; // Underflow

        reversed = reversed * 10 + digit; // ⬅️ Shift and add digit
        x = x / 10;                       // 🔄 Remove last digit
    }

    return reversed;
}

int main() {
    int number;
    cout << "Enter a number to reverse: ";
    cin >> number;

    int result = reverseInteger(number);

    if (result == 0 && number != 0) {
        cout << "❗ Overflow/Underflow occurred!" << endl;
    } else {
        cout << "✅ Reversed number: " << result << endl;
    }

    return 0;
}
```

📤 Example Outputs:

✅ Case 1: Positive input

```
Input: 123
Output: ✅ Reversed number: 321
```

✅ Case 2: Negative input

```
Input: -456
Output: ✅ Reversed number: -654
```

🚫 Case 3: Overflow case (e.g., 1534236469)

```
Input: 1534236469
Output: ❗ Overflow/Underflow occurred!
```

📚 Why Check for Overflow?

In C++, the range for `int` is:

- 🔼 `INT_MAX = 2147483647`
- 🔽 `INT_MIN = -2147483648`

To be safe before:

```
reversed = reversed * 10 + digit;
```

We **predict** whether multiplying by 10 and adding will overflow.

🎲 Analogy:

Imagine a jar 📦 can hold up to `2147483647` candies. Every digit you add is like tossing in more. Before you toss, make sure the jar won't **overflow** 🍬🧪.

⚙️ Division Operator (`/`) in Different Data Types

📌 General Syntax:

```
result = a / b;
```

The result **depends on the types** of `a` and `b`.

🧪 Type-wise Behavior

Type of `a`	Type of `b`	Result Type	Behavior
`int`	`int`	`int`	Integer division (truncates decimal)
`float`	`float`	`float`	Floating point division
`int`	`float`	`float`	Integer promoted to float
`double`	`int`	`double`	Integer promoted to double
`char`	`int`	`int`	`char` promoted to ASCII int

🔢 Examples with Outputs

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

int main() {
    int a = 10, b = 3;
    float x = 10.0f, y = 3.0f;
    double d = 10.0;
    char ch = 'A'; // ASCII 65

    cout << "◇ int / int = " << a / b << endl;          // 3
    cout << "◇ float / float = " << x / y << endl;      // 3.33333
    cout << "◇ int / float = " << a / y << endl;        // 3.33333
    cout << "◇ double / int = " << d / b << endl;       // 3.33333
    cout << "◇ char / int = " << ch / b << endl;        // 65 / 3 = 21
    return 0;
}
```

🧾 Output:

```
◇ int / int = 3
◇ float / float = 3.33333
◇ int / float = 3.33333
◇ double / int = 3.33333
◇ char / int = 21
```

⚠️ Important Notes

❗ 1. **Integer Division Truncates**

```
int a = 7, b = 2;
cout << a / b;  // Output: 3 (NOT 3.5)
```

💡 Analogy: Like dividing 7 apples 🍏 among 2 kids. Each kid gets **3 full apples**, and the half is discarded.

⚠ 2. **To Get Decimal Results: Use float/double**

```
float result = 7 / 2.0; // or cast one operand: (float)7 / 2;
```

☑ Output: `3.5`

⚠ 3. **Division by Zero**

- Integer division by zero: ❌ **runtime error**
- Floating-point division by zero: ☑ **Infinity or NaN**

```
int a = 5, b = 0;
float x = 5.0f, y = 0.0f;

// cout << a / b; // ❌ CRASH

cout << x / y; // ☑ Output: inf
```

🧠 Summary Table

Expression	Type	Result	Notes
`7 / 2`	`int / int`	`3`	Truncates
`7.0 / 2`	`float / int`	`3.5`	Promotes int to float
`7 / 2.0`	`int / double`	`3.5`	Promotes int to double
`'A' / 2`	`char / int`	`32`	'A' = 65
`7 / 0`	`int / int`	❌ Crash	Runtime error
`7.0 / 0.0`	`float / float`	`inf`	IEEE 754

📚 C++ MCQs – Operator Precedence, Associativity, Output & Common Mistakes

🧠 Multiple Choice Questions

---

1. Which operator has the highest precedence in C++?

A) Addition (`+`)
B) Multiplication (`*`)
C) Assignment (`=`)
D) Increment (`++`)

---

2. What is the associativity of the assignment operator `=` in C++?

A) Left-to-right
B) Right-to-left
C) None
D) Operator dependent

---

3. What would be the output of the following code snippet?

```cpp
#include <iostream>
int main() {
    int a = 5;
    int b = a++ + ++a;
    std::cout << b;
    return 0;
}
```

A) 10 B) 11 C) 12 D) 13

---

4. What mistake is commonly made when using the `while` loop in C++?

A) Using `=` instead of `==` for comparison B) Placing the increment/decrement statement outside the loop C) Using brackets `{}` for single statements D) All of these

---

✅ Answers

1. D) Increment (`++`)

2. B) Right-to-left

3. C) 12

    - 🔍 **Explanation:**

        - `a = 5`
        - `a++` → uses 5 then increments to 6

- `++a` → increments to 7, then uses 7
- So, `b = 5 + 7 = 12`

4. D) All of these

   - These are all common mistakes with `while` loops:
     - Using `=` instead of `==`
     - Forgetting to update loop variable
     - Misusing brackets

👨‍💻 Author

🚀 Darshan Vasani

💡 Full-Stack Developer | Software Engineer | Mentor

🔗 Connect with me! 🌍

🌐 **Portfolio:** dpvasani56.vercel.app
🐙 **GitHub:** github.com/dpvasani
💼 **LinkedIn:** linkedin.com/in/dpvasani56
🌳 **Linktree:** linktr.ee/dpvasani56
🎓 **Credly:** credly.com/users/dpvasani57
🐦 **Twitter:** x.com/vasanidarshan56
📢 **Topmate:** topmate.io/dpvasani56

🚀 Follow me for more cool DSA problems & solutions! 🌟
