Operators and Errors

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

Module Overview

This module introduces students to the concept of operators in C++ and the types of errors encountered during programming. Operators allow manipulation and comparison of data through arithmetic, relational, logical, assignment, and bitwise operations. Special attention is given to the distinction between integer and floating-point division, as well as practical applications of the modulus operator. The module also introduces common error types—syntax, runtime, and logical errors—along with strategies to identify, differentiate, and prevent them. Together, these topics equip students with the tools to write correct, efficient, and error-free programs.

Learning Objectives

- *Explain the role of operators in C++ and classify them into categories.*
- *Use arithmetic operators, including handling of integer vs. floating-point division and modulus.*
- *Apply relational operators to compare values and make decisions.*
- *Construct logical expressions using AND, OR, and NOT operators.*
- *Demonstrate assignment and compound assignment operators for updating variables.*
- *Perform bitwise operations to manipulate binary data.*
- *Differentiate between syntax, runtime, and logical errors with examples.*
- *Apply debugging strategies and best practices to prevent and resolve common errors.*

Contents

Chapter 1

Introduction

In C++, an *operator* is a special symbol that tells the computer to perform a specific action or operation on one or more pieces of data. Think of an operator like a symbol in math class: for example, the "+" sign tells you to add two numbers. In programming, operators work in a similar way—they take one or more inputs (called *operands*) and produce an output.

C++ provides many different operators, and we can group them into categories based on what they do. In this guide, we'll explore five main categories of operators: **arithmetic operators** (for basic math like addition and subtraction), **relational operators** (for comparing values, like checking which number is bigger), **logical operators** (for combining true/false values and making decisions), **assignment operators** (for assigning values to variables and updating them), and **bitwise operators** (for working with the individual bits that make up numbers). Each section below will introduce the operators in that category, provide a summary table, and show a simple C++ code example to demonstrate how they work.

1.1 Arithmetic Operators

Arithmetic operators are used to carry out basic mathematical calculations on numbers. They take numerical values (operands) and compute a new number. Common arithmetic operators include addition ("`+`" for adding two numbers), subtraction ("`-`" for subtracting one number from another), multiplication ("`*`" for multiplying), division ("`/`" for dividing), and modulo ("`%`" for finding the remainder of a division).

One important thing to note is how division works with integers: if both operands are integers, the division operator "`/`" will give the whole number part of the result and discard any decimals (for example, `7 / 3` results in `2`, not `2.333...`). If you want a fractional result, at least one number must be a decimal (floating-point). The modulo operator "`%`" gives the remainder after division (for example, `7 % 3` is `1`, because 7 divided by 3 leaves a remainder of 1). In addition to these, C++ provides the increment ("`++`") and decrement ("`--`") operators, which increase or decrease an integer variable's value by one. For instance, if `x` is `5`, then `x++` will make `x` become `6`, and `x--` would bring it back down to `5`.

As a quick reference, Table 1.1 below summarizes common arithmetic operators in C++.

Operator	Description	Example Usage
`+`	Adds two values	4 + 2 = 6
`-`	Subtracts one value from another	5 - 3 = 2
`*`	Multiplies two values	3 * 4 = 12
`/`	Divides one value by another	8 / 2 = 4
`%`	Computes the remainder of a division (modulo)	7 % 3 = 1
`++`	Increment (adds 1 to a value)	if `x` = 5, after `x++`, `x` = 6
`--`	Decrement (subtracts 1 from a value)	if `x` = 5, after `x--`, `x` = 4

Table 1.1: Common C++ arithmetic operators.

For example, the program below uses arithmetic operators and prints the results:

Example: Arithmetic Operators

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

int main() {
    int first_value = 7, second_value = 3;
    cout << "first_value + second_value = " << (first_value + second_value)
        << endl;
    cout << "first_value - second_value = " << (first_value - second_value)
        << endl;
    cout << "first_value * second_value = " << (first_value * second_value)
        << endl;
    cout << "first_value / second_value = " << (first_value / second_value)
        << endl;
    cout << "first_value % second_value = " << (first_value % second_value)
        << endl;

    int counter_value = 5;
    cout << "counter_value = " << counter_value << endl;
    counter_value++;
    cout << "After counter_value++, counter_value = " << counter_value <<
       endl;
    counter_value--;
    cout << "After counter_value--, counter_value = " << counter_value <<
       endl;
    return 0;
}
```

Output:

```
first_value + second_value = 10
first_value - second_value = 4
first_value * second_value = 21
first_value / second_value = 2
first_value % second_value = 1
counter_value = 5
After counter_value++, counter_value = 6
After counter_value--, counter_value = 5
```

1.1.1 The Critical Distinction: Integer vs. Floating-Point Division

Division is the most complex arithmetic operator because its behavior fundamentally changes based on operand types. This is the single most important concept in this chapter: **integer division truncates (discards) the decimal part, while floating-point division preserves it.**

When both operands are integers, C++ performs integer division, which always produces an integer result by discarding any remainder. When at least one operand is floating-point, C++ performs floating-point division, which can produce a decimal result.

Integer division doesn't round to the nearest integer - it truncates toward zero. This means it simply drops the decimal part:

- $7 \div 2 = 3$ (not 3.5 or 4)
- $-7 \div 2 = -3$ (not -3.5 or -4)

- $1 \div 3 = 0$ (not 0.333... or 1)

Division by Zero - The Fatal Error

Division by zero is undefined in mathematics and causes serious problems in programming:

- **Integer division by zero:** Causes a runtime error (program crash)
- **Floating-point division by zero:** Produces special values (infinity or NaN)

LHS	Expression	Expression Type & Value	Stored in LHS
`int r;`	`7 / 2`	`int` → `3`	`3` (no fractional part)
`int r;`	`7 / 2.0`	`double` → `3.5`	`3` (truncation)
`int r;`	`7.0f / 2`	`float` → `3.5f`	`3` (truncation)
`int r;`	`7.0 / 2.0f`	`double` (float→double) → `3.5`	`3` (truncation)
`int r;`	`(7 / 2.0) + 0.9`	`double` → `4.4`	`4` (truncation)
`double r;`	`7 / 2`	`int` → `3`	`3.0`
`double r;`	`7 / 2.0`	`double` → `3.5`	`3.5`
`double r;`	`7.0f / 2`	`float` → `3.5f`	`3.5` (widen to double)
`double r;`	`7.0 / 2.0f`	`double` (float→double) → `3.5`	`3.5`

Table 1.2: Division: evaluation (by operand types) versus assignment (by LHS type).

1.1.2 Modulus Operator (%)

Basic Concept

The modulus operator (also called remainder operator) returns the remainder after integer division. It only works with integer operands - you cannot use % with floating-point numbers. This operator is incredibly useful for many programming tasks.

The relationship between division and modulus is:

$$a = (a/b) \times b + (a\%b)$$

For example: $7 = (7/2) \times 2 + (7\%2) = 3 \times 2 + 1 = 7$

How Modulus Works

When you write `a % b`, the computer:

1. Performs integer division `a / b`
2. Multiplies the quotient by `b`
3. Subtracts this product from `a`
4. Returns the difference as the remainder

Important Applications

The modulus operator is essential for:

- **Checking divisibility:** If `a % b == 0`, then `a` is divisible by `b`
- **Even/Odd detection:** `n % 2 == 0` means `n` is even
- **Extracting digits:** `n % 10` gives the last digit
- **Wrapping values:** Creating circular patterns (like clock arithmetic)
- **Hash tables:** Mapping large numbers to array indices

1.2 Relational Operators

Relational operators (also called *comparison operators*) are used to compare two values. They allow a program to answer questions like "Is A equal to B?" or "Is A greater than B?". The result of a comparison is a boolean value: either `true` (meaning the comparison condition is satisfied) or `false` (meaning it is not).

C++ provides six common relational operators: `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to). Note that the equality comparison operator is a double equal sign `==` – a single `=` is used for assignment, not comparison.

Table 1.3 summarizes the relational operators and gives examples of expressions that evaluate to true or false.

Operator	Description	Example Usage
`==`	Checks if two values are equal	5 == 5 is `true`
`!=`	Checks if two values are not equal	5 != 5 is `false`
`<`	Checks if left value is less than right value	3 < 5 is `true`
`>`	Checks if left value is greater than right value	3 > 5 is `false`
`<=`	Checks if left value is less than or equal to right value	3 <= 3 is `true`
`>=`	Checks if left value is greater than or equal to right value	8 >= 5 is `true`

Table 1.3: C++ relational (comparison) operators.

This sample program demonstrates relational operators in action:

Example: Relational Operators

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

int main() {
    int first_number = 5, second_number = 8;
    cout << boolalpha;  // print bool as "true"/"false"
    cout << first_number << " == " << second_number << " is " << (
       first_number == second_number) << endl;
    cout << first_number << " != " << second_number << " is " << (
       first_number != second_number) << endl;
    cout << first_number << " < " << second_number << " is " << (
       first_number < second_number) << endl;
    cout << first_number << " > " << second_number << " is " << (
       first_number > second_number) << endl;
    cout << first_number << " <= " << first_number << " is " << (
       first_number <= first_number) << endl;
    cout << second_number << " >= " << first_number << " is " << (
       second_number >= first_number) << endl;
    return 0;
}
```

Output:

```
5 == 8 is false
5 != 8 is true
5 < 8 is true
5 > 8 is false
5 <= 5 is true
8 >= 5 is true
```

1.3 Logical Operators

Logical operators are used to combine or modify boolean values (values that are either `true` or `false`). These operators help build more complex conditions out of simpler ones. There are three logical operators in C++: logical AND (written as `&&`), logical OR (`||`), and logical NOT (`!`).

Logical AND (`&&`) gives `true` only if *both* conditions are true. Logical OR (`||`) gives `true` if *at least one* of the conditions is true. Logical NOT (`!`) takes a single boolean value and reverses it (turning `true` to `false`, or vice versa). For example, the condition `(x > 0 && x % 2 == 0)` is true only when `x` is positive *and* even. Typically, you'll use logical operators inside `if` statements or loops to decide if certain conditions are met.

Table 1.4 gives an overview of the logical operators with examples of outcomes.

Operator	Description	Example Usage
`&&`	Logical AND (true if both conditions are true)	`true && false` is `false`
`||`	Logical OR (true if at least one condition is true)	`true || false` is `true`
`!`	Logical NOT (true if the condition is false)	`!true` is `false`

Table 1.4: C++ logical operators (for boolean logic).

Below is an example program showing how logical operators evaluate combinations of true/false values:

Example: Logical Operators

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

int main() {
    bool is_raining = true;
    bool is_sunny = false;
    cout << boolalpha;  // print bool as "true"/"false"
    cout << is_raining << " && " << is_sunny << " = " << (is_raining &&
       is_sunny) << endl;
    cout << is_raining << " || " << is_sunny << " = " << (is_raining ||
       is_sunny) << endl;
    cout << "!" << is_raining << " = " << (!is_raining) << endl;
    cout << "!" << is_sunny << " = " << (!is_sunny) << endl;
    return 0;
}
```

Output:

```
true && false = false
true || false = true
!true = false
!false = true
```

1.4 Assignment Operators

Assignment operators are used to give a value to a variable. The basic assignment operator is the single equals sign `=`. Using `=` means "take the value on the right and store it in the variable on the left." For example, `x = 5` sets the variable `x` to 5. (Note that `=` is not the same as `==`; `==` checks for equality, while `=` assigns a value.)

C++ also provides *compound assignment* operators that update a variable by performing an operation and assignment in one step. These include `+=`, `-=`, `*=`, `/=`, and `%=`. For instance, `x += 3` adds 3 to the current value of `x` (equivalent to `x = x + 3`), and `x -= 2` subtracts 2 from `x` (equivalent to `x = x - 2`). Using these combined operators can make code shorter and easier to read.

Table 1.5 below lists the basic assignment operator and some compound assignment operators, with usage examples.

The following program shows how assignment and compound assignment operators update a variable step by step:

Operator	Description	Example Usage
`=`	Assigns the value on the right to the variable on the left	`x = 10`
`+=`	Adds the right value to the variable (and reassigns the sum)	if `x` = 5, then `x += 3` gives `x = 8`
`-=`	Subtracts the right value from the variable	if `x` = 5, then `x -= 2` gives `x = 3`
`*=`	Multiplies the variable by the right value	if `x` = 4, then `x *= 3` gives `x = 12`
`/=`	Divides the variable by the right value	if `x` = 12, then `x /= 4` gives `x = 3`
`%=`	Takes the remainder of the variable divided by the right value	if `x` = 7, then `x %= 3` gives `x = 1`

Table 1.5: C++ assignment and compound assignment operators.

Example: Assignment Operators

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

int main() {
    int current_value;
    current_value = 3;
    cout << "After current_value = 3, current_value = " << current_value <<
        endl;
    current_value += 2;
    cout << "After current_value += 2, current_value = " << current_value
       << endl;
    current_value -= 1;
    cout << "After current_value -= 1, current_value = " << current_value
       << endl;
    current_value *= 3;
    cout << "After current_value *= 3, current_value = " << current_value
       << endl;
    current_value /= 2;
    cout << "After current_value /= 2, current_value = " << current_value
       << endl;
    current_value %= 3;
    cout << "After current_value %= 3, current_value = " << current_value
       << endl;
    return 0;
}
```

Output:

```
After current_value = 3, current_value = 3
After current_value += 2, current_value = 5
After current_value -= 1, current_value = 4
After current_value *= 3, current_value = 12
After current_value /= 2, current_value = 6
After current_value %= 3, current_value = 0
```

1.5 Bitwise Operators

Bitwise operators allow you to manipulate numbers at the level of their individual binary digits (bits). Remember that computers represent all numbers in binary (a series of 0s and 1s). These operators treat the binary form of integers as a sequence of bits and operate on those bits directly.

C++ has six common bitwise operators: `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (left shift), and `>>` (right shift). The bitwise AND, OR, and XOR operators combine two numbers bit-by-bit: - **AND** (`&`) produces a 1 in each bit position where *both* operands have a 1 (and 0 otherwise). - **OR** (`|`) produces a 1 in each bit position where *at least one* operand has a 1. - **XOR** (`^`) (exclusive OR) produces a 1 in each bit position where *exactly one* operand has a 1 (one or the other, but not both). The bitwise NOT operator (`~`) is unary (takes one operand) and flips all bits of its operand (turning 1s into 0s and 0s into 1s).

The shift operators move all bits of a number left or right. The left shift operator `<<` moves bits to the left, and the right shift operator `>>` moves bits to the right. Shifting left by one position is equivalent to multiplying by 2, and shifting right by one position is equivalent to dividing by 2 (for positive numbers, discarding any remainder). For example, `5 << 1` shifts the bits of 5 (which is `0101` in binary) to the left by 1 place, resulting in `1010` (which is 10 in decimal). Similarly, `5 >> 1` would result in `0010` (which is 2 in decimal).

Table 1.6 provides a summary of the bitwise operators with examples of their effects on numbers.

Operator	Description	Example Usage
`&`	Bitwise AND (1 in a bit if both bits are 1)	`6 & 3 = 2`
`|`	Bitwise OR (1 in a bit if either bit is 1)	`6 | 3 = 7`
`^`	Bitwise XOR (1 in a bit if bits are different)	`6 ^ 3 = 5`
`~`	Bitwise NOT (inverts all bits)	`~6 = -7`
`<<`	Left shift (shifts bits to the left)	`5 << 1 = 10`
`>>`	Right shift (shifts bits to the right)	`5 >> 1 = 2`

Table 1.6: C++ bitwise operators (operate on binary bits).

Finally, here is a program demonstrating various bitwise operations on example numbers:

Example: Bitwise Operators

```
// Demonstrating bitwise operators
#include <iostream>
using namespace std;

int main() {
    int first_bits = 6, second_bits = 3;
    cout << "first_bits & second_bits = " << (first_bits & second_bits) <<
       endl;
    cout << "first_bits | second_bits = " << (first_bits | second_bits) <<
       endl;
    cout << "first_bits ^ second_bits = " << (first_bits ^ second_bits) <<
       endl;
    cout << "~first_bits = " << (~first_bits) << endl;

    int shift_value = 5;
    cout << "shift_value << 1 = " << (shift_value << 1) << endl;
    cout << "shift_value >> 1 = " << (shift_value >> 1) << endl;
    return 0;
}
```

Output:

```
first_bits & second_bits = 2
first_bits | second_bits = 7
first_bits ^ second_bits = 5
~first_bits = -7
shift_value << 1 = 10
shift_value >> 1 = 2
```

Chapter 2

Errors

2.1 Common Error Types in C++

When students begin writing programs using variables and operators, three classes of mistakes commonly appear: **syntax errors**, **runtime (execution) errors**, and **logical errors**. This short note defines each, shows a very simple example, explains why it happens, and suggests how to avoid it.

Quick Definitions

- **Syntax Error:** The code violates C++ grammar (compiler rules). Detected at *compile time*. Program does not build.
- **Runtime Error:** The code compiles, but something goes wrong *while running* (e.g., dividing by zero, invalid array access). Program crashes or throws an exception.
- **Logical Error:** The code compiles and runs without crashing, but the *answer is wrong* because the logic or formula is incorrect.

2.2 Syntax Errors

Missing Semicolon

Example: Missing semicolon after a statement

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

int main() {
    int value_x = 10    // <- ERROR: missing semicolon
    cout << value_x << "\n";
    return 0;
}
```

Why it happens: In C++, most statements must end with `;`.
Typical compiler message: "expected ';' before 'cout'."
How to avoid/fix:

- Scan the previous line when the compiler flags an error.
- Use an editor with syntax highlighting and auto-formatting.

Undeclared Identifier

Example: Using a variable that was never declared

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

int main() {
    value_y = 5;                    // <- ERROR: 'value_y' was not declared in
        this scope
    cout << value_y << "\n";
    return 0;
}
```

Why it happens: Every variable must be declared with a type before use.
Fix: `int value_y = 5;` before using `value_y`.

2.3 Runtime Errors

Division by Zero

Example: Division by zero at runtime

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

int main() {
    int dividend = 10;
    int divisor = 0;                    // divisor happens to be zero
    cout << (dividend / divisor) << "\n";    // <- RUNTIME ERROR (undefined
        behavior)
    return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

Why it happens: Integer division by zero is undefined. The compiler cannot always detect that `divisor` is zero ahead of time.
How to avoid/fix:

- Check the denominator before dividing: `if (divisor != 0) {...} else {...}`.
- Validate user input.

Out-of-Range Index (std::vector)

Example: Accessing an element that doesn't exist

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

int main() {
    vector<int> number_list = {1, 2, 3};
    cout << number_list[3] << "\n";        // <- RUNTIME ISSUE: valid
       indices are 0,1,2
    return 0;
}
```

Output:

```
0 (or garbage value, or segmentation fault)
```

Why it happens: Index `3` is out of bounds for a 3-element vector.
How to avoid/fix:

- Ensure index `i` satisfies `0 <= i && i < number_list.size()`.
- Use `number_list.at(i)` during development; it throws an exception on invalid access.

2.4 Logical Errors

Wrong Formula (Average)

Example: Average computed incorrectly

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

int main() {
    int first_num = 7, second_num = 8;
    // Intended: average of first_num and second_num as a double
    double average_value = first_num + second_num / 2;      // <- LOGICAL
       ERROR: operator precedence
    cout << average_value << "\n";
    return 0;
}
```

Output:

```
11
```

What went wrong: `second_num / 2` happens first (due to precedence), then `first_num + (second_num/2)`. With integers, `8/2` is `4`; then `7 + 4 = 11`.
Expected output: `7.5`
Fixes:

- Add parentheses: `double average_value = (first_num + second_num) / 2.0;`
- Use a floating literal to avoid integer division.

Using Assignment in a Condition

Example: '=' (assignment) instead of '==' (comparison)

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

int main() {
    int test_value = 0;
    if (test_value = 5) {                // <- LOGICAL BUG: assigns 5 to
        test_value; condition becomes true
         cout << "Hello\n";
    }
    return 0;
}
```

Output:

```
Hello
```

What went wrong: `=` assigns; `==` compares. After `test_value = 5`, the condition is nonzero (true), so "Hello" always prints regardless of the original value.

Fix: `if (test_value == 5) { ... }`. Some style guides prefer constants on the left (`if (5 == test_value)`) to catch accidental assignments.

2.5 How to Tell Them Apart & Practical Tips

- **Syntax error** → Compiler stops with messages pointing to a line/column. Fix grammar (missing `;`, wrong type, undeclared name).
- **Runtime error** → Program starts, then crashes or throws when a bad event happens (e.g., out-of-range, division by zero). Add checks and input validation.
- **Logical error** → Program runs but outputs wrong results. Use print statements, tiny test cases, and *trace* the computation.

2.6 Mini Checklist

- End statements with `;`. Declare every variable with a type.
- Re-check **operator precedence** and **integer vs. floating** division; add parentheses for clarity.
- Validate inputs before dividing or indexing arrays/vectors.
- Compare with `==`, not assign with `=`, inside conditions.
- Test with small, known examples and print intermediate values.

2.7 Best Practices for Error Prevention

1. **For Syntax Errors:**
    - Use a good IDE with syntax highlighting
    - Pay attention to compiler warnings

- Follow consistent coding style

2. **For Runtime Errors:**
   - Always validate input data
   - Check for null pointers before dereferencing
   - Use bounds checking for arrays
   - Handle exceptions appropriately
3. **For Logical Errors:**
   - Write comprehensive test cases
   - Use debugging tools and print statements
   - Perform code reviews
   - Document your logic clearly
   - Test edge cases thoroughly