C++ Programming Fundamentals

Master the essential building blocks of programming with hands-on examples and clear explanations

What You'll Learn Today

This week we're diving into the core operators and concepts that make C++ powerful. These building blocks will help you write smarter, more efficient code.

1

Modulus Operator

Finding remainders and working with cycles

2

Operator Precedence

Understanding the order of operations

3

Increment & Decrement

Prefix and postfix variations explained

4

Relational Operators

Comparing values and making decisions

The Modulus Operator (%)

The modulus operator is one of the most useful tools in programming. Think of it like division, but instead of giving you the answer, it gives you the **remainder**. Imagine you have 17 cookies and you want to divide them equally among 5 friends. Each friend gets 3 cookies (17 ÷ 5 = 3), but you have 2 cookies left over. That leftover amount—2—is what the modulus operator returns!

In real programming, the modulus operator is perfect for checking if numbers are even or odd, cycling through patterns, or determining if one number divides evenly into another.

Syntax: `dividend % divisor`

Example:

```
int result = 17 % 5;  // result is 2
int remainder = 23 % 4; // remainder is 3
int check = 10 % 2;     // check is 0 (10 is even)
```

Key Insight: If `a % b = 0`, then **a is divisible by b**. This is how we check for even numbers (number % 2 == 0) or if a number is a multiple of something.

Understanding Operator Precedence

When you write a mathematical expression in C++, the computer doesn't just calculate left to right. Just like in regular math, certain operations happen first. This is called **operator precedence**—the order in which operations are performed.

For example, in the expression `2 + 3 * 4`, multiplication happens before addition, so you get 14, not 20. If you wanted addition to happen first, you'd use parentheses: `(2 + 3) * 4 = 20`.

Common Precedence Order (highest to lowest):

1. **Parentheses ():** Always evaluated first
2. **Multiplication, Division, Modulus (*, /, %):** Left to right
3. **Addition, Subtraction (+, -):** Left to right

Example:

```
int a = 10 + 5 * 2;       // = 20 (multiply first, then add)
int b = (10 + 5) * 2;      // = 30 (parentheses first)
int c = 20 % 6 / 2;        // = 2 (left to right: 20%6=2, then 2/2=1... wait!)
int d = 20 / 2 % 6;        // = 2 (left to right: 20/2=10, then 10%6=4)
```

Always use parentheses when unsure! It makes your code clearer and prevents mistakes. `(10 + 5) * 2` is better than relying on the computer to know what you mean.

Increment & Decrement Operators

The increment (++) and decrement (--) operators are shortcuts for adding 1 or subtracting 1 from a variable. Instead of writing `x = x + 1`, you can simply write `x++` or `++x`. Both do the same thing, but they work differently depending on where you put them!

There are two versions: **prefix** (operator before the variable) and **postfix** (operator after the variable). The difference matters when you use the result in an expression.

Postfix (x++ or x--): The variable is changed, but the expression returns the **old value** first

```
int x = 5;
int y = x++;  // y gets 5, THEN x becomes 6
// Now: x = 6, y = 5
```

Prefix (++x or --x): The variable is changed first, and the expression returns the **new value**

```
int x = 5;
int y = ++x;  // x becomes 6 first, THEN y gets 6
// Now: x = 6, y = 6
```

Pro Tip: In simple loops, it doesn't matter which you use. But be careful in complex expressions!

Prefix vs Postfix: Real Examples

Let's see exactly how prefix and postfix operators behave in different situations. Understanding this difference is crucial for writing correct C++ code.

Postfix: x++

The current value is used first, then the variable increments.

```
int count = 10;
int result = count++;
// result = 10
// count = 11

cout << result << endl;  // prints 10
cout << count << endl;   // prints 11
```

Prefix: ++x

The variable increments first, then the new value is used.

```
int count = 10;
int result = ++count;
// count = 11
// result = 11

cout << result << endl;  // prints 11
cout << count << endl;   // prints 11
```

In Loop Statements: Both work the same because the return value isn't used

```
for (int i = 0; i < 5; i++) // Both i++ and ++i work identically here
 cout << i << " ";
// Output: 0 1 2 3 4
```

Relational Operators: Comparing Values

Relational operators allow you to compare two values and get a **true or false answer**. These operators are essential for making decisions in your code—they're how your program chooses which path to take.

When you use a relational operator, the result is always a **boolean value**: either `true` (1) or `false` (0). You'll use these operators constantly in if-statements, loops, and conditions.

The Six Main Relational Operators:

Operator	Symbol	Meaning
Equal to	`==`	Checks if two values are the same
Not equal to	`!=`	Checks if two values are different
Greater than	`>`	Left value is bigger than right
Less than	`<`	Left value is smaller than right
Greater than or equal	`>=`	Left is bigger or equal to right
Less than or equal	`<=`	Left is smaller or equal to right

Don't confuse `==` (comparison) with `=` (assignment)! This is one of the most common mistakes beginners make.

Using Relational Operators: Practical Examples

Relational operators return true or false, which we typically use in conditional statements to control program flow. Here's how they work in real code:

Simple Comparisons:

```
int age = 25;
bool canVote = (age >= 18);    // true
bool isTeenager = (age < 13);  // false
bool exactAge = (age == 25);   // true
```

Using in If-Statements:

```
int score = 85;

if (score >= 90) {
  cout << "Grade A" << endl;
}
else if (score >= 80) {
  cout << "Grade B" << endl;  // This executes
}
else if (score >= 70) {
  cout << "Grade C" << endl;
}
```

Combining Multiple Comparisons: (More on this next time!)

```
int temperature = 25;
bool isComfortable = (temperature >= 20 && temperature <= 30);
// true (if temperature is between 20 and 30)
```

Notice how we check conditions in order—the first true condition executes, and the rest are skipped. This is how programs make intelligent decisions!

Putting It All Together: A Complete Example

Let's create a program that uses all the operators we've learned: modulus for checking patterns, operator precedence for calculations, increment operators for loops, and relational operators for decision-making.

```
// Program: Check if numbers are even and display them
#include <iostream>
using namespace std;

int main() {
    int count = 0;
    int number = 1;

    // Loop while count is less than 5
    while (count < 5) {
        // Check if number is even using modulus
        if (number % 2 == 0) {
            cout << number << " is even" << endl;
            count++;  // Increment count (postfix doesn't matter here)
        }
        number++;  // Move to next number
    }

    // Calculate with precedence
    int result = 10 + 5 * 2;  // Multiply first: 10 + 10 = 20
    cout << "Result: " << result << endl;

    // Show prefix vs postfix
    int x = 3;
    cout << x++ << " " << ++x << endl;  // prints: 3 5

    return 0;
}
```

Output: `2 is even` / `4 is even` / `6 is even` / `8 is even` / `10 is even` / `Result: 20` / `3 5`

Key Takeaways & Next Steps

You've now mastered five essential C++ concepts that form the foundation of all programming!

Modulus (%)

Returns the remainder; use it to check even/odd or divisibility

Precedence

Parentheses first, then */%, then +-. Always use parens when unsure

Increment/Decrement

Prefix (++x) vs Postfix (x++) differ in when the value is used

Relational Ops

Compare values (==, !=, >, <, >=, <=) to control program flow

---

Before Your Next Lecture: Practice writing simple programs using these operators. Try checking if numbers are divisible by specific values, calculating expressions with mixed operators, and creating if-statements that compare different values. The more you practice, the more natural these concepts become!