Chapter 3

Operators

Introduction

This chapter will cover essential operators and input/output functions in C programming, which are fundamental for performing operations and handling data. Topics include arithmetic operators for basic calculations, relational operators for comparing values, logical operators for combining conditions, and bitwise operators for manipulating individual bits of data. Unary operators, assignment, and conditional operators will also be discussed, as well as how operator precedence and associativity affect the evaluation of expressions. Additionally, the chapter will introduce both unformatted and formatted input/output functions in C, enabling efficient handling of user input and data display. These topics provide the building blocks for controlling the flow and manipulating data in C programs.

Structure

The chapter covers the following topics:

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operator
* Unary operators
* Assignment and conditional operators
* Precedence and associativity of operators
* Input/output: Unformatted and formatted I/O function in C

Objectives

The objectives of this chapter are to equip learners with the knowledge and skills needed to use various types of operators in C, including arithmetic, relational, logical, bitwise, unary, assignment, and conditional operators. The chapter aims to enhance understanding of operator precedence and associativity in expressions, and to develop proficiency in using unformatted and formatted input/output functions for effective data handling. By the end of this chapter, learners should be able to confidently apply these operators and I/O functions to control program flow, manipulate data, and interact with users in real-world C programming scenarios.

Arithmetic operators

Arithmetic operators are fundamental tools in programming and mathematics used to perform basic calculations on numbers. These operators include addition (+), subtraction (-), multiplication (\*), division (/), modulus (%), exponentiation (\*\*), and floor division (//). Each operator serves a specific purpose: addition sums values, subtraction finds differences, multiplication calculates products, division computes quotients, modulus returns remainders, exponentiation raises numbers to powers, and floor division provides integer results by discarding the decimal part. These operators follow precedence rules to ensure accurate results in complex expressions. Here is a detailed explanation of common arithmetic operators used in programming and mathematics:

* **Addition (+):** The addition operator adds two numbers together. It is one of the most basic operations commonly used to sum numbers.
* Example in mathematics:

5 + 3 = 8

* Example in programming:

a = 5 + 3 → In this case, the variable a will hold the value eight after the addition operation is performed.

* **Usage:** This operator is frequently used to combine values, such as adding the prices of multiple items or calculating total scores.
* **Subtraction (-):** Subtraction takes the second number and subtracts it from the first number. It is used to calculate the difference between two values.
* Example in mathematics:

9 - 4 = 5

* Example in programming:

b = 9 - 4 → The variable b will hold the value five after the subtraction.

* **Usage:** Subtraction is used for finding the difference between numbers, such as determining how much money is left after a purchase or calculating the distance between two points.
* **Multiplication (\*):** The multiplication operator multiplies two numbers together, providing the product of the two values.
* Example in mathematics:

6 \* 7 = 42

* Example in programming:

c = 6 \* 7 → The variable c will hold the value 42.

* **Usage:** Multiplication is useful when scaling numbers, such as calculating the total cost when buying multiple units of an item or determining the area of a rectangle (length × width).
* **Division (/):**  Division divides the first number by the second number, yielding the quotient.
* Example in mathematics:

8 / 2 = 4

* Example in programming:

d = 8 / 2 → The variable d will hold the value 4.

* **Usage:** Division is used to distribute a quantity into equal parts, such as dividing total sales by the number of items sold to find the average price.

Note: In some programming languages, division between integers may return an integer result by rounding down (integer division), while others may return a floating-point result.

* **Modulus (%):** The modulus operator returns the remainder after dividing the first number by the second number.
* Example in mathematics:

10 % 3 = 1 (because 10 divided by 3 gives a quotient of 3 and a remainder of 1).

* Example in programming:

e = 10 % 3 → The variable e will hold the value 1.

* **Usage:** Modulus is commonly used to check whether a number is divisible by another number (i.e., if the remainder is zero), or for cyclic operations like determining the position in a circular buffer or rotating through options.
* **Exponentiation (\*\* or ^):** Exponentiation raises the first number (the base) to the power of the second number (the exponent). In many programming languages like Python, this operator is represented by \*\*, while in others, the caret symbol (^) is used.
* Example in mathematics:

2 \*\* 3 = 8 (2 raised to the power of 3 equals 8).

* Example in programming:

f = 2 \*\* 3 → The variable f will hold the value 8.

* **Usage:** Exponentiation is used in a variety of fields, such as calculating compound interest, determining growth rates, or performing scientific calculations.
* **Floor Division (//):** Floor division divides the first number by the second number and rounds down the result to the nearest integer. It discards any fractional part of the quotient.
* Example in mathematics:

7 // 2 = 3 (the result of dividing 7 by 2 is 3.5, but floor division rounds down to 3).

* Example in programming:

g = 7 // 2 → The variable g will hold the value 3.

* **Usage:** Floor division is useful when only whole units are meaningful, such as when dividing objects into groups and ignoring partial groups, or for working with integer-based data types where decimal values are unnecessary.

Table 3.1. summarizes these arithmetic operators in programming languages like Python, C, or Java:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example in Python / C / Java** |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (Remainder) | a % b |
| \*\* | Exponentiation (Python) | a \*\* b |
| // | Floor Division (Python) | a // b |

Relational operators

Relational operators in programming are used to compare two values or expressions. The result of these comparisons is always a Boolean value: True if the relationship holds, and False if it does not. Relational operators are crucial for decision-making in programming, as they allow developers to execute certain code based on the outcome of comparisons. These operators are often used in conditional statements (like if, else), loops (while, for), and more. Here is a detailed look at the most common relational operators in programming:

* **Equal to (==):** Compares if two values are equal.
* **Output***:* Returns True if the values are equal, otherwise False. Example:

In this example, the condition a == b evaluates to True because both a and b are equal.

* **Not equal to (!=):** Compares if two values are not equal.
* **Output***:* Returns True if the values are not equal, otherwise False. Example:

a = 5

b = 10

if a != b:

print("a is not equal to b") # This will print

In this case, a != b evaluates to True because a (5) and b (10) are different.

* **Greater than (>):** Checks if the left operand is greater than the right operand.
* **Output***:* Returns True if the left value is greater, otherwise False.
* Example:

Here, x > y evaluates to True because 7 is greater than 5.

* **Less than (<):** Checks if the left operand is less than the right operand. *Output:* Returns True if the left value is less, otherwise, False. Example:

Since 3 is less than 8, the condition x < y evaluates to True.

* **Greater than or equal to (>=)**: Checks if the left operand is greater than or equal to the right operand.
* **Output***:* Returns True if the left value is greater than or equal to the right, otherwise False. Example:

Since a is equal to b, the condition a >= b is True.

* **Less than or equal to (<=):** Checks if the left operand is less than or equal to the right operand.
* **Output***:* Returns True if the left value is less than or equal to the right, otherwise False. Example:

Since the score is equal to the passing score, the condition score <= passing\_score evaluates to True.

Relational operators in control structures

Relational operators are frequently used in control structures such as if, else, and loops, as they help in determining the flow of a program based on conditions.

Example with the if-else statement:

Example with the while loop:

Here, the while loop keeps running as long as the condition count <= 5 is True.

Logical combination with relational operators

Relational operators are often combined with logical operators (and, or, not) to form more complex conditions.

Example:

In this case, both conditions, age > 18 and income > 30000, need to be True for the statement inside the if block to be executed.

Logical operators

Logical operators are used in programming to perform logical operations, combining multiple conditions or expressions. These operators return a Boolean value (True or False) and are essential in controlling the flow of programs, especially in decision-making processes like if- else statements and loops. They allow us to evaluate complex conditions, often by combining the results of relational operators. The three main logical operators are AND, OR, and NOT.

AND (&& or AND)

The AND operator checks if both conditions are true. It returns True if both the operands (conditions) are true; otherwise, it returns False.

Symbols in different languages:

* In Python: and
* In C, Java, JavaScript, etc.: &&

The Truth Table for AND is mentioned in *Table 3.2*:

|  |  |  |
| --- | --- | --- |
| **Condition 1** | **Condition 2** | **Result** |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

Example:

In this case, both conditions (age > 18 and income > 30000) are true, so the program prints the statement. The AND operator ensures both conditions must be True for the statement inside the if block to execute. If either one of them is False, the program will skip or execute the else block.

OR (|| or or)

The OR operator checks if at least one condition is true. It returns True if any of the operands (conditions) is true; otherwise, it returns False.

Symbols in different languages:

* In Python: or
* In C, Java, JavaScript, etc.: ||

Truth Table for OR (*Table 3.3*):

|  |  |  |
| --- | --- | --- |
| **Condition 1** | **Condition 2** | **Result** |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

Example:

In this case, even though the age is less than 18, the second condition, parent\_permission, is True, so the overall condition evaluates to True. The OR operator only needs one of the conditions to be True to execute the code.

NOT (! or not)

The NOT operator reverses the result of the condition. If the condition is True, NOT makes it False, and if it is False, NOT makes it True.

Symbols in different languages:

* In Python: not
* In C, Java, JavaScript, etc.: !

Truth Table for NOT (*Table 3.4*):

|  |  |
| --- | --- |
| **Condition** | **Result** |
| True | False |
| False | True |

Example:

The condition is\_raining is False. Applying the NOT operator to False makes it True, so the program prints the first statement.

Logical operators and control structures

Logical operators are heavily used in control structures such as if-else statements and loops to make decisions based on multiple conditions.

Example with if-else:

Example with while loop:

Bitwise operator

In C programming, bitwise operators are used to manipulate data at the bit level, which means they operate on individual bits of the operands. Bitwise operations are often used for tasks like setting or clearing specific bits, performing shifts, masking, and other low-level tasks (*Table 3.5*).

Bitwise operators in C:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| ^ | Bitwise XOR (exclusive OR) |
| ~ | Bitwise NOT |
| << | Left Shift |
| >> | Right Shift |

**Table 3.5:** Bitwise operators

* **Bitwise AND (&):** Compares each bit of two operands. If both bits are 1, the resulting bit is 1; otherwise, it is 0.
* **Usage***:* Commonly used for masking certain bits. Example:
* **Bitwise OR (|):** Compares each bit of two operands. If at least one of the bits is 1, the resulting bit is 1; otherwise, it is 0.
* **Usage***:* Used to set specific bits to 1. Example:
* **Bitwise XOR (^):** Compares each bit of two operands. If one of the bits is 1 and the other is 0, the resulting bit is 1; otherwise, it is 0.
* **Usage***:* Useful for toggling specific bits. Example:
* **Bitwise NOT (~):** Inverts all the bits of the operand. Each 1 becomes 0, and each 0 becomes 1. In signed integers, this also performs a two's complement negation.
* Example:
* **Bitwise left shift (<<):** Shifts the bits of the first operand to the left by the number of positions specified by the second operand. Zeros are shifted in from the right.
* **Effect**: Left shifting by one position is equivalent to multiplying the number by 2. Example:

#include <stdio.h> int main() {

int a = 5; // 0101 in binary

int result = a << 1; // result is 10 (binary: 1010) printf("Left Shift: %d\n", result); // Output: 10

* **Bitwise right shift (>>):** Shifts the bits of the first operand to the right by the number of positions specified by the second operand. Depending on the system, either 0 or the sign bit is shifted in from the left (for signed integers).
* **Effect***:* Right shifting by one position is equivalent to dividing the number by 2. Example:

Practical applications of bitwise operators in C

The practical applications are as follows:

* **Setting/checking flags**: Bitwise operators are often used to work with flags, where each bit of an integer can represent an on/off state:
* **Bitmasking***:* You can use bitwise operators to create a mask and manipulate specific bits within a value.

Unary operators

Unary operators in programming are operators that operate on a single operand. They perform various operations, such as incrementing or decrementing a value, negating a number, or performing bitwise operations. Unary operators are commonly found in many programming languages, including C, C++, Java, and Python. Here’s a detailed explanation of the various unary operators:

The types of unary operators are as follows:

* **Unary plus (+):** Indicates that the value is positive. It does not change the value but is often used for clarity. Example:
* **Unary minus (-):** Negates the value of the operand. If the operand is positive, it becomes negative, and vice versa. Example:
* **Increment operator (++):** Increases the value of the operand by 1. It can be used in two forms:
* Prefix (++a): Increments the value before it is used in an expression.
* Postfix (a++): Increments the value after it is used in an expression.
* *Example:*

int a = 5;

int b = ++a; // a is 6, b is 6 (prefix) int c = a++; // a is 7, c is 6 (postfix)

* **Decrement operator (--):** Decreases the value of the operand by 1. Similar to the increment operator, it has prefix and postfix forms.
* Example:
* **Logical NOT (!):** Inverts the truth value of a boolean expression. If the expression is true, it becomes false, and vice versa.
* Example:
* **Bitwise NOT (~):** Inverts each bit of the operand. Each 1 becomes 0, and each 0 becomes 1. Example:

int a = 5; // 0101 in binary

int b = ~a; // b is -6 (in binary: 1010 in two's complement)

Example usage of unary operators in C

Here is a simple C program demonstrating various unary operators:

Increment and Decrement Operators

c

Copy code

#include <stdio.h>

int main() {

int a = 5;

int b = 10;

printf("a = %d\n", a++);

printf("After incrementing, a = %d\n", a);

printf("b = %d\n", --b);

printf("After decrementing, b = %d\n", b);

return 0;

}

**Output:**

a = 5

After incrementing, a = 6

b = 9

After decrementing, b = 9

int c = ++a;

// Pre-increment

int d = a--;

// Post-decrement

int e = !a;

// Logical NOT

int f = ~a;

// Bitwise NOT

printf("Original a: %d\n", 5);

printf("Unary Minus (b): %d\n", b); // Output: -5 printf("Pre-increment (c): %d\n", c); // Output: 6

printf("Post-decrement (d): %d\n", d); // Output: 6 (after decrement, a is 5) printf("Logical NOT (e): %d\n", e); // Output: 0 (since a is 5) printf("Bitwise NOT (f): %d\n", f); // Output: -6 (in two's complement)

return 0;

}

Summary of unary operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Unary plus | +a |
| - | Unary minus | -a |
| ++ | Increment | ++a or a++ |
| -- | Decrement | --a or a-- |
| ! | Logical NOT | !a |
| ~ | Bitwise NOT | ~a |

Assignment and conditional operators

Assignment operators are used to assign values to variables in programming. The simplest form of an assignment operator is the equals sign (=), but there are several compound assignment operators that combine assignment with another operation. Here is a detailed look at assignment operators, particularly in the context of C programming:

* **Basic assignment operator:**
* **Simple assignment (=):** Assigns the value of the right-hand operand to the left-hand operand. Example:
* **Compound assignment operators:** These operators combine an arithmetic operation with assignment, allowing for more concise code (*Table 3.7*).

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Equivalent expression** |
| += | Add and assign | a += b is equivalent to a = a + b |
| -= | Subtract and assign | a -= b is equivalent to a = a - b |
| \*= | Multiply and assign | a \*= b is equivalent to a = a \* b |
| /= | Divide and assign | a /= b is equivalent to a = a / b |
| %= | Modulus and assign | a %= b is equivalent to a = a % b |
| <<= | Left shift and assign | a <<= b is equivalent to a = a << b |
| >>= | Right shift and assign | a >>= b is equivalent to a = a >> b |
| &= | Bitwise AND and assign | a &= b is equivalent to a = a & b |
| ` | =` | Bitwise OR and assign |
| ^= | Bitwise XOR and assign | a ^= b is equivalent to a = a ^ b |

Example usage of assignment operators in C:

Conditional operators

Conditional operators, also known as ternary operators, are used to evaluate a boolean expression and return one of two values based on the result. The conditional operator is represented by the syntax ? : and is a concise way to write simple if-else statements.

Syntax: *condition ?*

* Boolean

Example usage of conditional operators in C:

Assignment Operators are essential for assigning values to variables, with simple (=) and compound forms (+=, -=, etc.), to combine assignment with arithmetic or bitwise operations.

Conditional Operators provide a shorthand for if-else statements, allowing for concise conditional expressions that evaluate one of two values based on a boolean condition. Understanding these operators is crucial for effective programming, as they form the basis for many operations and decision-making processes in code.

Precedence and associativity of operators

In programming, operator precedence and associativity determine the order in which operators are evaluated in expressions. Understanding these concepts is essential for writing clear and predictable code. Below is a detailed explanation of operator precedence and associativity, particularly in the context of C and similar languages.

Operator precedence

Operator precedence defines the order in which different operators in an expression are evaluated. Operators with higher precedence are evaluated before operators with lower precedence. For example, in the expression 2 + 3 \* 4, the multiplication operator (\*) has higher precedence than the addition operator (+), so the expression is evaluated as 2 + (3 \* 4).

Operator precedence table

Here is a table of common operators in C, sorted by precedence from highest to lowest (*Table 3.8*):

|  |  |  |
| --- | --- | --- |
| **Precedence** | **Operator** | **Description** |
| 1 | () | Parentheses (used to change precedence) |
| 2 | ++, -- | Postfix increment/decrement |
| 3 | ++, -- | Prefix increment/decrement |
| 4 | +, - | Unary plus/minus |
| 5 | \*, /, % | Multiplication, Division, Modulus |
| 6 | +, - | Addition, Subtraction |
| 7 | <<, >> | Left shift, Right shift |
| 8 | <, <=, >, >= | Relational operators |
| 9 | ==, != | Equality operators |
| 10 | & | Bitwise AND |
| 11 | ^ | Bitwise XOR |
| 12 | ` | ` |
| 13 | && | Logical AND |
| 14 | ` |  |
| 15 | ?: | Ternary conditional operator |
| 16 | = | Assignment operator |
| 17 | +=, -=, \*=, /=, %= | Compound assignment operators |
| 18 | , | Comma operator |

Associativity

Associativity determines the order in which operators of the same precedence level are evaluated. It specifies whether an expression is evaluated from left to right or right to left.

* **Left-to-right associativity**: Most operators (e.g., +, -, \*, &, |) are evaluated from left to right. Example:
* **Right-to-left associativity:** Some operators, such as the assignment operator (=) and the ternary operator (?:), are evaluated from right to left.

Example:

Understanding operator precedence and associativity helps to avoid ambiguity in expressions and ensures that calculations yield the expected results. It is essential to use parentheses effectively when needed to clarify the intended order of operations.

Unformatted and formatted I/O function in C

In C programming, input and output (I/O) functions are essential for interacting with users and handling data. There are two primary types of I/O in C: formatted I/O and unformatted I/O. Each type serves different purposes and uses different functions.

Formatted I/O

Formatted I/O functions in C, primarily printf for output and scanf for input, enable programmers to read and write data with specified formats, allowing for precise control over how data is presented and interpreted. The printf function formats data types as strings according to defined format specifiers (like %d for integers and %f for floating-point numbers) to produce human-readable output on the console. Conversely, scanf reads input from the user and stores it in specified variables, also using format specifiers to correctly interpret the input data types. This capability is essential for effective user interaction and data handling in C programming.

printf()

Used to output formatted text to the standard output (usually the console). Syntax:

**Parameters**:

**Format***:* A format string that specifies how to format the output. It can contain format specifiers (e.g., %d, %f, %s) that determine how subsequent arguments are displayed. Additional arguments will be formatted according to the format string.

Example of printf():

Output:

scanf()

Used to read formatted input from the standard input (usually the keyboard). Syntax:

Parameters:

A format string that specifies the expected input format. It contains format specifiers that indicate the type of data to read (e.g., %d for integers, %f for floats, %s for strings).

Pointers to variables where the input data will be stored.

Example of scanf():

Output:

Unformatted I/O

Unformatted I/O functions provide a way to read and write raw binary data without any formatting. These functions are useful for handling data files and performing operations that require precise control over the data format.

getc() and putc()

The definitions are:

* **getc():** Reads a single character from a file (or standard input).
* **putc():** Writes a single character to a file (or standard output).

Example of getc() and putc():

#include <stdio.h> int main() {

FILE \*file; char ch;

// Opening a file for writing

file = fopen("example.txt", "w");

Output:

fread() and fwrite()

The definitions are:

* **fread():** Reads a block of data from a file.
* **fwrite():** Writes a block of data to a file.

Example of fread() and fwrite():

#include <stdio.h>

Conclusion

In C programming, operators are crucial for performing calculations and controlling logic. Arithmetic operators (+, -, \*, /, %) handle basic math operations, while relational operators (==, !=, <, >, <=, >=) compare values. Logical operators (&&, ||, !) combine Boolean expressions, and bitwise operators (&, |, ^, ~, <<, >>) work directly on binary data. Unary operators (like ++, --, -, +) affect a single operand, and assignment operators (=, +=, -=, etc.) assign or modify variable values. Conditional (ternary) operators (? :) allow quick decision-making. Operator precedence and associativity rules govern the order in which expressions are evaluated, ensuring consistent results. For input and output, unformatted I/O functions like getchar() and putchar() deal with single characters, while formatted functions like printf() and scanf() handle structured data, making data input and output efficient and precise. The next chapter will focus on control flow structures in C programming, which are essential for directing the execution of a program based on certain conditions. It will cover the if statement for conditional branching, allowing decisions to be made within the program. The chapter will also introduce the switch statement, which provides a more efficient way to handle multiple conditions. Repetition constructs like loops, including for, while, and do-while, will be discussed, enabling the repetition of code blocks. The break and continue statements, used to control the flow of loops, will be explained. Finally, the chapter will touch on the goto statement, which allows for an unconditional jump within the program. These control structures provide flexibility and enable complex logic to be implemented in C programs.

Exercises

* Write a program that takes two numbers as input and performs all arithmetic operations on them, displaying the results.
* Create a program that compares two integers using relational operators and prints whether each comparison (like a < b, a == b) is true or false.
* Write a program that takes two Boolean variables and uses logical operators to display the results of AND, OR, and NOT operations.
* Using bitwise operators, create a program that swaps two numbers without using a temporary variable.
* Demonstrate the effect of prefix and postfix increments and decrements on an integer in a program.
* Write a program that demonstrates the use of compound assignment operators (+=, -=, \*=, /=) by modifying an initial variable’s value.
* Write a program that uses the conditional operator (? :) to determine the largest of two numbers.
* Operator Precedence and Associativity: Explain the output of the expression 5 + 3 \* 2 / 1 - 4 in a program, illustrating the impact of precedence and associativity.
* Use printf() and scanf() to take a user’s name, age, and height, then display them in a formatted sentence.
* Write a program that reads single characters using getchar() and prints them one by one until a specific character (e.g., ‘#’) is encountered.