Introduction

1. Overview

- **Purpose of Operators in C**
  Operators are essential building blocks in C that let you perform computations, comparisons, logical checks, and control data flow without writing full-fledged functions. They make code concise and intuitive, ranging from arithmetic to conditional operations.

2. Why Operators Matter

- They streamline operations like **addition, subtraction, and assignment**.
- Enable **conditional branching** and **decision-making** (e.g., `if`, `?:`).
- Enhance readability and efficiency, avoiding verbose code.

3. Types of Operators (Common in C)

These are generally covered—below are the main ones you'd expect to see and should consider including in your PDF:

Operator Category	Examples	Purpose
Arithmetic	`+`, `-`, `*`, `/`, `%`	Basic mathematical operations
Relational	`==`, `!=`, `<`, `>`, `<=`, `>=`	Compare values, yield true/false (1/0)
Logical	`&&`, `||`, `!`	Evaluate boolean logic
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	Assign or update variable values

Increment / Decrement	`++`, `--`	Increase or decrease values by one
Bitwise	`&`, `|`, `^`, `~`, `<<`, `>>``	For bit manipulation
Ternary (Conditional)	`?:`	Compact conditional expression `(cond) ? expr1 : expr2`
Comma	`,`	Separates multiple expressions in one statement
Address & Dereference	`&`, `*`	Work with pointers (get address or value at address)

Explanation of Key Operators (Concise)

Arithmetic Operators

- `+`, `-`, `*`, `/`, `%` — perform addition, subtraction, multiplication, division, and modulus.

Relational Operators

- `==`, `!=`, `<`, `>`, `<=`, `>=` — evaluate relationships between values, returning 1 or 0.

Logical Operators

- `&&` (logical AND), `||` (logical OR), `!` (logical NOT) — evaluate boolean logic, often used in `if` statements.

Assignment Operators

- `=` — simple assignment.
- `+=, -=, *=, /=, %=` — combine arithmetic and assignment for brevity and clarity.

Increment / Decrement

- `++`, `--` — modify a variable's value by one (prefix or postfix influencing evaluation timing).

Bitwise Operators

- `&`, `|`, `^`, `~`, `<<`, `>>` — operate at the level of bits, used in masks, performance-critical code, and embedded programming.

Ternary Conditional

- `?:` — shorthand for `if-else`, e.g. `x > 0 ? x : -x`.

Comma Operator

- `,` — allows multiple expressions in a single statement (rarely used, but useful in loops).

Address-of & Dereference

- `&` — gets the memory address of a variable.
- `*` — access or modify the value at a given address (pointer dereference).

Logical Operators in Breif

1. Types of Logical Operators

- **AND (`&&`)**
    - Returns `true (1)` if **both conditions** are true.

Example:

```
if (x > 0 && y > 0)  // true only if both x and y are positive
```

- **OR (`||`)**
    - Returns `true (1)` if **at least one condition** is true.

Example:

```
if (x > 0 || y > 0)  // true if either x or y is positive
```

- **NOT (`!`)**
    - Inverts the condition.

Example:

```
if (!(x > 0))  // true if x is not greater than 0
```

---

2. Concept of Short-Circuit Evaluation

- In **`&&` (AND)**:
    - If the first condition is **false**, second condition is **not evaluated**.
    - Saves time and prevents unnecessary checks.
- In `||` **(OR)**:

    - If the first condition is **true**, second condition is **not evaluated**.

- This is called **short-circuiting** because evaluation stops early.

Bitwise Operator in Brief

1. What Are Bitwise Operators?

- Bitwise operators handle operations directly on the binary representation of integer operands.

- This allows operations at the bit level, enabling more granular control and performance, particularly useful in system-level programming.

2. Types of Bitwise Operators Covered

- **Bitwise AND (`&`)**

    - Performs a bit-by-bit AND operation.

    - Example: `5 & 3` → binary `0101 & 0011` = `0001` (result 1).

- **Bitwise OR (`|`)**

    - Performs a bit-by-bit OR operation.

    - Example: `5 | 3` → `0101 | 0011` = `0111` (result 7).

- **Bitwise XOR (`^`)**

    - Performs exclusive OR: bit is `1` if exactly one operand's bit is `1`.

    - Example: `5 ^ 3` → `0101 ^ 0011` = `0110` (result 6).

- **Bitwise NOT (`~`)**

    - Unary operator that inverts all bits of its operand.

    - Example: `~5` → if using 8-bit representation, `~00000101` = `11111010`.

3. Code Examples (Likely Coverage)

AND Example:

```
int a = 5; // 0101

int b = 3; // 0011

int c = a & b; // 0001 → 1
```

OR Example:

```
int d = a | b; // 0111 → 7
```

XOR Example:

```
int e = a ^ b; // 0110 → 6
```

NOT Example:

```
int f = ~a; // bitwise NOT of 0101 → 1010 (in unsigned context)
```

1. Left Shift (`<<`)

- Shifts all bits of the operand to the **left** by the specified number of positions.
- Vacated rightmost bits are filled with zeros.
- Effectively multiplies the number by 2n2^n2n when shifting by *n* positions.
  For instance, `a << 1` doubles `a`; `a << 2` multiplies it by 4.

Code Example

```
unsigned char a = 21;  // binary: 00010101
// a << 1 = 42 (00101010)

printf("a << 1 = %d\n", (a << 1));
```

Deep Dive: Bitwise Right Shift (`>>`) in C

What It Does

- The `>>` operator shifts all bits of an operand **to the right** by a specified number of positions.

- Vacated bits on the left are filled depending on the data type (unsigned vs signed) and implementation details.

Arithmetic Effect

- For **unsigned types** or non-negative **signed types**, right shift acts like **integer division by powers of two**.
  e.g., `x >> 1` approximates `x / 2`.

Implementation-Defined Behavior with Signed Types

- When applied to **negative signed values**, the behavior is **implementation-defined**: some compilers perform **arithmetic shift** (preserving the sign bit), while others might differ. It's not strictly undefined, but results can vary across systems.

Risk of Undefined Behavior

- Shifting by a number **greater than or equal to the bit-width** of the type is **undefined behavior**. For instance, shifting a 32-bit integer by 32 or more is invalid.

Activities: Create a simple calculator program that takes user input and performs a calculation.

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Enter first integer: ");
    scanf("%d", &a);

    printf("Enter second integer: ");
    scanf("%d", &b);

    printf("Addition: %d\n", a + b);
    printf("Subtraction: %d\n", a - b);
    printf("Multiplication: %d\n", a * b);


    return 0;
}
```