Unit II

Operators and Expressions

in C Programming

A comprehensive presentation covering types of operators, expressions, operator precedence, associativity, and mathematical functions in C.

```
int result = (a + b) * c / (d - e); // Expression example
```

Table of Contents

1. Introduction to Operators
2. Arithmetic Operators
3. Relational Operators
4. Logical Operators
5. Assignment Operators
6. Increment and Decrement Operators
7. Conditional Operators
8. Bitwise Operators
9. Special Operators
10. Arithmetic Expressions & Evaluation
11. Operator Precedence & Associativity
12. Mathematical Functions
13. Summary & Conclusion

Introduction to Operators in C

Operators are special symbols in C used to perform operations on operands. **Operands** are data items on which operators act.

Classification Based on Operands

- **Unary Operators:** Work on single operand (e.g., ++ increment)
- **Binary Operators:** Work on two operands (e.g., + addition)
- **Ternary Operators:** Work on three operands (e.g., ?: conditional)

Types of Operators in C

- Arithmetic Operators
- Logical Operators
- Increment/Decrement
- Bitwise Operators
- Relational Operators
- Assignment Operators
- Conditional Operators
- Special Operators

Example:

```
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    int sum = a + b;    // + is an operator
                        // a and b are operands
    printf("Sum: %d\n", sum);
    return 0;
}
```

Output: **Sum: 15**

Common Operators:

`+` `-` `*` `/` `%` `++` `--` `<` `>` `==` `!=` `&&` `||`

Key Takeaway:
Operators are the basic building blocks that allow you to perform operations and manipulate data in C programs. Understanding operators is essential for writing efficient and effective C code.

Arithmetic Operators in C

Arithmetic operators perform mathematical calculations on numeric operands in C programs.

Basic Arithmetic Operators

+ (Addition): Adds two operands
Example: `a + b`

- (Subtraction): Subtracts right operand from left operand
Example: `a - b`

*** (Multiplication)**: Multiplies two operands
Example: `a * b`

/ (Division): Divides left operand by right operand
Example: `a / b` (integer division if both are int)

% (Modulus): Returns remainder after division
Example: `a % b` (works only with integers)

Unary Arithmetic Operators

+ (Unary Plus): Indicates positive value (rarely used)
Example: `+a`

- (Unary Minus): Negates an expression
Example: `-a`

Increment/Decrement Operators

++ (Increment): Increases value by 1
Prefix: `++a` (increment first, then use)
Postfix: `a++` (use first, then increment)

-- (Decrement): Decreases value by 1
Prefix: `--a` (decrement first, then use)
Postfix: `a--` (use first, then decrement)

Example Program:

```c
#include <stdio.h>
int main() {
    int a = 25, b = 5;

    printf("a + b = %d\n", a + b);   // 30
    printf("a - b = %d\n", a - b);   // 20
    printf("a * b = %d\n", a * b);   // 125
    printf("a / b = %d\n", a / b);   // 5
    printf("a % b = %d\n", a % b);   // 0

    printf("+a = %d\n", +a);         // 25
    printf("-a = %d\n", -a);         // -25

    int c = a++;
    printf("c = %d, a = %d\n", c, a); // c=25, a=26

    int d = --b;
    printf("d = %d, b = %d\n", d, b); // d=4, b=4

    return 0;
}
```

Important Notes:

- Division between integers results in an integer (truncated)
- Modulus operator only works with integer operands
- Pre/post increment/decrement behavior differs in expressions
- Be careful with increment/decrement in complex expressions

Key Takeaway:
Arithmetic operators in C provide the foundation for mathematical calculations. Understanding the nuances of division, modulus, and increment/decrement operators is crucial for writing correct and efficient code.

Relational Operators in C

Relational operators compare two values and determine the relationship between them. They return boolean values: **1 (true)** or **0 (false)**.

Operator	Description	Example	Result
<	Less than	5 < 10	1 (true)
>	Greater than	5 > 10	0 (false)
<=	Less than or equal to	5 <= 5	1 (true)
>=	Greater than or equal to	5 >= 10	0 (false)
==	Equal to	5 == 5	1 (true)
!=	Not equal to	5 != 5	0 (false)

Key Uses:

- Control flow with if, else if, while statements
- Conditional expressions using ternary operator
- Forming complex conditions with logical operators

Example:

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;

    // Using relational operators
    printf("a < b  : %d\n", a < b);
    printf("a > b  : %d\n", a > b);
    printf("a <= b : %d\n", a <= b);
    printf("a >= b : %d\n", a >= b);
    printf("a == b : %d\n", a == b);
    printf("a != b : %d\n", a != b);

    return 0;
}
```

Output:

```
a < b  : 0 (false)
a > b  : 1 (true)
a ⩽ b : 0 (false)
a ⩾ b : 1 (true)
a ═ b : 0 (false)
a ≠ b : 1 (true)
```

Common Mistakes:

= vs ==: Using assignment operator (=) instead of equality operator (==) in conditions

Floating Point Comparisons: Direct comparison of floating point values can be unreliable

Chaining Comparisons: `if(a < b < c)` doesn't work as expected in C

Key Takeaway:
Relational operators determine relationships between values and produce boolean results (1 or 0) that are fundamental for decision-making in C programs. They are essential for implementing program logic and flow control.

Logical Operators in C

Logical operators are used to perform logical operations on boolean expressions. They evaluate conditions and return either true (1) or false (0).

The Three Logical Operators in C

`&&` **Logical AND** - Returns true only if both operands are true

`||` **Logical OR** - Returns true if at least one operand is true

`!` **Logical NOT** - Returns the opposite of the operand's value

Truth Tables

Logical AND (&&)

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Logical OR (||)

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Example: Using Logical Operators

```c
#include <stdio.h>
int main() {
    int age = 25;
    int salary = 50000;

    // Logical AND example
    if (age > 18 && salary > 30000) {
        printf("Eligible for loan\n");
    }

    // Logical OR example
    if (age < 18 || salary < 20000) {
        printf("Not eligible for premium\n");
    }

    // Logical NOT example
    if (!(age < 18)) {
        printf("Adult\n");
    }

    return 0;
}
```

Output: **Eligible for loan**
Adult

Logical NOT (!)

A	!A
0	1
1	0

Important Notes:

- C treats any non-zero value as TRUE and zero as FALSE
- Logical operators always return either 1 (TRUE) or 0 (FALSE)
- Short-circuit evaluation: If the first operand of && is false, the second isn't evaluated
- Similarly, if the first operand of || is true, the second isn't evaluated

Key Takeaway:
Logical operators are essential for making decisions in programs based on multiple conditions. They allow you to combine or negate boolean expressions to create complex conditions for control flow statements like if, while, and for loops.

Assignment Operators in C

Assignment operators are used to assign values to variables. The basic assignment operator is =, which assigns the value on the right to the variable on the left.

Compound Assignment Operators

C provides shorthand operators that combine an operation with assignment:

Operator	Description	Equivalent to
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

Bitwise Assignment Operators

- &= Bitwise AND
- |= Bitwise OR
- ^= Bitwise XOR
- <<= Left shift
- >= Right shift

Compound Assignment Example:

```
#include <stdio.h>
int main() {
    int a = 10;

    a += 5;  // a = a + 5
    printf("a += 5: %d\n", a);

    a -= 3;  // a = a - 3
    printf("a -= 3: %d\n", a);

    a *= 2;  // a = a * 2
    printf("a *= 2: %d\n", a);

    return 0;
}
```

Output:

```
a += 5: 15
a -= 3: 12
a *= 2: 24
```

Bitwise Assignment Example:

```
int x = 5;  // 101 in binary
x &= 3;     // x = x & 3 (101 & 011)
// Result: x = 1 (001 in binary)
```

Key Takeaway:
Compound assignment operators provide a more concise way to write common assignment operations. They combine the operation and assignment into a single operator, making code shorter and often more readable.

Increment and Decrement Operators

Increment `++` and **decrement** `--` operators increase or decrease a variable's value by 1.

Two Forms

- **Prefix form**: `++a` or `--a`

  Increments/decrements the variable first
  Then returns the new value

- **Postfix form**: `a++` or `a--`

  Returns the current value first
  Then increments/decrements the variable

Key Points

- Can only be applied to variables, not constants or expressions
- Commonly used in loops and counters
- The choice between prefix and postfix matters when the value is used in an expression

Example: Prefix vs Postfix

```
#include <stdio.h>
int main() {
    int a = 5, b = 5;
    int pre, post;

    // Prefix increment
    pre = ++a;    // a is incremented to 6,
                  // then assigned to pre

    // Postfix increment
    post = b++;   // b (5) is assigned to post,
                  // then b is incremented to 6

    printf("pre = %d, a = %d\n", pre, a);
    printf("post = %d, b = %d\n", post, b);
    return 0;
}
```

Output:
pre = 6, a = 6
post = 5, b = 6

Prefix vs Postfix Behavior:

Operation	Prefix (++a)	Postfix (a++)
Step 1	Increment a	Return current value of a
Step 2	Return new value of a	Increment a
When used alone	Same effect as postfix	Same effect as prefix
Efficiency	Slightly more efficient	Creates a temporary copy

Key Takeaway:
The difference between prefix and postfix operators is critical when their values are used in expressions. Choose prefix form for slightly better performance when the return value is not needed.

Conditional Operators in C

Conditional Operator (also called **Ternary Operator**) is the only ternary operator in C. It provides a shorthand way to write simple if-else statements.

Syntax

```
condition ? expression1 : expression2;
```

- If **condition** is true (non-zero), expression1 is evaluated
- If **condition** is false (zero), expression2 is evaluated
- Only one expression is evaluated based on the condition

Equivalent If-Else Statement

Ternary operator:

```
result = (a > b) ? a : b;
```

Equivalent if-else:

```
if (a > b)
    result = a;
else
    result = b;
```

Best Practices

- Use for simple conditional assignments
- Avoid nested ternary operators (reduces readability)
- Use parentheses for clarity when needed

Example 1: Finding Maximum

```
#include <stdio.h>
int main() {
    int a = 5, b = 10;

    // Find maximum using ternary
    int max = (a > b) ? a : b;

    printf("Maximum: %d\n", max);
    return 0;
}
```

Output: **Maximum: 10**

Example 2: Even or Odd

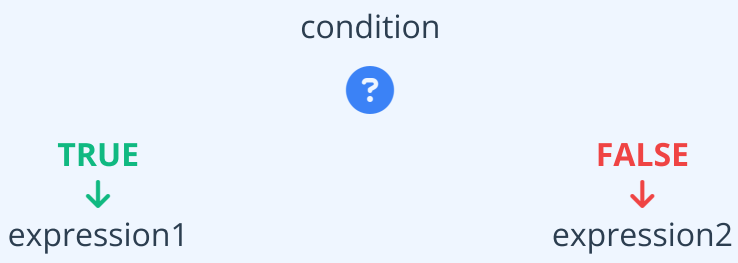
```
#include <stdio.h>
int main() {
    int num = 7;

    printf("%d is %s\n", num,
        (num % 2 == 0) ? "even" : "odd");

    return 0;
}
```

Output: **7 is odd**

Operator Flow:



Key Takeaway:
The conditional (ternary) operator provides a concise way to write simple conditional expressions. It's particularly useful for simple if-else scenarios, making code more compact and readable when used appropriately.

Bitwise Operators in C

Bitwise operators perform operations on individual bits of integer values. They operate at the binary level and are used for tasks like flags, masks, and hardware interaction.

- `&` **Bitwise AND**: Sets each bit to 1 if both bits are 1
- `|` **Bitwise OR**: Sets each bit to 1 if any of the two bits is 1
- `^` **Bitwise XOR**: Sets each bit to 1 if only one of the two bits is 1
- `~` **Bitwise NOT**: Inverts all the bits (0 becomes 1, 1 becomes 0)
- `<<` **Left Shift**: Shifts bits to the left, fills with 0s on right
- `>>` **Right Shift**: Shifts bits to the right, fills based on sign bit

Common Applications:

- Flag manipulation in system programming
- Efficient multiplication/division by powers of 2
- Hardware register manipulation
- Cryptography and hash functions
- Data compression algorithms

Binary Operation Example:

Let's see how bitwise AND works with numbers 12 and 10:

```
12        0000 1100
10        0000 1010
-------------------
12 & 10   0000 1000  = 8
```

```c
#include <stdio.h>
int main() {
    unsigned int a = 12;  // 00001100
    unsigned int b = 10;  // 00001010

    printf("a & b = %d\n", a & b);   // 8
    printf("a | b = %d\n", a | b);   // 14
    printf("a ^ b = %d\n", a ^ b);   // 6
    printf("~a = %d\n", ~a);         // -13
    printf("a << 1 = %d\n", a << 1); // 24
    printf("a >> 1 = %d\n", a >> 1); // 6

    return 0;
}
```

Optimization Tip:

Shifting left by n bits is equivalent to multiplying by 2^n. Shifting right by n bits is equivalent to dividing by 2^n (for unsigned integers).

Key Takeaway:

Bitwise operators provide low-level bit manipulation capabilities critical for system programming, optimization, and hardware interaction. Understanding binary representation is essential for effectively using these operators.

Special Operators in C

C provides several special operators that perform unique operations beyond basic arithmetic and logic.

sizeof Operator

Computes the size in bytes of a variable or data type at compile time.

```
sizeof(int); // Returns size of integer type
```

Comma Operator (,)

Evaluates multiple expressions in sequence, returning the value of the rightmost expression.

```
for(i=0, j=10; i < j; i++, j--) { ... }
```

Cast Operator

Converts one data type to another explicitly.

```
(float)10; // Converts integer to float
```

Address & Dereference Operators

& (Address-of): Returns memory address of a variable

***** (Dereference): Accesses value at a pointer's address

Member Access Operators

. (Dot): Accesses members of structures or unions

-> (Arrow): Accesses members via a pointer

Example Code:

```
#include <stdio.h>

int main() {
    // sizeof operator
    int num = 10;
    printf("Size: %lu bytes\n",
           sizeof(num));

    // Cast operator
    float result = (float)num / 3;

    // Address & dereference
    int *ptr = #
    *ptr = 20; // num is now 20

    return 0;
}
```

Structure Example:

```
struct Point {
    int x, y;
};

struct Point p1 = {10, 20};
struct Point *p_ptr = &p1

// Member access
p1.x = 15;       // Using dot
p_ptr->y = 25;   // Using arrow
```

Key Takeaway:
Special operators in C enhance the language's capabilities by allowing memory manipulation, type conversion, and complex structure access. The sizeof operator is particularly useful for portable code, as the size of data types can vary across different systems.

Arithmetic Expressions & Evaluation

Arithmetic expressions in C are combinations of variables, constants, and operators arranged to compute a value. The process of producing a value from an expression is called **evaluation**.

Components of Expressions

- **Operands:** Variables, constants, literals that are manipulated
- **Operators:** Symbols that specify operations to be performed
- **Parentheses:** Used to control the order of evaluation

Rules for Expression Evaluation

- **Operator precedence:** Higher precedence operators are evaluated first
- **Associativity:** Determines order of evaluation when operators have same precedence
- **Type conversion:** Operands may be converted to compatible types

Important Note:
The order of evaluation of operands is not guaranteed in C. Don't rely on it for expressions with side effects.

Expression Evaluation Example:

```
int result = 2 + 3 * 4 - 6 / 2;
```

Step-by-step evaluation:

1. `3 * 4` = **12** (multiplication first)
2. `6 / 2` = **3** (division next)
3. `2 + 12` = **14** (addition next)
4. `14 - 3` = **11** (subtraction last)

Final result: **11**

Parentheses Change Order:

```
int result = (2 + 3) * (4 - 6) / 2;
```

1. `(2 + 3)` = **5**
2. `(4 - 6)` = **-2**
3. `5 * -2` = **-10**
4. `-10 / 2` = **-5**

Final result: **-5**

Key Takeaway:
Understanding the order of evaluation in C expressions is crucial for writing correct code. Always use parentheses when in doubt to make the intended evaluation order explicit and improve code readability.

Operator Precedence & Associativity in C

Operator Precedence determines the order in which operators are evaluated in an expression with multiple operators. **Associativity** determines the order of evaluation when operators of the same precedence appear in an expression (left-to-right or right-to-left).

Precedence	Operator	Description	Associativity
1 (Highest)	() [] . ->	Parentheses, array subscript, member access	Left to right
2	! ~ ++ -- + - * & (type) sizeof	Unary operators, type cast, sizeof	Right to left
3	* / %	Multiplication, division, modulus	Left to right
4	+ -	Addition, subtraction	Left to right
5	<< >>	Bitwise shift left and right	Left to right
6	< <= > >=	Relational operators	Left to right
7	== !=	Equality operators	Left to right
8	&	Bitwise AND	Left to right
9	^	Bitwise XOR	Left to right
10	|	Bitwise OR	Left to right
11	&&	Logical AND	Left to right
12	||	Logical OR	Left to right
13	?:	Conditional operator (ternary)	Right to left
14	= += -= *= /= %= &= ^= |= <<= >>=	Assignment operators	Right to left
15 (Lowest)	,	Comma operator	Left to right

```
int result = 2 + 3 * 4;
// Evaluates to 14, not 20
// * has higher precedence than +
```

```
int x = 5, y = 10, z;
z = x++ + ++y;
// Evaluates to 16 (5 + 11)
// Postfix ++ has same precedence as prefix ++
// but different associativity
```

Common Precedence Rules to Remember:

- Parentheses override normal precedence
- Multiplication/division before addition/subtraction
- Arithmetic operations before relational operations
- Relational operations before logical operations
- Logical operations before assignments

Key Takeaway:
Understanding operator precedence and associativity is crucial for writing expressions that evaluate as intended. When in doubt, use parentheses to explicitly define the order of operations and improve code readability.

Mathematical Functions in C (math.h)

The math.h library provides a comprehensive set of functions for mathematical operations. To use these functions, include the header file: `#include <math.h>`

Common Math Functions

Basic Functions:

- **sqrt(x)** - Square root of x
- **pow(x, y)** - x raised to power y
- **fabs(x)** - Absolute value of x
- **exp(x)** - e raised to power x

Logarithmic Functions:

- **log(x)** - Natural logarithm (base-e)
- **log10(x)** - Base-10 logarithm

Trigonometric Functions:

- **sin(x)** - Sine of x (radians)
- **cos(x)** - Cosine of x (radians)
- **tan(x)** - Tangent of x (radians)
- **asin(x), acos(x), atan(x)** - Inverse trig

Rounding Functions:

- **ceil(x)** - Round up to nearest integer
- **floor(x)** - Round down to nearest integer
- **round(x)** - Round to nearest integer

Other Useful Functions:

- **fmod(x, y)** - Remainder of x/y (floating-point modulo)
- **modf(x, ∫_part)** - Splits x into integer and fractional parts
- **hypot(x, y)** - Hypotenuse of a right triangle ($\sqrt{(x^2+y^2)}$)

Example Code:

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 16.0;
    double y = 2.0;

    printf("sqrt(%.1f) = %.1f\n", x, sqrt(x));
    printf("pow(%.1f, %.1f) = %.1f\n", x, y, pow(
    printf("sin(0) = %.1f\n", sin(0));

    return 0;
}
```

Output:
sqrt(16.0) = 4.0
pow(16.0, 2.0) = 256.0
sin(0) = 0.0

Common Math Functions:

`sqrt()` `pow()` `sin()` `cos()` `log()` `log10()` `ceil()` `floor()` `fabs()` `fmod()`

Important Note:

- All math.h functions typically use **double** type for parameters and return values
- Link with math library using `-lm` flag when compiling (e.g., `gcc program.c -lm`)
- Check for domain errors when using functions with restricted input ranges

Key Takeaway:
The math.h library provides essential mathematical functions that simplify complex calculations in C programs. These functions improve code readability and efficiency by handling mathematical operations that would otherwise require custom implementations.

Summary & Conclusion

Key Points

- Operators are fundamental symbols for performing operations on data in C
- Different operator types (arithmetic, relational, logical, etc.) serve specific purposes
- Operator precedence and associativity determine expression evaluation order
- C's mathematical functions provide powerful tools for numerical computations
- Understanding operator behavior is essential for writing efficient C code

Operators Covered:

`+` `-` `*` `/` `%` `++` `--` `<` `>` `==` `!=`
`&&` `||` `!` `&` `|` `^` `~` `<<` `>>` `?:`

Next Steps

- Practice with various operators in code examples
- Solve expression evaluation problems
- Explore advanced mathematical functions
- Apply concepts in programming assignments

Conclusion

In this unit, you learned about different operators and expressions in C, how to evaluate them, the significance of operator precedence and associativity, and the utility of C's mathematical functions. Mastery of these is essential for effective C programming.

"Understanding operators and expressions is fundamental to mastering the C programming language."