CSE 1101: Structured Programming Language

Weeks 2-3: Operators in C

- Arithmetic Operators.
- Relational Operators.
- Logical Operators.
- Bitwise Operators.
- Assignment Operators.
- Other Operators.
- C Booleans
- Expression and Statement
- Type Conversion and Casting
- Operator Precedence

Lecture 10: Operators

4.1 Operators

Operators in C are symbols that perform specific operations on one or more operands (values or variables). C has several types of operators, including:

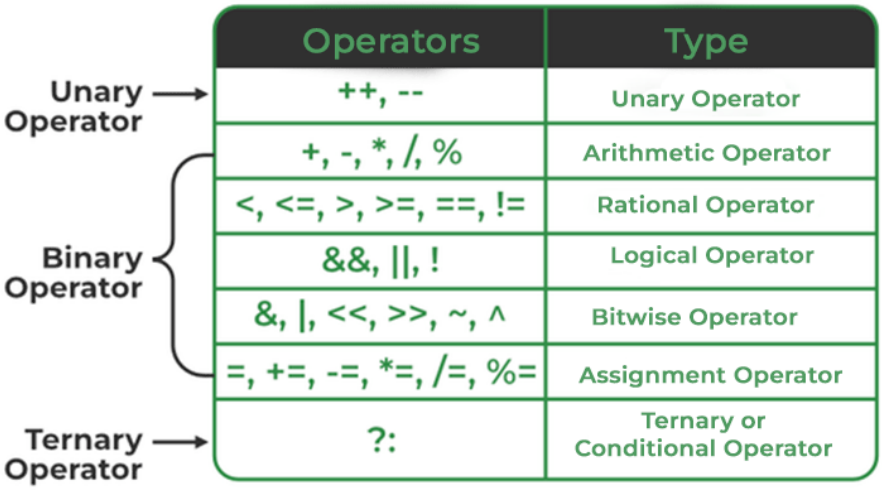
1. Arithmetic Operators: Used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.
2. Relational Operators: Used to compare values and determine relationships between them, such as equal to, less than, greater than, etc.
3. Logical Operators: Used to combine relational expressions and perform logical operations such as AND, OR, NOT, etc.
4. Assignment Operators: Used to assign values to variables, such as =, +=, -=, etc.
5. Bitwise Operators: Used to perform bit-level operations on values, such as AND, OR, NOT, etc.
6. Ternary Operators: Used to evaluate a conditional expression, such as (condition) ? x : y.
7. Sizeof Operators: Used to determine the size of a data type or a variable in bytes.
8. Comma Operators: Used to separate two or more expressions in a single statement.

Unary, Binary and Ternary Operators in C

Operators can also be classified into three types on the basis of the number of operands they work on:

- **Unary Operators:** Operators that work on single operand.
- **Binary Operators:** Operators that work on two operands.
- **Ternary Operators:** Operators that work on three operands.

Engr. Mohammad Mamun Hossain
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST
PhD (Pursuing) in CSE, RUET.
Associate Professor, Dept. of CSE, BAUST
Contact: mhossain@baust.edu.bd
linkedIn:mamunsust12
Resources: mamunsust12@GitHub
Emergency: 01717690847



Examples of each type of operator in C can be found in the following section:

4.1.1 Arithmetic Operators:

+ (Addition)
- (Subtraction)
* (Multiplication)
/ (Division)
% (Modulus, returns the remainder after division)

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Example:

```
int a = 10, b = 3, result;
result = a + b;   // result is 13
result = a - b;   // result is 7
result = a * b;   // result is 30
result = a / b;   // result is 3
result = a % b;   // result is 1
```

4.1.2 Relational Operators:

== (Equal to)
!= (Not equal to)
< (Less than)
> (Greater than)
<= (Less than or equal to)
>= (Greater than or equal to)

Operator	Name	Example	Description
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal
>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

Example:

```
int x = 5, y = 10;
if (x == y) {
   // code if x is equal to y
} else if (x < y) {
   // code if x is less than y
} else {
   // code if x is greater than y
}
```

4.1.3 Logical Operators:

&& (Logical AND)
|| (Logical OR)
! (Logical NOT)

Operator	Name	Example	Description
&&	Logical and	x < 5 && x < 10	Returns 1 if both statements are true
||	Logical or	x < 5 || x < 4	Returns 1 if one of the statements is true
!	Logical not	!(x < 5 && x < 10)	Reverse the result, returns 0 if the result is 1

Example:

```
int a = 1, b = 0;
if (a && b) {
   // code if both a and b are true
}
if (a || b) {
   // code if either a or b is true
}
if (!a) {
   // code if a is false
}
```

4.1.4 Assignment Operators:

= (Assignment)
+= (Add and assign)
-= (Subtract and assign)
*= (Multiply and assign)
/= (Divide and assign)
%= (Modulus and assign)

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Example:

```
int x = 5;
x += 3;  // equivalent to x = x + 3; (x is now 8)
x *= 2;  // equivalent to x = x * 2; (x is now 16)
```

4.1.5 Bitwise Operators:

& (Bitwise AND)
| (Bitwise OR)
^ (Bitwise XOR)
~ (Bitwise NOT)
<< (Left shift)
>> (Right shift)

Example:

```
unsigned int a = 5, b = 3, result;
result = a & b;  // Bitwise AND (result is 1)
result = a | b;  // Bitwise OR (result is 7)
result = a ^ b;  // Bitwise XOR (result is 6)
result = ~a;     // Bitwise NOT (result is 4294967290 on a 32-bit system)
result = a << 1;  // Left shift by 1 (result is 10)
result = a >> 1;  // Right shift by 1 (result is 2)
```

4.1.6 Other Operators

Others operator includes Increment and Decrement Operators, ternary, operators, sizeof operators and comma operators

4.1.6.1 Increment and Decrement Operators:

++ (Increment by 1)
-- (Decrement by 1)

Example:

```
int i = 5;
i++;  // equivalent to i = i + 1; (i is now 6)
i--;  // equivalent to i = i - 1; (i is now 5 again)
```

4.1.6.2 Ternary Operator (?:):

condition ? expression_if_true : expression_if_false

Example:

```
int x = 10, y = 20, result;
result = (x > y) ? x : y;  // If x is greater than y, result is x, otherwise result is y
```

4.1.6.3 sizeof Operator (?:):

sizeof(type) or sizeof(expression)

Example:

```
int integerVar;
size_t size = sizeof(int);  // Size of int data type in bytes
size_t varSize = sizeof(integerVar);  // Size of the variable integerVar in bytes
```

4.1.6.4 comma Operator (?:):

- The comma operator allows multiple expressions to be grouped together into a single expression. It evaluates each expression from left to right and returns the value of the rightmost expression.
- It is often used in places where multiple expressions are expected, such as in the initialization and increment parts of a for loop.

Example:

```
int a = 5, b = 10, c;
c = (a++, b++, a + b);  // Uses the comma operator to increment a and b, and assign the sum to c
```

Another common use is in the **for** loop:

```
for (int i = 0, j = 10; i < 5; i++, j--) {
   // Loop body
}
```

4.1.6.5 dot (.) and arrow (->) Operators

Member operators are used to reference individual members of classes, structures, and unions.
The dot operator is applied to the actual object.
The arrow operator is used with a pointer to an object.

Syntax

```
structure_variable . member;
and
structure_pointer -> member;
```

4.1.6.6 addressof (&) and Dereference (*) Operators

Pointer operator & returns the address of a variable.
For example &a; will give the actual address of the variable.
The pointer operator * is a pointer to a variable.
For example *var; will pointer to a variable var.
These are some of the fundamental operators in C. Keep in mind that the specific behavior of operators may vary depending on the data types involved.

4.1.7: Operator Precedence

4.1.7.1 Definition

Operator precedence determines the order in which operators are evaluated in an expression.

4.1.7.2 Precedence and Associativity Table

Precedence	Operators	Associativity
1 (Highest)	`()`, `[]`, `->`, `.`	Left to Right
2	`!`, `~`, `++`, `--`, `+` (Unary), `-` (Unary), `*` (Pointer)	Right to Left
3	`*`, `/`, `%`	Left to Right
4	`+`, `-`	Left to Right
5	`<<`, `>>`	Left to Right
...	...	...

4.1.7.3 Example Code

```c
#include <stdio.h>
int main() {
    int a = 5 + 2 * 3;
    printf("Result: %d\n", a); // Output: 11
    return 0;
}
```

4.1.7.4 BODMAS and PEMDAS Rule in C Programming

When evaluating mathematical expressions, **operator precedence** is essential. Two common mnemonics used to remember the order of operations are **BODMAS** and **PEMDAS**.

4.1.4.4.1 BODMAS Rule

BODMAS stands for:

- **B**rackets `()`
- **O**rders (Exponents: `^`, `sqrt()`)
- **D**ivision `/` and **M**ultiplication `*` (from left to right)
- **A**ddition `+` and **S**ubtraction `-` (from left to right)

```c
#include <stdio.h>
int main() {
    int result = 10 + 5 * 2;
    printf("Result: %d\n", result); // Output: 20 (Multiplication happens before addition)

    int result2 = (10 + 5) * 2;
    printf("Result: %d\n", result2); // Output: 30 (Brackets change precedence)

    return 0;
}
```

4.1.7.4.2 PEMDAS Rule

PEMDAS is another mnemonic used mostly in the U.S., which stands for:

- **P**arentheses `()`
- **E**xponents `^`, `sqrt()`
- **MD** Multiplication `*` and Division `/` (from left to right)
- **AS** Addition + and Subtraction – (from left to right)

Since Multiplication and Division have the same precedence, they are evaluated from **left to right**.

```c
#include <stdio.h>
#include <math.h> // For power function

int main() {
    double result = pow(2, 3) + 10 / 2;
    printf("Result: %.2f\n", result); // Output: 12.00 (Exponent first, then division, then add

    double result2 = 10 - 2 + 3;
    printf("Result: %.2f\n", result2); // Output: 11.00 (Left to right order for addition/subtr

    return 0;
```

Key Differences Between BODMAS and PEMDAS

Rule	Meaning	Order
BODMAS	Used in many countries	Brackets, Orders, Division & Multiplication, Addition & Subtraction
PEMDAS	Common in the U.S.	Parentheses, Exponents, Multiplication & Division, Addition & Subtraction

The order of precedence is the same in both, just different terminology.

Lecture 11: Booleans, Expression and Statements

4.2 C Booleans

C does not have a built-in boolean data type like some other programming languages such as C++, Java, or Python. However, the <stdbool.h> header was introduced in the C99 standard to provide a boolean type and the bool keyword. Here's a brief overview:

Very often, in programming, you will need a data type that can only have one of two values, like:

YES / NO, ON / OFF, TRUE / FALSE

For this, C has a bool data type, which is known as booleans.

Booleans represent values that are either true or false.

Boolean Variables : A boolean variable is declared with the bool keyword and can only take the values true or false:

```
bool isProgrammingFun = true;
bool isFishTasty = false;
```

Before trying to print the boolean variables, you should know that boolean values are returned as integers:

1 (or any other number that is not 0) represents true

0 represents false

Therefore, you must use the %d format specifier to print a boolean value:

```
// Create boolean variables
bool isProgrammingFun = true;
bool isFishTasty = false;

// Return boolean values
printf("%d", isProgrammingFun);   // Returns 1 (true)
printf("%d", isFishTasty);        // Returns 0 (false)
```

In the example below, we use the equal to (==) operator to compare different values:

```
printf("%d", 10 == 10); // Returns 1 (true), because 10 is equal to 10
printf("%d", 10 == 15); // Returns 0 (false), because 10 is not equal to 15
```

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote. In the example below, we use the >= comparison operator to find out if the age (25) is greater than OR equal to the voting age limit, which is set to 18:

```
int myAge = 25;
int votingAge = 18;
```

```
if (myAge >= votingAge) {
  printf("Old enough to vote!");
} else {
  printf("Not old enough to vote.");
}
```

4.3 Expression

An expression in C is a combination of values, variables, operators, and functions that can be evaluated to produce a single value. Expressions can be used in various places in C, such as in assignments, function calls, and control structures.

Examples of expressions in C are:

```
// Literals
10 + 20

// Variables
x + y

// Operators
a * b + c / d

// Functions
printf("Hello, world!\n");
```

Expressions can be evaluated using the following rules:

1. Parentheses are evaluated first.
2. Unary operators (e.g., ++, --, &, *, etc.) are evaluated next.
3. Multiplicative operators (e.g., *, /, %) are evaluated next.
4. Additive operators (e.g., +, -) are evaluated next.
5. Relational operators (e.g., <, >, <=, >=, ==, !=) are evaluated next.
6. Logical operators (e.g., &&, ||) are evaluated next.
7. Ternary operator (?:) is evaluated last.

For example, the following expression:

a + b * c - d / e is evaluated as: ***((a + (b * c)) - (d / e))***

4.4 Statements

A statement in C is a single command or instruction that specifies an action to be taken by the program. Statements are the basic building blocks of a C program, and they can be simple or complex.

Examples of statements in C are:

```
// Assignment statement
int x = 10;
```

```
// Function call statement
printf("Hello, world!\n");

// Conditional statement
if (x > 0) {
   printf("x is positive\n");
} else {
   printf("x is not positive\n");
}

// Loop statement
for (int i = 0; i < 10; i++) {
   printf("%d\n", i);
}
```

Statements in C can be executed in a sequence, conditionally, or repeatedly. They can be grouped together to form a block, which can be used to control the flow of the program. A block is a collection of statements surrounded by curly braces {}.

Lecture 12: Type Conversion and Casting in C

Type conversion and casting are important concepts in C when dealing with different data types. Type conversion refers to the process of converting a value from one data type to another, while casting is a specific syntax used to perform such conversions explicitly.

There are two types of type conversion in C: implicit conversion (also known as coercion) and explicit conversion (also known as casting).

4.5 Implicit Conversion (Coercion):

C allows certain automatic conversions between compatible data types. This happens implicitly without the need for explicit casting.
For example, when assigning a value of a smaller data type to a variable of a larger data type, the conversion is done implicitly.

Example:

```
int numInt = 10;
double numDouble = numInt;  // Implicit conversion from int to double
```

4.6 Explicit Conversion (Casting):

Explicit conversion is done using casting operators to convert a value from one data type to another.

There are two types of casting in C:

C-style casting: This involves using a specific syntax (type) to perform the conversion.
Type conversion functions: Certain functions like atoi, atof, itoa, etc., are used for explicit type conversion in specific scenarios.

Example (C-style casting):

```
double numDouble = 10.5;
int numInt = (int)numDouble;  // Explicit conversion from double to int
```

Example (Type conversion functions):

```
char strNum[] = "123";
int num = atoi(strNum);  // Converts a string to an integer
```

Type Conversion Functions:

C provides a set of standard library functions for explicit type conversion between different data types. Some commonly used functions include:

atoi(): Converts a string to an integer.

Engr. Mohammad Mamun Hossain
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST
PhD (Pursuing) in CSE, RUET.
Associate Professor, Dept. of CSE, BAUST
Contact: mhossain@baust.edu.bd
linkedIn:mamunsust12
Resources: mamunsust12@GitHub
Emergency: 01717690847

atof(): Converts a string to a double.
itoa(): Converts an integer to a string.

Example:

```
char strNum[] = "456";
int num = atoi(strNum);  // Converts the string "456" to an integer 456
```

4.7 Sizeof Operator:

The sizeof operator can be used to determine the size (in bytes) of a data type or a variable. It's often used to ensure that a type conversion won't result in data loss due to truncation.
Example:

```
int numInt = 10;
double numDouble = 20.5;
float numFloat = (float)numDouble;  // Explicit conversion using casting
size_t intSize = sizeof(int);      // Size of int in bytes
```

Note: It's important to be cautious with explicit type conversion, especially when it may result in loss of data or unexpected behavior. Always ensure that the conversion is appropriate for the data being manipulated.