

GXEST204

PROGRAMMING IN C

# Course Info

<b>Course Code</b>	<b>GXEST204</b>	<b>CIE Marks</b>	40
<b>Teaching Hours/Week(L: T:P: R)</b>	3:0:2:0	<b>ESE Marks</b>	60
<b>Credits</b>	4	<b>Exam Hours</b>	2 Hrs. 30 Min.
<b>Prerequisites (if any)</b>	None	<b>Course Type</b>	Theory

# Course Assessment Method

## **Course Assessment Method (CIE: 40 marks, ESE: 60 marks)**

### **Continuous Internal Evaluation Marks (CIE):**

<b>Attendance</b>	<b>Assignment/ Microproject</b>	<b>Internal Examination-1 (Written)</b>	<b>Internal Examination- 2 (Written)</b>	<b>Total</b>
<b>5</b>	<b>15</b>	<b>10</b>	<b>10</b>	<b>40</b>


# End Semester Examination

## **End Semester Examination Marks (ESE)**

*In Part A, all questions need to be answered and in Part B, each student can choose any one full question out of two questions*

<b>Part A</b>	<b>Part B</b>	<b>Total</b>
<ul style="list-style-type: none"><li>• 2 Questions from each module.</li><li>• Total of 8 Questions, each carrying 3 marks</li></ul> <p><b>(8x3 = 24 marks)</b></p>	<ul style="list-style-type: none"><li>• Each question carries 9 marks.</li><li>• Two questions will be given from each module, out of which 1 question should be answered.</li><li>• Each question can have a maximum of 3 sub divisions.</li></ul> <p><b>(4x9 = 36 marks)</b></p>	<b>60</b>

# Course Objectives

- C Programming is the Foundation of Other Languages
  - Understanding How Computers Work
  - High Performance and Efficiency
  - Problem-Solving Skills
- 
- A decorative graphic on the right side of the slide, consisting of several overlapping, curved, wavy shapes in shades of light blue, yellow, and a darker blue at the bottom right corner.

# Application in Various Fields

- **Embedded Systems:** If you are working with devices like microcontrollers, IoT devices, or robotics, C is the go-to language because of its efficiency and direct access to hardware.
- **Operating Systems:** Operating systems like Linux and older systems like UNIX are written in C because it allows direct hardware manipulation and efficient memory management.
- **Game Development:** Many game engines are built using C or C++ due to the speed and low-level control they offer.
- **Network Programming:** C is also used in developing network protocols, where efficiency and control are important.

# **MODULE 1**

- **C Fundamentals**

- Character Set, Constants, Identifiers, Keywords
- Basic Data types, Variables, Operators and its precedence, Bit-wise operators
- Expressions; Statements - Input and Output statements;
- Structure of a C program; Simple programs

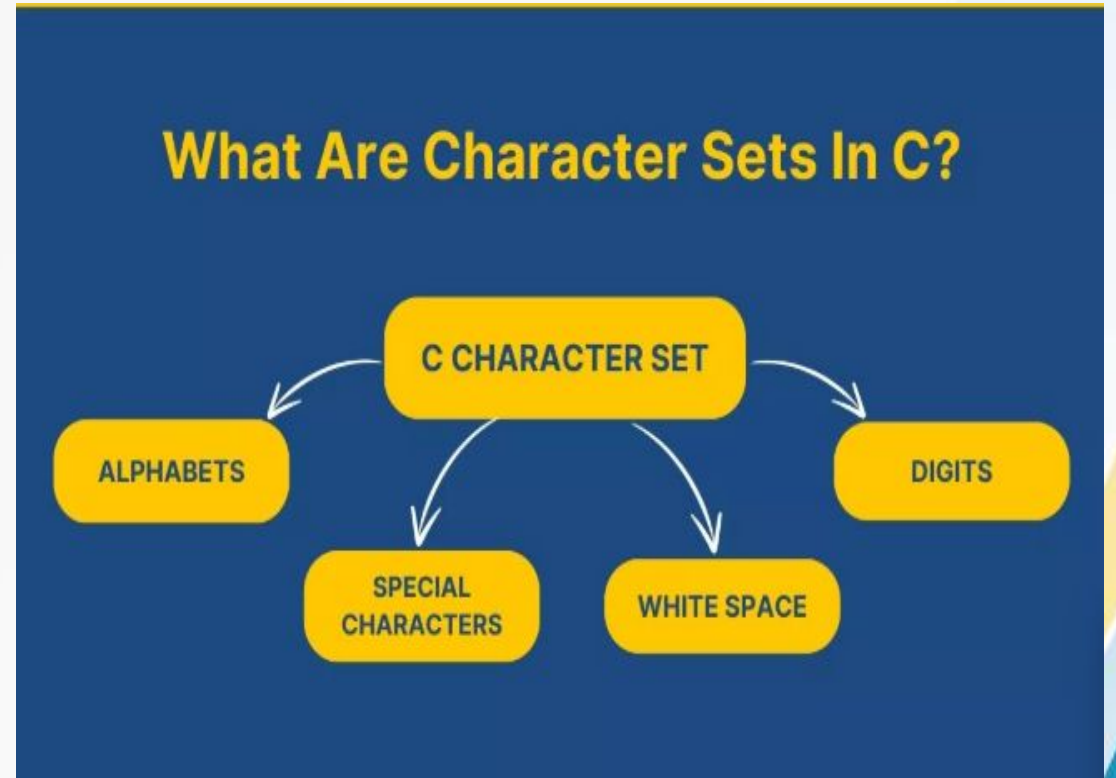
- **Control Statements**

- if, if-else, nested if, switch, while, do-while, for,
- break & continue, nested loops.



# Character Set

- In the C programming language, the character set refers to a set of all the valid characters that we can use in the source program for forming words, expressions, and numbers.





- Alphabets
  - A to Z (uppercase)
  - a to z (lowercase)
- Digits
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Whitespace Characters
  - Space ( )
  - Tab (\t)
  - Newline (\n)
  - Carriage return (\r)
- Escape Sequences (Special character combinations)
  - \n (Newline)
  - \t (Tab)
  - \r (Carriage return)
  - \b (Backspace)
  - \f (Form feed)
  - \\ (Backslash)
  - \' (Single quote)
  - \" (Double quote)

- Special Characters

- ~ (tilde)
- ! (exclamation mark)
- @ (at symbol)
- # (hash)
- \$ (dollar)
- % (percent)
- ^ (caret)
- & (ampersand)
- \* (asterisk)
- () (parentheses)
- {} (curly brackets)

[] (square brackets)

– : (colon)

– ; (semicolon)

– , (comma)

– . (dot)

– < > (less than, greater than)

– / (forward slash)

– \ (backslash)

– | (vertical bar)

– ? (question mark)

– ' (single quote)

– " (double quote)

# Types Of Character Sets In C

- The C character set is divided into two main categories based on the character type:
  - Source Character Set
    - The source character set in C consists of all the characters that can be used in the source code of the program. It allows programmers to write readable and maintainable code using familiar symbols and letters
  - Execution Character Set
    - The execution character set, also known as the control character set in C programming, comprises character sets that are used during the execution of a C program. Control characters are vital for managing the flow of the program, formatting output, and handling input

## Usage of Character Set in C

- Defining Identifiers(variable names, function names) → Use alphabets, digits, and `_`.
- Writing Constants and Keywords → Use alphabets, digits, and special characters.
- Operators and Symbols → Use special characters for arithmetic, logical, and relational operations.
- Formatting Output → Escape sequences like `\n` (newline), `\t` (tab), and `\\` (backslash) are used within character string constants to format output
- Punctuating Code: Punctuation marks such as semicolon(`;`), comma or file separator (`,`), dot(`.`), colon(`:`), and braces(`{}`) are used to optimize program structure by separating program statements, and defining code blocks

# CONSTANTS

- In C programming, constants are fixed values that do not change during the execution of a program.
- They are like "locked" values that remain the same throughout the program.
- There are four basic types of constants in C
  - Integer Constants
  - Floating point constants
  - Character constants
  - String constants

# Integer Constants



- Whole numbers (positive or negative)
- Example: 10, -25, 1000
- ie consists of sequence of digits
- can be written in 3 different number systems
  - Decimal, Octal, Hexadecimal
- Magnitude of integer constant can range from zero to some maximum value that varies from one computer to another computer

# Unsigned Integer Constants



- An unsigned integer constant in C is a whole number that can only store positive values (including zero).
- It cannot store negative numbers.
- By default, integers in C can be positive or negative (signed).
- However, when we declare an unsigned integer, it removes the ability to store negative values and doubles the maximum positive value it can hold.
- An unsigned integer constant can be identified by appending the letter U to the end of constant



- **Why Use Unsigned Integer Constants?**

-  More Memory for Positive Numbers – Useful when storing only positive values (e.g., population, age, IDs).
-  Prevents Negative Values – Ensures values like an account balance or product quantity never go negative.

# Long Integer constant

- A long integer constant in C is a whole number constant that has a larger storage size than a regular integer (int).
- It is used when you need to store very large numbers that exceed the range of normal integers.
- An long integer constant can be identified by appending the letter L to the end of constant
- **When to Use Long Integer Constants?**
  -  For astronomical, financial, or large-scale counting applications
  -  When dealing with high-precision calculations

# Floating point constants

- A floating point constant is a base 10 number that contains either a decimal point or an exponent
- Example: 3.14, -0.005, 2.718
- If an exponent is present, its effect is to shift the location of the decimal point to the right, if the exponent is positive or to the left, if exponent is negative.
- If a decimal point is not included within the number, it is assumed to be positioned to the right of the last digit.

# Character Constants

- A character constant is a single character, enclosed in apostrophes
- Character constants have integer values that are determined by the computer's particular character set.
- A character set is a system that assigns numeric values to characters
  - ASCII-American Standard Code for Information Interchange
  - EBCDIC-Extended Binary Coded Decimal Information Code
- Thus the value of a character constant may vary from one computer to another.
- The constants themselves, however, are independent of the character set.
- This feature eliminates the dependence of a C program on any particular character set

# Escape Sequences

- Escape sequences are special character combinations that start with a backslash (\) to represent things like newlines, tabs, quotes, and more.
- They help you format output and print special characters that would normally be difficult to include in a string.
- \n - Newline
  - This is used to move the cursor to the next line. It's like pressing Enter on your keyboard
- \t - Horizontal Tab
  - This adds a tab space (like pressing the Tab key on your keyboard) between text.

- `\\` - Backslash
  - This represents a single backslash (`\`). Since a backslash is used to start escape sequences, you need to use two backslashes if you want to print one.
- `\'` - Single Quote
  - This is used to print a single quote (`'`), which is tricky because a single quote is used to define character constants.
- `\"` - Double Quote
  - This is used to print double quotes (`"`), which is important because double quotes are used to define string literals in C.

- `\r` - Carriage Return
  - This moves the cursor to the beginning of the current line, without moving to the next line.
- `\b` - Backspace
  - This deletes the character before the cursor (like pressing the Backspace key).



## Examples

- `printf("Hello\nWorld");`
- `printf("Hello\tWorld");`
- `printf("This is a backslash: \\");`
- `printf("It's a sunny day.");`
- `printf("\"Hello, World!\");`
- `printf("Hello\rWorld");`
- `printf("Hello\bWorld");`

# String Constants

- String constants (or string literals) are sequences of characters that are enclosed within double quotes (" ")
- Eg: "Hello, World!"
- Every string constant in C automatically ends with a special character called null terminator (\0). This character is not visible when the string is displayed.
- They are stored in arrays of characters, and you cannot modify a string constant directly.

# **IDENTIFIERS**

- Identifiers are names that are given to various program elements such as variables, functions and arrays.
- Identifier consists of letters and digits in any order, except that first character must be a letter
- Both uppercase and lowercase letters are permitted, though common usage favours the use of lower case letters
- Uppercase and lowercase letters are not interchangeable
- Underscore character is also considered to be a letter, often used in the middle of identifier

- Valid identifiers

- x
- names
- y12
- sum\_1
- \_temperature
- area
- tax\_rate
- TABLE

- As a rule , an identifier should contain enough characters so that its meaning is readily apparent. On the other hand, excessive number of characters should be avoided

- Invalid identifiers

- 4th
- “x”
- order-no
- error flag
- Some implementations of C recognize only the first eight characters, though most of the implementations recognize more (31 characters)

# KEYWORDS

- Keywords are reserved words, that have standard, predefined meanings in C
- These keywords can be used only for intended purpose
- Note that keywords are all lowercase

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

# **BASIC DATA TYPES**

- Each variable in C has an associated data type.
- It specifies the type of data that the variable can store like integer, character, floating, double, etc.
- Each data type requires different amounts of memory and has some specific operations which can be performed over it.

Types	Description	Data Types
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.	int, char, float, double, void
<u>Derived Types</u>	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.	array, pointers, function
<u>User Defined Data Types</u>	The user-defined data types are defined by the user himself.	structure, union, enum



# Integer Data Type

- The integer datatype in C is used to store the integer numbers
- Range: -2,147,483,648 to 2,147,483,647
- Size: 4 bytes
- Format Specifier: %d
- Syntax of Integer: `int var_name;`

```
// C program to print Integer data types.
```

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

```
int main()
```

```
{
```

```
    // Integer value with positive data.
```

```
    int a = 9;
```

```
    // integer value with negative data.
```

```
    int b = -9;
```

```
    // U or u is Used for Unsigned int in C.
```

```
    int c = 89U;
```

```
    // L or l is used for long int in C.
```

```
    long int d = 99998L;
```

```
    printf("Integer value with positive data: %d\n", a);
```

```
    printf("Integer value with negative data: %d\n", b);
```

```
    printf("Integer value with an unsigned int data: %u\n",  
           c);
```

```
    printf("Integer value with an long int data: %ld", d);
```

```
    return 0;
```

```
}
```

# Character Data Type

- Character data type allows its variable to store only a single character.
- The size of the character is 1 byte.
- It is the most basic data type in C.
- It stores a single character and requires a single byte of memory in almost all compilers.
- Range: (-128 to 127) or (0 to 255)
- Size: 1 byte
- Format Specifier: %c
- Syntax: `char var_name;`

```
// C program to print Integer data types.
#include <stdio.h>

int main()
{
    char a = 'a';
    char c;

    printf("Value of a: %c\n", a);

    a++;
    printf("Value of a after increment is: %c\n", a);

    // c is assigned ASCII values
    // which corresponds to the
    // character 'c'
    // a-->97 b-->98 c-->99
    // here c will be printed
    c = 99;

    printf("Value of c: %c", c);

    return 0;
}
```

# Float Data Type

- In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.
- Range:  $1.2\text{E}-38$  to  $3.4\text{E}+38$
- Size: 4 bytes
- Format Specifier: %f
- Syntax: float var\_name;

```
// C Program to demonstrate use  
// of Floating types  
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    float a = 9.0f;
```

```
    float b = 2.5f;
```

```
    // 2x10-4
```

```
    float c = 2E-4f;
```

```
    printf("%f\n", a);
```

```
    printf("%f\n", b);
```

```
    printf("%f", c);
```

```
    return 0;
```

```
}
```

# Double Data Type

- A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision.
- The double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points.
- Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.
- Range: 1.7E-308 to 1.7E+308
- Size: 8 bytes
- Format Specifier: %lf
- Syntax: double var\_name;



```
// C Program to demonstrate
// use of double data type
#include <stdio.h>

int main()
{
    double a = 123123123.00;
    double b = 12.293123;
    double c = 2312312312.123123;

    printf("%lf\n", a);

    printf("%lf\n", b);

    printf("%lf", c);

    return 0;
}
```

# **VARIABLE**

- A variable is a data name that may be used to store a data value.
- A variable may take different values at different times during execution.
- Each variable has a specific storage location in memory where its value is stored.
- The variables are called symbolic variables because these are named.

- There are two values associated with a symbolic variable.
  - Data value: stored at some location in memory. This is sometimes referred to as a variable's r- value.
  - Location value: This is the address in memory at which its data value is stored. This is sometimes referred to as variable's l-value.

1051	1052	1053	1054	1055	memory address
	10			25	Data values of variables
	A			C	Variable's name

- r-value of A=10 and l-value of A=1052
- r-value of c=25 and l-value of c=1055
- Whenever we use the assignment operator, the expression to the left of an assignment operator must be an l-value.

That is, it must provide an accessible memory address where the data can be written to.

# Variable Declaration

- A declaration associates a group of variables with a specific data type.
- Declaration of variables must be done before they are used in the program.
- Syntax
- `Data_type variable_name;`
- Eg: `int a;`
- `float a, b, c;`
- `char name [10]; // Character array declaration`

# Variable Initialization

- The process of giving initial values to variables is called
- initialization.
- Eg: `int x=10;`
- `float n=22.889;`
- `char answer='y';`

# **OPERATORS AND ITS PRECEDENCE**

- Operators are symbols that represent operations to be performed on one or more operands
- The data items that operators act upon are called operands
- Eg :  $c = a + b$ ;
- Here, '+' is the operator known as the addition operator, and 'a' and 'b' are operands.

- In C, **operator precedence** determines the order of evaluation in an expression.
- Operators with higher precedence are evaluated before operators with lower precedence. If operators have the same precedence, their associativity (left-to-right or right-to-left) determines the evaluation order.

```
#include <stdio.h>

int main() {
    int result = (10 + 5) * 2;
    printf("Result: %d\n", result); // Output: 30
    return 0;
}
```



Precedence	Operators	Description	Associativity
1	<code>() [] -&gt; .</code>	Parentheses, array access, structure member, function call	Left to Right
2	<code>++ --</code>	Post-increment, post-decrement	Left to Right
3	<code>+ - ! ~ ++ --</code> <code>(type) * &amp;</code> <code>sizeof</code>	Unary operators: sign change, logical NOT, bitwise NOT, pre-increment, pre-decrement, type cast, indirection (dereference), address-of, <code>sizeof</code>	Right to Left
4	<code>* / %</code>	Multiplication, division, modulus	Left to Right
5	<code>+ -</code>	Addition, subtraction	Left to Right
6	<code>&lt;&lt; &gt;&gt;</code>	Bitwise shift left, bitwise shift right	Left to Right
7	<code>&lt; &lt;= &gt; &gt;=</code>	Relational (less than, less than or equal, greater than, greater than or equal)	Left to Right
8	<code>== !=</code>	Equality and inequality	Left to Right

- **Associativity** decides the direction (left-to-right or right-to-left) in which operators of the same precedence are evaluated.
- There are two types of associativity:
  - Left to Right ( $L \rightarrow R$ ): Operators are evaluated from left to right.
  - Right to Left ( $R \rightarrow L$ ): Operators are evaluated from right to left.

# Example of Left-to-Right Associativity

```
c

#include <stdio.h>

int main() {
    int result = 10 - 5 - 2;
    printf("Result: %d\n", result); // Output: 3
    return 0;
}
```

# Example of Right-to-Left Associativity

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = 5;
    int c = 2;

    int result = a = b = c;
    printf("a: %d, b: %d, c: %d\n", a, b, c); // Output: a: 2, b: 2, c: 2
    return 0;
}
```

- Left-to-right associativity applies to most binary operators like `+`, `-`, `*`, `/`, `&&`, `||`.
- Right-to-left associativity applies to assignment operators (`=`, `+=`, `-=`, `*=` etc.), unary operators (`++`, `--`, `!`, `~`), and the ternary operator (`?:`).

# **Types of Operators in C**

- Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Conditional Operators
- 
- A decorative graphic on the right side of the slide, consisting of several overlapping, curved, wavy shapes in shades of light blue, yellow, and a darker blue, creating a modern, abstract background element.

# Arithmetic Operators

- The arithmetic operators are used to perform arithmetic /mathematical operations on operands.
- There are 5 arithmetic operators
  - + Addition
  - - Subtraction
  - \* Multiplication
  - / Division
  - % Modulus Operator

- No exponential operator in C. Uses library function (pow) to carry out exponentiation
- The operands acted upon by arithmetic operators must represent numeric values
- The Modulus operator requires that both operands must be integers and second operand be non zero
- Similarly division operator requires that second operand be non zero
- Integer Division: Division of one integer quantity by another. This operation always result in truncated quotient



- If one or both operands represent negative values ,then addition, subtraction, multiplication and division operation will result in values whose signs are determined by usual rules of algebra
- Operands that differ in type may undergo type conversion before the expression takes on its final value.
- In general ,the final result will be expressed in the highest precision possible
- The value of an expression can be converted to a different data type if desired
  - (data type) expression
- This type of construction is known as cast

## Example: Implicit Type Conversion

c Copy Edit

```
#include <stdio.h>

int main() {
    int a = 5;
    float b = 2.5;

    float result = a + b; // 'a' (int) is implicitly converted to float before addition

    printf("Result: %f\n", result); // Output: 7.500000
    return 0;
}
```

## Example: Type Promotion in Mixed Expressions

```
c Copy Edit  
  
#include <stdio.h>  
  
int main() {  
    char c = 'A'; // ASCII value of 'A' is 65  
    int num = 5;  
  
    int result = c + num; // 'c' (char) is promoted to int before addition  
  
    printf("Result: %d\n", result); // Output: 70  
    return 0;  
}
```

# When Implicit Conversions Occur in C

- When operands have different data types, C promotes the smaller type to a larger type to maintain precision.
- When assigning a value of a smaller data type to a larger data type, implicit conversion happens without data loss.
- `char → short → int → long → float → double → long double`

c

Copy Edit

```
#include <stdio.h>

int main() {
    int a = 5, b = 2;

    float result = (float) a / b; // Explicitly converting 'a' to float before division

    printf("Result: %f\n", result); // Output: 2.500000

    return 0;
}
```

# Unary Operators

- Unary operators in C are operators that operate on a single operand. These operators perform operations like incrementing, decrementing, negating, or getting the size of a variable.
- Unary operators have higher precedence than Arithmetic operators
- **Unary Plus (+) and Unary Minus (-)**
  - + represents a positive value (usually not needed).
  - - negates a value.
- Example:
  - `int x = 10;`
  - `int y = -x; // y becomes -10`

- **Increment (++) and Decrement (--) Operators**

- These increase or decrease the value of a variable by 1.
- They can be used in prefix (++a, --a) and postfix (a++, a--) forms
- `int a = 5;`

`printf("%d", ++a); // Output: 6 (prefix increment)`

`printf("%d", a++); // Output: 6 (postfix increment, prints first, then increases)`

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

```
int main() {
```

```
    int a = 5;
```

```
    printf("%d\n", a++); // Post-increment
```

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

```
    return 0;
```

```
}
```



c

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

```
int main() {
```

```
    int x = 10;
```

```
    int y = ++x + x++;
```

```
    printf("%d %d\n", x, y);
```

```
    return 0;
```

```
}
```

- **Sizeof Operator (sizeof)**

- Determines the size of a data type or variable in bytes.
- `printf("%lu", sizeof(int));` // Output: 4 (on most systems)

- **Logical NOT (!)**

- It inverts a boolean value.
- true (1) becomes false (0) and vice versa.
- `int a = 0;`  
`printf("%d", !a);` // Output: 1 (because 0 is false)

- **Bitwise NOT (~)**

- Inverts all bits of a number (one's complement).
- $\sim x$  flips all bits: 0 to 1 and 1 to 0.
- `int a = 5;`  
`printf("%d", ~a);` // Output: -6 (in two's complement representation)
- For an integer  $x$ ,  $\sim x$  is equivalent to  $-(x + 1)$  in two's complement representation

- **Address-of (&) Operator**

- Returns the memory address of a variable.
- `int a = 10;`  
`printf("%p", &a);` // Prints address of 'a'
- In C, the `%p` format specifier is used in `printf` to print pointer values, which are the memory addresses of variables.

# Relational Operator

- Relational operators in C are used to compare two values.
- They return 1 (true) if the condition is met, otherwise they return 0 (false).

Operator	Meaning	Example	Output (if $a = 10$ , $b = 5$ )
<code>==</code>	Equal to	<code>a == b</code>	0 (false)
<code>!=</code>	Not equal to	<code>a != b</code>	1 (true)
<code>&gt;</code>	Greater than	<code>a &gt; b</code>	1 (true)
<code>&lt;</code>	Less than	<code>a &lt; b</code>	0 (false)
<code>&gt;=</code>	Greater than or equal to	<code>a &gt;= b</code>	1 (true)
<code>&lt;=</code>	Less than or equal to	<code>a &lt;= b</code>	0 (false)

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;
    printf("a == b: %d\n", a == b); // 0 (false)
    printf("a != b: %d\n", a != b); // 1 (true)
    printf("a > b: %d\n", a > b);    // 1 (true)
    printf("a < b: %d\n", a < b);    // 0 (false)
    printf("a >= b: %d\n", a >= b); // 1 (true)
    printf("a <= b: %d\n", a <= b); // 0 (false)
    return 0;
}
```

- The last 4 relational operators fall within the same precedence group, which is lower than arithmetic and unary operators
- Associativity of these operators is from left to right
- The equality operators fall into separate precedence group, below the relational operators
- These operators also have left to right associativity
- These six operators are used to form logical expressions, which represent conditions that are either true or false

# Logical Operators

- Logical operators in C are used to combine multiple conditions and return true (1) or false (0).
- They are mostly used in decision-making statements like if, while, and for loops.

Operator	Meaning	Example ( a = 5, b = 10 )	Result
&&	Logical AND	(a > 2) && (b < 20)	1 (true)
,		,	Logical OR
!	Logical NOT	!(a == 5)	0 (false)

- **Logical AND (&&)**

- Returns true (1) only if both conditions are true.
- If any condition is false, it returns false (0)

```
int a = 5, b = 10;  
if (a > 2 && b < 20) {  
    printf("Both conditions are true\n");  
}
```



- **Logical OR (||)**

- Returns true (1) if at least one condition is true.
- Returns false (0) only if both conditions are false.

```
int a = 5, b = 30;  
if (a > 2 || b < 20) {  
    printf("At least one condition is true\n");  
}
```

- **Logical NOT (!)**
- Reverses the condition:
  - true (1) becomes false (0).
  - false (0) becomes true (1)

```
int a = 5;
if (!(a == 5)) {
    printf("Condition is false\n");
} else {
    printf("Condition is true\n");
}
```

---

# Bitwise Operators


- Bitwise operators in C are used to manipulate individual bits of numbers.
- These operators perform operations at the binary level, making them very fast and efficient.

Operator	Symbol	Description
Bitwise AND	<code>&amp;</code>	Sets bits to <code>1</code> only if both bits are <code>1</code>
Bitwise OR	<code> </code>	
Bitwise XOR	<code>^</code>	Sets bits to <code>1</code> if bits are different
Bitwise NOT	<code>~</code>	Inverts all bits ( <code>0</code> to <code>1</code> , <code>1</code> to <code>0</code> )
Left Shift	<code>&lt;&lt;</code>	Shifts bits left, multiplying by <code>2</code>
Right Shift	<code>&gt;&gt;</code>	Shifts bits right, dividing by <code>2</code>

Rule:  $1 \& 1 = 1$ , otherwise 0

```
#include <stdio.h>

int main() {
    int a = 5, b = 3; // Binary: a = 0101, b = 0011
    int result = a & b; // 0101 & 0011 = 0001 (Decimal: 1)
    printf("Bitwise AND: %d\n", result);
    return 0;
}
```

 Output: Bitwise AND: 1

- Rule:  $1 \mid 1 = 1$ ,  $1 \mid 0 = 1$ ,  $0 \mid 0 = 0$


```
int a = 5, b = 3; // Binary: a = 0101, b = 0011
int result = a | b; // 0101 | 0011 = 0111 (Decimal: 7)
printf("Bitwise OR: %d\n", result);
```

---

Output: Bitwise OR: 7

- Rule:  $1 \wedge 1 = 0$ ,  $1 \wedge 0 = 1$ ,  $0 \wedge 1 = 1$ ,  $0 \wedge 0 = 0$

```
int a = 5, b = 3; // Binary: a = 0101, b = 0011
int result = a ^ b; // 0101 ^ 0011 = 0110 (Decimal: 6)
printf("Bitwise XOR: %d\n", result);
```

 Output: Bitwise XOR: 6


- Rule: Flips bits:  $0 \rightarrow 1$ ,  $1 \rightarrow 0$

```
int a = 5;    // Binary: 0101
int result = ~a; // ~0101 = 1010 (Decimal: -6 in two's complement)
printf("Bitwise NOT: %d\n", result);
```

 Output: Bitwise NOT: -6

- Rule: Shifts bits left, multiplying the number by 2

```
int a = 5; // Binary: 00000101
int result = a << 1; // 00001010 (Decimal: 10)
printf("Left Shift: %d\n", result);
```

 Output: Left Shift: 10

markdown

00000000 00000000 00000000 00000101 (Original: 5)

<< 2

-----  
00000000 00000000 00000000 00010100 (Result: 20)



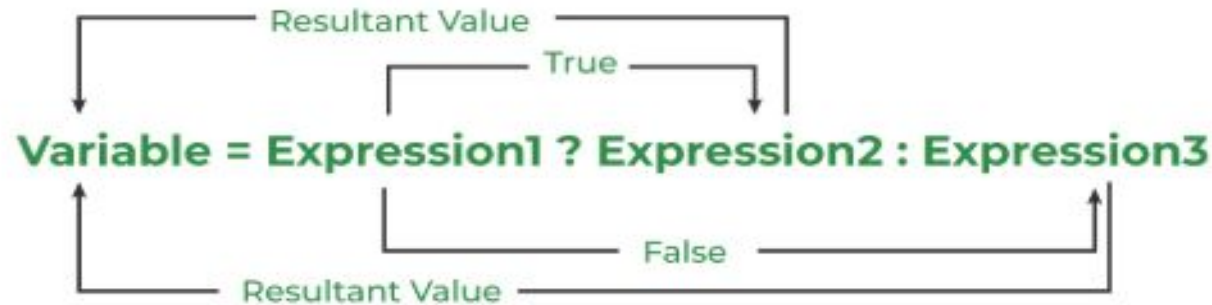
- Rule: Shifts bits right, dividing the number by 2

```
int a = 8;    // Binary: 00001000
int result = a >> 1; // 00000100 (Decimal: 4)
printf("Right Shift: %d\n", result);
```

 Output: Right Shift: 4

# Conditional Operator

- The conditional operator in C is kind of similar to the if-else statement
- The conditional operator takes less space and helps to write the if-else statements in the shortest way possible.
- It is also known as the ternary operator in C as it operates on three operands.
- `variable = (Expression1) ? Expression2 : Expression3;`

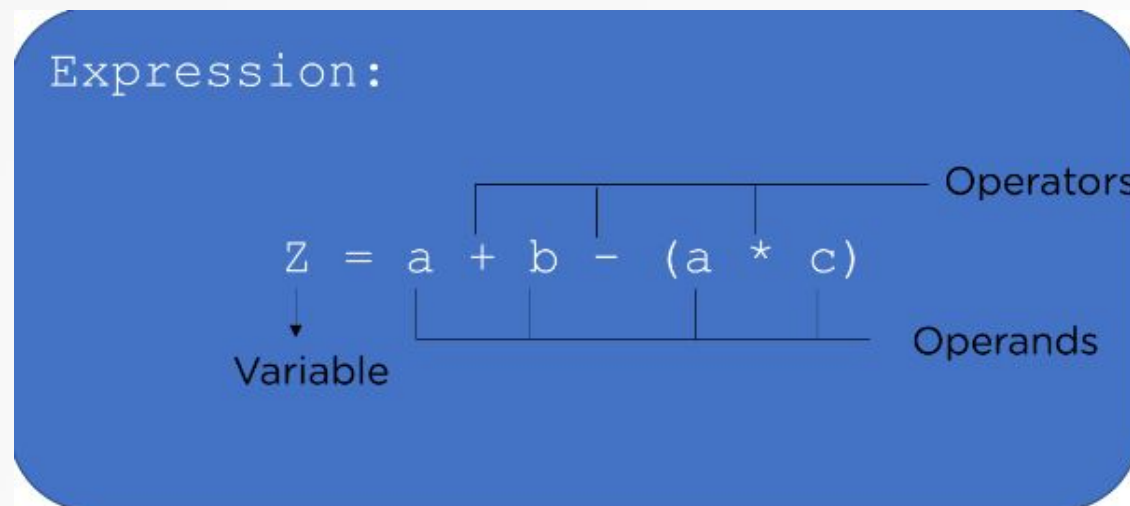


# Assignment Operator

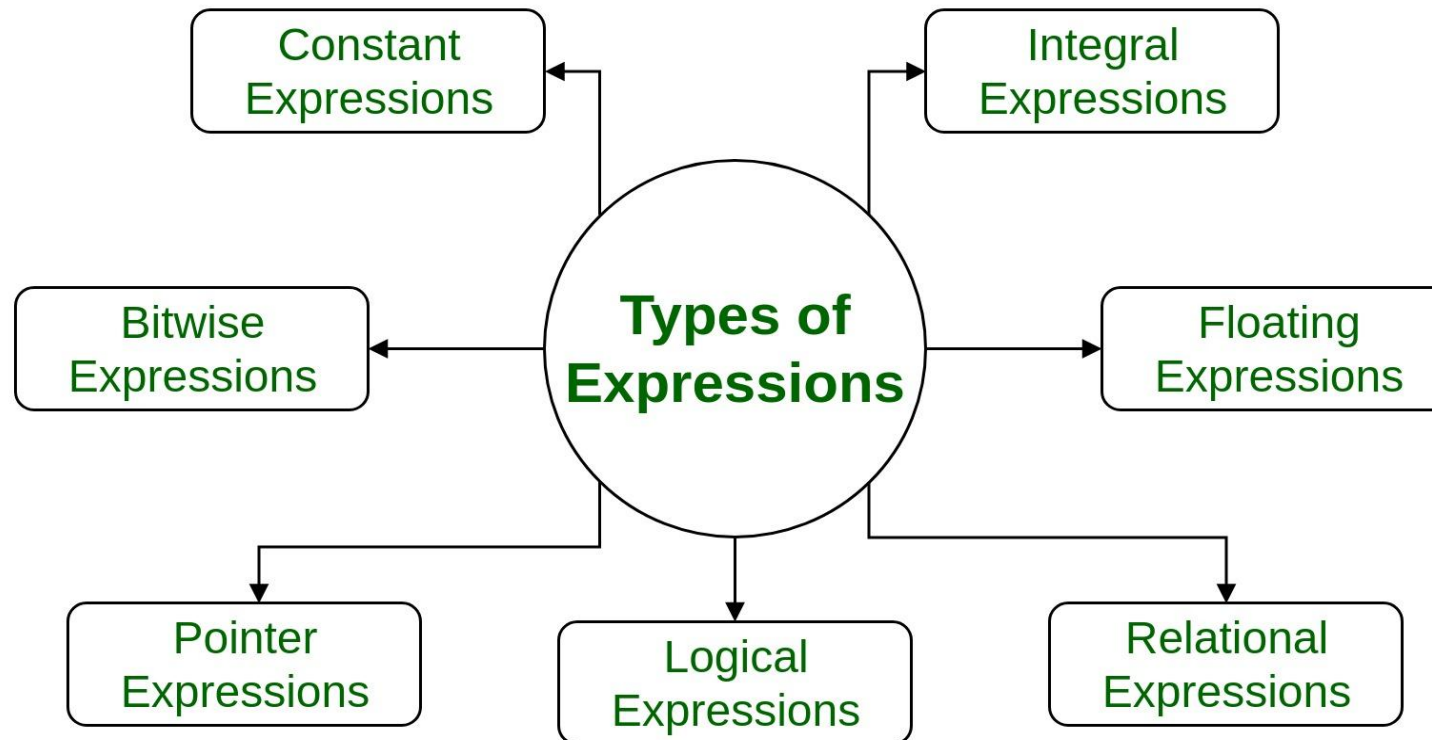
- Used to form assignment expressions, which assign the value of an expression to the identifier
- Most commonly used assignment operator is =
- General form : identifier=expression
  - where identifier represents a variable and expression represents a constant, variable or more complex expression
- Ex : a=3 //Assigns the integer value 3 to variable a
- sum=a+b //Assigns the value of arithmetic expression being assigned to a variable
- x=y //Assigns the value of variable y to variable x

# Expressions

- An expression is a combination of operands (variables, constants) and operators that produce a value.
- The evaluations of an expression in C happen according to the operator's precedence.
- Once the expressions are processed, the result will be stored in the variable.



# Types of Expressions



# Arithmetic expression

- It consists of arithmetic operators ( + , - , \* , and / ) and computes values of int, float, or double type.
- #include <stdio.h>

```
int main(){
```

```
//Arithmetic Expression
```

```
int a = (6 * 2) + 7 - 9;
```

```
printf("The arithmetic expression returns: %d\n", a);
```

```
return 0; }
```

# Relational Expressions

- Relational operators  $>$ ,  $<$ ,  $==$ ,  $!=$  etc are used to compare 2 operands.
- Relational expressions consisting of operands, variables, operators, and the result after evaluation would be either true or false.

Relational Expression:

$a=3, b=2, c=1$

$Z = a * b > a + c$

$Z = 3 * 2 > 3 + 1$   
     $\uparrow \quad \uparrow$   
 $= 6 > 3 + 1$   
             $\uparrow \quad \uparrow$   
 $= 6 > 4$   
  
 $= \text{true (1)}$

# Logical Expressions

- Relational expressions and arithmetic expressions are connected with the logical operators, and the result after an evaluation is stored in the variable, which is either true or false.

Logical Expression:

a=2, b=4, c=3

Z = (a + b) > c && a < b

Z = (2 + 4) > 3 && 2 < 4

= 6 > 3 && 2 < 4

= true



# Conditional Expressions

- The general syntax of conditional expression is:
- $\text{Exp1} ? \text{Exp2} : \text{Exp3}$
- From the given above expressions, the first expression (exp1) is conditional, and if the condition is satisfied, then expression2 will be executed; otherwise, expression3 will be performed.

Conditional Expression:

$a=5, b=3, c=1$

$a * b < c ? \text{true} : \text{false}$

$Z = 5 * 3 > 1 ? \text{'true'} : \text{'false'}$

$= 15 > 1 ? \text{'true'} : \text{'false'}$

$= \text{false}$

# INPUT OUTPUT STATEMENTS

- In C programming, "input and output" (I/O) statements refer to functions used to read data from external sources (like user input) and write data to external destinations (like the console), allowing a program to interact with the user
- The most commonly used I/O functions are `scanf()` for input and `printf()` for output, both found within the `<stdio.h>` header file

# getchar Function

- Single characters can be entered into computer using getchar function
- It returns a single character from a standard input device
- The function doesnot require any arguments,though a pair of empty paranthesis must follow the word getchar
- In general, `character_variable=getchar();`
  - `character_variable` refers to some previously declared character variable
- The getchar function can also be used to read multicharacter strings,by reading one character at a time within multipass loop

- If an EOF condition is encountered when reading a character with `getchar` function, the value of symbolic constant `EOF` will automatically be returned.
- The detection of EOF in this manner offers a convenient way to detect an end of file
  - It's like a special signal or a "stop" sign that tells the program, "Hey, you're at the end of the file, there are no more characters to read!" So, when `getchar()` reaches the end of the file, it will automatically return this special signal called `EOF`

# putchar function

- Single character output
- Single characters can be displayed using putchar function
- This function is complementary to character input function getchar
- It transmits a single character to a standard output device
- The character being transmitted will normally be represented as character type variable
- It must be expressed as an argument to the function ,enclosed in paranthesis

# **THE scanf FUNCTION**

- Input data can be entered into the computer from a standard input device by means of the C library function scanf
- This function can be used to enter any combination of numerical values, single characters and strings.
- The function returns the number of data items that have been entered successfully

- In general terms, the scanf function is written as
- `scanf(control string, arg1, arg2, . . . , argn)`
  - where control string refers to a string containing certain required formatting information
  - `arg1, arg2, . . . argn` are arguments that represent the individual input data items.
  - Actually, the arguments represent pointers that indicate the addresses of the data items within the computer's memory

- The control string consists of individual groups of characters, with one character group for each input data item.
- Each character group must begin with a percent sign (%).
- In its simplest form, a single character group will consist of the percent sign, followed by a conversion character which indicates the type of the corresponding data item.



<i>Conversion Character</i>	<i>Meaning</i>
c	data item is a single character
d	data item is a decimal integer
e	data item is a floating-point value
f	data item is a floating-point value
g	data item is a floating-point value
h	data item is a short integer
i	data item is a decimal, hexadecimal or octal integer
o	data item is an octal integer
s	data item is a string followed by a whitespace character (the null character \0 will automatically be added at the end)
u	data item is an unsigned decimal integer
x	data item is a hexadecimal integer
[ . . . ]	data item is a string which may include whitespace characters (see explanation below)

**EXAMPLE 4.5** Here is a typical application of a `scanf` function.

```
#include <stdio.h>

main()
{
    char item[20];
    int partno;
    float cost;

    . . . . .

    scanf("%s %d %f", item, &partno, &cost);

    . . . . .
}
```

# Printf function

- Output data can be written from a computer onto a standard output device using the library function printf
- General syntax
  - `printf(control string, arg1,arg2,.....argn);`
  - control string- refers to string that contains formatting information
  - `arg1,arg2,..agn` are arguments that represent the individual output data items

# Right Align the Output

- We can right align the output using the width specifier with positive value.
- `#include <stdio.h>`
- `int main() {`
- `char s[] = "Welcome to GfG!";`
- `// Printing right aligned string of width 100`
- `printf("%100s", s);`
- `return 0;`
- `}`

Output

- Welcome to GfG!