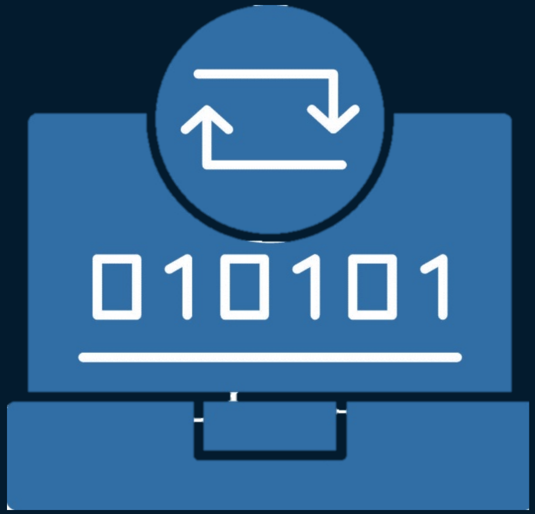
Embedded Systems

Embedded C Programming



Dawood Mazhar

Research Associate,
Namal University Mianwali

Work Experience:
- Namal University, Mianwali
- Riphah International University. Islamabad

Core Strengths:
- Embedded Systems Development
- Power-Efficient Embedded System Control
- Sensor Integration and Data Processing
- Low-Level Programming and Optimization
- Hardware and Software Interface
- Robotics and Prosthesis

Education

Bachelor of Science in Electrical Engineering, Pakistan Institute of Engineering and Applied Science (PIEAS)

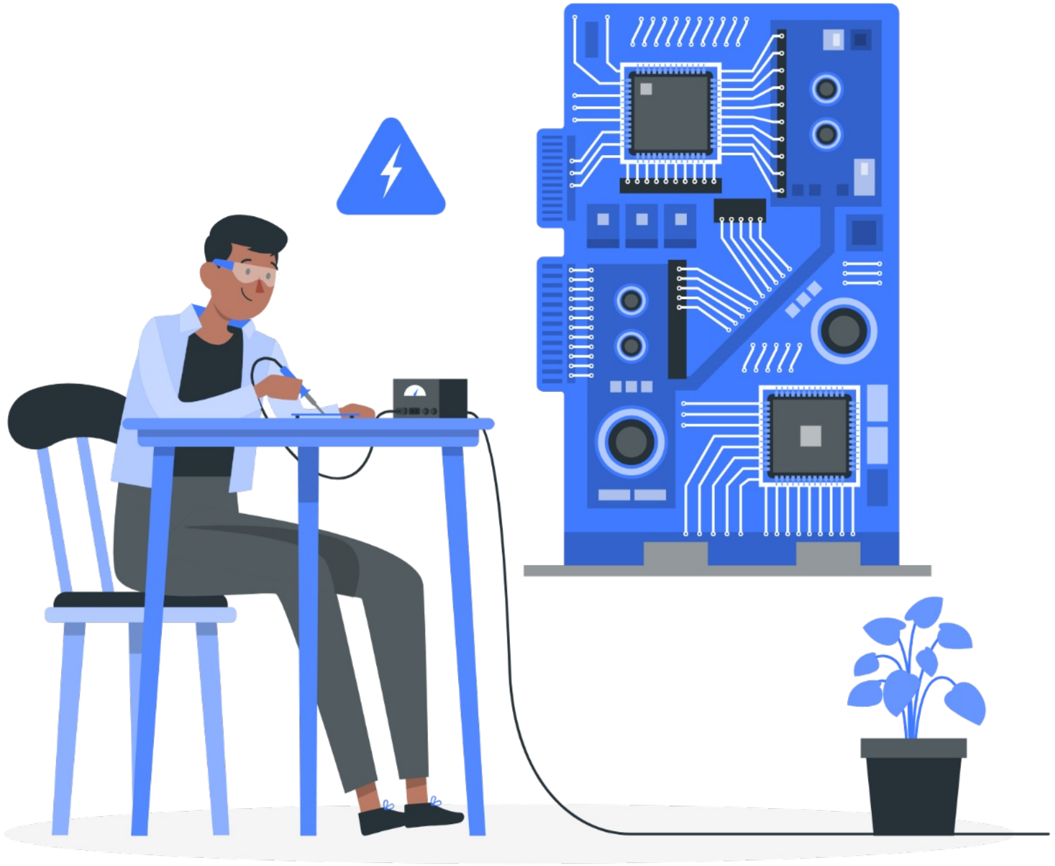
Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- Functions in C
- Memory Layout in C
- Arrays
- Operators in C

Introduction to Embedded C Programming

Embedded C

- Embedded C and standard C (often just called "C") are both programming languages used to write software, but they differ in their target environments, constraints, and some aspects of functionality.
- Embedded C can be considered as the subset of C language. It uses same core syntax as C.
- Embedded C programs need cross-compliers to compile and generate HEX code
- Embedded C is designed for embedded system programming with specific constraints, hardware interaction requirements, and specialized development tools.



C Language vs Embedded C Language

VS

C Language	Embedded C Language
Target Environment	
A structural and programming language used by developers to create desktop-based applications	An extension of C primarily used to develop microcontroller based applications.
Memory Constraint	
Typically used on systems with more resources.	Often used in environments with limited resources (memory, processing power).
Hardware Interaction	
Hardware interactions are managed by operating system or libraries, unless used in system-level programming.	Interacts directly with hardware components, such as registers, I/O ports, and peripheral devices.
Libraries and Extensions	
Uses standard libraries provided by the C standard library (e.g., stdio.h, stdlib.h) and other platform-specific or third-party libraries.	Uses specialized libraries and extensions for embedded systems (e.g., specific APIs for handling hardware interrupts, timers, and serial communication).

C Language vs Embedded C Language

VS

Development Tools

Typically uses general-purpose IDEs (e.g., Visual Studio, Eclipse) and compilers (e.g., GCC, Clang).

Specific Integrated Development Environments (IDEs), compilers, and debuggers designed for embedded system development (e.g., Keil, IAR, MPLAB).

Real-Time Constraint

It can be used in real-time applications, but it is not inherently designed for real-time constraints and may rely on external real-time extensions or operating systems.

Often used in real-time systems where meeting timing constraints is crucial. It may include real-time operating systems (RTOS) or bare-metal programming.

Code Portability

Code is generally more portable across different platforms, adhering to the C standard.

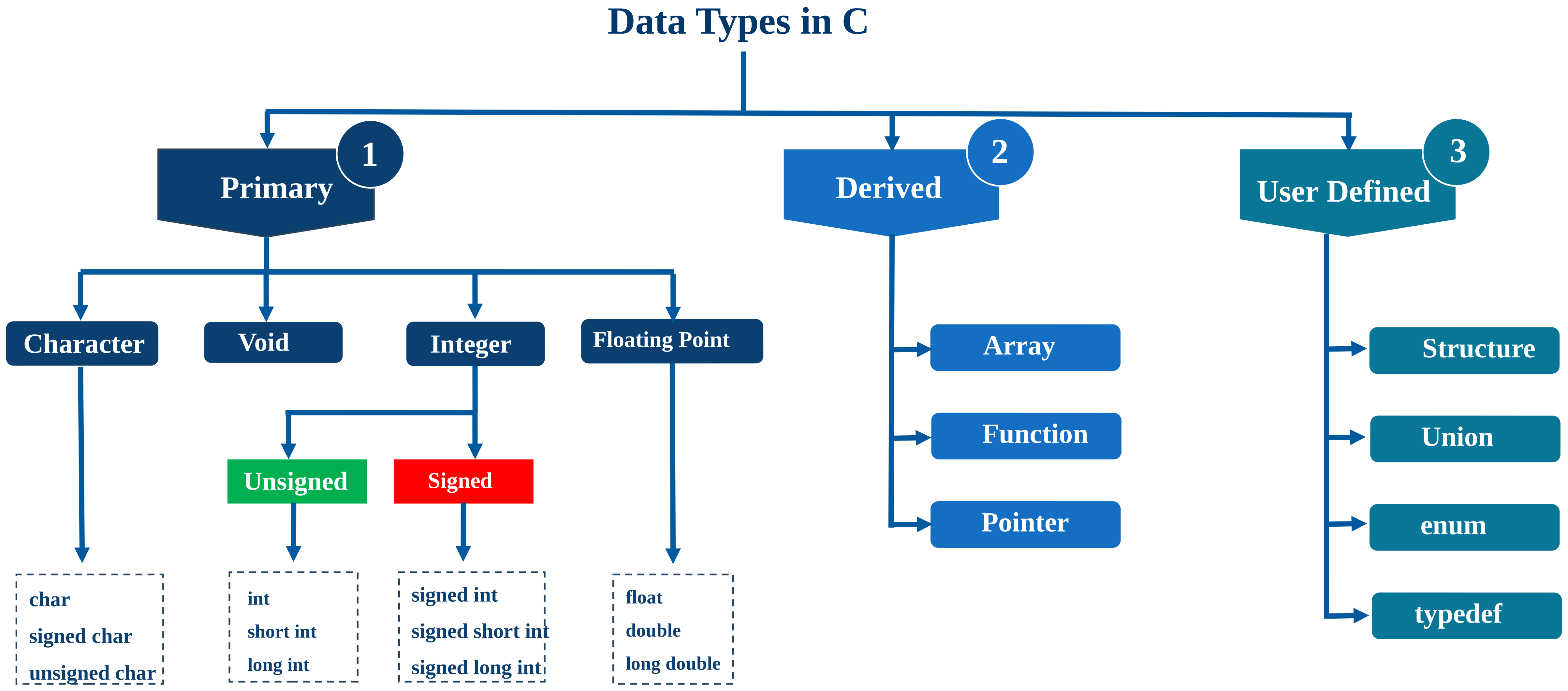
Code is often less portable due to hardware-specific dependencies and optimizations. Porting code between different embedded platforms can be challenging.

Basic C Program Structure

```
#include "debug.h"          /* I/O port/register names/addresses for the microcontrollers */

/* Global variables – accessible by all functions */
int count, bob;             //global (static) variables – placed in RAM

/* Function definitions*/
int function1(char x) {     //parameter x passed to the function, function returns an integer value
int i,j;                    //local (automatic) variables – allocated to stack or registers
– instructions to implement the function
}

 /* Main program */
void main(void) {
 unsigned char sw1;         //local (automatic) variable (stack or registers)
int k;                      //local (automatic) variable (stack or registers)
/* Initialization section */
-- instructions to initialize variables, I/O ports, devices, function registers
/* Endless loop */
while (1) {                 //Can also use: for(;;) {
 -- instructions to be repeated }
 /* repeat forever */ }
```

1. **Compilers directives & Header files**
2. **Global variables & Constants Declarations**
3. **Declarations of Functions**
4. **Main Functions**
5. **Sub-Functions**
6. **Interrupt Service Routines**

C Data Types, Modifiers, Qualifiers



Primary Data Types

Primary data-types are built-in data types provided by C language itself therefore all the compilers support these data-types. Following are the primary data-types that are available in C.

- **Character Data Type, char:**

  char data is a fixed length data type which stores a single character which typically takes 1 Byte memory.

  ```
  Syntax ====>    char variable_name = ‘A’;
  ```

  Character data type stores characters as individual and at memory location it contains an integer value that represents the ASCII code of that character (sometimes can be different coding scheme).

  Can we store numbers in char data type ????????

  Storing number in ‘char’:

  1) Storing character codes:

  You can store integer values directly in a char variable. These integer values represent the ASCII codes for characters.

  ```
  char ch = 65;  // ASCII code for 'A’
  printf("%c\n", ch); // Output: A
  ```

  In this case, 65 is the ASCII code for the character 'A'. When you print ch using the `%c` format specifier, it prints 'A'.

  2) Storing Numbers:

  You can store numeric characters (i.e., '0', '1', '2', ..., '9') in a char variable. These are just characters with specific ASCII values.

  ```
  char digit = '5'; // The character '5’
  printf("%c\n", digit); // Output: 5
  ```

  Here, the character '5' has an ASCII value of 53.

Primary Data Types

Primary Data Types (cont….)

- **Integer Data Type:**

Integer data types are used to store used for storing whole numbers (both positive and negative) without any fractional or decimal component.

The size and range of integer types depend on the system architecture and compiler implementation. In C, several integer types are provided to accommodate different sizes and ranges.

```
Syntax ====> int variable_name = value
```

eg. `int a = 100; int b = -50;`

Primary Data Types in C

Data type	Keyword	Qualifier	Final Definition	Memory bytes	Range
Character	char		char	1	
		unsigned	unsigned char	1	
Integer	int		int	2	
		unsigned	unsigned char	2	
		signed	signed char	2	
		short	short int	2	
		unsigned short	unsigned short int	2	
		signed short	signed short int	2	
		long	Long int	4	
		unsigned long	unsigned long int	4	
		signed long	signed long int	4	
Decimal	float		float	4	
	double		double	8	
		long	long double	10	

Embedded C Data Type Examples

- **Read bits from GPIOA (16 bits, non-numeric)**

–uint16_t n; n = GPIOA->IDR;

// Reads a 16-bit value from GPIO port A's Input Data Register, reflecting the state of GPIO pins.

- **Write TIM2 prescaler value (16-bit unsigned)**

– uint16_t t; TIM2->PSC = t; //or: unsigned short t;

// Sets the prescaler value for Timer 2, affecting the timer’s frequency by dividing the clock input.

- **Read 32-bit value from ADC (unsigned)**

– uint32_t a; a = ADC; //or: unsigned int a;

// Reads a 32-bit unsigned value from the ADC, representing the result of an analog-to-digital conversion.

- **System control value range [-1000...+1000]**

– int32_t ctrl; ctrl = (x + y)*z; //or: int ctrl;

//Holds control values within a range of -1000 to +1000, suitable for various system configurations.

- **Loop counter for 100 program loops (unsigned)**

– uint8_t cnt; //or: unsigned char cnt;

– for (cnt = 0; cnt < 20; cnt++) //Used as a loop counter with a range of 0 to 255, ideal for iteration in loops.

Decimal, Hexadecimal, Octal, and Character Values in C

- **Decimal is the default number format**

```
int m,n;            //16-bit signed numbers
m = 453; n = -25;
```

- **Hexadecimal: preface value with 0x or 0X**

```
m = 0xF312; n = -0x12E4;
```

- **Octal: preface value with zero (0)**

```
m = 0453; n = -023;
```

Don’t use leading zeros on “decimal” values. They will be interpreted as octal.

- **Character: character in single quotes, or ASCII value following “slash”**

```
m = ‘a’; //ASCII value 0x61
n = ‘\13’; //ASCII value 13 is the “return” character
```

- **String (array) of characters:**

```
unsigned char k[7];
strcpy(m,“hello\n”);
    //k[0]=‘h’, k[1]=‘e’, k[2]=‘l’, k[3]=‘l’, k[4]=‘o’,
    //k[5]=13 or ‘\n’ (ASCII new line character),
    //k[6]=0 or ‘\0’ (null character – end of string)
```

Syntax => No Prefix

Syntax => Prefix 0x

Syntax => Prefix 0

Single quotes for character literals, or ASCII value with a backslash

Double quotes for Strings with null Terminator \n

Program Variables in C Programming

Definition:

A variable is an addressable storage location used to hold information that can be referenced and manipulated by the program.

Declaration:

Purpose: To specify the size, type, and name of the variable.

Example:

```
int x, y, z;  // Declares 3 variables of type "int" (integer)
char a, b;    // Declares 2 variables of type "char" (character)
```

Storage Allocation:

- **Registers:** Fast, limited storage for frequently accessed variables.
- **RAM:** Dynamic memory for variables that change during program execution.
- **ROM/Flash:** Permanent storage, typically for constants or read-only data.

Variable Declaration in C

Note 1: The Data type and the Value used to store in the Variable must match.

Note 2: All declaration statements must end with a semi-colon (;)

Basic syntax for variable declaring in C is as follows

data_type variable name = value;

Example:

```
int z = 35; // declare and initialize variable z with value 35.
```

The refined syntax for declaring variables in C can be quite comprehensive, incorporating storage classes, type qualifiers, type modifiers, data types, pointers, arrays, and initial values. Adding these parameter the syntax will look like

storage-class type-qualifier type-modifier data-type *pointer variable-name[size] = initial-value;

Example:

```
static const unsigned int *configFlagPtr = (int *)0x40021000;
```

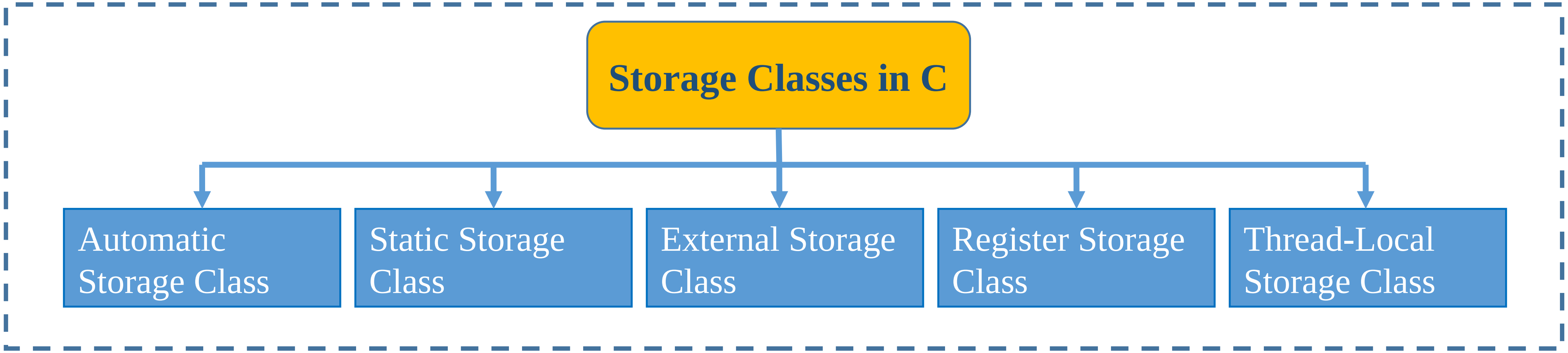
Storage-class, type-qualifier, type-modifier, pointer, array-size are all optional.

Storage Classes in C

In C programming, storage classes determine following characteristics of variables and functions.

1. **Scope:** Refers to variables or functions declared in another file or elsewhere in the same file.
2. **Lifetime:** Exists for the duration of the program.
3. **Visibility:** Visibility determines where a variable or function can be referenced within the program.
4. **Memory location:** The actual variable or function is defined elsewhere, usually in a different file.

Storage Classes control how variables are stored, accessed, and managed throughout the program. The key storage classes in C are:



Storage Classes in C: Automatic

Automatic Variable

- It is declared inside the function where it is used
- It are created when function is called and destroyed when the function is exited
- It is local to function and also called private variables
- It is also called as local or internal variables

Auto is Default storage class for all the local variables therefore, no need to use keyword auto

Example:

```
void function1 (void)
main()
{           int m =1000;
            function2();
            prinf("%d\n", m) }
Void function1(void)
{           int m =10;
            printf("%d"\n,m) }
```

Storage Classes in C: Automatic

Static Variable

- It persists at the function until the end of the program
- The keyword Static is used for declaration □ static int x;
- Static may be internal type or external type.
- Internal means it is declared inside the function
- The scope is up to end of the function
- It is used to retain the values between functions calls

Example:

```
void counterFunction() {
    static int count = 0;  // Static variable retains its
value between function calls
    count++;
    printf("Count: %d\n", count);
}
int main() {
    counterFunction();  // Output: Count: 1
    counterFunction();  // Output: Count: 2
    counterFunction();  // Output: Count: 3
    return 0;
}
```

Storage Classes in C

Scope

The scope of a variable or function refers to the region of the program where the variable or function can be accessed or used.

Types of Scope:

1. **Local Scope**: The region within a function or block where a variable or function is defined. Example: Variables declared inside a function or a block are local to that function or block.

Code Example:

```
void func() {
 int x = 10;  // x has local scope within func }
```

2. **Global Scope**: The region of the program where a variable or function is accessible throughout the entire program, typically from its point of declaration until the end of the file.

Example: Variables and functions declared outside of all functions.

Code Example:

```
int globalVar = 20;  // globalVar has global scope
void func() { // can use globalVar here }
```

Storage Classes in C

Visibility

Visibility determines where a variable or function can be referenced within the program. It specifies the extent to which a variable or function is accessible.

Types of Visibility:

1. **Internal Visibility:** Refers to variables or functions that are only accessible within the file they are declared. This is typically controlled using the static keyword.

Example:

```
static int internalVar = 30;  // Only visible within the same file
```

2. **External Visibility:** Definition: Refers to variables or functions that are accessible across different files. This is typically achieved using the extern keyword.

Example:

```
// File1.c
int externalVar = 40;  // Visible to other files
// File2.c
extern int externalVar;  // Reference to externalVar defined in File1.c
```

Storage Classes in C

Lifetime

The lifetime of a variable or function refers to the duration of time that the variable or function exists in memory and retains its value.

Types of Lifetime:

1. Automatic Lifetime: Variables with automatic lifetime are created when a function or block is entered and destroyed when it is exited. They are usually stored on the stack.

Example:

```
void func()
{ int autoVar = 50;  // Lifetime is limited to the duration of func }
```

2. Static Lifetime: Variables with static lifetime are created when the program starts and destroyed when the program ends. They retain their value between function calls or across files.

Example:

```
void func() {
static int staticVar = 60;  // Lifetime is the entire program duration }
```

3. Dynamic Lifetime: Variables with dynamic lifetime are allocated and deallocated manually using functions like malloc() and free(). Their lifetime is controlled by the programmer.

```
void func() {
int* dynamicVar = (int*)malloc(sizeof(int));  // Dynamic allocation
free(dynamicVar);  // Manual deallocation
}
```

Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- **Functions in C**
- Memory Layout in C
- Arrays
- Operators in C

Functions in C

- A function in C is a set of statements that when called perform some **specific task**.
- It is the basic building block of a C program that provides **modularity** and code **reusability**.
- The programming statements of a function are enclosed within { } braces, having certain meanings and performing certain operations.
- They are also called subroutines or procedures in other languages.

Syntax of Functions in C

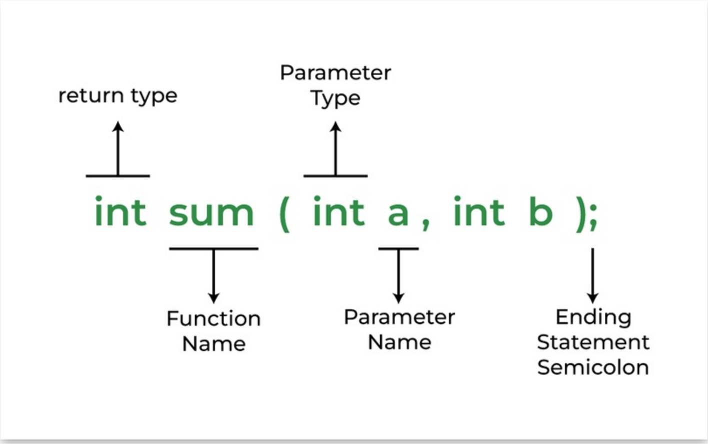
The syntax of function can be divided into 3 aspects:

- **Function Declaration**

```
return_type name_of_the_function (parameter_1, parameter_2);
```

Example:

```
int sum(int a, int b);  // Function declaration with parameter names
int sum(int , int);     // Function declaration without parameter names
```

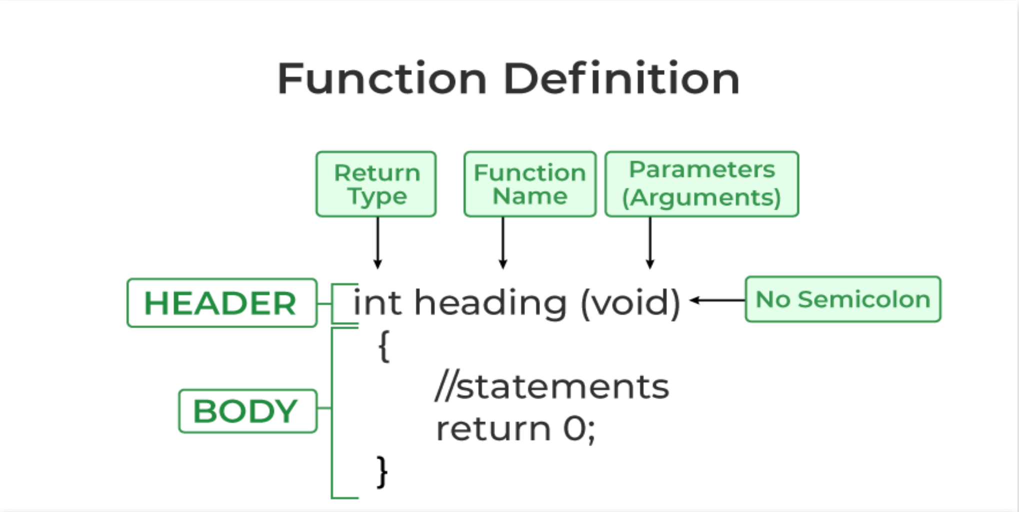


Functions in C

➢ **Function Definition**

The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

```
return_type function_name (parameter_1, parameter_2)
{
        // body of the function
}
```

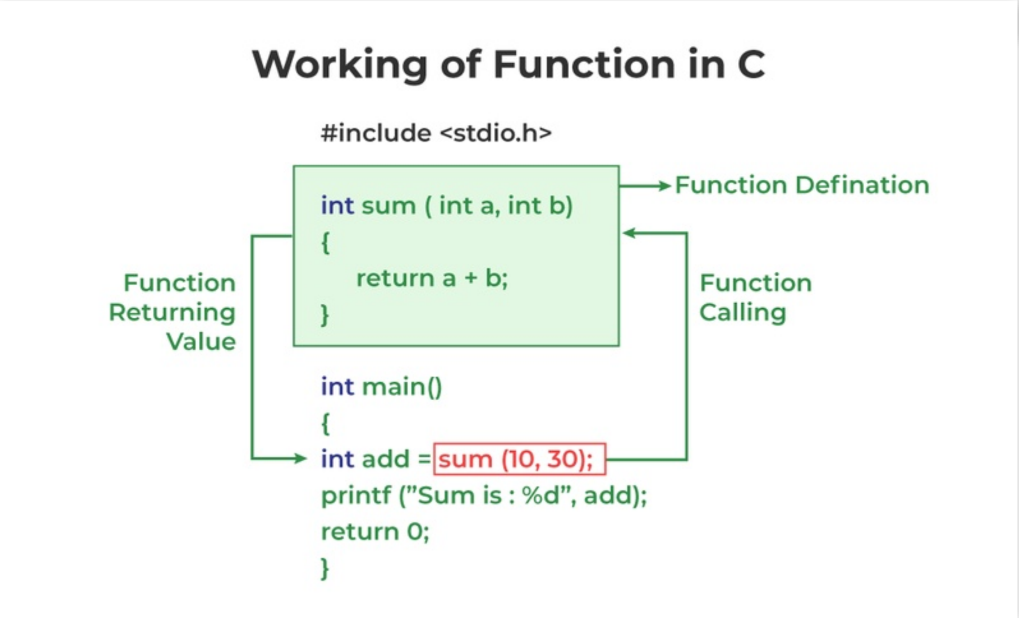


➢ **Function Calls**

A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

In the example,

- The first sum function is called and 10,30 are passed to the sum function.
- After the function call sum of a and b is returned and control is also returned back to the main function of the program.



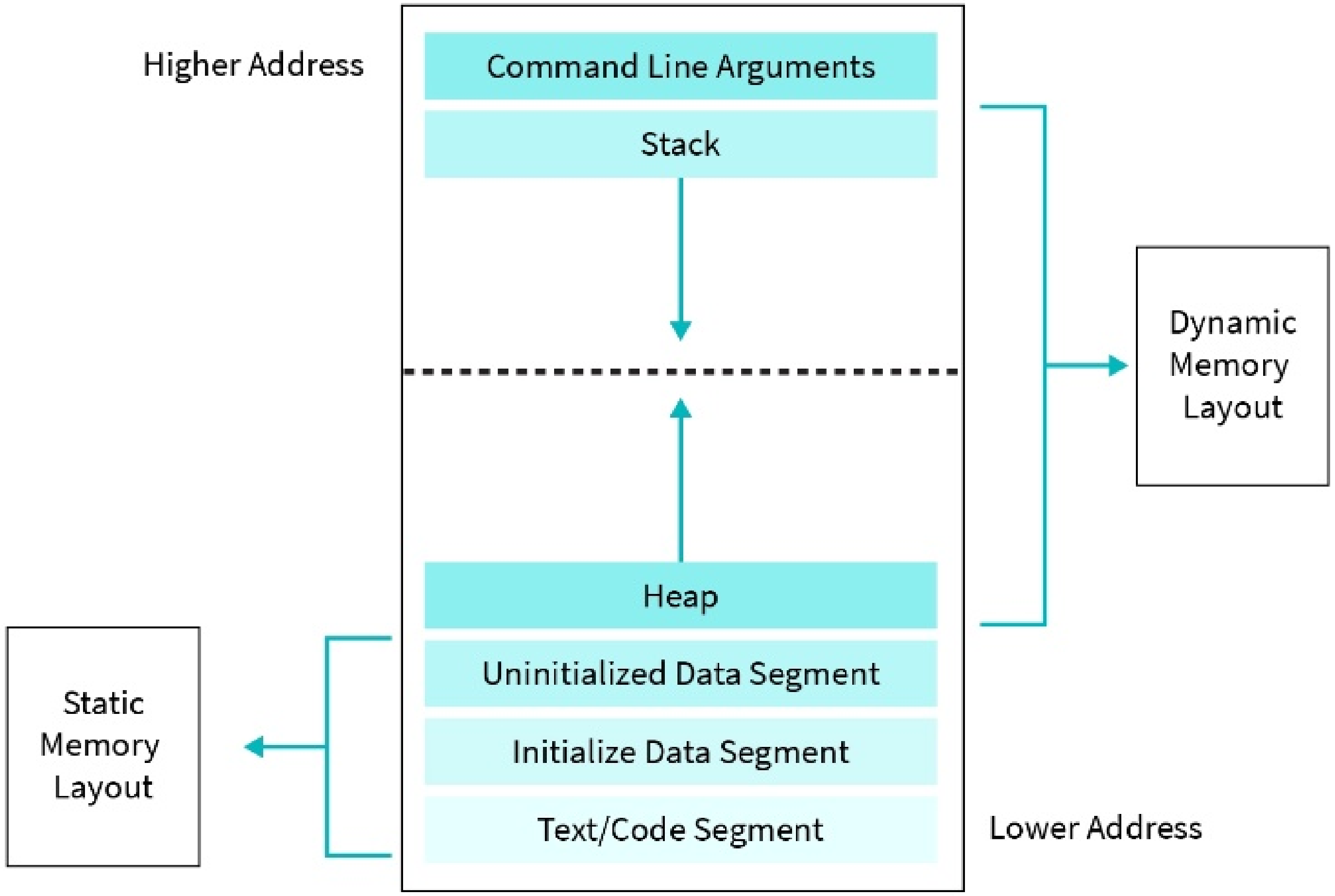
Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- Functions in C
- **Memory Layout in C**
- Arrays
- Operators in C

Memory Layout in C

A typical memory representation of a C program consists of the following sections.

- **Text/Code segment (i.e. instructions)**
- **Initialized data segment**
- **Uninitialized data segment (bss)**
- **Heap**
- **Stack**



Memory Layout

Memory Layout in C

Text/Code Segment

- A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.
- As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.
- The text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Example:

```
 int global_var = 5
// A function (text segment)
void print_message()
{
printf("Hello, World!\n");
}
int main()
{
print_message();            // Calls the function in the text segment return 0;
}
```

Segment	
Text Segment	void print_message() {...}
Initialized Data Segment	- int global_var = 5
Uninitialized Data Segment	-
Stack/Heap	-

Memory Layout in C Code

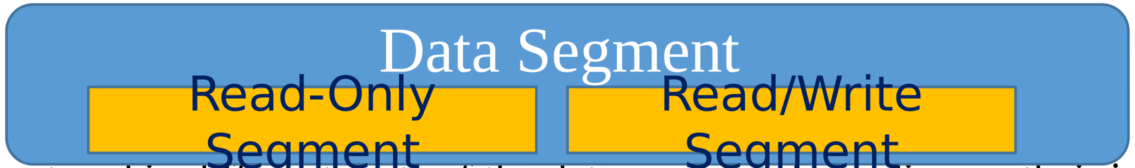
Initialized Data Segment

A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

```
int global_var = 10;    // Global variable initialized with 10
static int static_var = 20; // Static variable initialized with 20
```

Note that, the data segment is not read-only, since the values of the variables can be altered at run time.

Lifetime: Variables in the data segment exist for the lifetime of the program. They are initialized at program startup and persist until the program terminates.



In C, **variables** and **constants** are stored in different parts of the data segment depending on their initialization and attributes.

Type	Example	Memory Segment
Global Variable:	int debug = 1;	initialized read-write area
Global Constants:	const char* string = "hello world";	initialized read-only area
Global Static Variables	static int globalStatic = 20;	initialized read-write area
Static Variables in	void myFunction() {	initialized read-write

Memory Layout in C Code

Uninitialized Data Segment (bss)

- Also called the “BSS” segment (Block Started by Symbol).
- Contains global and static variables that are either:
  - Not explicitly initialized in the source code.
  - Initialized to zero.

Characteristics

- **Initialization:** The compiler initializes all variables in the BSS segment to zero before the program starts executing.
- **Memory Allocation:** The BSS segment occupies space in memory but does not store actual values; instead, it reserves space and initializes it to zero.
- **Memory Layout:** Comes after the initialized data segment in memory.

Examples:

```
static int i; // Static variables uninitialized
int j;        // Global variables uninitialized
```

Memory Layout in C Code

Stack:

- The stack is a region of memory that stores temporary data, following a Last In, First Out (LIFO) structure.
- Traditionally, it adjoined the heap and grew in the opposite direction.

Characteristics:

- **Memory Layout:**
  - The stack is typically located in the higher parts of memory and grows towards lower addresses.
  - In modern systems with large address spaces and virtual memory, the stack and heap can be placed almost anywhere, but they still generally grow in opposite directions.

- **Stack Pointer:**
  - A stack pointer register keeps track of the top of the stack.
  - Adjusted each time a value is pushed onto or popped from the stack.

- **Stack Frame:** The data associated with a function call is stored in a stack frame.

  Stack Frame at minimum includes:
  - Return Address: Address to return to after the function call is complete.
  - May also include local variables, function parameters, etc.

Memory Layout in C Code

The **size command** is used to check the sizes (in bytes) of these different memory segments.

Simple Program

```
#include<stdio.h>

int main() {
    return 0;
}
```

```
~$ gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     544       8    1970     7b2 file_1
```

Adding one global variable in program

```
#include<stdio.h>

int global_variable = 5;

int main() {
    return 0;
}
```

```
~$ gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     548       4    1970     7b2 file_1
```

Adding one global variable increased memory allocated by data segment (Initialized data segment) by 4 bytes, which is the actual memory size of 1 variable of type integer (sizeof(global_variable)).

Task: Day 3 Embedded C Programming

Prepare a brief report explaining the operation of stack memory with respect to function calls and the phenomenon of stack overflow.

Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- **Functions in C**
- Memory Layout in C
- Arrays
- Operators in C

Arrays in C Programming

Definition:

An array is a collection of data elements stored in consecutive memory locations. The array begins at a named address and contains a fixed number of elements.

---------------------------One-Dimensional Arrays----------------------------------

Declaration:

Syntax: type arrayName[size];

Example Code:

```
int n[5];          // Declares an array of 5 integers
n[3] = 5;     // Sets the value of the 4th element (index 3) to 5
```

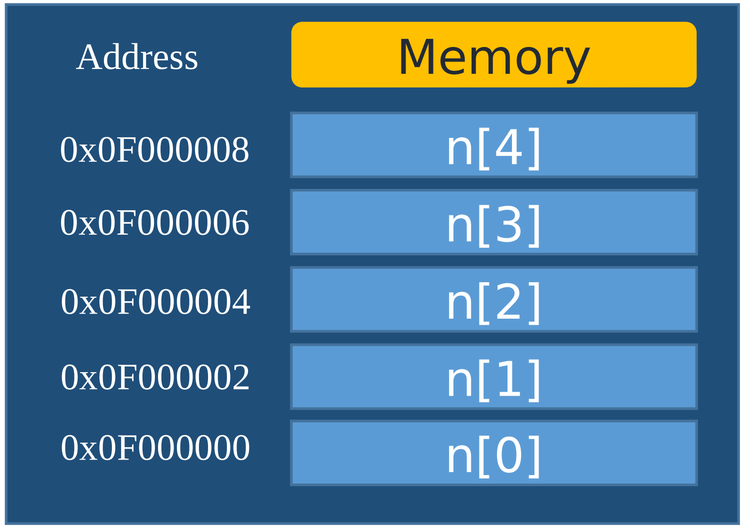
Array Indexing:

Indices: Start from 0 to N-1 where N is the number of elements.

Element Access: Access elements using arrayName[index].

Memory Layout: Array Elements: n[0] | n[1] | n[2] | n[3] | n[4]

Address	Value
A= (base Address)	n[0]
A+2	n[1]
A+4	n[2]
A+6	n[3]
A+8	n[4]



Arrays in C Programming

Address	Value
A(base Address)	n[0][0]
A+2	n[0] [1]
A+4	n[0] [2]
A+6	n[0] [3]
A+8	n[1] [0]
A+10	n[1] [1]
A+12	n[1] [2]

--------------------------Two-Dimensional Arrays---------------------------------

Declaration:

Syntax: type arrayName [rows][columns];

Example Code:

```
int matrix[3][4];  // Declares a 2D array with 3 rows and 4 columns
matrix[1][2] = 7;  // Sets the value of the element in the 2nd row and 3rd column to 7
```

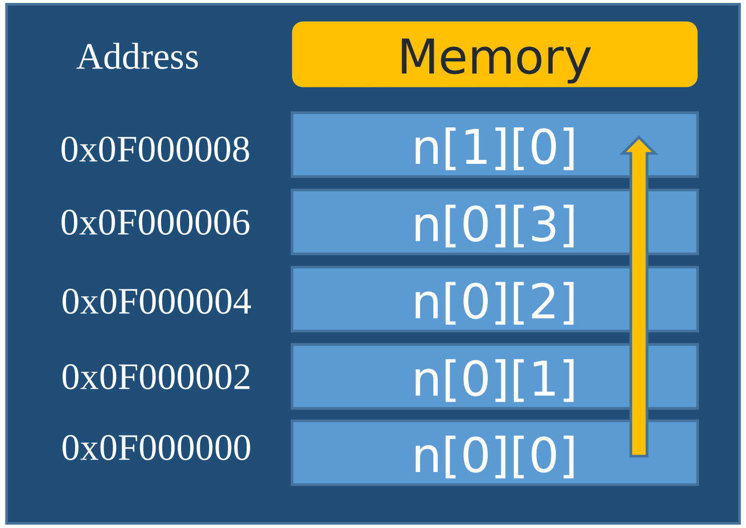
Array Indexing:

Indices: Start from 0,0 to N-1,N-1 where N is the number of elements.

Element Access: Access elements using arrayName[index].

Memory Layout: Array Elements (for matrix[3][4]):

matrix[0][0] | matrix[0][1] | matrix[0][2] | matrix[0][3]
matrix[1][0] | matrix[1][1] | matrix[1][2] | matrix[1][3]
matrix[2][0] | matrix[2][1] | matrix[2][2] | matrix[2][3]



Operators in C

An operator in C can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

Types of Operators in C

C language provides a wide range of operators that can be classified

into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

<table>
<tr><th>OPERATOR</th><th>TYPE</th><th>ASSOCIAVITY</th></tr>
<tr><td>( ) [ ] . -></td><td></td><td>left-to-right</td></tr>
<tr><td>++ -- + - ! ~ (type) * & sizeof</td><td>Unary Operator</td><td>right-to-left</td></tr>
<tr><td>* / %</td><td>Arithmetic Operator</td><td>left-to-right</td></tr>
<tr><td>+ -</td><td>Arithmetic Operator</td><td>left-to-right</td></tr>
<tr><td><< >></td><td>Shift Operator</td><td>left-to-right</td></tr>
<tr><td>< <= > >=</td><td>Relational Operator</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>Relational Operator</td><td>left-to-right</td></tr>
<tr><td>&</td><td>Bitwise AND Operator</td><td>left-to-right</td></tr>
<tr><td>^</td><td>Bitwise EX-OR Operator</td><td>left-to-right</td></tr>
<tr><td>|</td><td>Bitwise OR Operator</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>Logical AND Operator</td><td>left-to-right</td></tr>
<tr><td>||</td><td>Logical OR Operator</td><td>left-to-right</td></tr>
<tr><td>? :</td><td>Ternary Conditional Operator</td><td>right-to-left</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>Assignment Operator</td><td>right-to-left</td></tr>
<tr><td>,</td><td>Comma</td><td>left-to-right</td></tr>
</table>

Arithmetic Operators

The arithmetic operators are used to perform arithmetic/mathematical operations on operands.

There are 9 arithmetic operators in C language:

S. No.	Symbol	Operator	Description	Syntax
1	**+**	**Plus**	Adds two numeric values.	**a + b**
2	**–**	**Minus**	Subtracts right operand from left operand.	**a – b**
3	*****	**Multiply**	Multiply two numeric values.	**a * b**
4	**/**	**Divide**	Divide two numeric values.	**a / b**
5	**%**	**Modulus**	Returns the remainder after diving the left operand with the right operand.	**a % b**
6	**+**	**Unary Plus**	Used to specify the positive values.	**+a**
7	**–**	**Unary Minus**	Flips the sign of the value.	**-a**
8	**++**	**Increment**	Increases the value of the operand by 1.	a++
9	**—**	**Decrement**	Decreases the value of the operand by 1.	a–

Arithmetic Operators Example

Example Code:

```
int main()
{
        int a = 25, b = 5;
 // using operators and printing results
    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 = %d\n", +a);
    printf("-a = %d\n", -a);
    printf("a++ = %d\n", a++);
    printf("a-- = %d\n", a--);

    return 0;
}
```

Output

```
a + b = 30
a - b = 20
a * b = 125
a / b = 5
a % b = 0
+a = 25
-a = -25
a++ = 25
a-- = 26
```

Relational Operators in C

Relational Operators in C

The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison.

These are a total of 6 relational operators in C:

S. No.	Symbol	Operator	Description	Syntax
1	**<**	**Less than**	Returns true if the left operand is less than the right operand. Else false	**a < b**
2	**>**	**Greater than**	Returns true if the left operand is greater than the right operand. Else false	**a > b**
3	**<=**	**Less than or equal to**	Returns true if the left operand is less than or equal to the right operand. Else false	**a <= b**
4	**>=**	**Greater than or equal to**	Returns true if the left operand is greater than or equal to right operand. Else false	**a >= b**
5	**==**	**Equal to**	Returns true if both the operands are equal.	**a == b**
6	**!=**	**Not equal to**	Returns true if both the operands are NOT equal.	**a != b**

Relational Operators in C

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

S. No.	Symbol	Operator	Description	Syntax
1	**&&**	**Logical AND**	Returns true if both the operands are true.	**a && b**
2	**||**	**Logical OR**	Returns true if both or any of the operand is true.	**a || b**
3	**!**	**Logical NOT**	Returns true if the operand is false.	**!a**

Example

```
int main()
{
        int a = 25, b = 5;
        // using operators and printing
        results
            printf("a && b : %d\n", a && b);
            printf("a || b : %d\n", a || b);
            printf("!a: %d\n", !a);
        return 0;}
```

Output

```
a && b : 1
a || b : 1
!a: 0
```

Relational Operators Example

```
int main()
{

    int a = 25, b = 5;

    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);

    return 0;
}
```

Output

```
a < b  : 0
a > b  : 1
a <= b: 0
a >= b: 1
a == b: 0
a != b : 1
```

Bitwise Operators in C

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing. There are 6 bitwise operators in C:

<table>
<tr><th>S. No.</th><th>Symbol</th><th>Operator</th><th>Description</th><th>Syntax</th></tr>
<tr><td>1</td><td>&</td><td>Bitwise AND</td><td>Performs bit-by-bit AND operation and returns the result.</td><td>a & b</td></tr>
<tr><td>2</td><td>|</td><td>Bitwise OR</td><td>Performs bit-by-bit OR operation and returns the result.</td><td>a | b</td></tr>
<tr><td>3</td><td>^</td><td>Bitwise XOR</td><td>Performs bit-by-bit XOR operation and returns the result.</td><td>a ^ b</td></tr>
<tr><td>4</td><td>~</td><td>Bitwise First Complement</td><td>Flips all the set and unset bits on the number.</td><td>~a</td></tr>
<tr><td>5</td><td><<</td><td>Bitwise Leftshift</td><td>Shifts the number in binary form by one place in the operation and returns the result.</td><td>a << b</td></tr>
<tr><td>6</td><td>>></td><td>Bitwise Rightshilft</td><td>Shifts the number in binary form by one place in the operation and returns the result.</td><td>a >> b</td></tr>
</table>

Bitwise Operators: AND, OR, XOR, ~

```
C = A & B;          A   0 1 1 0 0 1 1 0
(AND)               B   1 0 1 1 0 0 1 1
                        ---------------
                    C   0 0 1 0 0 0 1 0

C = A | B;          A   0 1 1 0 0 1 0 0
(OR)                B   0 0 0 1 0 0 0 0
                        ---------------
                    C   0 1 1 1 0 1 0 0

C = A ^ B;          A   0 1 1 0 0 1 0 0
(XOR)               B   1 0 1 1 0 0 1 1
                        ---------------
                    C   1 1 0 1 0 1 1 1

B = ~A;             A   0 1 1 0 0 1 0 0
                        ---------------
(COMPLEMENT)        B   1 0 0 1 1 0 1 1
```

Bitwise Operators: Bit Masking

```
C = A & 0xFE;     A     a b c d e f g h
               0xFE     1 1 1 1 1 1 1 0
                  C     a b c d e f g 0
```

Clear selected bit of A

```
C = A & 0x01;     A     a b c d e f g h
               0xFE     0 0 0 0 0 0 0 1
                  C     0 0 0 0 0 0 0 h
```

Clear all but the selected bit of A

```
C = A | 0x01;     A     a b c d e f g h
               0x01     0 0 0 0 0 0 0 1
                  C     a b c d e f g 1
```

Set selected bit of A

```
C = A ^ 0x01;     A     a b c d e f g h
              0x01      0 0 0 0 0 0 0 1
                  C     a b c d e f g h'
```

Complement selected bit of A

Bitwise Operators: Shift Operator

```
B = A << 3;                A   1 0 1 0 1 1 0 1
(Left shift 3 bits)        B   0 1 1 0 1 0 0 0

B = A >> 2;                A   1 0 1 1 0 1 0 1
(Right shift 2 bits)       B   0 0 1 0 1 1 0 1

B = '1';                   B = 0 0 1 1 0 0 0 1  (ASCII 0x31)
C = '5';                   C = 0 0 1 1 0 1 0 1  (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
      (B << 4)         = 0 0 0 1 0 0 0 0
     (C & 0x0F)        = 0 0 0 0 0 1 0 1
             D         = 0 0 0 1 0 1 0 1  (Packed BCD 0x15)
```

Lab Task: LED Follower Logic

Generate a code to Print a number in binary and decimal format, then apply left shift operator 3 times then print number in binary and decimal

Bitwise Operators Example

```
int main()
{

    int a = 25, b = 5;

    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);

    return 0;
}
```

Output

```
a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800
```

Assignment Operators in C

S. No.	Symbol	Operator	Description	Syntax
1	=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
2	+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
3	-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b
4	*=	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	a *= b
5	/=	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	a /= b
6	%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b
7	&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
8	|=	OR and assign	Performs bitwise OR and assigns this value to the left operand.	a |= b
9	^=	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	a ^= b
10	>>=	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	a >>= b
		Leftshift and	Performs bitwise Leftshift and assign this value to the left	

Assignment Operators Example

```
int main()
{
    int a = 25, b = 5;

    // using operators and printing results
    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);
    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: 5
a += b: 10
a -= b: 5
a *= b: 25
a /= b: 5
a %= b: 0
a &= b: 0
a |= b: 5
a >>= b: 0
```