1. How is C represent and store characters? What are the special characters? Write a program to describe the relationship between ASCII and characters.
   sIon:
   In C, characters are represented using the **ASCII** encoding, where each character is stored as a **numeric value**. The char data type in C typically stores a single byte (8 bits), which can represent 256 different values (0-255).

   There are two type character:
   a. Printable (Letters, digits, punctuation marks, and special symbols.)
   b. Non-printable (newline, tab etc)

   List of some special character:

Escape Sequence	**Character Description**	**ASCII Value**	**Explaination**
\a	Bell (alert)	7	Causes the system to beep or make a sound (depends on the system).
\n	Newline	10	Moves the cursor to the next line (used to break the line).
\t	Horizontal tab	9	Moves the cursor to the next tab stop (typically 8 spaces).
\v	Vertical tab	11	Moves the cursor down to the next vertical tab stop (rarely used).
\f	Form Feed	12	Moves the cursor to the beginning of the next page (rarely used).
\r	Carriage Return	13	Moves the cursor to the beginning of the line.
\b	Backspace	8	Moves the cursor one character back.
\0	Null (end of string)	0	Mark the end of a string in C

\’	Single Quote	39	Represents the single quote (') character
\”	Double Quote	34	Represents the double quote (") character.
\\	Backslash	92	Represents the backslash (\ ) character.

Programm to show relation between ASCII value and character:

code:

```
#include <stdio.h>

int main() {
    char ch;

    // Display ASCII values for printable characters

    printf("ASCII values for characters A to Z:\n");
    for (ch = 'A'; ch <= 'Z'; ch++) {
        printf("Character: %c, ASCII: %d\n", ch, ch);
    }

    printf("\nASCII values for lowercase characters a to z:\n");
    for (ch = 'a'; ch <= 'z'; ch++) {
        printf("Character: %c, ASCII: %d\n", ch, ch);
    }

    printf("\nASCII values for digits 0 to 9:\n");
    for (ch = '0'; ch <= '9'; ch++) {
        printf("Character: %c, ASCII: %d\n", ch, ch);
    }

    printf("\nSpecial characters and their ASCII values:\n");
    printf("Character: '\\a' (Bell), ASCII: %d\n", '\a');
    printf("Character: '\\n' (Newline), ASCII: %d\n", '\n');
    printf("Character: '\\t' (Tab), ASCII: %d\n", '\t');
    printf("Character: '\\v' (Vertical Tab), ASCII: %d\n", '\v');
```

```
    printf("Character: '\\f' (Form Feed), ASCII: %d\n", '\f');
    printf("Character: '\\r' (Carriage Return), ASCII: %d\n", '\r');
    printf("Character: '\\b' (Backspace), ASCII: %d\n", '\b');
    printf("Character: '\\0' (Null), ASCII: %d\n", '\0');
    printf("Character: '\\'' (Single Quote), ASCII: %d\n", '\'');
    printf("Character: '\\\"' (Double Quote), ASCII: %d\n", '\"');
    printf("Character: '\\\\' (Backslash), ASCII: %d\n", '\\');

    return 0;
}
```

Output:

```
ASCII values for characters A to Z:
Character: A, ASCII: 65
Character: B, ASCII: 66
Character: C, ASCII: 67
Character: D, ASCII: 68
Character: E, ASCII: 69
Character: F, ASCII: 70
Character: G, ASCII: 71
Character: H, ASCII: 72
Character: I, ASCII: 73
Character: J, ASCII: 74
Character: K, ASCII: 75
Character: L, ASCII: 76
Character: M, ASCII: 77
Character: N, ASCII: 78
Character: O, ASCII: 79
Character: P, ASCII: 80
Character: Q, ASCII: 81
Character: R, ASCII: 82
Character: S, ASCII: 83
Character: T, ASCII: 84
Character: U, ASCII: 85
Character: V, ASCII: 86
Character: W, ASCII: 87
Character: X, ASCII: 88
Character: Y, ASCII: 89
Character: Z, ASCII: 90

ASCII values for lowercase characters a to z:
Character: a, ASCII: 97
Character: b, ASCII: 98
Character: c, ASCII: 99
```

Character: d, ASCII: 100
Character: e, ASCII: 101
Character: f, ASCII: 102
Character: g, ASCII: 103
Character: h, ASCII: 104
Character: i, ASCII: 105
Character: j, ASCII: 106
Character: k, ASCII: 107
Character: l, ASCII: 108
Character: m, ASCII: 109
Character: n, ASCII: 110
Character: o, ASCII: 111
Character: p, ASCII: 112
Character: q, ASCII: 113
Character: r, ASCII: 114
Character: s, ASCII: 115
Character: t, ASCII: 116
Character: u, ASCII: 117
Character: v, ASCII: 118
Character: w, ASCII: 119
Character: x, ASCII: 120
Character: y, ASCII: 121
Character: z, ASCII: 122

ASCII values for digits 0 to 9:
Character: 0, ASCII: 48
Character: 1, ASCII: 49
Character: 2, ASCII: 50
Character: 3, ASCII: 51
Character: 4, ASCII: 52
Character: 5, ASCII: 53
Character: 6, ASCII: 54
Character: 7, ASCII: 55
Character: 8, ASCII: 56
Character: 9, ASCII: 57

Special characters and their ASCII values:
Character: 'a' (Bell), ASCII: 7
Character: '\n' (Newline), ASCII: 10
Character: '\t' (Tab), ASCII: 9
Character: '\v' (Vertical Tab), ASCII: 11
Character: '\f' (Form Feed), ASCII: 12
Character: '\r' (Carriage Return), ASCII: 13
Character: '\b' (Backspace), ASCII: 8

Character: '\0' (Null), ASCII: 0
Character: '\'' (Single Quote), ASCII: 39
Character: '\"' (Double Quote), ASCII: 34
Character: '\\' (Backslash), ASCII: 92

2. How to define and represent different bases in C? Write code to illustrate
   soln:
   In C integers can represent in different bases.

Base	**Prefix**	**Example**	**Decimal Equivalent**	**Description**
Decimal	(none)	100	100	Default representation
Octal	0	0144	100	Leading 0 indicates octal (base 8)
Hexadecimal	0x or 0X	0x64	100	Leading 0x or 0X indicates hexadecimal (base 16).
Binary	0b (since C++14, GCC/Clang extension).	0b1100100	100	Binary literals are supported since C++14, GCC/Clang extension

example code :

```
#include <stdio.h>

int main() {
    int decimal = 100;              // Decimal (Base 10)
    int octal = 0144;               // Octal (Base 8) -> 0144 in octal is 100 in decimal
    int hexadecimal = 0x64;       // Hexadecimal (Base 16) -> 0x64 is 100 in decimal
```

```
    int binary = 0b1100100;       // Binary (Base 2) -> 0b1100100 is 100 in decimal
(GCC/Clang extension)

    printf("Decimal: %d\n", decimal);
    printf("Octal: %o (Decimal Equivalent: %d)\n", octal, octal);
    printf("Hexadecimal: %x (Decimal Equivalent: %d)\n", hexadecimal, hexadecimal);

    #ifdef __GNUC__             // Check if using GCC/Clang (supports binary literals)
        printf("Binary: %d\n", binary);
    #else
        printf("Binary literals are not supported in standard C.\n");
    #endif

    return 0;
}
```

3. How to define and use enumeration? Write a piece of code to describe

slon:

An enumeration (enum) is a user-defined type in C that consist **set of integer constants.**

It improves readability by replacing numeric value with meaningful name.

example code:

```
#include <stdio.h>

// Define an enumeration
enum Color {
    RED,          // 0
    GREEN,      // 1
    BLUE          // 2
};

enum Days {
    SUNDAY = 1,
    MONDAY,              // auto increment 2
    TUESDAY,             // auto increment 3
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
```

```
    SATURDAY
};



int main() {

        printf(" %d\n", BLUE);
        printf(" %d\n", MONDAY);

    return 0;
}
```

4. Why do we sometimes say that string is constant? Illustration

soln:

In C, string literals (e.g., "hello") are stored in read-only memory (.rodata section). This means that modifying them is not allowed, which is why we say a string is constant.

example:

```
#include <stdio.h>

int main() {
    char *str = "hello";  // String literal stored in read-only memory
    str[0] = 'H';              //  Undefined behavior (Segmentation Fault)

    return 0;
}
```

It outputs show segmentation fault (SIGSEGV)

But using array instead of a pointer, the string is modifiable.

example:

```
#include <stdio.h>

int main() {
    char str[] = "hello";  // Stored in writable memory (stack)
    str[0] = 'H';          //  Allowed

    printf("%s\n", str);
    return 0;
}
```

Output :
Hello

Summary:

Declaration	Stored In	Modifiable	Example
char *str = "hello";	Read-Only .rodata	**No** (Causes SegFault)	str[0] = 'H'; (Error)
char str[] = "hello";	Stack (Writable)	**Yes** (Safe)	str[0] = 'H'; (Works)

5. Is it possible to change the value of a const variable? How to do?
   soln:

   By definition, a `const` variable **should not be modified**. However, there are ways to **bypass** this restriction using techniques such as **pointers**, **type casting**.

   1. using pointer:

```
#include <stdio.h>

int main() {
    const int x = 10;
    int *ptr = (int *)&x;  // Cast away constness
    *ptr = 20;  // Modify x

    printf("x = %d\n", x);  // Undefined behavior (may or may not change)
    return 0;
}
```

      Output:
      x = 20 (if compiler allows modification)

   2. If a const variable is **global**, it is usually stored in the **read-only section** (.rodata) of memory.However, using a pointer, it might be modified (if it's stored in .data instead of .rodata).

```
#include <stdio.h>

const int y = 30;  // May be stored in read-only memory

int main() {
    int *ptr = (int *)&y;  // Cast away const
    *ptr = 50;  // Attempt to modify

    printf("y = %d\n", y);
    return 0;
}
```

In most system it shows segmentation fault.

6. Why do we need keyword static to declare a variable?
   soln:

   1. **Preserving variable values across function calls**.

```
#include <stdio.h>

void counter() {
    static int count = 0;            // Stored in the data segment (not stack)
    count++;                          // Value is preserved between function calls
    printf("Count: %d\n", count);
}

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

Output:
Count: 1
Count: 2
Count: 3

2. By default, global variables are **accessible from any file** in a multi-file program. Using `static` makes a global variable **only accessible in the file where it is declared** (internal linkage).

   File file1.c:

   ```
   #include <stdio.h>

   static int x = 10;          // x is only accessible inside file1.c

   void printX() {
      printf("x = %d\n", x);
   }
   ```
   File: file2.c

   ```
   #include <stdio.h>

   extern int x;               // Error: x is static in file1.c, so it cannot be accessed here

   int main() {
      printf("%d\n", x);     // This will cause a linking error
      return 0;
   }
   ```

3. **Allocating memory in the data segment instead of the stack** (for efficiency and lifetime control).

7. Where are the following variables stored in the memory layout? Provide the evidence
soln:

variable	Type	storage location
int a = 1;	Initialized Global	Initialized Data
int b;	Uninitialized Global	BSS(Uninitialized Data)
const int c = 3;	Constant global	Initialized Data
static int d = 4;	static global	Initialized Data
char *str1 = "hello_world";	Pointer(global)	Initialized Data
int aa = 1;	Initialized local	stack
int bb;	Uninitialized local	stack
const int cc = 3;	Constant local	stack
static int dd = 4;	static local	Initialized data
char *str2 = "hello_world";	Pointer(local)	stack

Evidence:

```
PS C:\Users\Admin\Documents\bdcom_coding_zone> objdump -h test.exe

test.exe:     file format pei-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00001178  0000000140001000  0000000140001000  00000600  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         000000a0  0000000140003000  0000000140003000  00001800  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000880  0000000140004000  0000000140004000  00001a00  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .pdata        00000210  0000000140005000  0000000140005000  00002400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .xdata        000001a4  0000000140006000  0000000140006000  00002800  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .bss          000001a0  0000000140007000  0000000140007000  00000000  2**4
                  ALLOC
  6 .idata        000004f0  0000000140008000  0000000140008000  00002a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .CRT          00000060  0000000140009000  0000000140009000  00003000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .tls          00000010  000000014000a000  000000014000a000  00003200  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .reloc        0000007c  000000014000b000  000000014000b000  00003400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .debug_aranges 00000080  000000014000c000  000000014000c000  00003600  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_info   000011db  000000014000d000  000000014000d000  00003800  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .debug_abbrev 0000013f  000000014000f000  000000014000f000  00004a00  2**0
                  CONTENTS, READONLY, DEBUGGING
 13 .debug_line   000000fd  0000000140010000  0000000140010000  00004c00  2**0
                  CONTENTS, READONLY, DEBUGGING
 14 .debug_frame  00000088  0000000140011000  0000000140011000  00004e00  2**0
                  CONTENTS, READONLY, DEBUGGING
 15 .debug_str    00000053  0000000140012000  0000000140012000  00005000  2**0
                  CONTENTS, READONLY, DEBUGGING
 16 .debug_line_str 0000016d  0000000140013000  0000000140013000  00005200  2**0
                  CONTENTS, READONLY, DEBUGGING
```