# **SOFTWARE ENGINEERING ASSIGNMENTS**

Module-2

Theory Exercise

**Introduction to Programming**

**Over View of C programming**

**1.Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.**

**-> Introduction of C**

C programming is one of the most influential and widely used programming languages in the history of computer science. Developed in the early 1970s, C has played a key role in the development of modern software and operating systems. Its simplicity, efficiency, and close interaction with hardware make it a timeless language in the programming world.

* **History and Evolution**
* · **Origin (Early 1970s)**:  
  C was developed by **Dennis Ritchie** at **Bell Labs** in **1972**. It evolved from earlier languages like **B** and **BCPL (Basic Combined Programming Language)**. The main goal was to create a language for writing operating systems.
* · **Unix and C**:  
  One of C's biggest milestones was that it was used to rewrite the **UNIX operating system**, which was originally written in assembly language. This made UNIX portable, faster to develop, and easier to maintain.
* · **Standardization (1980s)**:  
  In the mid-1980s, the **American National Standards Institute (ANSI)** established a standard version of C, known as **ANSI C** or **C89**. Later versions like **C99**, **C11**, and **C18** added more features like inline functions, variable-length arrays, and better support for multi-threading.

**Importance of C Programming**

1. Foundation for Many Languages:  
   C has influenced many other popular languages like C++, Java, C#, Objective-C, and even Python. Learning C helps understand how these languages work at a deeper level.
2. System-Level Programming:  
   C allows direct manipulation of memory using pointers, making it ideal for writing operating systems, device drivers, and embedded software.
3. High Performance:  
   C code is fast and efficient because it is compiled directly to machine code and gives programmers low-level control over system resources.
4. Portability:  
   Programs written in C can run on different machines with little or no modification, making it perfect for cross-platform development.

**Why C is Still Used Today**

* Despite being over 50 years old, C remains relevant due to:
* **Embedded Systems**:  
  Most embedded devices (like washing machines, routers, sensors) use C due to its speed and efficiency.
* **Operating Systems**:  
  Windows, Linux, macOS, and Android have large portions written in C.
* **Education**:  
  C is widely taught in computer science courses because it helps students understand computer memory, logic, and data structures.
* **Open Source Projects**:  
  Many popular open-source tools and systems like **Git**, **Linux Kernel**, and **MySQL** are written in C.

**Setting Up Environment**

1. **Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like Dev C++, VS Code, or Code Blocks.**

* Step 1: Install GCC Compiler

>Go to the official website: https://jmeubank.github.io/tdm-gcc/

> Download the installer (e.g., tdm-gcc-10.3.0.exe)

> Run the installer and choose:

* "Create" for a new installation
* Select “MinGW/GCC” compiler suite
* Proceed with default settings unless specific needs

> Wait for installation to complete

> Done! GCC is now installed.

Step 2: Choose and Install an IDE

-Option A: Dev-C++ (Easy & Beginner Friendly)

-Option B: Code::Blocks (IDE with built-in GCC)

-option C: Visual Studio Code (Powerful for future use)

* C language “hello world” program.

#include <stdio.h>

int main(){

printf(“hello world”);

}

**Basic Structure of a C Program**

1. **Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.**

1. Header Files

-> These are included at the beginning of a C program using #include. They allow access to standard functions (like printf, scanf etc.).

2. Main Function

-> The main() function is the entry point of every C program. The execution starts from here.

3. Comments

-> Used to explain the code. They are ignored during execution.

-> Single-line comment: // comment here

-> Multi-line comment:\*/ comment here /\*

4. Data Types

* Define the type of data a variable can store.

| **Data Type** | **Description** | **Example Values** |
| --- | --- | --- |
| int | Integer | 10, -5, 0 |
| float | Floating point | 3.14, -2.5 |
| char | Character | 'A', 'z' |
| double | Double-precision float | 3.14159 |

5. Variables

-> Used to store data. You must declare the data type before using a variable.

Example code :

#include <stdio.h> // Header file

int main() { // Main function

// Variable declaration

int age = 18;

float marks = 89.5;

char grade = 'A';

// Printing values

printf("Age: %d\n", age);

printf("Marks: %.2f\n", marks);

printf("Grade: %c\n", grade);

return 0; // End of program

}

**Operators in C**

1. **Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.**

**1. Arithmetic Operators**

Used to perform basic mathematical operations.

| **Operator** | **Description** | **Example (a=10, b=5)** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | 15 |
| - | Subtraction | a - b | 5 |
| \* | Multiplication | a \* b | 50 |
| / | Division | a / b | 2 |
| % | Modulus (remainder) | a % b | 0 |

**2. Relational Operators**

Used to compare two values. Returns either **true (1)** or **false (0)**.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | 0 |
| != | Not equal to | a != b | 1 |
| > | Greater than | a > b | 1 |
| < | Less than | a < b | 0 |
| >= | Greater or equal to | a >= b | 1 |
| <= | Less or equal to | a <= b | 0 |

**3. Logical Operators**

Used to combine multiple relational expressions.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | (a > 5 && b < 10) | 1 |
| ` |  | ` | Logical OR |
| ! | Logical NOT (negation) | !(a == b) | 1 |

**4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Description** | **Example** | **Equivalent To** |
| --- | --- | --- | --- |
| = | Simple assignment | a = 10 | — |
| += | Add and assign | a += 5 | a = a + 5 |
| -= | Subtract and assign | a -= 3 | a = a - 3 |
| \*= | Multiply and assign | a \*= 2 | a = a \* 2 |
| /= | Divide and assign | a /= 4 | a = a / 4 |
| %= | Modulus and assign | a %= 2 | a = a % 2 |

**5. Increment/Decrement Operators**

Used to increase or decrease a variable by 1.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| ++ | Increment (prefix/postfix) | ++a or a++ | a = a + 1 |
| -- | Decrement (prefix/postfix) | --a or a-- | a = a - 1 |

**6. Bitwise Operators**

Work on bits and perform bit-by-bit operations.

| **Operator** | **Description** | **Example (a=5, b=3)** | **Result** |
| --- | --- | --- | --- |
| & | AND | a & b (0101 & 0011) | 1 |
| ` | ` | OR | `a |
| ^ | XOR | a ^ b | 6 |
| ~ | NOT (1's comp.) | ~a | -6 |
| << | Left shift | a << 1 | 10 |
| >> | Right shift | a >> 1 | 2 |

**7. Conditional (Ternary) Operator**

Used as a shorthand for if-else.

c

Copy code

condition ? expr1 : expr2;

**Control Flow Statements in C**

1. **Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.**

* Decision making statements in c allow the program to execute certain parts of code based on specific conditions. These are crucial for implementing logic in a program.
* **1.if statement**

-Used to execute a block of code only if a specified condition is true.

Syntax:

if (condition) {

// Code to execute if condition is true

}

Example:

int num = 10;

if (num > 5) {

printf("Number is greater than 5\n");

}

**2. if-else statement**

-Execute one block of code if the condition is true, another block if the condition is false.

Syntax:

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

Example:

int num = 3;

if (num % 2 == 0) {

printf("Even number\n");

} else {

printf("Odd number\n");

}

**3. Nested if-else Statement:**

An if or else if statement inside another if or else.

**Syntax:**

if (condition1) {

if (condition2) {

// Code if both condition1 and condition2 are true

}

} else {

// Code if condition1 is false

}

**Example:**

int num = 25;

if (num > 0) {

if (num % 5 == 0) {

printf("Positive and divisible by 5\n");

} else {

printf("Positive but not divisible by 5\n");

}

} else {

printf("Number is not positive\n");

}

**switch Statement**

Used to execute different blocks of code based on the value of an expression. It's an alternative to multiple if-else statements when checking a single variable.

**Syntax:**

switch (expression) {

case value1:

// Code

break;

case value2:

// Code

break;

default:

// Code if no case matches

}

**Example:**

int day = 3;

switch (day) {

case 1:

printf("Monday\n");

break;

case 2:

printf("Tuesday\n");

break;

case 3:

printf("Wednesday\n");

break;

default:

printf("Invalid day\n");

}

**Looping in C**

**6 Compare and contrast while loops, for loops, and do while loops. Explain the scenario in which each loop is most appropriate.**

* **(1) while LOOP**

-Syntax

while (condition){

// code block

}

**Key Points:**

* Checks the condition **before** executing the loop body.
* Executes **0 or more times**.
* Good when **looping depends on a condition** that’s checked at the start and may never be true.

**Use When:**

* You **don’t know in advance** how many times to loop.
* The condition might be false initially.

**(2) for LOOP**

**Syntax:**

for (initialization; condition; increment) {

// code block

}

**Key Points:**

* Combines **initialization**, **condition**, and **update** in one line.
* Ideal for looping a **specific number of times**.
* Easier to manage **counter-controlled loops**.

**Use When:**

* You **know exactly** how many times to loop.
* You’re using a **loop counter**.

(3)do-while LOOPS

**Syntax:**

do {

// code block

} while (condition);

**Key Points:**

* Executes the loop body at least once, then checks the condition.
* Useful for menu-driven programs, or when user input is required at least once.

**Use When:**

* You want the loop to run at least once.
* Condition should be checked after one iteration

**7 Explain the use of break, continue, and goto statements in C. Provide examples of each.**

* **1. break Statement**

**Use:**

* Used to **exit a loop or switch** statement **immediately**, regardless of the condition.

**When to Use:**

* When you want to **terminate** a loop (or switch) early based on a condition.

**Example:**

c

Copy code

#include <stdio.h>

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break; // exits the loop when i is 5

}

printf("%d\n", i);

}

return 0;

}

**2. continue Statement**

**Use:**

* Skips the **current iteration** of the loop and **jumps to the next iteration**.

**When to Use:**

* When you want to **skip part of the loop body** for some condition but **continue looping**.

**Example:**

#include <stdio.h>

int main() {

for (int i = 1; i <= 5; i++) {

if (i == 3) {

continue; // skips when i is 3

}

printf("%d\n", i);

}

return 0;

}

**3. goto Statement**

**Use:**

* Transfers control **directly to a labeled statement**.
* Can jump **forward or backward** in the code.

**When to Use:**

* For **exiting deeply nested loops** or handling **error cleanup**.

**Example:**

c

#include <stdio.h>

int main() {

int num = 3;

if (num < 5) {

goto skip; }

printf("This will be skipped\n");

skip:

printf("Jumped using goto!\n");

return 0;

**Functions in C**

**8. What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.**

-> A **function** is a set of statements that perform a task when called.

* C has:
  + **Predefined (library) functions** like printf(), scanf()
  + **User-defined functions** written by the programmer

**1. Function Declaration (Prototype)**

* Tells the compiler about a function’s name, return type, and parameters **before its actual definition**.
* Usually placed **before main()**.

**Syntax:**

c

Copy code

return\_type function\_name(parameter\_list);

**Example:**

c

Copy code

int add(int a, int b); // Declaration

**2. Function Definition**

* Contains the **actual code** to be executed when the function is called.
* Written **outside** the main() function (usually after it).

**Syntax:**

return\_type function\_name(parameters) {

// body of function

return value;

}

**Example:**

int add(int a, int b) {

return a + b;

}

**3. Function Call**

* This is how you **execute** or **use** the function.
* You call the function in main() or any other function.

**Syntax:**

function\_name(arguments);

**Example:**

int result = add(10, 5); // Calls the add() function

**Pointers in C**

**9. Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?**

**->** Pointers in C are variables That store the memory address of another an other variable. They are declared using the ‘\*’symbol, and you can access the value stored at the same memory address using deference Operator ‘\*’. Pointers allow direct memory manipulation, making them powerful for dynamic memory allocation and working with arrays, functions, and structures. The address of a variable is obtained using address of operator ’&’.

**Pointer initialization:-**

You can initialize a pointer by assigning it. The address of a variable using the address of operator &:

**Arrays In C**

**10. Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.**

**Ans.**

* **Concept of Arrays in C:**

An array in C is a collection of elements of the same data type, stored in contiguous memory locations. It allows you to store and access multiple values using a single variable name with an index.

* **Key Points:**
* All elements must be of the same type (e.g., all int, all float).
* Indexing starts from 0.
* Arrays can be 1D (one-dimensional) or multi-dimensional (2D, 3D, etc.).
* **Difference Between 1D and Multi-Dimensional Arrays**

| **Feature** | **One-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| Structure | Linear (list) | Tabular (matrix/grid) |
| Declaration Example | int a[5]; | int b[3][4]; |
| Access Element | a[2] | b[1][3] |
| Use Case | Storing list of values | Storing tables, matrices, game boards etc. |

**Strings In C**

**11. Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.**

**Ans.**

**\_>** String handling functions in C are used to perform operations on strings such as finding length, copying, comparing, concatenating, or locating characters. These functions are available in the header file string.h.

**1. strlen()**

**Purpose:**  
Returns the length of a string (excluding the null character \0)

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char name[] = "Chirag";

printf("Length of string = %lu\n", strlen(name)); // Output: 6}

**Useful When:** You need to know how many characters are in a string to loop or allocate memory.

**2. strcpy()**

**Purpose:**  
Copies one string into another.

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "Hello";

char dest[10];

strcpy(dest, src);

printf("Copied string = %s\n", dest); // Output: Hello

}

**Useful When:** You want to duplicate a string into a new variable.

**3. strcat()**

**Purpose:**  
Appends (concatenates) one string to the end of another.

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char str1[20] = "Chirag";

char str2[] = "Bhure";

strcat(str1, str2);

printf("Concatenated string = %s\n", str1); // Output: Chirag Bhure

}

**4**.**strcmp()**  
**Purpose:**  
Compares two strings.  
**Returns:**

* 0 if strings are equal
* < 0 if first string is less than second
* > 0 if first string is greater than second

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char a[] = "apple", b[] = "apple";

if (strcmp(a, b) == 0)

printf("Strings are equal\n");

else

printf("Strings are not equal\n");

}

**Useful When:** You need to check if two strings are the same (like in login systems).

**5.strchr()**

**Purpose:**Searches for the first occurrence of a character in a string.

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char word[] = "example";

char \*ptr = strchr(word, 'a');

if (ptr != NULL)

printf("First occurrence of 'a' is at position: %ld\n", ptr - word);

else

printf("'a' not found.\n");

}

**Useful When:** You want to find if a character exists in a string or get its position.

**Structures In C**

**12. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.**

**Ans.**

**\_>** A structure in C is a user-defined data type that allows you to group different data types under one name. Structures are especially useful for creating records, such as student info, employee details, etc.

* + - 1. **Declaring a Structure:**

**Syntax:**

struct StructureName {

data\_type member1;

data\_type member2;

...

};

**Example:**

struct Student {

int id;

char name[20];

float marks;

};

(This defines a new type struct Student with 3 members.)

* + - 1. **Declaring Structure Variables:**

After defining the structure, you can declare variables of that structure type:

struct Student s1; // s1 is a variable of type struct Student

You can also declare variables at the time of defining the structure:

struct Student {

int id;

char name[20];

float marks;

} s1, s2;

* + - 1. **Initializing a Structure:**

1. Using direct assignment:

struct Student s1 = {101, "Chirag", 89.5};

1. Assigning values individually:

struct Student s2;

s2.id = 102;

strcpy(s2.name, "Raj"); // use strcpy for strings

s2.marks = 92.0;

* + - 1. **Accessing Structure Members:**

Use the dot (.) operator to access members:

printf("ID: %d\n", s1.id);

printf("Name: %s\n", s1.name);

printf("Marks: %.2f\n", s1.marks);

**5.Using Structures with Functions:**

By value

By reference (using pointers)

**Example by passing Refrence:**

void display(struct Student \*s) {

printf("ID: %d\n", s->id);

printf("Name: %s\n", s->name);

printf("Marks: %.2f\n", s->marks);

}

**Call:**

display(&s1);

**File Handing In C**

**13. Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.**

**Ans.**

**\_>**

**File handling** in C allows you to store data **permanently** in files, instead of losing it when the program ends (as in variables stored in RAM). It is important for:

* Saving user input/data
* Creating logs and reports
* Reading configuration files
* Processing large datasets

C provides standard library functions to **create**, **read**, **write**, **append**, and **close** files using the stdio.h header.

* 1. **Opening a File:**

FILE \*fp;

fp = fopen("data.txt", "w"); // "w" means write mode

**File Modes:**

* "r" – Read
* "w" – Write (creates a new file or truncates if exists)
* "a" – Append
* "r+" – Read & Write
* "w+" – Write & Read
* "a+" – Append & Read

**Always check if the file opened successfully:**

if (fp == NULL) {

printf("File cannot be opened.\n");

}

* 1. **Writing To A File:**

Using fprintf() or fputs():

fprintf(fp, "Name: %s\n", "Chirag");

fputs("Hello File!\n", fp);

* 1. **Reading From a File:**

**Using fscanf() or fgets():**

char name[20];

int age;

fscanf(fp, "%s %d", name, &age);

**Using loop:**

char ch;

while ((ch = fgetc(fp)) != EOF) {

putchar(ch); // prints file content

}

* 1. **Closing a File:**

fclose(fp); // Always close after finishing