The Ultimate C Notebook

~ By SAGAR BISWAS

C PROGRAMMING HANDBOOK

♠ WHAT IS PROGRAMMING?

Computer programming is a medium for us to communicate with computers. Just like we use ‘Bangla’ or ‘English’ to communicate with each other, programming is a way for us to deliver our instructions to the computer.

♠ WHAT IS C?

- C is a programming language.
- C is one of the oldest and finest programming languages.
- C was developed by Dennis Ritchie at AT&T’s Bell labs, USA in 1972.

♠ USES OF C

C language is used to program a wide variety of systems. Some of the uses of C are as follows:

1) Major parts of Windows, Linux and other **operating systems** are written in C.
2) C is used to write **driver** programs for devices like tablets, printers etc.
3) C language is used to program embedded systems where programs need to **run faster** in limited memory (Microwave, Cameras etc.)
4) C is used to develop games, an area where **latency** is very important, example: the computer must react quickly to user input.

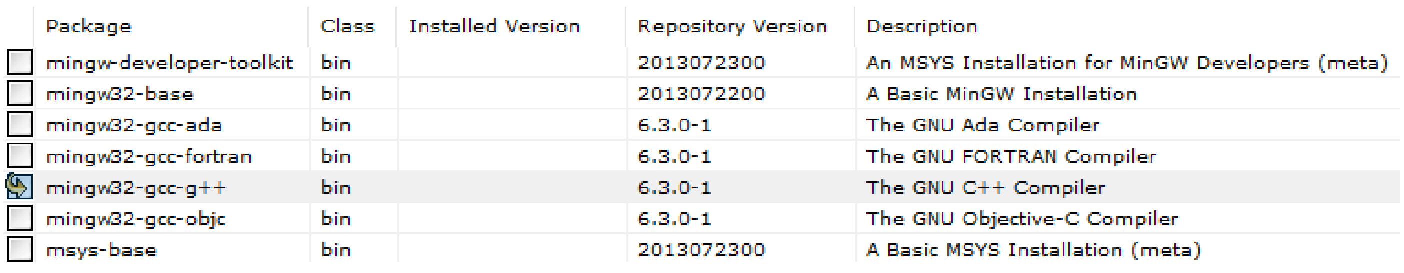
♠ INSTALLATION

We will use VS Code as our code editor to write our code and install MinGW gcc compiler to compile our C program.

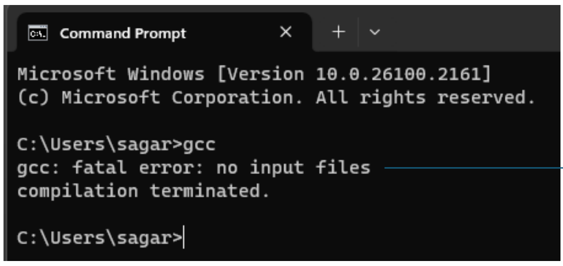
Compilation is the process of translating high-level source code written in programming languages like C into machine code, which is the low-level code that a computer's CPU can execute directly. Machine code consists of binary instructions specific to a computer's architecture.

We can install VS Code and MinGW from their respective websites

Download MinGW: https://sourceforge.net/projects/mingw/



	Package	Class	Installed Version	Repository Version	Description
☐	mingw-developer-toolkit	bin		2013072300	An MSYS Installation for MinGW Developers (meta)
☐	mingw32-base	bin		2013072200	A Basic MinGW Installation
☐	mingw32-gcc-ada	bin		6.3.0-1	The GNU Ada Compiler
☐	mingw32-gcc-fortran	bin		6.3.0-1	The GNU FORTRAN Compiler
	mingw32-gcc-g++	bin		6.3.0-1	The GNU C++ Compiler
☐	mingw32-gcc-objc	bin		6.3.0-1	The GNU Objective-C Compiler
☐	msys-base	bin		2013072300	A Basic MSYS Installation (meta)



```
Microsoft Windows [Version 10.0.26100.2161]
(c) Microsoft Corporation. All rights reserved.

C:\Users\sagar>gcc
gcc: fatal error: no input files
compilation terminated.

C:\Users\sagar>
```

MinGW g++ successfully installed.

PART 1: VARIABLES, CONSTANTS & KEYWORDS

♠ VARIABLES

A variable is a container which stores a 'value'. In kitchen, we have containers storing Rice, Dal, Sugar etc. Similar to that, variables in C stores value of a constant.

Example:

```
a = 3;       // a is assigned "3"
b = 4.7;     // b is assigned "4.7"
c = 'A';     // c is assigned 'A'
```

♠ RULES FOR NAMING VARIABLES IN C

1) First character must be an **alphabet or underscore** (_)
2) No **commas, blanks** are allowed.
3) No special symbol other than (_) allowed.
4) Variable names **are case sensitive.**

We must create meaningful variable names in our programs. This enhances readability of our programs.

♠ CONSTANTS

An entity whose value does not change is called as a constant. A variable is an entity whose value can be changed.

♠ TYPES OF CONSTANTS

Primarily, there are **three** types of constants:

1) Integer Constant	→ 1,6,7,9
2) Real Constant	→ 322.1, 2.5 ,7.0
3) Character Constant	→ 'a', '$', '@' (must be enclosed within single quotes)

cannot change a **constant** once it has been defined A **constant** is a value that remains fixed throughout the program, and trying to change it will cause an error.

```
const int myConstant = 10;
```

If you later try to change myConstant, like this:

```
myConstant = 20;  // This will cause an error
```

♠ KEYWORDS

These are reserved words, whose meaning is already known to the compiler. There are **32 keywords** available in C.

1. auto	9. double	17. int	25. struct
2. break	10. long	18. else	26. switch
3. case	11. return	19. enum	27. typedef
4. char	12. register	20. extern	28. union
5. const	13. short	21. float	29. unsigned
6. continue	14. signed	22. for	30. void
7. default	15. sizeof	23. goto	31. volatile
8. do	16. static	24. if	32. while

♠ OUR FIRST C PROGRAM

```
#include <stdio.h>
int main() {
      printf("Hello, I am learning C with Sagar");
      return 0;
}
```

♠ BASIC STRUCTURE OF A C PROGRAM

All C programs must follow a basic structure. A C program starts with a main function and executes instructions present inside it.

Each instruction is terminated with a semicolon (;).

There are some rules which are applicable to all the C programs:

1) Every program's execution starts from **main()** function.
2) All the statements are terminated with a **semicolon**.
3) Instructions are **case–sensitive.**
4) Instructions are executed in the same order in which they are written.

♠ COMMENTS

Comments are used to clarify something about the program in plain language. It is a way for us to add notes to our program. There are **two** types of comments in C.

1. Single line Comment: Single-line comments start with two forward slashes (//). Any information after the slashes // lying on the same line would be ignored (will not be executed).

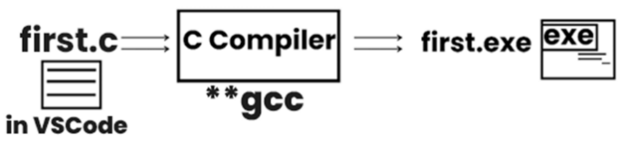
```
// This is a Single line comment.
```

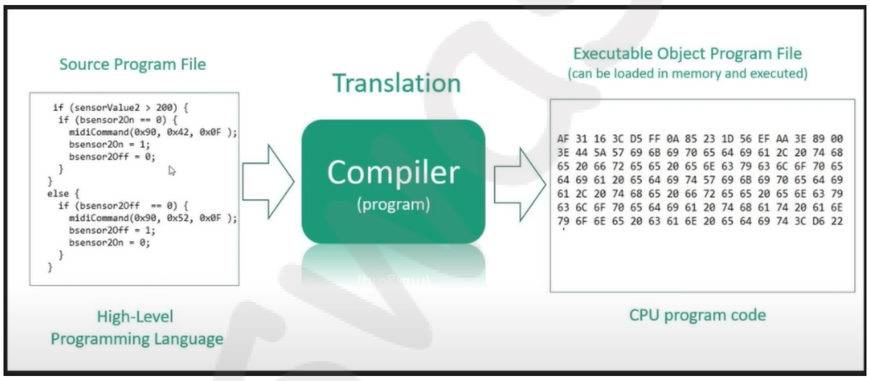
2. Multi-line Comment: A multi-line comment **starts with /* and ends with */.** Any information between /* and */ will be ignored by the compiler.

```
/*
This is a multi
- line comment.
*/
```

✓ Note: Comments in a C program are not executed and are ignored.

♠ COMPILATION AND EXECUTION




Source Program File
Translation
Executable Object Program File
(can be loaded in memory and executed)
Compiler
(program)
High-Level Programming Language
CPU program code

```
if (sensorValue2 > 200) {
  if (bsensor2On == 0) {
    midiCommand(0x90, 0x42, 0x0F );
    bsensor2On = 1;
    bsensor2Off = 0;
  }
}
else {
  if (bsensor2Off  == 0) {
    midiCommand(0x90, 0x52, 0x0F );
    bsensor2Off = 1;
    bsensor2On = 0;
  }
}
```

```
AF 31 16 3C D5 FF 0A 85 23 1D 56 EF AA 3E 89 00
3E 44 5A 57 69 6B 69 70 65 64 69 61 2C 20 74 68
65 20 66 72 65 65 20 65 6E 63 79 63 6C 6F 70 65
64 69 61 20 65 64 69 74 57 69 6B 69 70 65 64 69
61 2C 20 74 68 65 20 66 72 65 65 20 65 6E 63 79
63 6C 6F 70 65 64 69 61 20 74 68 61 74 20 61 6E
79 6F 6E 65 20 63 61 6E 20 65 64 69 74 3C D6 22
```

A compiler is a **computer program** which converts a C program into machine language. so that it can be easily understood by the computer.

A C program is written in plain text. This plain text is a combination of instructions in a particular sequence. The compiler performs some basic checks and finally converts the program into an executable.

♠ LIBRARY FUNCTIONS

C language has a lot of valuable library functions which is used to carry out certain tasks. **For instance** printf() function is used to print values on the screen.

```
#include <stdio.h>
int main() {
        int i = 10;
        printf("This is %d\n", i);
        // %d for integers
        // %f for real values (floating-point numbers)
        // %c for characters
        return 0;
}
```

♠ TYPES OF VARIABLES

1) Integer variables	→ int a=3;
2) Real variables	→ int a=7; float a=7.7;
3) Character variables	→ char a= 'b';

♠ RECEIVING INPUT FROM THE USER

In order to take input from the user and assign it to a variable, we use **scanf()** function.

Syntax: `scanf("%d", &i);`

‘&’ is the “address of” operator and it means that the supplied value should be copied to the address which is indicated by variable i.

PART 2: INSTRUCTIONS AND OPERATORS

A C program is a set of instructions. Just like a recipe - which contains instructions to prepare a particular dish.

♠ TYPES OF INSTRUCTIONS

1) Type declaration Instructions.
2) Arithmetic Instructions
3) Control Instructions.

♠ TYPE DECLARATION INSTRUCTIONS

This is how you declare a variable in C

```
int a;
float b;
char c;
```

♠ OTHER VARIATIONS:

Some other variations of this declaration look like this:

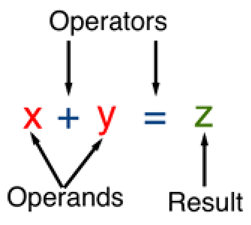
```
int a;                          // Declare an integer variable 'a'
float b;                        // Declare a float variable 'b'
int i = 10;                     // Declare and initialize 'i' with 10
int j = i;                      // Declare 'j' and initialize with 'i'
int a = 2, b = 3, c = 4, d = 5; // Declare and initialize multiple variables
int j1 = a + j - i;             // use previously defined variables
int a, b, c, d;
a = b = c = d = 30;             // a, b, c, d all equal to 30
```

♠ ARITHMETIC INSTRUCTIONS

Arithmetic instructions perform mathematical operations.

Here are some of the commonly used operators in C language:

+	(Addition)
-	(Subtraction)
*	(Multiplication)
/	(Division)
%	(Modulus)



Note:

1. Operands can be int/float etc. + - * / are arithmetic operators.

```
int b = 2, c = 3;
int z; z = b * c; //legal
int z; b* c = z; //illegal (not allowed)
```

2. % is the modular division operator

- % → returns the remainder
- % → **cannot be applied** on **float**
- % → sign is same as of numerator (-5%2=-1)

3. No operator is assumed to pe present.

```
int i = ab    // invalid
int i = a * b //valid
```

4. There is **no operator to perform exponentiation** in C however we can use **pow(x,y)** from (more later).

♠ TYPE CONVERSION

An Arithmetic operation between

- int and int → int
- int and float → float
- float and float → float

Example:

- 5/2 becomes 2 as both the operands are int.
- 5.0/2 becomes 2.5 as one of the operands is float.
- 2/5 becomes 0 as both the operands are int.

✓ NOTE:

In programming, type compatibility is crucial. For int a = 3.5;, the float 3.5 is demoted to 3, losing the fractional part because a is an integer. Conversely, for float a = 8;, the integer 8 is promoted to 8.0, matching the float type of a and retaining precision.
Example:

```
int a = 3.5; // In this case 3.5 (float) will be demoted to 3 (int)
// because a(int) is not able to store floats.
float a = 8; // a will store 8.0 | 8 -> 8.0 (promotion to float)
```

Quick Quiz: int k = 3.0 /9; value of k? and why?

Ans: 3.0/9 = 0.333. But since k is an int, it cannot store floats & value 0.33 is demoted to 0.

♠ OPERATOR PRECEDENCE IN C

Have a look at the below statement:

3*x – 8*y is (3x)-(8y) or 3(x-8y)?

In C language simple mathematical rules like **BODMAS**, no longer apply.

The answer to the above question is provided by operator precedence & associativity.

♠ OPERATOR PRECEDENCE

The following table lists the operator priority in C.

Priority	Operators
1st	* / %
2nd	+-
3rd	=

Operators of higher priority are evaluated first in the absence of parenthesis.

♠ OPERATOR ASSOCIATIVITY

When operators of equal priority are present in an expression, the tie is taken care of by associativity.

x*y/z → (x*y)/z

x/y*z → (x/y)*z

*, / follows **left to right** associativity

✓ Pro Tip: Always use parenthesis in case of confusion

♠ CONTROL INSTRUCTIONS

Determines the flow of control in a program four types of control instructions in C are:

1. Sequence Control instructions. (like stack)
   These are simple instructions that execute one after the other in sequence.
2. Decision Control instructions (if, else if, else)
   These are used to make decisions based on conditions.
3. Loop Control instructions (for, while, do while)
   These are used to repeat a block of code multiple times.
4. Case Control instructions. (switch)
   These are used for selecting one of many possible blocks of code to execute.

PART 3: CONDITIONAL INSTRUCTIONS

Sometimes we want to watch comedy videos on YouTube if the day is Sunday.

Sometimes we order junk food if it is our friend's birthday in the hostel.

You might want to buy an umbrella if it's raining, and you have the money.

You order the meal if dal or your favourite dish is listed on the menu.

All these are decisions which depends on a condition being met. In C language too, we must be able to execute instructions on a condition(s) being met.

♠ DECISION MAKING INSTRUCTIONS IN C

- if–else statement
- switch statement

♠ IF-ELSE STATEMENT

The syntax of an if-else statement in C looks like:

```
if (condition_to_be_checked) {
        // Statements if condition is true
}
else {
        // Statements if condition is false
}
```

♠ CODE EXAMPLE:

```
int a = 23;
if (a > 18)
{
        printf("you can drive \n");
}
```

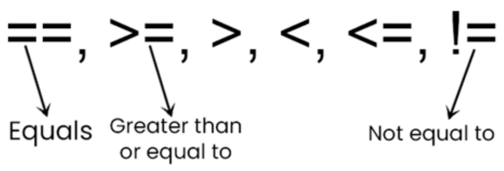
Note that else block is not necessary but optional.

♠ RELATIONAL OPERATORS IN C

Relational operators are used to evaluate conditions (true or false) inside the if statements.

Some examples of relational operators are: ==, >=, >, <=, !=

Important note: '=' is used for assignment whereas '==' is used for equality check.



The condition can be any valid expression. **In C a non-zero value is considered to be true.**

♠ LOGICAL OPERATORS

&&, || and !, are three logical operators in C. These are read as "AND", "OR" and "NOT"

They are used to provide logic to our C programs.

♠ USAGE OF LOGICAL OPERATORS:

1. && (AND) → is **true** when both the conditions are true. (condition && condition)

a. "1 and 0" is evaluated as false.

b. "0 and 0" is evaluated as false.

c. "1 and 1" is evaluated as true.

0 is considered **false**.

Any non-zero value (commonly 1) is considered **true**.

2. || (OR) → is **true** when at least one of the conditions is true. (1 or 0 → 1) (1 or 1 → 1) (0 or 0 → 0)

3. ! (NOT) → returns **true** if given false and **false** if given true.

a) !(3==3) → evaluates to false
b) !(3>30) → evaluates to true.

As the number of conditions increases, the level of indentation increases. This reduces readability. Logical operators come to rescue in such cases.

♠ ELSE IF CLAUSE

Instead of using multiple if statements, we can also use else if along with it thus forming an if-else if-else ladder.

♠ CODE EXAMPLE

A typical if - else if - else ladder look like this:

```
If(condition){
        // Statements
}
else if(condition){
        // Statements
}
else {
        // Statements
}
```

♠ IMPORTANT NOTE

1. Using if, if-else, else **reduces** nested statement.

2. The last "else" is optional.

3. Also, there can be any number of "else if".

4. Last else is executed only if all conditions fail.

♠ OPERATOR PRECEDENCE

Priority	Operators
1st	!
2nd	*, /, %
3rd	+, -
4th	<>, <=, >=
5th	==, !=
6th	&&
7th	||
8th	=

♠ CONDITIONAL OPERATORS

A shorthand "if – else" can be written using the conditional or ternary operators.

```
condition ? expression - if - true : expression - if - false
// Here "?" and ":" are called Ternary Operators
```

♠ SWITCH CASE CONTROL INSTRUCTION

switch-case is used when we have to make a choice between number of alternatives for a given variable.

```
#include<stdio.h>
int main() {
    int i = 2;

    switch (i) { // i of any integer condition.
    case 0:
        printf("The value of i is: %d\n", 0);
        break;
    case 1:
        printf("The value of i is: %d\n", 1);
        break;
    case 2:
        printf("The value of i is: %d\n", 2);
        break;
    default:
        printf("Default case\n");
    }
    return 0;
}
```

In this example, the program prints: "The value of i is: 2", if any case does not match with the condition, then it goes to default case.

The `break;` keyword is so much important because if we don't use `break;` then it prints:

The value of i is: 2
Default Case

Quick Quiz: Write a program to find grade of a student given his marks based on below:

90 – 100 => A

80 – 90 => B

70 – 80 => C

60 – 70 => D

50 – 60 => E

<50 => F

Some Important Notes:

- We can use switch-case statements even by writing cases in any order of our choice (not necessarily ascending).
- char values are allowed as they can be easily evaluated to an integer. characters are internally represented as integers based on their ASCII values.

Example:

```
#include <stdio.h>

int main() {
    char grade = 'B';

    switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
        printf("Well done!\n");
        break;
    case 'C':
        printf("Good\n");
        break;
    case 'D':
        printf("You passed\n");
        break;
    case 'F':
        printf("Better try again\n");
        break;
    default:
        printf("Invalid grade\n");
    }

    return 0;
}
```

- A switch can occur within another but in practice this is **rarely** done.

Example:

```
#include <stdio.h>

int main() {
    int age = 25;
    char category = 'A';

    switch (age) {
    case 18:
        printf("Just an adult.\n");
        break;
    case 25:
        printf("You are 25.\n");

        // Nested switch to evaluate the category
        switch (category) {
        case 'A':
            printf("Category A: Standard user\n");
            break;
        case 'B':
            printf("Category B: Premium user\n");
            break;
        default:
            printf("Unknown category\n");
        }
        break;
    default:
        printf("Age not specified\n");
    }

    return 0;
}
```

We can use default case as a else condition by using break;

PART 4: LOOP CONTROL INSTRUCTION

♠ WHY LOOPS

Sometimes we want our programs to execute few sets of instructions over and over again. For example: Printing 1 to100, first 100 even numbers etc.

Hence loops make it easy for a programmer to tell computer that a given set of instructions must be executed **repeatedly**.

♠ TYPES OF LOOPS

Primarily there are **three types** of loops in C language:

1. while loop
2. do–while loop
3. for loop

We will look into these one by one:

1) WHILE LOOP

```
while (condition is true) {
        // Code
        // The block keeps executing as long as the condition is true
}
```

Example:

```
int i = 0;
while (i < 10) {
        printf("the value of i is %d\n", i);
        i++;
    }
```

✓ Note: If the condition never becomes false, the while loop keeps getting executed. Such loop is known as an **infinite loop**.

Quick Quiz: Write a program to print natural numbers from 10 to 20 when initial loop counter is initialized to 0.

Solution:

```
#include <stdio.h>

int main() {
    int i = 0;  // Initialize counter to 0

    // Loop until i reaches 10 (which corresponds to printing 20)
    while (i <= 10) {
        printf("%d\n", i + 10);  // Print numbers from 10 to 20
        i++;
    }

    return 0;
}
```

✓ Note: The loop counter need not be int, it can be **float** as well.

♠ INCREMENT AND DECREMENT OPERATORS

i++ → i is increased by 1

i -- → i is decreased by 1

```
// Decrement i first and then print
printf("--i = %d\n", --i);
// Print i first and then decrement
printf("i-- = %d\n", i--);
```

✓ Note:

- +++ operator does not exist.
- i+=2 is compound assignment which translates to i = i + 2
- Similar to += operator we have other operators like -=, *=, /=, %=.

2) DO–WHILE LOOP

The syntax of do-while loop looks like this:

```
do {
        //code;
} while (condition);
```

The do–while loop works very similar to while loop.

- 'while' checks the condition & then executes the code.
- 'do-while' executes the code & then checks the condition.

In simpler terms we can say:

do-while loop = while loop which **executes at least once.**

Quick Quiz: Write a program to print first 'n' natural number using do-while loop.

3) FOR LOOP

The syntax of a typical 'for' loop looks like this:

```
for (initialize; test; increment or decrement)
{
        // code
}
```

- Initialize → Setting a loop counter to an initial value.
- Test → Checking a condition.
- Increment or decrement → Updating the loop counter.

Example:

```
 for (i = 0; i < 3; i++) {
        printf("%d", i);
}
```

```
Output:012
```

Quick Quiz: Write a program to print first 'n' natural numbers using for loop.

♠ A CASE OF DECREMENTING FOR LOOP

```
for(i = 5; i; i--) {
        printf(" %d", i);
}
```

```
5 4 3 2 1
```

This for loop will keep on running until i become 0.

The loop runs in following steps:

1. 'i' is initialized to 5.
2. The condition "i" (0 or none) is tested.
3. The code is executed.
4. 'i' is decremented.
5. Condition 'i' is checked & code is executed if it's not 0.
6. And so on until 'i' is non 0.

Quick Quiz: Write a program to print 'n' natural numbers in reverse order.

♠ THE BREAK STATEMENT IN C

The 'break' statement is used to exit the loop irrespective of whether the condition is true or false.

Whenever a "break" is encountered inside the loop, the control is **sent outside** the loop.

Let us see this with the help of an example:

```
for (i = 0; i < 1000; i++) {
        printf("%d", i);
        if (i == 5) {
                break;
        }
}
```

```
output: 012345
```

♠ THE CONTINUE STATEMENT IN C

The ‘continue’ statement is used to immediately move to the next iteration of the loop.

The control is taken to the next iteration thus **skipping** everything below “continue” inside the loop for that iteration.

Example:

```
#include <stdio.h>
int main() {
        int skip = 5;
        int i = 0;
        while (i < 10) {
                if (i == skip) {
                        i++;
                        continue; // skips the rest of the loop body for i == 5
                }
                printf("%d\n", i);
                i++;
        }
        return 0;
}
```

♠ Notes:

1. Sometimes, the name of the variable might not indicate the behavior of the program.

2. ‘break’ statement completely exits the loop.

3. ‘continue’ statement skips the particular iteration of the loop.

PART 5 – FUNCTIONS AND RECURSION

Sometimes our program gets bigger in size and it's not possible for a programmer to track which piece of code is doing what.

Function is a way to break our code into chunks so that it is possible for a programmer to reuse them.

♠ WHAT IS A FUNCTION?

A function is a block of code which performs a particular task.

A function can be reused by the programmer in a given program any number of times.

Syntax:

```
#include <stdio.h>
// Function prototype
void display();
```

```
int main() {
        int a; // Variable declaration
        display(); // Function call
        return 0; // Return statement
}
// Function definition
void display() {
        printf("hi i am display\n"); // Printing the message
}
```

♠ FUNCTION PROTOTYPE

A function prototype informs the compiler about a function that will be defined later in the program.

Example:

```
#include <stdio.h>

// Function prototype (declaration)
int add(int, int);

int main() {
    int result = add(5, 3);  // Function call
    printf("The sum is: %d\n", result);
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

✓ Note: The **void** keyword indicates that the function does not return any value.

♠ FUNCTION CALL

A function call instructs the compiler to execute the function's body when the call is made.

Note that program execution starts from the main function and follows the sequence of instructions written.

♠ FUNCTION DEFINITION

This part contains the exact set of instructions executed during the function call.

When a function is called from main(), the main function pauses and temporarily suspends. During this time, control transfers to the called function. Once the function finishes executing, main() resumes.

Quick Quiz: Write a program with three functions

1. Good morning function which prints “good morning”.
2. Good afternoon function which prints “good afternoon”.
3. Good night function which prints “good night”.

main() should call all of these in order 1→2→3

♠ IMPORTANT POINTS

- Execution of a C program starts from main().
- A C program can have more than one function.
- Every function gets called directly or indirectly from main().

Example:

```c
#include <stdio.h>

// Function prototypes
void functionA();
void functionB();
void functionC();

int main() {
    // Direct call to functionA
    printf("In main function\n");
    functionA();
    return 0;
}

// Function definitions
void functionA() {
    printf("In functionA\n");
    // Indirect call to functionB through functionA
    functionB();
}

void functionB() {
    printf("In functionB\n");
    // Indirect call to functionC through functionB
    functionC();
}

void functionC() {
    printf("In functionC\n");
}
```

♠ TYPES OF FUNCTIONS

There are **two** functions in C. Let's talk about them.

1. Library functions → Commonly required functions grouped together in a library file on disk. (pre-written, built-in functions)

2. User defined function → These are the functions declared and defined by the user.

♠ WHY USE FUNCTIONS

1. To avoid rewriting the same logic again and again.

2. To keep track of what we are doing in a program.

3. To test and check logic independently.

♠ PASSING VALUES TO FUNCTION

We can pass values to a function and can get a value in return from a function.

Have a look at the code snippet below:

```c
int sum(int a, int b)
```

A function prototype in programming is a declaration of a function that specifies its name, return type, and parameters (if any) but does not include the function body.

(So, a function prototype does not include the function body.)

The above prototype means that sum is a function which takes values 'a' (of type int) and 'b' (of type int) and returns a value of type int.

Function definition of sum can be:

```c
int sum(int a, int b) { // a and b are parameters
        int c;
        c = a + b;
        return c; // returning an integer value.
}
// Now we can call sum (2,3); from main to get 5 in return. Here 2 & 3 are arguments.
int d = sum(2, 3); // d becomes 5
```

✓ NOTE:

1. **Parameters** are the values or **variable** placeholders in the function definition. Example a & b.

2. **Arguments** are the actual **values** passed to the function to make a call. Example 2 & 3.

3. A function can **return only one value** at a time.

4. If the passed variable is changed inside the function, the function call doesn't change the value in the calling function.

It means that if a variable is passed by **value** to a function, changes made to it inside the function(local variable) do not affect the original variable in the calling function.

Example:

```c
#include <stdio.h>

void modifyValue(int x) {
    x = 100;  // This change only affects the local copy of x
    printf("Inside function: %d\n", x);
}

int main() {
    int num = 50;
    printf("Before function call: %d\n", num);

    modifyValue(num);  // num is passed by value (a copy is made)

    printf("After function call: %d\n", num);  // Original value is unchanged
    return 0;
}
```

```
Output:
Before function call : 50
Inside function : 100
After function call : 50
```

Quick Quiz: Use the library function to calculate the area of a square with side a.

♠ RECURSION

A function defined in C can call itself. This is called recursion. A function calling itself is also called 'recursive' function.

Example:

A very good example of recursion is factorial.

Factorial(n) = 1 x 2 x 3 … x n

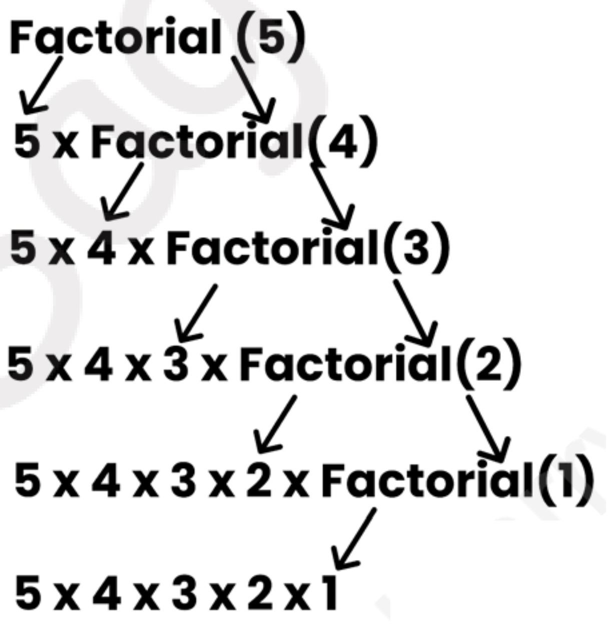
Factorial(n) = 1 x 2 x3 … (n-1) x n

Factorial(n) = Factorial (n-1) x n

Since we can write factorial of a number in terms of itself, we can program it using recursion.

```c
#include <stdio.h>

int factorial(int x) {
    int f;
    if (x == 0 || x == 1) {
        return 1;  // Base case: factorial of 0 or 1 is 1
    }
    else {
        f = x * factorial(x - 1);  // Recursive call
        return f;
    }
}

int main() {
    int x = 5;
    int num = factorial(x);
    printf("After function call, factorial of %d is: %d\n", x, num);
    return 0;
}
```

♠ DRY RUN OF RECURSIVE FACTORIAL PROGRAM



♠ IMPORTANT NOTES:

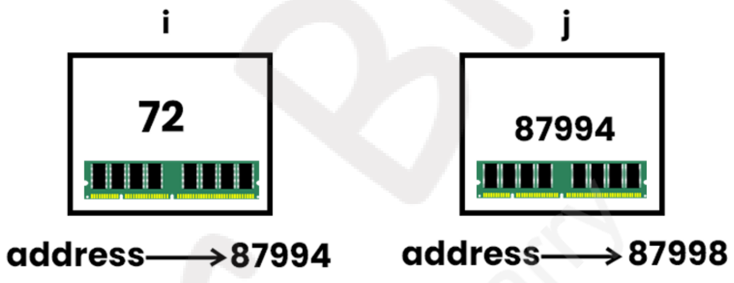
1. **Recursion is often a direct way to implement certain algorithms, but not always the most direct for every algorithm**. Recursion is particularly suited for problems that can be divided into smaller, similar subproblems (like factorial computation or tree traversal), but for some algorithms, iterative approaches might be more straightforward or efficient.

2. **The condition in a recursive function that stops further recursion is called the base case**. This correction clarifies that the base case is crucial as it prevents infinite recursion and ensures the function terminates correctly.

3. **Sometimes, due to an oversight by the programmer, a recursive function can continue to run indefinitely without reaching a base case, potentially causing a stack overflow or memory error**. This statement highlights the risk of infinite recursion and its consequences, emphasizing the importance of properly defining base cases in recursive functions.

PART 6- POINTERS

♠ A pointer is a variable which stores the address of another variable.



♠ THE "ADDRESS OF" (&) OPERATOR

The address of operator is used to obtain the address of a given variable.

If you refer to the diagrams above,

&i → 87994

&j → 87998

Format specifier for printing pointer address is '**%p** or **%u**'.

♠ THE 'VALUE AT ADDRESS' OPERATOR (*)

The value at address or * operator is used to obtain the value present at a given memory address. It is denoted by *.

~~*~~(~~&~~i) =72

~~~~(~~&~~j) = 87994

~~**~~(&~~&&~~k) = 892.2

Quarrel with & and * (The easiest way.)

♠ HOW TO DECLARE A POINTER?

A pointer is declared using the following syntax.

- int *j => declare a variable j of type int-pointer
- j=&i => storing address of i in j.

Just like pointer of type integer, we also have pointers to char, float etc.

```
int* in_ptr;       //pointer to integer
char* ch_ptr;      //pointer to character
float* fl_ptr;     //pointer to float
```

Although it's a good practice to use meaningful variable names, we should be very careful while reading and working on programs from fellow programmers.

♠ A PROGRAM TO DEMONSTRATE POINTERS

```
#include <stdio.h>
      int main() {
            int i = 8;
            int* j;
            j = &i;
            printf("add i= %u\n", &i);
            printf("add i= %u\n", j);
            printf("add j= %u\n", &j);
            printf("value i= %d\n", i);
            printf("value i= %d\n", *(&i));
            printf("value i= %d\n", *j);
            return 0;
      }
```

```
   Output
add i= 6422300
add i= 6422300
add j= 6422296
value i= 8
value i= 8
value i= 8
```

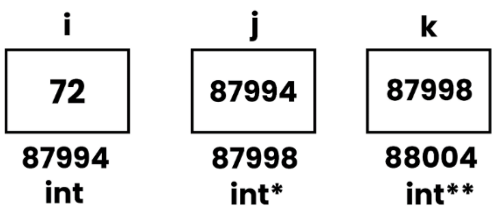
This program means it all. If you understand it, you have got the idea of pointers.

♠ POINTER TO A POINTER

Just like 'j' is pointing to 'i' or storing the address of 'i', we can have another variable k which can further store the address of 'j'. What will be the type of 'k'?

```
int** k;
k = &j;
```

We can even go further one level and create a variable 'l' of type int*** to store the address of 'k'. We mostly use int* and int** sometimes in real world programs.



```
#include <stdio.h>  // Include necessary library for printf

int main() {
    int* j;       // Define j as a pointer to an integer
    int** k;      // Define k as a pointer to a pointer to an integer

    k = &j;       // Now k points to the address of j

    printf("K = %p\n", k); // K = 0061FF18
    printf("J = %p\n", j); // J = 00000000 because value of j wasn't assigned

    return 0;      // Return 0 to indicate successful completion
}
```

Example 1:

```
#include<stdio.h>

int main() {

    int i = 10;
    int* j = &i; // ptr is a pointer to x. ptr is storing the address of x.
    printf("The address of i: %u\n", &i); // %u for integer address. 6422300

    printf("The address of i: %p\n", &i); // %p for pointer address. 0061FF1C --> Hexadecimal
    printf("The address of i: %p\n", j); // 0061FF1C --> Hexadecimal number

    printf("The value of i: %d\n", *j); // %d for integer value. 10

    /// * indicate the value of the address. and & indicate the address of the value

    printf("The value of i is: %d\n", *(&i)); // 10
    return 0;
}
```

Example 2: (Pointer to Pointer)

```
#include<stdio.h>

int main() {

    int i = 10;
    int* j = &i;
    int**~~*~~ k = ~~&~~j;

    printf("The value of i is: %d\n", i);           // 10

    printf("The value of i is: %d\n", *j);          // 10

    printf("The value of i is: %d\n", *(&i));       // 10

    printf("The value of i is: %d\n", **(&j));      // 10

    printf("The value of i is: %d\n", **k);         // 10

    printf("The value of *k or k is: %d\n", k);     // 6422292

    printf("The value of *k or k is: %d\n", *k);    // 6422292

    return 0;
}
```

♠ TYPES OF FUNCTION CALL

Based on the way we pass arguments to the function; function calls are of **two** types.

1. Call by value → Sending the **values** of arguments.
2. Call by reference → Sending the **address** of arguments.

♠ CALL BY VALUE

Here the values of the **arguments/values or variables** are passed to the function. Consider this example:

```
int c = sum(3, 4); //assume x=3 and y=4
```

If sum is defined as sum (int a, int b), the values 3 and 4 are copied to a and b. Now even if we change a and b, nothing happens to the variables x and y.

This is call by value. In C we usually make a call by value.

♠ CALL BY REFERENCE

Here the **address of the variables** is passed to the function as arguments.

Now since the addresses are passed to the function, the function can now modify the value of a variable in calling function using * and & operators.

```
#include<stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {

    int x = 10;
    int y = 20;
    swap(&x, &y);

    printf("value of x = %d and y = %d", x, y);

    return 0;
}
```

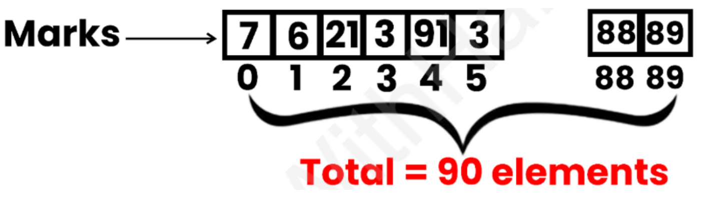
This function is capable of swapping the values passed to it. If a = 10 and b = 20 before a call to swap(a, b), then a = 20 and b = 10 after calling swap.

PART 7 – ARRAYS

An array is a collection of similar elements. Array allows a single variable to store multiple values.

♠ SYNTAX:

```
int marks[90]; // integer array
char name[20]; // character array or string
float percentile[90]; // float array
```



The values can now be assigned to make array like this:

```
marks[0] = 7;
marks[1] = 6;
```

✓ Note: It is very important to note that the array index starts with 0.

ACCESSING ELEMENTS

Elements of an array can be accessed using:

```
scanf("%d", &marks[0]); // input first value
printf("%d", marks[0]); // output first value of the array
```

Quick Quiz: Write a program to accept marks of five students in an array and print them on the screen.

INITIALIZATION OF AN ARRAY

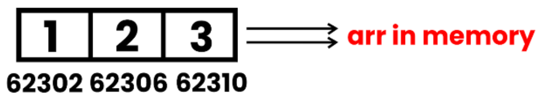
There are many other ways in which an array can be initialized.

```
int cgpa[3] = { 9, 8, 8 }; // arrays can be initialized while declaration
float marks[] = { 33, 40 };
```

ARRAYS IN MEMORY

Consider this array:

```
int arr[3] = {1, 2, 3} // 1 integer = 4 bytes
```



This will reserve 4 x 3 = 12 bytes in memory (4 bytes for each integer).

POINTER ARITHMETIC

A pointer can be incremented to point to the next memory location of that type.

Example:

```
#include<stdio.h>

int main() {

    int a = 5;
    int* ptr = &a;

    printf("The address of a is: %u\n", &a);
    printf("The address of a is: %u\n", ptr);
    ptr++;

    printf("The value of ptr is: %u\n", ptr);

    return 0;
}
```

```
int i = 32;
int* a = &i; // a = 87994
a++; // address of i or value of a = 87998
char a = 'A';
char* b = &a; // a= 87994
b++; // now a = 87995
float i = 1.7;
float* a = &i; // now a = 87994
a++; // now a = 87998
```

```
The address of a is : 6422296
The address of a is : 6422296
The value of ptr is : 6422300
```

Following operations can be performed on a pointer:

- Addition of a number to a pointer.
- Subtraction of a number from a pointer.
- Subtraction of one pointer from another.
- Comparison of two pointer variables.

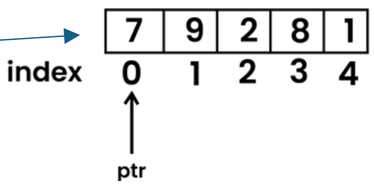
Quick Quiz: Try these operations on another variable by creating pointers in a separate program. Demonstrate all the four operations.

Solve:

```
#include <stdio.h>

    int main() {
        // Declare two integer variables
        int a = 10, b = 20;

        // Create pointers for these variables
        int* p1 = &a;
        int* p2 = &b;

        // 1. Addition of a number to a pointer
        printf("Original address in p1: %u\n", p1); // 6422288
        p1 = p1 + 1;
        printf("Address in p1 after adding 1: %u\n", p1); // 6422292

        // 2. Subtraction of a number from a pointer
        printf("Original address in p2: %u\n", p2); // 6422284
        p2 = p2 - 1;
        printf("Address in p2 after subtracting 1: %u\n", p2); // 6422280

        // 3. Subtraction of one pointer from another
        int diff = p1 - p2;
        printf("Difference between p1 and p2 (p1 - p2): %d\n", diff); // 3

        // 4. Comparison of two pointer variables
        if (p1 > p2) {
            printf("p1 is greater than p2\n"); // p1 is greater than p2
        }
        else if (p1 < p2) {
            printf("p1 is less than p2\n");
        }
        else {
            printf("p1 is equal to p2\n");
        }

        return 0;
    }
```

♠ ACCESSING ARRAY USING POINTERS

Consider this array:



If ptr points to index 0, ptr++ will point to index 1 & so on...

This way we can have an integer pointer pointing to first element of the array like this:

```c
#include<stdio.h>
int main() {
    int arr[] = { 7, 9, 2, 8, 1 };
    int* ptr = arr; // arr means arr[0].
    ptr++;
    printf("%d", *ptr); // 9
    return 0;
}
```

♠ PASSING ARRAY TO FUNCTIONS

Array can be passed to the functions like this:

```c
#include<stdio.h>
void printArray1(int array[], int size) { // array[] means array[0]
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
}
void printArray2(int* array, int size) { // array means array[0]
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
}
int main() {
    int arr[] = { 7, 9, 2, 8, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);

    printArray1(arr, n);
    printf("\n");
    printArray2(arr, n);
    return 0;
}
```

♠ MULTIDIMENSIONAL ARRAYS

An array can be of 2 dimension/ 3 dimension/ n dimensions.

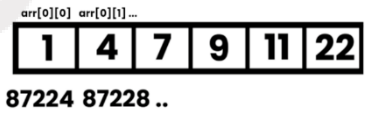
A 2 dimensions array can be defined like this:

```c
int arr[3][2] = { {1, 4}, {7, 9}, {11, 22} };
```

We can access the elements of this array as arr[0][0] , arr[0][1] & so on ...

♠ 2-D ARRAYS IN MEMORY

A 2d array like a 1d array is stored in contiguous memory blocks like this:



Quick Quiz: Create a 2-d array by taking input from the user. Write a display function to print the content of this 2-d array on the screen.

Solution:

```c
#include <stdio.h>

void display(int rows, int cols, int arr[rows][cols]) {
    printf("The 2D array is:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int rows, cols;

    // Get the number of rows and columns from the user
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &cols);

    int arr[rows][cols];

    // Input elements for the 2D array
    printf("Enter elements for the 2D array:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Element [%d][%d]: ", i, j);
            scanf("%d", &arr[i][j]);
        }
    }

    // Call the display function to print the array
    display(rows, cols, arr);
    return 0;
}
```

OUTPUT:

```
Enter the number of rows: 3
Enter the number of columns: 2
Enter elements for the 2D array:
Element [0][0]: 12
Element [0][1]: 13
Element [1][0]: 11
Element [1][1]: 10
Element [2][0]: 12
Element [2][1]: 123
The 2D array is:
12 13
11 10
12 123
```

PART 8 – STRINGS

A string is a 1-D character array terminated by a **null character ('\0')**

Example:

```c
#include <stdio.h>

int main() {
    // Define and initialize a string
    char str[] = "Hello, World!";

    // Print the string
    printf("String: %s\n", str);

    // Get the length of the string
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    printf("Length of string: %d\n", length);

    // Modify the string
    str[7] = 'C';
    printf("Modified string: %s\n", str);

    return 0;
}
```

A null character is used to denote the termination of a string. Characters are stored in contiguous memory locations.

INITIALIZING STRINGS

Since string is an array of characters, it can be initialized as follows:

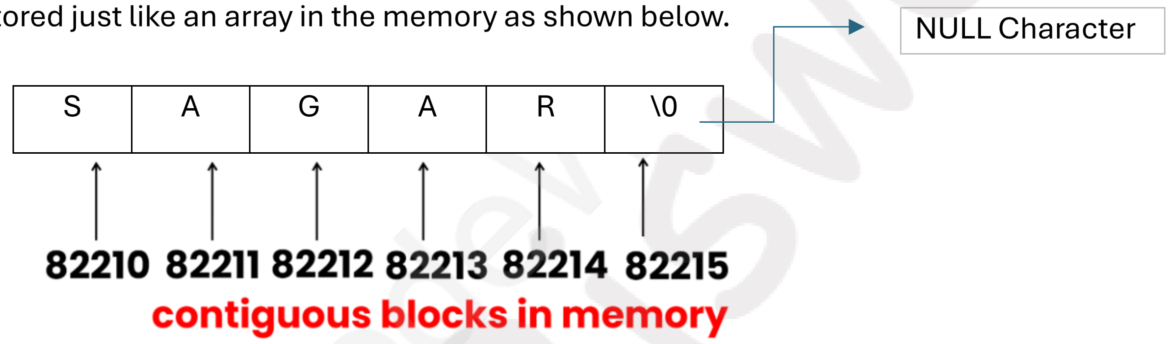
```
char s[] = { 'S', 'A', 'G', 'A', 'R', '\0' };
```

There is another shortcut for initializing string in C language:

```
char s[] = "SAGAR"; // In this case C adds a null character automatically.
```

STRINGS IN MEMORY

A string is stored just like an array in the memory as shown below.



Quick Quiz: Create a string using double quotes and print its content using a loop.

```
#include <stdio.h>

int main() {
    // Define and initialize a string
    char str[] = "Quick Quiz!";

    // Print the string using a loop
    for (int i = 0; str[i] != '\0'; i++) {
        printf("%c", str[i]);
    }
    printf("\n"); // Print a newline for better formatting
    return 0;
}
```

PRINTING STRINGS

A string can be printed character by character using printf and %c. **But there is another convenient way to print strings in C.**

```
#include<stdio.h>
int main() {
    char st[] = "Sagar";
    printf("%s", st); // print the entire string.
    return 0;
}
```

TAKING STRING INPUT FROM THE USER

We can use **%s** with scanf to take string input from the user:

```
char st[50];
```

```
scanf("%s", st);
```

scanf automatically adds a null character when the enter key is pressed.

✓ Note:

1. The string should be short enough to fit into the array.
2. scanf cannot be used to input **multi-word strings with spaces**.

♠ GETS() AND PUTS()

gets() is a function which can be used to receive a multi-word string.

```
char st[30];
gets(st); // The entered string is stored in st!
```

multiple gets() calls will be needed for multiple strings. Likewise, puts can be used to output a string.

```
puts(st); // Prints the string & places the cursor on the next line
```

♠ FGETS()

Syntax:

```
fgets(char_array, size, stdin);
```

Parameters:

- **char_array**: The character array (string) where the input will be stored.
- **size**: The maximum number of characters to read, usually the size of the character array.
- **stdin**: Specifies that the input should come from the standard input (keyboard). fgets can also be used with files by replacing stdin with a file pointer.

✓ Note: A **buffer limit** is the maximum amount of data that can be safely stored in a buffer to prevent overflow.

♠ DECLARING A STRING USING POINTERS

We can declare strings using pointers.

```
#include<stdio.h>
int main() {
    char* ptr = "Sagar";
    printf("%s", ptr); // Sagar // not same as regular method; not using *ptr for the value.
    return 0;
}
```

This tells the compiler to store the string in memory and assigned address is stored in a char pointer.

✓ Note:

1. Once a string is defined using char st [] = "Sagar", it **cannot be reinitialized** to something else.

Example: (1)

```
#include <stdio.h>

int main() {
    char st[] = "Sagar";    // Define an array initialized with "Sagar"
    printf("Original: %s\n", st);

    // Attempt to reinitialize
    st = "Asif";           // This line will cause a compilation error
    return 0;
}
```

2. A string defined using pointers **can be reinitialized.**

Example: (2)

```
#include <stdio.h>

int main() {
    const char* ptr = "Hello";  // Initialize ptr to point to the string "Hello"
    printf("Before: %s\n", ptr);

    ptr = "Sagar";               // Reinitialize ptr to point to the string "Sagar"
    printf("After: %s\n", ptr);
    return 0;
}
```

♠ STANDARD LIBRARY FUNCTIONS FOR STRINGS

C provides a set of standard library functions for string manipulation. Some of the most commonly used string functions are: (#include <string.h>)

- STRLEN()

This function is used to count the number of characters in the string excluding the null ('\0') characters.

```
int length = strlen(st);
```

These functions are declared under header file. (#include <string.h>)

- STRCPY()

This function is used to copy the content of second string into first string passed to it.

```
char source[] = "Sagar";
char target[30];
strcpy(target, source); //target now contains "Sagar"
```

target string should have enough capacity to store the source string.

- STRCAT()

This function is used to concatenate two strings

```
char s1[12] = "hello";
char s2[] = "sagar";
strcat(s1, s2); // s1 now contains "hellosagar" <no space in between>
```

- STRCMP()

This function is used to compare two strings.

- 0 if the strings are equal,
- a negative value if the first mismatching character in the first string has a lower ASCII value than the second string, and
- a positive value if the first mismatching character in the first string has a higher ASCII value than the second string.

```
strcmp("far", "joke"); // Negative value
strcmp("joke", "far"); // Positive value
```

Example:

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

int main() {
    // Define strings to compare
    char str1[] = "far";
    char str2[] = "joke";
    char str3[] = "joke";

    // Compare str1 and str2
    int result1 = strcmp(str1, str2);
    if (result1 == 0) {
        printf("'%s' and '%s' are equal.\n", str1, str2);
    }
    else if (result1 < 0) { // Negative value.
        printf("'%s' is less than '%s'.\n", str1, str2);  // Expected result
    }
    else {
        printf("'%s' is greater than '%s'.\n", str1, str2);
    }

    // Compare str2 and str1 (reverse comparison)
    int result2 = strcmp(str2, str1);
    if (result2 == 0) {
        printf("'%s' and '%s' are equal.\n", str2, str1);
    }
    else if (result2 < 0) {
        printf("'%s' is less than '%s'.\n", str2, str1);
    }
    else { // Positive value.
        printf("'%s' is greater than '%s'.\n", str2, str1);  // Expected result
    }

    // Compare str2 and str3 (same strings)
    int result3 = strcmp(str2, str3);
    if (result3 == 0) { // 0.
        printf("'%s' and '%s' are equal.\n", str2, str3);  // Expected result
    }
    else if (result3 < 0) {
        printf("'%s' is less than '%s'.\n", str2, str3);
    }
    else {
        printf("'%s' is greater than '%s'.\n", str2, str3);
    }

    return 0;
}
```

PART 9 – STRUCTURES

Array and strings → Similar data (int, float, char). Structures can hold → Dissimilar data. A C structure can be created as follows:

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

struct employee {
    int code; // This declares a new user defined data type!
    float salary;
    char name[10];
}; // semicolon is important
int main() {
    struct employee e1; creating a structure variable/object.
    // We can use this user defined data type as follows:
    strcpy(e1.name, "Sagar");
    e1.code = 100;
    e1.salary = 71.21212;

    printf("\nEmployee Details:\n");
    printf("Name: %s\n", e1.name);
    printf("Code: %d\n", e1.code);
    printf("Salary: %.2f\n\n", e1.salary);

    return 0;
}
```

```
Output:

Employee Details:
Name: Sagar
Code: 100
Salary: 71.21
```

So, a structure in C is a collection of variables of different types under a single name.

Quick Quiz: Write a program to store the details of 3 employees from user defined data. Use the structure declared above.

♠ WHY USE STRUCTURES?

We can create the data types in the employee structure separately but when the number of properties in a structure increases, it becomes difficult for us to create data variables without structures. In a nutshell:

a. Structures **keep the data organized.**

b. Structures **make data management easy** for the programmer.

♠ ARRAY OF STRUCTURES

Just like an array of integers, an array of floats and an array of characters, we can create an array of structures.

Example:

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

struct employee {
    int code;
    float salary;
    char name[20];
};
```

```
int main() {
    // Declare an array of structures to store information for 3 employees
    struct employee facebook[3];

    // Initialize data for each employee
    strcpy(facebook[0].name, " bear");
    facebook[0].code = 101;
    facebook[0].salary = 50000.0;

    strcpy(facebook[1].name, "grylls");
    facebook[1].code = 102;
    facebook[1].salary = 55000.0;

    strcpy(facebook[2].name, "Sagar");
    facebook[2].code = 103;
    facebook[2].salary = 60000.0;

    // Display information for each employee
    printf("Employee Details:\n");
    for (int i = 0; i < 3; i++) {
        printf("Employee %d:\n", i + 1);
        printf("Name: %s\n", facebook[i].name);
        printf("Code: %d\n", facebook[i].code);
        printf("Salary: %.2f\n\n", facebook[i].salary);
    }

    return 0;
}
```

```
Output:

Employee Details:
Employee 1:
Name:  bear
Code: 101
Salary: 50000.00

Employee 2:
Name:  grylls
Code: 102
Salary: 55000.00

Employee 3:
Name:  Sagar
Code: 103
Salary: 60000.00
```

♠ INITIALIZING STRUCTURES

Structures can also be initialized as follows:

```
 #include<stdio.h>
struct student {
    int roll;
    char name[20];
    float marks;
};
int main() {

    struct student s1 = { 1, "Sagar", 80.0 };
    struct student s2 = { 2, "Rahul", 90.0 };
    struct student s4 = { 0 }; //All elements set to 0

    printf("Student 1: Roll No: %d, Name: %s, Marks: %.2f\n", s1.roll, s1.name, s1.marks);
    printf("Student 2: Roll No: %d, Name: %s, Marks: %.2f\n", s2.roll, s2.name, s2.marks);
    printf("Student 4: Roll No: %d, Name: %s, Marks: %.2f\n", s4.roll, s4.name, s4.marks);

    return 0;
}
```

```
Output:

Student 1: Roll No: 1, Name: Sagar, Marks: 80.00
Student 2: Roll No: 2, Name: Rahul, Marks: 90.00
Student 3: Roll No: 3, Name: Rohit, Marks: 75.00
```

♠ STRUCTURES IN MEMORY

Structures are stored in contiguous memory locations. For the structure ‘s1’ of type struct employee, memory layout looks like this:

	1	“Sagar”	80.0
Address →	78810	78814	78818

In an array of structures, these employee instances are stored adjacent to each other.

♠ POINTER TO STRUCTURES

A pointer to structures can be created as follows:

```
struct Employee e1 = { 101, "Sagar", 55000.50 };
struct Employee* ptr;
ptr = &e1;
printf("Salary: %.2f\n", (*ptr).salary);
```

♠ ARROW OPERATOR

Instead of writing (*ptr).code, we can use arrow operator to access structure properties as follows:

```
(*ptr).salary
//or
ptr->salary
// here -> is known as the arrow operator.
```

Example with (*ptr) and ptr -> :

```
#include <stdio.h>

// Define a structure
struct Employee {
    int code;
    char name[20];
    float salary;
};

int main() {
    // Create a structure variable
    struct Employee e1 = { 101, "Sagar", 55000.50 };

    // Create a pointer to the structure
    struct Employee* ptr;

    // Assign the address of e1 to ptr
    ptr = &e1;

    // Access structure members using (*ptr).member notation
    printf("Using (*ptr).member notation:\n");
    printf("Code: %d\n", (*ptr).code);
    printf("Name: %s\n", (*ptr).name);
    printf("Salary: %.2f\n", (*ptr).salary);

    // Access structure members using ptr->member notation (arrow operator)
    printf("\nUsing ptr->member notation:\n");
    printf("Code: %d\n", ptr->code);
    printf("Name: %s\n", ptr->name);
    printf("Salary: %.2f\n", ptr->salary);

    return 0;
}
```

```
Output:

Using (*ptr).member notation:
Code: 101
Name: Sagar
Salary: 55000.50

Using ptr->member notation:
Code: 101
Name: Sagar
Salary: 55000.50
```

♠ PASSING STRUCTURE TO A FUNCTION

A structure can be passed to a function just like any other data type.

Example:

```
#include <stdio.h>

// Define a structure
struct Employee {
    int code;
    char name[20];
    float salary;
};
```

```
Output:

Displaying employee details (by value):
Employee Code: 101
Employee Name: Sagar
Employee Salary: 50000.00

Employee details after updating salary (by reference):
Employee Code: 101
Employee Name:  Sagar
Employee Salary: 60000.00
```

```
// Function to display employee details (passing by value)
void displayByValue(struct Employee emp) {
    printf("Employee Code: %d\n", emp.code);
    printf("Employee Name: %s\n", emp.name);
    printf("Employee Salary: %.2f\n", emp.salary);
}
// Function to update employee salary (passing by reference)
void updateSalaryByReference(struct Employee* emp, float newSalary) {
    emp->salary = newSalary;
}

int main() {
    // Initialize a structure variable
    struct Employee e1 = { 101, "Sagar", 50000.0 };

    // Passing structure to a function by value
    printf("Displaying employee details (by value):\n");
    displayByValue(e1);

    // Passing structure to a function by reference to update salary
    updateSalaryByReference(&e1, 60000.0);

    // Display the updated employee details
    printf("\nEmployee details after updating salary (by reference):\n");
    displayByValue(e1);

    return 0;
}
```

Quick Quiz: Complete this show function to display the content of employee.

♠ TYPEDEF KEYWORD

We can use the <u>‘typedef’ keyword to create an alias name</u> for data types in C.

‘typedef’ is more commonly used with structures.

Example:

```
#include <stdio.h>

// Define a structure with typedef
typedef struct {
    int code;
    char name[20];
    float salary;
} Employee;

void display(Employee emp) {
    printf("Employee Code: %d\n", emp.code);
    printf("Employee Name: %s\n", emp.name);
```

```
Output:

Employee Code: 101
Employee Name: Sagar
Employee Salary: 50000.00
```

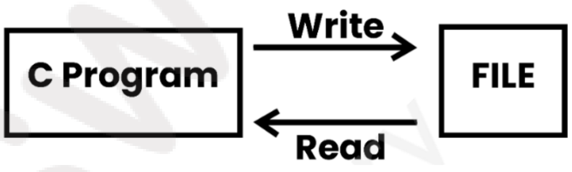
```c
    printf("Employee Salary: %.2f\n", emp.salary);
}

int main() {
    // Create a structure variable using the alias name "Employee"
    Employee e1 = { 101, "Sagar", 50000.0 };

    // Display employee details
    display(e1);

    return 0;
}
```

PART 10 – FILE I/O

The random-access memory is volatile, and its content is lost once the program terminates. In order to persist the data forever we use files.

A file is data stored in a storage device.

A C program can talk to the file by reading content from it and writing content to it.



♠ FILE POINTER

A "FILE" is a structure which needs to be created for opening the file.

A file pointer is a pointer to this structure of the file.

(FILE pointer is needed for communication between the file and the program).

A FILE pointer can be created as follows:

```c
FILE* ptr;
ptr = fopen("filename.ext"; "mode");
```

♠ FILE OPENING MODES IN C

C offers the programmers to select a mode for opening a file. Following modes are primarily used in C File I/O.

```
"r"     ->open for reading
"rb"    ->open for reading in binary
"w"     ->open for writing // If the file exists, the contents will be overwritten // deletes
        the previous content, then write.
"wb"    ->open for writing in binary
"a"     ->open for append  // If the file does not exist, it will be created // and if it
        exists then the system will write at the end of the file.
```

♠ TYPES OF FILES

Primarily, there are two types of files:

1. Text files (.txt, .c)

2. Binary files (.jpg, .dat)

♠ CLOSING THE FILE

It is very important to close the file after read or write. This is achieved using fclose as follows:

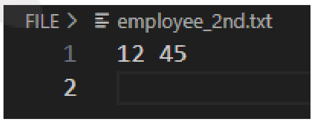
```
fclose(ptr);
```

This will tell the compiler that we are done working with this file and the associated resources could be freed.

♠ READING A FILE

Example: (taking variable values from a file):

```
#include<stdio.h>
int main() {
    FILE* ptr;
    ptr = fopen("employee_2nd.txt", "r");
    if (ptr == NULL) {
        printf("The file does not exist\n");
    }
    else {
        int num;
        fscanf(ptr, "%d", &num); // This will read an integer from file in num variables.
        printf("The value of num is: %d \n", num);

        fscanf(ptr, "%d", &num);
        printf("The value of num is: %d \n", num);

        // num value is changing because ptr is moving forward after reading a value.

        fclose(ptr);
    }

    return 0;
}
```



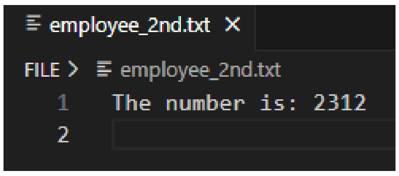
```
12 45
```

```
Output:

The value of num is: 12
The value of num is: 45
```

♠ WRITE TO A FILE

We can write to a file in a very similar manner like we read the file

```
#include<stdio.h>
int main() {
    FILE* ptr;
    ptr = fopen("employee_2nd.txt", "w");
    if (ptr == NULL) {
        printf("The file does not exist\n");
    }
    else {
        int num = 2312;
        fprintf(ptr, "The number is: %d\n", num);
        fclose(ptr);
    }
    // in write mode, if file does not exist, it will create a new file.
    /* in write mode, system first deletes the previous content of the file. this problem's
     solution is appending mode. */

    return 0;
}
```

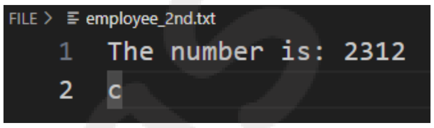


```
The number is: 2312
```

♠ FGETC() AND FPUTC()

fgetc and fputc are used to read and write a character from / to a file.

```c
#include<stdio.h>

int main() {
    FILE* ptr;
    ptr = fopen("employee_2nd.txt", "r");
    char c = fgetc(ptr); // c variable taking the first character of the text file.

    printf("The character is : %c\n", c); // The character is : T
    fclose(ptr);

    ptr = fopen("employee_2nd.txt", "a");
    fputc('c', ptr);

    return 0;
}
```


FILE > employee_2nd.txt

```
The number is: 2312
c
```

♠ EOF : END OF FILE

fgetc returns EOF when all the characters from a file have been read. So, we can write a check like below to detect end of file:

```c
#include<stdio.h>

int main()
{
    FILE* ptr;
    ptr = fopen("employee_2nd.txt", "r");

    while (1) {

        char ch = fgetc(ptr);
        printf("%c", ch);
        if (ch == EOF) { // when all the content of a file has been read break the loop!
            break;
        }
    }
}
```

PART 11 – DYNAMIC MEMORY ALLOCATION

C is a language with some fixed rules of programming. For example: **Changing the size of an array** is not allowed.

♠ DYNAMIC MEMORY ALLOCATION

Dynamic memory allocation is a way to allocate memory to a data structure during the runtime. We can use DMA function available in C to allocate and free memory during runtime.

♠ FUNCTION FOR DMA IN C

Following function are available in C to perform dynamic memory allocation:

1) free()
2) malloc()
3) calloc()
4) realloc()

♠ FREE() FUNCTION

We can use free() function to deallocate the memory. The memory allocated using calloc/malloc is not deallocated automatically.

Syntax:

```
free(ptr); //memory of ptr is released.
```

Quick Quiz: Write a program to demonstrate the usage of free() with malloc().

♠ MALLOC() FUNCTION

malloc stands for memory allocation. It takes number of bytes to be allocated as an input and returns a pointer of type void.

Syntax:

```
ptr = (int*)malloc(30 * sizeof(int))
```

The expression returns a null pointer if the memory cannot be allocated.

Example:

```
#include <stdio.h>
#include <stdlib.h> // Required for malloc and free

int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

 // Declaring a pointer to int that will store the address of the dynamically allocated memory
    int* ptr;

    // Using malloc to allocate memory for 'n' integers.
    // malloc returns a void pointer to the allocated memory, so we cast it to (int*).
    // 'n * sizeof(int)' calculates the total bytes required for 'n' integers.
    ptr = (int*)malloc(n * sizeof(int));

    // Checking if malloc was successful. If it fails, it returns NULL.
    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit the program if memory allocation fails
    }

    // Assigning values to the first three elements of the dynamically allocated array.
    // Note: Ensure that 'n' is at least 3 to avoid out-of-bounds access in a real scenario.
    ptr[0] = 10;
    ptr[1] = 20;
    ptr[2] = 30;

    // Printing the first three values in the array
    printf("%d %d %d\n", ptr[0], ptr[1], ptr[2]);

    // Freeing the dynamically allocated memory to avoid memory leaks
    free(ptr);

    return 0;
}
```

Quick Quiz: Write a program to create a dynamic array of 5 floats using malloc().

♠ CALLOC() FUNCTION

calloc stands for continuous allocation. It initializes each memory block with a default value of 0.

Syntax:

```
ptr = (float*)calloc(30, sizeof(float));
//allocates contiguous space in memory for 30 blocks (floats)
```

✓ Note: It the space is not sufficient, memory allocation fails, and a **NULL pointer** is returned.

Example:

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

int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Allocate memory for 'n' floats using calloc
    float* ptr = (float*)calloc(n, sizeof(float));

    // Check if calloc was successful
    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit the program if memory allocation fails
    }
    ptr[0] = 10.12;
    ptr[1] = 20.12;
    ptr[2] = 30.12;

    printf("%.3f %.3f %.3f\n", ptr[0], ptr[1], ptr[2]);

    // Freeing the allocated memory
    free(ptr);
    return 0;
}
```

Quick Quiz: Write a program to create an array of size n using calloc where n is an integer entered by the user.

♠ REALLOC() FUNCTION

Sometimes the dynamically allocated memory is insufficient or more than required. realloc is used to allocate memory of new size using the previous pointer and size.

Syntax:

```
ptr = realloc(ptr, newsize);
// or
ptr = realloc(ptr, 3 * sizeof(int));
```

Example:

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

int main() {
    int n = 1; // Initial size of the array

    // Allocate memory for 'n' integers using malloc
    int* ptr = (int*)malloc(n * sizeof(int));
```

```
        // Assign a value to the first element (since n is initially 1)
        ptr[0] = 10;

        // Reallocate the memory to expand the array to hold 'n + 5' elements
        int* ptr2 = (int*)realloc(ptr, (n + 5) * sizeof(int));

        // Assign values to the additional elements
        ptr2[1] = 20;
        ptr2[2] = 30;
        ptr2[3] = 40;
        ptr2[4] = 50;
        ptr2[5] = 60;

        // Print all the elements to verify the values
        for (int i = 0; i < (n + 5); i++) {
            printf("%d ", ptr2[i]);
        }

        // Free the reallocated memory
        free(ptr2);

        return 0;
    }
```

-------------- x --------------

PART 1- PRACTICE SET

1. Write a C program to calculate area of a rectangle:

    a. Using hard coded inputs.

    b. Using inputs supplied by the user.

2. Calculate the area of a circle and modify the same program to calculate the volume of a cylinder given its radius and height.

3. Write a program to convert Celsius (Centigrade degrees temperature to Fahrenheit).

4. Write a program to calculate simple interest for a set of values representing principal, number of years and rate of interest.

PART 2 – PRACTICE SET

1. Which of the following is invalid in C?

    a. int a=1; int b = a;

    b. int v = 3*3;

    c. char dt = '21 dec 2020';

2. What data type will 3.0/8 – 2 return?

3. Write a program to check whether a number is divisible by 97 or not.

4. Explain step by step evaluation of 3*x/y – z+k, where x=2, y=3, z=3, k=1

5. 3.0 + 1 will be:

    a. Integer.

    b. Floating point number.

    c. Character.

PART 3 – PRACTICE SET

1. What will be the output of this program

```
int a = 10;
if (a = 11)
        printf("I am 11");
else
        printf("I am not 11");
```

2. Write a program to determine whether a student has passed or failed. To pass, a student requires a total of 40% and at least 33% in each subject. Assume there are three subjects and take the marks as input from the user.

3. Calculate income tax paid by an employee to the government as per the slabs mentioned below:

Income Slab	Tax
2.5 – 5.0L	5%
5.0L - 10.0L	20%
Above 10.0L	30%

Note that there is no tax below 2.5L. Take income amount as an input from the user.

4. Write a program to find whether a year entered by the user is a leap year or not. Take year as an input from the user.

5. Write a program to determine whether a character entered by the user is lowercase or not.

6. Write a program to find greatest of four numbers entered by the user.

PART 4 – PRACTICE SET

1. Write a program to print multiplication table of a given number n.

2. Write a program to print multiplication table of 10 in reversed order.

3. A do while loop is executed:

   a. At least once.

   b. At least twice.

   c. At most once.

4. What can be done using one type of loop can also be done using the other two types of loops – true or false?

5. Write a program to sum first ten natural numbers using while loop.

6. Write a program to implement program 5 using 'for' and 'do-while' loop.

7. Write a program to calculate the sum of the numbers occurring in the multiplication table of 8. (consider 8 x 1 to 8 x 10).

8. Write a program to calculate the factorial of a given number using a for loop.

9. Repeat 8 using while loop.

10. Write a program to check whether a given number is prime or not using loops.

11. Implement 10 using other types of loops.

PROJECT 1: NUMBER GUESSING GAME

We will write a program that generates a random number and asks the player to guess it. If the player's guess is higher than the actual number, the program displays "Lower number please". Similarly, if the user's guess is too low, the program prints "Higher

number please".

When the user guesses the correct number, the program displays the number of guesses the player used to arrive at the number.

Hint: Use loop & use a random number generator.

PART 5 – PRACTICE SET

1. Write a program using function to find average of three numbers.

2. Write a function to convert Celsius temperature into Fahrenheit.

3. Write a function to calculate force of attraction on a body of mass 'm' exerted by earth. Consider g = 9.8m/s2.

4. Write a program using recursion to calculate n^th element of Fibonacci series.

5. What will the following line produce in a C program:

```
int a = 4;
printf("%d %d %d \n", a, ++a, a++);
```

6. Write a recursive function to calculate the sum of first ‘n’ natural numbers.

7. Write a program using function to print the following pattern (first n lines)

```
*

* * *

* * * * *
```

PART 6 – PRACTICE SET

1. Write a program to print the address of a variable. Use this address to get the value of the variable.

2. Write a program having a variable ‘i’. Print the address of ‘i’. Pass this variable to a function and print its address. Are these addresses same? Why?

3. Write a program to change the value of a variable to ten times of its current value.

4. Write a function and pass the value by reference.

5. Write a program using a function which calculates the sum and average of two numbers. Use pointers and print the values of sum and average in main().

6. Write a program to print the value of a variable i by using “pointer to pointer” type of variable.

7. Try problem 3 using call by value and verify that it does not change the value of the said variable.

PART 7 – PRACTICE SET

1. Create an array of 10 numbers. Verify using pointer arithmetic that (ptr+2) pointsto the third element where ptr is a pointer pointing to the first element of the array.

2. If S[3] is a 1-D array of integers then *(S+3) refers to the third element:

   (i) True.

   (ii) False.

   (iii) Depends.

3. Write a program to create an array of 10 integers and store multiplication table of 5 in it.

4. Repeat problem 3 for a general input provided by the user using scanf.

5. Write a program containing a function which reverses the array passed to it.

6. Write a program containing functions which counts the number of positive integers in an array.

7. Create an array of size 3 x 10 containing multiplication tables of the numbers 2,7and 9 respectively.

8. Repeat problem 7 for a custom input given by the user.

9. Create a three–dimensional array and print the address of its elements in increasing order.

PART 8 – PRACTICE SET

1. Which of the following is used to appropriately read a multi-word string.

    1. gets()
    2. puts()
    3. printf()
    4. scanf()

2. Write a program to take string as an input from the user using %c and %s confirm that the strings are equal.

3. Write your own version of strlen function from <string.h>

4. Write a function slice() to slice a string. It should change the original string such that it is now the sliced string. Take 'm' and 'n' as the start and ending position for slice.

5. Write your own version of strcpy function from <string.h>

6. Write a program to encrypt a string by adding 1 to the ascii value of its characters.

7. Write a program to decrypt the string encrypted using encrypt function in problem 6.

8. Write a program to count the occurrence of a given character in a string.

9. Write a program to check whether a given character is present in a string or not.

PART 9 – PRACTICE SET

1. Create a two-dimensional vector using structures in C.

2. Write a function 'sumVector' which returns the sum of two vectors passed to it. The vectors must be two–dimensional.

3. Twenty integers are to be stored in memory. What will you prefer- Array or structure?

4. Write a program to illustrate the use of arrow operator → in C.

5. Write a program with a structure representing a complex number.

6. Create an array of 5 complex numbers created in Problem 5 and display them

with the help of a display function. The values must be taken as an input from the user.

7. Write problem 5's structure using 'typedef' keywords.

8. Create a structure representing a bank account of a customer. What fields did you use and why?

9. Write a structure capable of storing date. Write a function to compare those dates.

10. Solve problem 9 for time using 'typedef' keyword.

PART 10 – PRACTICE SET

1. Write a program to read three integers from a file.

2. Write a program to generate multiplication table of a given number in text format. Make sure that the file is readable and well formatted.

3. Write a program to read a text file character by character and write its content twice in separate file.

4. Take name and salary of two employees as input from the user and write them to a text file in the following format:

   i. Name1, 3300

   ii. Name2, 7700

5. Write a program to modify a file containing an integer to double its value.

PROJECT 2: SNAKE, WATER, GUN

Snake, water, gun or rock, paper, scissors is a game most of us have played during school time. (I sometimes play it even now). Write a C program capable of playing this game with you. Your program should be able to print the result after you choose snake/water or gun.

PART 11 – PRACTICE SET

1. Write a program to dynamically create an array of size 6 capable of storing 6 integers.

2. Use the array in problem 1 to store 6 integers entered by the user.

3. Solve problem 1 using calloc().

4. Create an array dynamically capable of storing 5 integers. Now use realloc so that it can now store 10 integers.

5. Create an array of multiplication table of 7 upto 10 (7 x 10 = 70). Use realloc to make it store 15 number (from 7 x 1 to 7 x 15).

6. Attempt problem 4 using calloc().

-------------- x --------------