

## UNIT -I: Introduction to Algorithms and Programming Languages

**Algorithm** : The word **Algorithm** means “a process or set of rules to be followed in calculations or other problem-solving operations”. Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

Key features of Algorithms:

Key features of algorithm:

- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Examples of Algorithms:

In order to write an algorithm, following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm.
2. The constraints of the problem that must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output to be expected when the problem is solved.
5. The solution to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

Now let's design the algorithm with the help of above pre-requisites:

Algorithm to add 3 numbers and print their sum:

1. START
2. Declare 3 integer variables num1, num2 and num3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of variable sum
7. END

---

Q. explain about algorithm

What is algorithm and what are the advantages and disadvantages of algorithms

Explain key features of algorithm

---

Flowchart:

A [flowchart](#) is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.

### Flowchart Symbols

Flowcharts use special shapes to represent different types of actions or steps in a process. Lines and arrows show the sequence of the steps, and the relationships among them. These are known as flowchart symbols.






### Common Flowchart Symbols:

Rectangle Shape - Represents a process

Oval Shape - Represents the start or end

Diamond Shape - Represents a decision

Parallelogram - Represents input/output

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

---

Q. what is a flow chart and what are different flow chart symbols

---

the differences between algorithm and flow chart:-

S.NO	Algorithm	Flowchart
1.		Flowchart is a diagram created by

	Algorithm is step by step procedure to solve the problem.	different shapes to show the flow of data.
2.	Algorithm is complex to understand.	Flowchart is easy to understand.
3.	In algorithm plain text are used.	In flowchart, symbols/shapes are used.
4.	Algorithm is easy to debug.	Flowchart it is hard to debug.
5.	Algorithm is difficult to construct.	Flowchart is simple to construct.
6.	Algorithm does not follow any rules.	Flowchart follows rules to be constructed.
7.	Algorithm is the pseudo code for the program.	Flowchart is just graphical representation of that logic.

---

Q. what are the differences between flowcharts and algorithms

---

Pseudo code:

Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is a "text-based" detail (algorithmic) design tool.

The rules of Pseudocode are reasonably straightforward. All statements showing "dependency" are to be indented.

An algorithm is a procedure for solving a problem in terms of the actions to be executed and the order in which those actions are to be executed. An algorithm

is merely the sequence of steps taken to solve a problem. The steps are normally "sequence," "selection," "iteration," and a case-type statement.

In C, "sequence statements" are imperatives. The "selection" is the "if then else" statement, and the iteration is satisfied by a number of statements, such as the "while," "do," and the "for," while the case-type statement is satisfied by the "switch" statement.

The main difference between pseudocode and algorithm is that pseudocode describes the flow of the program while the algorithm is a step-wise procedure written to simplify and provide a solution to a given problem. Pseudocode is the basis of the algorithm while the algorithm is the basis of any programming language.

Q. what is pseudocode?

Compare pseudocode and algorithms

\*\*\* there could be a question on three things together like....

Write about pseudocode, algorithms and flow charts

---

\*\*this topic is foundation to rest of the syllabus

Programming Languages :

a program is defined as set of instructions. The programmers give instructions to system to fulfill the requirement of user, is called programming. The computer languages are used to develop programs, i.e. to give instructions to system.

The computer languages are classified into two types.

- i) Low level languages
- ii) High level languages

The languages that are capable of accessing processor features directly are called low level languages. With respect to systems, it is said that the system

understands low level languages. There is a problem with low level languages that a program designed in low level language for a specific modal processor cannot be executed on other modal processors because of change in features. This is called hardware dependency.

To overcome this problem of hardware dependency, high level languages are developed. But high level languages cannot access processor features directly, so they cannot be executed directly in system. With respect to system, it is said that the system doesn't understand high level languages.

For execution the high level programs must be converted into low level code called machine code with the help of compilers or interpreters.

The compilers and interpreters are software mechanisms used to convert high level code into processor appropriate low level code. The compilers convert high level code into low level code at once and stores as a file on hard disk. The interpreters convert line after line and loads into processor for execution directly. The interpreters won't create an extra file. When compared compilers are faster than interpreters.

The C language is provided with two compilers. The first compiler converts high level code into intermediate code and saves with extension “.obj” called object file and the second compiler converts intermediate code into low level code, saves with extension “.exe” and further loads into memory for execution. With this two phases of conversion mechanism, the C provides security to high level code.

---

Generation of Programming Languages : 1G: the first generation languages were machine level programming languages. In these, the instructions were directly given input through front panel switch of the computer system. These were in binary digits, and user gives input directly in binary system using either zeros or ones. No symbols were used. No compilation or assembling was required for this. It was very difficult for human to learn it and use it.

2G: The assembly language was classified into 2G. In this, the instructions were not in binary system. Rather English letters / words were used to give instructions. But when a program is written, it is need to be converted into machine code using ASSEMBLERS (like compilers in C). These are machine dependent.

3G: The third generation languages are machine independent. These are more user-friendly. When the program is written in 3GL, it needs to be converted into machine code using compilers or interpreters.

The 3GL are further classified into procedure-oriented languages, structured programming languages, object oriented languages etc. Before development these 3GL, the classification of “generation of languages” was never discussed and never recorded in technical books. The languages C, C++, java etc are examples of 3GL. The programs designed are more based on algorithms etc.

4GL: These are more specialized towards specific purposes. These are used for report generations from databases, mathematical optimization etc.

5GL: In these, the languages concentrate more on problem solving than the logic implemented. The algorithms are not much bothered. These are used in Artificial Intelligence

\*\* now a days, the term generation of languages is not much used in the IT market. Rather the classification of languages like structured programming languages , object oriented programming system are much used in the IT market.

---

Q. explain different generation of languages

---

Structured programming languages: when a project is developed, it contains LAKHS of lines of code. As it is difficult to implement such vast program as one, the 3GL provide a support of dividing project into modules (parts).

The languages that support dividing of project into modules called programs, and further into sub modules called functions, and link them together to execute as a software project are called structured programming languages.

The general standards of structured programming are:

1. They have one entry (starting) point and one exit (ending) point.
2. They support sequential execution of program
3. They doesn't support jumps in the program.
4. They execute instruction by instruction

So, these programs are simple, and can be well maintained. The advantages are :

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

---

Q. what is structured programming?

---

## Introduction to C

C is a high level programming language developed by a team of people under leadership of Dennis Ritchie in the year 1972. It was developed at AT &T's bell laboratories. It was developed on the systems consisting of UNIX operating system and later 83% of the UNIX is reconstructed in C. It is developed by combining the features of old languages B & BCPL and some of the own features of Dennis Ritchie.

Though C is a high level programming language, it can access some of processor features, so C is called middle level language.

General structure of C program



Comment section

Preprocessor

Global data members (variables) (if required)

Functions including statements and expressions

The comment section can be placed wherever required. These are used to make sure programmer understands what he does in program when revisiting. In C++ editor, the comments can be either single lined or multi lined. The single line comments are placed with // and multi lined comments are placed within /\* \*/

Ex: //this is single line comment

/\* this is multi-

Lined comments \*/

The Preprocessor is a location where the programmer can link library files or develop MACROs etc. the pre-processor is identified with a # symbol to runtime environment. The directive “include” is used to link library files and “define” is used to develop macros.

Ex:

# include< stdio.h >

# define PI 3.142857

Example program:

/\* the first program to display welcome message \*/

#include<stdio.h>

void main(){

printf(“Welcome to C World”);

}

Q. give an introduction to C language

## Files used in C Program

When a C program is developed, it is saved with extension .c and called source file. The C has two compilers, the first compiler converts high level code into intermediate code and saves with extension .obj and called object file and the second compiler converts object code into low level code and saves with extension .exe called executable file.

So when a C program is developed and executed, there will be 3 files created, a source file, object file and executable file. Along with these the C has library files called header files that are saved with extension .h that are included in preprocessor.

The C programs are compiled and executed in different ways. In case of UNIX environment, the command “gcc” is used to compile the code and in case of DOS, if TURBOC is installed, “tcc” command can be used. But on windows and other environments, as a special editor is provided, it has menus for program compilation and execution.

---

## Key Words

Every programming language has its own set of pre-defined important words called keywords, used to give instructions to system. Each keyword is reserved for a specific purpose. The keywords can be used only for that purpose for which they are reserved. The C language has 32 keywords. These cannot be used as identifiers.

## Identifiers

Identifier is a name given to an entity in program. Generally the variables, functions, structures etc are identified with names and these names are called identifiers.

There are some rules for identifiers:

- ➔ In C, identifier must have less than 8 characters (on unix it is < 8, but in windows it can be up to 32 characters)
- ➔ Identifier must start with alphabet and can have numbers in between.
- ➔ No special symbol including space is permitted other than underscore
- ➔ Keywords are not permitted

➔ Duplicate names not permitted

## Basic Data types

The data types are used to specify how the data is to be treated. These are basically used while declaring variables.

char : this is used to store text data. The variable of char data type occupies 1 byte memory in RAM and can store between -128 to 127 (ASCII value specification), i.e. one single character.

int : used to declare variables of integer type. It stores whole numbers and occupies 2 bytes memory (on 64 systems it occupies 4 bytes). Its range is between -32768 and 32767.

float : used to store real numbers. It occupies 4 byte memory, and supports 6 precisions only. This is used for general calculations.

double: used to store real numbers and occupied 8 bytes memory. It supports 10 precisions in the value. This is used for scientific calculations.

## Format Specifiers

%c : single char

%s : string (group of characters)

%d : decimal format

%i : integer format

%u : unsigned format

%o : octal format

%x : hexa-decimal format

%f : floating value

%e : scientific notation

## Variables

During program execution, the programmer needs to reserve some memory in RAM, to store data into that, to do process on it. As this reserved memory is capable of varying its value it is called a variable. The variables are declared for data types, and identified with an identifier (name).

Ex : int x; here 2 bytes memory is reserved of int type and it is named “x”.

## Constants

Constants are something that don't change its value. In C, the constants are classified into two types.

Constant variables: the variables declared with keyword "const" are called constant variables. These once declared and assigned with a value, can not be modified during program execution.

Ex : `const int x=5;`

Symbolic constants: in C, the macros developed are called symbolic constants.

Ex: `# define PI 3.142857`

In general practice, the symbolic constants are specified in upper case letters.

The constant variables when declared, they occupy some space in main memory based on its data type. But the symbolic constants do not occupy any memory.

---

Q. write about tokens of C

Write about different components used in programming

---

## IO Statements

The User Interface is providing interactivity between user and system. It is managed with IO statements (instructions). The libraries `stdio.h` and `conio.h` have functions pre-defined to handle IO operations.

`printf()` : is a standard output function, used to produce output onto monitor.

Syntax:

`printf("message to be displayed");`

`printf("message & format specifiers", source variables / values);`

ex:

`printf("Welcome to C");`

`scanf()` : is standard input function, used to read data from keyboard.

Syntax:

`scanf("format specifiers", address of variables);`

The main memory (RAM) includes no. of bytes memory , where each byte is identified with a byte number. When a variable is declared, the first byte number of reserved memory is called address of that variable and it is retrieved with the operator & in C.

Ex:

```
int a;  
scanf("%d",&a);
```

Q. explain basic IO statements in C

## Operators in C

The operators are used to develop process.

Operators are basically classified into three types based on number of operands they take.

Unary operators, Binary Operators and Ternary Operator.

Unary Operators : the operators that take single operand are called unary operators.

++, --, - (negation operator).

## Binary Operators :

The operators that take two operands are called binary operators. These are further classified into five types based on their purpose, viz.

- Arithmetic Operators : these are used to perform basic arithmetic operations on numbers.

+, - , \*, / and % (Modulus)

NOTE: The modulus operator can be used only with integer data type.

- Comparison operators: these are also called relational operators. These operators take two operands and give a result of either true or false and called Boolean values

<, <=, >, >=, == , !=.

These expressions developed using these are called either a condition or a Boolean expression.

➤ Logical Operators: these operators are used to combine two conditions.

&& (Logical and), || (logical or) and ! (logical not).

➤ Assignment Operator: this is simple =. This operator assigns R.H.S value to L.H.S variable. So L.H.S. must be a variable.

For example, `int a = 5;` is a correct statement. But `int a; 5 = a;` is wrong.

The arithmetic operators along with assignment operator perform dual operations. For example `a = a + 2` can be written as `a += 2`.

The operators `+=`, `-=`, `*=`, `/=`, `%=` can be used.

Ternary operator: the operator is `?` :

Syntax : `(condition) ? statement1 : statement2;`

The condition is developed using comparison operators, and they give a result of either true or false. Based on the result, if the result is true, the statement 1 will be executed by skipping the statement2, and vice versa.

Ex : for displaying user about biggest of two numbers, the code would look like,

```
(a > b) ? printf("a is big") : printf("b is big");
```

---

Q. what are different operators in C

---

program example

```
/* program to add two numbers */
#include <stdio.h>
#include <conio.h>
void main(){
    int a,b,c;
    clrscr();
    printf("Enter two numbers");
```

```

scanf("%d%d",&a,&b);
c=a+b;
printf("sum is %d",c);
}

```

---

## Type Conversion and Type Casting:

### Type Casting:

In typing casting, a data type is converted into another data type by the programmer using the casting operator during the program design. In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called narrowing conversion.

### Syntax/Declaration:-

destination\_datatype = (target\_datatype)variable;

() : is a casting operator.

target\_datatype: is a data type in which we want to convert the source data type.

example –

```
float x;
```

```
int y;
```

```
y=( int ) x;
```

here we are converting float(source) data type into int (target) data type.

## 2. Type conversion :

In type conversion, a data type is automatically converted into another data type by a compiler at the compiler time. In type conversion, the destination data type cannot be smaller than the source data type, that's why it is also called widening conversion. One more important thing is that it can only be applied to compatible data types.

example –

```
int x=30;
```

```
float y;  
y=x; // y==30.000000.
```

---

Q. explain what is type casting and type conversion

---

## UNIT – II

### Control structures and functions

Introduction to decision control statements :

When program executes, it gets executed line after line. But when some part of the program need to be executed based on conditions, then decision control statements are used. There are 2 decision control statements viz., Conditional Branching and Iteration.

Conditional Branching Statements:

In C, there are two decision control statements.

#### If-else branching

The if-else branching is used to divide the program into number of branches and execute them based on conditions. It can be used in following ways based on requirement.

When a single statement needs to be executed under any part, then it can be placed directly under either if or else. But when more than one statement is to be placed under any part, then all the statements must be grouped using { }.

1. Simple if statement : if the condition is true, then the statement under if part is executed.  
if ( condition )  
statement;
2. If-else statement: if the condition is true, the statement under if is executed and if the condition results false, statement under else gets executed



```
if( condition)
    statement;
else
    statement;
```

3. Nested if: Placing an if with in another is called nested if.

```
if(condition)
{
    if( condition )
        statement;
}
```

In this case, if the first if results true, then only the second if gets checked.

4. If-else-if: the else do not take condition. But if the code needs a condition to be placed with else, then an if can be placed along with else, to develop a chain called if-else-if ladder.

```
if( condition )
    statement;
else if( condition )
    statement;
else if( condition)
    statement;
else
    statement;
```

When the program needs to be divided into multiple branches and only one executed amongst them, then the if-else-if ladder is used.

---

Q. Write about branching control statements

What are different types of if-else statements

---

Switch-case statement

For decision making by comparing equality of values, the switch-case statement can be used. The switch keyword takes a variable and under switch there can be number of cases specified within a block.

The switch block executes from matching case till **end of switch block**. The switch block can also have a “default” option, that executes if no matching case exists.

```
switch(var)
{
    case 1: statements;
    case 2: statements;
    .....
    .....
    case n: statements;
    default : statements;
}
```

For controlled execution we may use the “break” keyword that break the switch block and passes control to rest of program.

The switch case in C works with ASCII code. The “int” or “char” data types can only be used in switch-case. The float , double cannot be used for comparing. So in C, the switch supports 256 cases. Later in C++ it supports more than 65000 cases.

In any case, the switch does not support duplicate cases written with in a block.

Ex:

```
char ch;
printf("Enter a character:");
scanf("%c",&ch);
switch(ch)
{
    case 'a':
    case 'e':
    case 'i':
```

```

    case 'o':
    case 'u':    printf("VOWEL"); break;
    default : printf("consonant");
}

```

---

Q. Write about switch-case statement

---

### **Iterative statements**

The loops are used for iteration. Iteration is repeated execution of process. The loops are controlled with a Boolean expression. If the Boolean expression results true, the loops continue its execution and otherwise stops.

Based on placement of the condition, the loops are classified into two types. i) entry controlled loops, ii) exit controlled loops. The loops that have Boolean expression at end of loop body are called entry controlled loops and that have at end of loop body are called exit controlled loops.

### **While statement:**

The while keyword takes a Boolean expression (condition) and till the expression results true, the loop continues its execution. The variable used in Boolean expression must be initialized prior its use and we must use appropriate counter in the loop body such that the Boolean expression becomes false at a state. Otherwise it turns into an infinite loop.

Syntax :

```

    Initialization;
    While(boolean expression)
    {
        Statements;
        Counter;
    }

```

We can directly place the Boolean value true instead of Boolean expression to make it as infinite loop.

The while statement is an entry controlled loop.

**do statement:** the do statement takes a while at end of loop, and it is called a do-while loop. This is similar to the do-while of C and C++.

The difference between the while and do-while is that the Boolean expression is placed at beginning of while and in do-while the Boolean expression is placed at end. So it is called an exit controlled loop.

When we use the do-while, the loop body is executed at least once even if the Boolean expression results false.

Syntax :

```
Initialization;
do{
    statements;
    counter;
}while(Boolean expression) ;
```

**for statement:**

the for is entry controlled loop and it takes the Boolean expression at beginning of loop body.

Syntax :

```
Initialization;
for( ; Boolean expression; )
{
    Statements;
    Counter;
}
```

As in for loop, the program control comes to the location that is before first ; we can logically shift initialization to that location and as every time the program control comes to the location that is after second ; we can logically shift the counter to that location, thus forming logical syntax of for as follows:

```
for(initialization ; condition ; counter)
    Statement;
```

### Nested loops

When a loop is placed within another loop, it is called nested loops. The inner loop executes no. of times for each run of outer loops. The count of total number of executions of statements is equal to the number of executions of outer loop multiplied by number of number of executions of inner loop.

continue: is a keyword when executes skips rest of loop once and passes control back to the loop.

Ex:

```
for(i=0;i<20;i++)
{
    if(i%3==0)
        Continue;
    printf("%d ",i);
}
```

The above code displays numbers between 0 and 20 by skipping multiples of 3.

break: is a keyword when executes, breaks the loop and passes control to rest of the program.

```
for(i=0;i<10;i++)
{
    if(i==5)
        break;
    printf("%d ",i);
}
```

Will display 0 1 2 3 4

---

Q. What are different types of loops

What are different iterative statements

---

goto: is keyword , used to make jumps in the program. Fr making jumps in the program, the location to which the program execution is to be jumped must be labeled and the same label is to be specified to the goto keyword. The goto makes program execution go to the location whose label is specified.

But one of the standards of the structured programming languages is “sequential execution of program”. With the goto statement, this standard is broken. So it is suggested to programmers not to user goto statements in program.

The programmers are to develop a controlled logic with goto statement, such that it should not develop infinite occurrence.

Ex:

```
void main()
{
    goto LABEL2;
    LABEL1:
        printf("@ label 1");
        goto LABEL3;
    LABEL2:
        printf("@ label 2");
        goto LABEL1;
    LABEL3:
        printf("completed");
}
```

---

Q. Write about goto statement

---

## **Functions**

### **Functions**

When a software project is developed, it includes lakhs of lines of code. Writing thousands of lines of code within one main() function leads to confusion and gives burden on system.

Almost all the high level languages provide a facility of dividing programs into modules called functions. A function is piece of code identified with a name.

A function has two aspects, a definition and a call. The function definition includes the process to be executed and the function call is invocation of definition. A function defined once can be called number of times.

Advantages of functions:

- ➔ Increase in modularity of program
- ➔ Increase in readability of program
- ➔ Easy identification of errors
- ➔ Easy debugging of code
- ➔ Reusability of code

Function declaration / prototyping:

In C, when a function defined later the call, i.e. when a function is called first and defined later, then it must be given with forward declaration, called prototyping. The forward declaration is an indication to Runtime Environment that a function is called prior its definition.

Syntax : `return_type fun_name( parameters ) ;`

The forward declaration can be done either globally or with in `main()` definition in declaration area. If it is declared within `main()` definition, then it can be called only with in `main()`. If declared globally, then it can be called anywhere in program.

Function Definition:

The Function definition includes the process to be done. The entire process to be done is grouped using braces and identified with a name. When the function is called the process given in definition executes.

Syntax:

```
return_type fun_name( formal arguments)
{
    Process to be done;
}
```

Function call:

The function call is invocation or activation of the definition. A function defined once, can be called number of times.

Syntax : fun\_name( actual arguments) ;

Passing arguments:

The functions are provided with ( ) to pass values from one function into another, from function call into function definition. The arguments given at function call are called actual arguments and the arguments at definition are called formal arguments. The actual arguments can be either variable names or values. The formal arguments are variables with individual declarations.

During program execution the values of actual arguments are copied into formal arguments, and the formal arguments are processed within definition without affecting actual arguments.

So, the number, data type and sequence of formal arguments must be matching with that of actual arguments.

Returning values:

The function definition after doing process can send values back from called function into calling function with the “return” keyword. The values can be returned for further use.

- ➔ The return can return only one value at a time
- ➔ The return is last statement to be executed in a function
- ➔ A function can have only one return executed.

The function’s return type is data type specification of the value being returned by the function. In C, the Runtime’s default data type is int. So, if no return type is specified, the C RE takes the return type as int. If the function won’t return any value, then the return type must be specified as “void”.

As the main() won’t return any value, it is generally returned on “void main()”.

Ex:

```
int add(int a, int b) ;
void main()
{   int a,b,c;
    printf(“Enter two numbers:”);
    scanf(“%d%d”,&a,&b);
    c=add(a,b);
```



```

        printf("sum is %d",c);
    }
    int add(int x, int y)
    {
        return x+y;
    }

```

---

Q. Write about functions

What is a function and explain function call, definition and declaration

---

SCOPE of variables:

The scope of variables are based on the location they are declared. The variables based on location of declaration, are classified into two types.

- ➔ Local variables: the variables declared with in a function definition are called local variables to that function. These have their scope only within that function. So they can be accessed in that function only. They are not accessible in other functions of the program, so the functions are given with ( ) to pass them as arguments.
- ➔ Global variables: the variables declared outside of all the functions are called global variable and they have their scope to entire program, so they can be accessed in all the functions. As these are accessible in all the functions, there is a chance of miss-usage of them, that leads to data corruption. So to provide security to data, it is suggested to programmers not to use global variables or reduce usage of global variables.

STORAGE CLASSES: The C has a concept called storage classes, that specifies the lifetime of the variable and its scope. The different storage classes are as follows.

- ➔ Auto variables : it is default storage class of C. when a variables is declared with data type keyword, it is by default auto variable. If it is

declared as local variable, it has its scope within that function and if declared as global, then it has its scope in the entire program. When declared these will have garbage values.

Ex: `auto int a;`

- ➔ **Extern variables:** In olden days, the programs used to be done in other languages like Cobol, Pascal etc. if there is a requirement of using code written in C, to be used in other language programs, then the code in C must be declared with keyword “extern”. In C, the variables and also functions can be declared as “extern”.

-> Ex: `extern int x;`

-> `extern void disp()`

```
{  
    Printf(“extern function”);  
}
```

- ➔ **Register variables:** during program execution, the values need to be transferred from main memory into CPU registers (processor cache), to undergo process. This data transfer between RAM and registers consumes much time. To reduce this time consumption, the variables can be directly declared within CPU registers with the keyword “register”. Though the time consumption of data transfer is reduced with register variables, the performance of the processor is also reduced. While programming, generally the variables that need to change their values frequently for number of times are declared as register variables. If CPU registers are not free, then the OS declares them as normal variables with in main memory.

In programming, as the loop controlling variables change their values frequently for number of times, only these are declared as register variables and as generally only integer type variables are used for loop control, in C, only int data type variables can be declared as register variables.

-> ex: `register int i;`

➔ Static variables: when a variable is declared as static within a function, the variable preserves its last update value even when it goes out of scope. It preserves its value and that can be used for the subsequent calls of the same function. Generally to reduce the usage of global variables, the programmers prefer using static variables.

-> ex : static int x;

---

Q. explain storage classes?

Q. what are the different storage classes?

---

Recursive Functions :

### RECURSIVE FUNCTIONS:

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Ex1:

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i) {

    if(i <= 1)
        return 1;
    return i * factorial(i - 1);
}

int main() {
    int i = 5;
```

```
printf("Factorial of %d is %d\n", i, factorial(i));  
return 0;  
}
```

Ex 2:

```
#include <stdio.h>  
  
int fibonacci(int i) {  
  
    if(i == 0) {  
        return 0;  
    }  
  
    if(i == 1) {  
        return 1;  
    }  
    return fibonacci(i-1) + fibonacci(i-2);  
}  
  
int main() {  
  
    int i;  
  
    for (i = 0; i < 10; i++) {  
        printf("%d \t", fibonacci(i));  
    }  
  
    return 0;  
}
```

---  
Q. Explain recursive function with an example  
---

## Arrays and Strings

### **Arrays:**

Introduction, Declaration of Arrays, Accessing elements of the Array, Storing Values in Array,

The array means arranged things. It is a collection of similar type of elements that have sequential memory. The term “data structure” is arranging data values in a specific order. There are different types of data structures and array is linear data structure.

When a program needs to store large number of data elements of similar type, then instead of using individual elements, the arrays can be preferred, and with arrays, data manipulation becomes easy.

In C-language, the [ ] symbol is used while declaring and manipulating the arrays. The [ ] are used to specify no. of elements to be grouped while declaring.

Syntax : data\_type array\_name[ int size];

Ex: int a[5];

When an array is declared, the C-RE gives index numbers to each element for identification as all elements are identified with same name. The index number of array elements start with ZERO. So the index of last element is always size-1.

The same [ ] are used to specify the index number of element while manipulating the array. i.e. the array members can be accessed with their index numbers.

Arrays can be stored with values either by initializing or by taking input. The initializing can be done in two ways.

int arr[5] = { 10, 20, 30,40, 50}; in this case, the number of initializers must be either matching with size of array or less than array. If less initializers given when size is specified, then the other elements are initialized with ZEROs.

or

int arr[ ]={ 10,20,30,40,50}; in this case, the size of array is based on number of initializers.

The loops can be used to read data from keyboard and store into the array.

Ex:

```
for(i=0;i<5;i++)  
    scanf("%d",&arr[i]);
```

Calculating the length of the Array : if the array elements are initialised, then we take help of sizeof() operator to find length of array.

For example if an array is initialised as....

```
Int a[]={1,3,2,5,6,9,8,7,66,55,12,32}, length;
```

Then the logic

Length=sizeof(a) / sizeof(a[0]); gives length of that array.

Operations that can be performed on Array,

There are different operations that can be performed with arrays like,

traversal : moving across the array elements using a loop.

Copying: copying elements from an array or into an array

Reversing : reversing array elements

sorting: arranging array elements either in ascending or descending order.

inserting / deleting : inserting or deleting values from array

searching : searching for given value

merging : merging multiple arrays into one array.

---

Q. Give an introduction to arrays

How to store values into array and access them

What is an array? What are the operations that can be done on arrays.

---

One dimensional array : when we need large data need to be stored sequentially and accessed sequentially, we prefer one dimensional array. In one dimensional array, each element will have one index referencing.

Ex: int arr[5];

Accessing one dimensional array: the one dimensional array elements can be accessed by specifying single index number for each element like arr[0], arr[1] etc. we can use a loop for easy accessing each element by traversing across each element as:

```
for(i=0;i<5;i++)
    printf("%d ", arr[i]);
```

Passing one dimensional array to function: we can also pass single dimensional array as argument to function. While passing single dimensional array to a function, we may specify the array size in formal arguments / forward declarations or we may not.

Ex:

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

```
void disp(int[ ],int);
```

```
void main()
{
    int a[]={10,20,30,40,50,60};
    int size=sizeof(a)/sizeof(a[0]);
    clrscr();
    disp(a,size);
}
```

```
void disp(int a[ ],int size)
{
    int i;
    for(i=0;i<6;i++)
        printf("%d ",a[i]);
}
```

Or

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

```
void disp(int[6]);
```

```
void main()
{
    int a[]={10,20,30,40,50,60};
    int size=sizeof(a)/sizeof(a[0]);
    clrscr();
    disp(a);
}
```

```
void disp(int a[6])
{
```

```

int i;
for(i=0;i<6;i++)
    printf("%d ",a[i]);
}

```

---

Q. Write about single dimensional arrays

How to use single dimensional arrays

How to use functions with single dimensional arrays

---

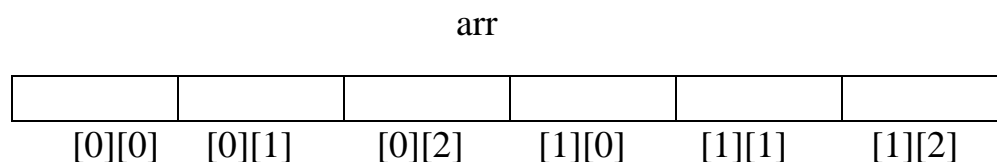
Two dimensional Arrays & Accessing two dimensional arrays :

2D and Multi dimensional arrays: the arrays can be used to manage large data by providing multiple index references, for easy storage and manipulation. Irrespective of number of dimensions specified, all the array elements will have sequential memory allocated.

In a 2-dimensional array the size of each dimension is specified with in a separate [ ].

Ex : int arr[2][3];

In the above 2-D array, the memory allocated is sequential as given below:



The above 2-D array can be manipulated using two loops, one to refer first index number and the second referring second index number.

```
for( i=0;i<2;i++)
```

```
for(j=0;j<3;j++)
```

```
    printf("%d ",arr[i][j]);
```

A 2-D array can be logically felt as rows and columns by displaying content row wise / column wise with below code.

```
for( i=0;i<2;i++)
```



```

{
    for(j=0;j<3;j++)
        printf("%d ",arr[i][j]);
    printf("\n");
}

```

It can be logically assumed in a tabular format as given below :

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]

arr

Similarly, the arrays can have any number dimensions, and the size of each dimension is specified with in a separate [ ].

Ex: int arr[2][3][2];

Irrespective of number of dimensions, the array elements will have sequential memory. But in practical live programming a 2-D array is much used as it can be used to store large data and managed in a tabular format. The multi-dimensional arrays that have more than 2 dimensions are not much used in regular programming and it is difficult to represent them.

Passing two dimensional arrays to functions: While passing 2 D array as argument to a function, we must specify size of each dimension in forward declaration and also in formal arguments, as the array will have sequential memory, the size of each dimension becomes must while passing arguments.

Ex:

```

#include<stdio.h>
#include<conio.h>

void disp(int[2][3],int,int);

void main()
{
    int a[2][3]={10,20,30},{40,50,60}};
    clrscr();
    disp(a,2,3);
}

```

```

}

void disp(int a[2][3],int r,int c)
{
    int i,j;
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
}

```

---

Q. Write about 2D arrays dimensional arrays  
 How to use multi dimensional arrays  
 How to use functions with multi dimensional or 2D arrays

---

**Strings:** Introduction,

String is sequence of characters. In C, the strings are managed with an array of char data type.

Ex: char name[20];

We can initialize the string as char name[] = "Gnanambica";

When a string is assigned to a char array while initializing, there will be null value represented as '\0' stored at end of string for identification.

The C has a special format specifier %s to be used while reading or writing strings. The %s won't read spaces in a string. We can also specify the characters to be read as format specifier instead of %s like "%[a-z]". The ^ (caret) can be used to specify the character NOT to be read. Many times programmers use "%[^\n]" as format specifier to read all symbols spaces etc till user hits ENTER.

For easy manipulation of strings, the string.h library has number of pre-defined functions like:

strcpy( char dest[ ], char source[ ] ) ; copies source [ ] into dest[ ].

strcat(char dest[ ], char source[ ] ) ; concatenates source [ ] to dest[ ].

strlen(char[ ]) : returns length of string

strcmp(char ch1[ ], char ch2 [ ]) : compares two strings lexicographically with their ASCII values.

strrev(char ch[ ]) : reverses given string.

For character handling, the ctype.h library has number of pre-defined functions like:

islower(ch) : returns true if lowercase letter and false if not

isupper(ch) : returns true if upper case letter

tolower(ch) : converts into lowercase

toupper(ch) : converts into uppercase

isspace(ch) : returns true if it is a space

isalpha(ch) : returns true if it is alphabet

isdigit(ch) : returns if it is a digit

isalnum(ch) : returns true if ch is an alphabet or a number

---

Q. write about Strings

Explain different string functions

Explain how to strings are managed in C

---

## **UNIT 4**

### **Pointers**

Understanding Computer Memory :

When the term memory is used in programming, it generally refers the RAM, the main memory of the system. When the program is compiled, the compiler decides how much memory is required for running that application. When the program execution starts, the application gets reserved with some memory in RAM to have its code loaded and memory allocated for the data members (variables).

The application's memory is logically divided into four parts.

- ➔ Code part
- ➔ Static / global variables part
- ➔ Stack memory
- ➔ Heap memory

The machine code, in case of C, the executable file that has extension “.exe” is loaded into CODE PART of the application's memory.

The static variables and global variables of the program are allocated in a special memory reserved for them.

The memory part in which the local variables are allocated with memory is called STACK. The memory in stack is managed by the operating system itself. The programmer need not specify explicitly how much memory is to be allocated and this memory is released automatically soon after the program execution completed. The stack size is fixed based on number of variables declared. Once program execution starts, the stack size can not be either increased / decreased.

The memory part in which the memory is assigned dynamically is called HEAP MEMORY. This dynamic memory is managed with pointers. The memory assigned using malloc() etc functions, reserves the memory in the heap memory of application's reserved memory. The heap size can be modified. It can be increased / decreased and also released.

The dynamic memory managed in HEAP, must be allocated explicitly and must be released explicitly. If the memory is not released explicitly, then that memory cannot be further used for any other purpose. And the heap size will be decreasing for all programs that affect all the processes. This case is called “MEMORY LEAK”. A good programmer should not permit “memory leak” in the program. So the dynamic memory assigned in HEAP must be released explicitly.

---

Q. Write about computer memory and application memory management?

Q. Explain how application memory is managed for each application?

Q. What are stack and heap memories and explain them?

---

## Introduction to Pointers, Declaring Pointer Variable, Pointer Expressions and Pointer Arithmetic operations:

The Pointer is a special data type used to manipulate values by holding address of the memory location. The pointer is declared with an ASTERISK. The pointers can be used in two ways, either by allocating memory dynamically to the pointer or by storing address of other static memory.

Ex:

```
void main(){
int a=5;
int *p;
p=&a;      //storing address of static memory (variable)
printf("%d",*p); //will display 5
}
```

The pointer manipulation is bit different from normal operations done on variables. As the pointer points to memory addresses, when the pointer is manipulated with asterisk, they manipulate the values in that addresses and when these are manipulated without asterisk, they manipulate the addresses. When the address in the pointer is modified, the pointer points to a new location whose address is stored within.

Ex:

```
void main(){
    int a,b,*p,*q;
    a=5;
    b=10;
    p=&a;
    q=&b;
    printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```

Now the above will display "a=5; b=10; \*p=5; \*q=10;"

Ex:

```
void main(){
    int a,b,*p,*q;
    a=5;
    b=10;
    p=&a;
    q=&b;

    p=q; //pointer manipulated without *

    printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```

Now the above will display "a=5; b=10; \*p=10; \*q=10;"

When the pointer is manipulated without \*, so the address held in the pointer "q" is copied into the pointer "p". So the pointer "p" now points to the new location. And the value in the location pointed by "p" is not modified.

Now the above will display "a=5; b=10; \*p=10; \*q=10;"

Ex:

```
void main(){
    int a,b,*p,*q;
    a=5;
    b=10;
    p=&a;
    q=&b;

    *p=*q;      //pointer manipulated with *

    printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```

Now the above will display "a=10; b=10; \*p=10; \*q=10;"

When the pointer is manipulated with \*, the values in the locations pointed by the pointers are manipulated. The variable "a" is assigned with value pointed by

the pointer “q” and the value is assigned “NOT THROUGH variable” but through the POINTER. So the value at location pointed by pointer “p” is modified, i.e. the value of “a” become 10.

Now the above will display “a=10; b=10; \*p=10; \*q=10;”

---

Q. explain pointers and their usage

Q. Explain about pointer arithmetic operations.

Q. what is the difference between normal arithmetic operations and pointer arithmetic operations

---

Null Pointers:

When a pointer is declared and not assigned with any address, it is suggested to store a null value in that pointer, so that it won't point to a garbage address. The stdio.h library has a pre-defined macro NULL, that can be assigned to the pointer.

Ex: `int *p; p=NULL;`

Passing Arguments to Functions using Pointer,

The functions when called, they can be passed either variables or addresses. Based on what is passed, the function calls are classified into two types.

Function call by value:

When calling functions, the local variables of calling functions are passed as arguments to called function as arguments and in called function, they are received as formal arguments and processed. The process done in the called function will not affect the values in actual arguments.

Ex:

```
void swap(int,int);
```

```
void main()
```

```
{
```

```
    int a,b;
```

```

    a=5;
    b=10;
    printf("before swapping a=%d b=%d\n",a,b);
    swap(a,b);
    printf("after swapping a=%d b=%d\n",a,b);
}
void swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
    printf("in swap function a=%d b=%d\n",a,b);
}

```

Output:

before swapping a=5 b=10

in swap function a=10 b=5

after swapping a=5 b=10

though the function swapped, it is not affected in actual arguments.

Function call by reference:

To make sure the functions make do the process on values stored in actual arguments, the programmer can pass address of actual arguments and receive them into the pointers declared as formal arguments and to do process through pointers. Though the pointers are declared in called function, as they hold address of actual arguments, when the process is done through pointers, they affect the actual arguments.

Ex:

```

void swap(int*, int*);
void main()
{
    int a,b;
    a=5;
    b=10;
    printf("before swapping a=%d b=%d\n",a,b);
}

```



```

        swap(&a,&b);
        printf("after swapping a=%d b=%d\n",a,b);
    }
void swap(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
    printf("in swap function a=%d b=%d\n",*a,*b);
}

```

Output:

before swapping a=5 b=10

in swap function a=10 b=5

after swapping a=10 b=5

here as the pointers in called function points to the address of actual arguments, the actual arguments are processed.

---

Q. Explain different ways of passing arguments to functions.

Q. what is function call by value and call by reference?

Q. Explain function call by reference with swap function example

---

Pointer and Arrays –

The pointers can be used to point to large memory like arrays.

```

void main(){
    int a[]={5,10,15,20,25};
    int *p;
    p=&a[0];
    printf("%d",*p); //will display 5
    p++;
    printf("%d",*p); //will display 10
}

```

```
}
```

The pointer arithmetic operation is different from normal arithmetic operation. It is done on number of bytes based on data type of the pointer. The `int*` when increased by 1 it gets increased by 2 bytes and the `float*` will increase by 4 bytes. In any case, the pointer moves to next element in the memory.

- So the programmer must use appropriate pointer in the program, the `int*` can be used to point only to int memory.

```
void main(){
    int a[]={5,10,15,20,25};
    int *p;
    p=&a[0];
    printf("%d",*p); //will display 5
    printf("%d",*p+1); //will display 6
}
```

Here it displays 6, as pointer has higher priority, first `*p` is evaluated and +1 done.

The priority can be changed using ( ).

```
void main(){
    int a[]={5,10,15,20,25};
    int *p;
    p=&a[0];
    printf("%d",*p); //will display 5
    printf("%d",*(p+1)); //will display 10
}
```

Here first `p+1` is done and then `*` applied to that new address. In `p+1`, as “p” is an int pointer, it will increase by 2 bytes, and points to next element.

When we declare an array in C, the C runtime creates a pointer with same name of the array and stores the base address of array into that pointer.

So in above example, when the array `int a[]` is declared, the RE has a pointer with name “a” itself. That built in pointer can be manipulated as of normal pointer.

```

void main(){
    int a[]={5,10,15,20,25};
    int i;
    for(i=0;i<5;i++)
        printf("%d ",*(a+i) );
}

```

Conclusion : in C or C++, we can declare arrays and use either array or pointer syntax and also, we can declare pointers and use array or pointer syntaxes. The difference is arrays are static memory and pointers support dynamic memory.

Passing Array to Function: we can pass an array to a function and receive it into a pointer and manipulate data using pointer.

Q. write about pointers and arrays

Q. what are the difference between arrays and pointers

Memory Allocation in C Programs, Memory Usage , Dynamic Memory Allocation:

In C, language, the memory management can be done in two ways.

1. Static memory management :

The memory once assigned at beginning of the program is called static memory and the static memory is allocated with memory in the stack and its size is fixed. It cannot be modified with its size and can not be released.

Ex: variables, arrays, structure / union variables etc.

2. Dynamic memory management: the memory that can be assigned during running of program is called dynamic memory. This need not be assigned at beginning of program. This can be assigned once program execution starts. So it is called dynamic memory. The dynamic memory can be increased /

decreased with its size and also released. The dynamic memory is managed with in heap of the application's memory.

The dynamic memory is managed with the help of pointers. Functions related with dynamic memory management are defined in the library "alloc.h" library.

`malloc()` : this function allocates specified number of bytes memory and returns its base address that can be held by a pointer.

This function must be type casted based on type of pointer used.

Ex:

```
int *p;  
p= (int*) malloc( sizeof(int) *5 );
```

here the `sizeof(int)` returns memory required for 1 integer variable and it is multiplied by 5, gives 10. The `malloc()` allocates 10 bytes memory and returns the base address of the reserved memory that is type casted as "int\*" and held by the pointer "p". Now the pointer "p" points to 5 integers memory.

`calloc()` : it is similar to `malloc()`, but the `malloc()` leaves garbage values in the reserved memory and the `calloc()` initializes all elements with zeros.

`realloc()` : is used to increase or decrease the size of memory reserved, but it is not much used, as it leads to data loss.

`free()` : this function is used release the memory assigned to the pointer.

Ex: `free(p);`

The memory assigned once to a pointer with `malloc()` or `calloc()` or `realloc()` is reserved with in heap of the memory, and it must be released with `free()`. If not released, then the reserved memory can not be further used by the OS, and this is called "MEMORY LEAK" situation.

A pointer when freed, it still holds the garbage address, and mis-behaves. Such pointer is called DANGLING pointer. A good programmer should not leave dangling pointers in program. So once a pointer is freed, then it must be assigned with NULL.

---

Q Write about different memory managements

Q. what is dynamic memory management and explain its functions

---

Drawbacks of Pointers.

Advantages of pointers :

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allow to perform dynamic memory allocation and de-allocation.

Drawbacks of pointers in c:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- Dangling pointers lead to memory crash
- If pointers are updated with incorrect values, it might lead to memory corruption.
- Basically, pointer bugs are difficult to debug. Its programmers responsibility to use pointers effectively and correctly.

---

Q explain advantages and disadvantages of pointers

Q. what are the drawbacks of pointers

---

Structures: Introduction to structures, Nested Structures.

Structure is a user defined data type, used to group no. of elements of different data types with different names. A structure is defined with the keyword “struct”.

Syntax :

```
struct <structure_name>
{
    Member variables;
};
```

Ex:

```
struct emp
{
    int no, sal;
    char name[20];
};
```

In above declaration, “struct emp” is data type. A variable has to be declared for this data type to make use of its members. The structure variable is declared as normal variable with syntax `data_type var_name;`. As here “struct emp” is data type, the variable is declared as “struct emp e;”.

In above declaration the “e” is called structure variable and it is allocated with memory for its members no, sal and name sequentially. The structure members are accessed with structure variable with a DOT “.” operator called PERIOD.

Ex:

```
e,no=5;
strcpy(e.name,“your_name”);
e.sal=5000;
```

Nested structures:

We can declare a structure variable as member within another structure, like a normal other variables, and it is called nested structures.

In this case, to access the nested structure’s members, the period operator is used twice as “`sttuct_var . member_struct_var. member_var`”

struct address

```

{
    char city[20];
    int pin;
};
struct employee
{
    int no;
    char name[20];
    struct address add;
};

void main ()
{
    struct employee e;
    printf("Enter number:");
    scanf("%d",&e.no);
    printf("Enter name:");
    scanf("%s",&e.name);
    printf("Enter City:");
    scanf("%s",&e.add.city);
    printf("Enter PIN code:");
    scanf("%d",&e.add.pin);

    printf("%d %s %s %d",e.no,e.name,e.add.city,e.add.pin);

}

```

Arrays of structures:

The structures can be created with arrays of structure variables. In that case, a loop is used to refer the index of structure variable's array.

Ex:

```

struct emp{
    int no,sal;

```

```

        char name[20];
    } ;

void main()
{
    struct emp e[5];
    int i;
    for(i=0;i<5;i++)
    {
        printf("Enter 5 employee data");
        scanf("%d%s%d",&e[i].no,&e[i].name, &e[i].sal);
    }

    printf("Employee detailes are \n");
    for(i=0;i<5;i++)
        printf("%d  %s  %d\n",e[i].no, e[i].name,e[i].sal);
}

```

### Structures & functions:

The structure can be declared either as local to a function or global. If a structure is defined as a local structure, it can be created with variable only within that function. If it is declared globally, it can be created with variable in any function. It is generally preferred to declare structures as global.

The structure variables can also be declared as global or as local. If the structure variable is declared as local variable, then to access it in other functions, it need to be passes as arguments from function call into function definition.

Ex:

```

struct emp{
    int no,sal;
    char name[20];
} ;

void disp(struct emp);

```



```

void main()
{
    struct emp e;
    printf("Enter employee data");
    scanf("%d%s%d",&e.no,&e.name, &e.sal);
    disp(e);
}
void disp(struct emp e)
{
    printf("%d %s %d",e.no,e.name.e.sal);
}

```

\*\* There can be 5 marks questions in the above each concept.. like

Q. write about structures and functions

Q. explain array of structures

Q. explain nested structures.

\*\* Here you may need to explain with code.

Union:

Unions are similar to structures but declared with keyword “union”. Similar to structures, the unions are created with variables to make use of its members. But when a union variable is declared, the largest member of the union is allocated with memory and shared by all other members.

For example:

```

union abc
{
    int x;
    float y;
};

```

If a union is declared as above and created with a variable, “union abc a;”, then the union variable “a” is allocated with 4 bytes memory and shared by both members “x” and “y”. the first 2 bytes is called “x” and all 4 bytes together is called “y”. thus unions provide better memory management.

But there is a restriction on unions. As the members share same memory, programmer can use only one member at a time. In above case, if a value stored into member “x” as “a.x=5;”, then “a.y” cannot be used. And if a.y is stored with a value, then a.x cannot be used. At a given time, user can use only one member of the union variable.

Programmer has to apply his own logic while using unions like in below example.

```
union person{
    int pensioner_id, employee_id;
};
void main(){
    union person p;

}
```

In above case, the union variable “p” is assigned with only 2 bytes memory and the same is called pensioner\_id and the same itself is called employee\_id. A person can have either employee\_id or pensioner\_id. In this case, as there is a requirement of using only one at a time, the programmers can prefer unions instead of structures. Similar to structures, the unions can also be declared with array of variables. We can declare nested structures or nested unions.

\*\*\* there can be a question like....

Q. explain unions.

Q. What are the differences between structures and unions?

\*\*

## Enumerated Data Types

It is a user-defined data type that consists of integer values, and it provides meaningful names to these values. The use of enumerated data type in C makes the program easy to understand and maintain. The enumerated data type is defined by using the “enum” keyword.

Ex:

```
enum week{ sun,mon,tue,wed,thu,fri,sat };
```

in above declaration, the first value sun is given with ZERO by default and rest of the names are given with 1,2,3 etc numbers by default in a sequence.

`printf(“%d”,sun) ;` will display 0

The programmer can also assign required values for each or to the first name. if a value is given only to the first name, then rest elements are given with values incrementing by one. Or programmer may also assign individual values to the names. Wherever programmer needs those values, instead of the values, he may use the names of the enum.

Ex:

```
enum months{jan=1,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec };
```

```
void main()
{
    clrscr();
    printf(“%d - %d - %d”,16, apr, 2021);
}
```

will display “16-4-2021”.

Ex:

```
enum result{third=35,second=50,first=60,dist=75 };
```

```
void main()
{
    Marks taken input and calculated totals.....
    if(tot<third)
        printf(“fail”);
    else if(tot<second)
        printf(“third class”);
    else if(tot<first)
```

```

        printf("second class");
    else if(tot<dist)
        printf("first class");
    else
        printf("distinction");
}

```

---

Q what are enums?

Q. write about enumerated data types?

Q. what is the use of enumerations?

---

## UNIT-5 FILES

When a program is developed, the code is divided into three parts viz., designing user interface, process and data storage.

The user interface in C is developed with IO functions and the process is managed with operators, branching, loops, functions etc.

The data storage is managed with variables, arrays, pointers, structures, unions etc. but all these are managed in RAM, which is volatile and stores data temporarily. To use data in future, the data can be stored onto hard disk (secondary memory) in the form of a file.

Through a C program, the programmer can create a new file, write data to it, read data from it, append data to existing file etc.

Modes of operation:

Read mode	:	<b>"r"</b>
Write mode	:	<b>"w"</b>
Append mode	:	<b>"a"</b>

If a file is opened in “w” mode, it creates a new file. If opened in “a” mode, if a file exists, then data is appended at end of existing file and if no file exists, the a new file is created.

A file opened in a specific mode can be operated for that purpose only. As the file exists in hard disk and the program runs in RAM, the C provides a facility of using a pointer in program to point to the file by storing its address. The library stdio.h has a predefined typedef structure FILE, to which a pointer is created and used for file manipulations.

Functions related with file processing:

fopen() : this function opens given file in specified mode and returns its address.

Syntax : fopen(“file name and path”, “mode”);

Ex: FILE \*fp;

fp=fopen(“c: / college /data.txt”, “w”); this code creates a file data.txt in college folder for data writing purpose.

fclose() : closes the file whose pointer is given. A file must be closed explicitly once processing is done, unless which the file gets corrupted. when a file is closed with fclose(), the C Runtime places a null value at end of file for identification called EOF (End Of File).

Ex:

fclose(fp);

fseek() : used to place the file pointer to a specific byte in the file

ftell() : returns current byte number of pointer in the file.

feof() : this function is used to identify where the file pointer reached EOF or not.

---

Q. write about file handling

Q. write about reading / writing data from/to files

---

### **Writing data to files:**

The following functions are used to write different data values to a file after opening the file.

Fputc() : used to write single character to a file  
fputw() : used to write one integer to a file  
fputs() : used to read a string from file  
fprintf() : used to write formatted data to a file  
fwrite() : used to write different data type values to a file

Ex:

```
void main()
{
    FILE *fp;
    int no,sal;
    char name[20];
    fp=fopen("data.txt", "w");
    printf("Enter no,name and salary");
    scanf("%d%s%d",&no,name,&sal);
    fprintf(fp, "%d %s %d\n",no,name,sal);
    fclose(fp);
}
```

### **Reading data from files:**

The following functions are used to read data from a file after opening the file.

Fgetc(): used to read single character from file  
fgetw() : used to read one integer value from file  
fgets() : used to write a string to a file  
fscanf() : used to read formatted data from a file  
fread() : used to read different type of data values from a file.

Ex:

```
void main()
{
    FILE *fp;
    int no,sal;
    char name[20];
```

```

fp=fopen("data.txt", "r");
fscanf(fp, "%d%s%d", &no, name, &sal);
while(!feof(fp))
{
    printf( "%d %s %d", no, name, sal);
    fscanf(fp, "%d%s%d", &no, name, &sal);
}
fclose(fp);
}

```

The EOF is End Of File , that can be detected in two ways. Either by checking whether the character read through pointer has EOF or with the function feof().

- ch=fgetc(fp);  
          if( ch==EOF)  
              exit(1);

OR

- if( !feof( fp) )  
          exit(1);

---

Q. Write about how to read data from a file and how to write data?

---

“Error handling” during file processing:

There can be some common errors that raise while doing file processing.  
They are.....

- When trying to read a file beyond indicator.
- When trying to read a file that does not exist.
- When trying to use a file that has not been opened.
- When trying to use a file in an appropriate mode i.e., writing data to a file that has been opened for reading.

- When writing to a file that is write-protected i.e., trying to write to a read-only file.

The function `ferror()` returns true if any of the above error occurs during file processing. If it returns true the programmer must terminate the program with `exit(1)`.

---

Q. Write about Error Handling in files.

---

**This is not in your syllabus. But it is important topic for interviews. So added this here.**

### **Command Line Arguments:**

The arguments passed from command line into the `main()` definition are called command line arguments. These are passed into program, while starting its execution from command prompt. Generally we pass the ‘configurations required for program execution’ as command line arguments into program.

The arguments passed from command line are received into `main()` definition, for further processing. The `main()` can have 2 formal arguments. The first one is an integer that keeps the count of arguments, generally represented with name “`argc`” and the second is char pointers array, that gets stored with argument values and it is generally named “`argv`”.

When command line arguments are passed, the first argument is program name itself.

Ex:

```
void main(int argc, char *argv[])
```



```
{  
    int i;  
    clrscr();  
    for(i=0;i<argc;i++)  
        printf("%s\n", argv[i]);  
}
```

Once this program is compiled, and the executable file is created, the executable file is executed from command prompt by passing arguments along with program name.

Ex:

C:\turboc3\bin\ myprogram Welcome To Command Line Arguments <ENTER>

Will display :

myprogram

Welcome

To

Command

Line

Arguments

\*\* here program name is the first argument.