CS25C01 - Computer Programming : C

Introduction to C : Problem Solving, Problem Analysis Chart, Developing an Algorithm, Flowchart and pseudocode, Program Structure, Compilation & Execution process, Interactive and Script mode, Comments, Indentation, Error messages, Primitive data types, Constants, Variables, Reserved words, Arithmetic, Relational, Logical, Bitwise, Assignment, Conditional operators, Input/output Functions, Built-in Functions.

➤ INTRODUCTION TO C :

* C is a general purpose structured Programming language.

* C was designed by Dennis. M. Ritchie in 1972 AT T. Bell Labs. USA.

* C is originated from a language called 'B' which was an improved version of the language called BCPL (Basic Combined Programming Language) developed by Martin Richards in 1960's.

* C language is a multi-paradigm language. It includes the features of imperative, structural and functional programming paradigms.

* The C Programming language is a High-level Procedure-Oriented Structured Programming language.

⇒ Features of 'C' program :

- → Simplicity and Efficiency
- → Portability
- → Low-level access
- → Modularity
- → Rich library support

C is used for writing

- Operating Systems
- Compilers
- Interpreters
- Assemblers
- Editors
- Spreadsheets
- Database Management Systems
- CAD, COM and Animators.
- Embedded Systems in small devices.
- Game Engine
- IOT (Internet of Things).

* Advantages of C:

→ Structured language. It has few fundamental data types and keywords. This makes the language simple and small.

→ Middle level language, hence suitable for developing application software as well as System software.

→ Portable language that is Programs written in C in one machine can be easily run on some other machine without many modifications.

→ It has wide variety of derived data structures like arrays, pointers, structures and unions apart from fundamental data types like integers, floating point numbers and characters.

→ Provides a rich set of built-in functions.

→ Its Compiler is Compact and Can translate a program into efficient machine instructions.

→ Fast execution of programs when compared to other languages.

→ It has ability to deal efficiently with bits, bytes, words, addresses etc.

* Disadvantages of C :

→ C does have the OOPS feature, so C++ developed.

→ There is no runtime checking in the C Programming language.

→ As the program extends, it is very difficult to fix the bugs.

➤ PROBLEM SOLVING

* What is Problem ?

→ A Problem is a State of difficult that need to be resolved.

→ When a problem exists then that means there is some uncertainity in the solution. While solving a problem there is desire to attain some specific goal.

Here are some examples of various problems faced in our everyday life.

1. Will I get proper transport to go to my workplace ?
2. Should I wear shoes today ?
3. Which mobile phone should I buy ?
4. Which College should I choose for admission ?

If bad decision is made then time and resources may get wasted. Hence it is very important to make proper decisions while solving the problem.

Normally decisions can be made using a problem solving approach.

*Six steps of problem solving

→ Identify the problem
→ Understand the problem

Before solving any problem it is important to understand it.

Three aspects based on which the problem can be understood.

1. Knowledgebase
2. Subject
3. Communication (For understanding the problem, the developer must communicate with the client).

→ Identify the alternative ways to solve the problem.
→ Select the best way to solve the Pblm from list of alternative solutions.
→ List the instruction using the selected solution.
→ Evaluate the solution.

Problem description:

→ Ravi wants to decide which course to opt for his graduation?

Step 1: Identify problem

Pblm: Enrolling a decent college and a branch.

Goal: Name the college and branch he will join this year.

Step 2 : Understand the problem :

Facts :

i) Searching for the colleges and available branches.

ii) Consider his percentage of marks in entrance examinations.

iii) His inclination towards the subjects

iv) Skills he posses.

Step 3 : List possible solutions to the problem :

Possible solutions :

i) Colleges available in home university

ii) Colleges out of his hometown.

iii) Autonomous colleges.

iv) Colleges / university abroad.

Step 4 : Select the best solution to the problem

Criteria based

i) List high ranked colleges with desired branch

ii) Affordable fees

iii) Hostel accommodation / not far away from residential area.

iv) Good placement service.

Step 5 : List instructions :

i) Enroll to admission procedure

ii) Fill up the option form for admission.

iii) Wait for merit list to get displayed.

iv) Get admission to desired course in desired College.

v) Pay the fees.

vi) Attend the college.

Step 6 : Evaluate the solution :

Option 1 (Success) : Path enjoys his choice of College so no need to evaluate.

Option 2 (Reevaluate) :

i) Path feels homesick.

ii) Ravi doesnot find interest in the subject or his grades are low - so he must change the solution.

⇒ Problem Solving with Computers :

Solution :

It is the instruction that must be followed to produce the best results. The result may be :

More efficient, faster or more understandable.

* Results :

It is an outcome or the completed computer assisted answer. It can be in any form such as printout, updated files, output to monitor, speakers and so on.

* Programs :

Computer programs are set of instructions executed to obtain solution to certain problem. These programs are written in some specific programming language.

The field of computer that deals with solving the heuristic problems is called artificial intelligence.

PROBLEM ANALYSIS CHART

A problem Analysis chart (PAC) is a tool used in problem-solving to break down a given problem into smaller, structured parts before writing the program.

* PAC is a graphical/tabular representation that identifies:

1. Input (What data is given?)
2. Process (What operation are performed?)
3. Output (What result is expected?)

It is sometimes called the IPO chart (Input-Process-Output).

* Structure of a problem Analysis chart:

The PAC is used to draw as a simple table with three columns:

Input	Process	Output.
-> Data given to the Program	-> Steps, -> Formulas -> Logics applied	Final result after Processing.

* Importance of PAC in C Programming:

-> Helps in understanding the problem clearly before coding.

-> Reduces chances of logical errors.

→ Makes debugging easier

→ Provides a step-by-step guide for writing algorithms, flowcharts and C programs

→ Useful for all the people to visualize Program flow.

* Input : What the program needs to read from the user or file.

* Processing : What actions or steps the Program needs to perform.

* Output : What the program will show or store as a result

* Formula/ conditions : Any calculations, logic or conditions to apply.

Method 1

⇒ Example 1 : Check whether a number is even or odd ?

Input	Process	output
A number : n	1. Read n 2. If n%2 == 0 → Even Else → odd	Msg : "Even number or odd number

Condition : n%2 == 0 means Even Otherwise odd.

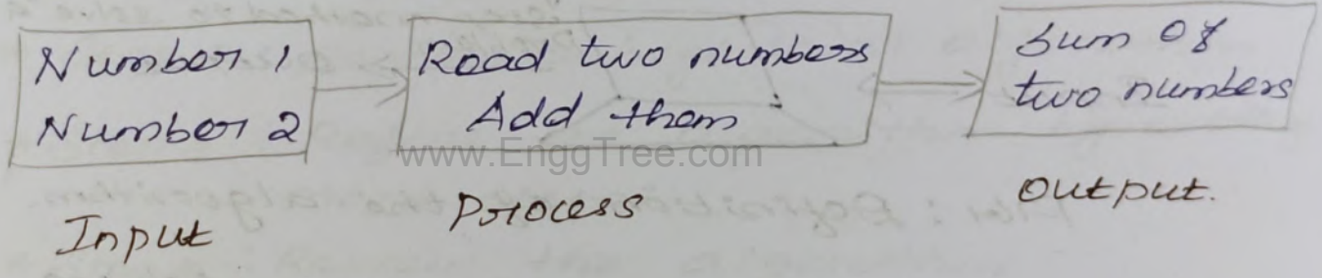
Example 2: Write a IPO chart to calculate the area of a rectangle.

Method 2

Input	Processing	output	Formula/Condition
Length, Width	Multiply length by width ↳ Length * Width.	Area	Area = Length * Width

Method 3

Ex 3: Add two numbers



➤ DEVELOPING AN ALGORITHM

* Algorithm

The sequence of steps to be performed in order to solve problem by the computer is known as an algorithm.

Program = Algorithm + Data

An algorithm is a finite set of instructions for programming (performing) a particular task.

• The instructions are nothing but the statements in the simple english language.

• Another way to describe language (alg) is the sequence of unambiguous instructions. It starts from an initial input of ins that describe a computation that proceeds through a finite no. of well-defined successive steps, producing an output and a ending state.

• Algorithm was first developed by Persian scientist, mathematician - Abdullah bin, al-khwarizmi in 9th century.

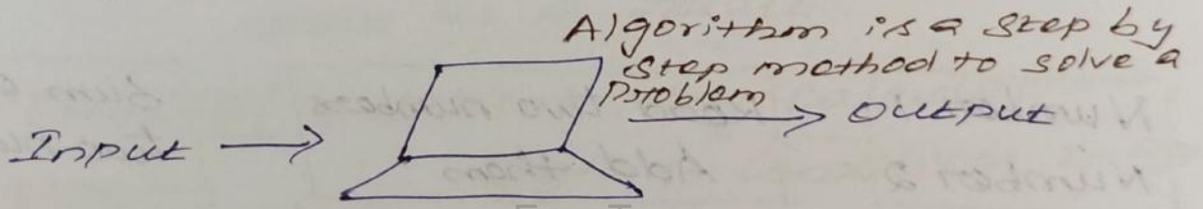
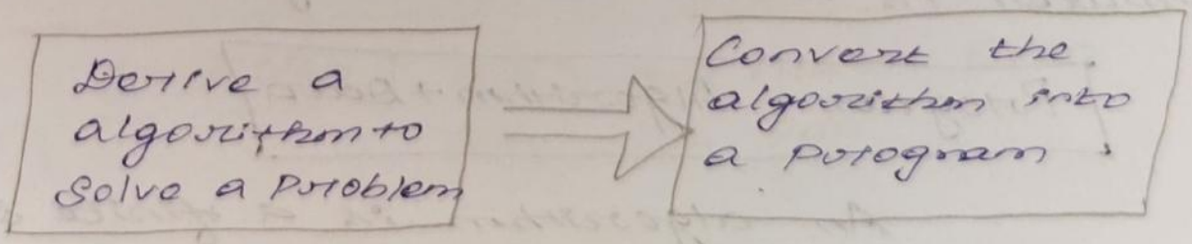


Fig: Definition of the algorithm.

Algorithmic problem solving actually comes in two process:

1. Derivation of an algorithm that solves the problem.
2. Conservation of the algorithm into program code.



Phase 1: Algorithmic Phase

Phase 2: Coding Phase

Fig: Phases of algorithm

* Algorithm : Development Process

An algorithm is a plan for solving Problem. There are many ways to write an algorithm.

Some are very important, some are quite formal and mathematical in nature and some are quite graphical.

The algorithm development Process Consists of five major steps.

* Step 1 : Obtain a description of the Problem.
* Step 2 : Analyze the Problem
* Step 3 : Develop a high level algorithm
* Step 4 : Refine the algorithm by adding more detail
* Step 5 : Review the algorithm

⇒ Properties / Characteristics of Algorithm:

→ Each algorithm is supposed with zero or more inputs.

→ Each algorithm must produce atleast one output.

→ Each algorithm should have definite-ness i.e) each instruction must be clear and unambiguous.

→ It must be possible to perform each instruction.

* Generality : The algorithm must be able to work for a set of inputs rather

than a simple input.

→ Finiteness : The algorithm must always terminate after a finite no. of steps.

Example 1 : Adds the two numbers and store the result in a third variable.

Step 1 : Start
Step 2 : Read the first number in variable a
Step 3 : Read the second number in variable b
Step 4 : Perform the addition of both the numbers and store the result in variable 'c'.
Step 5 : Print the values of 'c' as a result of addition.
Step 6 : Stop.

Example 2 : Find the area of a circle of radius r.

Soln: Inputs to the algorithm :
Radius 'r' of the Circle.
Expected output :
Area of the Circle
Algorithm :
Step 1 : Start
Step 2 : Read input the Radius r of the Circle
Step 3 : Area ← PI * r * r // Calculation of Area
Step 4 : Print Area
Step 5 : Stop

NOTATION

Algorithms can be expressed in many different notations, including Natural language, Pseudo code, flow charts and programming languages.

→ 1. Pseudocode, is something that represents the algorithm through structured human language.

→ 2. Flowchart is something that represents the algorithm graphically.

→ 3. Programming languages are intended for expressing algorithms in a form that can be executed by a computer.

1. Pseudocode :

* Pseudocode is an artificial and informal lang 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. These include while, do, for, if, switch.

⇒ Rules to follow when writing Pseudocode :

1. Only one statement per line.
2. Capitalized initial keyword.
   * Keywords like READ, WRITE etc are in caps.
3. Indent to show hierarchy.
   → In loops, Stales and iterations the logically dependent statements must be indented.
4. End - multiline structures.

→ To improve readability the initial start and end of the several lines must be specified properly.

* Common keywords used in pseudocode:

Commonly used keywords in pseudocode are,

* // : This keyword used to represent a comment.
* BEGIN, END : Begin is the first statement and end is the last statement.
* INPUT, GIET, READ : The keyword is used to inputting data.
* COMPUTE, CALCULATE : Used for calculation of the result of the given expression.
* ADD, SUBTRACT, INITIALIZE : Used for addition, subtraction and initialization.
* OUTPUT, PRINT, DISPLAY : It is used to display the output of the program.
* IF, ELSE, ENDIF : Used to make decision.
* WHILE, ENDWHILE : used for iterative statements
* FOR, ENDFOR : Another iterative incremented / decremented tested automatically.

⇒ Advantages

→ It is easy to translate pseudo code into a programming language.

→ It can be easily modified as Compared to flowchart.

→ It cannot be Compiled nor executed.

→ It doesn't provide visual representation of the programs logic.

=> Disadvantage :

-> There is no standardized style or format.

-> For a beginner, it is more difficult to follow the logic or write pseudocode as compared to flowchart.

* Example : Whether given number is positive or negative.

```
BEGIN
GET n
IF(n==0) THEN
    DISPLAY "n is zero"
ELSE IF(n>0) THEN
    DISPLAY "n is positive"
ELSE
    DISPLAY "n is Negative"
END IF
END IF
END
```


FLOWCHART

* Flowcharts are the graphical representation of the algorithms.

* The algorithms and flowcharts are the final steps in Organizing the solutions.

* Using the algorithms and flowcharts the programmers can find out the bugs in the Programming logic and then can go for coding.

* Flowcharts can show errors in the logic and set of data can be easily tested using flowcharts.

• A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows each shape represents a step in the process and the arrows show the order in which they occur.

* Types of flowcharts :

There are many different types of flowcharts.

→ Document flowcharts - Showing controls over a document - flow through system.

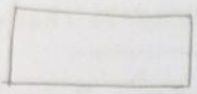
→ Data flowcharts - showing controls over a data-flow in a system.

→ System flowcharts - Showing controls at a physical or resource level.

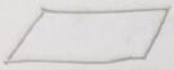
→ Program flowchart - Showing the controls in a program within a system.

* Flowchart Symbols (or) Symbols used in flowchart.

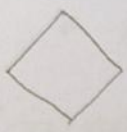
Symbol	Meaning / Explanation
1. ↑ ↓ → ←	* Flowlines are used to indicate the flow of data. The arrow heads are important for flowlines. The flow lines are also used to connect the different blocks in the flowchart
2. (ellipse) Start / Stop / End / Exit	* These are termination symbols. The start of the flowchart is represented by the name of the module in the ellipse and the end of the flowchart is represented by the keywords End or Stop or Exit.

3. 

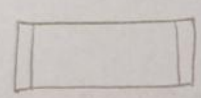
* The rectangle indicates the processing. It includes calculations, opening and closing files and so on.

4. 

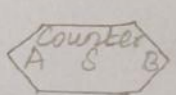
*The parallelogram indicates input and output.

5. 

* The diamond indicates the decision. It has one entrance and two exits. One exit indicates the true and other indicates the false.

6. 

* The process module has only one entrance and one exit.

7. 

* This polygon indicates the loop A indicates the starting of the counter S indicates the stop by which the counter is incremented or decremented.

B indicates the ending value of the counter the no. of times the looping instruction gets executed.

8. 

*The On-page connector connects the two different sections on the same page. A letter is written inside the circle.

9. 

* The Off-page connector connects the two different sections on the different pages.

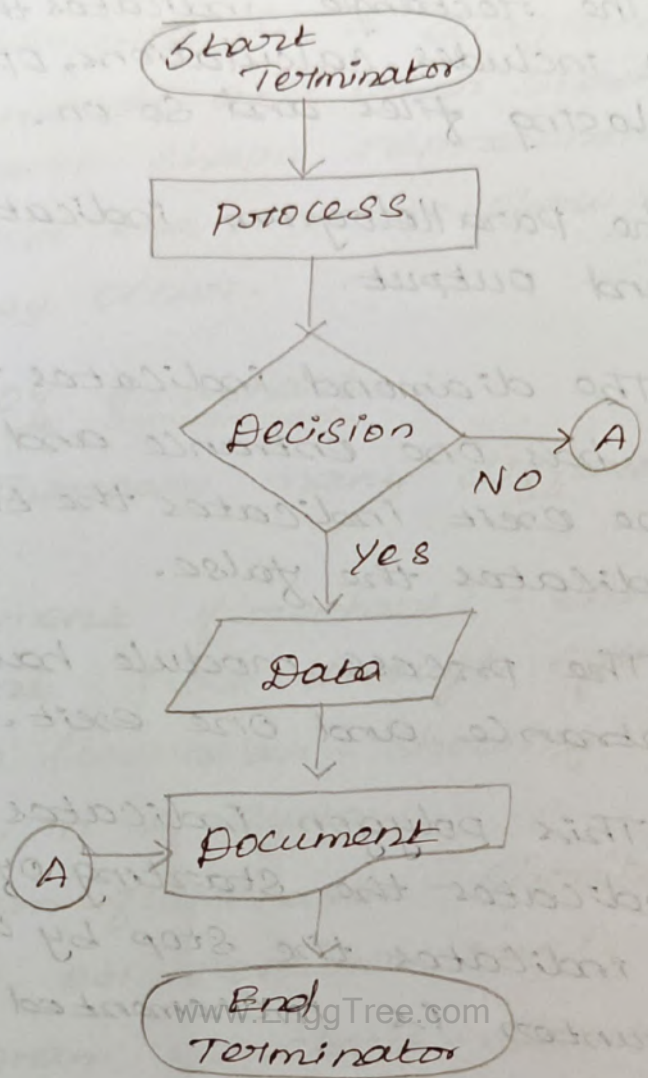
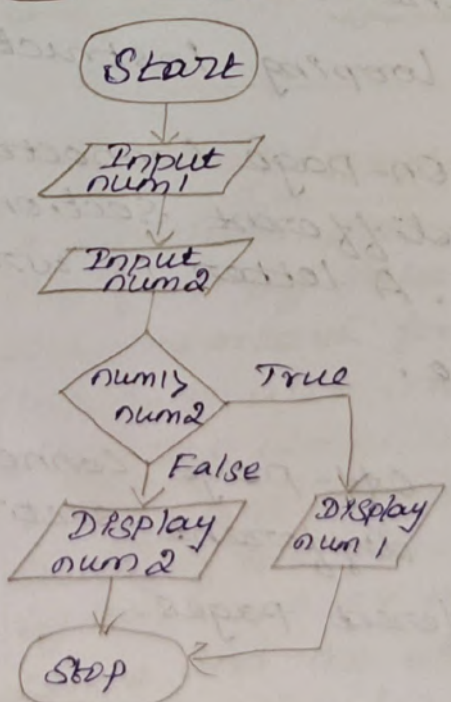


FIG : Flowchart Symbols with names.

Example : Find the largest of two numbers :



=> Advantages of flowcharts :

- Communication
- Effective analysis
- Proper documentation
- Efficient coding
- Proper debugging
- Efficient program maintanance.

=> Limitations of using flowcharts :

- It is complex logic
- Alterations and modifications.

➤ PROGRAM STRUCTURE / STRUCTURE OF C PROGRAM

The basic Structure of C program is divided into 6 parts which makes it easy to read, modify, document and understand in a particular format.

C Program must follow the below-mentioned outline in order to successfully compile and execute.

*Sections of the C program :

There are 6 basic sections responsible for the proper execution of a program.

Sections are mentioned below,

1. Documentation
2. Preprocessor Section
3. Definition
4. Global Declaration
5. Main() Function
6. Sub programs.

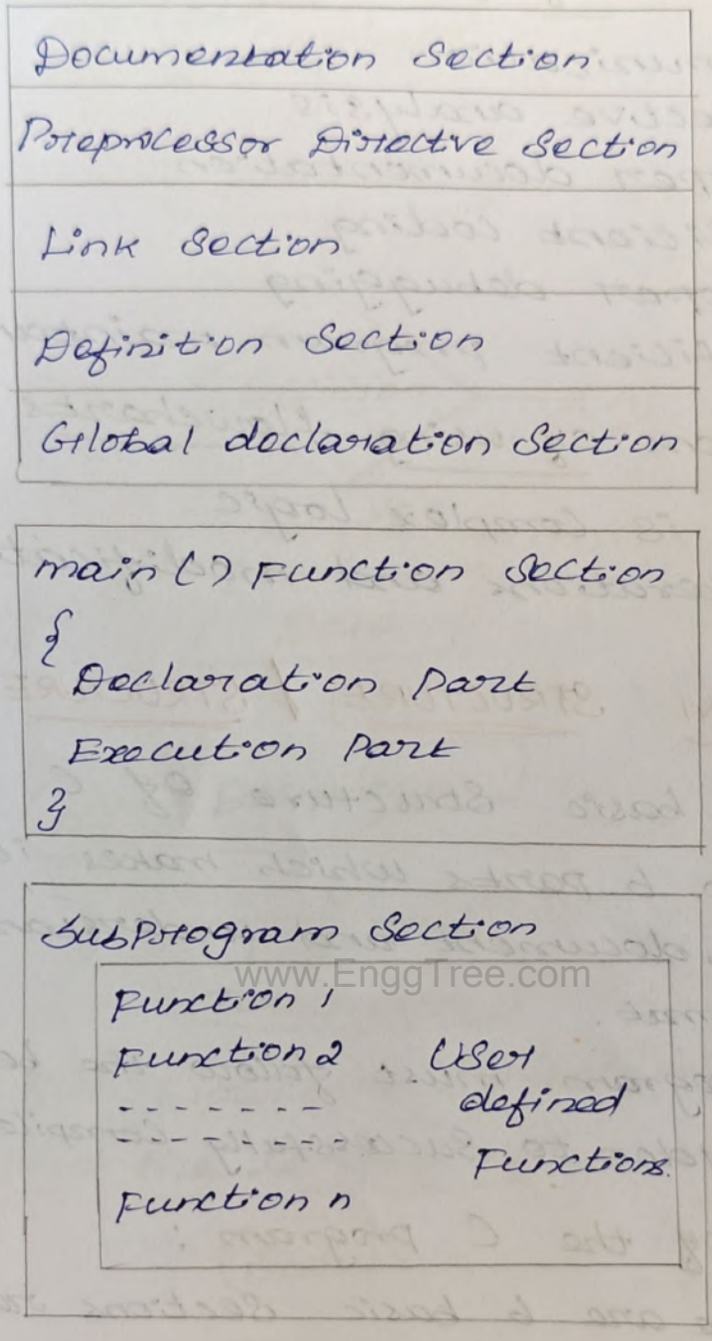


Fig: Structure of C Program.

* Documentation Section:

→ It contains comments that describe the purpose and functionality of the program.

→ The documentation section consists of a set of comment lines giving the name of the Program, the author, programme description and other details that the programmer.

→ Comments can be Single-line (// Comment) or multi-line (/* Comment */).

Ex:

// description, name of the program, Programmer name, date, time etc.

(or)

/* description, name of the program, programmer name, date, time etc.

* Header Files or preprocessor Directives:

These lines are included at the beginning of a C program that instructs the compiler to include Standard or user-defined header files before compilation starts.

- They are identified by the #include directive

→ The Statements Starting with the '#' Symbol are preprocessor directives in a C program.

The program provides many inbuilt preprocessor directives handle by the preprocessor before the Compiler starts execution.

It includes library files needed by the program. This files contain the definitions and prototypes of functions and Constants used in the program.

Ex: Syntax: #define Name value (define format)
↳ #include<Standard header file> Format:

#include <Stdio.h> - Std i/o function

#include <Stdlib.h> - Standard utility functions.

#include <filename.h>
(or)
#include "filename.h"

* Link Section :

The Link Section provides instructions to the Compiler to link functions from the System library.

* Definition Section : / Define directive.

The definition section defines all symbolic Constants. Here we need to define all the constants. using the #define preprocessor directive.

Also, any global variable declarations can be given below,

Syntax :

#define Name value

Ex. #define PI 3.14

There are multiple steps which are involved in the writing and execution of the program.

Preprocessor directives start with the '#' Symbol.

* Global declaration :

The global declaration section Contains global variables, function declaration and static variables.

Variables and functions which are declared in this Scope can be used anywhere in the program.

⇒ Format : Data type followed by variable names; function prototypes.

Ex: int num = 18;

* This section also declares all the user-defined functions.

The statements present in the Global declaration section will be accessible by all the functions in the program.

* **Function :**

A function is a self-contained block of code that performs a specific task. It is not possible to write a C program without a function.

Every function in a C program must start with open curly brace { and ends with closed curly brace }.

➡ **Main() function section :**

The main() function is where the execution of the program begins. Every C program must have one main function section.

This main (section) contains two parts :

→ Declaration part
→ Executable part.

Syntax :

```
main ()
{
  function body
}
```

- **Declaration part :**

The declaration part declares all the variables used in the executable part.

- **Executable part :**

There is atleast one statement in the executable part.

* The return type of the main() function can be int as well as void too.

Void() main tells the Compiler that the Program will not return any value.

The int main() tells the compiler that the Program will return an integer value.

Ex:

Void main() (Or)

int main().

* Sub-Program

It contains all the user-defined function that are called in the main() function.

User-defined functions are generally placed immediately after the main() function, although they may appear in any order.

Except for the main() function section, all sections may be absent when not required.

A function definition includes the function name, return type, Parameters (if any) and the body of the function enclosed in braces { }.

Ex: Add two numbers:

```
#include <stdio.h> // Preprocessor
directive int add(int, int); // Function declaration
int main()
{
 int num1, num2, sum;
```

```
Printf("Enter two numbers:");
Scanf("%d %d", &num1, &num2); // Taking input from the user
Sum = add(num1, num2); // Calling the add function
Printf("Sum = %d\n", sum); // Output the sum
return 0; // End of program
}
```

➤ COMPILATION AND EXECUTION PROCESS :

This process contain detailed, step-by-step instructions for entering, Compiling and running C Programs, in a no. of environments.

There are three basic steps to preparing and running a program.

1. Editing (Create or modify the source program files)
2. Compiling and linking the program
   ↳ (Reads the source file, Compile them and then turns them into an executable program)
3. Run.
   ↓
   Executes the program

* Editing, Compiling and Running C Program : (MS-DOS)

If we are using an older MS-Dos Compiler, it may be invoked from the Command Line, in which it will act somewhat similarly to the UNIX Compilers.

Several MS-Dos Compilers which "roll their own" graphical user interfaces, and these act as GUI Compilation Environment.

Step 1 : Entering or Editing the Source code :

Most MS-DOS compilers → a kind of source code editor for typing in our own programs, but if not, we have to use an ordinary text editor.

In any case, whichever editor we use, make sure to type in the program exactly as shown, including all punctuation, and with the lines laid out just as they are in the code.

When finished typing to save the file in .c (filename.c).

Step 2 : Compiling the program :

Microsoft Compilers (Ex : MSC V6.00) is a Command-line Syntax are,

C:\> cl hello.c

C:\> bcc hello.c

C:\> tc hello.c.

It is also possible in two-step Compilation requesting the Compiler to build an object (.OBJ) corresponding to .C file

When we attempt to compile HELLO.C, the bcc Command creates HELLO.OBJ but not HELLO.EXE. It means to invoke the linker ourself.

C:\> link hello ;

(or)

C:\> tlink hello ;

```
Printf("Enter two numbers:");
Scanf("%d%d", &num1, &num2); // Taking input from the user
Sum = add(num1, num2); // Calling the add function
Printf("Sum = %d\n", sum); // output the sum
return 0; // End of program
}
```

➤ COMPILATION AND EXECUTION PROCESS :

This process contain detailed, step-by-step instructions for entering, Compiling and running C Programs, in a no. of environments.

There are three basic steps to preparing and running a program.

1. Editing (Create or modify the source program files)
2. Compiling and Linking the program
   ↳ (Reads the source file, Compile them and then turns them into an executable program)
3. Run.
   ↓
   Executes the program

✱ Editing, Compiling and Running C Program : (MS-DOS)

If we are using an older MS-DOS compiler, it may be invoked from the Command Line, in which it will act somewhat similarly to the UNIX Compilers.

Several MS-DOS Compilers which "roll their own" graphical user interfaces, and these act as GUI Compilation Environment.

Step 1 : Entering or Editing the Source Code :

Most MS-DOS compilers → a kind of source code editor for typing in our own programs, but if not, we have to use an ordinary text editor.

In any case, whichever editor we use, make sure to type in the program exactly as shown, including all punctuation, and with the lines laid out just as they are in the code.

When finished typing to save the file in .c (filename.c).

Step 2 : Compiling the program :

Microsoft Compilers (Ex : MSC V6.00) is a Command-line Syntax are,

```
C:\> cl hello.c
C:\> bcc hello.c
C:\> tc hello.c.
```

It is also possible in two-step Compilation requesting the Compiler to build an object (.OBJ) corresponding to .c file

When we attempt to compile HELLO.C, the bcc Command creates HELLO.OBJ but not HELLO.EXE. It means to invoke the linker ourself.

```
C:\> link hello ;
(or)
C:\> tlink hello ;
```

Step 3: Running the Program:

Once created the executable file now attempt to run it by typing its name:

C:\>hello.

If the program prints output to the screen and if we want to save it, then type

C:\>hello>hello.out.

* Editing, Compiling and Running C Program using Dev C++ / Turbo C++:

It is important to note that C++ is the superset of C and has the same syntax.

Therefore a C program can be compiled using C++ compiler. In the windows OS we can use DEV C++ (or) TURBO C++ compiler.

Step 1: Entering or Editing the Source Code:

To enter a new program in Turbo C++ editor / Dev C++ simply type the program into the editing area on a line-by-line basis and press the enter key at the end of each line.

To edit the program in the line, to use the mouse or the cursor movement (arrow) keys to locate the beginning of the edit area.

Then use backspace or delete keys to remove unwanted characters.

The Cut and paste selections in the Edit

menu allows movement of block of lines from one location to another.

Once the program has been entered, it should be saved before it is executed. In TURBO C++/DEVC++ this is accomplished by selecting Save AS from the file menu, and supplying a program name, such as PROGRAM.C.

Once the program has been saved, it can again be saved after recent changes at some later time simply by selecting Save from the File menu.

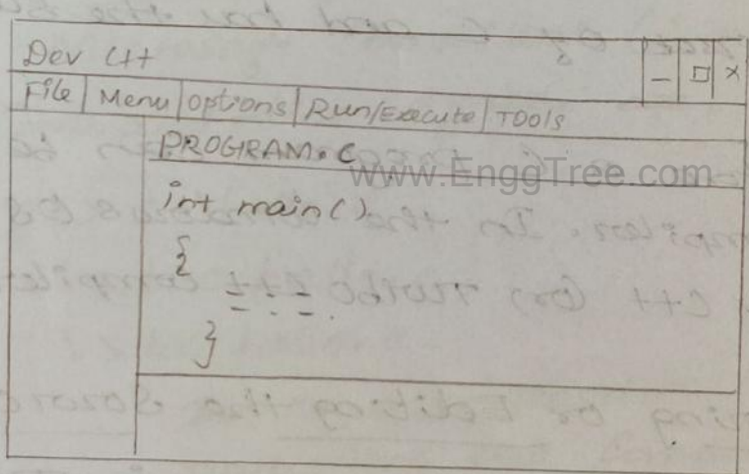


FIG: DEV C++ Editor with Edit menu

Step 2 : Compiling the program :

It cannot directly execute the source file. Therefore, Once the program has been entered into the computer, edited and saved, it can be compiled by selecting Compile from the Compile menu.

An interpreter menu (window in the bottom) gives the information about warnings and errors in the program if any.

*If the program doesnot compile successfully, a list of error messages will appear in a interpreter window.

We can also compile the program file by using short-cut keys Alt+F9.

* Step 3: Linking the program:

In this case of C the standard input and output functions are contained in a library (Stdio.h). So even the most basic program will require a library function.

The object file can be linked by selecting Link from the Compile menu. A successful linking Produces an executable file with the extension .exe, the .exe stands for Executable.

It is important to note that Build All from the Compile menu does the compilation of the Program and if Compilation is error free, it does linking of the program immediately after the Compilation.

* Step 4: Running the Program:

The text editor produces .C source files, which go to the Compiler, which Produces .obj object files, which go to the linker, which produces .exe executable file.

INTERACTIVE AND SCRIPT MODE

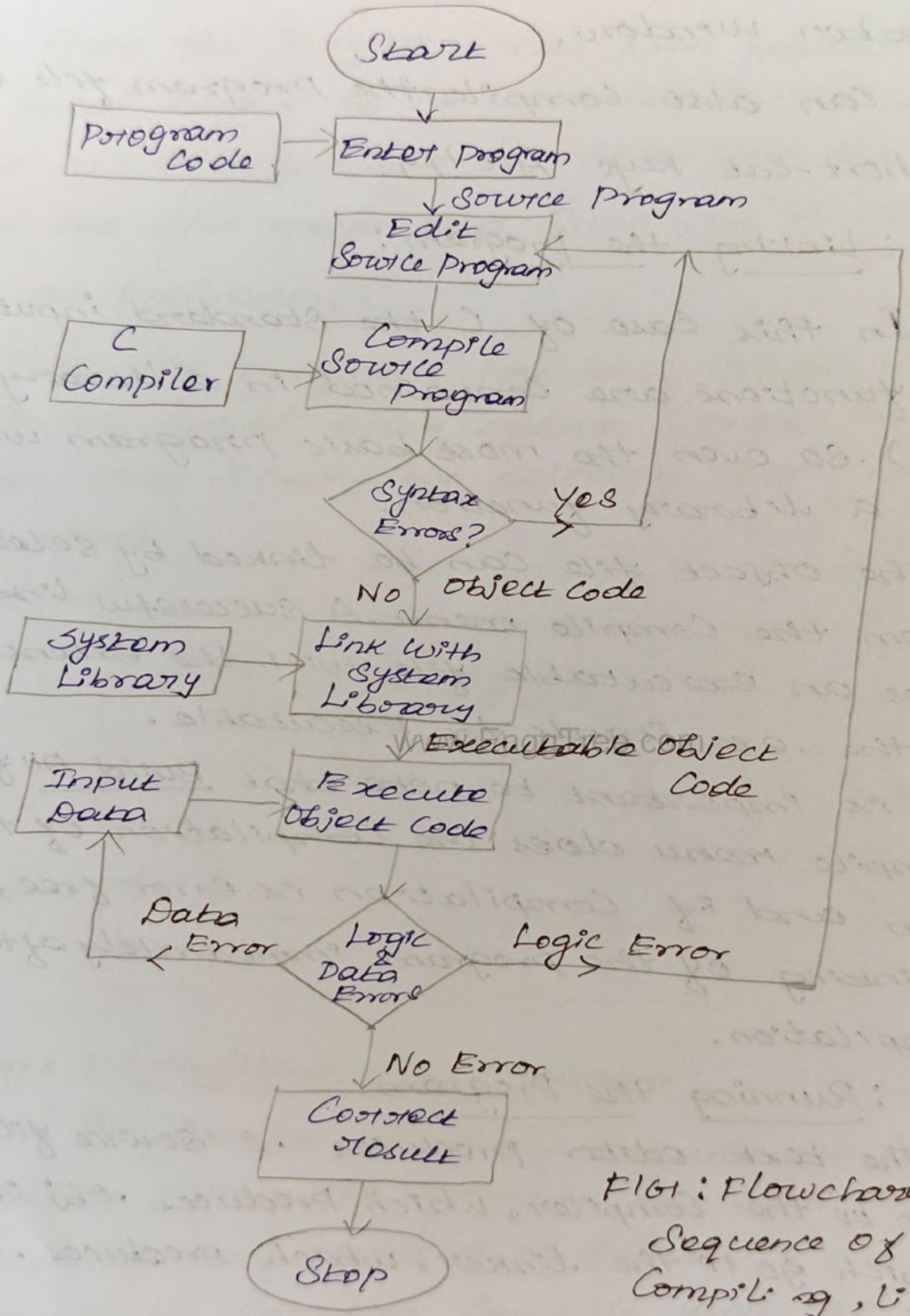


FIG: Flowchart showing Sequence of editing, Compiling, Linking and Executing C Program.

⇒ Interactive mode :

It is used for testing small pieces of code quickly.

Write a few lines (like a printf or simple calculation) compile and run immediately.

Good for checking logic step by step.

This is like typing commands and getting results immediately.

In C, we don't usually have a built-in interactive shell like Python.

But in Dev C++, can simulate interaction mode by:

→ Writing small snippets of code (like printing a message, performing a calculation).

→ Compiling and running immediately.

→ Getting instant output in the console window.

Ex :

```
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
    printf("5+3=%d\n", 5+3);
    return 0;
}
```

Each time user change something, compile (F9) and run (Ctrl+F10), the user interactively test it.

So interaction mode in Dev-C++ = Writing & executing short test code repeatedly to check outputs immediately.

SCRIPT MODE :

This is when the user write an entire C/C++ program as a script (a file with multiple lines of code), save it, then compile and run it as a whole.

In Dev-C++, this means :

→ Create a new project or source file.

→ Write the complete program with #include, main() function, variables, loops etc.

→ Compile once → Run the whole Program.

→ It is good for final execution with input, logic and output.

> Interaction Mode = Quick testing of small code
> Script Mode = Writing and running full program

Interaction mode	Script Mode.
Usage - Quick testing of code	Full Program development
Execution - Run small snippets	Run Complete saved file
Compilation - Frequent, after small edits	Once after full code
Ex: Printf("Hi");	Full Program with input, logic, output.

COMMENTS in C programming :

Comments in C are non-executable statements that are added to programs to explain the code. They are ignored by the compiler and do not affect Program execution.

They are mainly used for :

- → Improving code readability
- → Documentation
- → Debugging and testing.

* Types of Comments in C :

C Programming supports two types of Comments,

1) Multi-line Comment (/* .... */)
2) Single-line Comment (//)

1) Single-line Comment (//)

- → This Comment introduced in C99 Standard.
- → Used for short explanations on one line.
- → Everything after // on the same line is ignored by the Compiler.

Ex :

```
#include<stdio.h>
int main(){
    int age = 20; // declaring and initializing variable
    Printf("Age = %d", age); // Printing the variable
    return 0;
}
```

2) Multi-line Comment (/*.... */)

→ Used to write longer descriptions that span multiple lines.

→ Everything between /* and */ is ignored by the compiler.

Ex:

```
#include <Stdio.h>
int main() {
    /* This is a multi-line comment.
       It can span across multiple lines.
       Useful for large explanations or code
                                documentation. */
    Printf("Hello, World!");
    return 0;
}
```


* Purpose of using Comments

1. Code Readability

It makes programs easier to understand for other programmers or for yourself when revisiting the code later.

2. Documentation:

It provides important details such as the purpose of functions, variables or logic.

3. Debugging:

User can "Comment out" parts of code to test/debug without deleting them.

Ex:

```
#include <stdio.h>
int main() {
    int a=5, b=10, Sum;
    // Sum = a+b; // temporarily commented for testing
    Printf("Testing code...");
    return 0;
}
```

* Rules & Best practices for Comments

1. Keep Comments short and clear
2. Donot overuse comments - write self-explanatory code whenever possible
3. Update Comments if the code changes (to avoid confusion).
4. Avoid using Comments to explain obvious statements.

Ex: Bad Example

```
int x=10; // assign 10 to x.
```

Good Example

```
Int maxStudents=50; // maximum number of
                       Students in a class.
```

* C doesnot support nested Comments.
* If you try to put /*.... /*.... */.... */, it will cause a Compilation error.

Ex:

```
/* outer Comment
*/ /* Inner Comment */ } Invalid Comment.
```

INDENTATION :

Indentation in C Programming refers to the systematic spacing of code lines (using spaces or tabs) to make the structure of the program clear and more readable.

The C compiler ignores indentation (extra spaces and line breaks), but programmers must use indentation to make code understandable for humans.

* What is Indentation ?

Indentation means placing spaces or tabs at the begining of a line of code.

It is used to show hierarchy and structure in a program.

It helps to visually separate blocks like loops, conditionals and functions.

• Proper Indentation

```
#include <stdio.h>
int main() {
    int num = 10;
    if (num > 0) {
        Printf("Number is
                positive");
    }
    else {
        Printf("Number is not
                positive");
    }
    return 0;
}
```

• Without Indentation (Hard to read)

```
#include <stdio.h>
int main() {
int num = 10;
if (num > 0) {
Printf ("Number is positive");
} else {
Printf ("Number is not positive");
}
return 0;
}
```

* Common mistakes in indentation:

1. Mixing spaces and tabs (leads to misaligned code).
2. Writing all code in one line (hard to debug)

To ~~avoid~~ (use practices below):

* Use 4 spaces (or a tab) per indentation level.)
* Use one indentation style throughout the program.

ERROR MESSAGES

Errors in C are mistakes in a Program that prevent it from Compiling, linking or running correctly.

When errors occur, the compiler or runtime system generates error messages to help programmers identify and fix them.

Errors can occur at different stages of Program development.

Types of Errors can occur three stages.

* Types of Errors in C:

There are three types of errors,

1. Compilation time
2. Linking time
3. Runtime

1. Compile-time Errors : by the Compiler

* These errors are detected before the Program runs (execution), during Compilation.

* It includes Syntax errors and Semantic errors.

* Program will not run until these are fixed.

Error type	Example	Explanation
Syntax Error	int main({	Missing or extra characters like) or {
Undeclared Variable	x=10 ;	Using variable without declaring it
Missing Semicolon	Printf("Hi")	Forgetting ; at the end of a statement
Wrong Data Type	int x="hello";	Assigning String to an int

Ex :

```
#include <Stdio.h>
int main () {
    int x
    Printf("%d",x) ;
    return 0 ;
}
```

Error Message :

error : expected ';' before 'printf'

2. Linker Errors / Link-time Errors

- Occur when the compiler cannot link functions or variables properly.
- Usually caused by missing function definitions or incorrect linking of files.
- These errors occur during the linking stage, when the compiler tries to connect your code with libraries.

Ex:

```
#include <stdio.h>
int main() {
    hello(); // function not defined
    return 0;
}
```

Error message:

Undefined reference to 'hello'

3. Run-time Errors:

- Detected when the program is running.
- Causes abnormal termination or incorrect results.

Ex: division by zero → `int x=10/0;`
invalid memory access, file not found. → Uninitialized pointer.

- These happens after the program is compiled when its running.

Ex:

```
#include<stdio.h>
int main() {
    int a=5, b=0;
    Printf("%d", a/b); // division by zero
    return 0;
}
```

- Error message:
  Floating point exception (core dumped).

4. Logical Errors:
   → Program compiles and runs but produces wrong output.
   → Hard to detect because no error message.

Ex:

```
#include<stdio.h>
int main(){
    int a=5, b=10;
    int sum = a-b; //should be a+b
    printf("sum = %d", sum);
    return 0;
}
```

Output: (Error)

Sum = -5

* Common Error messages in C:

1. Syntax Errors:
   → Caused by wrong grammar or missing symbols.
   → Ex: Missing ;, mismatched braces { }, undeclared variables.

Ex:

```
Int x=10
```

Error Message:

error: expected ';' before 'return'

2. Undeclared Identifier:
   When a variable or function is used without declaration.

Ex:

```
Printf("%d", y);
```

Error message:

error: 'y' undeclared (first use in this function)

3. Type mismatch Errors:

Occur when wrong data types are used.

Ex:

```
int x;
x = "hello";
```

Error message:

Warning: assignment makes integer from pointer without a cast.

4. Missing Return Statement:

Occurs in non-void functions without a return.

Ex:

```
int add(int a, int b){
    a+b; //no return
}
```

Error message:

Warning: Control reaches end of non-void fn.

5. Segmentation Fault:

A run-time error when a program tries to access invalid memory.

Ex:

```
int *p = NULL;
*p = 10; //accessing null pointer
```

Error message:

Segmentation fault (core dumped)

6. Linker Error: Undefined Reference

When function is declared but not defined.

Ex:

```
int square(int x);
int main(){
    square(5);
    return 0;
}
```

Error:

Undefined reference to 'square'

PRIMITIVE DATA TYPES :

A data type defines a set of values and the operations that can be performed on them. Every data item (constant, variable, etc.,) in a C Program has a data type associated with it.

C Supports several different types of data, each of which may be represented differently within the computer's memory.

All C Compilers supports four fundamental data types, namely integer (int), character (char), floating point (float), and double-precision floating point (double).

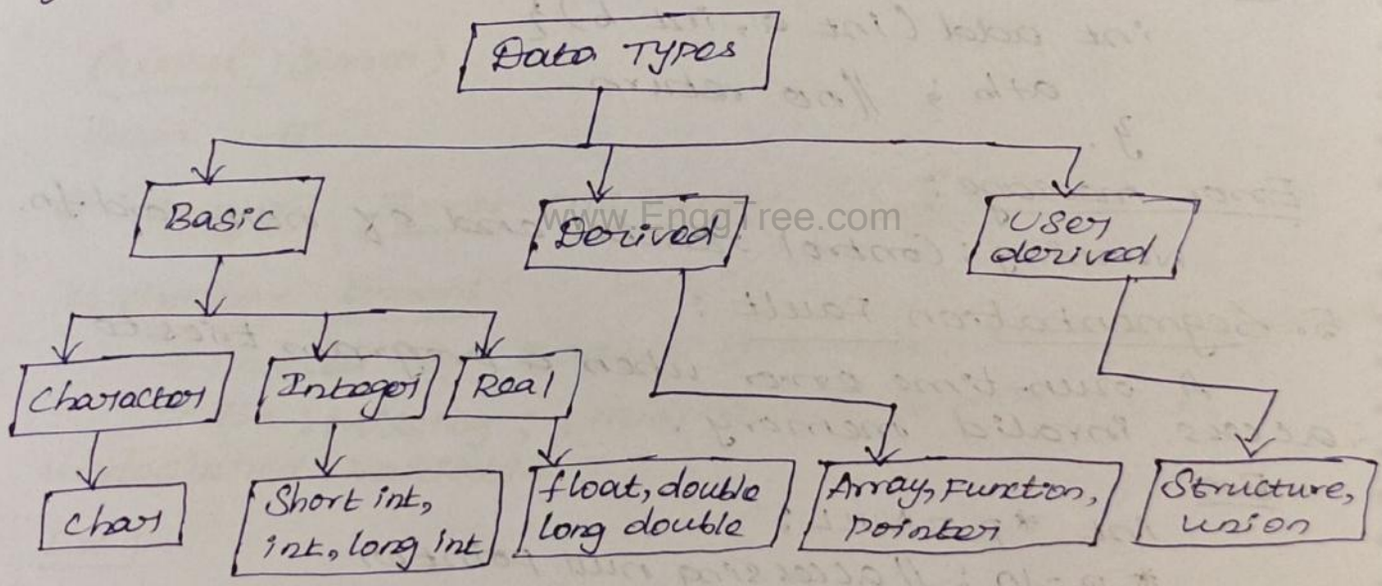


FIG : Data types supported by C.

The RANGE of the basic four types are shown below table.

Data Type	Range of Values.
char	-128 to 127
int	-32,768 to 32,767
float	3.4 e-38 to 3.4 e+38
double	1.7 e-308 to 1.7 e+308

* Integer Data Types

Integers are whole numbers with a range of values supported by a particular machine.

Generally, integers occupy one word of storage, and since the word sizes of machines vary, the size of an integer that can be stored depends on the computer.

Data Type	Description	Required Memory	Min Value	Max Value
int	Represent integers: Signed	2 Bytes	-32,768	32767
	unsigned		0	65,535
int	Represent integers: Signed	4 Bytes	-2147,483,648	2,147,483,647
	Unsigned		0	4,294,967,295
Short int	Represent integers: Signed	2 Bytes	-32,768	32767
	Unsigned		0	65535
long int	Represent integers: Signed	4 Bytes *	-2,147,483,648	2,147,483,647
	Unsigned		0	4,294,967,295

* Floating-Point Types:

Floating point (or) Real numbers are stored in 32 bits, with 6 digits of precision.

Floating point numbers are defined in C by the keyword float. When the accuracy provided by a float point (numbers) is not sufficient, the type double data type number used 64 bits giving a precision of 14 digits.

These are known as double precision numbers.

Data Type	Required Memory	Min value	Max value
float	4 Bytes	3.4e-38	3.4e+38
double	8 Bytes	1.7e-308	1.7e+308
Long double	10 Bytes	3.4e-4932	1.1e+4932

* Character Type :

A Single character can be defined as a character (char) data type. Characters are usually stored in 8-bits of internal storage.

The qualifier Signed (or) unsigned may be explicitly applied to char.

While unsigned characters have values between 0 to 255, Signed characters have values from -128 to 127.

Data Type	Description	Required memory	Min Value	Max Value
Char	Signed Single character	1 Bytes	-128	127
Char	Unsigned Single Character	1 Bytes	0	255

CONSTANTS

Constants Provide a way to define a variable which Cannot be modified during the execution of the Program.

Constants can be defined by placing the keyword Const in front of any variable declaration.

* Integer Constants : 0, 37, 2001
* floating Constants : 0.8, 199.33, 1.0
* Character Constants : 'a', '5', '+'
* String Constants : "a", "Monday"

⇒ Integer Constant :

An Integer Constant is an integer valued number. It can represent decimal, octal or hexadecimal values.

-> Decimal Constant :

A decimal Constant contains any of the digits 0 through 9, preceded by an optional - or + sign. The first digit cannot be 0.

Integer Constants begining with the digit 0 are interpreted as an octal Constant; rather than a decimal Constant.

In decimal Constants embedded spaces, Commas, and non-digital characters are not permitted between digits.

Ex : 0. -9, 22 etc.

-> Hexadecimal Constants :

A hexadecimal Constant begins with the 0 digit followed by either an x or X, followed by any Combination of the digits 0 through 9 and the letters a through f or A through F.

The letters A (or a) through F (or f) represent the values 10 through 15, respectively.

Hexadecimal digits : 0 1 2 3 4 5 6 7 8 9 A B C D E F

Ex:
0x7f, 0x2a, 0x521 etc.

→ Octal Constant :

Octal digits : 0 1 2 3 4 5 6 7

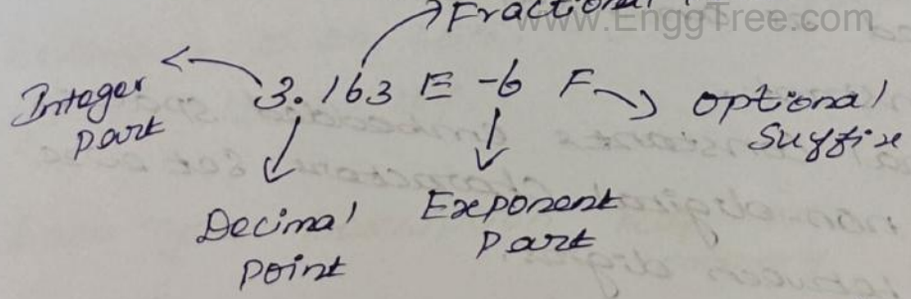
Ex : 021, 077, 033 etc.

⇒ Floating-Point Constants

These are numbers with decimal parts.

A floating point constant consists of :

1) An integral part
2) A decimal point
3) A fractional part
4) An exponent part
5) An optional suffix



→ A suffix of f or F indicates a type of float, and a suffix of l or L indicates a type of long double. If a suffix is not specified, the floating-point constant has a type double.

→ A plus (+) or minus (-) symbol can precede a floating-point constant. However, it is not part of the constant; it is interpreted as a unary operator.

→ The limits for floating-point value are set in the float.h include file.

=> Character Constants

A character constant is a single character, enclosed within the pair of single quotation marks (apostrophes).

Ex:

'A' 'z' ';' '6' '?' '2'

* A character constant is enclosed in single quotes.

* Any number enclosed in single quotes is a character constant, however it is not same as number.

For Ex, character constant '6' is not same as number 6.

* The last character constant in the above examples is blank space, which is typed by pressing the apostrophe key, the space bar and the key.

The range of character constant is -128 to the

=> String Constants

A String constant or literal contains a sequence of zero or more characters or escape sequences enclosed in double quotation mark.

Ex; "" /* the empty string */

"a" /* String with one character */

"Hello Students" /* String with more than one character */

* A String constant is an array of characters

VARIABLES

A variable is a data name, used to store a data value that value may change during the execution of a program.

A variable in C is a named piece of memory which is used to store data and access it whenever required. It allows us to use the memory without having to memorize the exact memory address.

* Rules for variable declaration in C:

→ A variable name must begin with a letter or underscore.

→ Variables are case-sensitive

→ They can be constructed with digits and letters.

→ No special symbols are allowed other than underscores.

→ Sum, height - value are some examples of the variable name.

→ Cannot be a reserved word.

* Declaring a C variable:

Variables should be declared in the C Program before use.

Memory space is not allocated for a variable during declaration. It happens only on the variable definitions.

Declaration tells the Compiler about the data type and size of the variable. The variable can be declared many times in a program.

The variable declaration specifies the name and type of the variable.

Syntax:

```
data_type variable_name ;
```

Ex:

```
int x, y, z ;
```

* Defining a variable in C:

Declare a variable in C#, C it stores some garbage value by default. Here, garbage value means an unexpected value (it might be zero also) so after the declaration, when we assign some value to the variable.

It is called variable initialization.

Syntax:

```
data-type variable_name ;
Variable_name =value ;
```


Ex:

```
int x, y ;
x=50, y=30 // Here, x is defined with a
              value of 50, and y is defined
              with a value of 30.
```

* Variable initialization in C:

Defining a variable with a value at the time of its declaration is known as variable initialization in C.

Syntax:

```
data-type variable_name = value ;
```

Ex:

```
int X = 0 ;
```

RESERVED WORDS (KEYWORDS) IN C :

In C there are certain reserve characters, called Keywords. They have standard, predefined meanings.

Keywords can be used only for their intended purpose, they cannot be used as programmer defined identifier.

Cannot use keywords as identifiers, if we do, the compiler reports an error.

Reserved words (keywords) are predefined words that have special meaning in C and Cannot be used as variable names.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

FIG : ANSI C keywords

OPERATORS IN C :

C includes a large number of operator which fall into different categories.

Various operators supported by C.

These are,

- → Arithmetic operators
- → Relational operators
- → Logical operators
- → Assignment operator
- → Unary operators
- → Conditional operators
- → Bit-wise operators.

ARITHMETIC OPERATORS

To solve most programming problems need to perform arithmetic operations by writing arithmetic expressions. Each operator manipulates two operands, which may be constants, variables or other arithmetic expression.

The arithmetic operators +, -, *, and / may be used with int or double data type operands. On the other hand, the remainder operator also known as modulus operator can be used with integer operands to find the remainder of the division.

When both operands in an arithmetic expression are integers, the expression is called an integer expression, and the operation is called integer arithmetic.

When both operands in an arithmetic expression are floating point numbers, the expression is called floating point expression or real expression

and the operation is called floating point arithmetic or real arithmetic.

Arithmetic Operators			
Arithmetic operator	Meaning	Declarations: int a=5, b=16 double c=3.0, d=7.5	
		Integer Examples	Floating point Examples.
+	addition	a+b is 21	c+d is 10.5
-	Subtraction	a-b is -11 b-a is 11	c-d is -4.5 d-c is 4.5
*	Multiplication	b*a is 80	c*d is 22.5
/	division	a/b is 0 b/a is 3	c/d is 0.4 d/c is 2.5
%	Remainder (Modulo division)	a%b is 5 b%a is 1	

* The remainder operator (%) requires that both operands be integers and the second operand be non zero.

* The division operator requires the second operand by nonzero.

* The result of integer division results in a truncated quotient. (i.e) the decimal portion of the quotient will be dropped).

If a division operation is carried out with two floating-point numbers or with one floating point number and one integer, the result will be a floating point quotient.

Ex: 7.5/5 = 1.5

If one or both operands are negative then the addition, Subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra.

The result of the remainder operation always gets the sign of first operand. For Ex,

If a = 14 and b = 5 then a % b = 4

If a = -14 and b = 5 then a % b = -4

If a = -14 and b = -5 then a % b = -4

If a = 14 and b = -5 then a % b = 4

Operand 1	Operand 2	Result
short	int	int
int	float	float
double	float	double
double	int	double
int	long	long

Ex:

```
#include <stdio.h>
main()
{
  /* Local Definitions */
  int a=14;
  int b=5;
/* statements */
  Printf("%d + %d = %d\n", a, b, a+b);
  Printf("%d - %d = %d\n", a, b, a-b);
  Printf("%d * %d = %d\n", a, b, a*b);
  Printf("%d / %d = %d\n", a, b, a/b);
  Printf("%d %% %d = %d\n", a, b, a%b);
}
```

Output :

14 + 5 = 19
14 - 5 = 9
14 * 5 = 70
14 / 5 = 2
14 % 5 = 4

RELATIONAL OPERATORS

Relational operators are used in C language to compare values, typically in a conditional control statements.

The four relational operators in C are <, <=, >, >=. There are two equality operators == and !=.

They are closely associated with the relational operators.

Relational Operators	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
Equality operator	**Meaning**
==	equal to
!=	not equal to

The double equal to sign == used to compare equality is different from the single equal to '=' sign which is used as an assignment operator.

These SIx operators are used to form logical expressions, which represent conditions that are either true or false. The result of the expression is of type Integer, since true is represented by the Integer value 1 and false represented by the value 0.

Examples of relational operators		
Declarations int i=3, j=4 float f=5.5, g=4.5		
Expression	Interpretation	Value
i<4	True	1
(i+j)<=10	True	1
f>g	True	1
f<=g	False	0
j*i>=22	False	0
f-g==1.0	True	1
j/i!=2	False	0

Among the relational and equality operators each operator is a Complement of another operator in group.

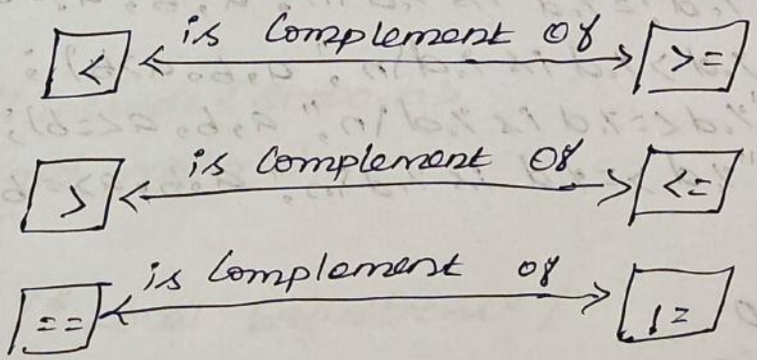


Fig: Operator Complements

Using Complement operators can simplify the expressions as,

Original expression	Simplified expression
!(a<b)	a>=b
!(a>b)	a<=b
!(a!=b)	a==b
!(a<=b)	a>b
!(a>=b)	a<b
!(a==b)	a!=b

Fig: Simplification of operators using their Complements

Ex:

```
#include <stdio.h>
Void main()
{
  /* Local Definitions */
   int a=8;
   int b=-5;
   /* Statements */
   Printf("%d<%d is %d\n", a,b, a<b);
   Printf("%d==%d is %d\n", a,b, a==b);
   Printf("%d!=%d is %d\n", a,b, a!=b);
   Printf("%d>%d is %d\n", a,b, a>b);
   Printf("%d<=%d is %d\n", a,b, a<=b);
   Printf("%d>=%d is %d\n", a,b, a>=b);
}
```

Output:

```
8 < -5 is 0
8 == -5 is 0
8 != -5 is 1
8 > -5 is 1
8 <= -5 is 0
8 >= -5 is 1
```

LOGICAL OPERATORS

In addition to arithmetic and relational operators, C has three logical operators for combining logical values and creating new logical values.

Logical operator	Meaning
!	NOT
&&	Logical AND
||	Logical OR

NOT:

The not operator (!) is a unary operator. It changes a true value (1) to false (zero) and a false value (0) to true (1).

AND:

The AND operator (&&) is a binary operator. Its result is true only when both operands are true, otherwise it is false.

OR:

The OR operator (||) is a binary operator. Its result is false only when both the operands are false, otherwise it is true.

Ex:

```
#include<stdio.h>
void main()
{
 /* Local Definitions */
 int a=8;
 int b=-5;
 int c=0;
```

```
/* Statements */
Printf("%d && %d  is %d\n", a, b, a && b);
Printf("%d && %d  is %d\n", a, c, a && c);
Printf("%d && %d  is %d\n", c, a, c && a);
Printf("%d || %d  is %d\n", a, c, a || c);
Printf("%d || %d  is %d\n", c, a, c || a);
Printf("%d || %d  is %d\n", c, c, c || c);
Printf("!%d && !%d  is %d\n", a, c, !a && !c);
Printf("!%d && %d  is %d\n", a, c, !a && c);
Printf("%d && !%d  is %d\n", a, c, a && !c);
}
```

Output:

```
8 && -5  is 1
8 && 0   is 0
0 && 8   is 0
8 || 0   is 1
0 || 8   is 1
0 || 0   is 0
!8 && !0  is 0
!8 && 0   is 0
8 && !0   is 1
```


Examples of logical operators

Declarations: int i=3, j=7
double f=5.5, g=4.5
char ch='T'

Expression	Interpretation	value
(i<=5) && (ch=='T')	True	1
(j<8) || (ch=='L')	True	1
(f+g)==10.0 || i<2	True	1
f>=6 || (i*j)<15	False	0
!(f>5.0)	False	0

BITWISE OPERATORS

The bitwise operators are the bit manipulation operators. They can manipulate individual bits within the piece of data. These operators can operate on integers and characters but not on floating point numbers or numbers having double data type.

Operator	Description	Example
~	The bitwise complement operator is an unary operator. It complements each bit of the operand	operand = 1111 0000 1111 0000 ~operand Result = 0000 1111 0000 1111
&	It compares each bit of its first operand to the corresponding bits of its second operand. If both bits are 1 the result set 0. Otherwise the result bit set to 0.	operand 1 = 1111 1111 1111 0000 operand 2 = 1111 0000 1111 0000 x = operand 1 & operand 2 x = 1111 0000 1111 0000
|	The bitwise-inclusive OR operator Compares each bit of its 1st operand to the corresponding bit of its 2nd operand. If either of bits is 1, the result bit set to 1 o/w, result bit set to 0	operand 1 = 0000 1111 0000 0000 operand 2 = 1111 1111 0000 1111 x = operand 1 | operand 2 x = 1111 1111 0011 1111
^	The bitwise-exclusive OR operator Compares each bit of its 1st operand to the corresponding bit of its 2nd operand. If one bit is 0 and the other bit is 1, the result bit is set to 1. o/w the result bit is set to 0	operand 1: 1111 0000 1100 0011 operand 2: 1111 1111 0011 0011 x = operand 1 ^ operand 2 x: 0000 1111 1111 0000
<<	The bitwise shift left operator is a binary operator. The 1st operand is the value to be shifted and the 2nd operand specifies the no. of bits to be shifted.	operand 1: 1111 0000 1111 0001 operand 2: 1 x = operand 1 << operand 2 x = 1110 0001 1110 0010 zero is inserted on right

>>

The bitwise right shift operator is a binary operator. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted

operand1 : 1111 0000 1111 0000
operand2 : 2
x = operand1 >> operand2
x = 00 1111 0000 1111 00

Two zero bits are inserted from left

ASSIGNMENT OPERATOR

Assignment operator assigns the value of expression on the right side of it to the variable on the left on it.

The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left.

assignment expressions that make use of assignment operator have the form:

Identifier = expression ;

When identifier represents a variable and expression represents a constant, a variable or a more complex expression.

Ex:

```
a = 4 ;  /* Integer value is assigned to variable a */
area = length * width ; /* value of length * width is
                           assigned to variable area */
f = 1.034 ; /* floating point value is assigned to
               Variable f */
```

* Key Points :

→ If the two operands in an assignment expression are of different data types, then the value of the expression on the right side of assignment operator will automatically be converted to the type of the identifier on the left of assignment operator.

For Ex,

- A floating value may be truncated, if it is assigned to an integer identifier.
- A double value may be rounded, if it is assigned to a floating-point identifier.
- An integer value may be changed if it is assigned to a short integer identifier or to a character identifier.

Examples with different operand types

Declarations : int i, j = 5 ;

Expression	Value	Expression	Value
i = 5.3	5	i = 2 * j / 2	5
i = -5.6	-5	i = 2 * (j / 2)	4
i = j / 2	2	i = 'A'	65

It is important to note that the expression i = 2 * j / 2 and expression i = 2 * (j / 2) give different results.

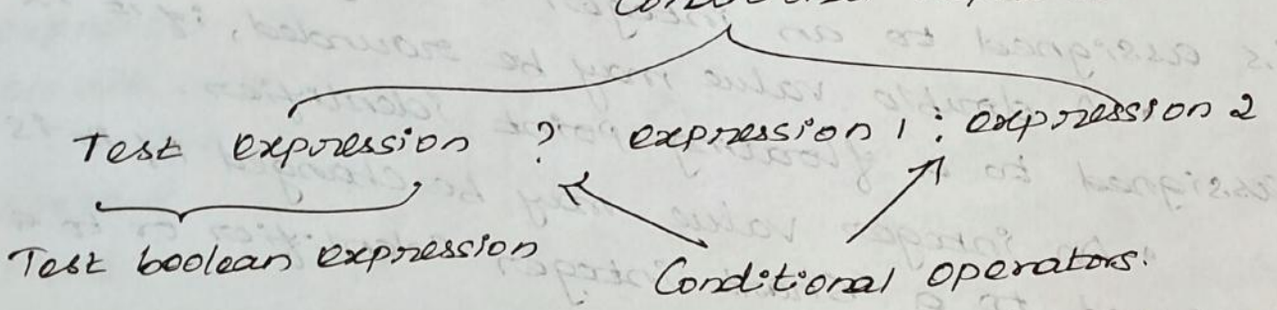
In the latter one, the division is truncated and then followed by the multiplication.

CONDITIONAL OPERATORS / EXPRESSIONS :

Simple Conditional operations can be carried out with the Conditional operator (? :). The Conditional operator (? :) is a ternary operator. Since it takes three operands.

The general form of Conditional expression is,

Conditional operator works as follows :



→ The test expression is implicitly converted to boolean expression. It is evaluated.

→ If the result of test expression is true(1) the expression 1 is evaluated and the Conditional expression takes on value of expression 1.

→ If the result of test expression is false(0) the expression 2 is evaluated and the Conditional expression takes on value of expression 2.

Therefore, the result of the conditional operator is the result of whichever expression is evaluated. the first or the second.

Only one of the last two expression is evaluated in a Conditional expression.

Assume that i and x are the integer variables in conditional expression given below.

Key Points:

→ If the two operands in an assignment expression are of different data types, then the value of the expression on the right side of assignment operator will automatically be converted to the type of the identifier on the left of assignment operator.

For Ex,

- A floating value may be truncated, if it is assigned to an integer identifier.
- A double value may be rounded, if it is assigned to a floating-point identifier.
- An integer value may be changed if it is assigned to a short integer identifier or to a character identifier.

Examples with different operand types

Declarations: int i, j = 5;

Expression	Value	Expression	Value
i = 5.3	5	i = 2 * j / 2	5
i = -5.6	-5	i = 2 * (j / 2)	4
i = j / 2	2	i = 'A'	65

It is important to note that the expression i = 2 * j / 2 and expression i = 2 * (j / 2) give different results.

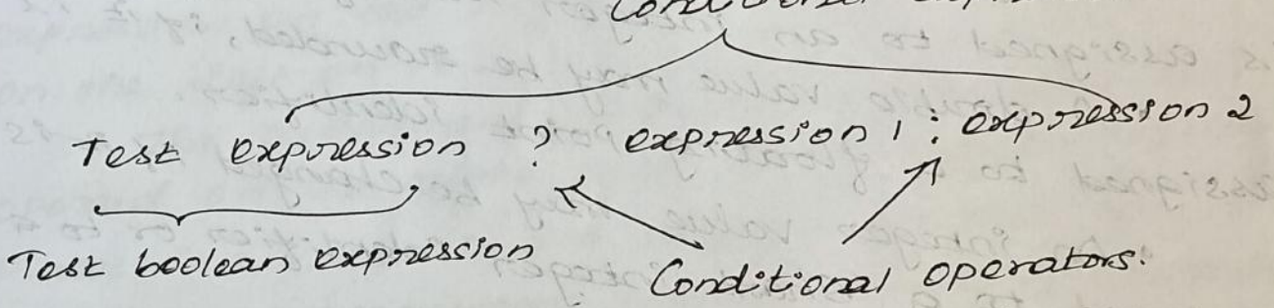
In the latter one, the division is truncated and then followed by the multiplication.

CONDITIONAL OPERATORS / EXPRESSIONS :

Simple Conditional operations can be carried out with the Conditional operator (? :). The Conditional operator (? :) is a ternary operator. Since it takes three operands.

The general form of Conditional expression is,

Conditional operator works as follows :



→ The test expression is implicitly converted to boolean expression. It is evaluated.

→ If the result of test expression is true(1) the expression 1 is evaluated and the Conditional expression takes on value of expression 1.

→ If the result of test expression is false(0) the expression 2 is evaluated and the Conditional expression takes on value of expression 2.

Therefore, the result of the Conditional operator is the result of whichever expression is evaluated - the first or the second.

Only one of the last two expression is evaluated in a Conditional expression.

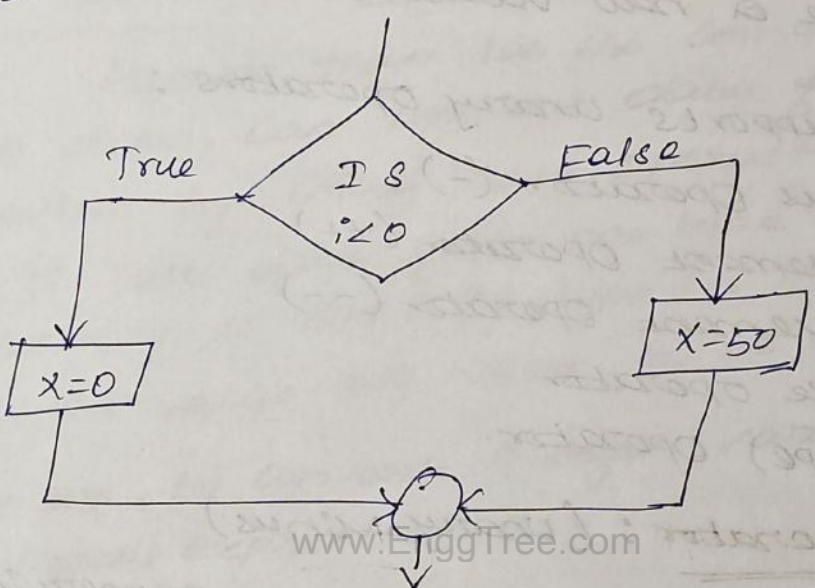
Assume that i and x are the integer variables in Conditional expression given below.

Conditional Expression

```
X = (i<0) ? 0 : 50 ;
```

Test boolean expression — Expression1 — Expression 2

This expression will store 0 in x if i is less than 0 ; Otherwise it will store 50 in x.



Case 2 :

Assume that f, g and y are floating point variables in the conditional expression given below.

```
y = (f<g) ? g : f;
```

Ex :

```
#include<Stdio.h>
Void main()
{
  /* Local Definitions */
  int a=8, b=10, op1, op2;
  /* Statements */
  Op1 = a<b ? a : b;
  Printf("The option 1 = %d\n", op1);
  Op2 = a>b ? a : b;
  Printf("The option 2 = %d\n", op2);
}
```

Output:

The option 1 = 8
The option 2 = 10

EXTRA

➤ UNARY OPERATORS:

The operators that act upon a single operand to produce a new value are known as unary operators.

C supports unary operators:

→ Minus operator (-)
→ Increment operator (++)
→ Decrement operator (--)
→ size operator
→ (type) operator.

1) Minus operator: (Unary Minus)

Unary minus, where a numerical constant, variable or expression is preceeded by a minus sign.

It denotes subtraction (-).

Ex:

-123 , -qty , -(a+b)

2) Increment and decrement operator:

The increment operator (++) adds 1 to the operand, whereas the decrement operator (--) subtracts 1 from the operand.

```
a++; /* Operator is written after the operand */
     Post-increment operator
++a; /* Operator is written before the operand */
     Pre-increment operator
```

3. Size of Operator :

In C programming, an Operator sizeof is used to calculate size of various datatypes. These can be basic or primitive datatypes present in the language, as well as the ones, created by the programmer.

4. Type conversion (Cast Operator)

Rather than let the compiler implicitly convert data, can convert data from one type to another by using cast operator.

To cast data from one type to another it is necessary to specify type in parantheses before the value we want to be converted.

For Ex, to convert integer variable count, to a float, expression as,

(float) count ;

For Ex:

(float)(a+b) ;

➤ INPUT / OUTPUT FUNCTIONS :

I/o Functions are grouped into two Categories

→ Unformatted I/o Functions
→ Formatted I/o Functions

The formatted I/o Functions allow Programmers to specify the type of data and the in which it should be read in or written out.

On the other hand, Unformatted I/o functions do not specify the type of data and the way

is should be read or written.

Function	Formatted	Unformatted
Input	scanf()	getchar(), gets()
Output	printf()	putchar(), puts()

Most of the input/output functions are provided as a part of the C standard library to which can get access through the header file called stdio.h. The stdio.h is an abbreviation for standard input-output header file.

Include this header file in our program using preprocessor directive,

#include<stdio.h>.

It tells compiler to place contents of header file in the program. After compilation, the contents of the header file become a part of the source code.

* Formatted output - The printf Function

C provides the printf function to display the data on the monitor. This function can be used to display any combination of numerical values, single characters and strings.

```
printf("My roll number is %d.\n", rollno);
```

- printf — Function name
- "My roll number is %d.\n" — Control string
- rollno — Print list
- "My roll number is %d.\n", rollno — Function arguments

Eg1: Printf Statement.

The printf statement consist of two parts:

→ Function name → Function arguments enclosed in parentheses.

The arguments for printf consists of a control string (in quotes) and a print list (the variable) ● variable rollno is 10 then,

My roll number is 10.

The %d in the printf statement is known as placeholder or conversion character or format code or format specifier.

The printf function uses different placeholders to display different type of data item.

Placeholder	Type of data item displayed
%.c	Single character
%.d	Signed decimal integer
%.e	floating point number with an exponent.
%.f	Floating point number without an exponent.
%.g	FP number either with exponent or without exponent depending on value.
%.i	Signed decimal number
%.o	Octal number, without leading no zero
%.s	String
%.u	Unsigned decimal integer
%.x	hexadecimal number, without leading zero.

Fig: Commonly used placeholders in printf function.

h → letter h tells short integers.
l → letter l tells long integers.
L → L letter as double long.

* Formatting Integer output:

It seen that how integer numbers can be displayed on the monitor using %d placeholder.

In case of integer numbers, the placeholder can accept modifiers to specify the minimum field width and left justification.

By default, all output is right justified. It force the information to be left justified by putting a minus sign directly after the %.

Printf Statement	Output (assume x = 1234)
Printf("%d", x);	1 2 3 4
Printf("%6d", x);	_ _ 1 2 3 4 Right justified.

However, if the number is greater than the minimum field width, the great number is printed in full, overriding the minimum field width specification.

It is also possible to pad the empty places with zeros.

Printf("%3d", x);	1 2 3 4	overriding the minimum field width
Printf("%06d", x);	0 0 1 2 3 4	Padding with zeros.

* Formatting Floating point Output:

To add a modifier to place a decimal point followed by the precision after the field width specifier.

For Ex, %.8.4f will display a number atleast 8 characters wide including decimal point with four decimal places.

Printf Statement | Output (assume y=3456.120650)

Printf("%f", y); → 3 4 5 6 . 1 2 0 6 5 0 (default Precision is 6.)

6 decimal places

Printf("%7.2f", y); → 3 4 5 6 . 1 2

2 decimal places

7 characters wide

* Formatting string output:

When the precision modifier is applied to string, the number preceding period specifies the minimum field width and the number following the period specifies the no. of characters of the string to be displayed.

For Ex: %16.9s will display a string that will be atleast sixteen character long, however only 1st 9 characters from the string are displayed with remaining blank characters.

Assume str: Programming IN C

Ex:

Printf("%.8s", str): P R O G R A M M I N G | I N | C

* Enhancing the readability of output:

Commonly use two specifiers to enhance the readability of the output.

\t : tab character.

\n : newline character.

Escape Sequences	Function
\n	Newline
\t	Tab
\b	Backspace
\f	Form feed
\'	Single quote
\\	Backslash
\"	Double quote
\r	Carriage return
\a	Alert.

=> Unformatted Output - putchar and puts Function

The unformatted output functions such as putchar() and puts() concerned with displaying a single character or string of characters on the display monitor.

They are included in the Stdio.h header file.

#include <Stdio.h> Smt as a preprocessor directive

* Single character Output - The putchar Function

The putchar function displays one character on the display monitor. The character to be displayed is of type char.

Syntax for putchar fn:

putchar(ch-var);

* Character String Output - The puts Function

The puts function displays a string of characters on the display monitor followed by a newline.

Prototype is,

puts(string);

⇒ Formatted Input - The Scanf function

C provides the scanf function to read data from the standard input device. It is general purpose input routine.

It can read all the built-in data types and automatically convert numbers into the proper internal format.

scanf ("control string", address_list);
↑ function name. ← function arguments

Like printf function, the scanf function uses format specifiers preceded by a % sign to tell scanf function what type of data is to be read.

* Unformatted Input - getchar and gets function:

The unformatted input functions such as getchar() and gets() are concerned with reading a single character or multiple characters from the keyboard.

Ex: #include <stdio.h>

* Single Character Input - The getchar function

The getchar function accepts a single character from the keyboard. The fn does not require any arguments, though a pair of empty parentheses must follow the word getchar as a syntax.

```
ch_var = getchar();
```

* String Input - The gets Function :

The gets function reads a string of characters from the keyboard and stores them at the address pointed by its argument.

It reads the string of characters until the Enter or Return key is pressed.

The Prototype for get() is

```
gets(str);
```

➤ Built-In Functions in C :

Built-in functions in c are part of Standard libraries like Stdio.h, math.h, string.h etc. This functions help with common tasks.

Header File	Function	Purpose
math.h	Sqrt(x)	Square root
	Pow(x,y)	Power (x^y)
String.h	Strlen(s)	Length of string
	Strcpy()	Copy one string to another
	Strcmp()	Compare two strings
Ctype.h	Toupper(c)	Convert to uppercase
	Tolower(c)	Convert to lowercase

Ex: Using math.h

```
#include <Stdio.h>
#include <math.h>
int main()
{
```

```
    double x = 16.0;
    Printf("Square root of %.2f is %.2f\n", x,
                                        sqrt(x));
    Printf("2 raised to 3 is %.2f\n", pow(2,3));
    return 0;
}
```

Example using string.h

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "Hello";
    char str2[20];
    strcpy(str2, str1); // copy str1 to str2
    Printf("Copied string: %s\n", str2);
    Printf("Length of string: %lu\n", strlen(str1));
    return 0;
}
```

CHAPTER-2 (UNIT-II)

Control Structures : if, if-else, nested if, Switch-Case, while, do-while, for, nested loops, Jump Statements.

CONTROL STRUCTURES / FLOW CONTROL / CONTROL STATEMENTS IN C :

Control Statements enable us to specify the order in which the various instructions in the program are to be executed.

It determines the flow of control in C.

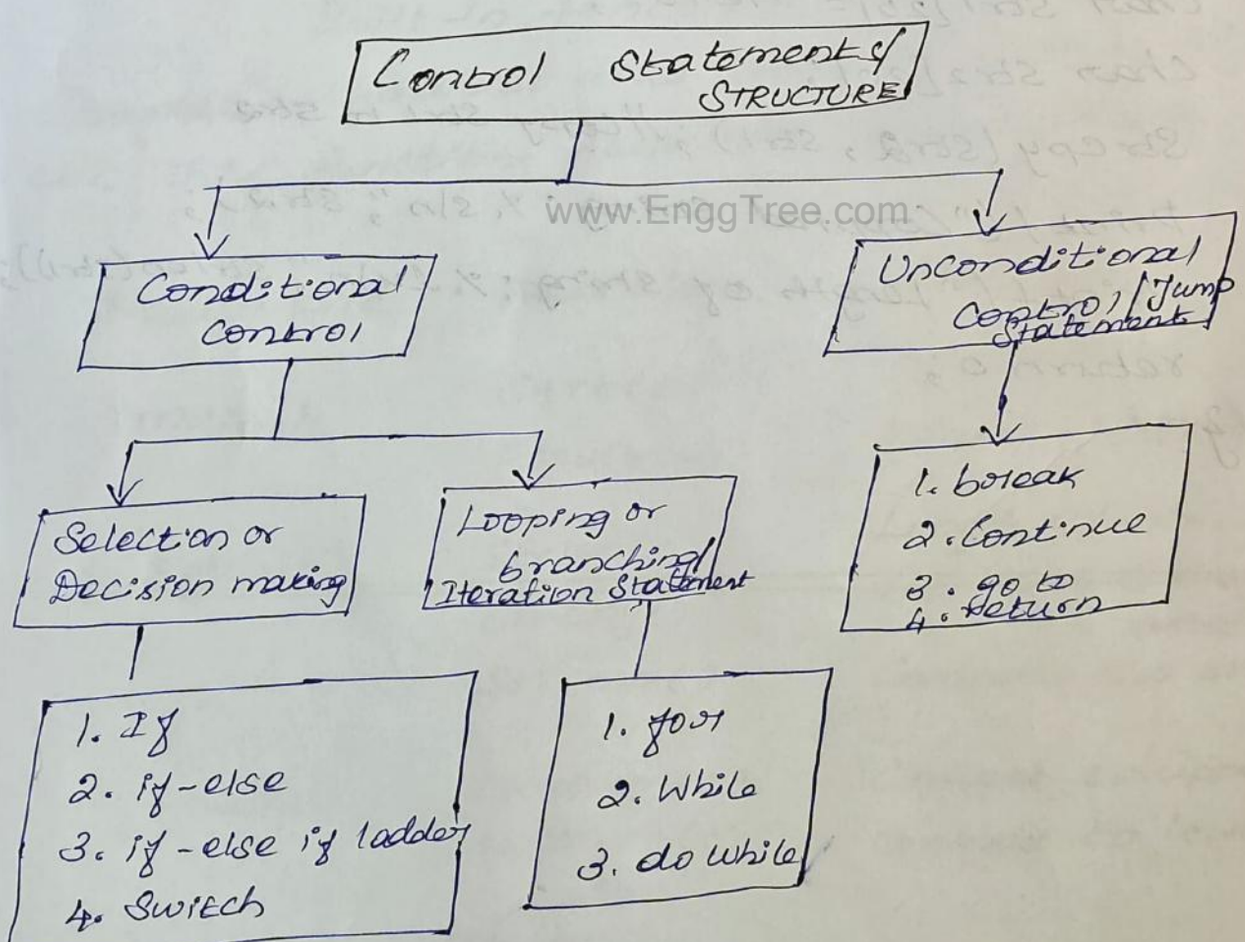


FIG : Control Structures provided by C.

Compound Statements or a block are a group of Statements which are enclosed within a pair of curly braces {}.

A Compound Statement is syntactically equivallent to a Single Statement.

➤ DECISION MAKING : The if-else Statement :

If → The if Statement is the simplest decision making Statement. It is used to decide whether to do Something at a special point, or to decide between two courses of action.

If-else Bi-directional Conditional Control Statement.

The Statement tests one ore more conditions and executes the block based on the outcome of the test.

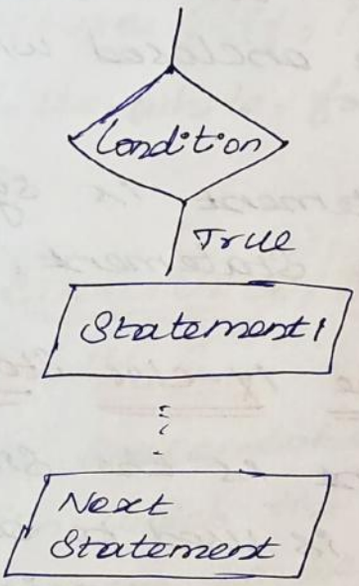
Any non-zero value is regarded as true, Whereas 0 is regarded as false.

* if Statement : (Single if Statement).

Executes code only if condition is true.

Syntax :

```
if (condition)
    Statement1 ;
    or
if (condition)
  {
    Statement ;
      :
      -
  } -
```



Ex:

```
#include<Stdio.h>
Void main()
{
  int x=10;
  if(x>5)
  {
    Printf("x is greater than 5\n");
  }
}
```


* If-else Statement

- Chooses between two blocks.

If we wish to have more than one Statement following the if or the else, they should be grouped together between curly brackets.

Such a grouping is called a Compound Statement or a block.

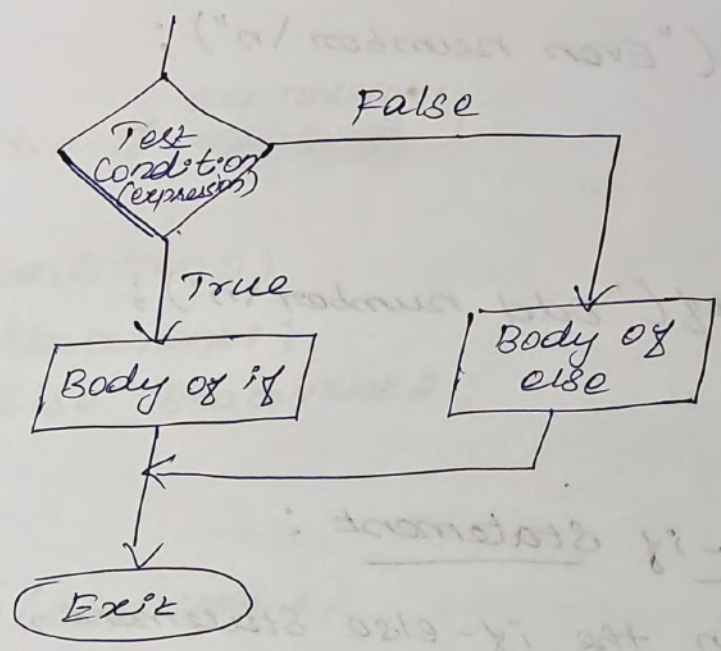


Fig: operation of if-else statement

Syntax:

```
if (condition)
    Statement1;
else
    Statement2;
```

(or)

```
if(condition)
 {
  Statement;
 }
else
 {
  Statement;
 }
```

Ex:

```
#include <stdio.h>
void main()
{
  int num = 5;
  if (num % 2 == 0)
  {
```

```
    Printf("Even number \n");
}
else
{
    Printf("Odd number \n");
}
}
```

* Nested - if Statement :

In the if-else statement, if body of either if or else or both include another if-else statement, the statement is known as nested if statement. Nested ifs are very common in programming.

They are used when there are two or more alternatives to select.

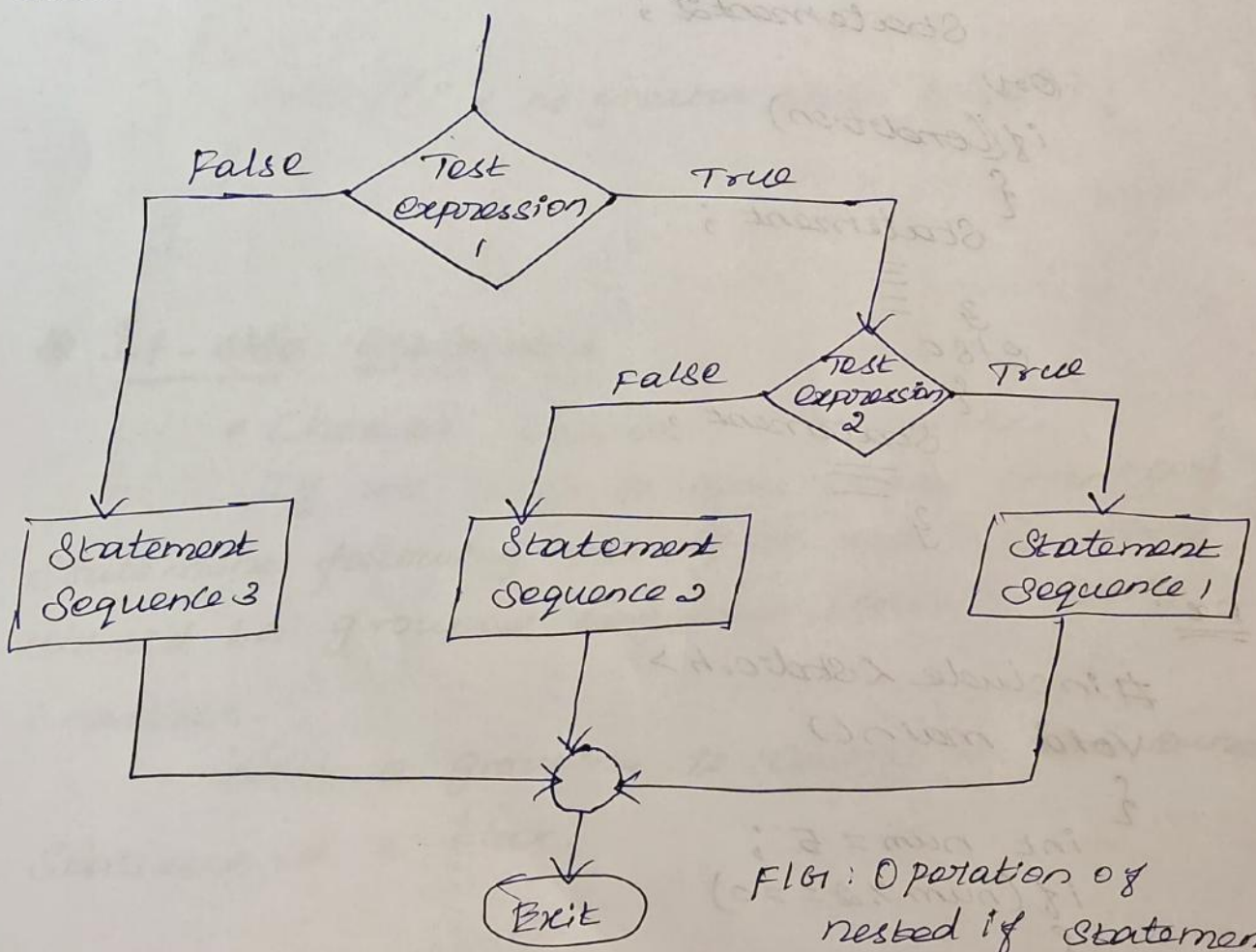


FIG : Operation of nested if statement

Syntax:

```
if (condition1)
    {
      if (condition2)
         Statement1;
         else Statement2;
      }
    else
    {
      if (condition 3)
         Statement 3;
      else
         Statement 4;
      }
```

Ex:


```
#include <stdio.h>
void main()
{
   int n1, n2;
   Printf("Enter two numbers: \n");
   Scanf("%d%d," &n1, &n2);
   if (n1>=n2)
      if (n1>n2)
         Printf("%d is greater than %d", n1, n2);
      else
         Printf("%d is equal to %d," n1, n2);
   else
      Printf("%d is less than %d", n1, n2);
}
```

Output: (Run 1)

Enter two numbers: 11 12

11 is less than 12.

O/P

Enter two numbers : 45 40
45 is greater than 40.

O/P

Enter two numbers : 15 15
15 is equal to 15.

* Dangling else problem :

Once start nesting if-else statements, we may encounter a classic problem known as dangling else.

This problem is created when there is no matching else for every if. The solution to this Problem is to follow the simple rule,

Always Pair an else to the most recent unpaired if in the current block.

else is always paired with the most recent unpaired if.

From the Code alignment, Conclude that the programmer intended the else statement to be Paired with the first if. However the compiler will pair it with the second if.

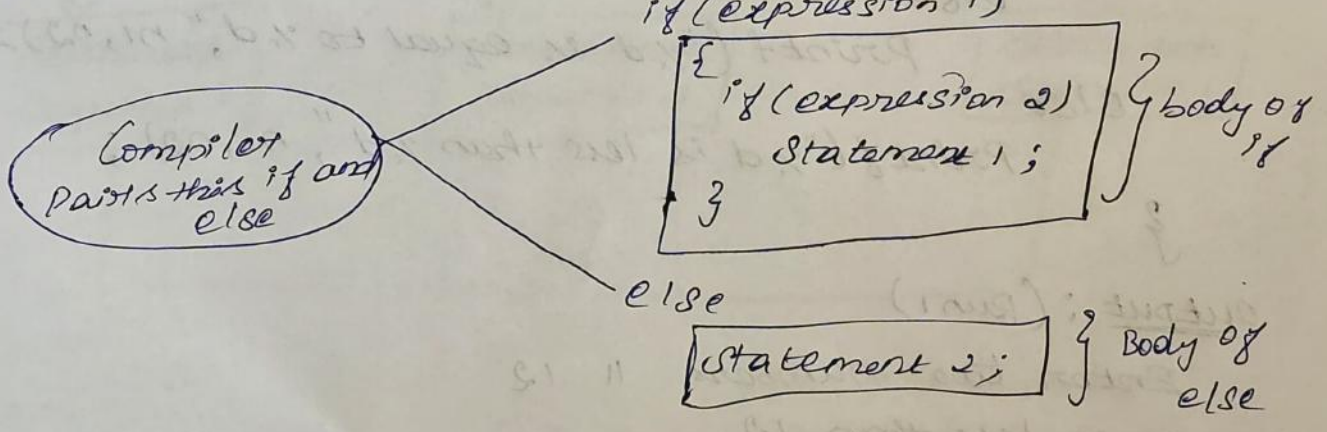


Fig: Dangling else solution.

* The if-else-if Ladder

Sometimes we wish to make a multi-way decision based on several conditions. The most general-way of doing this is by using the else if variant on the if statement.

This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed.

If none of the conditions is true, the final else is executed. The final else statement is optional. If final else is not present, no action takes place if all other conditions are false.

Syntax:

```
if (expression)
    Statement ;
else if (expression)
    Statement ;
else if (expression)
    Statement ;
else if (expression)
    Statement ;
else
    Statement ;
```

It is up to the programmer to device the correct structure for each programming problem.

Ex:

```
#include<Stdio.h>
Void main()
{
    int m;
    Printf("Enter the marks:");
    Scanf("%d",&m);
    if(m>=75)
        Printf("Grade:Distinction");
    else if(m>=60)
        Printf("Grade:First class");
        else if(m>=50)
            Printf("Grade:Second class");
                elseif(m>=40)
                    Printf("Grade:Pass class");
                        else
                            Printf("Grade:Fail");
}
```

O/P

Enter the marks:68
Grade:First class

O/P

Enter the marks:15
Grade:Fail

O/P:

Enter the marks:89
Grade:Distinction.

Switch Statement :

The Switch Statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values and branches accordingly.

This is another form of the multi-way decision. It is well-structured, but can only be used in certain cases.

Where,

Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).

Each possible value of the variable can control a single branch. Default branch may optionally be used to trap all unspecified cases.

- Multi-directional conditional control statement.
- Popularly used for menu driven programs.

Uses three keywords : switch, case, break, default.

* Expression : Can be anything :

1) An expression yielding an integer value.
2) Value of any integer
3) Character variable
4) Function call returning a value.

Syntax:

```
Switch (expression)
{
  Case constant 1 : Statement ;
                    break ;
  Case constant 2 : Statement ;
                    break ;
  ------
  ------
  ------
  default Statements ;
}
```

*

Should be integer or character type.

Can be constants or constant expressions.

Ways of writing switch expression and constants

Valid

```
int a,b,c; float f;
char d,e;
Switch(a)
Switch(a>b)
Switch(d+e-4)
Switch(a>b&&b>c
Switch(func(c,d))
```

Case constants

```
Case 4 :
Case 'a' :
Case 'a'+5' :
```

Invalid

```
int a,b,c; flat f;
char d, e;
Switch(f)
Switch(a+b-2)
```

Case constants

```
Case second :
Case 3.2 :
Case a :
Case a>b :
Case b+4 :
Case 2,b,8
Case 10 : 11 : 12
```

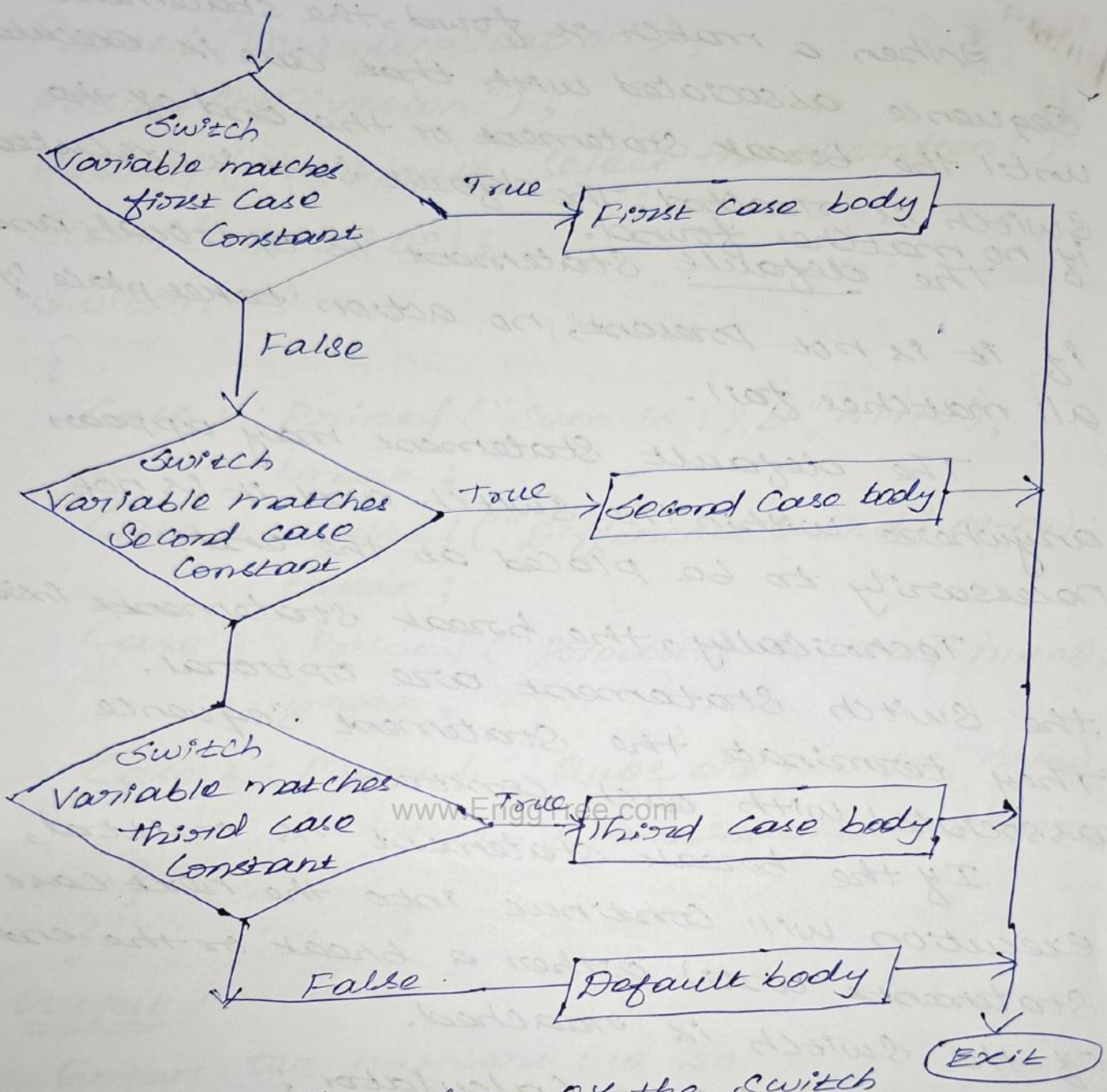


FIG : Operation of the switch Statement.

The keyword switch is followed by a switch variable or expression in parenthesis. Each case keyword is followed by a constant, which is not in parenthesis but is followed by a colon.

When the switch statement is executed, the expression is evaluated and its value is successively compared with the case constants.

When a match is found, the statement sequence associated with that case is executed until the break statement or the end of the switch is reached. The default stmt is executed if no matches found.

The default statement is optional, and if it is not present, no action takes place if all matches fail.

The default statement may appear anywhere within the switch and it is not necessarily to be placed at the end.

Technically, the break statements inside the switch statement are optional. They terminate the statement sequence associated with each constant.

If the break statement is omitted, execution will continue into the next case's statements until either a break or the end of the switch is reached.

Ex: Menu driven Calculator:

```
#include <stdio.h>
Void main()
{
  int n1, n2, OP;
  Printf("Enter two numbers:");
  Scanf("%d %d", &n1, &n2);
  Printf("\n Menu");
  Printf("\n1: Addition");
  Printf("\n2: Subtraction");
```

```
Printf("\n3.Multiplication");
Printf("\n4.Division");
Printf("\n Please Select the operation:");
Scanf("%d",&OP);
Switch(OP)
{
  Case 1 : Printf("Sum is : %d", n1+n2);
           break;
  Case 2 : Printf("Difference is : %d", n1-n2);
           break;
  Case 3 : Printf("Product is : %d", n1*n2);
           break;
  Case 4 : Printf("Quotient is : %d", n1/n2);
           Printf("\n Remainder is : %d", n1%n2);
  }
}
```

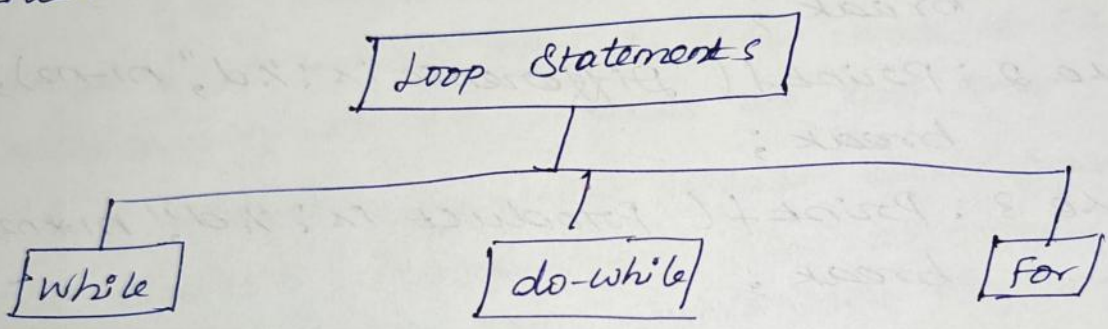
Output :

```
Enter two numbers : 10 20
Menu
1 : Addition
2 : Subtraction
3 : Multiplication
4 : Division
Please select the Operation : 1
Sum is : 30
```

LOOP STATEMENTS:

Loops used when one wants to execute a part of a program or a block of statements several times.

With the help of loops, one can execute a part of program repeatedly till some condition is true.



* It repeats a statement or group of statements until a given condition is true.
* It tests the condition before executing the loop body.

* Similar to while
* It tests the condition at the end of loop body.

* Executes the sequence of stmts multiple times and abbreviates the code segment that manages the loop variable.

Fig: Types of loop statements.

WHILE STATEMENT

The while loop is a pre-test loop. Since it is a pre-test loop, it tests the expression before every iteration of the loop.

It repeats a loop body until the logical test at the top proves false.

The while loop looks like a simplified version of the for loop.

It contains a test expression but no initialization or update expressions. Although there is no initialization expression, the loop variable must be initialized before the loop begins.

The loop body must also contain some statement that updates the value of the loop variable, otherwise the loop would never end.

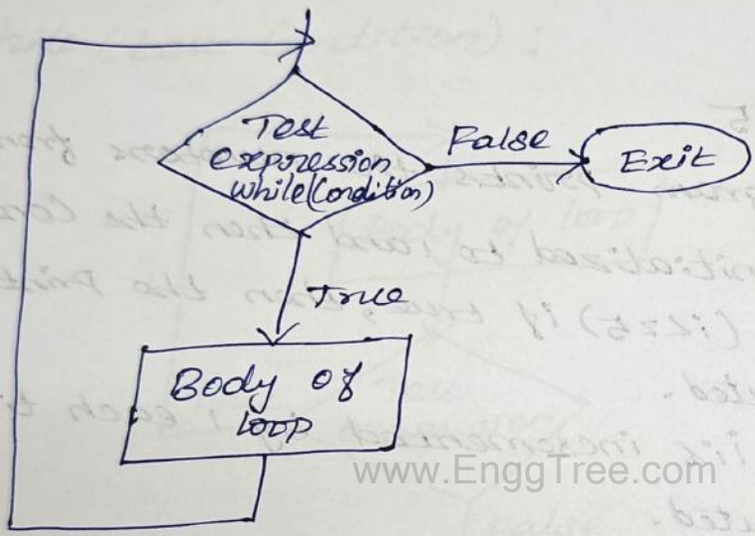


Fig: Operation of the while loop.

* Syntax:

```
While (test condition)
    Statement ; //Body of the loop with single stmt
```

Note: Curly braces are optional if loop body consists of single statement.

```
While (test condition)
{
    Statement ;    //Body of the loop with multiple
    Statement ;                Statements
}
```

Ex:

```
#include<Stdio.h>
main()
{
int i=1;
while(i<=5)
{
printf("%d\t", i);
i++;
}
}
```

Output: 1 2 3 4 5

This C program prints the numbers from 1 to 5. The variable i is initialized to 1 and then the condition is checked whether (i<=5) if true, then the Printf Statement is executed.

The value of i is incremented by 1 each time the loop is executed.

DO-WHILE STATEMENT

The do-while loop is very similar to the while loop except that the test occurs at the end of the loop body.

This guarantees that the loop is executed atleast once before terminating.

The do while loop is frequently used where data is to be read. The test then verifies the data and loops back to read again if it was unacceptable.

Syntax:

```
do
Statement;          //Body of the loop with
                      Single Statement.
while(test Condition);
```

Note : Curly braces are optional if loop body consists of single statement.

```
do
{
  Statement ;
  Statement ;        // Body of the loop with multiple Statements
}
while (test Condition) ;
```

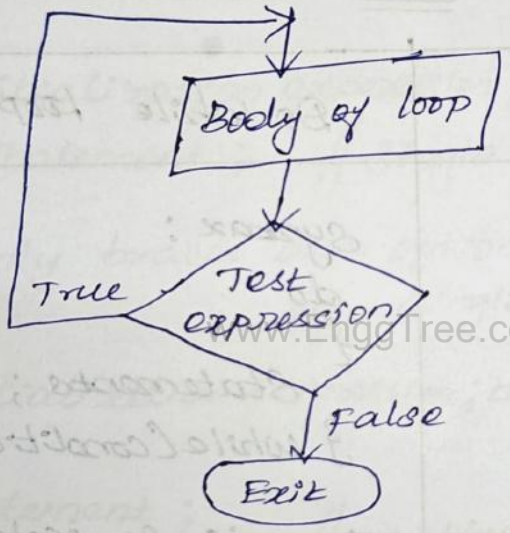


Fig: Operation of the do-while loop.

Ex: Reverse the digits of given integer number.

```
#include<stdio.h>
int main()
{
   int n, reverse = 0;
   Printf("Enter a number to reverse\n");
   Scanf("%d", &n);
   do
   {
     reverse = reverse * 10;
     reverse = reverse + n % 10;
     n = n/10;
   }
```

```
while(n!=10);
{
  Printf("Reverse of entered number is = %d\n", reverse);
}return 0;
}
```

Output:

Enter a number to reverse
Reverse of entered number is = 4321

⇒ Difference between while and do-while loop:

While Loop	Do-while Loop

1. Syntax:
while(condition)
{
statements;
}
	Syntax:
	do
	{
	statements;
	} while(condition);

2. NO semicolon after Condition in the Syntax	There is semicolon after Condition in the syntax.
3. Entry Controlled loop.	Exit Controlled loop.
4. If the Condition is not true loop is never executed.	Loop is executed atleast once eventhough condition is not true.

FOR LOOP STATEMENT

The for loop is a most commonly used loop Statement. It is pre-test loop and it is used when the number of iterations of the of the loop is known before the loop is entered.

The head of the loop consists of three parts (initialization expression, test expression and update expression) separated by Semicolons.

Syntax:

```
for(initialization expression ; test expression ; update expression)
    Statement ;  // Single Statement loop body
```

Note: Curly braces are optional if loop body consists of single Statement.


```
for(initialization expression ; test expression ; update expression)
{
  Statement ;
  Statement ;   // Multiple Statement loop body.
  Statement ;
}
```

The initialization expression is run before the loop is entered. This is usually the initialisation of the loop variable.

The second is a test, the loop is exited when this returns false.

The third is a statement to be run every time the loop body is completed. This is usually an increment or decrement of the loop counter.

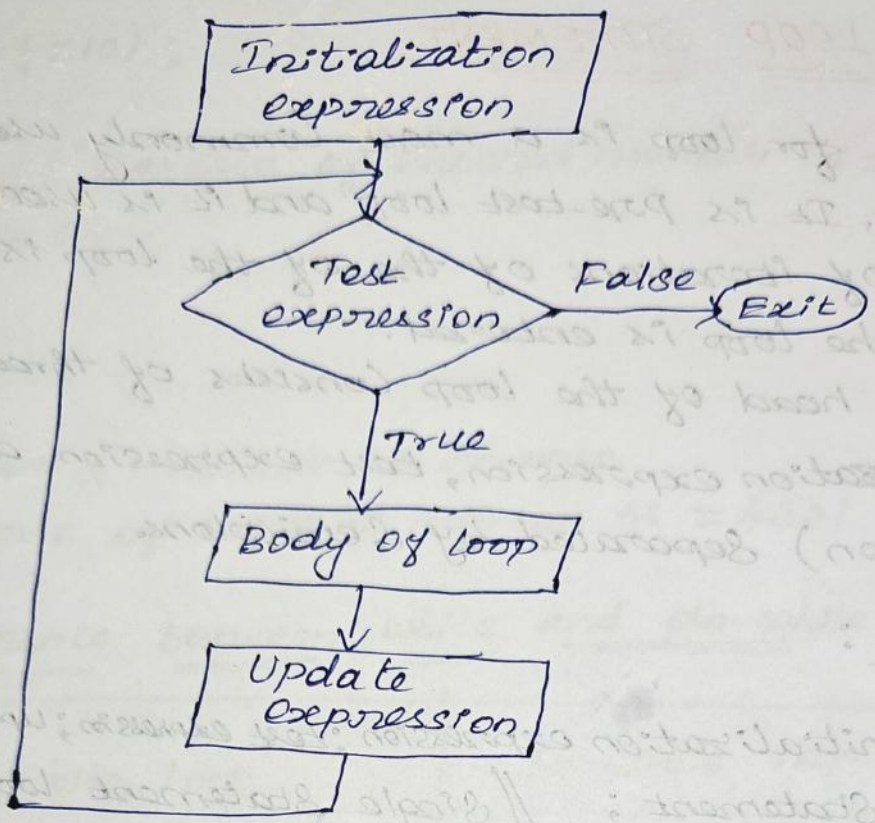


Fig: Operation of the for loop.

Ex: Displays the square of numbers

```
/* This program displays the square of numbers
   written by:
   Date:
*/
#include<stdio.h>
main()
{
  int n, sq;
  for(n=10; n>=5; n--)
  {
    sq = n*n;
    Printf("\n square of %d is %d", n, sq);
  }
}
```

Output :

```
Square of 10 is 100
Square of 9 is 81
Square of 8 is 64
Square of 7 is 49
Square of 6 is 36
Square of 5 is 25
```

Explanation : The body of for loop contains multiple(2) statements. Thus the body of for loop is enclosed in the curly braces. Note that the loop is negative running : n is initialized to 10 and it is decremented by 1 each time the loop repeats.

➤ NESTED FOR LOOPS :

Like if statement, for statement can be nested. The one for statement within another for statement is called nested for loops.

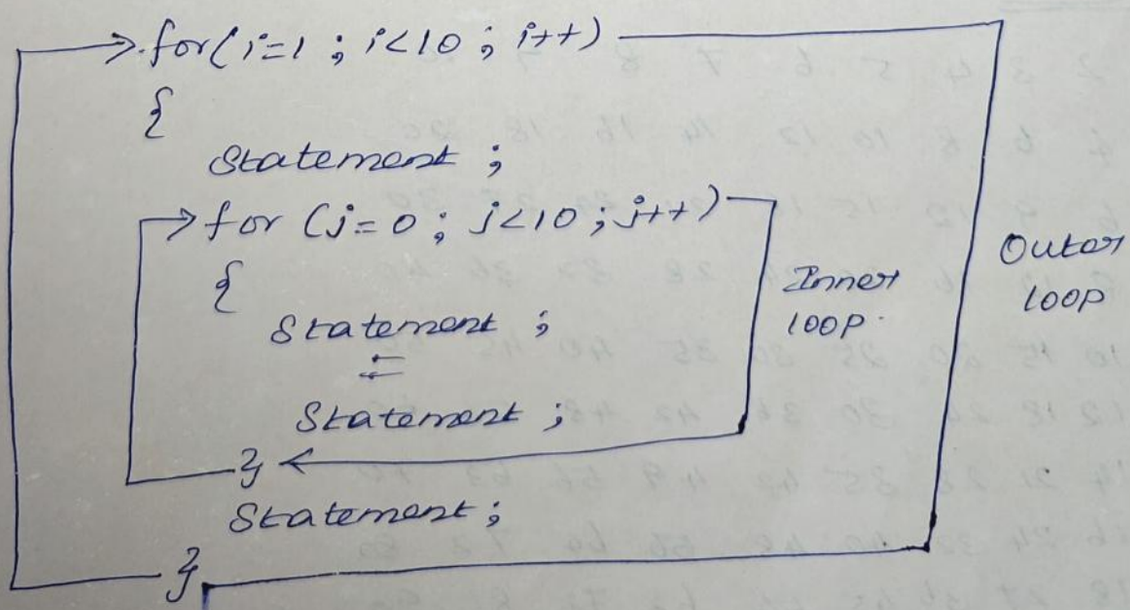


Fig : Nested for loops.

To understand how nested for loops work, look at the programs given below.

Ex:

```
#include <stdio.h>
Void main()
{
  int i,j;
  for(i=1; i<=10; i++) /*Outer for loop*/
  {
    for(j=2; j<=10; j++) /*Inner for loop*/
    {
      Printf("\t%d",j*i);
    }
    Printf("\n"); /*Display on the next line*/
  }
}
```

Output:

2	3	4	5	6	7	8	9	10
4	6	8	10	12	14	16	18	20
6	9	12	15	18	21	24	27	30
8	12	16	20	24	28	32	36	40
10	15	20	25	30	35	40	45	50
12	18	24	30	36	42	48	54	60
14	21	28	35	42	49	56	63	70
16	24	32	40	48	56	64	72	80
18	27	36	45	54	63	72	81	90
20	30	40	50	60	70	80	90	100

JUMP STATEMENT

Jump Statements are,

→ break
→ Continue
→ goto
→ return.

1) Break Statement :

The break statement has two uses. To seen the use of it to terminate a case in the switch statement.

Also use break statement to force immediate termination of loop, bypassing the normal loop conditional test when break is encountered inside any loop, control automatically passes to the first statement after the loop.

A break is usually associated with an 'if'.

Syntax :

```
break;
```

* Popularly used in switch statements.

While loop :

```
While (condition)
{
  Statement;
  if (Condition to break) {
    break;
  }
  Statement;
}
```

→ Control.

Do-while loop

```
do{
  Statement;
  if(condition to break){
    break;
  }
  Statement;
}while(condition);
```

→ Control

For loop:

initialization ↑ Condition ↑ Updation ↑

```
for (expression 1; expression 2; expression 3)
  { Statement;
    if (Condition to break) {
      break; }
    Statement;
  }
```

→ Control

Ex:

```
#include<stdio.h>
void main()
{
  int i;
  for(i=0; i<100; i++)
  {
    Printf("%d\t",i);
    if (i==10)
      break;
  }
}
```


Output: 0 1 2 3 4 5 6 7 8 9 10

Continue Statement:

Used when one wants to skip some Statements and execute the next iteration.

Syntax:

```
continue;
```

→ Generally used with Condition Statements, when a Condition Statements is encountered and is true, then the Statements after the keyword Continue are skipped & the loops Condition is

checked for next iteration.

In break, the loop terminates and control is transferred to outside. In continue the current iteration terminates and the control is transferred to the beginning of the loop.

Ex:

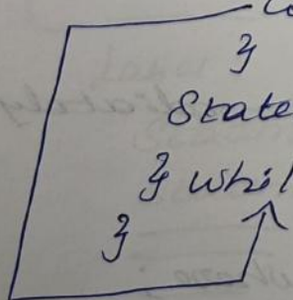
While loop.

```
While(condition){
    Statement ;
    if (condition)
    ——————
    }
    Statement ;
```

As soon as the condition of if is true, the continue statement is executed. This results in control being given to the while condition and next statements are skipped.

Do-While Loop

```
do {
    Statement ;
    if(condition){
        Continue ;
    }
    Statement ;
} while (condition) ;
}
```



* For Loop :

```
for (expression 1 ; expression 2 ; expression 3)
  {
   Statement ;
   if (Condition) {
      Continue ;
     }
    Statement ;
  }
```

* Goto Statement :

Unconditional control Statement

Transfers the flow of control to another part of the program.

Syntax :

```
goto label ;
   Statement ;
   Statement ;
  - - - -
  - - - -
  - - - - -
label :
   Statement ;
   Statement ;
  - - - -
  - - - - -
```

The control is transferred immediately after the label.

The label can be placed anywhere ;

Goto

Forward goto

→ Also known as forward jump
→ The label is placed after the goto statement
→ Control is transferred forward
→ The statements between goto and label are not executed.

Backward goto

→ Also known as backward jump.
→ The label is placed before the goto statement.
→ Control is transferred backward.
→ The statements between the goto and label are executed repeatedly.

→ The control in goto can be transferred within a function.

Forward goto

```
goto label;
Statement;
Statement;
Statement;
=====
label:
Statement;
Statement;
=====
```

Skipped

executed

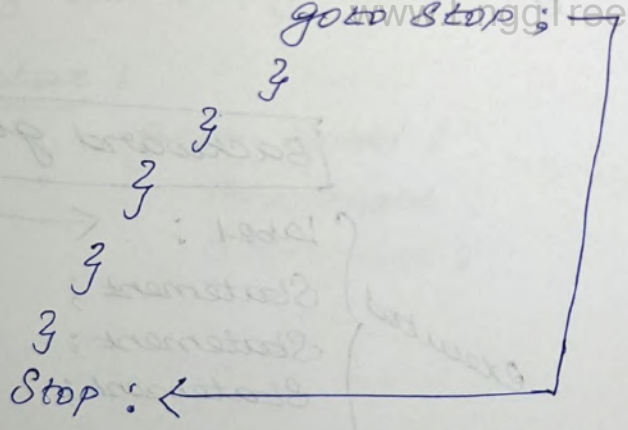
Backward goto

```
label:
Statement;
Statement;
Statement;
=====
goto label;
Statement;
Statement;
=====
```

executed

Skipped

```
for(initalization ; Condition ; Updation)
  {
    while(condition)
    { for(initialization ; condition ; updation)
      {
        do
          {
            ===
            ===
            ===
          }
          while(condition);
        while(condition)
        {
          if(condition)
            {
              goto Stop;
            }
        }
      }
    }
  }
Stop :
```



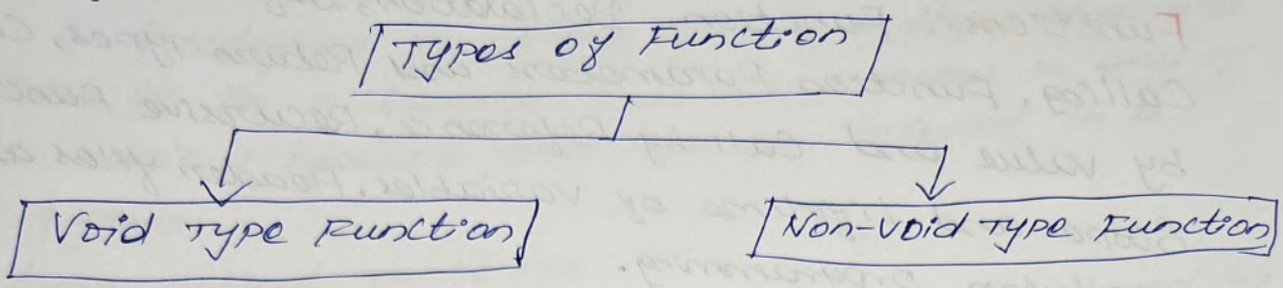
* exit from the deeply nested loops.

* Return Statement :

Ends the execution of function and returns the control of execution to the calling function.

Doesn't mandatorily need any conditional statements.

Depending on the type of function it may or may not return values.



- Return Statement may or may not return a value.
- Return Statement must return a value.

Syntax : return expression;

Ex: Void Function

```
Void fun()
{
}
```


Non-void function

```
return-datatype fun()
{
  return value;
}
```

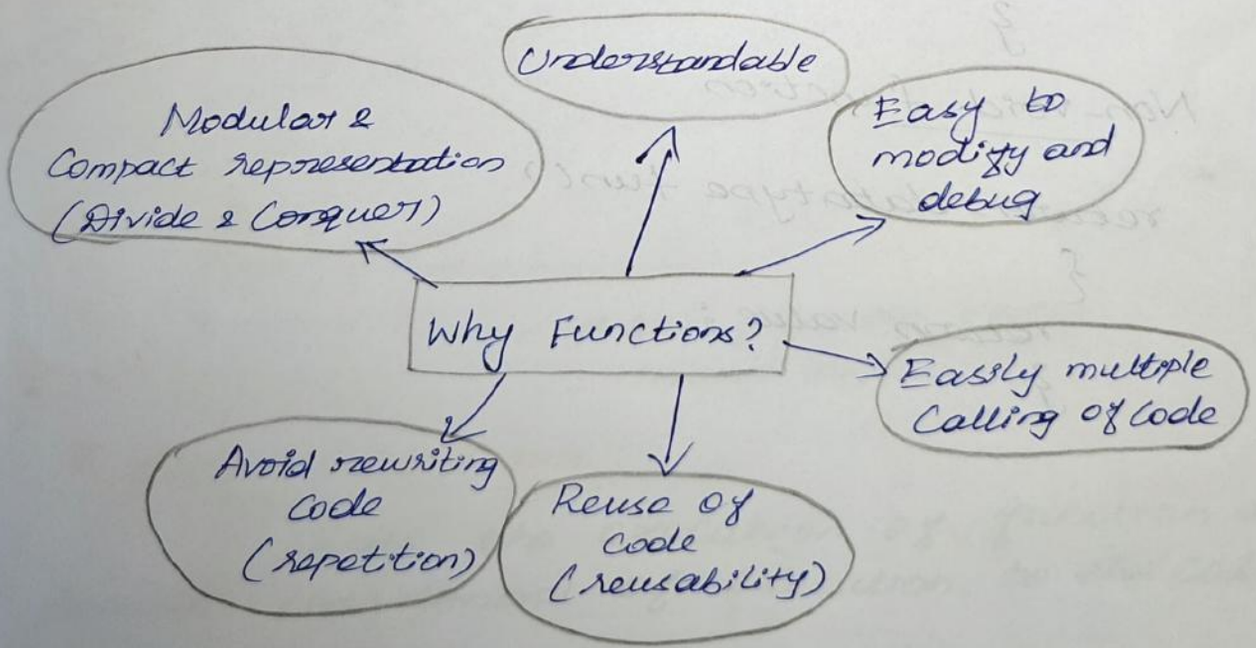
CHAPTER-III [UNIT-III]

Functions: Function Declaration, Definition and Calling, Function Parameters and Return Types, Call by value and Call by Reference, Recursive Functions, Scope and Lifetime of Variables, Header files and Modular programming.

➤ CONCEPT OF FUNCTION / Function Definition:

Self Containing block of one or more Statements or a Sub-program which is designed for a particular task is called **function**.

Every C program has atleast one function, which is main() and all the most trivial programs can define additional functions.



* Library functions or built in functions or Predefined functions:

Predefined functions in C are those that are part of the C library. These are present in any of the various headers that can be included in a program.

* User-defined functions:

User defined function has to be developed by the user at the time of writing a program.

⇒ Elements of user-defined Functions:

There are three elements,

1) Function declaration: All identifiers in C need to be declared before they are used. This is true for functions as well as variables. For functions the declaration needs to be before the first call of the function.

2) Function definition: It is actual code for the function to perform the specific task.

3) Function call: In order to use function in the program use function call to access the function.

The program or function that calls the function is referred to as calling program or calling function.

⇒ General form of a Function / Function Definition

The function definition contains the function-name with parameter list and the type specification of the value return by the function (optional) and the function body.

- Function Definition Syntax:

```
returntype function name (parameters)
{
  variable declarations;
  Statement block;
  return value;
}
```

Note: Return type, parameters, variable declaration and return value are optional.

* Key-points:

- return-type : type of value returned by function or void if none.
- Function-name : Unique name identifying function
- Parameters : Comma separated list of types and names of parameters.
- Value : Value returned upon termination (not needed if return-type void)

The list of parameters is a declaration of the form type 1 par 1, ..., type n par n and represents external values needed by the function. The list of parameters can be empty.

All declarations and statements that are valid in the main function can be used in the function definition.

Ex: Computing averages

```
double average (double x, double y)
{
  return (x+y)/2;
}
```

FUNCTION DECLARATION / FUNCTION PROTOTYPE

Like variables a function must be declared before it can be used (called). The function declaration includes the return type and the number and type of the arguments.

This is also called the function prototype. The declaration of a function resembles the definition,

* Function declaration:

return_type function-name (Parameter types);

Ex: declaring average

double average (double, double).

In declaration, the function body is replaced by a semi-colon.

The function Parameters declared in the header of the function declaration is called Formal Parameters. This list defines and declare the variables that will contain the data received by the function.

Formal Parameters need not be named, it is suffficient to specify their types.

* Accessing a Function: / Function call

A function can be accessed (i.e, called) by specifying its name, followed by a list of Parameters enclosed in parenthesis and separated by commas.

When function doesn't require any argument it is called with function name followed by an empty pair of parenthesis.

The arguments in the function call are known as actual parameters. The actual parameters identify the values that are to be sent to the called function.

They match the function's formal parameters in type and order in the parameter list.

* Function Types:

For better understanding of arguments and return value from the function, user-defined functions can be categorized as:

→ Function with no arguments and no return value.

→ Function with no arguments and a return value.

→ Function with arguments and no return value

→ Function with arguments and a return value.

Ex: C Program to Calculate the Sum of two integer numbers using 4 categories of user-defined function.

1. Function with no argument and no return value

```
#include<Stdio.h>
Void addition(); // Function declaration
Void main()
{
 addition(); // Function call with no arguments
}
```

```
Void addition()
{
  int Sum, a=10, b=20;
  Sum = a+b;
  Printf("\n Sum of a = %d and b = %d is = %d", a,
                                        b, Sum);
}
```

Output :

Sum of a=10 and b=20 is =30.

2. Function with no argument and a return value :

```
#include <Stdio.h>
int addition(); //Function Declaration
Void main()
{
  int Sum;
  Sum = addition(); //Function call with no argum ents;
  Printf("%d", Sum);
}
int addition()
{
  int a=10, b=20;
  Printf("\n Sum of a = %d and b = %d is = ", a, b);
  return(a+b); //Return value
}
```


Output :

Sum of a=10 and b=20 is=30

* Function with argument and no return value:

```
#include<Stdio.h>
Void addition(int, int) //Function declaration
Void main()
{
   int x=10, y=20;
   addition(x,y); // Function call with arguments
}
Void addition(int a, int b)
{
   int sum;
   Sum=a+b;
   Printf("\n Sum of a=%d and b=%d is=%d", a,
                                   b, Sum);
}
```

Output:

Sum of a=10 and b=20 is=30.

* Function with argument and return value:

```
#include<Stdio.h>
int addition(int, int); //Function Declaration
Void main()
{
   int a=10, b=20, Sum;
   Sum=addition(a, b); // Function call with arguments
   Printf("\n Sum of a=%d and b=%d is=%d", a, b,
                                   Sum);
}
int addition(int x, int y) // Function definition
{
   return(x+y); //return value
}
```

Calling Function?
→ main()
Called Function?
→ sum()

O/P

Sum of a=10 and b=20 is=30.

Return type * Function Prototype :

Function Name → arguments / Parameters

```
int demo (int a, int b)
{
    int Sum;
    Sum = a+b;
    return Sum;
}
```

} Function definition.

FUNCTION PARAMETERS

The calling function conveys the information to the called function using the mechanism called arguments.

Unknowingly used the arguments in the Printf() and Scanf() functions, the format string and the list of variables inside the parenthesis in these functions are arguments.

The arguments are also called Parameters

There are two methods of passing arguments / Parameters to a function.

→ Call by value
→ call by reference.

CALL BY VALUE

A way of calling function where the actual arguments are copied to formal arguments.

Memory is allocated for both actual and formal arguments.

When a function passes values to the called function for ex, numbers in the example the value of the arguments are copied to the corresponding parameters.

This ensures that the changes made to the parameters inside the function will not affect the argument (the value of variable in the calling function).

Ex: Illustrates the passing arguments using call by value.

```
#include <stdio.h>
void decrease (int i)
{
  i--;
  Printf("%d", i);
}
main()
 {
   int i=1;
   Printf("%d", i);
   decrease(i);
   Printf("%d\n", i);
 }
```


Output:

```
1 0 1
```

Program Explanation:

Here, function declaration is

not necessary because function definition Precedes function call. The original value of i (i.e, i=1) is displayed when main function begins execution.

This value is then passed to the function decrement. In the function value is decremented and displayed.

Finally, the value of i within main is again displayed, after control is transferred back to main from decrement. The output shows that alteration of value within the function doesnot affect the value of i within main function.

It is important to note that main function and decrement function use same variable name i; however they use different memory locations to store their values as they are declared in different functions.

Passing argument by value has advantages and disadvantages. On the plus side it Protects the value of the variable from alterations within the function.

On the otherhand, it doesnot allow information to be transferred back to the calling function via arguments.

Thus, call(passing) by value is a One-way transfer of information.

CALL BY REFERENCE

A way of passing the arguments by copying the reference of an arguments into the parameters.

Memory is allocated only for actual arguments. The formal arguments share the memory.

It is the second way of passing arguments to the function. In this method, the address of an argument is copied into the parameter.

Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

C only passes parameters by value. It makes use of pointers to simulate pass by references.

Ex:

```
Void main()
{
  int i=10;
  func(i);
  Printf("%d", i);
}
func(int &p)
{
  P=20;
  Printf("%d", p);
}
```

Output

20 20

RETURN TYPES

* Return Statements

The return statement causes an immediate exit from the function. That is, it causes program execution to return to the calling code. The return statement also can be used to return a value.

* Returning from a function:

A function terminates execution and returns to the caller in two ways.

1) One way of return occurs when the last statement in the function has executed, and conceptually, the function's ending curly brace (}) is encountered.

This is the default way of returning from a function.

2) Another way of return occurs by the use of return statement. The use of return statement returns value to the caller and also makes code simpler and more efficient.

A function may contain several return statements. These are used to simplify coding. The following example shows that it uses three return statements.

First returns the value of addition of two numbers.

Second returns the value of Subtraction of two numbers. According to Program logic third return Statement is never executed.

Ex: Function having multiple return Statements :

```
#include <Stdio.h>
int addsub(int, int, int) /* Function declaration */
Void main(void)
{
  int x=20, y=10;
  Printf("Result is : %d\n", addsub(x, y); /* Call with op : Add */
  Printf("Result is : %d\n", addsub(x, y, 1); /* call with op: subtract */
}
int addsub(int a, int b, int OP)
{
if (OP==0)  /* check operation */
{
  return (a+b) /* return the value of addition */
}
if(OP==1) /* check operation */
{
  return(a-b); /* return the value of Subtraction */
}
return 0;  /* Statement never executed */
}
```


Output :

Result is 30

Result is 10

* Returning values:

All functions, except those of type void, return a value. This value is specified by the return statement. Therefore, if function is not declared as void, can use it as an operand in an expression.

Let us see how valid expressions in which functions is used as an operand.

```
y = pow(a, b);
Printf("%f \n", log(val));
if(min(x, y) < 100) Printf("Smaller");
```

It is important to note that function returns a value. However, it is not compulsory to assign the returned value to some variable. Since a value is being returned.

If there is no assignment specified, the return value is simply discarded.

For Ex, Printf() returns the number of characters written. However, it is unusual to find a Program that actually checks this.

Let us see following three expressions:

```
1. S = add(a,b);
2. Printf("%d", add(a,b));
3. add(a,b);
```

In the first expression, the return value of add() is assigned to S.

In the second expression, the return is not actually assigned, but it is used by the Printf() function.

Finally, in the third expression, the return value is lost because it is neither assigned to another variable nor used as part of an expression.

* Functions of type void:

When function return type is declared as a void. It is not suppose to return any value.

This prevents their use in any expression and helps avert accidental misuse.

Ex:

```
#include <stdio.h>
Void Print_sq(int);
int main(void)
{
  int i, a[] = {2,4,5,6,7,9};
  for(i=0; i<5; i++)
  Print_sq(a[i]);
  return 0;
}
Void print_sq(int n)
{
  Printf("The square of %d is: %d\n", n, n*n);
}
```

RECURSIVE FUNCTIONS

In C, function can call itself. In this case, the function is said to be recursive.

A simple example of a recursion function is fact(), which computes the factorial of an integer. The factorial of a number n is the product of all the whole numbers between 1 and n.

Ex: 4! is 4×3×2×1 => 24.

Recursive function uses stack. A stack is a last In first out (LIFO) data structure. This means that the last data item to get stored on the stack (often called push operation) is the first data item to get out of the stack (often called Pop operation).

This is similar to that of stack is the first one to get out of it.

The recursive function must have the conditional statement, such as 'if', somewhere to force the function to return without the recursive call being executed.

Recursion Function

```
#include<stdio.h>
int fact(int);
void main()
{
  int n, answer;
  Printf("Enter the number");
  Scanf("%d",&n);
```

/* function declaration */

Non-recursion Function

```
#include<stdio.h>
int fact(int);
void main()
{
  int n, answer;
  Printf("Enter the no.:");
  SCanf("%d",&n);
```

```
answer = fact(n);
Printf("Factorial of %d is
        %d", n, answer);
}
fact(int n) /*Function
              definition*/
{
  int factorial;
  if(n==1)
  {
    return(1);
  }
  else
  {
    factorial=n*fact(n-1); /*
                    Recursion
                    call*/
    return(factorial);
  }
}
```

Output:

```
Enter the number: 5
Factorial of 5 is 120.
```

```
answer = fact(n);
Printf("Factorial of %d is %d", n,
                        answer);
}
fact(int n)
{
  int i, factorial=1;
  for(i=0; i<=n; i++)
  {
    factorial=factorial*i;
  }
  return(factorial);
}
```

Output:

```
Enter the number: 5
Factorial of 5 is 120.
```


SCOPE AND LIFETIME OF VARIABLE

Two key concepts that control this behavior are the Scope and Lifetime of Variables in C.

"Scope" defines where a variable can be accessed in your code, while "Lifetime" determines how long the variables exists in memory.

* Scope of variable in C:

Scope of variables in C language refers to the region or part of the code where

a variable can be accessed or used. It defines the visibility of a variable within different parts of a program.

For Ex:

If a variable is declared inside a function it can only be used within that function, and this is called local scope.

If a variable is declared outside of all functions, it can be accessed from anywhere in the program, and this is known as global scope.

* Types of scope variables in c:

1) Local Scope

A variable has local scope when it is declared inside a function or block (within {} braces).

This means that the variable is accessible only within that function or block and is not visible to other parts of the program.

* Function level scope: variables declared inside a function are accessible only within that function.

* Block-Level scope: variables declared inside a block (if stmt or loop) are accessible only within that block.

Ex:

```
Void exfunction(){
    int localvar=10; //Fn level local variable
    if (localvar==10){
        int blockvar=20; // Block-level local variable
```

```
    Printf("blockvar : %d\n", blockvar);
}
// Printf("%d", blockvar); // Error: blockvar is not accessible here
Printf("localvar : %d\n", localvar);
}.
```

2) Global scope :

A variable has global scope when it is declared outside of all functions. This means the variable is accessible from any function in the Program.

Ex:

```
int globalvar = 100; // global variable with global scope
Void function1() {
    Printf("Global var in fn1: %d\n", globalvar);
}
Void function2() {
    globalvar = 200; // modifying global variable
    Printf("Global var in fn2: %d\n", globalvar);
}
int main() {
    function1(); // Accesses globalvar
    function2(); // Modifies and accesses globalvar
    return 0;
}
```


3) Function scope :

Function parameters have function scope meaning they are only accessible within the body of that function.

Ex:

```
Void multiply (int a, int b) {
{ int result = a*b;
```

```
    printf("Result: %d\n", result);
}
int main() {
    multiply(5,3); // Passing values to fn parameters
    return 0;
}
```

Lifetime of variables in C:

Variable lifetime in C refers to the Period during which a variable exists in memory and retains its value.

It defines how long the variables allocated memory is valid and available for use in the Program. The lifetime of a variable depends on where and how the variable is declared.

Types:

1) Automatic (local) variables:

It automatically created when a function is called and destroyed when the function returns. These variables are created using the auto keyword.

Automatic variables only exist during the fun's execution, and their values do not Persist between function calls.

Characteristics:

- → Created when the function starts
- → Destroyed when the function ends
- → They do not retain their values between function calls.
- → Stored in the STACK.

Ex:

```
#include <stdio.h>
Void exFn( ) {
    int localvar = 10 ; // Automatic var
    Printf(" Local variable : %.d\n", localvar) ;
} // localvar is destroyed here
int main() {
    exFn() ; // localvar is created and destroyed
                during the fn call
    return 0;
}
```

2. Static variables:

Static variables are variables that retain their value between multiple fn calls. If a static var is declared inside a fn, it is initialized only once, and its value persists across fn calls.

Characteristics:

- → Created when the Program Starts
- → Retain their value across multiple fn call.
- → Destroyed when the program ends
- → Stored in the data segment of mly.

Ex:

```
#include <Stdio.h>
Void CountCalls() {
  Static int Count=0 ; // Static variable initialized
                          once
  Count++;
  Printf(" Fn called %.d times\n", Count) ;
}
int main() {
    CountCalls() ; // Fn Called 1st time
    CountCalls() ; //  "    "  2nd time
    CountCalls() ; //  "    "  3rd time
    return 0; }
```

3. Global variables:

It declared outside all functions, making them accessible from any fn in the pgm. They have a lifetime that spans the entire duration of the program.

Characteristics:

→ Created when the pgm starts.
→ Accessible from any function
→ Destroyed when the program ends
→ Stored in the data segment.

Ex:

```
#include<stdio.h>
int globalvar=100 // global variable
Void fn1(){
    Printf("Global var in fn1:%d\n", globalvar);
}
Void fn2(){
    globalvar=20; //modify global var
    Printf("Global var in fn2:%d\n", globalvar);
}
int main(){
    function1(); //Access globalvar
    function2(); //Modify and access globalvar
    return 0;
}
```


4) Dynamic variables:

They are created and destroyed ligetime at runtime using dynamic memory allocation functions (like malloc(), calloc() and free().

HEADER FILES AND MODULAR PROGRAMMING IN C:

Header files and modular programming are fundamental concepts in C that facilitate organized, reusable, and maintainable code.

* Modular Programming in C:

Modular Programming involves breaking down a large program into smaller, independent units called modules. Each module encapsulates a specific set of functionalities and data, promoting code organization, readability and reusability in C.

Modules are typically implemented using separates source code files (.c files) and their corresponding header files (.h files).

* Header files in C:

Header files serve as the interface for modules. They contain declaration of functions, variables, macros and data structures that are intended to be shared and accessed by other parts of the program.

By including a header file in a source file the compiler gains knowledge of the declared entities, enabling correct compilation and linking.

* Key aspects of header files and Modular Programming:

1) Separation of Interface and Implementation

Header files defines the public interface of a module, while the corresponding .c file contains the actual implementation details.

This separation promotes information hiding and encapsulation.

2. Code Reusability:

Function and data structures declared in header files can be reused across multiple source files and even in different projects, reducing redundancy and development time.

3) Improved readability and maintainability:

Breaking down a large program into smaller, self contained modules makes the code easier to understand, navigate and debug.

4) Collaboration:

Modular programming enables multiple developers to work on different modules simultaneously, fostering parallel development and improving team efficiency.

Ex:

```
// math_operations.h (Header file)
#ifndef MATH_OPERATIONS_H
#define MATH_OPERATIONS_H
int add(int a, int b);
int Subtract(int a, int b);
#endif
// math_operations.c (Implementation file)
#include "math_operations.h"
int add(int a, int b){
    return a+b;
}
```

```
int subtract(int a, int b) {
    return a-b;
}
// main.c (client file).
#include <stdio.h>
#include "math_operations.h"
int main() {
    int sum = add(5,3);
    int difference = subtract(10,4);
    Printf("Sum : %d\n", sum);
    Printf("Difference : %d\n", difference);
    return 0;
}
```

In this example, math_operations.h declares the add and subtract functions, while math_operations.c provides their implementations. main.c includes math_operations.h to access these functions. demonstrating modular programming with header files.

UNIT-IV STRINGS AND POINTERS.

One-dimensional and Multi-dimensional Arrays, Array operations and traversals, String Handling: String declaration, Input/output, String library functions, Pointer arithmetic, Pointers and Arrays, Pointers to function, Dynamic memory Allocation.

INTRODUCTION TO ARRAYS:

There are different types of data structures such as array, stack, queue, linklist, structure, tree, file etc.

Array can be defined as a collection of variables of the same type that are referred through a common name. A specific element in an array is accessed by an index.

In C all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

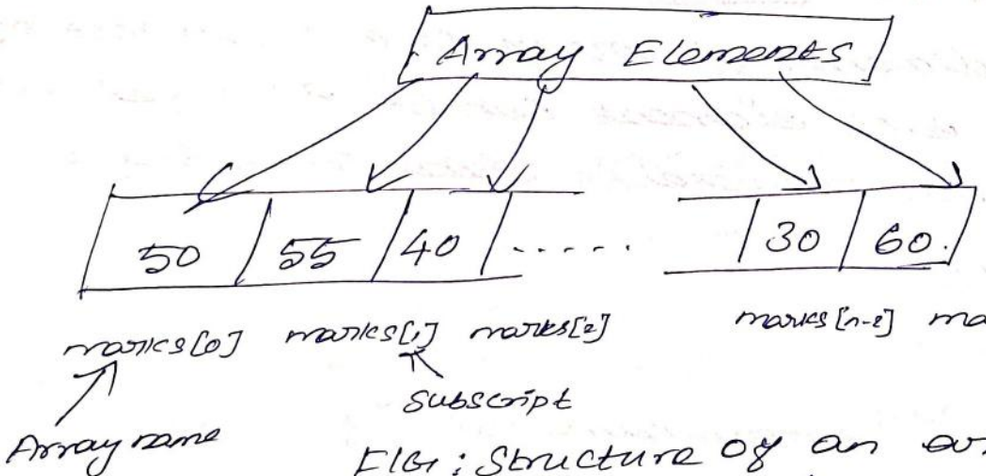


Fig: Structure of an array named mark having n elements.

* Array Declaration and Definition:

Like other variables, an array needs to be declared and defined at the beginning so that the computer will know what is the data type of

~~int subtract(int a, int~~

array elements and how large the array size. i.e) the maximum no. of elements that array can hold.

The general form of declaring an array is,

Syntax:

```
data_type Variable_name [size];
```

Ex: float marks[50];

* Array Initialization:

Array declaration only ~~de~~reserves memory for the elements in the array. No values will be stored. The values of array elements are stored when array is defined. This is called array initialization.

Syntax:

type array_name [size] = { value 0, value 1, ... };

Ex:

```
int numbers[5] = {10, 11, 12, 13, 14};
```

➤ ONE_DIMENSIONAL ARRAYS:

A one-dimensional array is a linear list of elements, where each element can be accessed using a single index. It represents data in a single row or column.

Syntax:

```
data_type array_name [size];
```

Ex:

```
int numbers[5];
numbers[0]=10;
Printf("%d", numbers[2]);
```

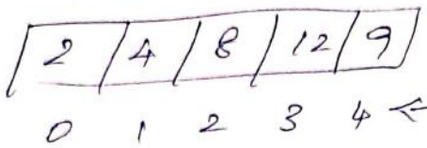
→ Accessing Elements:

Ex:

int grades[5] = {85, 90, 76, 92, 88};

Elements are accessed using a single subscript, with the index starting from 0. For Ex, grades[0] accesses the 1st element 85 and grades[4] accesses the last element 88.

→ Memory Representation: Elements are stored sequentially in a single contiguous block of mly.



MULTI-DIMENSIONAL ARRAYS:

It is essentially an "array of arrays" and is used to stored data in a complex, grid like format, such as tables or matrices.

The most common is the two dimensional (2D) array.

Syntax:

data-type array-name [size1] [size2]...[sizeN];

Ex: Declaration and Initialization:

In the definition of the array if all values are not specified, the specified values are assigned to corresponding array element and remaining array elements are initialized to zero value.

```
// A 2x3 matrix (2 rows, 3 columns)
int matrix[2][3] = {
        {1,2,3}, // Row 0
    };  {4,5,6}// Row 1
```

Elements require multiple indices (subscripts) to specify their position, e.g) row and column.

For Ex, matrix[1][0] accesses the element at the second row and first column.

Even though they have multiple dimensions logically they are still stored in linear, contiguous memory locations in row-major order.

Ex: Write a Program to find the transpose for a matrix.

```
#include<Stdio.h>
int main()
{
 int n, m, A[15][15], i, j;
 Printf("Enter the value of N and M:");
 Scanf("%d%d", &m, &m);
 Printf("Enter rows and columns of matrix\n");
 Printf("Row wise\n\n");
 for(i=0; i<n; i++)
    for(j=0; j<m; j++)
       Scanf("%d", &A[i][j]);
 Printf("Given matrix\n");
 for(i=0; i<m; i++).
 {
   for(j=0; j<n; j++)
    {
     Printf("%d\t", A[i][j]);
    }
    Printf("\n");
 }
 Printf("Transpose of given matrix\n");
 for(i=0; i<m; i++)
  {for(j=0; j<n; j++)
    {Printf("%d\t", A[i][j]);
    }Printf("\n"); }return 0;
}
```

O/P

```
Enter the value of N and M : 2 3
Enter rows and columns of matrix
Row wise
1 2 3
3 4 5
Given matrix
 1  2  3
 3  4  5
Transpose of Given matrix
  1  3
  2  4
  3  5
```

➢ ARRAY OPERATIONS IN C :

1) Array Declaration and Initialization :

An array needs to be declared and defined at the beginning.

An array must be declared and defined before it can be used.

Array declaration only reserves memory for the elements in the array. No values will be stored. The values of array elements are stored when array is defined it called array initialization.

Ex:

```
int arr[5] ; // Declares an int array of size 5
int initialized_arr[] = {10,20,30}; // Declares & Initialize array
```

2) Accessing Elements :

Elements are accessed using their index, which starts from 0.

Ex:

```
int value = arr[2]; // Accesses the 3rd element
arr[0] = 5; // Assigns a val to 1st element.
```

3) Finding Array Length:

The size of operator can be used to determine the no. of elements.

```
int length = sizeof(arr) / sizeof(arr[0]);
```

4) Copying Arrays: Elements must be copied one by one, typically using a loop.

Ex:

```
int source_arr[] = {1, 2, 3};
int dest_arr[3];
for (int i=0; i<3; i++) {
    dest_arr[i] = source_arr[i];
}
```


5) Searching: Iterating through the array to find a specific element.

```
int key = 20;
for (int i=0; i<length; i++) {
    if (arr[i] == key) {
        // element found at index i
    }
}
```

* Sorting:

Implementing sorting algorithms.

ex) bubble sort, selection sort, quick sort) to arrange elements in a specific order.

* Insertion and deletion:

These operations often involve shifting elements to maintain contiguity as C arrays have fixed sizes.

Array Traversal in C:

Traversal is the process of visiting each element in an array, typically to perform an operation on it or to display its value.

The most common method for traversal is using a loop.

Ex:

```
#include<stdio.h>
int main()
{
int numbers[] = {10,20,30,40,50};
int size = sizeof(numbers)/sizeof(numbers[0]);
Printf("Array elements:");
for(int i=0; i<size; i++){
    Printf("%d", numbers[i]); // Accessing and Printing each element
}
Printf("\n");
return 0;
}
```


In this example, the for loop iterates from i=0 to size-1, accessing each element numbers[i] sequentially. This allows for processing or displaying every element within the array.

STRING HANDLING:

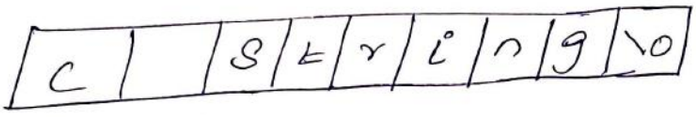
STRING DECLARATION:

In C Programming a String is a sequence of characters terminated with a null character \0.

For ex:

```
Char C[] = "C String";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.



* How to declare a String?

· The following declaration stores a string of size 5 characters.

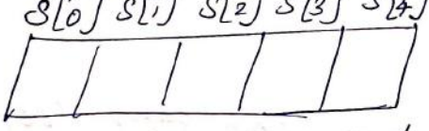

```
Char Str[6];
```

The following declaration creates a string variable of a specific size at the time of program execution.

```
Char *Str = (char*) malloc(15);
```

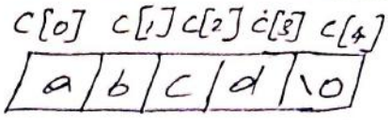
Here's how to declare Strings:

```
Char S[5];
```



Here declared a string of 5 characters.

```
Char C[] = "abcd";
Char C[50] = "abcd";
Char C[] = {"a",'b','c','d','\0'};
Char c[5] = {'a','b','c','d','\0'};
```



Let's take another example:

Char c[5] = "abcde";

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters.

* Assigning values to strings:

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For Ex,

```
char c[100];
c = "C Programming";
```

String value is assigned using the following two methods.

1) At the time of declaration (initialization)
2) After declaration.

=> Read string from the user:

Use the scanf() function to read a string. The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc).

Ex of assigning string value:

```
int main()
{
 char str1[6] = "Hello";
 Char str2[] = "Hello!";
 char name1[] = {'s','m','a','r','t'};
 char name2[6] = {'s','m','a','r','t'};
```

STRING HANDLING:

```
char title[20];
*title = "btech smart class";
return 0;
}
```

STRING INPUT/OUTPUT (OR) READING STRING VALUE:

* Reading string value from user in C:

Read a string value from the user during the program execution. We use the following two methods.

1) Using Scanf() method - Reads single word
2) Using gets() method - Reads a line of text.

Using Scanf() method we can read only one word of string. We use %s to represent string in Scanf() and Printf() methods.

Ex1: Scanf() to read a string.

```
#include <Stdio.h>
int main()
{
  char name[20];
  Printf("Enter name:");
  Scanf("%s", name);
  Printf("Your Name %s", name);
  return 0;
}
```

O/P

Enter name: Dennis Ritchie
Your name is Dennis.

* gets() and puts()

Functions gets() and puts() are two String functions to taking string input from the user and display it respectively.

```
#include <stdio.h>
int main()
{
  char name[30];
  Printf("Enter name:");
  gets(name); //Functions to read string from user.
  Printf("Name:");
  Puts(name); //Function to display string
  return 0;
}
```

When we want to read multiple words or a line of text, we use a pre-defined method gets().

The gets() method terminates the reading of text with Enter character.

* How to read a line of text?

Use the fgets() function to read a line of string. And use puts() to display the string.

Ex: fgets() and puts()

```
#include <stdio.h>
int main()
{
  char name[30];
  Printf("Enter name:");
  fgets(name, sizeof(name), stdin); //read string
  Printf("Name:");
  Puts(name); //display string
  return 0;
}
```

Output:

Enter name: C Program
Name: C Program.

Here, we used fgets() function to read a string from the user.

fgets(Name, sizeof(name), stdin); //read string.

The sizeof(name) result to 30. Hence, can take a maximum of 30 characters as i/p which is the size of the name string.

To print the string, we used puts(name);

Note: The gets() function can also be to take i/p from the user. However, it is removed from the C standard.

It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

* Passing Strings to Functions:

Strings can be passed to a function in a similar way as arrays.

Ex: Passing string to a Function:

```
#include <stdio.h>
void displayString(char str[]);
int main()
{
  char str[50];
  printf("Enter string: ");
  fgets(str, sizeof(str), stdin);
  displayString(str); // Passing string to a function.
  return 0;
}
```

```
Void displayString (char Str[])
{
 Printf ("String output :");
 Puts(Str);
}
```

➤ STRING LIBRARY FUNCTIONS : / String manipulation in C Programming using Library Functions :

Most of the time string manipulation can be done manually, but this makes programming complex and large.

To solve this, C supports a large number of string handling functions in the standard library "string.h".

Few commonly used string handling functions are discussed below :

Function	Work of Function
Strlen()	Computes string's length
Strcpy()	Copies a string to another
Strcat()	Concatenates (joins) two strings
Strcmp()	Compares two strings
Strlwr()	Converts string to lowercase
Strupr()	Converts string to uppercase

String handling functions are defined under "string.h" header file.

#include<string.h>

Note : To include the code below to run string handling functions.

1) C Strcat()

In C Programming, the Strcat() function concatenates (joins) two strings.

The function definition of Strcat() is:

```
Char *Strcat(char *destination, const char *source);
```

It is defined in the string.h header file.

Strcat() arguments:

The Strcat() function takes two arguments:

destination : destination string
Source : Source string.

The Strcat() function concatenates the destination string and the Source string, and the result is stored in the destination string.

Ex: C Strcat() function

```
#include <stdio.h>
#include <string.h>
int main() {
   Char Str1[100] = "This is ", Str2[] = "C Pgm";
// Concatenates Str1 and Str2
// the resultant string is stored in Str1.
 Strcat(Str1,Str2);
 Puts(Str1);
 Puts(Str2);
 return 0;
}
```

O/P

```
This is C Pgm
C Pgm.
```

3. C Strcpy():

The function prototype of strcpy() is:

Char * Strcpy(char * destination, Const char * source)

The Strcpy() function copies the string pointed by source (including the null character) to the destination.

The Strcpy() function also returns the Copied string.

The Strcpy() function is defined in the String.h header file.

Ex:

```
#include<Stdio.h>
#include<String.h>
int main() {
   Char Str1[20] = "C Pgm";
   Char Str2[20];
//copying String Str1 to Str2
   Strcpy(Str2, Str1);
   PutS(Str2); //C Pgm
   return 0;
}
```

O/P

C Pgm

4) C Strlen()

The Strlen() function calculates the length of a given String.

The Strlen() function takes a String as an argument and returns its length. The returned value is of type Size_t (the unsigned integer type).

It is defined in the <String.h> header file.

Ex:-

```
#Include<Stdio.h>
#include<String.h>
int main()
{
  Char a[20]="pgm";
  Char b[20]={"p",'g','m','\0'};
  //using the %zu format specifier to print size_t
  Printf("Length of String a=%zu\n",Strlen(a));
  Printf("Length of String b=%zu\n",Strlen(b));
  return 0;
}
```

O/p

Length of String a = 3
Length of String b = 3

5. C Strcmp()

TO Compare two Strings using the Strcmp() function.

The Strcmp() Compares two Strings Character by Character. If the Strings are equal, the function returns 0.

* C Strcmp() Protype:

The function prototype of Strcmp() is;

```
int Strcmp(Const Char* Str1, Const Char* Str2);
```

Strcmp() Parameters:

This function takes two parameters:

Str2, Str1 - a String

* Return value from Strcmp()

Return value	Remarks
0	if Strings are equal
non-zero	if Strings are not equal

The Strcmp() function is defined in the String.h header file.

Ex:

```
#include<Stdio.h>
#include<string.h>
int main(){
char Str1[]="abcd", Str2[]="abCd", Str3="abcd";
int result;
//Comparing strings Str1 and Str2
result = Strcmp(Str1, Str2);
Printf("Strcmp(Str1, Str2) = %d\n", result);
//Comparing strings Str1 and str3;
Printf("Strcmp(str1, Str3) = %d\n", result);
return 0;
}
```

Output:

```
Strcmp(Str1, Str2) = 1
Strcmp(Str1, Str3) = 0
```


In the program,

Strings Str1 and Str2 are not equal. Hence the result is a non-zero integer.

Strings Str1 and Str3 are equal. Hence the result is 0.

POINTERS:

A Pointer is a variable that represents the location of a data item, such as a variable or an array element.

It is a variable that holds a memory address. This address is the location of another variable or an array element in memory.

Ex: If one variable contains the address of another variable, the 1st variable is said to point to the second.

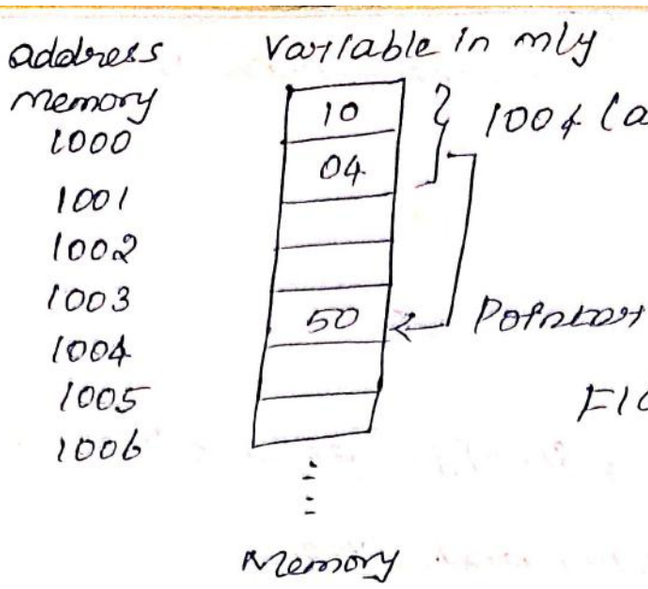


FIG: One variable (pointer) Points to another

➤ Pointer Declaration:

If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an * and the variable name.

The general form for declaring a pointer variable is,


```
data_type *var_name;
```

For Ex, Consider the variable declaration:

```
int *ptr;
```

The '*' informs the compiler that want a Pointer variable, i.e) to set aside number of bytes required to store an address in memory.

The int says that intend to use our Pointer variable to store the address of an integer.

* Initializing Pointer variable:

The Process of assigning the address of a variable to a pointer variable is known as Initialization of pointer variable.

Store the address of our integer variable x in ptr (pointer variable).

`ptr = &k`

The & operator retrieves the address of k, even though k is on the right hand side of the assignment operator '=' and copies that to the contents of our pointer ptr.

* Accessing variable through pointer:

To store address in the pointer variable. And also store the value at the address specified by the pointer variable.

```
int k, *ptr;
ptr = &k;
*ptr = 7;
```

Similarly we can write,

```
Printf("%d\n", *ptr);
```


Ex:

```
#include<stdio.h>
main()
{
  int balance;
  int *address;
  int value;
  balance = 5000;
  address = &balance;
  value = *address;
  Printf("Balance is : %d\n", value);
}
```

Output:

Balance is 5000

POINTER ARITHMETIC:

Let us consider why need to identify the type of variable that a pointer points to, as in:

```
int *ptr;
```

Once ptr points to something, if we write:

```
*ptr = 2;
```

The compiler will know how many bytes to copy into that memory location pointed to by ptr.

If ptr was declared as pointing to an integer, 4 bytes would be copied.

Similarly for floats and doubles the appropriate number will be copied. But defining the type that pointer points to allow a number of other interesting ways a compiler can interpret code.

For Ex: Consider a block in memory consisting of ten integers in a row. That is 40 bytes of memory are set aside to hold 10 integers.

Now, Point our integer pointer, ptr at the first of these integers. Let us assume that integer is located at memory location 1000 (decimal).

```
ptr+1;
```

Because the compiler knows this is a Pointer (i.e) its value is an address) and that it Points to an integer, it adds 4 to ptr instead of 1, So the pointer "points to" the next integer, at mly location 1004.

Similarly, if the ptr was declared as a Pointer to a short, it would add 2 to it instead of

The same goes for other data type such as floats, doubles. or even user defined data type such as structures.

This is obviously not the same kind of addition that normally think of. In c, it is overloaded to as addition using pointer arithmetic.

Similarly, since ++ptr and ptr++ are both equivalent to ptr+1, incrementing a pointer using the unary ++ operator, either Pre- or Post-, increments the address it stores by the amount sizeof(type) where type is the type of the object pointed to. (i.e. 4 for an integer.)

Ex:

```
#include<stdio.h>
main(){
   int *p, num;
   p=&num;
   *p=100;
   Printf("%d\n",num);
   (*p)++;
   Printf("%d\n",num);
   (*p)--;
   Printf("%d\n",num);
}
```

Output:

```
100
101
100
```

➤ POINTERS AND ARRAYS:

Array elements are located continuously in memory and array name is really pointer to the first element in the array.

This brings up an interesting relationship

between arrays and pointers. Consider the following:

```
int array[] = {1, 20, 30, 40};
```

Here the array containing 8 integers. To refer each of these integers by means of a subscript to array, i.e) using array[0] through array[7]. But could alternatively access them via a pointer as follows,

```
int *ptr;
ptr = &array[0]; /* Point our pointer at the 1st
                    integer in our array */
```

And then Print out our array either using the array notation or by dereferencing our pointer.

Ex:


```
#Include<Stdio.h>
int array[] = {1, 30, 40, 35, 45};
int *ptr;
Void main (Void)
{
  Int i;
  Ptr = &array[0]; /* Point our pointer to the
                      1st element of the array */
  Printf("\n");
  for(i=0; i<5; i++)
  {
    Printf("array("array[%d] = %d", i, array[i]);
                                   /* Line A */
    Printf("\tptr+%d = %d\n", i, *(ptr+i));
                                   /* Line B */
```

O/P

```
array[0] = 1     ptr+0 = 1
array[1] = 30    ptr+1 = 30
array[2] = 40    ptr+2 = 40
```

array[3] = 35 ptr+3 = 35
array[4] = 45 ptr+4 = 45

It is important to note that printf() statements in Line A and Line B prints the same values.

Also observe how to dereference our pointer in line B, i.e, we first added i to it and then dereferenced the new pointer.

Change line B to read:

printf("ptr+%d = %d\n", i, *ptr++);

and run it again to get the same output.

In C, the standard states that wherever we might use &var_name[0] can replace that with var_name, in our code.

```
ptr = &array[0];
```


and also,

```
ptr = array;
```

to achieve the same result. However can't write,

```
array = ptr; /* wrong */
```

* Pointers and Multi-dimensional Arrays:

In two dimensional array, array elements are stored row by row. When pass a 2D array to a function must specify the number of columns - the no. of rows is irrelevant.

This is because C needs to know how many columns in order that it can jump from row to row in memory.

Consider int a[5][35] to be passed in a function:

```
func(int a[][35])
{....
}
```

Or even:

```
func(int (*a)[35])
{....
}
```

Here, need Parantheses (*a) since [] have a higher precedence than *. Thus

Now look at the difference between Pointers and arrays. Strings a

```
Char *name[10];
Char Aname[10][20];
```


POINTERS TO FUNCTIONS

C arguments are passed to functions by value, i.e) Only the values of argument expressions are passed to the called function.

Some Programming languages allow arguments passed by reference, which allows the called function to make changes in arguments objects.

C allows only call by value, not call by reference, however, if a called function is to change the value of an object defined in the calling function, it can pass a value which is

a pointer to the object, i.e) address of the object. The called function can then dereference the pointer to access the object indirectly using its address.

However, by indirect access, i.e) pass by reference using pointers a "called function" can effectively return several values.

Only one value is actualy returned as the value of the function, all other values may be indirectly stored in objects in the calling function.

This use of pointer variables is one of the most common in C.

The ability to pass pointers to function is very useful. If we want to write a pgm that takes a number and adds five to it, we write something like following:

Ex: Passing by value:

```
#include <stdio.h>
Void AddFive(int Number)
{
Number = Number+5;
}
void main()
{
Int nMyNumber = 18;
Printf("My original number is %d\n", nMyNumber);
AddFive(nMyNumber);
Printf("My new number is %d\n", nMyNumber);
}
```

O/P

My original number is 18
My new number is 18.

However, the pblm with this is that the Number referred to in AddFive is a copy of the variable nMynumber passed to the function, not the variable itself.

Therefore; the line 'Number = Number + 5" adds five to the copy of the variable, in main() unaffected. This is because the function argument is passed by value.

* Pointers to Functions:

A Pointer to a function Points to the address of the executable code of the function. Use pointers to call functions and to pass functions as arguments to other functions.

Can obtain the address of a function by using the fn name without any parentheses or arguments.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator() has a higher Precedence than the dereference operator *.

Without them the Compiler interprets the statement as a function that returns a pointer to a specified return type.

For Ex,

```
int *f(int a); /* fn f returning an int* */
int (*g)(int a); /* pointer g to a function
                    returning an int */
```

In the first declaration, f is interpreted as a function that takes an int as argument, and returns a pointer to an int.

In the second declaration, g is interpreted as a pointer to a function that takes an int argument and that returns an int.

Ex: Illustrating Pointers to functions

```
Void young(int);
Void old(int);
int main(void)
{
 Void (*fp)(int);
 int age;
 Printf("How old are you?");
 Scanf("%d", &age);
 fp = (age > 30) ? old : young;
 /* If age > 30 fp is pointer to old function;
    Otherwise it is pointer to young function */
 fp(age);
 return 0;
}
Void young(int n)
{
 Printf("Being only %d, you sure young, \n", n);
}
Void old(int m)
{
 Printf("Being already %d, you sure are old. \n", m);
}
```

DYNAMIC MEMORY ALLOCATION

Sometimes, when it is convenient to allocate memory at run time using malloc(), Calloc() or other allocation functions.

This approach allows postponing the decision on the size of the memory block needed to store an array, for ex, until run time or it allows using a part of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other use.

Four memory management functions are used with dynamic memory.

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

To use these functions we have to include Standard library header file, stdlib.h in our program.

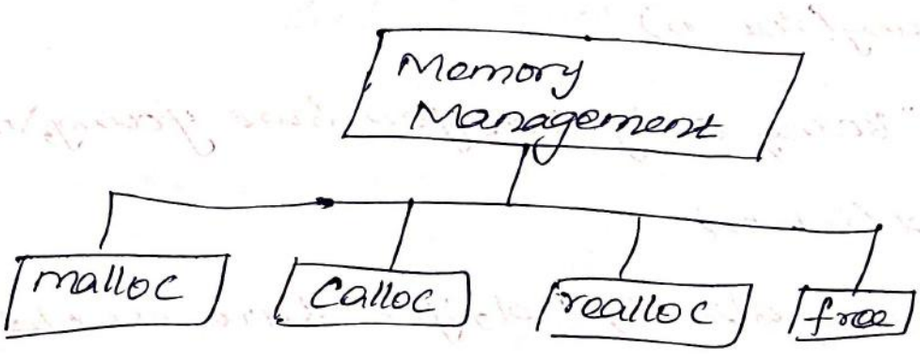


fig: Memory management functions.

1. malloc()

The malloc() function allocates a block of memory that contains the number of bytes specified in this parameter.

It returns a void pointer to the first byte of the allocated memory.

The prototype for malloc function is,

```
void * malloc (size_t size);
```

The type 'size_t' is defined in <stdlib.h> header file. The type is usually an unsigned integer, and by the standard it is guaranteed to be large enough to hold the maximum address of the computer.

Use the sizeof operator to specify no. of bytes to be allocated in malloc function.

```
char *p;
p=malloc(1000); /* Allocates 1000 bytes */
```

After the assignment P, points to the first of 1000 bytes of free memory. In the above ex, no type cast is used to assign the return value of malloc() to P.

A void * pointer is automatically converted to the type of pointer on the left side of an assignment operator.

This example allocates space for 50 integers,

```
int *p;
p=malloc(50 * sizeof(int));
```

Memory's allocated by C's dynamic memory allocation function is obtained from the heap. The heap is free memory that is not used by the Program, the OS or any other Pgm running in the computer. Since the size of the heap is not infinite, every time when we call malloc function may not get the allocation of memory.

Hence, Whenever allocate the memory, it is necessary to check the value returned by malloc() to make sure that is not null before using the Pointer.

Using a null pointer will almost certainly Crash the program. The proper way to allocate memory and test for a valid pointer is,

```
P=malloc(1000);
if (!P)
{
Printf("Out of memory.\n");
exit(1);
}
```

Here, we are free to use other sort of error handler in place of the call to exit().

2. Free:

The free() function is the opposite of the malloc() in that it returns the previously allocated memory to the system.

Once the memory has been freed, it may be reused by a subsequent call to malloc(). The Prototype for free() is,

```
Void free(void *p);
```

Here, p is a pointer to memory that was Previously allocated using malloc().

3. Calloc() and realloc().

Calloc() is for allocating the required amount of memory. The syntax is,

```
Void *calloc(size_t nitems, size_t size)
```

This function returns the pointer to the allocated memory. The calloc function is just similar to malloc but it initializes the allocated memory to zero and malloc doesnot.

The realloc() function modifies allocated memory size by malloc() and calloc() functions to new size. The syntax is,

```
Void *realloc(void *Ptr, size_t size)
```

If sufficient memory exists then to expand the memory block, realloc() is used.

*Dynamically allocated Arrays:

The Program which allocates the space for a string (character array) dynamically, requests user for the input of string and then prints the string in the reverse order.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Void main(void)
{
  char *s;
  int i;
  s=malloc(80);
  /* Request for memory allocation */
  if(!s)
  {
    Printf("Memory allocation failed.\n");
    exit(1);
          /* if request is failed exit */
  }
  Printf("Enter the String:");
  gets(s); /* Read the string from keyboard */
  for(i=strlen(s)-1; i>=0; i--)
  {
    putchar(s[i]); /* Print single character */
  }
  free(s); /* Return the previously allocated mlg */
}
```


O/P

Enter the String : This is a String.

Hai Yetavore.

O/P (if allocation unsuccessful)

Mly allocation failed.

UNIT-5 STRUCTURES AND UNIONS

Defining and using Structures, Array of Structures, Pointers to Structures, Unions and their uses, Enumerations..

STRUCTURES:

In C Structure is a user-defined data type that can be used to group items of possibly different types into a single type.

The struct keyword is used to define a structure. The items in the structure are called its members and they can be of any valid data type.

Used to store related fields of different data types.

Capable of storing heterogeneous data.

DEFINING A STRUCTURE:

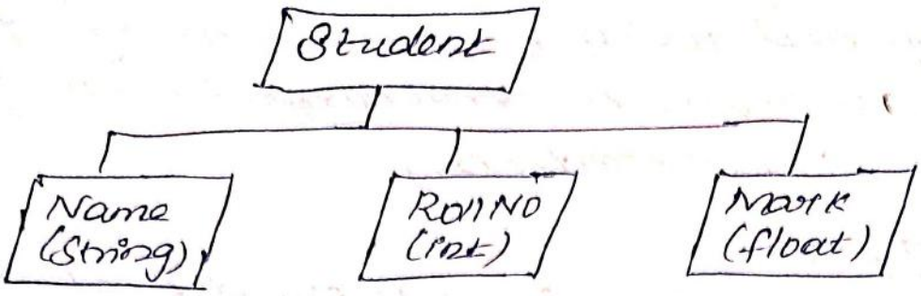
In C, a structure is a derived data type consisting of a collection of member elements and their data types.

Thus a variable of a structure type is the name of a group of one or more members which may or may not be of the same data type.

In programming terminology, a structure data type is referred to as a record data type and the members are called fields.

For defining the structures, must specify the names and types of each of the fields of the structure and declare variables of

that type.



Syntax :

```
Struct Student {
    Char name[20];
    Int roll no;
    float marks;
};
```

This defines a new data type Student. It contains three members.

20 → Element character array, name [20].
Integer quantity roll no,
float quantity marks.

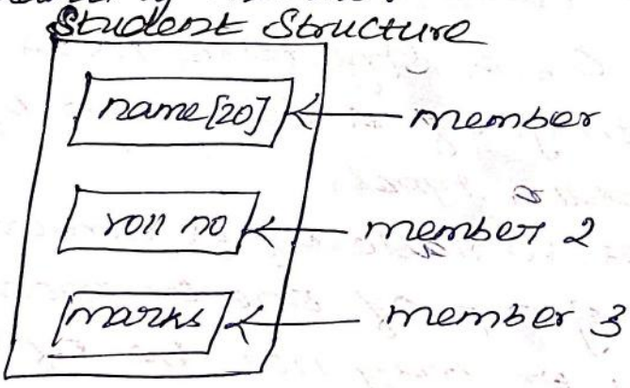


FIG : Structure Student.

In general terms, the composition of Structure may be defined as,

```
Struct tag {
        member1;
        member2;
        =
        =
        membern;
};
```

Where Struct is a keyword.

Tag is a name that identifies structure of this type.

The left brace indicates the beginning of a structure and right brace indicates the end of the structure.

The members of the structure are specified within the pair of curly braces separated by semicolon.

Each member of structure may belong to a diffgerent type of data. The individual members can be ordinary variables, arrays, pointers or other structures.

```
Struct Student S1, S2, S3 ;
```

The S1, S2, S3 are variables of data type Student.

* Initialization of a structure:

Initialization of array elements, structure members can be initialized within the variable declaration part. In such initialization, the values of members must appear in the orders in which they are defined in the structure.

```
Struct Student S1 = { "Ponko", "C Pgm", 21, 2025};
Static Struct Student S2 = { "Hathi", "Java" 25, 2025};
```

* Accessing and Processing a Structure:

The members of a structure can be accessed and processed as separate entities. A structure members can be accessed by using dot (.), also called period operator. The syntax is

Variable.member

Where variable refers to the name of a structure type variable, and member refers to the name of a member within the structure.

Ex:

```
#include <stdio.h>
void main(void)
{
 Struct demo
  {
    int i; int j;
  }s;
  int i;
  i=10;
  s.i=100;
  s.j=101;
  Printf("%d %d %d", i, s.i, s.j);
}
```

Output:

```
10  100  101
```

➢ ARRAY OF STRUCTURES:

When working with a group of entities and their attributes, need to create array of structures.

For Ex, if want to work with students data base we can create a structure student, in which can store the information about the address, age, marks obtained by the student and so on.

By creating an array of structure student can store information of student structure and quickly and easily work with the student's information.

Once the Structure Student is defined, can create the array just as we would create array of integers.

```
Student StudentA[40];
```

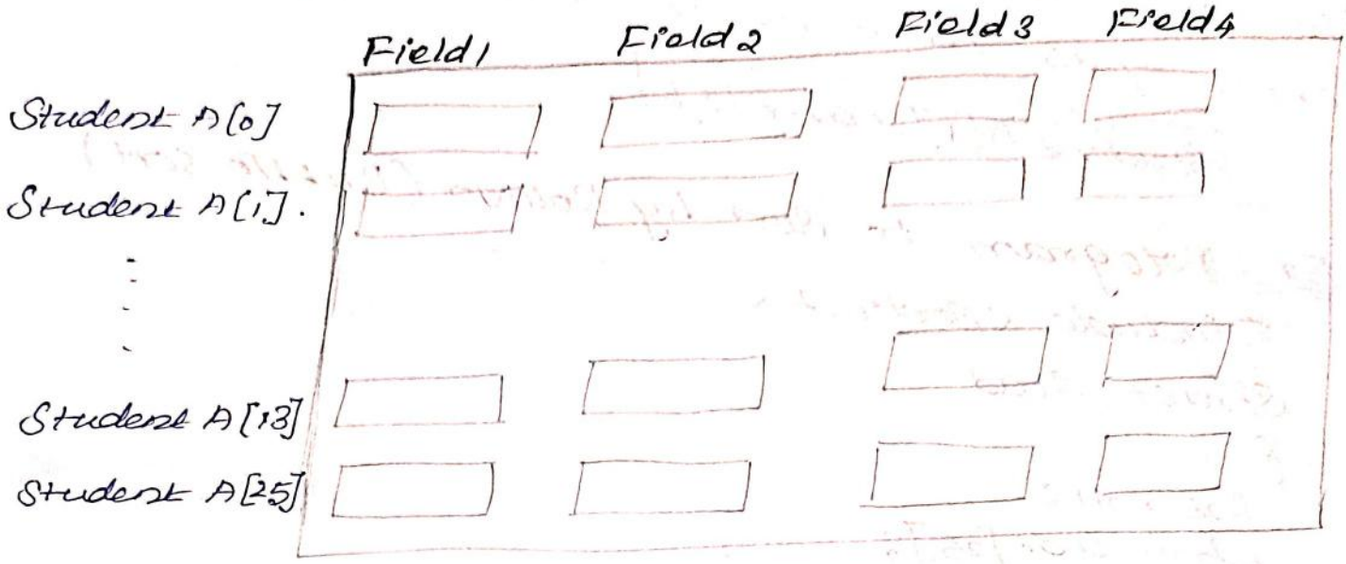


Fig: Array of Structure.

It is important to note that Student array contains two more arrays: name and course.

However, it is not a multidimensional array. To be a multidimensional array, each level ~~of the~~ must have the same data type.

In this case, data types are different: The StudentA is Student, while name and course is Character.

To access the data for one Student, we need to refer only to the Structure name with an index as,

```
StudentA[i]
```

However, to access an individual element in one of the Student's array such as the age for the 4th Student, need to specify the field

name as given below,

`Student[3].age`

To access specific letter in the character array of Student structure, we have to specify the field with an index. To access the first letter of the name of the 5th student we write,

`Student[4].name[0].`

Ex: Program to sort by RollNo (Bubble sort).

```
#include <stdio.h>
Struct Stud
{
  int roll;
  Char code[25];
  float marks;
};

main()
{ Struct Stud class[100], t;
int i, k, n;
Scanf("Enter the no. of Students", &n);
for(k=0; k<n; k++)
Scanf("Enter roll no, dept code and marks"
   %d %s %f", &class[k].roll, &Class[k].code, &class[k].marks);
for(j=0; j<n-1; j++)
for(k=j+1; k<n; k++)
{ if(class[j].roll > class[k].roll)
 { t=class[j];
  class[j]=class[k];
  class[k]=t;
 }}
for(k=0; k<n; k++){
  Printf("roll no %d code %s marks %.f\n", class[k].roll, class[k].
    code, class[k].marks); }
```

* Arrays within Structure:

C allows the use of arrays as structure members.

Ex:

```
Struct student{
Char name[20];
int rollno;
int submarks[4];
};
```

The array submarks represent subject marks. Each student has four subject and each subject mark is denoted by submarks[0], submarks[1], submarks[2], submarks[3].

➤ POINTERS TO STRUCTURES

A Pointer to a Structure Stores the Start address of a Structure variable.

It can define a pointer to a Structure in the same way as any pointer to any type.

For Ex:

```
Struct emprec *ptr
```

Defines a pointer to any type. For Ex. emprec. Use a pointer to a Struct in more or less the same way as any pointer but the use of qualified names makes it look slightly different. For Ex.

```
*ptr.age
```

It is the age component of the emprec Structure that ptr points at an int.

Here need the brackets because '.' has a higher Priority than '*'.

The use of a pointer to a Struct is so common, and the pointer notation so ugly, that there is an equivalent and more elegant way of writing the same thing.

Ptr → age

It mean the same thing as (*Ptr).age. The notation gives a clearer idea of what is going on - ptr points (i.e) → to the structure and .age picks out which component of the structure.

The → (Indirection operator) usually called the arrow operator, consists of the minus sign followed by a greater than sign. It is used in place of dot operator we are accessing a structure member through a pointer to the structure.


```
Struct Student {
char name[20];
int rollno;
int marks;
};
Struct Student Str, *Ptr;
```

Ptr is a pointer that can point to the variable of type Struct student.

Ex:

```
#include<Stdio.h>
#include<String.h>
Struct book {
   char title[10];
   double price;
   int pages;
};
int main() {
```

```
Struct book b1 = {"Learn C", 675.50, 325};
Struct book *strptr;
strptr = &b1;
Printf("Title : %s\n", strptr->title);
Printf("Price : %.1f\n", strptr->price);
Printf("No. of pages : %d\n", strptr->pages);
return 0;
}
```

O/P

Title : Learn C
Price : 675.5000
No. of pages : 325

➤ UNIONS AND THEIR USES :

It is also a collection of heterogeneous data types.

The only difference between Struct and union is the way memory is allocated to its members.

In Structure : Each member has its own memory location.

In UNION : Members share the same memory location.

When a or the union is declared, the Compiler allocates sufficient memory to hold the largest member in the union.

Note : Union is used for saving memory.

C Provides a data structure which fits our needs in this case called a union data type. A union type variable can store objects of different types at different times, however at any given moment it stores an object of only one of the specified types.

*UNION declaration:

The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure template.

For Ex, TO declare a union variable, person, with two members, a string and an integer. If the name is entered, use person to store the string, if an identification number is entered, use person to store an integer.

Syntax:

```
union name
{
 datatype member1;
 datatype member2;
 ---
 ---
};
```


Similar to structures, unions can have union variables to access the members.

Ex:

```
Union human
 {
  int id;
  Char name[25];
 }person;
```

Likewise, it is possible to declare just a tag, and later use the tag to declare variables:

```
union human
{
 int id;
 Char name[25]; }; Union human person, *ppos;
```

The members of a union variable may be accessed in the same manner as are members of a structure variable:

```
Var_name.member_name;
Ptr_to_union_var -> member_name
```

For Ex:

```
Ppers = &Person;
Person.id = 12;
if (ppers->id == 12)
  ....
Printf("Id = %d\n", Person.Id);
```

The type of data accessed is determined by the member name used to qualify the variable name.

In our ex, Person.id will access an integer, while person.name will access a string (a character Pointer).

The best way is to declare a structure containing both the union variable as a field and another field that indicates the type of data stored in the union.

For Ex: Can declare such a structure type and a structure array as follows.

```
#define NAME 0
#define ID 1
Struct Record
{ int ptype;
  union human person;
};
Struct record list[MAX];
```

Now, as read information about each element of list, if the information is numeric, we store it as 'id';

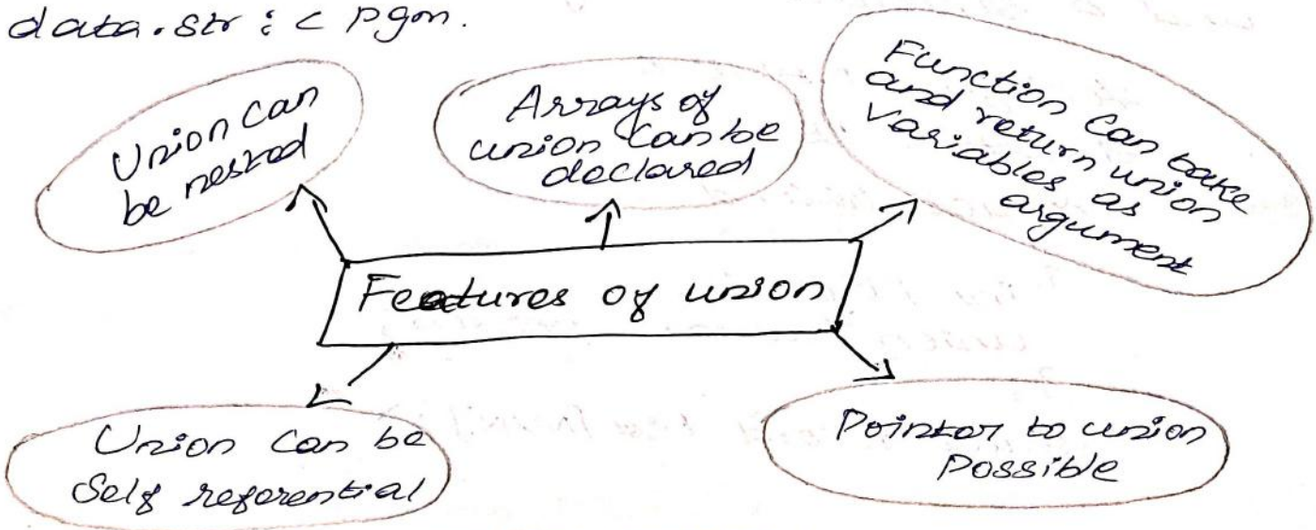
Otherwise store it as name. also store the type, ID or NAME in the member, ptype.

Ex:

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main()
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy(data.str, "C Pgm");
    Printf("data.i: %d\n", data.i);
    Printf("data.f: %f\n", data.f);
    Printf("data.str: %s\n", data.str);
    return 0;
}
```

O/P

```
data.i: 1917853763
data.f: 4122360532778604527.0000
data.str: C Pgm.
```



* Typedef

It allows you to define a new name for an existing datatype.

Syntax: typedef datatype_name new_name;

	Syntax
1) Pointers	typedef datatype *pointer_name; Ex: typedef float *fptr; fptr is synonym for float pointer Valid: fptr p, q, *r;
2) Arrays	typedef datatype arrayname[size]; Ex: typedef int arr[10]; arr is a synonym for integer array of 10 element. Valid: arr a, b, c[10];
3) Functions	typedef datatype func_name(arguments) Ex: typedef float fun(float, int); func is synonym for any function taking two argument one float and other integer. Valid: func add, sub, null;
4) Structures	typedef struct structure_name variable_name; Ex: typedef struct student Std; now structures student has a synonym Std. Valid: Std Stu1, Stu2;

ENUMERATIONS IN C:

In C, an enumeration (or enum) is a user defined data type that contains a set of named integer constants. It is used to assign meaningful names to integer values, which makes a program easy to read and maintain.

An enum must be defined before we can use it in program.

```
enum enum_name{
    n1, n2, n3, ....
};
```

Here the enum names must be unique in their scope, For Ex, the below code fails as two enum items and Calculate have same named integer PRODUCT.

```
enum Calculate{
    SUM, PRODUCT, DIFFERENCE
};
enum Item{
    PRODUCT, SERVICE
};
```

* Enum Initialization:

In enum variable can be initialized either with a name defined in the enum definition or directly with its integer value.

Ex:

```
#include<stdio.h>
//defining enum
```

```
enum direction {
    EAST, NORTH, WEST, SOUTH
  };
int main() {
    //Creating enum variable
    enum direction dir = NORTH;
    Printf("%d\n", dir);
    //This is valid too
     dir = 3;
     Printf("%d", dir);
     return 0;
    }
```

O/P

1

3

* Assign values manually:

The automatic assignment and integer values to the enumerated constant is overwritten if we explicitly assign values to the enumerated constants.

For Ex,

```
enum months {
    jan = 1, feb = 2, mar = 3,
    apr = 4 };
```

The above assignment is perfectly good. However, we can shorten it a little by using the Compiler.

Ex:

```
#include<stdio.h>
//Defining enum
enum enm {
    a = 3, b = 2, c
  };
```

```
int main(){
    //creating enum variable
    enum enm v1=a;
    enum enm v2=b;
    enum enm v3=c;
    Printf("%d %d %d", v1, v2, v3);
    return 0;
}
```

O/P

3 2 3

* Size of Enum:

Enum definition is not allocated any memory, it is only allocated for enum variables. Usually, enums are implemented as integers, so their size is same as that of int.

But some compilers may be short int to represent enum.

Ex:
```
#include<stdio.h>
// Defining enum
enum direction{
    EAST, NORTH, WEST, SOUTH};
int main(){
    enum direction dir=NORTH;
    // checking the size of enum
    Printf("%ld bytes", sizeof(dir));
    return 0;
}
```

O/P

4 bytes.

UNIT-6 FILE OPERATIONS

Open, read, write, close file operations Binary Vs Text files, File pointers, Error handling in File operations.

➤ CONCEPT OF A FILE

File handling in C is the process of handling file operations such as creating, Opening, reading, writing data and deleting, Closing the C language functions.

With the help of these functions, we can perform file operations to store and retrieve the data in/from the file in our Pgm.

* Types of Files:

A file represents a sequence of bytes. There are two types of files.

→ text files
→ Binary files.

→Text File:

A text file contains data in the form of ASCII characters and is generally used to store a stream of characters.

Each line in a text file ends with a new line character ("\n") and generally has a ".txt" extension.

→**Binary File**: A binary file contains data in raw bits (0 and 1). Different application pgms have different ways to represent bits and bytes and use different file formats.

The image files (.png, .jpg) the executable files (.exe, .com) etc are the examples of binary files.

* FILE Pointer (FILE*)

While working with file handling, need a file pointer to store the reference of the FILE structure returned by the fopen() function.

The file pointer is required for all file-handling operations.

The fopen() function returns a pointer of the FILE type. File is a predefined struct type in stdio.h and contains attributes such as the file descriptor, size and position etc.

```
typedef struct {
    int fd; /* FILE descriptor */
    unsigned char *buf; /* Buffer */
    size_t size; /* Size of the file */
    size_t pos; /* Current position in the file */
} FILE;
```


- **Declaring a file pointer:**

Syntax:

```
FILE** file_pointer;
```

⇒ 1. Opening / Creating a file:

A file must be opened to perform any perform any operation. The fopen() function is used to create a new file or open a existing file.

TO Specify the mode in which you want to open. The fopen() function returns a FILE pointer which will be used for other operations such as reading, writing and closing the files.

Syntax:

FILE *fopen(const char *filename, const char *mode);

Here filename is the name of the file to be opened, and mode defines the file opening mode.

⇒ File Opening Modes:

The file access modes by default open the file in the text or ASCII mode. If the binary files use the following access modes instead of the above-mentioned ones:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

There are various modes in which a file can be opened. The following are the different file opening modes.

Mode	Description.
r	Opens an existing text file for reading purposes.
w	Opens a text file for writing. If it doesn't exist, then a new file is created.
a	Opens a text file for writing in appending mode. If it doesn't exist then a new file is created.
r+	Opens a text file for both reading and writing.
	Opens a text file for both reading and writing.

It first truncates the file to zero length if it exists, otherwise creates a file if it doesn't exist.

Ex: Creating a File:

The file mode to create a new file will be "w" (write-mode).

```
Ex: #include<Stdio.h>
    int main(){
      FILE *file;
      //Creating a file
        file=fopen("file1.txt","w");
      //Checking whether file is
      //created or not.
      if(file==NULL){
        Printf("Error in creating file");
        return 1;
      }
      Printf("File created");
      return 0;
    }
```

O/P

File created.

Opening a File:

Ex: The file mode to open an existing file will be "r" (read-only). Note: There must be a file to be opened.

```
Ex:
    #include<Stdio.h>
    int main(){
      FILE *file;
      //opening a file
        file=fopen("file1.txt","r");
      //Checking whether file is
```

```
//opened or not
if (fp==NULL){
    Printf("Error in opening file");
    return 0;
}
Printf("File opened");
return 0;
}
```

O/P

File opened.

→ 3. Closing a file:

Each file must be closed after Performing operations on it. The fclose() function Closes an opened file.

Syntax:

```
int fclose(FILE *fp);
```

The fclose() function returns 0 on Success, or EOF if there is an error in closing the file.

The fclose() function actually flushes any data still pending in the buffer to the file, closes the file and releases any memory used for the file.

The EOF is a constant defined in the header file stdio.h.

Ex:

```
#include<stdio.h>
int main(){
    FILE *file;
    //opening a file
    file=fopen("file1.txt","w");
```

```
//checking whether file is opened or not
if (file == NULL) {
    Printf("Error in opening file");
    return 1;
}
Printf("File opened");
//closing the file
fclose(file);
Printf("\n File closed.");
return 0;
}
```

O/P

File opened
File closed

⇒ 4. Writing to a Text file:

The following library functions are provided to write data in a file opened in writeable mode.

1. fputc() → Writes a single character to a file
2. fputs() → Writes a string to a file.
3. fprintf() → Writes a formatted string (data) to a file.

* Writing single character to a file:

The putc() function is an unformatted function that writes a single character value of the argument "c" to the output stream referenced by "fp".

```
int fputc(int c, FILE *fp);
```

Ex:
One character from a given char array is written into a file opened in the "w" mode.

```
#include <stdio.h>
int main(){
  FILE *fp;
  char *string="c pgm";
  int i;
  char ch;
  fp=fopen("file1.txt","w");
  for(i=0;i<strlen(string);i++){
     ch=string[i];
     if(ch==EOF)
        break;
     fputc(ch,fp);
  }
  Printf("\n");
  fclose(fp);
  return 0;
}
```

O/P

"file1.txt" file will be created.

* Writing String to a File:

The fputs() function writes the string "s" to the output stream referenced by "fp". It returns a non-negative value on success, else EOF is returned in case of any error.

```
int fputs(const char *s, FILE *fp);
```

Ex:

```
#include <stdio.h>
int main(){
  FILE *fp;
```

```
char *sub[] = {"cpgm\n", "c++pgm\n", "python\n"};
fp = fopen("file2.txt", "w");
for(int i=0; i<4; i++){
    fputs(sub[i], fp);
}
fclose(fp);
return 0;
}
```

O/P

```
C Pgm
C++ Pgm
Python.
```

* Writing Formatted String to a file:

The printf() function sends a formatted stream of data to the disk file represented by the FILE pointer.


```
int fprintf(FILE *stream, const char *format
                                [, argument, ...])
```

Ex:

The array of struct type called "employee". The structure has a string, an integer and a float element. Using the fprintf() function, the data is written to a file.

Ex:

```
#include <stdio.h>
struct employee{
    int age;
    float percent;
    char *name;
};
```

```
int main() {
    FILE *fp;
    Struct employee emp[] = {
        {25, 65.5, "ABC"},
        {21, 75.5, "CDE"},
        {24, 60.5, "EFG"}
    };
    Char *String;
    fp = fopen("file3.txt", "w");
    for(int i=0; i<3; i++) {
        fprintf(fp, "%d %f %s\n", emp[i].age,
    emp[i].percent, emp[i].name);
    }
    fclose(fp);
    return 0;
}
```

5. Reading from a Text file:

The following library functions are Provided to read data from a file that is opened in read mode.

- → fgetc() → Reads a single character from a file
- → fgets() → Reads a string from a file
- → fscanf() → Reads a formatted string from a file.

* Reading single character from a file:

The fgetc() function reads a character from the input file referenced by "fp". The return value is the character read, or in case of any error, returns EOF.

```
int fgetc(FILE *fp);
```

Ex:

```
#include <stdio.h>
int main() {
    FILE *fp;
    char ch;
    fp = fopen("File1.txt", "r");
    while(1) {
        ch = fgetc(fp);
        if(ch == EOF)
        break;
        Printf("%c", ch);
    }
    Printf("\n");
    fclose(fp);
}
```

* Reading String from a FILE:

The fgets() function reads up to "n-1" characters from the input stream referenced by "fp". It copies the read string into the buffer "buf", appending a null character to terminate the string.

Ex:

```
#include <stdio.h>
int main() {
    FILE *fp;
    char *String;
    fp = fopen("file2.txt", "r");
    while (!feof(fp)){
        fgets(String, 256, fp);
        Printf("%s", String);
    }
    fclose(fp);
}
```

* Reading Formatted String from a FILE:

The fscanf() function in C pgming language is used to read formatted input from a file.

```
int fscanf(FILE *Stream, Const char *format,...)
```

➢ FILE HANDLING BINARY READ AND WRITE FUNCTIONS

The read/write operations are done in a binary form in the case of a binary file. Need to include a character "b" in the access mode ("wb" for writing a binary file, "rb" for reading a binary file).

There are two functions that can be used for binary input and output: the fread() function and the fwrite() function.

Both of these functions should be used to read or write blocks of memories, usually arrays or structures.

⇒ Writing to Binary File:

The fwrite() function writes a specified chunk of bytes from a buffer to a file opened in binary write mode. Here is the prototype to use this function.

```
fwrite(*buffer, size, no, FILE);
```

Ex:

In the following program, an array of a struct type called "employee" has been declared. Use the fwrite() function to write one block of byte, equivalent to the size of one employee data, in a file that is opened in "wb" mode.

```
#include<stdio.h>
Struct employee {
    int age;
    float percent;
    Char name[10];
};
int main(){
    FILE *fp;
    Struct employee e[] = {
        {25, 65.5, "ABC"},
        {21, 75.5, "CDE"},
        {24, 60.5, "EFG"}
    };
  Char *string;
 fp=fopen("file4.dat", "wb");
 for(int i=0; i<3; i++){
    fwrite(&e[i], sizeof(struct employee), 1, fp);
 }
 fclose(fp);
 return 0;
}
```

⇒ Reading from Binary file:

The fread() function reads a specified chunk of bytes from a file opened in binary read mode to a buffer of the specified size. Here is the prototype to use this function.

```
fread (*buffer, size, no, FILE);
```

Ex:

In the following program, an array of a struct type called "employee" has been declared. We use the fread() fn to read one block of byte equivalent to the size of one employee data, in a file that is opened in "rb" mode.


```
#include <stdio.h>
Struct employee {
    int age;
    float percent;
    Char name[10];
};
int main() {
    File *fp;
    Struct employee e;
    fp=open("file4.dat", "rb");
    if (fp==NULL){
        puts("Cannot open file");
        return 0;
    }
    while (fread(&e, sizeof(struct employee), 1, fp)==1)
    Printf("Name: %.s Age: %.d Percent: %f \n", e.name,
            e.age, e.percent);
    fclose(fp); return 0; }
```

* Renaming a File:

The rename() function is used to rename an existing file from an old file name to a new file name.

```
int rename(const char *old_filename, const char
                                    *new file_name);
```

Ex:

```
#include<stdio.h>
int main(){
    // old and new file names
    char *file_name1 = "file1.txt";
    char *file_name2 = "file2.txt";
  //Renaming old filename to new one
  if (rename(file_name1, file_name2)==0){
      printf("File renamed successfully.\n");
  }else{
      perror("There is an error.");
  }
  return 0;
}.
```

* Renaming a File:

The rename() function is used to rename an existing file from an old file name to a new file name.

```
int rename(const char *old_filename, const char
                                    *new file_name);
```

Ex:

```
#include<stdio.h>
int main(){
    // Old and new file names
    char *file_name1 = "file1.txt";
    char *file_name2 = "file2.txt";
    //Renaming old filename to new one
    if (rename(file_name1, file_name2)==0){
        Printf("File renamed successfully.\n");
    }else{
        perror("There is an error.");
    }
    return 0;
}.
```


ERROR HANDLING IN FILE OPERATIONS IN C:

Error handling in file operations is an essential part of Programming in C. When working with files, things can go wrong for various reasons.

For Ex, The file may not exist, the pgm might not have the correct permissions to access the file, or an unexpected end-of-file (EOF)

may be encountered. Handling those errors gracefully ensures that your program doesn't crash and provides useful feedback to the user.

* Common Errors in File Operations:

Some of the common errors might encounter while working with files include:

1) File not found → The file you are trying to open doesn't exist.

2) Permission issues → The file cannot be opened due to lack of read or write permissions.

3) End-of-File reached → Unexpectedly reaching the end of the file while reading

4) File write error → Issues encountered while trying to write data to the file.

* Error Handling Functions in C:

C provides several standard functions to help handle errors during file operations.

1. fopen() → Checking if the file was opened successfully.
2. feof() → Detecting the end of the file
3. ferror() → checking if an error occured during file operations.
4. perror() → Displaying a descriptive error messag based on the last file related error.

1. Checking if a file opened successfully (fopen())

The fopen() function returns a file pointer if the file was successfully opened. If the file can't be opened.

fopen() returns NULL. You can check for this condition and handle the error accordingly.

Ex:

```
FILE *filepointer;
filepointer = fopen("nonexistent.txt", "r");
if (filepointer == NULL) {
    Printf("Error: File could not be opened!\n");
    return 1; // Exit the pgm or handle the error.
```

2. Checking for End of File (feof())

The feof() function returns non-zero (true) if the end of the file has been reached. This function is useful when reading data from a file and when reached the end.

Syntax:

```
int feof(FILE *filepointer);
```

Ex:

```
FILE *filepointer;
Char buffer[100];
filepointer = fopen("data.txt", "r");
if (filepointer == NULL) {
    Printf("Error: File could not be opened?\n");
    return 1;
}
while (fgets(buffer, 100, filepointer) != NULL) {
    Printf("%s", buffer);
```

```
if (feof (filepointer)) {
    Printf("\n End of file reached.\n");
} else {
    Printf ("\n Error : End of file not reached as
                        expected.\n");
}
fclose (filepointer);
```

3. Checking for file Errors (ferror())

The ferror() function returns non-zero(true) if an error occurred during the last file operation. This function is useful to check if something went wrong while reading from or writing to a file.

Syntax:

```
int ferror (FILE *filepointer);
```


Ex:

```
FILE *filepointer;
Char buffer[100];
filepointer = fopen ("data.txt", "r");
if (filepointer == NULL) {
    Printf("Error : File could not be opened!\n");
    return 1;
}
//Reading from the file.
fgets (buffer, 100, filepointer);
//Checking for errors after reading.
if (ferror (filepointer)) {
    Printf("Error : Problem occured while
            reading the file \n");
} else {
    Printf("File read successfully.\n");
}fopen(filepointer);
```

4. Displaying Descriptive Error Messages (perror())

The perror() function prints a descriptive error message based on the most recent error encountered. It is particularly useful for printing system-level error messages when file operation fail.

Syntax:

```
void perror(const char *message);
```

Message: A custom message want to print before the system error message.

Ex:

```
FILE *filepointer;
filepointer = fopen("nonexistent.txt", "r");
if (filepointer == NULL) {
    perror("Error opening file"); // Print a custom error msg
    return 1;
}
```


O/P

Error opening file: No such file or directory.

⇒ Best practices for Error Handling in File Operations:

1. Always check file pointers: Always check if fopen() returns NULL before performing any operations on the file. This helps prevent crashes.

2. Use feof() to Detect End of File: When reading data, always use feof() to detect when you've reached the end of the file. It ensures that you don't try to read past the end of the file, which can cause errors.

3. Use ferror() After file operations: After Performing file operations (reading or writing), use ferror() to check if any error occured during the Process. It helps in identifying issues like corrupted files or write failures.

4. Use perror() for Descriptive Error Messages: Use perror() to display helpful error messages to the user. It adds context to the errors and making debugging easier.

5. Close Files Properly: Always close files using fclose() after operations. Check for errors when closing files, especially when writing data.

*Error Handling during File operations in C

Error	Cause
1. File Not Found	Trying to open a file that doesn't exist.
2. Permission denied	Insufficient permissions to access the file.
3. Disk Full	No space left on the disk for writing data.

4. File Already Exists	Attempting to create a file that already exists in w mode.
5. Invalid File Pointer	Using a null or invalid file pointer for file operations.
6. End-of-file (EOF)	Attempting to read past the end of the file.
7. File not open	Attempting to perform operations on a file that was not opened successfully.

Failure to check for errors then the program may behave abnormally therefore an unchecked error may result in premature termination for the program or incorrect output.