

# **MOMBASA TECHNICAL TRAINING INSTITUTE**



## **DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY**

### **STRUCTURED PROGRAMMING**

**P.O. BOX 81220, TEL 041-2226458, MOMBASA**

<b>STRUCTURED PROGRAMMING.....</b>	<b>4</b>
DEFINITION OF TERMS.....	4
<b>INTRODUCTION TO STRUCTURED PROGRAMMING.....</b>	<b>6</b>
HISTORY OF PROGRAMMING LANGUAGES .....	6
PROGRAMMING PARADIGMS.....	8
SOFTWARE CONSIDERATIONS.....	9
ADVANTAGES C LANGUAGE .....	10
<b>PROGRAM DEVELOPMENT AND DESIGN .....</b>	<b>11</b>
PROGRAM DEVELOPMENT CYCLE .....	11
STRUCTURED PROGRAMMING DESIGN CONCEPTS .....	12
PROGRAM DESIGN TOOLS .....	12
<b>3. PROGRAM STRUCTURE.....</b>	<b>17</b>
STRUCTURE OF A C PROGRAM.....	17
SOURCE CODE FILES .....	20
C DATA TYPES.....	21
VARIABLES .....	22
CONSTANTS .....	25
TYPE CASTING .....	25
C PROGRAMMING OPERATORS .....	26
ARITHMETIC OPERATORS .....	26
INCREMENT AND DECREMENT OPERATORS – Unary Operators .....	27
ASSIGNMENT OPERATORS – Binary Operators .....	27
RELATIONAL OPERATORS - Binary Operators .....	28
LOGICAL OPERATORS - Binary Operators .....	28
CONDITIONAL OPERATOR – Ternary Operators.....	29
PRECEDENCE OF OPERATORS .....	30
ASSOCIATIVITY OF OPERATORS .....	30
<b>CONTROL STRUCTURES.....</b>	<b>32</b>
IMPORTANCE OF CONTROL STRUCTURES .....	32
TYPES OF CONTROL STRUCTURES .....	32
SELECTION STRUCTURES .....	33
THE IF SELECTION STRUCTURE .....	33
THE IF/ELSE.....	34
THE IF...ELSE IF...ELSE STATEMENT .....	34
NESTED IF STATEMENTS.....	35
SWITCH STATEMENT.....	36
NESTED SWITCH STATEMENTS .....	38
REPETITION/ITERATIVE/LOOP STRUCTURES .....	39
WHILE LOOP IN C.....	39
FOR LOOP IN C.....	40
DO...WHILE LOOP IN C.....	42
NESTED LOOPS IN C.....	43
BRANCHING STATEMENTS .....	45
BREAK STATEMENT IN C .....	45
CONTINUE STATEMENT IN C.....	45
GOTO STATEMENT IN C.....	46
THE RETURN STATEMENT .....	47
THE INFINITE LOOP.....	47
<b>SUBPROGRAMS IN C .....</b>	<b>48</b>
TYPES OF FUNCTIONS .....	48
LIBRARY FUNCTION .....	48
USER DEFINED FUNCTION .....	49
DEFINING A FUNCTION.....	50

FUNCTION DECLARATIONS .....	51
CALLING A FUNCTION .....	51
FUNCTION ARGUMENTS .....	53
TYPES OF VARIABLES .....	53
LOCAL VARIABLES .....	53
GLOBAL VARIABLES .....	53
<b>DATA STRUCTURES .....</b>	<b>56</b>
ARRAYS .....	57
DECLARING ARRAYS .....	57
INITIALIZING ARRAYS .....	58
ACCESSING ARRAY ELEMENTS .....	58
SORT TECHNIQUES .....	59
SEARCHING ARRAYS .....	67
LINKED LISTS .....	69
POINTERS .....	69
C STRINGS .....	71
QUEUES .....	75
STACKS .....	76
<b>FILE HANDLING .....</b>	<b>77</b>
OPENING FILES .....	77
CLOSING A FILE .....	77
WRITING A FILE .....	78
READING A FILE .....	78
BINARY I/O FUNCTIONS .....	79
<b>SOFTWARE DOCUMENTATION .....</b>	<b>81</b>
PROCESS DOCUMENTATION .....	81
PRODUCT DOCUMENTATION .....	81

# STRUCTURED PROGRAMMING

## DEFINITION OF TERMS

1. **HEADER FILE** - A header file is a file with extension **.h** which contains C function declarations to be shared between several source files. A header file is used in a program by including it with the use of the preprocessing directive **#include**, which comes along with the compiler.  
**#include<stdio.h>**
2. **EXPRESSION** – These are statements that return a value. Expressions combine variables and constants to create new values or logical conditions which are either true or false e.g.  
**x + y, x <= y etc**
3. **KEYWORD** - A keyword is a reserved word in C. Reserved words may not be used as constants or variables or any other identifier names. Examples include auto, else, Long, switch, typedef, break etc
4. **IDENTIFIER** - A C **identifier** is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore **\_** followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, \$, and % within identifiers.
5. **COMMENT** - These are non-executable program statements meant to enhance program readability and maintenance- they document the program.
6. **FUNCTION** - A function is a group of statements, enclosed within curly braces, which together perform a task.
7. **STATEMENT**- Statements are expressions, assignments, function calls, or control flow statements which make up C programs. Statements are terminated using a semicolon.
8. **SOURCE CODE** – Program instructions in their original form. C source code files have an extension **.c**
9. **OBJECT CODE** – Code produced by a compiler from source code and exists in machine readable language.
10. **EXECUTABLE FILE** – Refers to a file in a format that a computer can directly execute and is created by a compiler.
11. **STANDARD LIBRARY** – Refers to a collection of precompiled functions/routines that a program can use. The routines are stored in object format and contain descriptions of functions to perform I/O, string manipulations, mathematics etc.
12. **SIGNED INTEGER** – This is an integer that can hold either positive or negative numbers.
13. **COMPILER** – This is a program that translates source code into object code.
14. **PREPROCESSOR COMMAND** - The C preprocessor modifies a source file before handing it over to the compiler for instance by including header files with **#include** as **I #include <stdio.h>**
15. **LINKER/Binder/Link Editor** – This is a program that combines object modules to form an executable program. The linker combines the object code, the start up code and the code for

library routines used in the program (all in machine language) into a single file- the executable file.

16. **OPERATOR** - A symbol that represents a specific action. For example, a plus sign (+) is an operator that represents addition. The basic mathematic operators are + **addition**, - **subtraction**, \* **multiplication**, / **division**

17. **OPERAND** - Operands are the objects that are manipulated by operators in expressions. For example, in the expression  $5 + x$ ,  $x$  and  $5$  are operands and  $+$  is an operator. All expressions have at least one operand.

18. **EXPRESSION** – This is a statement that returns a value. For example, when you add two numbers together or test to see whether one value is equal to another.

19. **VARIABLE** - A variable is a memory location whose value can change during program execution. Variable declaration must have a type, which defines what values that variable can hold.

20. **Data type** – The data type of a variable etc determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

21. **CONSTANT** - a constant is a value that never changes during program execution.

22. **WHITESPACE** - A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it. Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as `int`, ends and the next element begins. Therefore, in the following statement:

```
int age;
```

There must be at least one whitespace character (usually a space) between `int` and `age` for the compiler to be able to distinguish them. On the other hand, in the following statement

## CHAPTER 1

# INTRODUCTION TO STRUCTURED PROGRAMMING

**Programming** means to convert problem solutions into instructions for the computer. It also refers to the process of developing and implementing various sets of instructions to enable a computer to do a certain task.

**Structured programming** (sometimes known as *modular programming*) is an approach to writing programs that are easier to test, debug, modify and maintain by enforcing a modular approach which breaks a large complex problem into sub-problems.

A programming language is a vocabulary and set of grammatical rules designed for instructing a computer to perform specific tasks.

## HISTORY OF PROGRAMMING LANGUAGES

### First-Generation Programming Languages – Machine Language

A first-generation of programming languages includes **machine-level** programming languages. These languages were introduced in the 1940s and had the following characteristics:

- Instructions were entered directly in binary format (1s and 0s) and therefore they were tedious and error prone. Programmers had to design their code by hand then transfer it to a computer using a punch card, punch tape or flicking switches.
- Instructions were executed directly by a computer's central processing unit (CPU) i.e. they were executed very fast.
- Memory management was done manually.
- Programs were very difficult to edit and debug.
- Used to code simple programs only.

### Second-Generation Programming Languages (2GL) – Low Level Programming Languages/Assembly Languages

They were introduced to mitigate the error prone and excessively difficult nature of binary programming.

- Introduced in the 1950s
- Improved on first generation by providing human readable **source code** which must be compiled/assembled into machine code (binary instructions) before it can be executed by a CPU
- Specific to platform architecture i.e. 2GL source code is **not portable** across processors or processing environments.
- Designed to support logical structure and debugging.

By using codes resembling English, programming becomes much easier. The use of these **mnemonic codes** such as **LDA** for **load** and **STA** for **store** means the code is easier to read and write. To convert an assembly code program into object code to run on a computer requires an **Assembler** and each line of assembly can be replaced by the equivalent one line of object (machine) code:

## Assembly Code

## Machine Code

LDA A		000100110100
ADD #5		001000000101
STA A	-> Assembler ->	001100110100
JMP #3		010000000011

Such languages are sometimes still used for *kernels* and *device drivers*, i.e. the core of the operating system and for specific machine parts. More often, such languages are used in areas of intense processing, like *graphics programming*, when the code needs to be **optimized for performance**.

Almost every CPU architecture has a companion assembly language. Most commonly used are the assembly languages today like Autocoder for IBM mainframe systems, Linoreum, MACRO -11, etc.

## Third-Generation Languages (3GL) – High-Level Languages

Third generation languages are the primary languages used in general purpose programming today. They each vary quite widely in terms of their particular abstractions and syntax. However, they all share great enhancements in logical structure over assembly languages.

- Introduced in the 1950s
- Designed around ease of use for the programmer (Programmer friendly)
- Driven by desire for reduction in **bugs**, increases in **code reuse**
- Based on natural language
- Often designed with structured programming in mind
- The languages are architecture independent e.g. C, Java etc.

### Examples:

Most Modern General Purpose Languages such as C, C++, C#, Java, Basic, COBOL, Lisp and ML.

## Fourth Generation Languages

Fourth-generation programming languages are high-level languages built around database systems. They are generally used in commercial environments.

- Improves on 3GL and their development methods with **higher abstraction** and **statement power**, to reduce errors and increase development speed by reducing programming effort. They result in a reduction in the cost of software development.
- A 4GL is designed with a **specific purpose** in mind. For example languages to query databases (**SQL**), languages to make reports (Oracle Reports) etc.
- 4GL are more oriented towards problem solving and systems engineering.

Examples: Progress 4GL, PL/SQL, Oracle Reports, Revolution language, SAS, SPSS, SQ

## Fifth Generation Languages

Improves on the previous generations by skipping algorithm writing and instead provide **constraints/conditions**.

While 4GL are designed to build specific programs, 5GL are designed to make the computer solve a

given problem without the programmer. The programmer only needs to worry about **what problems needed to be solved and only inputs a set of logical constraints**, with no specified algorithm, and the **Artificial Intelligence (AI)-based compiler builds the program based on these constraints**

Examples: Prolog, OPS5, Mercury

## Low-Level Languages Versus High-Level Languages

**Low-level languages** such as **machine language** and **assembly language** are closer to the hardware than are the high-level programming languages, which are closer to human languages. Low-level languages are converted to machine code without using a compiler or interpreter, and the resulting code runs directly on the processor. A program written in a low-level language runs **very quickly**, and with a **very small memory footprint**; an equivalent program in a high-level language will be more heavyweight. Low-level languages are **simple**, but are considered **difficult to use**, due to the numerous technical details which must be remembered.

**High-level languages** are closer to human languages and further from machine languages.

The main advantage of high-level languages over low-level languages is that they are **easier to read, write, and maintain**. Ultimately, programs written in a high-level language must be translated into machine language by a compiler or interpreter.

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including Ada, Algol, BASIC, COBOL, C, C++, FORTRAN, LISP, Pascal, and Prolog.

## PROGRAMMING PARADIGMS

A programming paradigm is a fundamental style of computer programming, a way of building the structure and elements of computer programs. There are four main paradigms:

### a) Unstructured Programming

In unstructured programs, the statements are executed in sequence (one after the other) as written. This type of programming uses the GoTo statement which allows control to be passed to any other section in the program. When a GoTo statement is executed, the sequence continues from the target of the GoTo. Thus, to understand how a program works, you have to execute it. This often makes it difficult to understand the logic of such a program.

### b) Structured Programming

The approach was developed as a solution to the challenges posed by unstructured/procedural programming. Structured programming frequently employs a **top-down design model**, in which developers **break the overall program** structure into **separate subsections**. A defined function or set of similar functions is coded in a separate module or sub-module, which means that **code can be loaded into memory more efficiently** and that **modules can be reused in other programs**. After a module has been tested individually, it is then integrated with other modules into the overall program structure.



Program flow follows a simple hierarchical model that employs looping constructs such as "for," "repeat," and "while." Use of the "GoTo" statement is discouraged.

Most programs will require thousands or millions of lines of code. (Windows 2000 – over 35 millions lines of code). The importance of splitting a problem into a series of self-contained modules then becomes obvious. A module should not exceed 100 lines, and preferably short enough to fit on a single page or screen.

Examples of structured programming languages include:

- C
- Pascal
- Fortran
- Cobol
- ALGOL
- Ada
- dBASE etc.

### **c) Object-oriented programming (OOP)**

This is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

### **d) Visual Programming**

A visual programming language uses a visual representation (such as graphics, drawings, animation or icons, partially or completely). A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions

A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols, used either as elements of syntax or secondary notation. For example, many VPLs (known as dataflow or diagrammatic programming) are based on the idea of "boxes and arrows", where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations. An example of visual programming languages is Microsoft Visual Basic which was derived from BASIC and enables the rapid application development (RAD) of graphical user interface (GUI) applications.

Programming in VB is a combination of visually arranging components or controls on a form, specifying attributes and actions for those components, and writing additional lines of code for more functionality.

### **e) Internet Based Programming**

This is programming oriented to the development of internet applications using languages and tools such as PHP, ASP, Perl, JavaScript, HTML, Java etc.

## **SOFTWARE CONSIDERATIONS**

Before you can start programming in C, you will need text editor such as a plain text Notepad Editor though it does not offer code completion or debugging. Many programmers prefer and recommend using an Integrated Development Environment (IDE) instead of a text editor on which to code, compile and test their programs.

Memory requirements

Disk space required

## ADVANTAGES C LANGUAGE

1. **Modularity:** modularity is one of the important characteristics of C. we can split the C program into no. of modules instead of repeating the same logic statements (sequentially). It allows reusability of modules.
2. **General purpose programming language:** C can be used to implement any kind of applications such as math's oriented, graphics, business oriented applications.
3. **Portability:** we can compile or execute C program in any operating system (UNIX, dos, windows).
4. **Powerful and efficient programming language:** C is very efficient and powerful programming language; it is best used for data structures and designing system software. Efficient in that it is a modular programming language and thus makes efficient use of memory and system resources.

## CHAPTER 2

# PROGRAM DESIGN AND DEVELOPMENT

**Program/System design** is the process of defining the software structure (components/ modules, interfaces, and data) to satisfy/realize specified requirements.

**Program development** refers to all the activities and processes involved between the conception of the desired software through to the final manifestation of the software, in a planned and structured process.

## PROGRAM DEVELOPMENT CYCLE

This refers to the stages that form the framework for planning and controlling the creation of an information system. Several approaches to program development have been devised and the System Development Life Cycle (SDLC) is one of the most popular. The SDLC is a methodology that aims at producing a high quality system that **meets or exceeds customer expectations, reaches completion within times and cost estimates, works efficiently and is inexpensive to maintain and cost-effective to enhance.**

- 1) Project **planning, feasibility study:** The fundamental process of understanding why a system should be built and determining how the project team will go about building it. It should also establish a clear understanding of the current system. It involves
  - a. Technical feasibility study: can the system be built
  - b. Economic feasibility study: will the system provide business value, and what are the risks?
  - c. Organizational feasibility study: if built, will it be used.
- 2) Systems **analysis, requirements definition:** the phase identifies the users of the system, what the system will do. It involves
  - a. Analysis of the old system and ways to design the new system
  - b. Requirement gathering. Various tools for collecting information are used. These include interviews, questionnaires, observation etc.
  - c. Development of the new system proposal document.
- 3) Systems **design:** describes how the system will operate, in terms of hardware, software, network infrastructure, user interface, forms and reports that will be used, the specific programs, databases and files that will be needed. Design phase steps include:
  - a. Design strategy: method of development, in-house, outsourced, or purchased
  - b. Architecture design – hardware , software, internet infrastructure, and user interface
  - c. Database and file specification
  - d. Program design: defines the program that needs to be done and exactly what each will do.
- 4) System **Implementation:** The real code is written here.
- 5) **Integration and testing:** Brings all the pieces of the project together into a special testing environment, then checks for errors, bugs and interoperability.
- 6) **Acceptance, installation, deployment:** The final stage of initial development, where the software is put into use and runs actual business.
- 7) **Maintenance:** What happens during the rest of the software's life: changes, correction, additions, and moves to a different computing platforms etc. This step, perhaps most important of all, goes on seemingly forever.

The SDLC is a cycle i.e. iterative in that a new requirement might initiate the whole process again.

# STRUCTURED PROGRAMMING DESIGN CONCEPTS

## 1. TOP-DOWN DESIGN

A **top-down** approach (also known as **stepwise design** or **deductive reasoning**, and in many cases used as a synonym of analysis or decomposition) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. Top-down approach starts with the big picture. It breaks down from there into smaller segments.

Top-down design (also called "**Modular programming**" and "**stepwise refinement**") therefore, is a software design technique that emphasizes separating the functionality of a program into independent modules such that each module is designed to execute only one aspect of the desired functionality.

## ADVANTAGES OF MODULAR PROGRAMMING

- Allows a problem to be split in stages and for a team of programmers to work on the different modules thereby reducing program development time.
- Program modification and debugging is easier since changes can be isolated to specific modules.
- Modules can be tested and debugged independently.
- Since a module performs a specific and well defined task, it is possible to develop and place commonly used modules in a user library so that they can be accessed by different programs. This is also called code reuse. (E.g. Validation, Uppercase, Text color etc.)
- If a programmer cannot continue through the entire project, it is easier for another programmer to continue working on self-contained modules.

## 2. BOTTOM-UP DESIGN

A **bottom-up** approach (also known as inductive reasoning, and in many cases used as a synonym of synthesis) is the piecing together of systems to give rise to larger systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

**With this approach, there is more user and business awareness of the product.** Benefits are also realized in the early phases of development.

## 3. MONOLITHIC DESIGN

The monolithic design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function

A monolithic application describes a software application which is designed without modularity.

# PROGRAM DESIGN TOOLS

## 1. Pseudo-code

Pseudo-code is an informal high-level description of the operating principle of a computer program or other algorithm and is intended for human reading rather than machine reading. Pseudo-code is a way to describe the algorithm in order to transform the algorithm into real source code.

For example, the Pseudo-code for comparing three numbers might be written:

**Begin**  
**If A is greater than B**  
**And if A is greater than C, A is the Biggest**  
**Otherwise C is the Biggest**  
**Otherwise**  
**If B is greater than C B is the Biggest**  
**Otherwise C is the Biggest**  
**End**

Pseudo-code cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step - an important one - in producing the final code

## 2. Algorithm

This refers to an established, computational procedure for solving a problem in a finite number of steps. Algorithms can be expressed in any language including natural languages such as English. Algorithm means a method/ logic for solving a given problem.

**An algorithm to find the largest among three different numbers entered by user.**




Step 1: Start  
 Step 2: Declare variables a, b and c.  
 Step 3: Read variables a, b and c.  
 Step 4: If a>b  
     If a>c  
         Display a is the largest number.  
     Else  
         Display c is the largest number.  
 Else  
     If b>c  
         Display b is the largest number.  
     Else  
         Display c is the greatest number.  
 Step 5: Stop







## 3. Flowchart

A **flowchart** is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem. Flowcharts are used in designing and documenting complex processes or programs. Like other types of diagrams, they help to visualize what is going on and thereby help the viewer to understand a process, and perhaps also find flaws/errors, bottlenecks, and other less-obvious features within it.

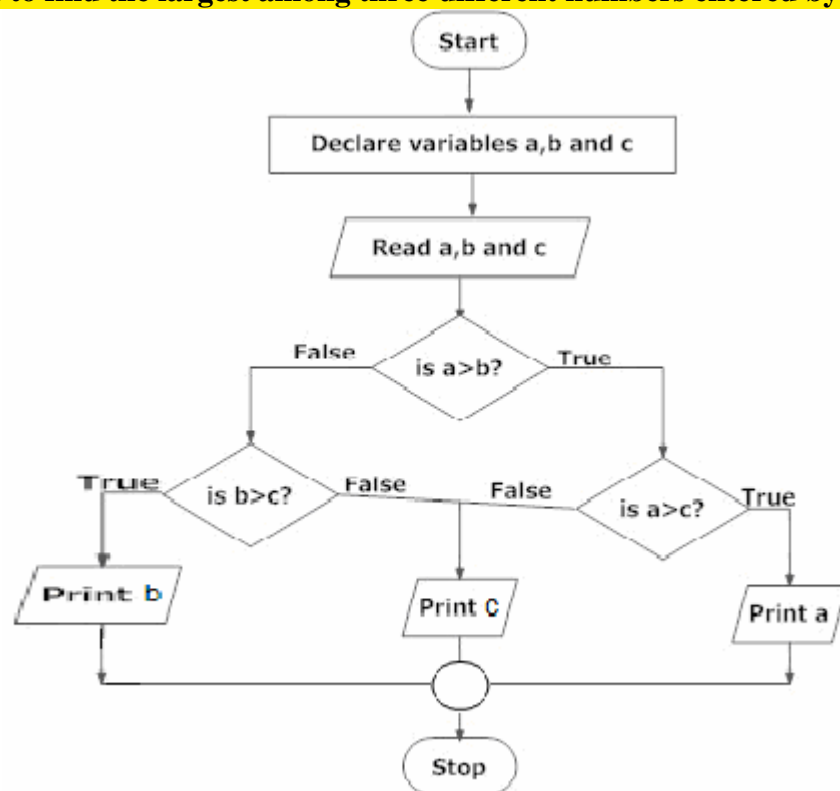
## SYMBOLS USED IN FLOWCHART

Different symbols are used for different states in flowcharts. The table below describes all the symbols that are used in making flowchart

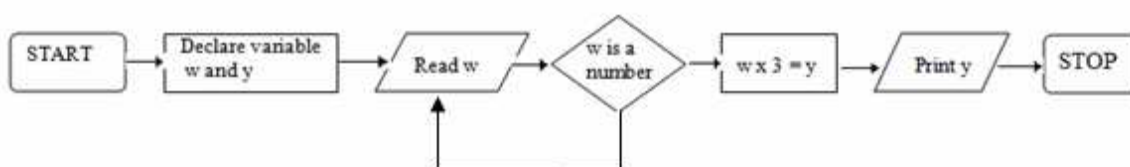
Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.

Symbol	Purpose	Description
	Processing	Used for arithmetic operations and data-manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flow lines
	Off-page Connector	Used to connect flowchart portion on different pages.
	Comment	Used to add comments or clarification
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

**Draw flowchart to find the largest among three different numbers entered by user.**



**Or a flowchart to ask for a number from user and multiply with another number and print result as follows:**



**Examples of flowcharts include Activity diagram, Data flow diagram and sequence diagrams etc.**

## **STRUCTURE CHARTS (module chart, hierarchy chart)**

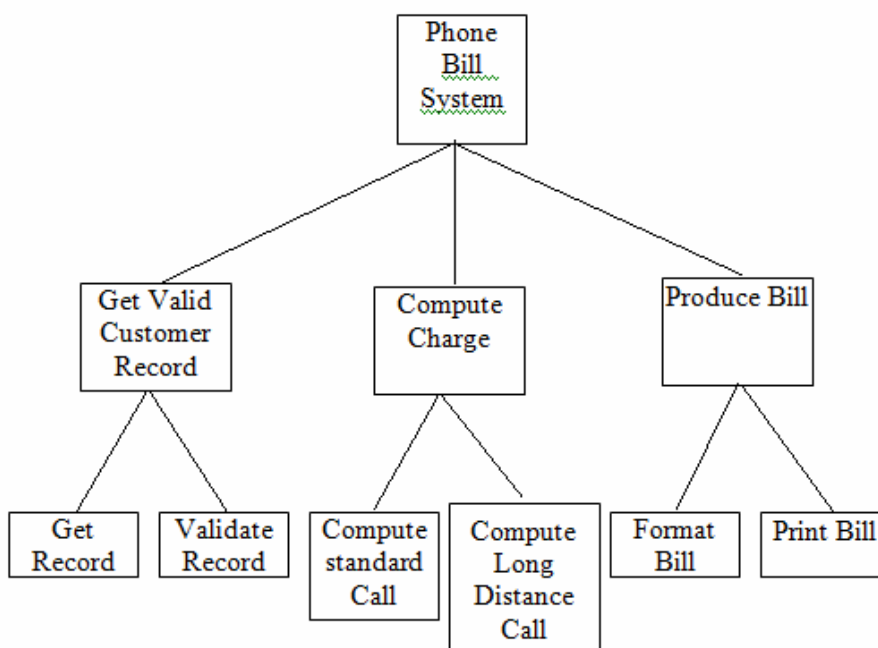
A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. A structure chart is a graphic depiction of the decomposition of a problem. It is a tool to aid in software *design* but is particularly helpful on large problems.

A structure chart illustrates the partitioning of a problem into sub-problems and shows the **hierarchical relationships among the parts**. A classic "organization chart" for a company is an example of a structure chart.

The top of the chart is a box representing the entire problem, the bottom of the chart shows a number of boxes representing the less complicated sub-problems (e.g. Phone Bill System).

A structure chart is NOT a flowchart. It has nothing to do with the logical sequence of tasks. It does NOT show the order in which tasks are performed. It does NOT illustrate an algorithm.

Each block represents some function in the system, and thus should contain a verb phrase, e.g. "Print report heading."



## **Decision Tables**

Decision tables provide a handy and compact way to represent complex business logic. In a decision table, business logic is well divided into **conditions**, **actions** (decisions) and **rules** for representing the various components that form the business logic.

There is one row for each condition and each vertical column for each combination of values and resulting actions. **Conditions** are the factors to consider when making certain business decision. **Actions** are the possible actions to take when certain business decision is made.

Each vertical column of a decision table is called a **rule** and each rule symbolizes the combinations of condition(s) and action(s) that form the business decision. If constructed properly, the decision

table has a rule to cover every combination. Rules are made of selectors symbolized by Y (Yes), N (No) and – (for redundant or irrelevant rules).

### COMPLETING THE TABLE

- If the number of conditions identified is n, then the number of possible rules is found by applying the formula; Number of rules =  $2^n$  i.e.  $2^{\text{number of conditions}}$
- Entries opposite the lowest condition should be completed first using Y and N alternately until all the vertical rules have been dealt with.
- Entries opposite the second lowest condition should be completed using Y and N in pairs until all the vertical rules have been dealt with.
- Entries for the next condition are then completed next using Ys and Ns in fours.
- This process continues using twice the number of Ys and Ns each time until all conditions are completed.
- Once entered, the rules are read vertically in each column and an X entered at the action that appropriately completes that rule. Actions which are mutually exclusive can be combined on a single line.

### EXAMPLE

A student who passes the examinations and completes the coursework and project satisfactorily is awarded a pass. If the course work and the project are unsatisfactory, the student is asked to resubmit the unsatisfactory work, as long as the exams have been passed. A student who fails the examinations is deemed to have failed the whole course unless both the course work and the project are satisfactory, in which case the student is allowed to re-sit the examination.

### SOLUTION

#### Before an overall pass

		Rules							
	Exams Passed?	Y	Y	Y	Y	N	N	N	N
<b>Conditions</b>	Completed Course work?	Y	Y	N	N	Y	Y	N	N
	Completed Project?	Y	N	Y	N	Y	N	Y	N
	Pass	X							
	Re-sit Exam					X			
<b>Actions</b>	Resubmit Coursework			X	X				
	Resubmit Project		X		X				
	Failed						X	X	X

### ASSIGNMENT 1

Candidates are accepted for employment if they pass the interview and their qualifications and reference are satisfactory. If they pass the interview and the qualifications or references (but not both) are unsatisfactory, a job for probationary period is offered. In all other circumstances, the candidate's application is rejected.

### Assignment 2

- Design the business logic (at least three conditions) that is applicable when a customer is applying for a bank loan.
- Draw a decision table to represent the business logic.



## CHAPTER 3

### 3. PROGRAM STRUCTURE

#### STRUCTURE OF A C PROGRAM

The **C programming language** was designed by Dennis Ritchie as a systems programming language for Unix.

A C program basically has the following structure:

- Preprocessor Commands
- Functions
- Variable declarations
- Statements & Expressions
- Comments

Example:

```
#include <stdio.h>

int main()
{
    /* My first program*/
    printf("Hello, World! \n");

    return 0;
}
```

#### Preprocessor Commands

These commands tell the compiler to do **preprocessing before doing actual compilation**. Like `#include <stdio.h>` is a preprocessor command which tells a C compiler to include **stdio.h** file before going to actual compilation. The standard input and output header file (stdio.h) allows the program to interact with the screen, keyboard and file system of the computer.

```
# include <stdio.h> }
# include <math.h>  } header files
# include <stdlib.h> }
```

NB/ Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler.

#### Functions

These are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. When this function is prefixed with keyword *int*, it means this function returns an integer value when it exits. This integer value is returned using *return* statement.

The C Programming language provides a set of built-in functions. In the above example `printf()` is a C built-in function which is used to print anything on the screen.

**A function is a group of statements that together perform a task.** A C program can be divided up into separate functions but logically the division usually is so each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
body of the function/Function definition
}
```

## Variable Declarations

In C, all variables must be declared before they are used. Thus, C is a **strongly typed** programming language. Variable declaration ensures that appropriate memory space is reserved for the variables. Variables are used to hold numbers, strings and complex data for manipulation e.g.

```
Int x;
Int num; int z;
```

## Statements & Expressions

**Expressions** combine variables and constants to create new values e.g.

```
x + y;
```

**Statements** in C are **expressions, assignments, function calls, or control flow** statements which make up C programs.

An assignment statement uses the assignment operator “=” to give a **variable** on the operator’s left side the **value** to the operator’s right or the result of an expression on the right.

```
z = x + y;
```

## Comments

These are **non-executable program statements** meant to **enhance program readability** and allow **easier program maintenance**- they **document the program**. They are **ignored by the compiler**. These are used to give additional useful information inside a C Program. All the comments will be put inside /\*...\*/ or // for single line comments as given in the example above. A comment can span through multiple lines.

```
/* Author: Mzee Moja */
```

or

```
/*
 * Author: Mzee Moja
 * Purpose: To show a comment that spans multiple lines.
 * Language: C
 */
```

or

```
Fruit = apples + oranges; // get the total fruit
```

## Escape Sequences

Escape sequences (also called back slash codes) are character combinations that begin with a backslash symbol used to format output and represent difficult-to-type characters.

They include:

```
\a      Alert/bell
\b      Backspace
\n      New line
\v      Vertical tab
\t      Horizontal tab
\\      Back slash
\'      Single quote
```

\” Double quote

\0 Null

## Note the following

- C is a **case sensitive** programming language. It means in C *printf* and *Printf* will have different meanings.
- End of each C statement must be marked with a semicolon.
- Multiple statements can be on the same line.
- Any combination of spaces, tabs or newlines is called a **white space**. C is a free-form language as the C compiler chooses to ignore whitespaces. Whitespaces are allowed in any format to improve readability of the code. **Whitespace is the term used in C to describe blanks, tabs, newline characters and comments.**
- Statements can continue over multiple lines.
- A C **identifier** is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore \_ followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, \$, and % within identifiers.
- A **keyword is a reserved word** in C. Reserved words may not be used as constants or variables or any other identifier names

## SAMPLE PROGRAM

```
//First program
```

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

```
main()
```

```
{
```

```
    int num; // Declaration
```

```
    num =1; // Assignment statement
```

```
    printf(" My favorite number is %d because", num);
```

```
    printf(" it is first.\n");
```

```
    return 0;
```

```
}
```

The program will output (print on screen) the statement “**My favorite number is 1 because it is first**”.

The %d instructs the computer where and in what form to print the value. %d is a **type specifier** used to specify the output format for integer numbers.

## Keywords

The following list shows the **reserved words** in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	Long	switch
break	enum	register	typedef

case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_packed
double			

## SOURCE CODE FILES

When you write a program in C language, **your instructions** form the source code/file. C files have an extension **.c**. The part of the name before the period is called the **extension**.

## Object Code, Executable Code and Libraries

An **executable file** is a file containing ready to run machine code. C accomplishes this in two steps.

- ✓ Compiling –The compiler converts the source code to produce the intermediate **object code**.
  - ✓ The linker combines the intermediate code with other code to produce the executable file.
- You can compile individual modules and then combine modules later.

**Linking** is the process where the object code, the start up code and the code for library routines used in the program (all in machine language) are combined into a single file- the executable file.

- **NB/** An **interpreter** unlike a **compiler** is a computer program that directly executes, i.e. performs, instructions written in a programming, without previously compiling them into a machine language program.
- If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a **cross-compiler**.
- A program that translates from a low level language to a higher level one is a **decompiler**.
- A program that translates between high-level languages is usually called a **source-to-source compiler** or **transpiler**.

## Library Functions

There is a minimal set of library functions that should be supplied by all C compilers, which your program may use. This collection of functions is called the C standard library. The standard library contains functions to perform disk I/O (input/ output), string manipulations, mathematics and much more. When your program is compiled, the code for library functions is automatically added to your program. One of the most common library functions is called **printf()** which is a general purpose output function. The quoted string between the parenthesis of the printf() function is called an argument.

### Printf(“This is a C program\n”)

The \n at the end of the text is an **escape sequence** tells the program to print a new line as part of the output.

## C DATA TYPES

In the C programming language, data types refer to a system used for declaring variables or functions of different types. A data type is, therefore, a data storage format that can contain a specific type or range of values. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The basic data types in C are as follows:

Type	Description
Char	Character data and is used to hold a single character. A character can be a letter, number, space, punctuation mark, or symbol - 1 byte long
Int	A signed whole number in the range -32,768 to 32,767 - 2 bytes long
Float	A real number (that is, a number that can contain a fractional part) – 4 bytes
Double	A double-precision floating point value. Has more digits to the right of the decimal point than a float – 8 bytes
Void	Represents the absence of type. i.e. represents “no data”

## USING C’S DATA TYPE MODIFIERS

The five basic types (int, float, char, double and void) can be modified to your specific need using the following specifiers.

- **Signed**

Signed Data Modifier implies that the data type variable can store positive values as well as negative values.

The use of the modifier with integers is redundant because the default integer declaration assumes a signed number. The signed modifier is used with char to create a small signed integer. Specified as signed, a char can hold numbers in the range -128 to 127.

- **Unsigned**

If we need to change the data type so that it can only store positive values, “unsigned” data modifier is used.

This can be applied to char and int. When char is unsigned, it can hold positive numbers in the range 0 to 255.

- **Long**

Sometimes while coding a program, we need to increase the Storage Capacity of a variable so that it can store values higher than its maximum limit which is there as default.

This can be applied to both **int** and **double**. When applied to **int**, it doubles its **length**, in bits, of the base type that it modifies. For example, an integer is usually 16 bits long. Therefore a **long int** is 32 bits in length. When **long** is applied to a double, it roughly doubles the precision.

- **Short**

A “short” type modifier does just the opposite of “long”. If one is not expecting to see high range values in a program.

For example, if we need to store the “age” of a student in a variable, we will make use of this type qualifier as we are aware that this value is not going to be very high

The type modifier precedes the type name. For example this declares a **long integer**.

**long int age;**

## Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

Type	Storage size	Value range
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

## Floating-Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

## The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

	Types and Description
1	Function returns as void. There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
2	Function arguments as void. There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, int rand(void);
3	Pointers to void A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function void *malloc( size_t size ); returns a pointer to void which can be casted to any data type.

## VARIABLES

A variable is a memory location whose value can change during program execution. In C a variable must be declared before it can be used.

### Variable Declaration

**Declaring** a variable tells the compiler to reserve space in memory for that particular variable. A **variable definition** specifies a data type and the variable name and contains a list of one or more variables of that type. Variables can be declared at the start of any block of code. A declaration begins with the **type**, followed by the name of one or more variables. For example,

Int high, low;

int i, j, k;

char c, ch;

float f, salary;

Variables can be initialized when they are declared. This is done by adding an equals sign and the required value after the declaration.

```
Int high = 250;           /*Maximum Temperature*/
Int low = -40;            /*Minimum Temperature*/
Int results[20];          /* series of temperature readings*/
```

## TYPES OF VARIABLES

The Programming language C has two main variable types

- Local Variables
- Global Variables

### Local Variables

A local variable is a variable that is declared inside a function.

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

### Global Variables

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it. Global variables are declared outside **all** functions.

#### Sample Program.

```
#include <stdio.h>
int area; //global variable

int main ()
{
    int a, b; //local variable

    /* actual initialization */
    a = 10;
    b = 20;

    printf("\t Side a is %d cm and side b is %d cm long\n",a,b);

    area = a*b;
    printf("\t The area of your rectangle is : %d \n", area);

    return 0;
}
```

## Variable Names

Every variable has a name and a value. The name identifies the variable and the value stores data. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters. C is case sensitive i.e. it recognizes upper and lower case characters as being different. You cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names:

X result outfile x1 out\_file etc

It is conventional in C not to use capital letters in variable names. These are used for names of constants.

## Declaration vs Definition

A declaration provides basic attributes of a symbol: its type and its name. A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored. Often, the compiler only needs to have a declaration for something in order to compile a file into an object file, expecting that the linker can find the definition from another file. If no source file ever defines a symbol, but it is declared, you will get errors at link time complaining about undefined symbols. In the following short code, the definition of variable x means that the storage for the variable is that it is a global variable.

```
int x;
int main()
{
    x = 3;
}
```

## Inputting Numbers From The Keyboard Using Scanf()

Variables can also be initialized during program execution (run time). The **scanf()** function is used to read values from the keyboard. For example, to read an integer value use the following general form:

```
scanf("%d", &var_name)
```

As in

```
scanf("%d", &num)
```

The %d is a format specifier which tells the compiler that the second argument will be receiving an integer value.

The & preceding the variable name means "address of". The function allows the function to place a value into one of its arguments.

The table below shows format specifiers or codes used in the scanf() function and their meaning.

%c	Read a single character
%d	Read an integer
%f	Read a floating point number
%lf	Read a double
%s	Read a string
%u	Read a an unsigned integer



When used in a printf() function, a type specifier informs the function that a different type item is being displayed.

### SAMPLE PROGRAM USING SCANF()

```
#include <stdio.h>
int area; //global variable

int main ()
{
    int a, b; //local variables

    /* actual initialization */
    printf("Enter the value of side a: ");
    scanf("%d", &a);

    printf("Enter the value of side b: ");
    scanf("%d", &b);
    printf("\n");
    printf("\t You have entered %d for side a and %d for side b\n", a, b);

    area = a*b;
    printf("\t The area of your rectangle is : %d \n", area);

    return 0;
}
```

## CONSTANTS

C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed during program execution.

The const keyword is used to declare a constant, as shown below:

```
int const A = 1;
const int A =2;
```

These fixed values are also called **literals**.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

## TYPE CASTING

Type casting is a way to convert a variable from one data type to another. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the **cast operator** as follows:

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

```
#include <stdio.h>

main()
{
    int sum = 17, count = 5;
    double mean;

    mean = (double) sum / count;
    printf("Value of mean is %d \n", mean );
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

**Type conversions** can be implicit which is performed by the compiler automatically, or it can be specified **explicitly** through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

## C PROGRAMMING OPERATORS

Operator is the symbol which operates on a value or a variable (operand). For example: + is an operator to perform addition.

C programming language has a wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

### OPERATORS IN C PROGRAMMING

1. Arithmetic Operators
2. Increment and Decrement Operators
3. Assignment Operators
4. Relational Operators
5. Logical Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

## ARITHMETIC OPERATORS

Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator - remainder of after an integer division	B % A will give 0

**Note:** % operator can only be used with integers.

## INCREMENT AND DECREMENT OPERATORS – Unary Operators

In C, ++ and -- are called increment and decrement operators respectively. Both of these operators are **unary operators**, i.e, used on single operand. ++ adds 1 to operand and -- subtracts 1 to operand respectively. For example:

```
Let a=5
a++; //a becomes 6
a--; //a becomes 5
++a; //a becomes 6
--a; //a becomes 5
```

### Difference between ++ and -- operator as postfix and prefix

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then increment it. This can be demonstrated by an example:

```
#include <stdio.h>
int main(){
    int c=2;
    printf("%d\n",c++); /*this statement displays 2 then,
                        only c incremented by 1 to 3.*/
    printf("%d",++c);   /*this statement increments 1 to
                        c then, only c is displayed.*/
    return 0;
}
```

### Output

```
2
4
```

## ASSIGNMENT OPERATORS – Binary Operators

The most common assignment operator is =. This operator assigns the value in the right side to the left side. For example:

```
var=5 //5 is assigned to var
a=c;  //value of c is assigned to a
5=c;  // Error! 5 is a constant.
```

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b

Operator	Example	Same as
/=	a/=b	a=a/b
%=	a%=b	a=a%b

NB/ += means Add and Assign etc.

## RELATIONAL OPERATORS - Binary Operators

Relational operators check relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0. For example:

$a > b$

Here,  $>$  is a relational operator. If  $a$  is greater than  $b$ ,  $a > b$  returns 1 if not then, it returns 0.

Relational operators are used in decision making and loops in C programming.

Operator	Meaning of Operator	Example
==	Equal to	$5 == 3$ returns false (0)
>	Greater than	$5 > 3$ returns true (1)
<	Less than	$5 < 3$ returns false (0)
!=	Not equal to	$5 != 3$ returns true (1)
>=	Greater than or equal to	$5 >= 3$ returns true (1)
<=	Less than or equal to	$5 <= 3$ return false (0)

## LOGICAL OPERATORS - Binary Operators

Logical operators are used to combine expressions containing relational operators. In C, there are 3 logical operators:

Operator	Meaning of Operator	Example
&&	Logical AND	If $c=5$ and $d=2$ then, $((c == 5) \&\& (d > 5))$ returns false.
	Logical OR	If $c=5$ and $d=2$ then, $((c == 5)    (d > 5))$ returns true.
!	Logical NOT	If $c=5$ then, $!(c == 5)$ returns false.

The following table shows the result of operator && evaluating the expression  $a \&\& b$ :

&& OPERATOR (and)		
a	b	$a \&\& b$
true	true	true
true	false	false
false	true	false
false	false	false

The operator `||` corresponds to the Boolean logical operation OR, which yields true if either of its operands is true, thus being false only when both operands are false. Here are the possible results of `a || b`:

OPERATOR (or)		
a	b	a    b
true	true	true
true	false	true
false	true	true
false	false	false

### Explanation

For expression, `((c==5) && (d>5))` to be true, both `c==5` and `d>5` should be true but, `(d>5)` is false in the given example. So, the expression is false. For expression `((c==5) || (d>5))` to be true, either the expression should be true.

Since, `(c==5)` is true. So, the expression is true. Since, expression `(c==5)` is true, `!(c==5)` is false.

## CONDITIONAL OPERATOR – Ternary Operators

Conditional operator takes three operands and consists of two symbols `?` and `:`. Conditional operators are used for decision making in C. For example:

```
c = (c > 0) ? 10 : -10;
```

If `c` is greater than 0, value of `c` will be 10 but, if `c` is less than 0, value of `c` will be -10.

## BITWISE OPERATORS

Bitwise operators work on bits and performs bit-by-bit operation.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A   B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

## PRECEDENCE OF OPERATORS

If more than one operator is involved in an expression then, C language has a **predefined rule of priority of operators**. This rule of priority of operators is called **operator precedence**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. **Within an expression, higher precedence operators will be evaluated first.**

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

## ASSOCIATIVITY OF OPERATORS

Associativity indicates in which order two operators of same precedence (priority) executes. Let us suppose an expression:

a = b != c

Here, operators == and != have the same precedence. The associativity of both == and != is **left to right**, i.e., the expression in left is executed first and execution take place towards right. Thus, a==b!=c equivalent to :

(a==b)!=c

Operators may be **left-associative** (meaning the operations are grouped from the left), **right-associative** (meaning the operations are grouped from the right)

Operator Name	Associativity	Operators
Unary	right to left	++ -- ! ~
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise AND	left to right	&
Bitwise Exclusive OR	left to right	^
Bitwise Inclusive OR	left to right	
Logical AND	left to right	&&
Logical OR	left to right	
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^=  =
Comma	left to right	,

## Chapter 4

# CONTROL STRUCTURES

### Definition

**Control structures** represent the forms by which statements in a program are executed. **Flow of control** refers to the order in which the individual statements, instructions or function calls of a program are executed or evaluated.

## IMPORTANCE OF CONTROL STRUCTURES

Generally, a program should execute the statements one by one until the defined end. This type of a program structure is called **sequential structure**. The functionality of this type of program is limited since it flows in a single direction. However, all high-level programming languages enable the programmer to change the flow of program execution. This is done by the use of control structures whose main benefits are to enable **decision making** and **repetition** as well as giving the power to do far more **complex processing** and **provide flexibility with logic**. The sophisticated logic is necessary for a program to solve complex problems.

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- continuation at a different statement i.e. unconditional jump e.g. GoTo statements
- executing a set of statements only if some condition is met i.e. choice
- executing a set of statements zero or more times, until some condition is met i.e. loop
- executing a set of distant statements, after which the flow of control usually returns e.g. subroutines/functions

## TYPES OF CONTROL STRUCTURES

There are three types in C:

### 1. Sequence structures

Program statements are executed in the sequence in which they appear in the program.

### 2. Selection structures/Decision Structures

Statement block is executed only if some condition is met. These structures include **if**, **if/else**, and **switch**. Selection structures are extensively used in programming because they allow the program to decide an action based upon user's input or other processes for instance in password checking.

### 3. Repetition/Iterative structures

This is where a group of statements in a program may have to be executed repeatedly until some condition is satisfied. These include **while**, **do/while** and **for**



# SELECTION STRUCTURES

## (a) THE IF SELECTION STRUCTURE

- Used to choose among alternative courses of action i.e. the if statement provides a junction at which the program has to select which path to follow. The **if selection** performs an action only if the condition is **true**,

General form

*If (expression)*  
*statement*

**Pseudocode:**

*If student's marks is greater than or equal to 600*  
*Print "Passed"*

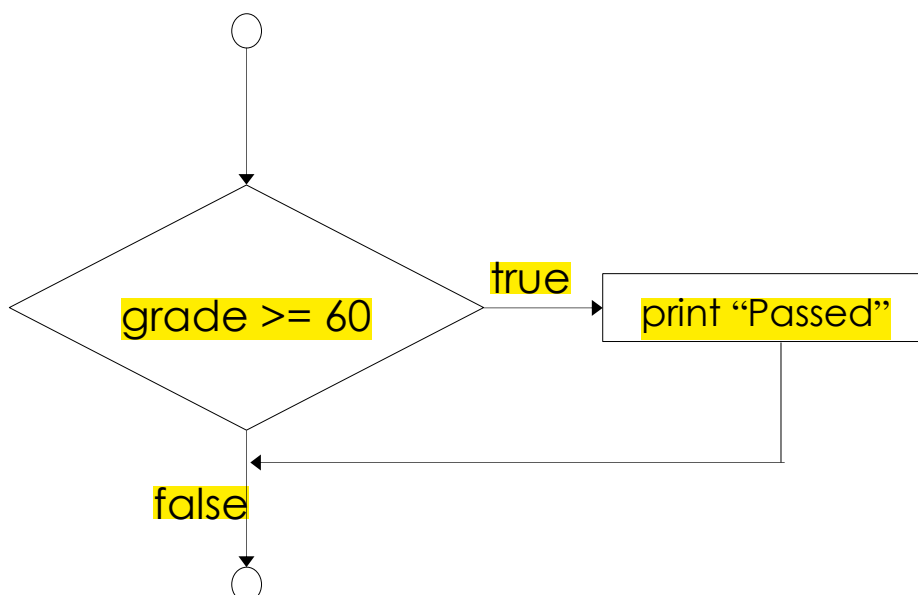
As in

```
if (marks >= 600)
    printf("Passed");
```

If condition is **true**

- Print statement executed and program goes on to next statement
- If **false**, print statement is ignored and the program goes onto the next statement

**NB/ Indenting makes programs easier to read**



Flow chart for the **if** selection structure

NB/ The statement in the if structure can be a single statement or a block (Compound statement). If it's a block of statements, it must be marked off by braces.

```

if (expression)
{
    Block of statements
}

```

As in

```

If (salary > 5000)
{
    tax_amount = salary * 1.5;
    printf("Tax charged is %f", tax_amount);
}

```

## (b) THE IF/ELSE

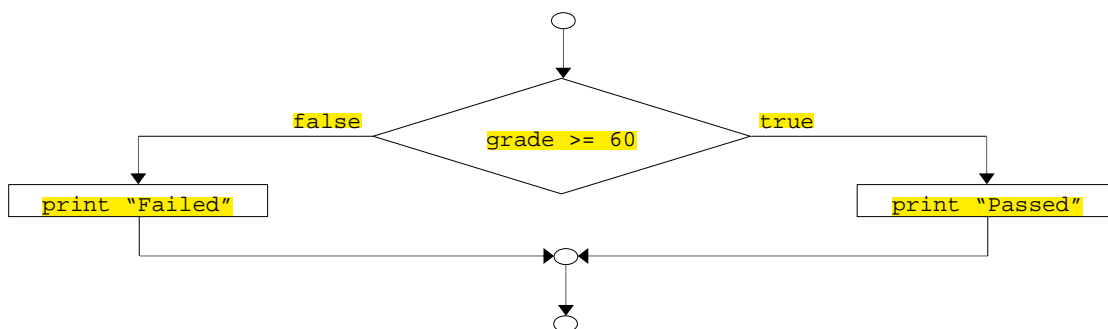
While **if** only performs an action if the condition is **true**, **if/else** specifies an action to be performed both when the condition is **true** and when it is **false**. E.g.

### Pseudocode:

```

If student's grade is greater than or equal to 60
Print "Passed"
else
Print "Failed"

```



Flow chart for the **if/else** selection structure

### Example

```

if (x >= 100)
{
    printf("Let us increment x:\n");
    x++;
}

else

    printf("x < 0 \n");

```

## (c) THE IF...ELSE IF...ELSE STATEMENT

- Test for multiple cases/conditions.
- Once a condition is met, the other statements are skipped
- Deep indentation usually not used in practice

### Pseudocode for an if..else if..else structure

```

If student's grade is greater than or equal to 90
    Print "A"
Else If student's grade is greater than or equal to 80
    Print "B"
else If student's grade is greater than or equal to 70
    Print "C"
    else If student's grade is greater than or equal to 60
        Print "D"
    else
        Print "F"

```

### Example

```

#include <stdio.h>

main()
{
    int marks;
    printf("Please enter your MARKS:");
    scanf("%d", &marks);

    if (marks>=90 && marks <=100)
        printf("Your grade is A\n");

    else if (marks>=80 && marks <=89)
        printf("Your grade is B\n");

    else if (marks>=70 && marks <=79)
        printf("Your grade is C\n");

    else if (marks>=60 && marks <=69)
        printf("Your grade is D\n");
    else if (marks >100)
        printf("Marks out of range\n");
    else
        printf("Your grade is F\n");
}

```

### (d) NESTED IF STATEMENTS

One **if** or **else if** statement can be used inside another **if** or **else if** statement(s).

#### Syntax

The syntax for a **nested if** statement is as follows:

```

if (boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2)

```

```

    {
    /* Executes when the boolean expression 2 is true */
    }
}

```

You can nest else **if...else** in the similar way as you have **nested if** statement.

Example

```

#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
/* check the boolean condition */
if( a == 100 )
{
    /* if condition is true then check the following */
    if( b == 200 )
    {
        /* if condition is true then print the following */
        printf("Value of a is 100 and b is 200\n" );
    }
}
return 0;
}

```

When the above code is compiled and executed, it produces the following result:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

## (e) SWITCH STATEMENT

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

### Syntax

The syntax for a switch statement in C programming language is as follows:

```

switch(expression)
{
case constant-expression
statement(s);
break;
case constant-expression :
statement(s);
break;
/* you can have any number of case statements */
default :
statement(s);
}

```

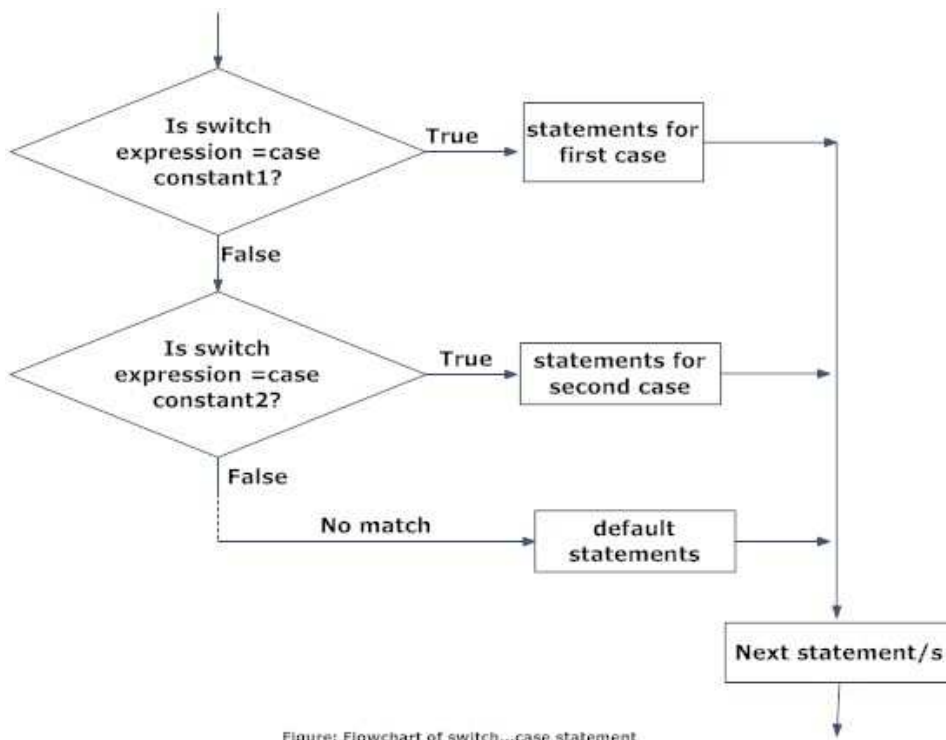


Figure: Flowchart of switch...case statement

### The following rules apply to a switch statement:

- 1) You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- 2) The constant-expression for a case must be the same data type as the variable in the switch
- 3) When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- 4) When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- 5) Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- 6) A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

### Sample Switch Statement

```

#include<stdio.h>
void main()
{
    char grade;

    printf("Enter your grade:");
    scanf("%c", &grade);

    switch (grade)
    {
        case 'A':
            printf("Excellent!\n");
            break;
        case 'B':
            printf("Very Good!\n");
    }
}
  
```

```

        break;
case 'C':
    printf("Good!\n");
    break;
case 'D':
    printf("Work harder!\n");
    break;
default:
    printf("Fail!\n");
}
}

```

## (f) NESTED SWITCH STATEMENTS

It is possible to have a **switch** as part of the statement sequence of an **outer switch**. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

### Syntax

The syntax for a **nested switch** statement is as follows:

```

switch(ch1) {
case 'A':
    printf("This A is part of outer switch" );
    switch(ch2) {
    case 'A':
        printf("This A is part of inner switch" );
        break;
    case 'B':
    }
    break;
case 'B':
}

```

### Example

```

#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
switch(a) {
case 100:
    printf("This is part of outer switch\n", a );
    switch(b) {
    case 200:
        printf("This is part of inner switch\n", a );
        printf("A is equals to %d and B is equals to %d", a, b);
    }
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
}

```

}

When the above code is compiled and executed, it produces the following result:

This is part of outer switch

This is part of inner switch

A is equals to 100 and B is equals to 200

Exact value of a is : 100

Exact value of b is : 200

## REPETITION/ITERATIVE/LOOP STRUCTURES

A loop statement allows the execution of a statement or a group of statements multiple times until a condition either tests true or false. There are two types of loops: **Pre-test** and **post-test** loops.

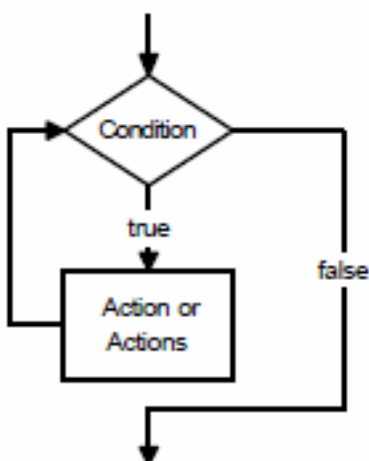
In a pretest loop, a logical condition is checked before each repetition to determine if the loop should terminate. These loops include:

– *while loop*

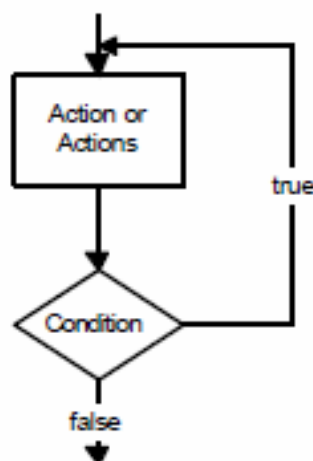
– *for loop*

Post-test loops check a logical condition after each repetition for termination. The do-while loop is a *post-test loop*.

Pretest Loop



Posttest Loop



### (a) WHILE LOOP IN C

A **while** loop statement repeatedly executes a target statement **as long as** a given condition is **true**.

The syntax of a while loop in C programming language is:

```
while(condition)
{
    statement(s);
    update expression
}
```

The statement(s) may be a single statement or a block of statements. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

## Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10; //loop index

/* while loop execution */
while( a < 20 )
{
printf("value of a: %d\n", a);
a++;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

## (b) FOR LOOP IN C

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

### Syntax

The syntax of a **for** loop in C programming language is:

```
for ( initial expression; test expression/logical condition; update
expression )
{
statement(s);
}
```

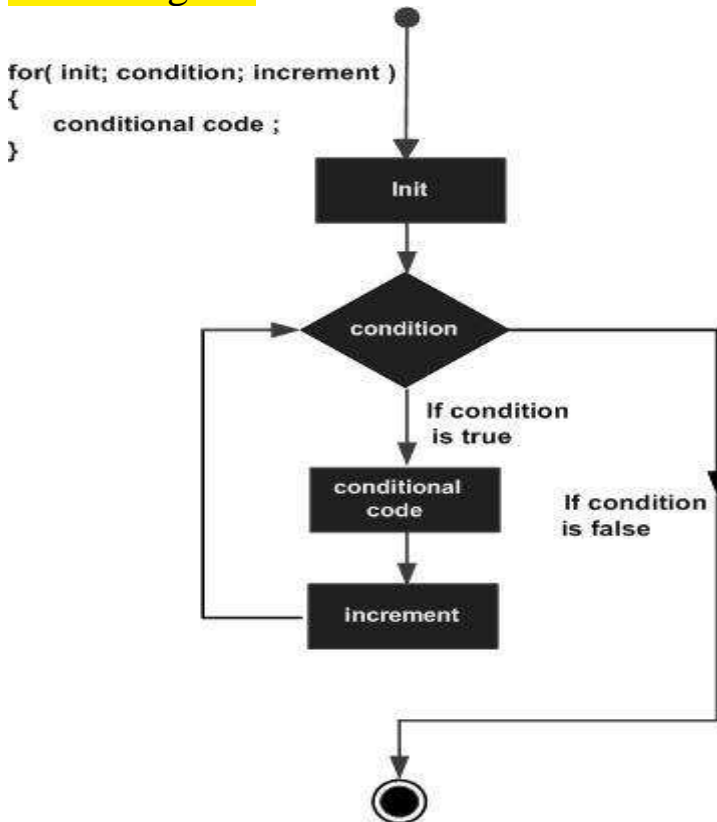
Here is the flow of control in a for loop:

1. This step initializes any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.



3. After the body of the for loop executes, the flow of control jumps back up to the update expression. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the for loop terminates.

## Flow Diagram



## Example

```
#include <stdio.h>
int main ()
{
    int a;//loop index
    /* for loop execution */
    for(a = 10; a < 20; a++)
    {
        printf("value of a: %d\n", a);
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
```

value of a: 17  
value of a: 18  
value of a: 19

## (c) DO...WHILE LOOP IN C

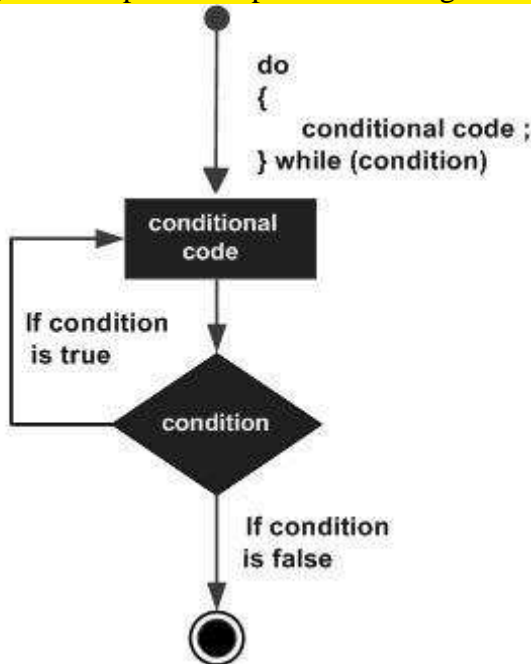
Unlike for and while loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a **do...while** loop is guaranteed to execute at least one time. The structure, therefore loops until a condition tests false i.e. loop until.

### Syntax

```
do  
{  
statement(s);  
}while( condition );
```

If the condition is **true**, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes **false**.



### Example

```
#include <stdio.h>  
int main ()  
{  
/* local variable definition */  
int a = 10;  
/* do loop execution */  
do  
{  
printf("value of a: %d\n", a);  
a = a + 1;  
}while( a < 20 );  
return
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

#### **(d) NESTED LOOPS IN C**

C programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

##### **Syntax**

The syntax for a nested for loop statement in C is as follows:

```
for ( init; condition; increment )
{
    for ( init; condition; increment )
    {
        statement(s);
    }
    statement(s);
}
```

The syntax for a nested while loop statement in C programming language is as follows:

```
while(condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s);
}
```

The syntax for a nested do...while loop statement in C programming language is as follows:

```
do
{
    statement(s);
    do
    {
        statement(s);
    }while( condition );
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a **for loop** can be inside a while loop or vice versa.

## Example

```
#include <stdio.h>

int main()
{
    int n, c, k;

    printf("Enter number of rows:");
    scanf("%d",&n);

    for ( c = 1 ; c <= n ; c++ )
    {
        for( k = 1 ; k <= c ; k++ )
        {
            printf("%d",k);
        }
        printf("\n");
    }

    return 0;
}
```

Result:

If the user entered 5 as the number of rows, the output would be:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

## TERMINATING LOOPS

- Counter-controlled loops - a loop controlled by a counter variable, generally where the number of times the loop will execute is known ahead of time especially in *for loops*.
- Event-controlled loops - loops where termination depends on an event rather than executing a fixed number of times for example when a zero value is keyed in or search through data until an item is found. Used mostly in while loops and do-while loops.

### Using a Sentinel

- The value -999 is sometimes referred to as a sentinel value. The value serves as the “guardian” for the termination of the loop. It is a good idea to make the sentinel a constant:

```
#define STOPNUMBER -999
while (number != STOPNUMBER) ...
```

# BRANCHING STATEMENTS

## (a) BREAK STATEMENT IN C

The **break** statement has the following two uses:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement.
3. If you are using **nested loops** (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
do
{
if( a == 15)
{
/* skip the iteration */
break;
}
printf("value of a: %d\n", a);
a++;
}while( a < 20 );
return 0;
}
```

## (b) CONTINUE STATEMENT IN C

The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between. For the **for loop**, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

## Example

```
//program to demonstrate the working of continue statement in C programming
# include <stdio.h>
int main(){
    int i,num,product;
    for(i=1,product=1;i<=4;++i){
        printf("Enter num%d:",i);
        scanf("%d",&num);
        if(num==0)
            continue; /*In this program, when num equals to zero, it skips
the statement product*=num and continue the loop. */
        product*=num;
    }
    printf("product=%d",product);
}
```

```
return 0;
}value of a: 19
```

## (c) GOTO STATEMENT IN C

A **goto** statement provides an unconditional jump from the **goto** to a labeled statement in the same function.

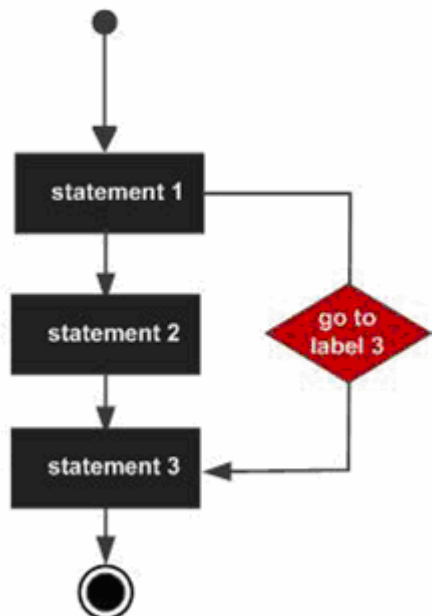
**NOTE:** Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **goto** can be rewritten so that it doesn't need the **goto**.

## Syntax

The syntax for a **goto** statement in C is as follows:

```
goto label;
```

```
--
-
```



## Example

```
#include <stdio.h>
int main ()
{
    /* for loop execution */
    int a,userinput,sum=0;

    for(a = 0; a < 5;a++)
    {
        printf("Enter a number: ");
        scanf("%d",&userinput);
        if (userinput <1)
            goto jump;

        sum+=userinput;
    }

    jump:
}
```

```
printf("The sum of the values is %d\n", sum);  
return 0;  
}
```

## (d) THE RETURN STATEMENT

The last of the branching statements is the `return` statement. The `return` statement exits from the current function, and control flow returns to where the function was invoked. The `return` statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the `return` keyword.

```
return count;
```

The data type of the returned value must match the type of the method's declared return value. When a function is declared `void`, use the form of `return` that doesn't return a value.

```
return;
```

## THE INFINITE LOOP

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>  
int main ()  
{  
for( ; ; )  
{  
printf("This loop will run forever.\n");  
}  
return 0;  
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.

**NOTE:** You can terminate an infinite loop by pressing **Ctrl + C** keys.

## CHAPTER 5

# SUBPROGRAMS IN C

A sub-program is a series of C statements that perform a specific task in a program. A subprogram can be called within another procedure. Every C program has at least one function, which is **main()**. A C program can be divided up into separate functions.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. The C standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc. However, a **function** returns a value while a **procedure** doesn't: it just executes commands.

A Subprogram is:

- a part of a program that performs one or more related tasks
- has its own name
- written as an independent part of the program

Benefits of using sub-procedures in programming are:

- Sub-programs help to break programs into several but logical sections. The smaller programs/routines make **programming, debugging** and subsequent **maintenance** easier.
- They also help in **coding repeated operations** such as frequently used calculations, text etc thus making programming less repetitive and faster.
- Procedures used in one program can act as building blocks for other programs with slight modifications i.e. **code re-use**.
- Programmers working on large projects can divide the workload by making different functions.

## TYPES OF FUNCTIONS

Basically, there are two types of functions in C on basis of whether it is defined by user or not.

- Library function
- User defined function

### LIBRARY FUNCTION

Library functions are the in-built function in C programming system. For example:



## **printf()**

- **printf()** is used for displaying output in C.

## **scanf()**

- **scanf()** is used for taking input in C.

## **USER DEFINED FUNCTION**

C allows programmers to define their own function according to their requirements known as user defined functions.

### **How user-defined function works in C Programming?**

```
#include <stdio.h>
void function_name(){
    .....
    .....
}
int main(){
    .....
    function_name();
    .....
}
```

As mentioned earlier, every C program begins from `main()` and program starts executing the codes inside `main()` function. When the control of program reaches to `function_name()` inside `main()` function. The control of program jumps to `void function_name()` and executes the codes inside it. When, all the codes inside that user-defined function are executed, control of the program jumps to the statement just after `function_name()` from where it is called. Analyze the figure below for understanding the concept of function in C programming.

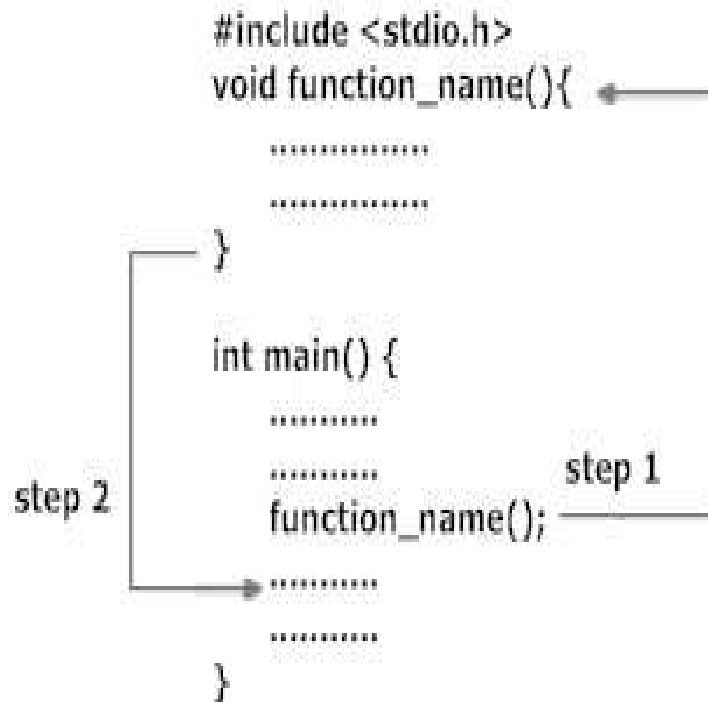


Fig: Working of Functions

Remember, the function name is an identifier and should be unique.

## DEFINING A FUNCTION

The general form of a function definition in C programming language is as follows:

```

return_type function_name ( parameter list )
{
    body of the function
}

```

A **function definition** in C programming language consists of a function header and a function body. Here are all the parts of a function:

1. **Return Type:** A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return\_type** is the keyword **void**.
2. **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the **function signature**.
3. **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the **formal parameter**. This value is referred to as **actual parameter** or **argument**. The **parameter list** refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
4. **Function Body:** The function body contains a collection of statements that define what the function does.

## Example

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */

int max(int num1, int num2)
{
    /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

## FUNCTION DECLARATIONS

A **function declaration** tells the compiler about a function name and how to call the function. The actual **body of the function** can be defined separately.

A function declaration has the following parts:

*return\_type function\_name( parameter list );*

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration; only their type is required, so the following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

## CALLING A FUNCTION

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. Therefore, the calling program is suspended during execution of the called subprogram.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```

#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
int ret;
/* calling a function to get max value */
ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
/* local variable declaration */
int result;
if (num1 > num2)
result = num1;
else
result = num2;

return result;
}

```

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
<b>Call by value</b>	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
<b>Call by reference</b>	This method copies the address of an argument into the formal 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.

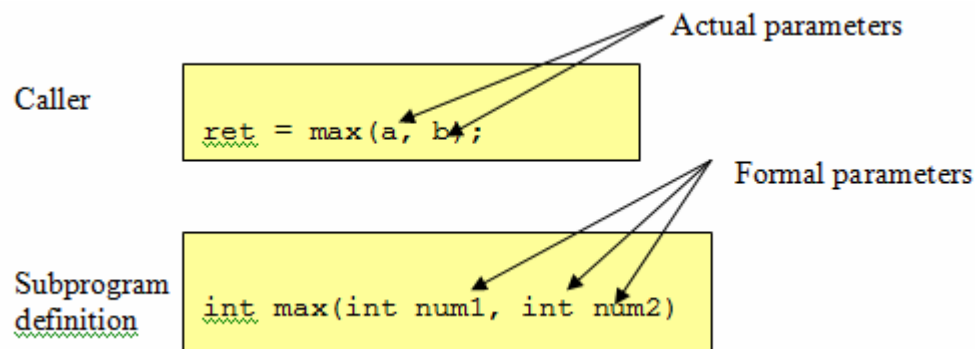
By default, C uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## FUNCTION ARGUMENTS

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram. An actual parameter represents a value used in the subprogram call statement.



When `max()` is called, we pass it the arguments which the function uses as the values of `ret`. This process is called **parameter passing**.

\*\*\*\*\*

*Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

## TYPES OF VARIABLES

The Programming language C has two main variable types

- Local Variables
- Global Variables

## LOCAL VARIABLES

**A local variable is a variable that is declared inside a function.**

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

## GLOBAL VARIABLES

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it. Global variables are declared outside **all** functions.

### **Sample Program.**

```
#include <stdio.h>
int area; //global variable
int main ()
{
    int a, b; //local variable

    /* actual initialization */
    a = 10;
    b = 20;
    printf ("\t Side a is %d cm and side b is %d cm long\n", a, b);
    area = a*b;
    printf ("\t The area of your rectangle is : %d \n", area);
    return 0;
}
```

## **EXERCISES**

- 1. Write a C program to add two integers. Define a function add to add integers and display sum in main() function.**

```
//main function
#include<stdio.h>
int add(int a, int b);
int main()
{
    int a, b, sum;

    printf ("Enter the first integer:");
    scanf ("%d", &a);

    printf("Enter the second integer:");
    scanf("%d", &b);

    sum = add(a,b);
    printf("The sum of the two numbers is %d\n", result);

}

//start of second function
int add(int a,int b)
{
    int result;
    result = a+b;
    return result;
}
```

2. Write a C program– *max()*– to determine the greater of two integers. Call the function from *main()* and supply it with two integers and then display the greater of the two.

```
#include <stdio.h>

/* function declaration */
int max(int, int);

int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
    ret = max(a, b);

    printf( "Max value is : %d\n", ret );

    return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

## CHAPTER 6

# DATA STRUCTURES

These refer to **groups of data elements that are organized in a single unit so that they can be used more efficiently** as compared to the simple data types such as integers and strings. An example of a data structure is the array. Ordinary variables store one value at a time while an array will store more than one value at a time in a single variable name.

Data structures are important for grouping sets of similar data together and passing them as one. For example, if you have a method that prints a set of data but you don't know when writing the procedure how large that set is going to be, you could use an array to pass the data to that method and loop through it.

Data structures can be **classified** using various criteria.

### **a) Linear**

In linear data structures, values are arranged in linear fashion. A linear data structure traverses the data elements sequentially. The elements in the structure are adjacent to one another and every element has exactly two neighbour elements to which it is connected. Arrays, linked lists, stacks and queues are examples of linear data structures.

### **b) Non-Linear**

The data values in this structure are not arranged in order but every data item is attached to several other data items in a way that is specific for reflecting relationships. Tree, graph, table and sets are examples of non-linear data structures.

### **c) Homogenous**

In this type of data structures, values of the same types of data are stored, as in an array.

### **d) Non-homogenous**

In this type of data structures, data values of different types are grouped, as in structures and classes.

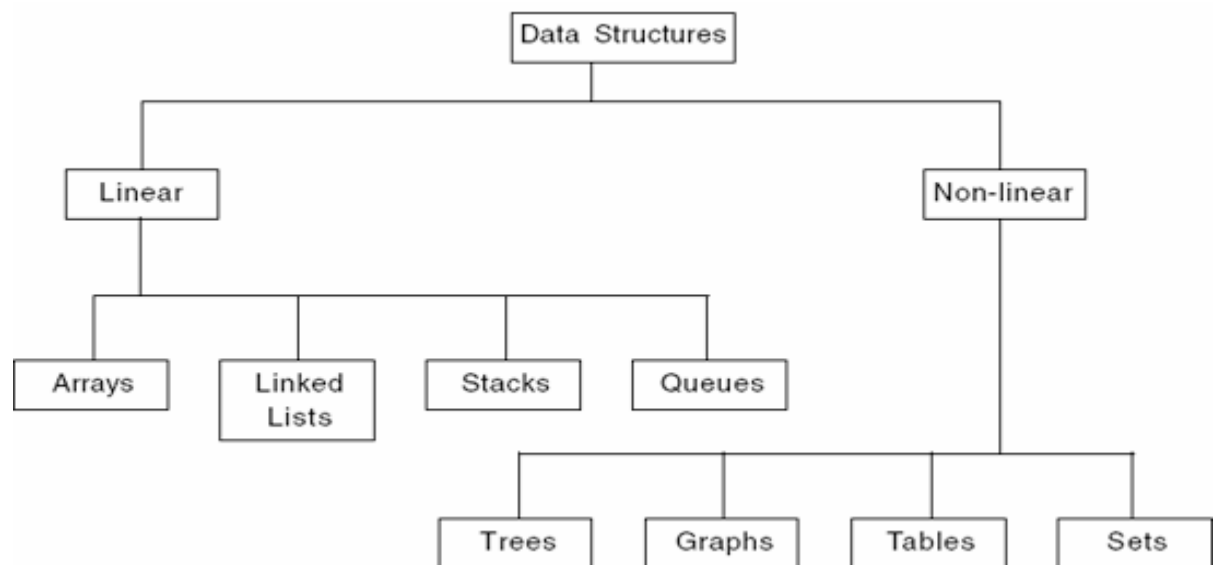
### **e) Dynamic**

In dynamic data structures such as references and pointers, size and memory locations can be changed during program execution. These data structures can grow and shrink during execution.

### **f) Static**

With a static data structure, the size of the structure is fixed. Static data structures such as arrays are very good for storing a well-defined number of data items.



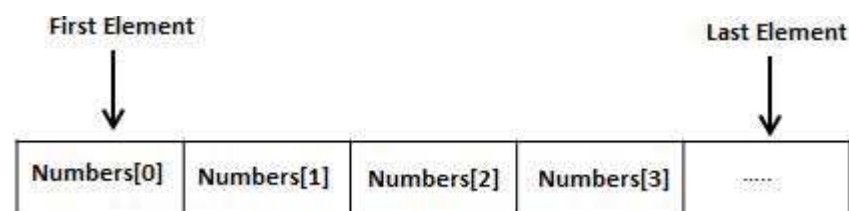


## ARRAYS

An array is a named list of elements, all with the same data type. It is better defined as a consecutive group of memory locations all of which have the same name and the same data type. Arrays store a fixed-size sequential collection of elements of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

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



## DECLARING ARRAYS

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

`type arrayName [ arraySize ];`

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type **double**, use this statement:

`double balance[10];`

Now *balance* is a variable array which is sufficient to hold up to 10 double numbers.

## INITIALIZING ARRAYS

You can initialize an array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

## ACCESSING ARRAY ELEMENTS

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

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

```
int main ( )  
{
```

```
    int n[ 10 ]; /* n is an array of 10 integers */  
    int i, j;
```

```

/* initialize elements of array n to 0 */
for ( i = 0; i < 10; i++ )
{
    n[ i ] = i + 100; /* set element at location i to i
+ 100 */
}

/* output each array element's value */
for ( j = 0; j < 10; j++ )
{
    printf("Element[%d] = %d\n", j, n[j] );
}

return 0;
}

```

When the above code is compiled and executed, it produces the following result:

```

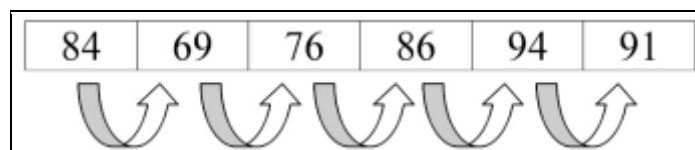
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

## SORT TECHNIQUES

### Bubble Sort

In the **bubble sort**, as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares **adjacent elements** of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order.



When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again.

The table below follows an array of numbers before, during, and after a bubble sort

for *descending* order. A "pass" is defined as one full trip through the array comparing and if necessary, swapping, **adjacent** elements. Several passes have to be made through the array before it is finally sorted

<b>Array at beginning:</b>	84	69	76	86	94	91
<b>After Pass #1:</b>	84	76	86	94	91	69
<b>After Pass #2:</b>	84	86	94	91	76	69
<b>After Pass #3:</b>	86	94	91	84	76	69
<b>After Pass #4:</b>	94	91	86	84	76	69
<b>After Pass #5 (done):</b>	94	91	86	84	76	69

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished. In actuality, the process is finished before this last pass is made.

### // Bubble Sort Function for Descending Order

```
#include<stdio.h>
main()
{
    int control , control2, marks, total=0, temp;float meanmark;
    int allmarks[5];

    for (control = 0; control <= 4; control++)
    {
        printf("Please enter student's marks:");
        scanf("%d", & marks);
        allmarks[control]=marks;
        total = total + marks;
    }
    meanmark = (float) total/control;

    for (control = 0; control < 4; control++) {
        for (control2 = 0; control2 < 4; control2++) {
            if (allmarks[control2] > allmarks[control2+1])
            {
                temp = allmarks[control2];
                allmarks[control2]= allmarks[control2+1];
                allmarks[control2+1] = temp;
            }
        }
    }

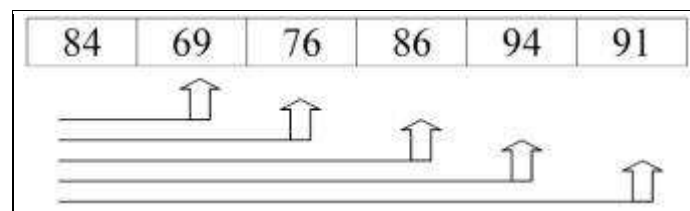
    printf("\nThe sorted list of marks is:\n");

    for (control=0; control<=4;control++)
    {
        printf("%d\n", allmarks[control]);
    }
}
```

```
printf("\nThe total marks is %d\n", total);
printf("Mean marks is %f\n", meanmark);
}
```

## Exchange Sort

The **exchange sort** is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are not in their proper positions. (Some people refer to the "exchange sort" as a "bubble sort".) The difference between these two sorts is the manner in which they compare the elements. **The exchange sort compares the first element with each following element of the array, making any necessary swaps.**



When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order. This sorting process continues until the entire array is ordered.

Let's examine our same table of elements again using an exchange sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

<b>Array at beginning:</b>	84	69	76	86	94	91
<b>After Pass #1:</b>	94	69	76	84	86	91
<b>After Pass #2:</b>	94	91	69	76	84	86
<b>After Pass #3:</b>	94	91	86	69	76	84
<b>After Pass #4:</b>	94	91	86	84	69	76
<b>After Pass #5 (done):</b>	94	91	86	84	76	69

The exchange sort, in some situations, is slightly more efficient than the bubble sort. It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.

## //Exchange Sort Function for Descending Order

```
#include <stdio.h>
void main()
```

```

{
    int i, j;
    int temp; // holding variable
    int num[5];

    //intialize array
    for(i =0; i<=4; i++){
        printf("Enter a number:");
        scanf("%d",&num[i]);
    }
    //sort array
    for (i=0; i< (4); i++) // element to be compared
    {
        for(j = (i+1); j < 5; j++) // rest of the elements
        {
            if (num[i] < num[j]) // descending order
            {
                temp= num[i]; // swap
                num[i] = num[j];
                num[j] = temp;
            }
        }
    }

    //print sorted array
    printf("\nSorted array:\n");

    for(i =0; i<=4; i++){
        printf("\t%d\n", num[i]);
    }

    return;
}

```

## Selection Sort

The **selection sort** is a combination of searching and sorting.

**During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array.**

The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

Let's look at our same table of elements using a selection sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

<b>Array at beginning:</b>	84	69	76	86	94	91
<b>After Pass #1:</b>	84	91	76	86	94	69

<b>After Pass #2:</b>	84	91	94	86	76	69
<b>After Pass #3:</b>	86	91	94	84	76	69
<b>After Pass #4:</b>	94	91	86	84	76	69
<b>After Pass #5 (done):</b>	94	91	86	84	76	69

While being an easy sort to program, the selection sort is one of the least efficient. The algorithm offers no way to end the sort early, even if it begins with an already sorted list.

## // Selection Sort Function for Descending Order

```
void main()
{
    int i, j, first, temp;
    int num[5]
    for (i= 4; i > 0; i--)
    {
        first = 0;           // initialize to subscript of first element
        for (j=1; j<=i; j++) // locate smallest between positions 1 and i.
        {
            if (num[j] < num[first])
                first = j;
        }
        temp = num[first]; // Swap smallest found with element in position i.
        num[first] = num[i];
        num[i] = temp;
    }
    return;
}
```

## Shell Sort

The **shell sort** is named after its inventor D. L. Shell. Instead of comparing adjacent elements, like the bubble sort, the shell sort repeatedly compares elements that are a certain distance away from each other ( $d$  represents this distance). The value of  $d$  starts out as half the input size and is halved after each pass through the array. The elements are compared and swapped when needed. The equation  $d = (N + 1) / 2$  is used. Notice that only integer values are used for  $d$  since integer division is occurring.

Let's look at our same list of values for descending order with the shell sort. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

<b>Array at beginning:</b>	84	69	76	86	94	91	$d$
<b>After Pass #1:</b>	86	94	91	84	69	76	3
<b>After Pass #2:</b>	91	94	86	84	69	76	2

<b>After Pass #3:</b>	94	91	86	84	76	69	1
<b>After Pass #4 (done):</b>	94	91	86	84	76	69	1

**First Pass:**  $d = (6 + 1) / 2 = 3$ . Compare 1st and 4th, 2nd and 5th, and 3rd and 6th items since they are 3 positions away from each other))

**Second Pass:** value for  $d$  is halved  $d = (3 + 1) / 2 = 2$ . Compare items two places away such as 1st and 3rd .....

**Third Pass:** value for  $d$  is halved  $d = (2 + 1) / 2 = 1$ . Compare items one place away such as 1st and 2nd .....

**Last Pass:** sort continues until  $d = 1$  and the pass occurs without any swaps.

This sorting process, with its comparison model, is an efficient sorting algorithm.

## //Shell Sort Function for Descending Order

```
void main()
{
    int I,d , temp, length[5];
    while( (d > 1))    // boolean flag (true when not equal to 0)
    {
        d = (d+1) / 2;
        for (i = 0; i < (5 - d); i++)
        {
            if (num[i + d] > num[i])
            {
                temp = num[i + d];    // swap positions i+d and i
                num[i + d] = num[i];
                num[i] = temp;
                flag = 1;              // tells swap has occurred
            }
        }
    }
    return;
}
```

## Quick Sort

The **quicksort** is considered to be very efficient, with its "divide and conquer" algorithm. This sort starts by dividing the original array into two sections (partitions) based upon the value of the first element in the array. Since our example sorts into descending order, the first section will contain all the elements greater than the first element. The second section will contain elements less than (or equal to) the first element. It is possible for the first element to end up in either section.

Let's examine our same example

<b>Array at beginning:</b>	84	69	76	86	94	91
<b>= 1st partition</b>	86	94	91	84	69	76
<b>= 2nd partition</b>	94	91	86	84	69	76
	94	91	86	84	69	76



	94	91	86	84	69	76
<b>Done:</b>	94	91	86	84	76	69

This sort uses recursion - the process of "calling itself". Recursion will be studied at a later date.

## //Quick Sort Functions for Descending Order

### // (2 Functions)

```
void main()
{
    // top = subscript of beginning of array
    // bottom = subscript of end of array

    int middle;
    if (top < bottom)
    {
        middle = partition(num, top, bottom);
        quicksort(num, top, middle); // sort first section
        quicksort(num, middle+1, bottom); // sort second section
    }
    return;
}
```

## //Function to determine the partitions

// partitions the array and returns the middle subscript

```
int main()
{
    int x = array[top];
    int i = top - 1;
    int j = bottom + 1;
    int temp;
    do
    {
        do
        {
            j --;
        } while (x > array[j]);

        do
        {
            i ++;
        } while (x < array[i]);

        if (i < j)
        {
            temp = array[i];
            array[i] = array[j];
```

```

        array[j] = temp;
    }
    }while (i < j);
    return j;        // returns middle subscript
}

```

## Merge Sort

The **merge sort** combines two **sorted** arrays into one larger sorted array. As the diagram below shows, Array A and Array B merge to form Array C.

Arrays to be merged **MUST be SORTED FIRST!!**

Be sure to declare Array C in main( ) and establish its size.

### Example: Ascending Order

Array A: {7, 12}

Array B: {5, 7, 8}

Array C: {5, 7, 7, 8, 12} after merge

*Here is how it works:* The first element of array A is compared with the first element of array B. If the first element of array A is smaller than the first element of array B, the element from array A is moved to the new array C. The subscript of array A is now increased since the first element is now set and we move on.

If the element from array B should be smaller, it is moved to the new array C. The subscript of array B is increased. This process of comparing the elements in the two arrays continues until either array A or array B is empty. When one array is empty, any elements remaining in the other (non-empty) array are "pushed" into the end of array C and the merge is complete.

### //Function to merge two pre-sorted arrays

```

void main()
{
    int indexA = 0;    // initialize variables for the subscripts
    int indexB = 0;
    int indexC = 0;
    int arrayC[5];

    while((indexA < 5) && (indexB < 5))
    {

        if (arrayA[indexA] < arrayB[indexB])
        {
            arrayC[indexC] = arrayA[indexA];
            indexA++;    //increase the subscript
        }
        else
        {
            arrayC[indexC] = arrayB[indexB];
            indexB++;    //increase the subscript
        }
    }
}

```

```

    }
    indexC++;    //move to the next position in the new array
}
// Move remaining elements to end of new array when one merging array is empty
while (indexA < 5)
{
    arrayC[indexC] = arrayA[indexA];
    indexA++;
    indexC++;
}
while (indexB < 5)
{
    arrayC[indexC] = arrayB[indexB];
    indexB++;
    indexC++;
}
return;
}

```

## SEARCHING ARRAYS

When working with arrays, it is often necessary to perform a search or "lookup" to determine whether an array contains a value that matches a certain key value. The process of locating a particular element value in an array is called searching. There are two types of search mechanisms: **serial/linear search** and **binary search**.

### a) Serial Search

The technique used here is called a **serial search**, because the integer elements of the array are compared one by one to the user input being looked for (userValue) until either a match is found or all elements of the array are examined without finding a match.

In the code below, if a match is found, the text "There is a match" is printed on the form and the execution of the procedure is terminated (Exit Sub). If no match is found, the program exits the loop and prints the text "No match found".

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

```
int main()
```

```
{
```

```
    int array[5]={10,7,8,2,5}, searchvalue, c;
```

```
    printf("\nEnter the number to search: ");
```

```
    scanf("%d", &searchvalue);
```

```
    for (c = 0; c < 5; c++)
```

```
    {
```

```
        if (array[c] == searchvalue)    // if required element
found
```

```
    {
```

```

        printf("\n\t%d is present at location %d.\n",
searchvalue, c+1);
        break;
    }
}

if (c == 5) // if looped more than 5 times ie 6 times
    printf("\n\t%d is not present in the array.\n",
searchvalue);

return 0;
}

```

### **Binary Search**

Binary search uses the concept of splitting your searchable array in two, discarding the half that does not have the element for which you are looking.

You place your items in an array and sort them. Then you simply get the middle element and test if it is <, >, or = to the element for which you are searching. If it is less than, you discard the greater half, get the middle index of the remaining elements and do it again. Binary search divides your problem in half every time you execute your loop.

```

#include <stdio.h>

int main()
{
    int c, first, last, middle, n, search, array[100];

    printf("Enter number of elements\n");
    scanf("%d",&n);

    printf("Enter %d integers\n", n);

    for ( c = 0 ; c < n ; c++ )
        scanf("%d",&array[c]);

    printf("Enter value to find\n");
    scanf("%d",&search);

    first = 0;
    last = n - 1;
    middle = (first+last)/2;

    while( first <= last )
    {
        if ( array[middle] < search )
            first = middle + 1;
        else if ( array[middle] == search )
        {
            printf("%d found at location %d.\n", search, middle+1);
            break;
        }
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
}

```

```

    }
    if ( first > last )
        printf("Not found! %d is not present in the list.\n", search);

    return 0;
}

```

## LINKED LISTS

A linked list is a dynamic data structure whose length can be increased or decreased at run time.

How Linked lists are different from arrays? Consider the following points :

- An array is a static data structure. This means the length of array cannot be altered at run time. While, a linked list is a dynamic data structure.
- In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but still connected to each other.

When to prefer linked lists over arrays? Linked lists are preferred mostly when you don't know the volume of data to be stored. For example, In an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join. In scenarios like these, linked lists (or other dynamic data structures) are used as their capacity can be increased (or decreased) at run time (as and when required).

### How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Linked lists are connected through pointers.

## POINTERS

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk \* you used to declare a pointer is the

same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## HOW TO USE POINTERS?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator `*` that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
    int var = 20; /* actual variable declaration */
    int *ip;      /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

## NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()
{
    int *ptr = NULL;

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

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

The value of ptr is 0

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

## C STRINGS

In C, one or more characters enclosed between double quotes is called a string. C does not have built-in string data type. Instead, C supports strings using one-dimensional arrays. A string is defined as a *null terminated* array i.e. \0. This means that you must define the array that is going to hold a string to be one byte larger than the largest string it is going to hold, in order to make room for the null.

To read a string from the keyboard, you must use another of C's standard library functions, **gets( )**, which requires the **stdio.h** header file. The gets ( ) function reads characters until you press <ENTER>. The carriage return is not stored, but it is replaced by a null, which terminates the string. E.g.

```
#include<stdio.h>
Main ()
Char str [80];
Int I;
Printf (ËNter a string: \n");
```

```
gets(str);

for (I = 0; str[i]; i++)
printf("%c", str[i]);
}
```

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

## Initialization of strings

In C, string can be initialized in a different number of ways.

```
char c[] = "abcd";
OR,
char c[5] = "abcd";
OR,
char c[] = {'a', 'b', 'c', 'd', '\0'};
OR;
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

String can also be initialized using pointers

```
char *c = "abcd";
```

## Reading Strings from user.

### Reading words from user.

```
char c[20];
scanf("%s", c);
```

String variable *c* can only take a word. It is because when white space is encountered, the `scanf()` function terminates.

### Write a C program to illustrate how to read string from terminal.

```
#include <stdio.h>
int main(){
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```



## Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Here, program will ignore Ritchie because, `scanf()` function takes only string before the white space.

C supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	<b>strcpy(s1, s2);</b> Copies string s2 into string s1.
2	<b>strcat(s1, s2);</b> Concatenates string s2 onto the end of string s1.
3	<b>strlen(s1);</b> Returns the length of string s1.
4	<b>strcmp(s1, s2);</b> Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	<b>strchr(s1, ch);</b> Returns a pointer to the first occurrence of character ch in string s1.
6	<b>strstr(s1, s2);</b> Returns a pointer to the first occurrence of string s2 in string s1.

The C library function **int strcmp(const char \*str1, const char \*str2)** compares the string pointed to by **str1** to the string pointed to by **str2**.

Following is the declaration for `strcmp()` function.

```
int strcmp(str1, str2)
```

## PARAMETERS

- **str1** -- This is the first string to be compared.
- **str2** -- This is the second string to be compared.

## RETURN VALUE

This function returned values are as follows:

- if Return value if  $< 0$  then it indicates str1 is less than str2
- if Return value if  $> 0$  then it indicates str2 is less than str1
- if Return value if  $= 0$  then it indicates str1 is equal to str2

## Example

The following example shows the usage of strcmp() function.

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

int main ()
{
    char str1[15];
    char str2[15];
    int ret;

    strcpy(str1, "abcdef");
    strcpy(str2, "ABCDEF");

    ret = strcmp(str1, str2);

    if(ret > 0)
    {
        printf("str1 is less than str2");
    }
    else if(ret < 0)
    {
        printf("str2 is less than str1");
    }
    else
    {
        printf("str1 is equal to str2");
    }

    return(0);
}
```

## More Examples

### 1) C Program to Find the Length of a String

```
#include <stdio.h>
int main()
{
    char s[1000],i;
    printf("Enter a string: ");
    scanf("%s",s);
    for(i=0; s[i]!='\0'; ++i);
    printf("Length of string: %d",i);
    return 0;
}
```

### Output

```
Enter a string: Programiz
Length of string: 9
```

### 2) Code to Concatenate Two Strings Manually

```
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i, j;
```

```

printf("Enter first string: ");
scanf("%s",s1);
printf("Enter second string: ");
scanf("%s",s2);
for(i=0; s1[i]!='\0'; ++i); /* i contains length of string
s1. */
for(j=0; s2[j]!='\0'; ++j, ++i)
{
    s1[i]=s2[j];
}
s1[i]='\0';
printf("After concatenation: %s",s1);
return 0;
}

```

## Output

```

Enter first string: lol
Enter second string: :)
After concatenation: lol:)

```

## QUEUES

Queue is a specialized data storage structure (Abstract data type). Unlike arrays, access of elements in a Queue is restricted. It has two main operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue. It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

### Primitive operations

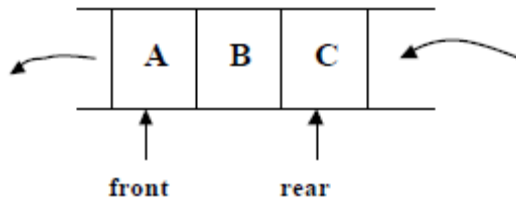
- a) enqueue (q, x): inserts item **x** at the rear of the queue **q**
- b) x = dequeue (q): removes the front element from **q** and returns its value.
- c) isEmpty(q) : true if the queue is empty, otherwise false.

### Example

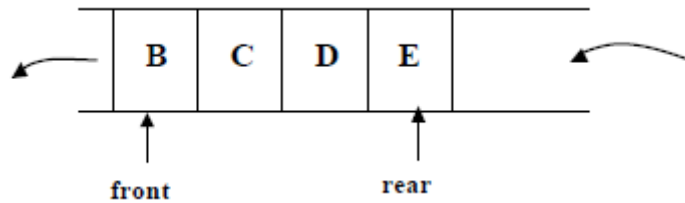
```

enqueue(q, 'A');
enqueue(q, 'B');
enqueue(q, 'C');
x = dequeue(q);
enqueue(q, 'D');
enqueue(q, 'E');

```



`x = dequeue (q) -> x = 'A'`



## STACKS

A stack is a data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack. Consequently, a stack is said to have "first in last out" behavior (or "last in, first out"). The first item added to a stack will be the last item removed from a stack.

## CHAPTER 7

# FILE HANDLING

This chapter explains how C programmers can create, open and close text or binary files for their data storage. A file represents a sequence of bytes, does not matter if it is a text file or binary file.

## OPENING FILES

You can use the **fopen()** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

```
"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"
```

## CLOSING A FILE

To close a file, use the **fclose()** function. The prototype of this function is:

```
int fclose( FILE *fp );
```

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This 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**.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

## WRITING A FILE

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument **c** to the output stream referenced by **fp**. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by **fp**. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE \*fp, const char \*format, ...)** function as well to write a string into a file. Try the following example:

```
#include <stdio.h>

main()
{
    FILE *fp;

    fp = fopen("/tmp/test.txt", "w+");
    fprintf(fp, "This is testing for fprintf...\n");
    fputs("This is testing for fputs...\n", fp);
    fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in **/tmp** directory and writes two lines using two different functions. Let us read this file in next section.

## READING A FILE

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

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 it returns **EOF**. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** 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.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

main()
{
    FILE *fp;
    char buff[255];

    fp = fopen("/tmp/test.txt", "r");
    fscanf(fp, "%s", buff);
    printf("1 : %s\n", buff );

    fgets(buff, 255, (FILE*)fp);
    printf("2: %s\n", buff );

    fgets(buff, 255, (FILE*)fp);
    printf("3: %s\n", buff );
    fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

## BINARY I/O FUNCTIONS

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,  
             size_t number_of_elements, FILE *a_file);
```

```
size_t fwrite(const void *ptr, size_t size_of_elements,  
             size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.



## Chapter 8

# SOFTWARE DOCUMENTATION

**Software documentation** is written text that accompanies computer software. It both explains how the software operates or how to use it and may mean different things to people in different roles.

### Importance of software documentation

1. Provide for communication among team members
2. They should provide information for management to help them plan, budget and schedule the software development process.
3. It acts as an information repository to be used by maintenance engineers
4. Describe to users how to operate and administer the system
5. In all software projects some amount of documentation should be created prior to any code being written for example Design docs, etc.
6. Documentation should continue after the code has been completed for example User's manuals, etc.

The two main types of documentation created are *Process* and *Product* documents

## PROCESS DOCUMENTATION

- (a) Used to record and track the development process
  - Planning documentation
  - Cost, Schedule, Funding tracking
  - Schedules
  - Standards e.t.c.
- (b) This documentation is created to allow for successful management of a software product
- (c) Has a relatively short lifespan
- (d) Only important to internal development process
- (e) Except in cases where the customer requires a view into this data
- (f) Some items, such as papers that describe design decisions should be extracted and moved into the product documentation category when they become implemented

## PRODUCT DOCUMENTATION

Describes the delivered product

Must evolve with the development of the software product

There are two main categories of process documentation:

### 1. System Documentation

This describes how the system works, but not how to operate it

Examples:

- Requirements Spec
- Architectural Design

- Detailed Design
- Commented Source Code
- Including output such as JavaDoc
- Test Plans
- Including test cases
- V&V plan and results
- List of Known Bugs

## **2. User Documentation**

User Documentation has two main types

- End User
- System Administrator

In some cases these are the same people. The target audience must be well understood. There are five important areas that should be documented for a formal release of a software application. These do not necessarily each have to have their own document, but the topics should be covered thoroughly. These include:

- ✓ Functional Description of the Software
- ✓ Installation Instructions
- ✓ Introductory Manual
- ✓ Reference Manual
- ✓ System Administrator's Guide

## **Document Quality**

Providing thorough and professional documentation is important for any size product development team

### **Document Structure**

All documents for a given product should have a similar structure

The authors "best practices" are:

- Put a cover page on all documents
- Divide documents into chapters with sections and subsections
- Add an index if there is lots of reference information
- Add a glossary to define ambiguous terms