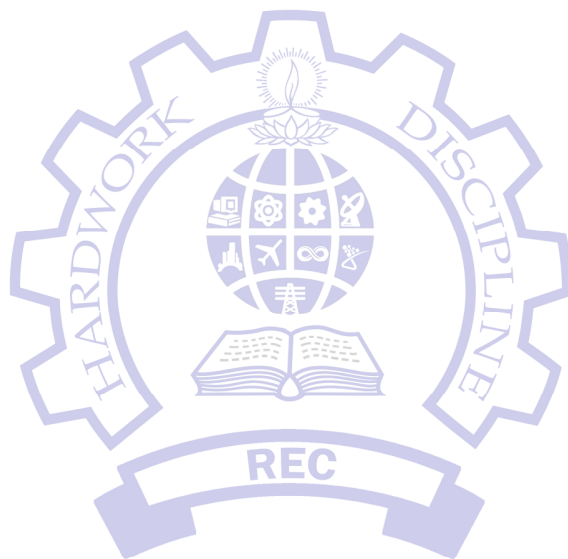


**RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
THANDALAM, CHENNAI – 602105.**



**RAJALAKSHMI
ENGINEERING COLLEGE**

An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

**GE19141
PROGRAMMING USING C**

Computer – components of a computer system-Algorithm, and Flowchart for problem solving with Sequential Logic Structure, Decisions and Loops.

INTRODUCTION

The term computer is derived from the word *compute*. The word *compute* means *to calculate*. A *computer* is an electronic machine that accepts data from the user, processes the data by performing calculations and operations on it, and generates the desired output results. Computer performs both simple and complex operations, with speed and accuracy.

Basic functions of computer are

accepts data	Input
processes data	Processing
produces output	Output
stores results	Storage

Input (Data):

Input is the raw data entered into a computer from the input devices. It is the collection of letters, numbers, images etc.

Process:

Process is the operation on data as per given instruction. It is totally internal process of the computer system.

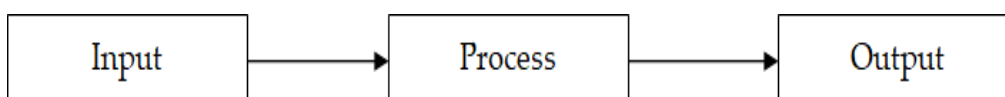
Output:

Output is the processed data or information given by computer after data processing. Output is also called as result. We can save these results in the storage devices for the future use.

Storage:

Storage technology consists of recording media which is used to retain the data.

Computers store information in the form of "1" and "0"s in different types of storages such as memory, hard disk, and usb drives etc. The smallest unit of data in a computer is called Bit. Data storage units are: bit, byte, kilobyte (kb), megabyte (mb), gigabyte (gb), terabyte (tb), petabyte and exabyte, Zettabyte, Yottabyte



DIGITAL AND ANALOG COMPUTERS

A *digital computer* uses distinct values to represent the data internally. All information are represented using the digits 0s and 1s. The computers that we use at our homes and offices are digital computers.

Analog computer is another kind of a computer that represents data as variable across a continuous range of values. The earliest computers were analog computers. Analog computers are used for measuring of parameters that vary continuously in real time, such as temperature, pressure and voltage. Analog computers may be more flexible but generally less precise than digital computers. Slide rule is an example of an analog computer.

CHARACTERISTICS OF COMPUTER

Speed, accuracy, diligence, storage capability and versatility are some of the key characteristics of a computer.

Speed The computer can process data very fast, at the rate of millions of instructions per second. Some calculations that would have taken hours and days to complete otherwise, can be completed in a few seconds using the computer. For example, calculation and generation of salary slips of thousands of employees of an organization, weather forecasting that requires analysis of a large amount of data related to temperature, pressure and humidity of various places, etc.

Accuracy Computer provides a high degree of accuracy. For example, the computer can accurately give the result of division of any two numbers up to 10 decimal places.

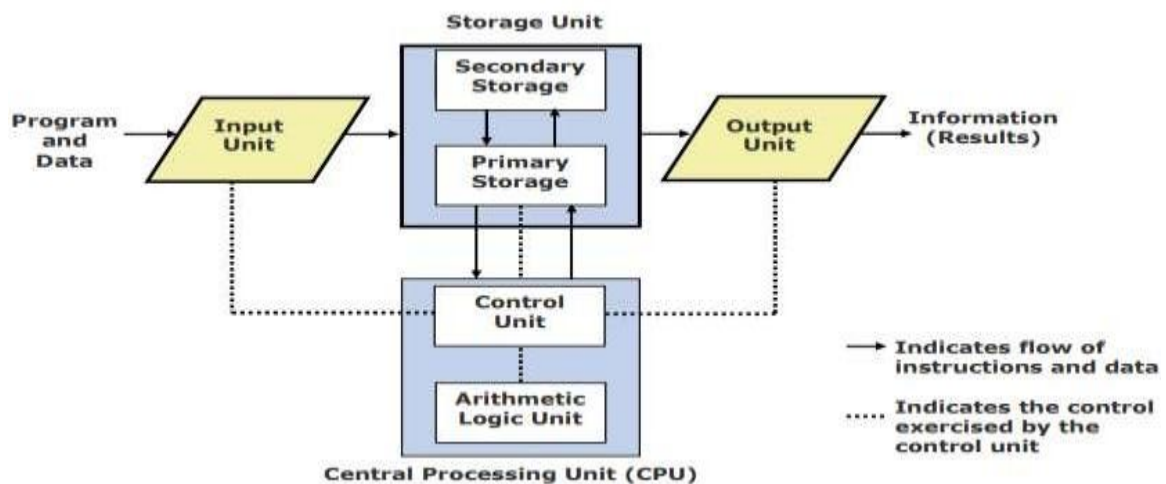
Diligence When used for a longer period of time, the computer does not get tired or fatigued. It can perform long and complex calculations with the same speed and accuracy from the start till the end.

Storage Capability Large volumes of data and information can be stored in the computer and also retrieved whenever required. A limited amount of data can be stored, temporarily, in the primary memory. Secondary storage devices like floppy disk and compact disk can store a large amount of data permanently.

Versatility Computer is versatile in nature. It can perform different types of tasks with the same ease. At one moment you can use the computer to prepare a letter document and in the next moment you may play music or print a document.

Computers have several limitations too. Computer can only perform tasks that it has been programmed to do. Computer cannot do any work without instructions from the user. It executes instructions as specified by the user and does not take its own decisions.

Basic Computer Organization



Input Unit:

An input device is a hardware or peripheral device used to send data to a computer. An input device allows users to communicate and feed instructions and data to computers for processing, display, storage and/or transmission.

Some of the most popularly used input devices are:

- a) Mouse
- b) Light Pen
- c) Touch Screen

- d) Keyboard
- e) Scanner
- f) OCR and MICR
- g) Bar Code Reader
- h) Joy Stick etc.

OutputUnit:

- ❖ The processed data is displayed in the form of result through the output device.

Some of the most popularly used Output devices are:

- a) Visual Display Unit(Monitor)
- b) Printer: Dot Matrix, Line Printers, Ink-jet, Laser Printer
- c) Plotters etc.

Central Processing Unit:

The Central Processing Unit (CPU) is known as the heart of the computer which takes control of the entire processing system of a computer.

- ❖ It performs the basic arithmetical, logical, and input/output operations of a computer system.
- ❖ The part of a computer that interprets and carries out instructions.
- ❖ It also transfers information to and from other components, such as a disk drive or the keyboard.

The CPU has three important sub units.

- 1) Arithmetic-Logic unit
- 2) Control Unit
- 3) Memory Unit

Arithmetic-Logic Unit(ALU):

- ❖ The ALU is an electronic circuit used to carry out the arithmetic operations like addition, subtraction, multiplication and division.
- ❖ It performs the operation on the data provided by the input devices.
- ❖ A comparison operation allows a program to make decisions based on its data input and results of the previous calculations.
- ❖ Logical operations can be used to determine whether particular statement is TRUE or FALSE.
- ❖ The ALU operates on the data available in the main memory and sends them back after processing again to main memory.

Control Unit:

- ❖ The control unit coordinates the activities of all the other units and in the system.
- ❖ Its main functions are to control the transfer of data and information between various units and to initiate appropriate actions by the arithmetic-logic unit.
- ❖ The control unit fetches instructions from the memory, decodes them, and directs them to various units to perform the on specified tasks.

Memory Unit:

Computer memory is divided into two types:

Primary memory Secondary memory

Primary Memory

- ❖ The Primary memory is also called Main memory, is used to store data during processing. Once the CPU has carried out an instruction, it needs the result to be stored. This storage space is provided by the computer's memory.

The storage capacity of the memory is generally measured in megabytes. 1 nibble=4 bits

8 Bits = 1 Byte

1024 Bytes= 1 Kilobyte (KB)

1 024 Kilobytes= 1 Megabyte (MB)

1024 Megabytes= 1 Gigabyte (GB)

Different kinds of primary memory are

- ❖ Random Access Memory (RAM)and
- ❖ Read Only Memory(ROM).

RAM

- ❖ RAM is a volatile **memory**, which means that the stored information is lost when the power is switched off.
- ❖ used to read and write data in RAM

ROM

- ❖ We can only read the data from ROM and you cannot write anything into it and the data is permanent.
- ❖ ROM is a non – **volatile memory**

Secondary Memory

- ❖ The data stored in it is permanent.
- ❖ Data can be deleted if necessary.
- ❖ It is cheaper than primary memory.
- ❖ It has high storage capacity.

There are different kinds of secondary storage devices available. Few of them are :

- ❖ Floppy Disk
- ❖ Fixed or Hard Disk
- ❖ Optical Disk like: CD (Compact Disk) DVD (Digital Versatile Disk)
- ❖ Magnetic Tape Drive

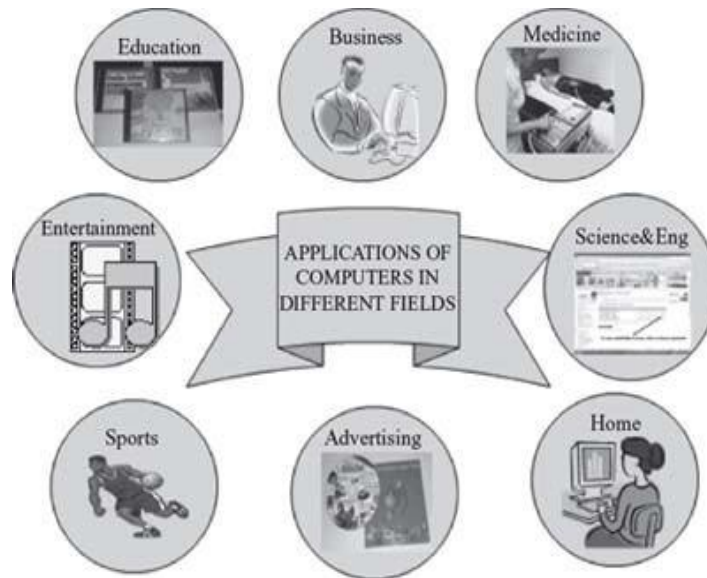
Difference between ROM and RAM

ROM (Read Only Memory)	RAM (Random Access Memory)
ROM is non-volatile	RAM is volatile
ROM is cheaper than RAM	RAM is very expensive
ROM cannot be updated or corrected	RAM can be updated and corrected
ROM serves as permanent data storage	RAM can serve as temporary data storage

APPLICATION OF COMPUTERS

Computers have proliferated into various areas of our lives. For a user, computer is a tool that provides the desired information, whenever needed. You may use computer to get information about the reservation of tickets (railways, airplanes and cinema halls), books in a library, medical history of a person, a place in a map, or the dictionary meaning of a word. The information may be presented to you in the form of text, images, video clips, etc.

Figure shows some of the applications of computer.



Business

A computer has high speed of calculation, diligence, accuracy, reliability, or versatility which made it an integrated part in all business organizations.

Computer is used in business organizations for:

- ❖ Payroll calculations
- ❖ Budgeting
- ❖ Sales analysis
- ❖ Financial forecasting
- ❖ Managing employees database
- ❖ Maintenance of stocks etc.

Banking

Today, banking is almost totally dependent on computers. Banks provide following facilities:

- ❖ Banks provide online accounting facility, which includes current balances, deposits, overdrafts, interest charges, shares, and trustee records.
- ❖ ATM machines are making it even easier for customers to deal with bank transactions.

Insurance

Insurance companies are keeping all records up-to-date with the help of computers. The insurance companies, finance houses and stock broking firms are widely using computers for their concerns. Insurance companies are maintaining a database of all clients with information showing

- ❖ procedure to continue with policies

- ❖ starting date of the policies
- ❖ next due installment of a policy
- ❖ maturity date
- ❖ interests due
- ❖ survival benefits
- ❖ bonus

Education

The computer has provided a lot of facilities in the education system.

- ❖ The computer provides a tool in the education system known as CBE (Computer Based Education).
- ❖ CBE involves control, delivery, and evaluation of learning.
- ❖ The education in schools and colleges is made easy by using computers.

Marketing

Computers are used in the field of marketing for promoting their products by

- ❖ **Advertising** - With computers, advertising professionals create art and graphics, write and revise copy, and print and disseminate ads with the goal of selling more products.
- ❖ **At Home shopping** – Shopping from home has been made possible through use of computers (Online shopping) that provide access to product information and permit direct entry of orders to be filled by the customers.

HealthCare

Computers play an important role in hospitals, labs etc., for diagnosis. ECG, EEG, Ultrasounds and CT Scans etc., are also done by computerized machines.

Some major fields of health care in which computers are used are:

- ❖ **Diagnostic System** - Computers are used to collect data and identify cause of illness.
- ❖ **Lab-diagnostic System** - All tests can be done and reports are prepared by computer.
- ❖ **Patient Monitoring System** - These are used to check patient's signs for abnormality such as in Cardiac Arrest, ECGetc.
- ❖ **Pharma Information System** - Computer checks Drug-Labels, Expiry dates, harmful drug's side effects etc.
- ❖ **Surgery:** Nowadays, computers are also used to perform surgery.

Engineering Design

Computers are widely used in engineering purpose.

One of major areas is CAD (Computer aided design) for creation of architectural plans and designs such as

- ❖ **Structural Engineering** - Requires stress and strain analysis for design of Ships, Buildings, Budgets, and Airplanes etc.
- ❖ **Industrial Engineering** - Computers deal with design, implementation and improvement of integrated systems of people, materials and equipments.
- ❖ **Architectural Engineering** - Computers help in planning towns, designing buildings, determining a range of buildings on a site using both 2D and 3Ddrawings.

Military

Computers are largely used in defense. Some military areas where a computer has been used are:

- ❖ Missile Control
- ❖ Military Communication
- ❖ Military Operation and Planning
- ❖ Smart Weapons

Communication

Communication means to convey a message, an idea, a picture or speech that is received and understood clearly and correctly by the person for whom it is meant for. Some main areas in this category are:

- ❖ E-mail
- ❖ Chatting
- ❖ Usenet
- ❖ FTP
- ❖ Telnet
- ❖ Video-conferencing

Government

Computers play an important role in government. Some major fields in this category are:

- ❖ Budgets
- ❖ Sales tax department
- ❖ Income tax department
- ❖ Male/Female ratio
- ❖ Computerization of voters lists
- ❖ Computerization of driving licensing system
- ❖ Computerization of PAN card
- ❖ Weather forecasting

PROGRAM DEVELOPMENT LIFECYCLE

As stated earlier, a program is needed to instruct the computer about the way a task is to be performed. The instructions in a program have three essential parts:

1. Instructions to accept the input data that needs to be processed,
2. Instructions that will act upon the input data and process it, and
3. Instructions to provide the output to user

The instructions in a program are defined in a specific sequence. Writing a computer program is not a straightforward task. A person who writes the program (computer programmer) has to follow the Program Development Life Cycle.

Let's now discuss the steps that are followed by the programmer for writing a program:

Problem Analysis - The programmer first understands the problem to be solved. The programmer determines the various ways in which the problem can be solved, and decides upon a single solution which will be followed to solve the problem.

Program Design - The selected solution is represented in a form, so that it can be coded. This requires three steps:

An *algorithm* is written, which is an English-like explanation of the solution.

A *flowchart* is drawn, which is a diagrammatic representation of the solution. The solution is represented diagrammatically, for easy understanding and clarity.

A *pseudo code* is written for the selected solution. Pseudo code uses the structured programming constructs. The pseudo code becomes an input to the next phase.

Program Development

The computer programming languages are of different kinds—low-level languages, and high-level languages like C, C++ and Java. The pseudo code is coded using a suitable programming language.

The coded pseudo code or program is compiled for any syntax errors. Syntax errors arise due to the incorrect use of programming language or due to the grammatical errors with respect to the programming language used. During compilation, the syntax errors, if any, are removed.

The successfully compiled program is now ready for execution.

The executed program generates the output result, which may be correct or incorrect. The program is tested with various inputs, to see that it generates the desired results. If incorrect results are displayed, then the program has *semantic error* (logical error). The semantic errors are removed from the program to get the correct results. The successfully tested program is ready for use and is installed on the user's machine.

Program Documentation and Maintenance - The program is properly documented, so that later on, anyone can use it and understand its working. Any changes made to the program, after installation, forms part of the maintenance of program. The program may require updating, fixing of errors etc. during the maintenance phase.

Table summarizes the steps of the program development cycle.

Program Analysis	<ul style="list-style-type: none">• Understand the problem.• Have multiple solutions.• Select a solution.
Program Design	<ul style="list-style-type: none">• Write Algorithm.• Write Flowchart.• Write Pseudocode.
Program Development	<ul style="list-style-type: none">• Compile the program and remove syntax errors, any.• Execute the program.• Test the program. Check the output results with different inputs. If the output is incorrect, modify the program to get correct results.• Install the tested program on the user's computer.
Program Documentation and maintenance	<ul style="list-style-type: none">• Document the program, for later use.• Maintain the program for updating, removing errors, changing requirements etc.

Table Program development life cycle

ALGORITHM

Algorithm is an ordered sequence of finite, well defined, unambiguous instructions for completing a task. Algorithm is an English-like representation of the logic which is used to solve the problem. It is a step- by-step procedure for solving a task or a problem. The steps must be ordered, unambiguous and finite in number.

For accomplishing a particular task, different algorithms can be written. The different algorithms differ in their requirements of time and space. The programmer selects the best- suited algorithm for the given task to be solved.

Let's now look at two simple algorithms to find the greatest among three numbers, as follows:

Algorithm to find the greatest among three numbers:

ALGORITHM 1

- Step 1: Start
- Step 2: Read the three numbers A, B, C
- Step 3: Compare A and B. If A is greater perform step 4 else perform step 5.
- Step 4: Compare A and C. If A is greater, output "A is greatest" else output "C is greatest". Perform step 6.
- Step 5: Compare B and C. If B is greater, output "B is greatest" else output "C is greatest".
- Step 6: Stop

ALGORITHM 2

- Step 1: Start
- Step 2: Read the three numbers A, B, C
- Step 3: Compare A and B. If A is greater, store A in MAX, else store B in MAX.
- Step 4: Compare MAX and C. If MAX is greater, output "MAX is greatest" else output "C is greatest".
- Step 5: Stop

Both the algorithms accomplish the same goal, but in different ways. The programmer selects the algorithm based on the advantages and disadvantages of each algorithm. For example, the first algorithm has more number of comparisons, whereas in the second algorithm an additional variable MAX is required.

CONTROLSTRUCTURES

The logic of a program may not always be a linear sequence of statements to be executed in that order. The logic of the program may require execution of a statement based on a decision. It may repetitively execute a set of statements unless some condition is met. Control structures specify the statements to be executed and the order of execution of statements.

Flowchart and Pseudo code use control structures for representation. There are three kinds of control structures:

- Sequential - instructions are executed in linear order
- Selection (branch or conditional) - it asks a true/false question and then selects the next instruction based on the answer
- Iterative (loop) - it repeats the execution of a block of instructions.

FLOWCHART

A *flowchart* is a diagrammatic representation of the logic for solving a task. A flowchart is drawn using boxes of different shapes with lines connecting them to show the flow of control. The purpose of drawing a flowchart is to make the logic of the program clearer in a visual form. There is a famous saying “A photograph is equivalent to thousand words”. The same can be said of flowchart. The logic of the program is communicated in a much better way using a flowchart. Since flowchart is a diagrammatic representation, it forms a common medium of communication.

Flowchart Symbols

A flowchart is drawn using different kinds of symbols. A symbol used in a flowchart is for a specific purpose. Figure shows the different symbols of the flowchart along with their names. The flowchart symbols are available in most word processors including MS-WORD, facilitating the programmer to draw a flowchart on the computer using the word processor.






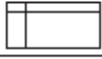























Process 	Alternate process 	Decision 	Data 	Predefined process 
Internal storage 	Document 	Multi document 	Terminator 	Preparation 
Manual input 	Manual operation 	Connector 	Off-page connector 	Card 
Punched tape 	Summing junction 	OR 	Collate 	Sort 
Extract 	Merge 	Stored data 	Delay 	Sequential access storage 
Magnetic disk 	Direct access storage 	Display 	Flow lines 	

Figure Flowchart symbols (available for use in MS-WORD)

Preparing a Flowchart

A flowchart may be simple or complex. The most common symbols that are used to draw a flowchart are - Process, Decision, Data, Terminator, Connector and Flow lines. While drawing a flowchart, some rules need to be followed-(1) A flowchart should have a start and end, (2) The direction of flow in a flowchart must be from top to bottom and left to right, and (3) The relevant symbols must be used while drawing a flowchart. While preparing the flowchart, the sequence, selection or iterative structures may be used wherever required. Figure shows the sequence, selection and iteration structures.

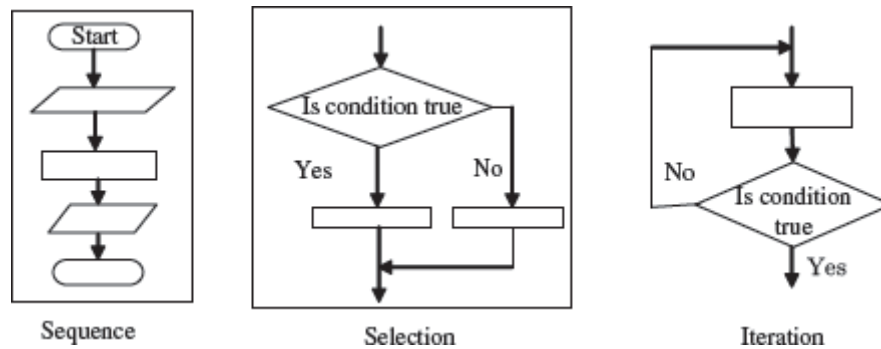


Figure Control structures in flowchart

We see that in a sequence, the steps are executed in linear order one after the other. In a selection operation, the step to be executed next is based on a decision taken. If the condition is true (yes) a different path is followed than if the condition evaluates to false (no). In case of iterative operation, a condition is checked. Based upon the result of this conditional check, true or false, different paths are followed. Either the next step in the sequence is executed or the control goes back to one of the already executed steps to make a loop.

Here, we will illustrate the method to draw flowchart, by discussing three different examples. To draw the flowcharts, relevant boxes are used and are connected via flow lines. The flowchart for the examples is shown in Figure below.

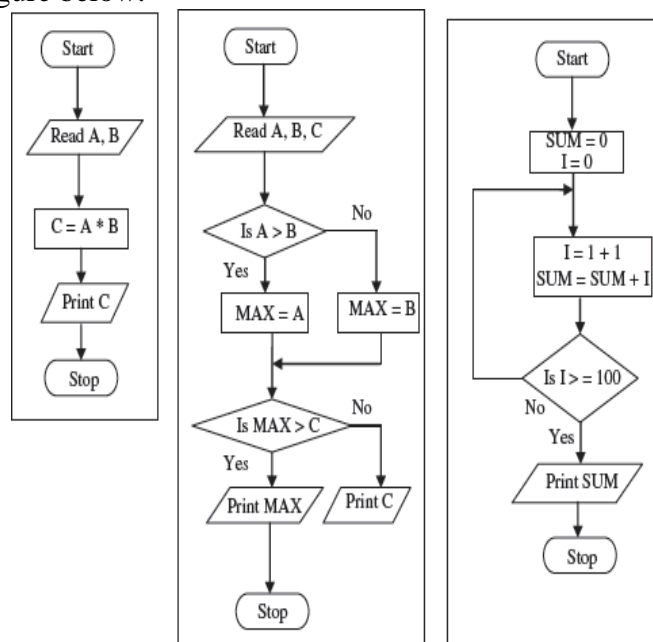


Figure Examples of flowchart

The first flowchart computes the product of any two numbers and gives the result. The flowchart is a simple sequence of steps to be performed in a sequential order.

The second flowchart compares three numbers and finds the maximum of the three numbers. This flowchart uses selection. In this flowchart, decision is taken based upon a condition, which decides the next path to be followed, i.e. If A is greater than B then the true (Yes) path is followed else the false (No) path is followed. Another decision is again made while comparing MAX with C.

The third flowchart finds the sum of first 100 integers. Here, iteration (loop) is performed so that some steps are executed repetitively until they fulfill some condition to exit from the repetition. In the decision box, the value of I is compared with 100. If it is false (No), a loop is created which breaks when the condition becomes true(Yes).

Flowcharts have their own benefits; however, they have some limitations too. A complex and long flowchart may run into multiple pages, which become difficult to understand and follow. Moreover, updating a flowchart with the changing requirements is a challenging job.

PSEUDOCODE

Pseudo code consists of short, readable and formally-styled English language used for explaining an algorithm. Pseudo code does not include details like variable declarations, subroutines etc. Pseudo code is a short-hand way of describing a computer program. Using pseudo code, it is easier for a programmer or a non-programmer to understand the general working of the program, since it is not based on any programming language. It is used to give a sketch of the structure of the program, before the actual coding. It uses the structured constructs of the programming language but is not machine-readable. Pseudo code cannot be compiled or executed. Thus, no standard for the syntax of pseudo code exists. For writing the pseudo code, the programmer is not required to know the programming language in which the pseudo code will be implemented later.

Preparing a Pseudo Code

Pseudo code is written using structured English. In a pseudo code, some terms are commonly used to represent the various actions. For example, for inputting data the terms may be (INPUT, GET, READ), for outputting data (OUTPUT, PRINT, DISPLAY), for calculations (COMPUTE, CALCULATE), for incrementing (INCREMENT), in addition to words like ADD, SUBTRACT, INITIALIZE used for addition, subtraction, and initialization, respectively.

The control structures—sequence, selection, and iteration are also used while writing the pseudo code. Figure below shows the different pseudo code structures. The *sequence structure* is simply a sequence of steps to be executed in linear order. There are two main *selection constructs*—if- statement and case statement. In the *if-statement*, if the condition is true then the THEN part is executed otherwise the ELSE part is executed. There can be variations of the if-statement also, like there may not be any ELSE part or there may be nested ifs. The case statement is used where there are a number of conditions to be checked. In a case statement, depending on the value of the expression, one of the conditions is true, for which the corresponding statements are executed. If no match for the expression occurs, then the OTHERS option which is also the default option, is executed.

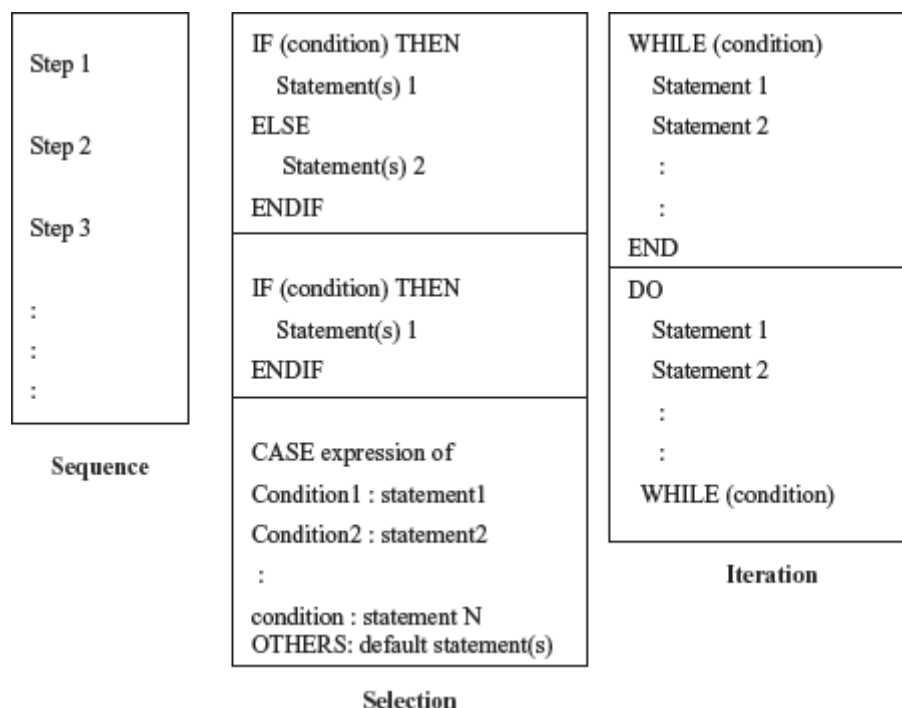


Figure Control structures for pseudo code

WHILE and DO-WHILE are the two iterative statements. The WHILE loop and the DO-

WHILE loop, both execute while the condition is true. However, in a WHILE loop the condition is checked at the start of the loop, whereas, in a DO-WHILE loop the condition is checked at the end of the loop. So the DO-WHILE loop executes at least once even if the condition is false when the loop is entered.

In the following Figure, the pseudo code is written for the same three tasks for which the flowchart was shown in the previous section. The three tasks are—(i) compute the product of any two numbers, (ii) find the maximum of any three numbers, and (iii) find the sum of first 100 integers.

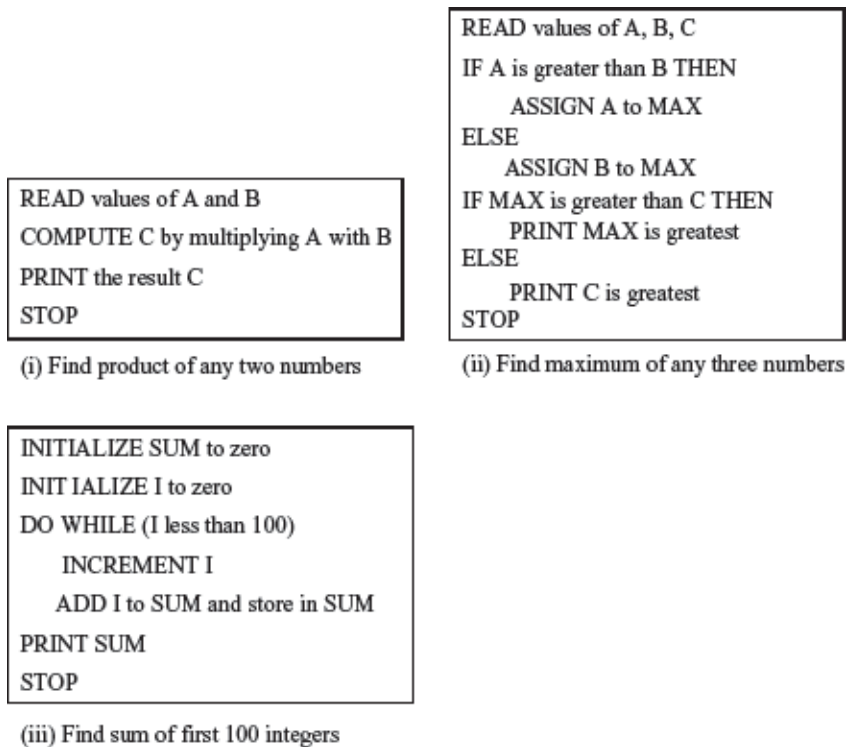


Figure Examples of pseudo code

A pseudo code is easily translated into a programming language. But, as there are no defined standards for writing a pseudo code, programmers may use their own style for writing the pseudo code, which can be easily understood. Generally, programmers prefer to write pseudo code instead of flowcharts.

Difference between Algorithm, Flowchart, and Pseudo Code: An algorithm is a sequence of instructions used to solve a particular problem. Flowchart and Pseudo code are tools to document and represent the algorithm. In other words, an algorithm can be represented using a flowchart or a pseudo code. Flowchart is a graphical representation of the algorithm. Pseudo code is a readable, formally styled English like language representation of the algorithm. Both flowchart and pseudo code use structured constructs of the programming language for representation. The user does not require the knowledge of a programming language to write or understand a flowchart or a pseudo code.

Introduction- C Structure- syntax and constructs of ANSI C - Variable Names, Data Type and Sizes, Constants, Declarations - Arithmetic Operators, Relational Operators, Logical Operators, Type Conversion, Increment and Decrement Operators, Bitwise Operators, Assignment Operators and Expressions, Precedence and Order of Evaluation, proper variable naming and Hungarian Notation.

Overview of C:

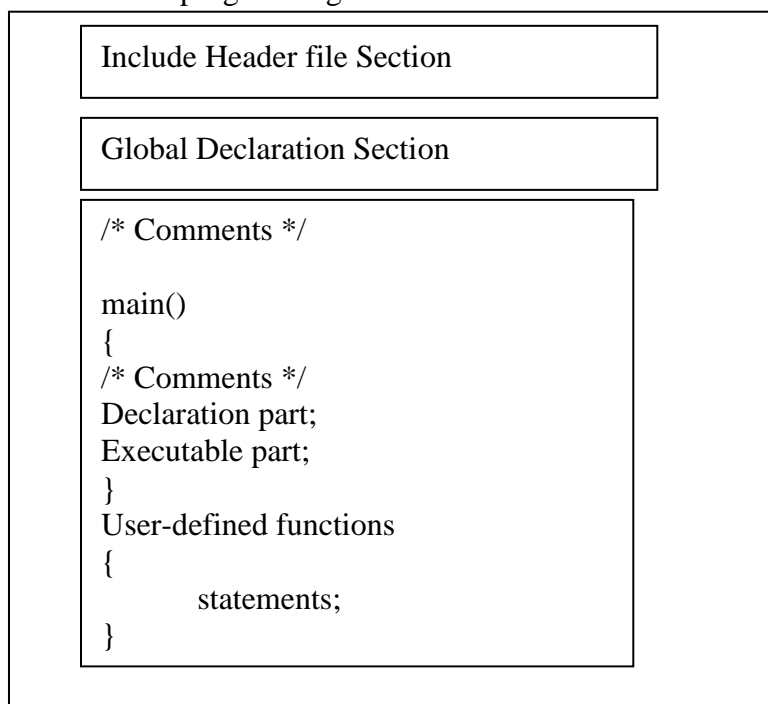
C is one of the most popular programming languages. It was developed by Mr. Dennis Ritchie at AT&T Bell Laboratories at USA in 1972. It is an upgraded version of two earlier languages, called BCPL and B, which were also developed at Bell Laboratories. C is called a middle level language. It performs the task of low level language as well as high level language. UNIX operating system is written in C Language.

Features and Applications of C Language:

1. C is a general purpose, structured programming language.
2. C is a powerful, efficient, compact and flexible language.
3. C is highly portable. It can run on different operating systems.
4. It has got rich set of operators.
5. C is widely available.
6. C is a middle level language.
7. C allows manipulation of data at the lowest level.
8. C allows dynamic memory allocation.
9. C is well suited for writing system software
10. C is a robust language.

Structure of a C Program

The Structure of a C program is given below:



- i. **Include Header file section:** C program depends upon some header files for function definition that are used in program.
Eg. `#include <stdio.h>`
- ii. **Global Declaration section:** This section declares some variables that are used in more than one function. These variables are known as global variables.
- iii. **Function main:** Every program written in C language must contain main function. It is the starting point of every C program.
- iv. **Declaration part:** It contains the entire variables that are used in the executable part. The initialization of variables is also done here.
- v. **Executable part:** This part contains the statements following the declaration of the variables.
- vi. **User defined functions:** The functions defined by the users are called user defined functions. They are generally defined after the main function.
- vii. **Comments:** To understand the flow of the program, the programmer can include comments in the program.

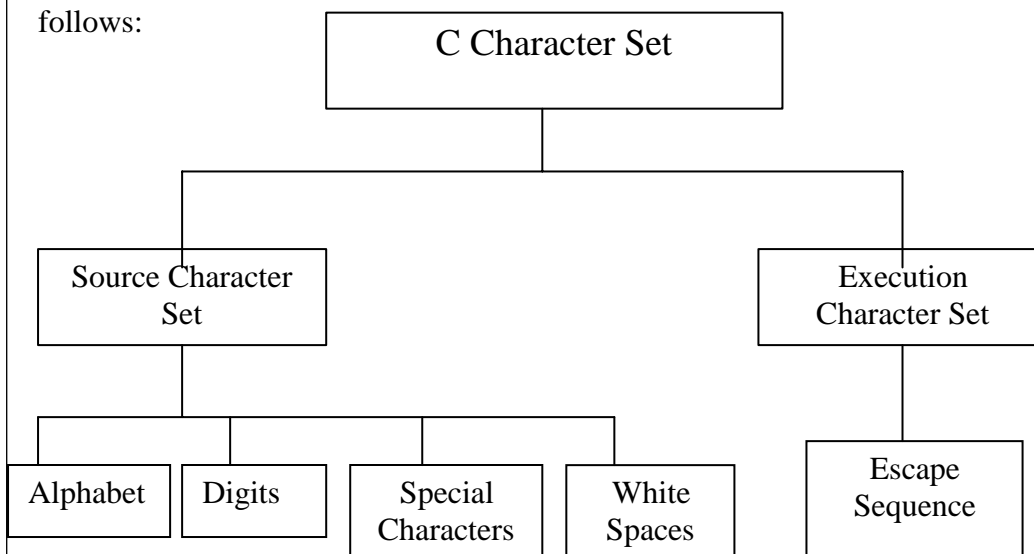
Programming Rules:

1. All statements must be in lower case letters.
2. Blank spaces may be inserted between the words.
3. The opening and closing braces must be balanced.
4. The statements can be written anywhere between the opening and closing braces.
5. Every statement must be terminated by a semicolon (;).

Lexical Elements of C

Character Set of C:

The character set is the fundamental set of any language. The character set of C can be represented as follows:



Source Character Set:

They are used to construct the statements in the source program.

S.No.	Source Character Set	Notation
1.	Alphabets	A to Z and a to z
2.	Decimal Digits	0 to 9
3.	White Spaces	Blank space
4.	Special Characters	+, -, *, /, %, \$, # etc.

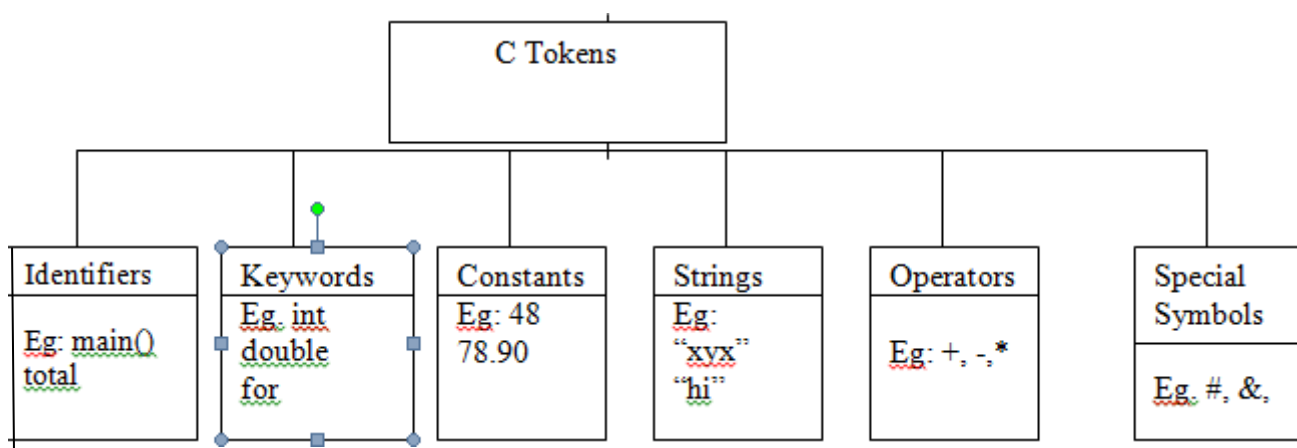
Execution Character Set:

These are employed at the time of execution. This set of characters are also called as non graphic characters because, these characters are invisible and cannot be printed directly. These characters will have effect only when the program is executed. They are also called as “escape sequences”.

Example:

<u>CODE</u>	<u>MEANING</u>
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical Tab
\a	Alert

C Tokens: The tokens are usually referred as individual text and punctuation in a passage of text. The C language program contains individual units called the C tokens and has the following



Identifiers & Keywords:

In C language, every word is classified into either a keyword or an identifier.

Identifiers: Identifiers are names given to various program elements, such as variables, functions and arrays etc.

Rules for naming an identifier:

- Identifiers consist of letters and digits in any order.
- The first character must be a letter or character or may begin with underscore.
- An identifier can be of any length while most of the C compiler recognizes only the first 31 characters.
- No space and special symbols are allowed between the identifier.
- The identifier cannot be a keyword.

Valid Identifiers are:

stdname, sub, tot_marks

Invalid Identifiers are:

Return, std name, * say

Keywords: There are certain reserved words called keywords that have standard and predefined meaning in C language. They are the basic building blocks for program statements.

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

Data Types:

Data type is the type of the data that are going to access within the program. C supports different data types; each data type may have predefined memory requirement and storage representation.

C Data Types			
Primary / Basic	User Defined	Derived	Empty
char int float double	typedef structures union	arrays pointers	void

Primary Data types:

Data type	Description	Memory Bytes	Control String
int	Integer Quantity	2 Bytes	%d or %i
Char	Single Character	1 Byte	%c
Float	Floating point no.	4 Bytes	%f
Double	Double precision floating point nos.	8 Bytes	%lf

VARIABLES:

A Variable is an identifier that is used to represent some specified type of information within a designated portion of the program. A variable may take different values at different times during the execution.

Rules for naming the variables:

1. A variable name can be any combination of 1 to 8 alphabets, digits or underscore.
2. The first character must be an alphabet or an underscore.
3. The length of the variable cannot exceed upto 8 characters long.
4. No commas or blank spaces are allowed within a variable name.
5. No special symbol, an underscore can be used in a variable name.

Variable Declaration:

Any variable to be used in the program is to be declared before it is used in the program.

The declaration of variable is given below:

<datatype> v1,v2,v3,.....,vn;

where

datatype – type of data (eg.int, char etc.)

v1,v2,v3,....vn- the list of variables.

Variable definition / Initializing variables:

It means providing an initial value to the variables. It is done using assignment operator.

Description:

Datatype variable=constant; (eg. int b=90)

or

Variable=constant; (eg. a=10, a=b=87)

Hungarian notation

Hungarian notation is a naming convention in computer programming that indicates either the type of object or the way it should be used. It was originally proposed by Charles Simonyi, a programmer at Xerox PARC in the early 1980s. There are two variations of Hungarian notation: Systems and Apps. They both involve using a special prefix as part of the name to indicate an object's nature.

Hungarian notation prefixes

The prefix used is up to the programmer, but standard prefixes include:

Prefix	Data type
b	boolean.
by	byte or unsigned char.
c	char.
cx / cy	short used as size.
dw	DWORD, double word or unsigned long.
fn	function.
h	handle.
i	int (integer).
l	Long.
n	short int.
p	a pointer variable containing the address of a variable.
s	string.
sz	ASCIIZ null-terminated string.
w	WORD unsigned int.
x, y	short used as coordinates.

Do you know difference between variable declaration & definition?

S.NO	VARIABLE DECLARATION	VARIABLE DEFINITION
1	Declaration tells the compiler about data type and size of the variable.	Definition allocates memory for the variable.
2	Variable can be declared many times in a program.	It can happen only one time for a variable in a program.
3	The assignment of properties and identification to a variable.	Assignments of storage space to a variable.

Scope of the variables:

Scope of the variable implies the availability of variables within the program. There are two types.

1. **Local Variables:** The variables which are defined inside a function is called a local variable.

Eg:

```
void main()
{
    int a,b;
    ----
    ----
}
```

Here, a & b are local variables.

2. **Global/External Variables:** The variables that are declared before the main function are called global variables.

Eg:

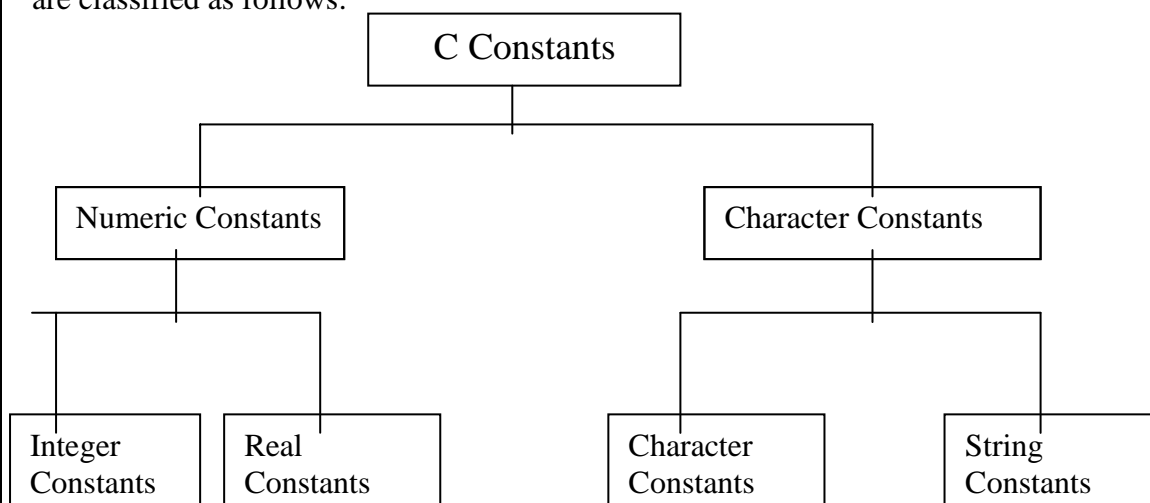
```
int a =2;

void main()
{
    -----
    -----
}
```

Here a is a global variable

Constants:

The data items whose values cannot be changed during the execution of program are called constants. They are classified as follows:



Numeric Constants:

- a. **Integer Constants:** An integer constant is formed with the sequence of digits.

Eg. 67,90 etc.

- b. **Real Constants:** A real constant is made up of a sequence of numeric digits with presence of a decimal point.

Eg. 127.90,89.78 etc.

Character Constants:

- a. **Single Character Constants:** The character constant contains a single character enclosed within a pair of single inverted commas both pointing to the left.

Eg. 's', 'M', etc.

- b. **String Constants:** A string constant is a sequence of characters enclosed in double quotes.

Eg. "hi", "Hello", etc.

Delimiters:

Delimiters are special symbols, which have special meaning and got significance. Some of the delimiters are:

Symbol	Name	Meaning
#	Hash	Pre-processor directive
,	Comma	Separator between variables
:	Colon	Label
;	Semi-colon	Statement terminator
()	Parenthesis	Used in functions & expressions

Statements:

Statements can be defined as set of declaration of sequence of action. Statement causes the system to perform some action. All statements end with a semi-colon except conditional and control structures.

There are 3 different types of statements. They are:

1. **Assignment Statement:** Assignment operator is used to assign value to the variables.

Eg: a=6

Sum = 0

2. **Null Statement:** A statement without any characters and it has only semi colon is called a Null statement.

Eg. ;

3. **Expression Statement:** Any statement which contains an expression on right side is called Expression statement.

Eg: a=b+c

OPERATORS

An operator is a symbol that specifies an operation to be performed on the operands. The data items that operators act upon are called operands. Some operators require two operands. They are called binary operators. Some operators require one operand. They are called unary operator.

There are different types of operators. They are:

1. Arithmetic Operator
2. Relational Operator
3. Logical Operator
4. Assignment Operator
5. Increment & decrement Operator
6. Conditional Operator
7. Bitwise Operator
8. Special Operator

1. **Arithmetic Operator:** The basic arithmetic operators are:

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum= sum* 2;	4	8
Divide	/	sum= sum/ 2;	4	2
Addition	+	sum= sum+ 2;	4	6
Subtraction	-	sum= sum -2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum= sum% 3;	4	1

Example Program:

```
#include <stdio.h>
void main()
{
    int a,b,c,d,e,f;
    a=10;
    b=7;
    c=a+b;
    d=a-b;
    e=a*b;
    f=a/b;
    printf("\n%d",c);
    printf("\n%d",d);
    printf("\n%d",e);
    printf("\n%d",f);
}
```

Output:

```
17
3
70
1
```

2. **Relational Operator:** Relational operators are used to compare two or more operands. Operands may be variables, constants or expressions. The various relational operators are

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

The conditions are checked using relational operators. They return either 1 or 0 as true value or false value respectively.

Example Program

```
#include<stdio.h>
void main()
{
clrscr();
printf("\nCondition : Return Value");
printf("\n5!=5 : %d", (5!=5));
printf("\n10<11 : %d", (10<11));
printf("\n12>9 : %d", (12>9));
printf("\n55>=90 : %d", (55>=90));
printf("\n45<=91 : %d", (45<=91));
printf("\n78==78 : %d", (78==78));
getch();
}
```

Output:

```
Condition : Return Value
5!=5 : 0
10<11 : 1
12>9 : 1
55>=90 : 0
45<=91 : 1
78==78 : 1
```

3. **Logical Operator:** It is used to combine the results of two or more conditions.

Operator	Meaning	Example	Return Value (Result)
&&	Logical AND	(4<6)&&(8==9)	0
	Logical OR	(7<9) (3<1)	1
!	Logical NOT	!(29>89)	1

Example Program:

```
#include<stdio.h>
void main()
{
clrscr();
printf("\nCondition : Return Value");
printf("\n5!=5&&6<9 : %d", (5!=5)&&(6<9));
printf("\n10<11||10==10 : %d", (10<11)|| (10==10));
printf("\n!(12>9): %d", !(12>9));
getch();
}
```

Output:

```
Condition : Return Value
5!=5&&6<9 : 0
10<11||10==10 : 1
!(12>9) : 0
```


4. **Assignment Operator**: It is used to assign a value or an expression to another variable.

Syntax:

Variable=expression (or) value

Compound Assignment:

Operator	Example	Meaning
+ =	x+=y	x=x+y
- =	x-=y	x=x-y
* =	X*=y	x=x*y
/ =	x/=y	x=x/y

Example Program:

```
#include <stdio.h>
void main()
{
int a,b;
clrscr();
a=10; //Assignment Statement
b=6;
a+=b; // Compound Statement
b-=a;
printf("\nThe value of a is %d",a);
printf("\nThe value of b is %d",b);
getch();
}
```

Output:

The value of a is 16

The value of b is -10

5. **Increment & Decrement Operators**: These operators are called Unary operators. They are used for incrementation & decrementation operation.

Operator	Meaning
++x	Pre-increment
X++	Post increment
--y	Pre-decrement
Y--	Post decrement

Example Program:

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

```

int a,b;
clrscr();
a=10;
b=6;
printf("\nThe value of a is %d",a++);
printf("\nThe value of a is %d",++a);
printf("\nThe value of b is %d",b--);
printf("\nThe value of b is %d",--b);
getch();
}

```

Output:

The value of a is 10

The value of a is 12

The value of b is 6

The value of b is 4

6. **Conditional Operator:** It checks the condition and executes the statements depending on the condition.

Syntax: condition? exp1: exp2

The ? Operator acts as a ternary operator. If the condition is true, it evaluate the first expression otherwise it evaluates the second expression.

Sample Program:

```

#include <stdio.h>
void main()
{
int a,b,big;
clrscr();
a=10;
b=6;
big=a>b?a:b;
printf("\nThe biggest value is %d",big);
getch();
}

```

Output:

The biggest value is 10

7. **Bitwise Operators:** They are used to manipulate the data at bit level. They operate on integers only.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	One's Complement

>>	Right Shift
<<	Left Shift

Example Program:

```
#include <stdio.h>
void main()
{
int a,b,c,d,e;
clrscr();
a=10;
b=6;
c=a&b;
printf("\nThe value of c(AND) is %d",c);
d=a|b;
printf("\nThe value of d(OR) is %d",d);
e=a^b;
printf("\nThe value of e(XOR) is %d",e);
getch();
}
```

Output:

The value of c(AND) is 2

The value of d(OR) is 14

The value of e(XOR) is 12

8. **Special Operator:** C Language supports some special operators. They are:

Operator	Meaning
,	Comma operator
Sizeof	Size of the Operands
& and *	Pointer operator

The comma operator is used to separate the variables.

The size of operator is used to get the size of every operands.

The pointer operator is used to get the address of the operand.

Example Program:

```
#include <stdio.h>
void main()
{
int a;
char b;
clrscr();
printf("\nThe size of a is %d",sizeof(a));
printf("\nThe size of b is %d",sizeof(b));
getch();
}
```

Output:

The size of a is 2

The size of b is 1

Operator Precedence & Associativity of Operators

The Arithmetic Operators are evaluated from the left to right using the precedence of operators, when the expression is written without the parenthesis.

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	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
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Rules for evaluation of expression:

1. Any expression within the parenthesis is evaluated first.
2. Arithmetic expression is evaluated from left to right using the rule of precedence.
3. Within the parenthesis, highest precedence operator is evaluated first.
4. If the operators have the same precedence, associativity is to be applied.
5. If the parenthesis is nested, the innermost sub-expression is evaluated first.

Example:

Given expression $45 + 8 - 5 * 7$

Step 1: $45 + 8 - \underline{5 * 7}$

Step 2: **45 + 8** - 35

Step 3: 53 - 35

Step 4: 18

The result is 18

UNIT-III I/O AND CONTROL FLOW

Standard I/O, Formatted Output – Printf, Variable-length argument lists- Formatted Input – Scanf, Statements and Blocks, If-Else-If, Switch, Loops – while, do, for, break and continue, GoTo Labels.

Basics of Formatted Input/Output in C

Concepts

- I/O is essentially done one character (or byte) at a time
- **stream** -- a sequence of characters flowing from one place to another
 - *input stream*: data flows from input device (keyboard, file, etc) into memory
 - *output stream*: data flows from memory to output device (monitor, file, printer, etc)
- **Standard I/O streams** (with built-in meaning)
 - stdin: standard input stream (default is keyboard)
 - stdout: standard output stream (defaults to monitor)
 - stderr: standard error stream
- **stdio.h** -- contains basic I/O functions
 - scanf: reads from standard input (stdin)
 - printf: writes to standard output (stdout)
 - There are other functions similar to printf and scanf that write to and read from other streams
 - How to include, for C or C++ compiler
 - `#include <stdio.h>` // for a C compiler
 - `#include <cstdio>` // for a C++ compiler
- **Formatted I/O** -- refers to the conversion of data to and from a stream of characters, for printing (or reading) in plain text format
 - All text I/O we do is considered *formatted I/O*
 - The other option is reading/writing direct binary information (common with file I/O, for example)

Output with printf

- The basic format of a **printf** function call is:
- `printf (format_string, list_of_expressions);`
where:
 - *format_string* is the layout of what's being printed
 - *list_of_expressions* is a comma-separated list of variables or expressions yielding results to be inserted into the output

To output string literals, just use one parameter on printf, the string itself

- `printf("Hello, world!\n");`
- `printf("Greetings, Earthling\n\n");`

Conversion Specifiers

A conversion specifier is a symbol that is used as a placeholder in a formatting string. For integer output (for example), %d is the specifier that holds the place for integers.

Here are some commonly used conversion specifiers (not a comprehensive list):

%d	int (signed decimal integer)
%u	unsigned decimal integer
%f	floating point values (fixed notation) - float, double
%e	floating point values (exponential notation)
%s	string
%c	character

Printing Integers

- To output an integer, use %d in the format string, and an integer expression in the *list_of_expressions*.

```
int numStudents = 35123;
printf("FSU has %d students", numStudents);
```

```
// Output:
// FSU has 35123 students
```

- We can specify the field width (i.e. how many 'spaces' the item prints in). Defaults to right-justification. Place a number between the % and the d. In this example, field width is 10:

```
printf("FSU has %10d students", numStudents);
```

```
// Output:
// FSU has    35123 students
```

- To left justify, use a negative number in the field width:

```
printf("FSU has %-10d students", numStudents);
```

```
// Output:
// FSU has 35123    students
```

- If the field width is too small or left unspecified, it defaults to the minimum number of characters required to print the item:

```
printf("FSU has %2d students", numStudents);
```

```
// Output:
// FSU has 35123 students
```

- Specifying the field width is most useful when printing multiple lines of output that are meant to line up in a table format

Printing Floating-point numbers

- Use the %f modifier to print floating point values in fixed notation:

```
double cost = 123.45;
printf("Your total is $%f today\n", cost);
```

```
// Output:
// Your total is $123.450000 today
```

- Use %e for exponential notation:

```
printf("Your total is $%e today\n", cost);
```

```
// Output:
// Your total is $1.234500e+02 today
```

Note that the e+02 means "times 10 to the 2nd power"

- You can also control the decimal precision, which is the number of places after the decimal. Output will round to the appropriate number of decimal places, if necessary:

```
printf("Your total is $%.2f today\n", cost);
```

```
// Output:
// Your total is $123.45 today
```

- Field width can also be controlled, as with integers:

```
printf("Your total is $%9.2f today\n", cost);
```

```
// Output:
// Your total is $   123.45 today
```

In the conversion specifier, the number before the decimal is field width, and the number after is the precision. (In this example, 9 and 2).

- %-9.2 would left-justify in a field width of 9, as with integers

Printing characters and strings

- Use the formatting specifier %c for characters. Default field size is 1 character:

```
char letter = 'Q';
printf("%c%c%c\n", '*', letter, '*');
```

```
// Output is: *Q*
```

- Use %s for printing strings. Field widths work just like with integers:


```
printf("%s%10s%-10sEND\n", "Hello", "Alice", "Bob");
```

```
// Output:
```

```
// Hello   AliceBob   END
```

scanf

- To read data in from standard input (keyboard), we call the **scanf** function. The basic form of a call to scanf is:

- `scanf(format_string, list_of_variable_addresses);`
 - The format string is like that of printf
 - But instead of expressions, we need space to store incoming data, hence the list of variable addresses

- If **x** is a variable, then the expression **&x** means "address of x"
- scanf example:

```
int month, day;  
printf("Please enter your birth month, followed by the day: ");  
scanf("%d %d", &month, &day);
```

- **Conversion Specifiers**
 - Mostly the same as for output. Some small differences
 - Use %f for type float, but use %lf for types double and long double
- The data type read, the conversion specifier, and the variable used need to match in type
- White space is skipped by default in consecutive *numeric* reads. But it is *not* skipped for character/string inputs.

Example

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

```
int main()
```

```
{
```

```
int i;
```

```
float f;
```

```
char c;
```

```
printf("Enter an integer and a float, then Y or N\n> ");
```

```
scanf("%d%f%c", &i, &f, &c);
```

```
printf("You entered:\n");
```

```
printf("i = %d, f = %f, c = %c\n", i, f, c);
```

```
return 0;
```

```
}
```

Sample run #1

User input underlined, to distinguish it from program output

Enter an integer and a float, then Y or N

> 34 45.6Y

You entered:

i = 34, f = 45.600, c = Y

Sample Run #2

Enter an integer and a float, then Y or N

> 12 34.5678 N

You entered:

i = 12, f = 34.568, c =

Note that in this sample run, the character that was read was **NOT** the letter 'N'. It was the space. (Remember, white space not skipped on character reads).

This can be accounted for. Consider if the scanf line looked like this:

```
scanf("%d%f %c", &i, &f, &c);
```

There's a space between the %f and the %c in the format string. This allows the user to type a space. Suppose this is the typed input:

```
12 34.5678 N
```

Then the character variable c will now contain the 'N'.

Interactive Input

You can make input more interactive by prompting the user more carefully. This can be tedious in some places, but in many occasions, it makes programs more user-friendly.

Example:

```
int age;
double gpa;
char answer;

printf("Please enter your age: ");
scanf("%d", &age);
printf("Please enter your gpa: ");
scanf("%lf", &gpa);
printf("Do you like pie (Y/N)? ");
scanf("%c", &answer);
```

printf/scanf with C-strings

An entire C-style string can be easily printed, by using the %s formatting symbol, along with the name of the char array storing the string (as the argument filling in that position):

```
char greeting[] = "Hello";

printf("%s", greeting);    // prints the word "Hello"
```

Be careful to **only** use this on char arrays that are being used as C-style strings. (This means, only if the null character is present as a terminator).

Similarly, you can read a string into a char array with scanf. The following call allows the entry of a word (up to 19 characters and a terminating null character) from the keyboard, which is stored in the array word1:

```
char word1[20];  
  
scanf("%s", word1);
```

Characters are read from the keyboard until the first "white space" (space, tab, newline) character is encountered. The input is stored in the character array and the null character is automatically appended.

Note also that the & was not needed in the scanf call (word1 was used, instead of &word1). This is because the name of the array by itself (with no index) actually IS a variable that stores an address (a *pointer*).

VARIABLE LENGTH ARGUMENTS

- Variable length arguments is an advanced concept in C language offered by c99 standard. In c89 standard, fixed arguments only can be passed to the functions.
 - When a function gets number of arguments that changes at run time, we can go for variable length arguments.
 - It is denoted as ... (3 dots)
 - stdarg.h header file should be included to make use of variable length argument functions.
-
- Here we use macros to implement the functionality of variable arguments.

Use va_list type variable in the function definition.

```
int a_function(int x, ...)  
{  
    va_list a_list;  
    va_start( a_list, x );  
}
```

- Use int parameter and va_start macro to initialize the va_list variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use va_arg macro and va_list variable to access each item in argument list.
- macro va_end to clean up the memory assigned to va_list variable.

Example 1:

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

```

int add(int num,...);

int main()
{
    printf("The value from first function call = " \
           "%d\n", add(2,2,3));
    printf("The value from second function call= " \
           "%d \n", add(4,2,3,4,5));

    /*Note - In function add(2,2,3),
               first 2 is total number of arguments
               2,3 are variable length arguments
               In function add(4,2,3,4,5),
               4 is total number of arguments
               2,3,4,5 are variable length arguments
    */
    return 0;
}

int add(int num,...)
{
    va_list valist;
    int sum = 0;
    int i;

    va_start(valist, num);
    for (i = 0; i < num; i++)
    {
        sum += va_arg(valist, int);
    }
    va_end(valist);
    return sum;
}

```

OUTPUT:

The value from first function call = 5
The value from second function call= 14

In the above program, function “add” is called twice. But, number of arguments passed to the function gets varies for each. So, 3 dots (...) are mentioned for function ‘add’ that indicates that this function will get any number of arguments at run time.

Example 2:

Example, to find minimum of given set of integers.

// C program to demonstrate use of variable

```

// number of arguments.
#include <stdarg.h>
#include <stdio.h>

// this function returns minimum of integer
// numbers passed. First argument is count
// of numbers.
int min(int arg_count, ...)
{
    int i;
    int min, a;

    // va_list is a type to hold information about
    // variable arguments
    va_list ap;

    // va_start must be called before accessing
    // variable argument list
    va_start(ap, arg_count);

    // Now arguments can be accessed one by one
    // using va_arg macro. Initialize min as first
    // argument in list
    min = va_arg(ap, int);

    // traverse rest of the arguments to find out minimum
    for (i = 2; i <= arg_count; i++)
        if ((a = va_arg(ap, int)) < min)
            min = a;

    // va_end should be executed before the function
    // returns whenever va_start has been previously
    // used in that function
    va_end(ap);

    return min;
}

int main()
{
    int count = 5;
    printf("Minimum value is %d", min(count, 12, 67, 6, 7, 100));
    return 0;
}

```

Example 3:

// C program to demonstrate working of variable arguments to find average of multiple numbers.

```

#include <stdarg.h>

```

```

#include <stdio.h>

int average(int num, ...)
{
    va_list valist;

    int sum = 0, i;

    va_start(valist, num);
    for (i = 0; i < num; i++)
        sum += va_arg(valist, int);

    va_end(valist);

    return sum / num;
}

int main()
{
    printf("Average of {2, 3, 4} = %d\n",
           average(2, 3, 4));
    printf("Average of {3, 5, 10, 15} = %d\n",
           average(3, 5, 10, 15));

    return 0;
}

```

Output:

Average of {2, 3, 4} = 3

Average of {3, 5, 10, 15} = 10

STATEMENTS

C has three types of statement.

1. assignment
 =
2. selection (branching)
 if (expression)
 else
 switch
3. iteration (looping)
 while (expression)
 for (expression;expression;expression)
 do {block}

BLOCKS

These statements are grouped into *blocks*, a block is identified by curly brackets...There are two types of block.

- statement blocks
 if (i == j)

```

    {
        printf("martin \n");
    }

```

The *statement block* containing the **printf** is only executed if the **i == j expression** evaluates to TRUE.

- function blocks


```

int add( int a, int b)    /* Function definition */
{
    int c;
    c = a + b;
    return c;
}

```

DECISION MAKING IN C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met

C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if....else statement
3. Nested if....else statement
4. Using else if statement

Simple if statement

The general form of a simple if statement is,

```

if(expression)
{
    statement inside;
}
statement outside;

```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```

#include <stdio.h>
int main( )
{

```

```

int x, y;
x = 15;
y = 13;
if (x > y )
{
    printf("x is greater than y");
}
return 0;
}

```

Output:
x is greater than y

if...else statement

The general form of a simple if...else statement is,

```

if(expression)
{
    statement block1;
}
else
{
    statement block2;
}

```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

```

#include <stdio.h>
int main( )
{
    int x, y;
    x = 15;
    y = 18;
    if (x > y )
    {
        printf("x is greater than y");
    }
    else
    {
        printf("y is greater than x");
    }
    return 0;
}

```

output:
y is greater than x

Nested if....else statement

The general form of a nested if...else statement is,

```

if( expression )

```



```

{
    if( expression1 )
    {
        statement block1;
    }
    else
    {
        statement block2;
    }
}
else
{
    statement block3;
}

```

if expression is false then statement-block3 will be executed, otherwise the execution continues and enters inside the first if to perform the check for the next if block, where if expression 1 is true the statement-block1 is executed otherwise statement-block2 is executed.

Example:

```

#include <stdio.h>
int main( )
{
    int a, b, c;
    printf("Enter 3 numbers...");
    scanf("%d%d%d",&a, &b, &c);
    if(a > b)
    {
        if(a > c)
        {
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    return 0;
}

```

else-if ladder

The general form of else-if ladder is,

```
if(expression1)
{
    statement block1;
}
else if(expression2)
{
    statement block2;
}
else if(expression3 )
{
    statement block3;
}
else
    default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example

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

```
int main( )
{
    int a;
    printf("Enter a number...");
    scanf("%d", &a);
    if(a%5 == 0 && a%8 == 0)
    {
        printf("Divisible by both 5 and 8");
    }
    else if(a%8 == 0)
    {
        printf("Divisible by 8");
    }
    else if(a%5 == 0)
    {
        printf("Divisible by 5");
    }
    else
    {
        printf("Divisible by none");
    }
    return 0;
}
```

Points to Remember

In if statement, a single statement can be included without enclosing it into curly braces { ... }

```
int a = 5;
if(a > 4)
    printf("success");
```

No curly braces are required in the above case, but if we have more than one statement inside if condition, then we must enclose them inside curly braces.

== must be used for comparison in the expression of if condition, if you use = the expression will always return true, because it performs assignment not comparison.
Other than 0(zero), all other values are considered as true.

```
if(27)
    printf("hello");
```

In above example, hello will be printed.

Switch statement in C

When you want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and you need to choose only one at a time, then, switch statement is used.

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in switch evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present). The general form of switch statement is,

```
switch(expression)
{
    case value-1:
        block-1;
        break;
    case value-2:
        block-2;
        break;
    case value-3:
        block-3;
        break;
    case value-4:
        block-4;
        break;
    default:
        default-block;
        break;
}
```

Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. break statements are used to **exit** the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.

```
int i = 1;
switch(i)
{
    case 1:
        printf("A");    // No break
    case 2:
        printf("B");    // No break
    case 3:
        printf("C");
        break;
}
```

output:
A B C

1. The output was supposed to be only **A** because only the first case matches, but as there is no break statement after that block, the next blocks are executed too, until it a break statement is encountered or the execution reaches the end of the switch block.
2. **default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.
3. Nesting of switch statements are allowed, which means you can have switch statements inside another switch block. However, nested switch statements should be avoided as it makes the program more complex and less readable.

Example of switch statement

```
#include<stdio.h>
int main( )
{
    int a, b, c, choice;
    while(choice != 3)
    {
        /* Printing the available options */
        printf("\n 1. Press 1 for addition");
```

```

printf("\n 2. Press 2 for subtraction");
printf("\n Enter your choice");
/* Taking users input */
scanf("%d", &choice);

switch(choice)
{
    case 1:
        printf("Enter 2 numbers");
        scanf("%d%d", &a, &b);
        c = a + b;
        printf("%d", c);
        break;
    case 2:
        printf("Enter 2 numbers");
        scanf("%d%d", &a, &b);
        c = a - b;
        printf("%d", c);
        break;
    default:
        printf("you have passed a wrong key");
        printf("\n press any key to continue");
}
}

return 0;

}

```

Difference between switch and if,

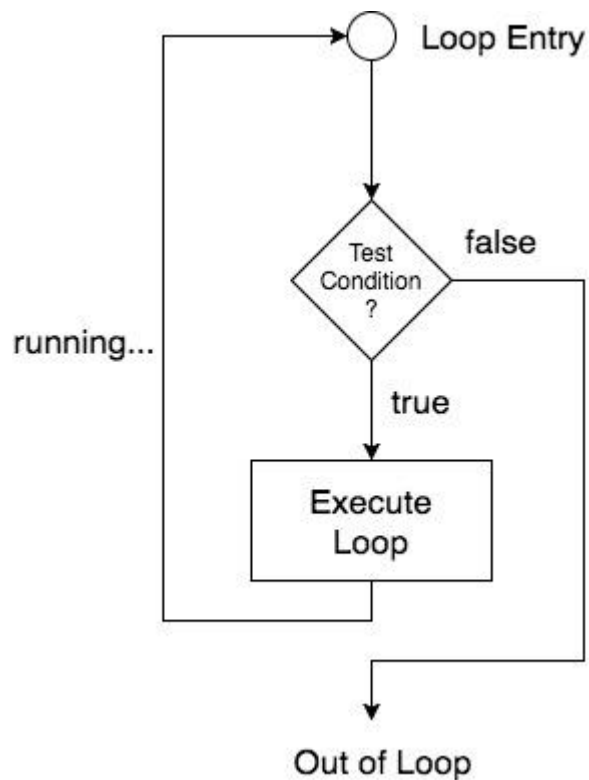
- if statements can evaluate float conditions. switch statements cannot evaluate float conditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

How to use Loops in C

In any programming language including C, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

How it Works

The below diagram depicts a loop execution,



As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

The sequence of statements to be executed is kept inside the curly braces { } known as the **Loop body**. After every execution of the loop body, **condition** is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed, and execution breaks out of the loop.

Types of Loop

There are 3 types of Loop in C language, namely:

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

while loop

while loop can be addressed as an entry control loop. It is completed in 3 steps.

Variable initialization.(e.g int x = 0;)

condition(e.g while(x <= 10))

Variable increment or decrement (x++ or x-- or x = x + 2)

Syntax :

variable initialization;

```

while(condition)
{
    statements;
    variable increment or decrement;
}

```

Example: Program to print first 10 natural numbers

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

```
int main( )
```

```

{
    int x;
    x = 1;
    while(x <= 10)
    {
        printf("%d\t", x);
        /* below statement means, do x = x+1, increment x by 1*/
        x++;
    }

    return 0;
}

```

output

1 2 3 4 5 6 7 8 9 10

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be **false**. General syntax is,

```

do
{
    .....
}
while(condition);

```

Example: Program to print first 10 multiples of 5.

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

```
int main()
```

```

{
    int a, i;
    a = 5;
    i = 1;
}

```

```

do
{
    printf("%d\t", a*i);
    i++;
}
while(i <= 10);
return 0;
}

```

Output:

5 10 15 20 25 30 35 40 45 50

for loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop**.. General format is,
for(initialization; condition; increment/decrement)

```

{
    statement-block;
}

```

The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example: Program to print first 10 natural numbers

```

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

```

output

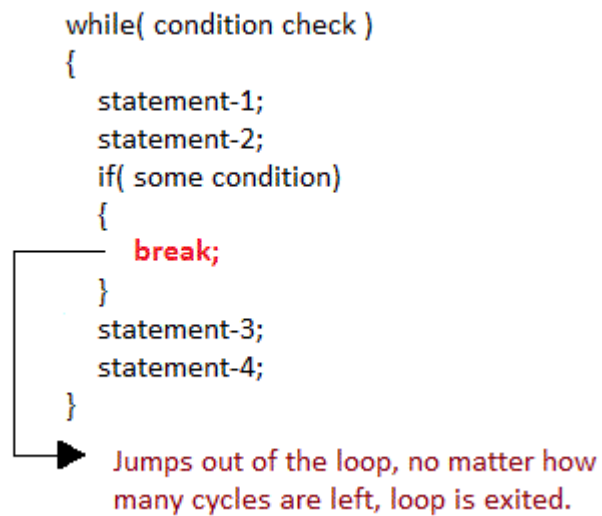
1 2 3 4 5 6 7 8 9 10

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

1) break statement

When **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.



Example:

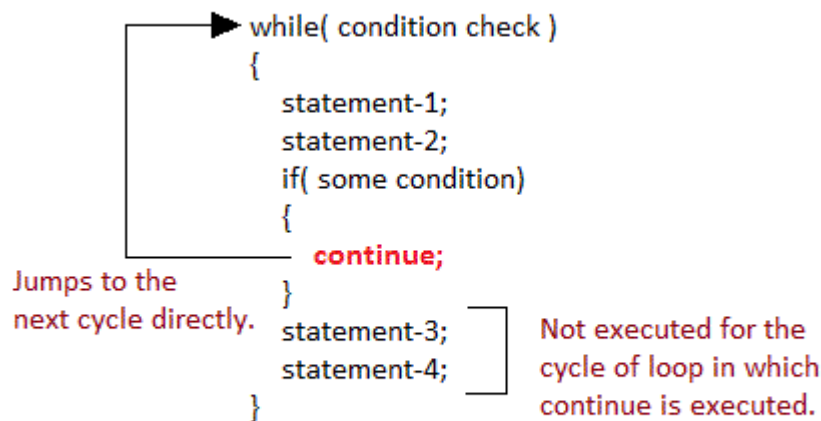
```
#include<stdio.h>
int main ()
{
    int i;
    for(i = 0; i<10; i++)
    {
        printf("%d ",i);
        if(i == 5)
            break;
    }
    printf("came outside of loop i = %d",i);
    return 0;
}
```

output:

0 1 2 3 4 5 came outside of loop i = 5

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering **continue**, cursor leave the current cycle of loop, and starts with the next cycle.



```

#include<stdio.h>
int main ()
{
    int i = 0;
    while(i!=10)
    {
        printf("%d", i);
        continue;
        i++;
    }
    return 0;
}

```

1 2 3 4 6 7 8 9 10

As you can see, 5 is not printed on the console because loop is continued at $i==5$.

goto labels

A **goto** statement in C programming 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 to avoid them.

Syntax

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

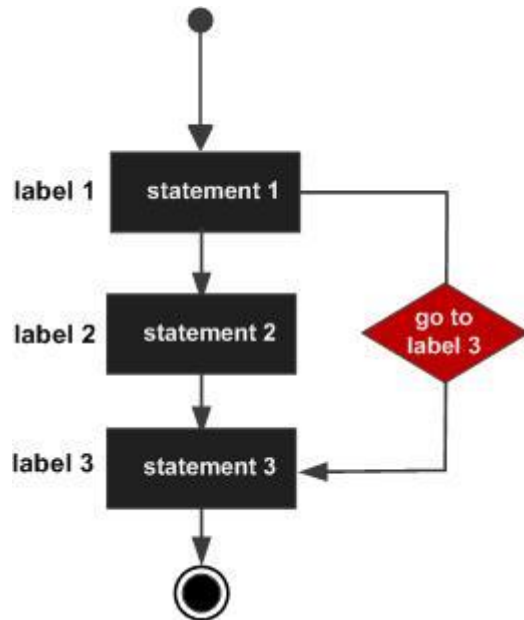
```

goto label;
..

```

label: statement;

here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.



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

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    LOOP:do {

        if( a == 15) {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }

        printf("value of a: %d\n", a);
        a++;

    }while( a < 20 );

    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: 16
 value of a: 17
 value of a: 18
 value of a: 19

<p>1. <u>AREA AND CIRCUMFERENCE OF A CIRCLE</u></p> <pre>#include<stdio.h> int main() { float r,a,c; printf("enter the radius of the circle"); scanf("%f",&r); a=3.14*r*r; c=2*3.14*r; printf("area %f",a); printf("circumference %f",c); return 0; }</pre> <p>Output : area 78.500000 circumference 31.400000</p>	<p>2. <u>CONVERSION BETWEEN CELSIUS AND FAHRENHEIT</u></p> <pre>#include<stdio.h> main () { float f, c; printf("enter Celsius\n"); scanf("%f",&c); f= (1.8*c) +32; printf("\n Fahrenheit value %f", f); printf("enter the Fahrenheit\n"); scanf("%f",&f); c= (f-32)/1.8; printf("celcius %f ",c); return 0; }</pre> <p>Output : Enter Celsius 5 Fahrenheit value 41.000000 Enter the Fahrenheit 20 Celsius is -6.666667</p>
<p>3. <u>ODD OR EVEN NUMBER</u></p> <pre>#include<stdio.h> int main() { int n ; printf("Enter a number "); scanf("%d", &n); if(n%2==0) printf("the number is even"); else printf("the number is odd"); return 0; }</pre>	<p>4. <u>LEAP YEAR CACLULATION</u></p> <pre>#include<stdio.h> int main() { int n ; printf("Enter a year "); scanf("%d", &n); if(n%4==0) printf("Leap year"); else printf("Not a leap year"); return 0; }</pre>

<pre> } </pre> <p>Output : Enter a number 10 the number is even</p>	<pre> } </pre> <p>Output : Enter a year 1996 Leap year</p>
<p>5. <u>SWAPPING TWO NUMBERS</u></p> <pre> #include<stdio.h> int main() { int a,b,temp; printf("Enter a, b"); scanf("%d%d",&a,&b); printf("Before swap %d %d", a, b); temp = a; a = b; b = temp; printf("After swap %d %d", a, b); return 0; } </pre> <p>Output : Enter a, b 10 20 Before swap 10 20 After swap 20 10</p>	<p>6. <u>SWAPPING WITHOUT TEMPORARY VARIABLES</u></p> <pre> #include<stdio.h> int main() { int a,b; printf("Enter a, b"); scanf("%d%d",&a,&b); printf("Before swap %d %d", a, b); a=a+b; b=a-b; a=a-b; printf("After swap %d %d", a, b); return 0; } </pre> <p>Output : Enter a, b 10 20 Before swap 10 20 After swap 20 10</p>
<p>7. <u>GREATEST OF THREE NUMBERS</u></p> <pre> #include<stdio.h> int main() { int a,b,c; printf("enter the three numbers"); scanf("%d%d%d",&a,&b,&c); if(a>b && a>c) printf("%d is biggest",a); else if(b>c) printf("%d is biggest",b); else printf("%d is biggest",c); } </pre>	<p>8. <u>PROGRAM TO CHECK PRIME NUMBER</u></p> <pre> int main() { int n,i=2; printf("Enter the number\n"); scanf("%d",&n); while(i<n) { if(n% i == 0) { printf("It not a prime number"); break; } i++; } } </pre>

<pre>} Output: Enter the three numbers 20 10 15 20 is biggest</pre>	<pre>if(i== n) printf("Prime number"); } Output : Enter the number 7 Prime number</pre>
<p style="text-align: center;">9. <u>ARITHMETIC OPERATORS</u></p> <pre>int main() { int a,b, choice; printf("enter two numbers:"); scanf("%d%d",&a,&b); printf("1.add\n2.subtract\n3.multiply\n4.division\n5.modulo"); printf("enter your choice"); scanf("%d",&choice); switch(choice) { case 1: printf("sum is %d",a+b); break; case 2: printf("difference is %d",a-b); break; case 3: printf("product is %d",a*b); break; case 4: printf("quotient is %d",a/b); break; case 5: printf("modulo is %d",a%b); break; default: printf("invalid operation"); break; } return 0; }</pre> <p>Output :</p> <pre>Enter two numbers: 10 20 1.add 2.subtract 3.multiply</pre>	<p style="text-align: center;">10. <u>PROGRAM TO REVERSE A NUMBER & TO CHECK PALINDROME NUMBER</u></p> <pre>#include <stdio.h> int main() { int n,a,rev=0,rem; printf("Enter the number \n"); scanf("%d",&n); a=n; while(n>0) { rem=n%10; rev=(rev*10)+rem; n=n/10; } printf("The reverse number of the %d is %d\n",a, rev); if(a==rev) printf("%d is palindrome\n",a); else printf("%d is not a palindrome\n",a); return 0; }</pre> <p>Output:</p> <pre>Enter the number 121 The reverse number of the 121 is 121 121 is a palindrome</pre>

4.division 5.modulo enter your choice 1 sum is 30	
<p align="center">11. <u>PROGRAM TO SUM OF N NATURAL NUMBERS</u></p> <pre>#include<stdio.h> int main() { int i,n,sum; i=1; sum=0; printf("Enter the value of n \n"); scanf("%d",&n); while(i<=n) { sum=sum+i; i = i+1; } printf("The sum of numbers is %d ",sum); return 0; }</pre> <p>Output : Enter the value of n : 5 The sum of numbers is 15</p>	<p align="center">12. <u>PROGRAM TO CALCULATE FACTORIAL OF A NUMBER</u></p> <pre>#include<stdio.h> int main() { int n, fact=1,i; printf("Enter the number"); scanf("%d",&n); for(i=1;i<=n;i++) { fact=fact*i; } printf("The factorial of %d is %d",n,fact); return 0; }</pre> <p>Output : Enter the number 5 The factorial of 5 is 120</p>
<p align="center">13. <u>PROGRAM TO CALCULATE SUM OF INDIVIDUAL DIGITS OF A NUMBER</u></p> <pre>#include<stdio.h> int main() { int n,rem,sum=0; printf("Enter n :"); scanf("%d",&n); while(n>0)</pre>	<p align="center">14. <u>PROGRAM TO CALCULATE SUM OF SQUARE OF INDIVIDUAL DIGITS OF A NUMBER</u></p> <pre>#include<stdio.h> int main() { int n,rem,sum=0; printf("Enter n :"); scanf("%d",&n); while(n>0)</pre>

```
{
rem=n%10;
sum=sum+rem;
n=n/10;
}
printf("\nsum of digits is %d",sum);
return 0;
}
```

Output :

Enter n 123
sum of digits is 6

```
{
rem=n%10;
sum=sum+(rem+rem);
n=n/10;
}
printf("\nsum of square of digits is %d",sum);
return 0;
}
```

Output :

Enter n 123
sum of square of digits is 14

15. PROGRAM TO CHECK ARMSTRONG NUMBER

```
#include<stdio.h>
int main()
{
int n,a,rem,sum=0;

printf("Enter n :");
scanf("%d",&n);
a=n;
while(n>0)
{
    rem=n%10;
    sum=sum+(rem*rem*rem);
    n=n/10;
}
if(a == sum)
    printf("\n It is an Armstrong
number");
else
    printf("\n It is not an Armstrong
number");
return 0;
}
```

Output :

Enter n 153
It is an Armstrong number

16. PROGRAM TO GENERATE FIBONACCI SERIES

```
#include<stdio.h>
int main()
{
int a=-1,b=1,c=0,n,i;

printf("Enter the number");
scanf("%d",&n);
for(i=0;i<n;i++)
{
    c=a+b;
    printf("\n %d",c);
    a=b;
    b=c;
}
return 0;
}
```

Output :

Enter the number: 6
0 1 1 2 3 5

Basics of functions, parameter passing and returning type, External, Auto, Local, Static, Register Variables, Scope Rules, Block structure, Initialisation, Recursion, C Preprocessor, Standard Library Functions and return types.

BASICS OF FUNCTIONS

A C program is nothing but a combination of one or more functions. Every C program starts with a user defined function main().

The C language supports two types of functions. They are:

1. Library Functions.
2. User defined functions

Library functions are pre-defined set of functions. The users can use the functions but cannot modify or change them.

User defined functions are defined by the user according to his/her requirements. The user can modify the functions. The user understands the internal working of the functions.

Definition of Function:

A function is a self-contained block or a sub program of one or more statements that performs a specific task when called.

The general syntax is as follows:

```
<data type of the return value> function name(argument/parameter list)
{
variable declaration;
statement 1;
statement 2;
return(value);
}
```

Working of a Function

The working of a function is given below:

int abc(int l, int k); **Function Declaration**

void main()

```
{
    ----;
    ----;
    abc(x,y);
```

Function call

x, y are actual arguments

```
    ----
    ----
}
```

int abc(int l, int k) **Function Definition**

```
{
    ---
    ---
    ---
}
```

l, k are formal or dummy arguments

```

    ----
    ----
    return();      return value
}

```

1. **Function Declaration:** A function declaration tells the compiler about a function name and how to call the function. Function declaration contains,
 - a. *Return Type:* A function may return a value. The return type is the data type of the value the function returns.
 - b. *Function Name:* A name given to the function similar to a name given to a variable.
 - c. *Argument/Parameter List:* The variable name enclosed within the parenthesis is the argument list.

General Form :

```
return_type function_name( parameter list );
```

2. **Function Definition:** A function definition provides the actual body of the function. A function definition in C programming consists of a *function header* and a *function body*.
 - a. *Function Header:* Function Header contains Return Type, Function Name, and Argument/Parameter List similar to Function Declaration.
 - b. *Function Body:* The function body contains a collection of statements that define what the function does. Function Body also has a return statement.
 - c. *Return value:* The result obtained by the function is sent back by the function to the function call through the return statement. It returns one value per call.

General Form :

```

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

```

3. **Function call:** To use a function, you will have to call that function to perform the defined task. To call a function, you simply need to pass the required parameters(actual) along with the function name, and if the function returns a value, then you can store the returned value.

General Form:

```
function_name(parameters);
```

4. **Arguments:** There are two types of arguments. They are
 - a. *Actual arguments:* The arguments of calling functions are called actual arguments.
 - b. *Formal/Dummy arguments:* The arguments of called functions are called formal or dummy arguments.

5. **Variables:** There are two kinds of variables. They are

- a. *Local variables:* The variables which are declared inside the function definition is called local variable.
- b. *Global Variable:* Variables which are declared outside the main function is called global variable.

SCOPE RULES IN C

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language.

- Inside a function or a block which is called **local variables**.
- Outside of all functions which is called **global variables**.
- In the definition of function parameters which are called **formal parameters**.

(a) Local Variables :

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Example:

```
#include <stdio.h>

int main ()
{
    /* local variable declaration */
    int a, b;
    int c;
    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Here all the variables a, b, and c are local to main() function.

(b) Global Variables :

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Example:

```
#include <stdio.h>

/* global variable declaration */
int g;

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

    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference.

Example:

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
{
    /* local variable declaration */
    int g = 10;
    printf ("value of g = %d\n", g);
    return 0;
}
```

Output:

value of g = 10

(c) Formal Parameters :

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

```
int a = 20;
int main ()
{
/* local variable declaration in main function */
int a = 10;
int b = 20;
int c = 0;
printf ("value of a in main() = %d\n", a);
c = sum( a, b);
printf ("value of c in main() = %d\n", c);
return 0;
}
/* function to add two integers */
int sum(int a, int b)
{
/*formal parameters */
printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b);
return a + b;
}
```

Output:

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

INITIALIZATION OF LOCAL AND GLOBAL VARIABLES

When a local variable is defined, it is not initialized by the system, you must initialize it yourself.

Global variables are initialized automatically by the system when you define them as follows,

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly; otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

RETURNING FUNCTION RESULTS

This is done by the use of the keyword return, followed by a data variable or constant value. The datatype of the data variable or constant value must match with that of the declared return_data_type for the function.

```
float add_numbers(floatn1,floatn2)
{
return n1+n2;           /*legal*/
return 6;               /*illegal, not the same datatype*/
return 6.0;             /*legal*/
}
```

It is possible for a function to have multiple return statements, provided only one return statement will be executed. In the below example, based on user input only one return statement will be executed.

```
int validate_input(char command)
{
    switch(command)
    {
        case '+':
        case '-':return 1;
        case '*':
        case '/':return 2;
        default :return 0;
    }
}
```

PARAMETER PASSING

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 definition for 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 Statement of the function max() */
    return result;
}
```


FUNCTION PROTOTYPES

The Function prototypes are classified into four types. They are:

1. Function with no argument and no return value.
2. Function with no argument and with return value.
3. Function with argument and no return value.
4. Function with argument and with return value.

1. Function with no argument and no return value:

- ❑ Neither data is passed through the calling function nor the data is sent back from the called function.
- ❑ There is no data transfer between calling and the called function.
- ❑ The function is only executed and nothing is obtained.
- ❑ The Function acts independently. It reads data values and print result in the same block.

Example program:

```
#include<stdio.h>
void add(); /* No parameter-list No return type*/
void main()
{
    clrscr();
    add();
    getch();
}
void add()
{
    int a,b,c;
    a=10;
    b=5;
    c=a+b;
    printf("\nThe sum=%d",c);
}
```

OUTPUT:

The sum=15

2. Function with no argument and with return value

- ❑ In this type of function, no arguments are passed through the main function. But the called function returns the value.

- ❑ The called function is independent. It reads values from the keyboard and returns value to the function call.
- ❑ Here both the called and calling functions partly communicate with each other.

Example program:

```
#include<stdio.h>
int add(); /* No parameter-list but only return type*/
void main()
{
    int a,b,c;
    clrscr();
    c=add();
    printf("\nThe sum=%d",c);
    getch();
}
int add()
{
    int a,b,c;
    a=10;
    b=5;
    c=a+b;
    return(c);
}
```

OUTPUT:

The sum=15

3. Function with argument and no return value

- ❑ In this type of function, arguments are passed through the calling function. The called function operates on the values. But no result is sent back.
- ❑ The functions are partly dependent on the calling function. The result obtained is utilized by the called function.

Example program:

```
#include<stdio.h>
void add(int,int); /* parameter-list is given but no return type*/
void main()
```

```

{
    int a,b;
    clrscr();
    a=10;b=5;
    add(a,b);
    getch();
}

void add(int a,int b)
{
    int c;
    c=a+b;
    printf("\nThe sum=%d",c);
}

```

OUTPUT:

The sum=15

4. Function with argument and with return value

- ❑ In this type of function, data is transferred between calling and called function.
- ❑ Both communicate with each other by passing parameters to the called function and return values to the called function.

Example Program:

```

#include<stdio.h> /* parameter-list and return type both are given*/
int add(int,int);
void main()
{
    int a,b,c;
    clrscr();
    a=10;b=5;
    c=add(a,b);
    printf("\nThe sum=%d",c);
    getch();
}

void add(int a,int b)
{
    int c;
    c=a+b;
    return(c);
}

```

OUTPUT:

The sum=15

PARAMETER PASSING METHODS

There are two ways by which arguments are passed in the function. They are:

1. Call by value.
2. Call by reference.

1. Call by value:

In this type, values of actual arguments are passed to the formal arguments and the operation is done on the dummy arguments. Any change made in the formal arguments does not affect the actual arguments because formal arguments are photocopies of actual arguments.

Example Program:

```
#include<stdio.h>
void swap(int,int);
void main()
{
    int a,b;
    clrscr();
    a=10;
    b=5;
    swap(a,b);
    printf("\nAfter swapping\n");
    printf("Actual Arguments\n");
    printf("a=%d",a);
    printf("\nb=%d",b);
    getch();
}
void swap(int a,int b)
{
```

```
int c;  
c=a;  
a=b;  
b=c;  
printf("After swapping\n");  
printf("Formal Arguments\n");  
printf("a=%d",a);  
printf("\nb=%d",b);  
}
```

OUTPUT:

After swapping

Formal Arguments

a=5

b=10

After swapping

Actual Arguments

a=10

b=5

2. Call by Reference:

In this type, addresses are passed. Function operates on addresses rather than values. Here, the formal arguments are pointers to the actual arguments. Here changes are made in the arguments as permanent. So, any change made to the formal arguments reflects actual arguments.

Example Program:

```
#include<stdio.h>  
void swap(int*,int*);  
void main()  
{  
int a,b;  
clrscr();  
a=10;b=5;  
swap(&a,&b);
```

```
printf("\nAfter swapping\n");
printf("Actual Arguments\n");
printf("\na=%d",a);
printf("\nb=%d",b);
getch();
}
void swap(int *x,int *y)
{
int c;
c=*x;
*x=*y;
*y=c;
}
```

OUTPUT:

After swapping
Actual Arguments
a=5
b=10

C – STORAGE CLASS SPECIFIERS

Storage Classes are used to describe the features of a variable/function. The features basically include lifetime, visibility, memory location, and initial value of a variable. It helps us to trace the existence of a particular variable during the runtime of a program.

SYNTAX: storage_specifier data-type variable _name

TYPES OF C – STORAGE CLASS SPECIFIERS:

There are 4 storage class specifiers available in C language.

1. auto
2. static
3. extern

4. register

S.No.	Storage Specifier	Storage place	Initial default value /	Scope	Life
1	Auto	CPU Memory	Garbage value	local	Within the function
2	static	CPU memory	Zero	local	Retains the value of the variable between different function calls.
3	Extern	CPU memory	Zero	Global	Till end of the main program. Variable definition might be anywhere in the C program
4	register	Register memory	Garbage value	local	Within the function

NOTE:

- For faster access of a variable, it is better to go for register specifiers rather than auto specifiers.
- Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.
- Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.

EXAMPLE PROGRAM FOR C AUTO VARIABLE:

The scope of this auto variable is within the function only. It is equivalent to local variable. All local variables are auto variables by default. Hence, the keyword auto is rarely used while writing programs in C language. The memory assigned to automatic variables gets freed upon exiting from the block.

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

```

increment();
increment();
increment();
}
void increment(void)
{
auto int i = 0 ;
printf ( "%d \n", i ) ;
i++;
}

```

OUTPUT

```

0
0
0
0

```

EXAMPLE PROGRAM FOR C STATIC VARIABLE:

In static variables retain the value of the variable between different function calls. They are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

```

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

```



```
static int i = 0 ;  
printf ( "%d \n", i ) ;  
i++;  
}
```

OUTPUT

```
0  
1  
2  
3
```

EXAMPLE PROGRAM FOR EXTERN VARIABLE:

The scope of this extern variable is throughout the main program. It is equivalent to global variable. Definition for extern variable might be anywhere in the C program. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

```
#include<stdio.h>  
int x = 10 ;  
int main( )  
{  
extern int y ;  
printf ( "The value of x is %d \n", x ) ;  
printf ( "The value of y is %d", y ) ;  
}  
int y = 50 ;
```

OUTPUT

```
The value of x is 10  
The value of y is 50
```

EXAMPLE PROGRAM FOR REGISTER VARIABLE:

Register variables are also local variables, but stored in register memory. Whereas auto variables are stored in main CPU memory. Register variables will be accessed very faster than the normal

variables since they are stored in register memory rather than main memory. But, only limited variables only can be used as register since register size is very low. (16bits, 32 bits or 64 bits). An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

```
#include<stdio.h>
int main()
{
register int i, arr[5];    // declaring array
arr[0] = 10;    // Initializing array
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
for (i=0;i<5;i++)
{
    // Accessing each variable
    printf("value of arr[%d] is %d \n", i, arr[i]);
}
}
```

OUTPUT

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

RECURSION

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi

(TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Consider the calculation of 6!(factorial)

i.e., $6! = 6 * 5 * 4 * 3 * 2 * 1$

This is computed as,

$6! = 6 * 5!$ i.e., $6! = 6 * (6-1)!$

$5! = 5 * 4!$ i.e., $5! = 5 * (5-1)!$

:

:

$1! = 1 * 0!$

So,

$n! = n * (n-1)!$

Example:

/ example for demonstrating recursion */*

#include <stdio.h>

int factorial(int); */*function declaration/prototype*/*

*/*function definition*/*

int factorial(int n)

{

int result;

if(n==0)

result=1;

else

result = n * factorial(n-1); */* Recursive Call Factorial Call Itself */*

return result;

}

main()

{

int j;

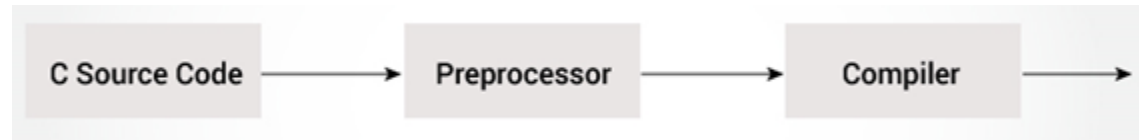
printf("Enter Number to find factorial");

scanf("%d",&j);

```
printf("%d",factorial(j));  
}
```

C PREPROCESSOR:

The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be inclusion of header file, macro expansions etc.



All preprocessing directives begins with a # symbol. For example,

```
#define PI 3.14
```

Some of the common uses of C preprocessor are:

Including Header Files

The #include preprocessor is used to include header files to a C program. For example,

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

Here, "stdio.h" is a header file. The #include preprocessor directive replaces the above line with the contents of stdio.h header file which contains function and macro definitions.

That's the reason why you need to use #include <stdio.h> before you can use functions like scanf() and printf().

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

```
#include "my_header.h"
```

Visit this page to learn on using header files.

Macros using #define

You can define a macro in C using #define preprocessor directive.

A macro is a fragment of code that is given a name. You can use that fragment of code in your program by using the name. For example,

```
#define c 299792458 // speed of light
```

Here, when we use c in our program, it's replaced with 299792458.

Example 1: #define preprocessor

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

```

#define PI 3.1415
int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    // Notice, the use of PI
    area = PI*radius*radius;
    printf("Area=%.2f",area);
    return 0;
}

```

You can also define macros that works like a function call, known as function-like macros. For example,

```

#define circleArea(r) (3.1415*(r)*(r))

```

Every time the program encounters circleArea(argument), it is replaced by (3.1415*(argument)*(argument)).

Suppose, we passed 5 as an argument then, it expands as below:

circleArea(5) expands to (3.1415*5*5)

Example 2: Using #define preprocessor

```

#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)
int main()
{
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);
}

```

```
    return 0;  
}
```

Conditional Compilation

In C programming, you can instruct preprocessor whether to include certain chunk of code or not. To do so, conditional directives can be used. It's similar like a if statement. The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals is used to include (or skip) certain chunks of code in your program before execution.

Uses of Conditional

use different code depending on the machine, operating system

compile same source file in two different programs

to exclude certain code from the program but to keep it as reference for future purpose

How to use conditional?

To use conditional, #ifdef, #if, #defined, #else and #elseif directives are used.

#ifdef Directive

#ifdef MACRO

conditional codes

#endif

Here, the conditional codes are included in the program only if MACRO is defined.

#if, #elif and #else Directive

#if expression

conditional codes

#endif

Here, expression is a expression of integer type (can be integers, characters, arithmetic expression, macros and so on). The conditional codes are included in the program only if the expression is evaluated to a non-zero value.

The optional `#else` directive can be used with `#if` directive.

`#if expression`

conditional codes if expression is non-zero

`#else`

conditional codes if expression is 0

`#endif`

You can also add nested conditional to your `#if...#else` using `#elif`

`#if expression`

conditional codes if expression is non-zero

`#elif expression1`

conditional codes if expression is non-zero

`#elif expression2`

conditional codes if expression is non-zero

... ..

`else`

conditional codes if all expressions are 0

`#endif`

`#defined`

The special operator `#defined` is used to test whether certain macro is defined or not. It's often used with `#if` directive.

`#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048`

conditional codes

Predefined Macros

There are some predefined macros which are readily for use in C programming.

Predefined macro	Value
<code>__DATE__</code>	String containing the current date
<code>__FILE__</code>	String containing the file name
<code>__LINE__</code>	Integer representing the current line number

<code>__STDC__</code>	If follows ANSI standard C, then value is a nonzero integer
<code>__TIME__</code>	String containing the current date.

Example 3: Get current time using `__TIME__`

The following program outputs the current time using `__TIME__` macro.

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

```
int main()
```

```
{
```

```
    printf("Current time: %s",__TIME__); //calculate the current time
```

```
}
```

Output

Current time: 19:54:39

C STANDARD LIBRARY FUNCTIONS

C Standard library functions or simply C Library functions are inbuilt functions in C programming. The prototype and data definitions of the functions are present in their respective header files, and must be included in your program to access them.

For example: If you want to use `printf()` function, the header file `<stdio.h>` should be included.

There is at least one function in any C program, i.e., the `main()` function (which is also a library function). This function is automatically called when your program starts.

Advantages of using C library functions

- 1. Easy to use:** These functions have gone through multiple rigorous testing and are easy to use.
- 2. The functions are optimized for performance:** Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.
- 3. It saves considerable development time:** It saves valuable time and your code may not always be the most efficient.

4. The functions are portable : With ever changing real world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer. This saves time, effort and makes your program portable.

C Library Functions Under Different Header File

C Header Files	
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

C <ctype.h> header

Header file <ctype.h> includes numerous standard library functions to handle characters (especially test characters).

Function	Description
isalnum()	checks alphanumeric character
isalpha()	checks whether a character is an alphabet or not
isctrl()	checks control character

Function	Description
<u>isdigit()</u>	checks numeric character
<u>isgraph()</u>	checks graphic character
<u>islower()</u>	checks lowercase alphabet
<u>isprint()</u>	checks printable character
<u>ispunct()</u>	checks punctuation
<u>isspace()</u>	check white-space character
<u>isupper()</u>	checks uppercase alphabet
<u>isxdigit()</u>	checks hexadecimal digit character
<u>tolower()</u>	converts alphabet to lowercase
<u>toupper()</u>	converts to lowercase alphabet

C <math.h> header

There are various standard library functions and a macro defined under <math.h> to perform mathematical operations in C programming.

Function	Description
<u>acos()</u>	computes arc cosine
<u>acosh()</u>	computes arc hyperbolic cosine
<u>asin()</u>	computes arc sine
<u>asinh()</u>	computes the hyperbolic of arc sine of an argument
<u>atan()</u>	computes the arc tangent of an argument
<u>atan2()</u>	computes the arc tangent of an argument.

Function	Description
<u>atanh()</u>	computes arc hyperbolic tangent
<u>cbrt()</u>	computes cube root of a number
<u>ceil()</u>	computes the nearest integer greater than argument
<u>cos()</u>	computes the cosine of an argument.
<u>cosh()</u>	computer hyperbolic cosine.
<u>exp()</u>	computes the exponential raised to the argument
<u>fabs()</u>	computes absolute value
<u>floor()</u>	calculates the nearest integer less than argument
<u>hypot()</u>	computes hypotenuse
<u>log()</u>	computes natural logarithm of an argument.
<u>log10()</u>	computes the base 10 logarithm of an argument.
<u>pow()</u>	Computes power of a number
<u>sin()</u>	compute sine of a number
<u>sinh()</u>	computes the hyperbolic sine of an argument.
<u>sqrt()</u>	computes square root of a number
<u>tan()</u>	computes tangent
<u>tanh()</u>	computes the hyperbolic tangent of an argument

C <string.h> header

There are various standard library functions and a macro defined under <string.h> to manipulate and perform operations on strings and array of characters in C programming.

Function	Description
----------	-------------

Function	Description
<code>strcat()</code>	Concatenates two strings
<code>strcmp()</code>	compares two strings
<code>strcpy()</code>	copies string
<code>strlen()</code>	calculates the length of a string
<code>strrev()</code>	Finds the reverse of a string

UNIT – V

Pointers and addresses, Pointers and Function Arguments, Pointers and Arrays, Address Arithmetic, character Pointers and Functions, Pointer Arrays, Pointer to Pointer, Multi-dimensional arrays, Strings, Initialisation of Pointer Arrays, Command line arguments, Pointers to functions, complicated declarations. Basic Structures, Structures and Functions, Array of structures, Pointer of Structures, Self-referential Structures, Table look up, typedef, Unions, Bit-fields, File Access -Error Handling, Line I/O, Miscellaneous Functions.

POINTER

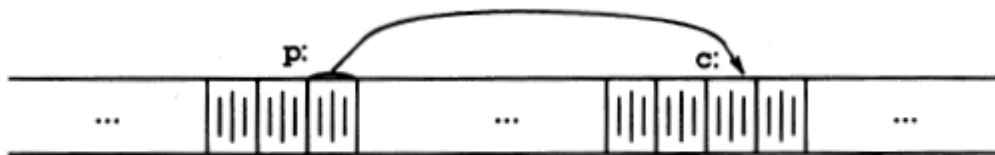
A **pointer** is a variable that contains the address of a variable. Pointers are much used in C, partly, because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways.

POINTERS AND ADDRESSES

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups.

One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long.

A pointer is a group of cells (often two or four) that can hold an address. So if *c* is a char and *p* is a pointer that points to it, we could represent the situation this way:



The unary operator **&** gives the address of an object, so the statement

```
p = &c;
```

assigns the address of *c* to the variable *p*, and *p* is said to "point to" *c*. The **&** operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator ***** is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that *x* and *y* are integers and *ip* is a pointer to into this artificial sequence shows how to declare a pointer and how to use **&** and *****:

```
int x = 1, y = 2, z[10];
int *ip;           /* ip is a pointer to int */

ip = &x;           /* ip now points to x */
y = *ip;           /* y is now 1 */
*ip = 0;           /* x is now 0 */
ip = &z[0];        /* ip now points to z[0] */
```

The declarations of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression *ip is an *int*. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression *dp and atof(s) have values of type double, and that the argument of atof is a pointer to char.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a "pointer to void" is used to hold any type of pointer but cannot be dereferenced itself.)

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments *ip by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1, and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

POINTERS AND FUNCTION ARGUMENTS

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called swap. It is not enough to write

swap(a, b);

where the swap function is defined as

```
void swap(int x, int y) /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above only swaps copies of a and b.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed

swap(&a, &b);

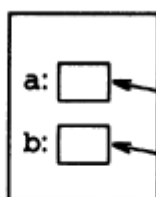
Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;

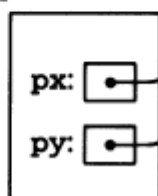
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pictorially,

in caller:



in swap:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input.

These values have to be passed back by separate paths, for no matter what value is used for EOF, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function.

The following loop fills an array with integers by calls to `getint`:

```
int n, array[SIZE], getint(int *);
```

```
for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++);
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`.

Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns EOF for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch())) /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* it's not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Throughout `getint`, `*pn` is used as an ordinary `int` variable. so the one extra character that must be read can be pushed back onto the input.

POINTERS AND ARRAYS

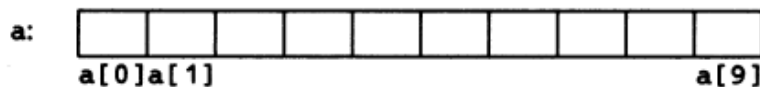
In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers.

The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array **a** of size 10, that is, a block of 10 consecutive objects named **a[0]**, **a[1]**, ..., **a[9]**.



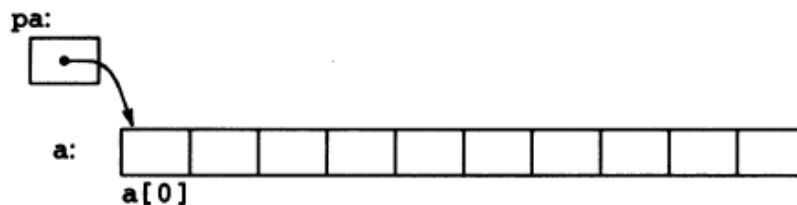
The notation **a[i]** refers to the *i*-th element of the array. If **pa** is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets **pa** to point to element zero of **a**; that is, **pa** contains the address of **a[0]**.



Now the assignment

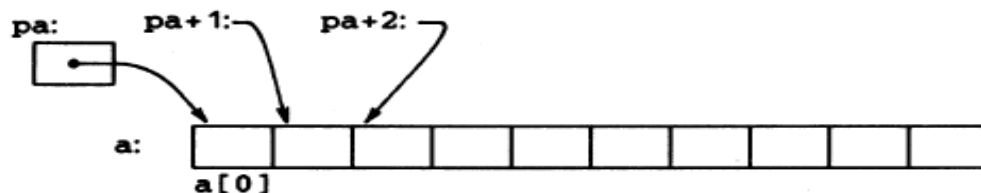
```
x = *pa;
```

will copy the contents of **a[0]** into **x**.

If **pa** points to a particular element of an array, then by definition **pa+1** points to the next element, **pa+i** points *i* elements after **pa**, and **pa-i** points *i* elements before. Thus, if **pa** points to **a[0]**,

```
*(pa+1)
```

refers to the contents of **a[1]**, **pa+i** is the address of **a[i]**, and ***(pa+i)** is the contents of **a[i]**.



These remarks are true regardless of the type or size of the variables in the array `a`. The meaning of “adding 1 to a pointer,” and by extension, all pointer arithmetic, is that `pa+1` points to the next object, and `pa+i` points to the *i*-th

object beyond `pa`.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

Rather more surprising, at least at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent. Applying the operator `&` to both parts of this equivalence, it follows that `&a[i]` and `a+i` are also identical: `a+i` is the address of the *i*-th element beyond `a`. As the other side of this coin, if `pa` is a pointer, expressions may use it with a subscript; `pa[i]` is identical to `*(pa+i)`. In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
strlen("hello, world"); /* string constant */
strlen(array);          /* char array[100]; */
strlen(ptr);            /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if *a* is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function *f* the address of the subarray that starts at *a*[2]. Within *f*, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as *f* is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; *p*[-1], *p*[-2], and so on are syntactically legal, and refer to the elements that immediately precede *p*[0]. Of course, it is illegal to refer to objects that are not within the array bounds.

ADDRESS ARITHMETIC

If *p* is a pointer to some element of an array, then *p*++ increments *p* to point to the next element, and *p*+=*i* increments it to point *i* elements beyond where it currently does. These and similar constructions are the simplest forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language.

Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, *a110c* (*n*), returns a pointer *p* to *n* consecutive character positions, which can be used by the caller of *a110c* for storing characters.

The second, *afree* (*p*), releases the storage thus acquired so it can be re-used later. The routines are "rudimentary" because the calls to a *free* must be made in the opposite order to the calls made on *a110c*. That is, the storage managed by *alloc* and *a free* is a stack, or last-in, first-out list.

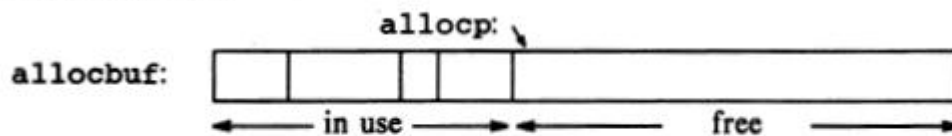
The standard library provides analogous functions called *malloc* and *free* that have no such restrictions; The easiest implementation is to have *alloc* hand out pieces of a large character array that we will call *allocbuf*. This array is private to *alloc* and *afree*. Since they deal in pointers, not array indices, no other

routine need know the name of the array, which can be declared static in the source file containing alloc and afree, and thus be invisible outside it.

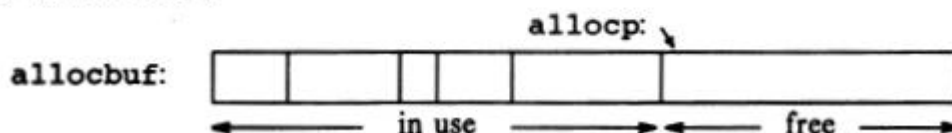
In practical implementations, the array may well not even have a name; it might instead be obtained by calling malloc or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of allocbuf has been used. We use a pointer, called allocp that points to the next free element. When alloc is asked for n characters, it checks to see if there is enough room left in allocbuf. If so, alloc returns the current value of allocp (i.e., the beginning of the free block), then increments it by n to point to the next free area. If there is no room, alloc returns zero. afree (p) merely sets allocp to p if p is inside allocbuf.

before call to alloc:



after call to alloc:



In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the addresses of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines allocp to be a character pointer and initializes it to point to the beginning of allocbuf, which is the next free position when the program starts. This could have also been written

```
static char *allocp = &allocbuf[0];
```

```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(int n) /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else /* not enough room */
        return 0;
}

void afree(char *p) /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

since the array name is the address of the zeroth element. The test

```
if (allocbuf + ALLOCSIZE - allocp >= n)
```

checks if there's enough room to satisfy a request for *n* characters. If there is, the new value of *allocp* would be at most one beyond the end of *allocbuf*.

If the request can be satisfied, *alloc* returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, *alloc* must return some signal that no space is left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case, no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant *NULL* is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. *NULL* is defined in *<stdio.h>*. We will use *NULL* henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n)
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If *p* and *q* point to members of the same array, then relations like *==*, *!=*, *<*, *>=*, etc., work properly. For example,

```
p < q
```

is true if *p* points to an earlier member of the array than *q* does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the *n*-th object beyond the one *p* currently points to. This is true regardless of the kind of object *p* points to; *n* is scaled according to the size of the objects *p* points to, which is determined by the declaration of *p*. If an *int* is four bytes, for example, the *int* will be scaled by four.

Pointer subtraction is also valid: if *p* and *q* point to elements of the same array, and *p < q*, then *q - p + 1* is the number of elements from *p* to *q* inclusive. This fact can be used to write yet another version of *strlen*:

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

In its declaration, `p` is initialized to `s`, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the, `'0'` at the end is seen. Because `p` points to characters, `p++` advances `p` to the next character each time and `p - s` gives the number of characters advanced over, that is, the string length.

Pointer arithmetic is consistent: if we had been dealing with floats, which occupy more storage than chars, and if `p` were a pointer to float, `p++` would advance to the next float. Thus we could write another version of `alloc` that maintains floats instead of chars, merely by changing `char` to `float` throughout `alloc` and `afree`. All the pointer manipulations automatically take into account the size of the object pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for `void *`, to assign a pointer of one type to a pointer of another type without a cast.

CHARACTER POINTERS AND FUNCTIONS

A string constant, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character `'\0'` so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is not a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions

```
char amessage[] = "now is the time"; //an array
```

```
char *pmessage = "now is the time"; //a pointer
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may

subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents

The first function is `strcpy (s , t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version is first

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

For contrast, here is a version of `strcpy` with pointers:

```
/* strcpy:  copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied to `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up to and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```

/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}

```

Although this may seem cryptic at first sight, the notational conveniences considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The strcpy in the standard library «string. h» returns the target string as its function value. The second routine that we will examine is strcmp(s,t), which compares the character strings sand t, and returns negative, zero or positive if s is lexicographically less than, equal to, or greater than t. The value is obtained by subtracting the characters at the first position where s and t disagree

```

/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}

```

The pointer version of strcmp:

```

/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}

```

Since ++ and -- are either prefix or postfix operators, other combinations of * and ++ and -- occur, although less frequently. For example,

```

*--p

```

decrements p before fetching the character that p points to. In fact, the pair of expressions

```

*p++ = val;    /* push val onto stack */
val = *--p;    /* pop top of stack into val */

```

The header <string. h> contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

POINTER ARRAYS; POINTERS TO POINTERS

Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program sort.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array.

The sorting process has three steps

- read all the lines of input
- sort them
- print them in order

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output.

The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal line count like -1 if too much input is presented. .

The output routine only has to print the line in the order in which they appear in the array of pointers.

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

#define MAXLINES 5000      /* max #lines to be sorted */

char *lineptr[MAXLINES];  /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;      /* number of input lines read */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```

```

#define MAXLEN 1000    /* max length of any input line */
int getline(char *, int);
char *alloc(int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}

```

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr` is an array of `MAXLINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the *i*-th saved text line.

Since `lineptr` is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and `writelines` can be written instead as

```

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}

```

Initially `*lineptr` points to the first line; each increment advances it to the next line pointer while `nlines` is counted down. With input and output under control, we can proceed to sorting.

```

/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)    /* do nothing if array contains */
        return;          /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}

```

Similarly, the swap routine needs only trivial changes:

```

/* swap:  interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

```

Since any individual element of **v** (alias **lineptr**) is a character pointer, **temp** must be also, so one can be copied to the other.

MULTI-DIMENSIONAL ARRAYS

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: **day_of_year** converts the month and day into the day of the year, and **month_day** converts the day of the year into the month and day.

Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets **m** to 2 and **d** to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```

static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}

```

Recall that the arithmetic value of a logical expression, such as the one for leap, is either zero (false) or one (true), so it can be used as a subscript of the array day tab.

The array day tab has to be external to both day_of_year and month_day, so they can both use it. We made it char to illustrate a legitimate use of char for storing small non-character integers.

day tab is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

day tab[i][j]

rather than

day tab[i,j]

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two dimensional array is initialized by a corresponding sub-list. We started the array day tab with a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 ints. In this particular case, it is a pointer to objects that are arrays of 13 ints. Thus if the array day tab is to be passed to a function f, the declaration of f would be

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets `[]` have higher precedence than `*`. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

INITIALISATION OF POINTER ARRAYS

Consider the problem of writing a function `month_name(n)`, which returns a pointer to a character string containing the name of the *n*-th month. This is an ideal application for an internal static array. `month_name` contains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized.

The syntax is similar to previous initializations

```
/* month_name: return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the *i*-th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

COMMAND LINE ARGUMENTS

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.

We customarily use multiple levels of pointers to manipulate these character strings. The simplest illustration is the program `echo`, which echoes its command line arguments on a single line, separated by blanks. That is, the command

echo hello, world

prints the output

hello, world

By convention, `argv [0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is 1, there are no command-line arguments after the program name. In the example above, `argc` is 3, and `argv [0]`, `argv [1]`, and `argv[2]` are "echo", "hello, ", and "world" respectively. The first optional argument is `argv[1]` and the last is `argv[argc-1]`; additionally, the standard requires that `argv[argc]` be a null pointer.

The first version of `echo` treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variation is based on incrementing `argv`, which is a pointer to pointer to char, while `argc` is counted down. Since `argv` is a pointer to the beginning of the array of argument strings, incrementing it by 1 (`++argv`) makes it point at the original `argv [1]` instead of `argv [0]`. Each successive increment moves it along to the next argument; `*argv` is then the pointer to that argument.

At the same time, `argc` is decremented; when it becomes zero, there are no arguments left to print. Alternatively, we could write the `printf` statement as

```
printf ((argc > 1) ? "%s " : "%s", *++argv);
```

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

This shows that the format argument of `printf` can be an expression too. As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us change the program so the pattern to be matched is specified by the first argument on the command line.

```

#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}

```

POINTERS TO FUNCTIONS

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument -n is given; it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts—a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort.

Lexicographic comparison of two lines is done by `strcmp`, as before; we will also need a routine `numcmp` that compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These functions are declared ahead of `main` and a pointer to the appropriate one is passed to `qsort`. We have skimped on error processing for arguments, so as to concentrate on the main issues.

In the call to `qsort`, `strcmp` and `numcmp` are addresses of functions. Since they are known to be functions, the `&` operator is not necessary, in the same way that it is not needed before an array name.

We have written `qsort` so it can process any data type, not just character strings. As indicated by the function prototype, `qsort` expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type `void *` is used for the pointer arguments. Any pointer can be cast to `void *` and back again without loss of information, so we can call `qsort` by casting arguments to `void *`. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

```

/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right)    /* do nothing if array contains */
        return;          /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}

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

#define MAXLINES 5000    /* max #lines to be sorted */
char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines;          /* number of input lines read */
    int numeric = 0;     /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
              (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}

```


The declarations should be studied with some care. The fourth parameter of `qsort` is

```
int (*comp)(void *, void *)
```

which says that `comp` is a pointer to a function that has two `void *` arguments and returns an `int`.

The use of `comp` in the line

```
if ((*comp)(v[i], v[left]) < 0)
```

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *)    /* WRONG */
```

`comp` is a function returning a pointer to an `int`, which is very different. We have already shown `strcmp`, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value, computed by calling `atof`:

```
#include <stdlib.h>

/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

The `swap` function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to `void *`.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

COMPLICATED DECLARATIONS

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used. The difference between

```
int *f();
and
int (*pf());
```

illustrates the problem: * is a prefix operator and it has lower precedence than (), so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with typedef.

As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right. The first, del, is the more complex. It converts a C declaration into a word description, as in these examples:

```
char **argv
    argv: pointer to pointer to char
int (*daytab)[13]
    daytab: pointer to array[13] of int
int *daytab[13]
    daytab: array[13] of pointer to int
void *comp()
    comp: function returning pointer to void
void (*comp)()
    comp: pointer to function returning void
char ((*x())[5])()
    x: function returning pointer to array[] of
    pointer to function returning char
char ((*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

BASIC STRUCTURES

Structure Definition

Rules for declaring a structure

Array Vs Structure:

Accessing structure elements

Structure Initialization

STRUCTURE DEFINITION

The structure can be declared with the keyword struct following the name and opening brace with data elements of different type then closing brace with semicolon.

The general format of a structure definition is as follows:

```
struct structure _ name
```

```
{
```

```
structure_element 1;
```

```
structure_element 2;
```

```
structure_element 3;
```

```
-----
```

```
-----
```

```
};
```

```
struct structure_name v1,v2...vn;  
v1,v2....vn are structure variable.
```

Example:

```
struct book  
{  
char title[20];  
char author[15];  
int pages;  
float price;  
};  
Struct book b1,b2,b3;
```

Rules for declaring a structure

The template is terminated with a semicolon.

While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.

The tag name such as book _ bank can be used to declare structure variables of its type, later in the program.

ARRAY VS STRUCTURE

ARRAY	STRUCTURE
An array is a collection of related datatypes	Structure can have elements of different types
An array is derived data type	Structure is a user-defined datatype
Any array behaves like a built-in data type	It must be declared and defined
An array can be increased or decreased	A structure element can be added if necessary.

Accessing structure elements

After declaring the structure type, variables and members, the member of the structure can be accessed by using the structure variable along with the dot(.) operator.

```
struct std  
{  
int no;  
char name[10];  
int marks;  
};
```

```
struct std s;  
for accessing the structure members from the above example.  
s.no; s.name; s.marks;  
where s is the structure variable
```

STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```

main()
{
struct
{
int weight;
float height;
}
student={60, 180.75};
.....
.....
}

```

This assigns the value 60 to student. weight and 180.75 to student. height. There is a one-to-one correspondence between the members and their initializing values.

```

struct st _record
{
int weight;
float height;
};

```

```

main()
{
struct st_record student1={60, 180.75};
struct st_record student2={53, 170.60};
.....
.....
}

```

C language does not permit the initialization of individual structure member within the template. The initialization must be done only in the declaration of the actual variables.

```

#include<stdio.h>
#include<conio.h>
struct student
{
int rno;
char name[20];
int m1;
int m2;
int m3;
int total;
float avg;
}
struct std s;
void main()
{
printf("Enter the roll no,name,mark1,mark2,mark3");
scanf("%d%s%d%d%d",&s.rno,&s.name,&s.m1,&s.m2,&s.m3); total=s.m1+s.m2+s.m3;
avg=total/3;
printf("%d\t%s\t%f\t%f\t",s.no,s.name,total,avg);
}

```

```
}
```

OUTPUT

Enter the roll no,name,mark1,mark2,mark3

1 raja 99 95 97

1 raja 291.00 97.00

UNION:

Union is user defined data type used to stored data under unique variable name at single memory location. Union is similar to that of structure. Syntax of union is similar to structure. But the major

Difference between structure and union is 'storage.'

union	structure
union keyword is used to define the union type.	structure keyword is used to define the union type.
Memory is allocated as per largest member.	Memory is allocated for each member.
All field share the same memory allocation.	Each member have the own independent memory.
We can access only one field at a time. union Data { int a; // can't use both a and b at once char b; } Data; union Data x; x.a = 3; // OK x.b = 'c'; // NO! this affects the value of x.a	We can any member any time. struct Data { int a; // can use both a and b simultaneously char b; } Data; struct Data y; y.a = 3; // OK y.b = 'c'; // OK
Altering the value of the member affect the value of the other member.	Altering the value of the member does not affect the value of the other member.
Flexible array is not supported by the union.	Flexible array is supported by the structure.
With the help of union we can check the endianness of the system.	We cannot check the endianness.

Syntax:

union union_name

```
{
```

<data-type> element 1;

```

<data-type> element 2;
<data-type> element 3;
}union_variable;

```

Example:

```

union techno
{
int comp_id;

```

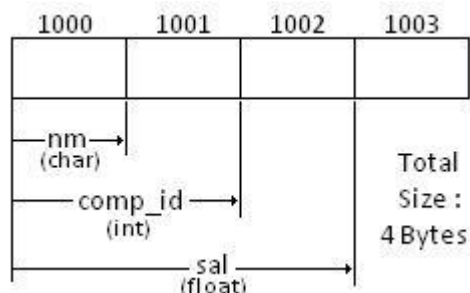
```

char nm;
float sal;
}
tch;

```

In above example, it declares tch variable of type union. The union contains three members as data type of int, char, float. We can use only one of them at a time.

Memory allocation:



Memory allocation for union

To access union members, we can use the following syntax.

```

tch.comp_id
tch.nm
tch.sal

```

Program :

```

#include <stdio.h>
#include <conio.h>
union techno
{
int a;
char b[2];
};
void main()
{
clrscr();
union name c;
c.a=99
printf("\n\t c.a value is :%d\n ",c.a);
printf("\n\t c.b[0] value is :%d\n ",c.b[0]);
printf("\n\t c.b[1] value is :%d\n ",c.b[1]);
getch();

```

```
}
```

Output :

c.a value is :99

c.b[0] value is :0

c.b[1] value is :1

ARRAY OF STRUCTURES

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>
#include<string.h>
struct student
{
char name[30]; char branch[25]; int roll;
};
void main()
{
struct student s[200]; int i;
s[i].roll=i+1;
printf("\nEnter information of students:"); for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll); printf("\nEnter the name:"); scanf("%s",s[i].name);
printf("\nEnter the branch:"); scanf("%s",s[i].branch); printf("\n");
}
printf("\nDisplaying information of students:\n\n"); for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name); printf("\nBranch:"); puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in above example

POINTER OF STRUCTURES

The pointer is a variable which points to the address of another variable of any data type

like int, char, float etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
struct dog
{
char name[10];
char breed[10];
int age;
char color[10];
};
struct dog spike;
// declaring a pointer to a structure of type struct dog
struct dog *ptr_dog
```

This declares a pointer `ptr_dog` that can store the address of the variable of type `struct dog`. We can now assign the address of variable `spike` to `ptr_dog` using `&` operator.

```
ptr_dog = &spike;
```

Now `ptr_dog` points to the structure variable `spike`.

Accessing members using Pointer

There are two ways of accessing members of structure using pointer:

Using indirection (*) operator and dot (.) operator.

Using arrow (->) operator or membership operator.

Using Indirection (*) Operator and Dot (.) Operator

At this point `ptr_dog` points to the structure variable `spike`, so by dereferencing it we will get the contents of the `spike`. This means `spike` and `*ptr_dog` are functionally equivalent. To access a member of structure write `*ptr_dog` followed by a dot (.) operator, followed by the name of the member. For example:

`(*ptr_dog).name` – refers to the name of dog

`(*ptr_dog).breed` – refers to the breed of dog

Parentheses around `*ptr_dog` are necessary because the precedence of dot (.) operator is greater than that of indirection (*) operator.

Using arrow operator (->)

The above method of accessing members of the structure using pointers is slightly confusing and less readable, that's why C provides another way to access members using the arrow (->) operator. To access members using arrow (->) operator write pointer variable followed by -> operator, followed by name of the member.

`ptr_dog->name` - refers to the name of dog

`ptr_dog->breed` - refers to the breed of dog

Here we don't need parentheses, asterisk (*) and dot (.) operator. This method is much more readable and intuitive.

We can also modify the value of members using pointer notation.

```
strcpy(ptr_dog->name, "new_name");
```

Here we know that the name of the array (`ptr_dog->name`) is a constant pointer and points to the 0th element of the array. So we can't assign a new string to it using assignment operator (=), that's why `strcpy()` function is used.

```
--ptr_dog->age;
```

In the above expression precedence of arrow operator (->) is greater than that of prefix decrement operator (--), so first -> operator is applied in the expression then its value is decremented by 1.

The following program demonstrates how we can use a pointer to structure.

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

```
struct dog
{
    char name[10];
    char breed[10];
    int age;
    char color[10];
}
```



```

};

int main()
{
    struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
    struct dog *ptr_dog;
    ptr_dog = &my_dog;

    printf("Dog's name: %s\n", ptr_dog->name);
    printf("Dog's breed: %s\n", ptr_dog->breed);
    printf("Dog's age: %d\n", ptr_dog->age);
    printf("Dog's color: %s\n", ptr_dog->color);

    // changing the name of dog from tyke to jack
    strcpy(ptr_dog->name, "jack");

    // increasing age of dog by 1 year
    ptr_dog->age++;

    printf("Dog's new name is: %s\n", ptr_dog->name);
    printf("Dog's age is: %d\n", ptr_dog->age);

    // signal to operating system program ran fine
    return 0;
}

```

Expected Output:

```

Dog's name: tyke
Dog's breed: Bulldog
Dog's age: 5
Dog's color: white

```

After changes

```

Dog's new name is: jack
Dog's age is: 6

```

SELF REFERENTIAL STRUCTURES

Structures pointing to the same type of structures are self-referential in nature.

Example:

```

struct node {
    int data1;
    char data2;
    struct node* link;
};

```

```

int main()
{
    struct node ob;

```

```
    return 0;
}
```

TYPEDEF in C

‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

The C programming language provides a keyword called typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers –

```
typedef unsigned char BYTE;
```

After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example..

```
BYTE b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows –

```
typedef unsigned char byte;
```

You can use typedef to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows –

```
#include <stdio.h>
#include <string.h>
typedef struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;

int main( ) {

    Book book;

    strcpy( book.title, "C Programming");
    strcpy( book.author, "Nuha Ali");
    strcpy( book.subject, "C Programming Tutorial");
    book.book_id = 6495407;

    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);

    return 0;
}
```

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

Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences –

typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, q., you can define 1 as ONE etc.

typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor.

The following example shows how to use #define in a program –

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

```
#define TRUE 1
#define FALSE 0
int main() {
    printf("Value of TRUE : %d\n", TRUE);
    printf("Value of FALSE : %d\n", FALSE);

    return 0;
}
```

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

Value of TRUE : 1

Value of FALSE : 0

TABLE LOOK UP

Table-lookup package, to illustrate more aspects of structures. This code is typical of what might be found in the symbol table management routines of a macro processor or a compiler. For example, consider the #define statement. When a line like

```
#define IN 1
```

is encountered, the name IN and the replacement text 1 are stored in a table. Later, when the name IN appears in a statement like

```
state = IN;
```

it must be replaced by 1.

There are two routines that manipulate the names and replacement texts. install(s,t) records the name sand the replacement text t in a table; s and t are just character strings. lookup(s) searches for s in the table, and returns a pointer to the place where it was found, or NULL if it wasn't there.

The algorithm is a hash-search - the incoming name is converted into a small non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is NULL if no names have hashed to that value.

A block in the list is a structure containing pointers to the name, the replacement text, and the next block in the list. A null next-pointer marks the end of the list.

```
struct nlist {    /* table entry: */
    struct nlist *next; /* next entry in chain */
    char *name;        /* defined name */
}
```

```
char *defn;      /* replacement text */
};
```

BIT FIELDS

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

For example, consider the following declaration of date without use of bit fields.

```
#include <stdio.h>

// A simple representation of date
struct date
{
    unsigned int d;
    unsigned int m;
    unsigned int y;
};

int main()
{
    printf("Size of date is %d bytes\n", sizeof(struct date));
    struct date dt = {31, 12, 2014};
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Output:

Size of date is 12 bytes

Date is 31/12/2014

The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>

// A space optimized representation of date
struct date
{
    // d has value between 1 and 31, so 5 bits
    // are sufficient
    unsigned int d: 5;

    // m has value between 1 and 12, so 4 bits
    // are sufficient
    unsigned int m: 4;

    unsigned int y;
};

int main()
{
    printf("Size of date is %d bytes\n", sizeof(struct date));
```

```
struct date dt = {31, 12, 2014};
printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
return 0;
}
```

Output:

Size of date is 8 bytes

Date is 31/12/2014

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

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

```
// A structure without forced alignment
```

```
struct test1
{
    unsigned int x: 5;
    unsigned int y: 8;
};
```

```
// A structure with forced alignment
```

```
struct test2
{
    unsigned int x: 5;
    unsigned int: 0;
    unsigned int y: 8;
};
```

```
int main()
{
    printf("Size of test1 is %d bytes\n", sizeof(struct test1));
    printf("Size of test2 is %d bytes\n", sizeof(struct test2));
    return 0;
}
```

Output:

Size of test1 is 4 bytes

Size of test2 is 8 bytes

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

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

```
struct test
{
    unsigned int x: 5;
    unsigned int y: 5;
    unsigned int z;
};
```

```
int main()
{
    struct test t;
```

```
// Uncommenting the following line will make
// the program compile and run
printf("Address of t.x is %p", &t.x);

// The below line works fine as z is not a
// bit field member
printf("Address of t.z is %p", &t.z);
return 0;
}
```

Output:

error: attempt to take address of bit-field structure member 'test::x'

3) It is implementation defined to assign an out-of-range value to a bit field member.

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

```
struct test
```

```
{
    unsigned int x: 2;
    unsigned int y: 2;
    unsigned int z: 2;
};
```

```
int main()
```

```
{
    struct test t;
    t.x = 5;
    printf("%d", t.x);
    return 0;
}
```

Output:

Implementation-Dependent

4) In C, we can have static members in a structure/class, but bit fields cannot be static.

// The below C program compiles and runs fine

```
struct test1 {
    static unsigned int x;
};
int main() { }
```

// cannot be static

```
struct test1 {
    static unsigned int x: 5;
};
int main() { }
```

// error: static member 'x' cannot be a bit-field

5) Array of bit fields is not allowed. For example, the below program fails in compilation.

```
struct test
{
    unsigned int x[10]: 5;
};
```

```
int main()
{
```

```
}
```

Output:

error: bit-field 'x' has invalid type

File Access

File Operation opening a file:

Before performing any type of operation, a file must be opened and for this fopen() function is used.

syntax:

file-pointer=fopen("FILE NAME ", "Mode of open"); example:

FILE *fp=fopen("ar.c", "r");

If fopen() unable to open a file then it will return NULL to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the address of a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

FILE* var;

Modes of open

The file can be open in three different ways as

Read mode 'r'/rt Write mode 'w'/wt Appended Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file Syntax:

character_variable=getc(file_ptr);

Writing a character into a file

putc() is used to write a character into a file

puts(character-var, file-ptr);

CLOSING A FILE

fclose() function close a file. fclose(file_ptr);

fcloseall () is used to close all the opened file at a time

File Operation

The following file operation carried out the file (1)creation of a new file

(3)writing a file (4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using a new type called **file pointer**.

Operation with fopen()

File pointer=fopen("FILE NAME", "mode of open");

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Reading and writing a characters from/to a file fgetc() is used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

```
/*Program to copy a file to another*/ #include<stdio.h>
void main()
{
FILE *fs,*fd; char ch;
If(fs=fopen("scr.txt","r")==0)
{
printf("sorry....The source file cannot be opened"); return;
}
If(fd=fopen("dest.txt","w")==0)
{
printf("Sorry.....The destination file cannot be opened"); fclose(fs);
return;
}
while(ch=fgets(fs)!=EOF) fputc(ch,fd);
fcloseall();
}
```

Reading and writing a string from/to a file **getw()** is used for reading a string from the file

Syntax:

gets(file pointer);

putw() is used to writing a character to a file

Syntax:

fputs(integer,file_pointer);

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
FILE *fp;
/*place the word in a file*/ fp=fopen("dgt.txt","wb"); If(fp==NULL)
{
printf("Error opening file"); exit(1);
}
word=94; putw(word,fp); If(ferror(fp))
printf("Error writing to file\n"); else
printf("Successful write\n"); fclose(fp);
/*reopen the file*/ fp=fopen("dgt.txt","rb"); If(fp==NULL)
{
printf("Error opening file"); exit(1);
}
/*extract the word*/ word=getw(fp);
If(ferror(fp))
```



```
printf("Error reading file\n");
else
printf("Successful read:word=%d\n",word);

/*clean up*/

fclose(fp)
}
```

Reading and writing a string from/to a file **fgets()** is used for reading a string from the file **Syntax:**

fgets(string, length, file pointer);

fputs() is used to writing a character to a file

Syntax:

fputs(string,file_pointer);

```
#include<string.h>
#include<stdio.h>
void main(void)
{
FILE*stream;
char string[]="This is a test"; char msg[20];
/*open a file for update*/ stream=fopen("DUMMY.FIL","w+");

/*write a string into the file*/ fwrite(string,strlen(string),1,stream);
/*seek to the start of the file*/ fseek(stream,0,SEEK_SET);
/*read a string from the file*/ fgets(msg,strlen(string)+1,stream);
/*display the string*/ printf("%s",msg); fclose(stream);
}
```

ERROR HANDLING

C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main () {
    FILE * pf;
    int errnum;
    pf = fopen ("unexist.txt", "rb");
    if (pf == NULL)
    {
        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
    }
    else
    {
        fclose (pf);
    }

    return 0;
}
```

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

Value of errno: 2

Error printed by perror: No such file or directory

Error opening file: No such file or directory

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

The code below fixes this by checking if the divisor is zero before dividing –

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int dividend = 20;
    int divisor = 0;
    int quotient;
    if( divisor == 0){
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(-1);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );

    exit(0);
}
```

```
}
```

When the above code is compiled and executed, it produces the following result –
Division by zero! Exiting...

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows –

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int dividend = 20;
    int divisor = 5;
    int quotient;
    if( divisor == 0) {
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(EXIT_FAILURE);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
    exit(EXIT_SUCCESS);
}
```

When the above code is compiled and executed, it produces the following result –
Value of quotient : 4

LINE I/O:

The standard library provides an input and output routine fgets that is similar to the getline function"

char *fgets(char *line, int maxline, FILE *fp)
fgets reads the next input line (including the newline) from file fp into the character array line; at most maxline-1 characters will be read. The resulting line is terminated with '\0'. Normally fgets returns line; on end of file or error it returns NULL. (Our getline returns the line length, which is a more useful value; zero means end of file.)

For output, the function fputs writes a string (which need not contain a newline) to a file:

```
int fputs(char *line, FILE *fp)
```

It returns EOF if an error occurs, and non-negative otherwise.

The library functions gets and puts are similar to fgets and fputs, but operate on stdin and stdout. Confusingly, gets deletes the terminating '\n', and puts adds it.

To show that there is nothing special about functions like fgets and fputs, here they are, copied from the standard library on our system:

```
/* fgets: get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;
    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
}
```

```

    return (c == EOF && cs == s) ? NULL : s;
}

```

```

/* fputs: put string s on file iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}

```

For no obvious reason, the standard specifies different return values for ferror and fputs. It is easy to implement our getline from fgets:

```

/* getline: read a line, return length */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}

```

MISCELLANEOUS FUNCTION IN C

Descriptions and example programs for C environment functions such as getenv(), setenv(), putenv() and other functions perror(), random() and delay() are given below

Miscellaneous functions	Description
getenv()	This function gets the current value of the environment variable
setenv()	This function sets the value for environment variable
putenv()	This function modifies the value for environment variable
perror()	Displays most recent error that happened during library function call
rand()	Returns random integer number range from 0 to at least 32767
delay()	Suspends the execution of the program for particular time

EXAMPLE PROGRAM FOR GETENV() FUNCTION IN C:

This function gets the current value of the environment variable.

Let us assume that environment variable DIR is assigned to “/usr/bin/test/”. Below program will show you how to get this value using getenv() function.

C

```

#include <stdio.h>
#include <stdlib.h>
int main()

```

```

{
printf("Directory = %s\n", getenv("DIR"));
return 0;
}

```

COMPILE & RUN

OUTPUT:

```
/usr/bin/test/
```

EXAMPLE PROGRAM FOR SETENV() FUNCTION IN C:

This function sets the value for environment variable.

Let us assume that environment variable "FILE" is to be assigned "/usr/bin/example.c". Below program will show you how to set this value using setenv() function.

C

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
setenv("FILE", "/usr/bin/example.c", 50);
printf("File = %s\n", getenv("FILE"));
return 0;
}

```

COMPILE & RUN

OUTPUT:

```
File = /usr/bin/example.c
```

EXAMPLE PROGRAM FOR PUTENV() FUNCTION IN C:

This function modifies the value of environment variable.

Below example program shows that how to modify an existing environment variable value.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
setenv("DIR", "/usr/bin/example/", 50);
printf("Directory name before modifying = " \
"%s\n", getenv("DIR"));

putenv("DIR=/usr/home/");
printf("Directory name after modifying = " \
"%s\n", getenv("DIR"));
return 0;
}

```

OUTPUT:

```

Directory name before modifying = /usr/bin/example/
Directory name after modifying = /usr/home/

```