



An ISO 9001:2008 Certified CompanyTM

n | u | m | e | r | i | c

infosystem private limited

C **PROGRAMMING** for the absolute **beginner**



PROGRAMMING

"FIRST SOLVE THE PROBLEM.
THEN, WRITE THE CODE "

India's Leading IT Training Provider..16 Years of Academic Excellence

6

Weeks / Months

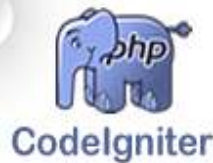
**Project Based
Industrial Training**



An ISO 9001:2008 Certified CompanyTM

numeric
infosystem private limited
development | consultancy | training

Training With Real Time Project



Authorized
Training
&
Examination
Centre



**Preparation Program For
AMCAT/e-Litmus/CoCubes/CRT**

**Campus Drive For Every
Registered Students**

Address.13,Gulabchand ki Bagichi Behind Jhawar Estate Thatipur Chouraha ,Gwalior
Madhya Pradesh 474011

Contact: Sandeep Sappal+91 9301123085, Nivрати Jain:+91 8962533532

CRT



► Dedicates Sessions On

- ✓ Logical Reasoning
- ✓ Verbal Ability
- ✓ Soft Skills
- ✓ Technical Skills

► Comprehensive

- ✓ Verbal & Soft Skills
- ✓ Sessions

► Assess Yourself

- ✓ First Time @ Gwalior
- ✓ Dedicated CRT Lab
- ✓ For Topic & Company Wise Assessments

► Authorised Placement Partner:



Address: 13, Gulabchand ki Bagichi, Behind, Jhawar Estate Thatipur Chouraha Gwalior Madhya Pradesh 474011
Contact: +91 9301123085, +91 8962533532, 0751-4077760
Website: www.numericinfosystem.in

Syllabus - Campus Recruitment Training (CRT)

Description of Training on Quantitative Aptitude and Reasoning

- Number System
- Logarithms
- Average
- Problem on Ages
- Percentages
- Profit & Loss
- Ratio & Proportion
- Time & Work
- Pipes & Cisterns
- Time, Speed & Distance
- Allegations and Mixture
- Simple Interest & Compound Interest
- Permutation & Combination
- Probability
- Geometry

LOGICAL REASONING

- Analogy /Series Completion
- Coding – Decoding
- Blood Relations
- Puzzle test
- Sitting Arrangement
- Direction Sense Test
- Logical Venn Diagrams
- Number, Ranking & Time Sequence Test
- Data Sufficiency
- Crypt arithmetic Questions

VERBAL ABILITY

- Reading Comprehension
- Jumbled Paragraph Questions
- Vocabulary Based Questions
- Fill in Blanks
- Miscellaneous Questions
- Error identification
- Error correction
- Antonyms
- Synonyms / Ordering sentences

**10 Campus Drive For
Registered Students**

TECHNICAL INTERVIEW PREPARATION

Topics to be covered in C

- Basic Concepts of C Language.
- Basic Programming Skills.
- Arrays, Pointers & Structures.

Topics to be covered in C++:

- Introduction to C++
- Object-Oriented Programming Concepts
- The Basics of C++
- Pointers and Arrays
- Function and Operator Overloading
- Reusing classes
- Virtual functions and Polymorphism

Topics to be covered in Datastructures:

- Introduction to datastructures.
- Stacks/Queue
- Linked list
- Tree
- Sorting Algorithms.

Topics to be covered in Databases:

- Introduction to Database.
- Introduction to Normalization.
- Introduction to Data Definition Language/DML.
- Introduction to SQL/SQL Functions.
- Introduction to Set Operators, Groups, Reports.

SOFT SKILLS

- Training need analysis (TNA)
- Behavioural training
- English language & Communication training
- Group Discussion
- Resume \ Curriculum vitae
- Interview Skills
- Email Etiquettes
- Business Etiquette and Customer Handling
- Etiquette of dressing
- Mock Group Discussions
- Mock Interviews
- Mock Recruitment Drive

Course Fee: ₹ 9999/-

Special Features:-

1. Comprehensive Study Material/ Practice Sheets/ Test paper.
2. Online Mock test will be conduct after the completion of every topic.
3. 8 Diagnostic Career Test and 2 Pre-Assessment Test
4. Get directly interview calls from companies only after a good score in Pre-Assess.

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091

Algorithm

The term algorithm may be formally defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired results will be obtained. The latter condition actually states that an algorithm must terminate and should not repeat one or more instructions infinitely. In other words, algorithm represents the logic of the processing to be performed. However, in order to qualify as an algorithm, a sequence of instructions must possess the following characteristics:

1. Each and every instruction should be precise and unambiguous.
2. Each instruction should be such that it can be performed in a finite time.
3. After performing the instructions, that is after the algorithm terminates, the desired results must be obtained.

Algorithm:

Input: two numbers x and y

Output: the average of x and y

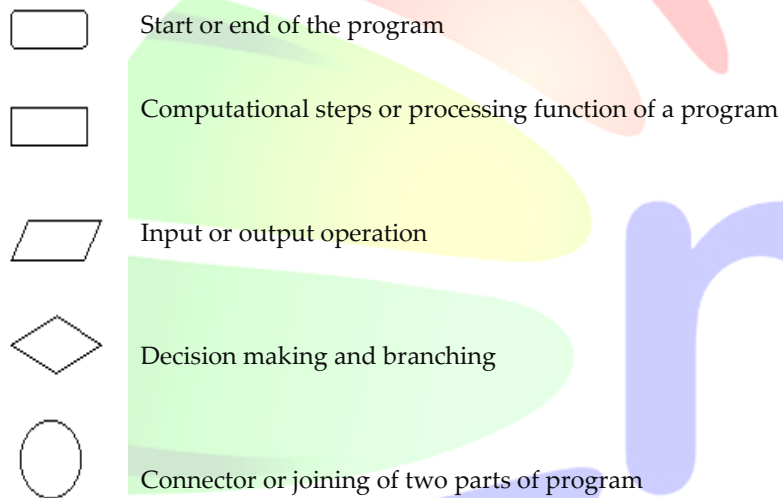
Steps:

1. input x
2. input y
3. $sum = x + y$
4. $average = sum / 2$
5. output average

Introduction To Flowcharts

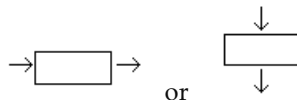
A flowchart is a graphical representation of an algorithm. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

Flowcharts are usually drawn using some standard symbols; however,

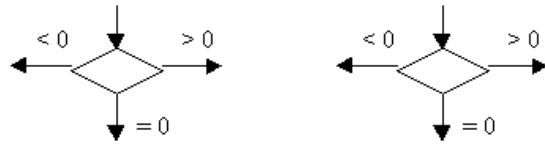


The following are some guidelines in flowcharting:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
- The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should come out from a process symbol.



- e. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.



- f. Only one flow line is used in conjunction with terminal symbol.



- h. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.
- i. Ensure that the flowchart has a logical start and finish.
- j. It is useful to test the validity of the flowchart by passing through it with a simple test data.

Example of a flowchart:

Problem 1: Write an algorithm and draw the flowchart for finding the average of two numbers

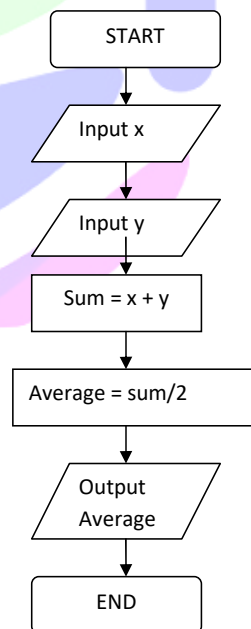
Algorithm:

Input: two numbers x and y

Output: the average of x and y

Steps:

6. input x
7. input y
8. $sum = x + y$
9. $average = sum / 2$
10. output average



Advantages Of Flow Charting

Easy To Understand:-

Since pictures convey the idea more clearly to words hence a solution depicted through flowcharts is more easy to understand as compared to algorithms.

Conciseness:-

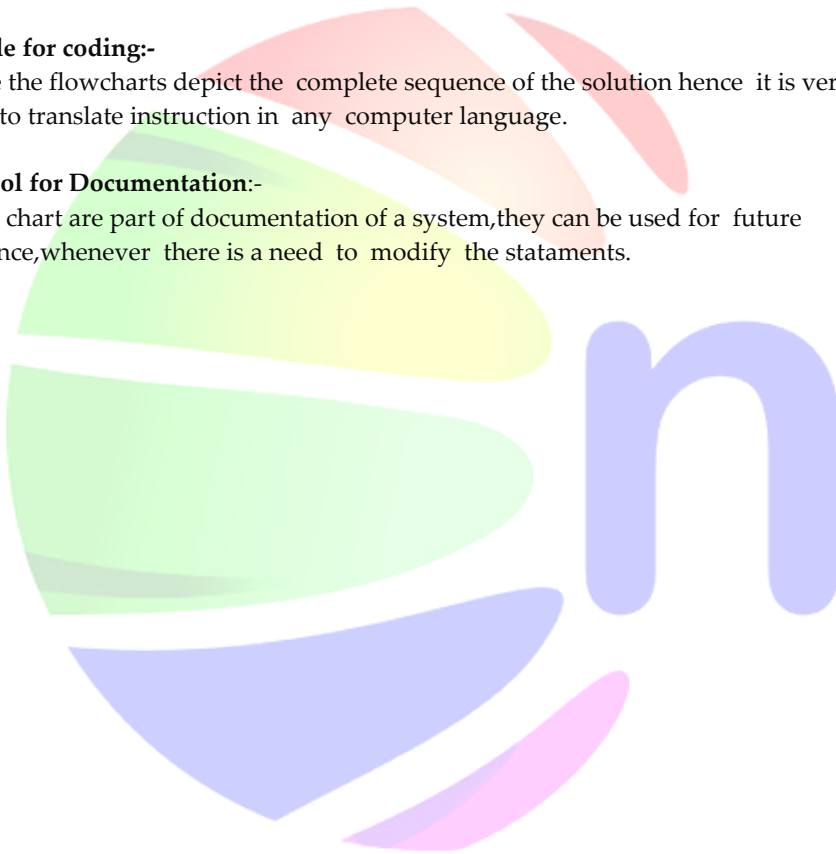
Multiple words can be represented through a single figure hence flowcharts are concise tools for representing the solution.

Guide for coding:-

Since the flowcharts depict the complete sequence of the solution hence it is very easy to translate instruction in any computer language.

A Tool for Documentation:-

Flow chart are part of documentation of a system,they can be used for future refrence,whenever there is a need to modify the stataments.



Program development life cycle:

❖ Problem analysis or understand the problem:

Before we think of a solution procedure to the problem, we must fully understand the nature of the problem and what we want the program to do. So it is necessary to read the problem carefully and understand all the inputs and outputs. We must carefully decide the following at this stage:-

- ✓ -what kind of data will go in,
- ✓ -what kind of outputs are needed, and
- ✓ -what are the constraints and conditions under which the program has to operate.

❖ Plan the solution

In this stage one has to plan various steps that will be carried out for providing the solution. This is basically development of logic for providing solution. Ex: Flowchart, Algorithm.

❖ Program Coding

The algorithm developed in the previous section must be translated to a set of instructions that a computer can understand. The major emphasis in coding should be simplicity and clarity. A program written by one may have been written by other users also. Therefore, it should be readable and simple to understand. Complex Logic and tricky Coding should be avoided.

❖ Program testing and debugging.

Testing and debugging refer to the tasks of detecting and removing errors in a program so that the program produces the desired results on all occasions. It is therefore necessary to make efforts to detect, isolate and correct any errors that are likely to be present in the program.

❖ Types of error.

Errors can be classified under four types, namely, syntax errors, run-time errors, logical errors, and latent errors.

- ✓ Syntax Error: Any violation of rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program.
- ✓ Run Time Errors: Errors such as mismatch of data types or giving out of range array element go undetected by the compiler. Program with this mistake runs but produces erroneous results and therefore named run time errors.

- ✓ Logical Error: These error related to logic of the program execution. Logical errors do not show up compiler-generated error messages, they cause incorrect results. These error occurs due to the poor understanding of the problem.

- ✓ Latent Error: It is hidden error that shows up only when particular set of data is used.

EX:

$\text{ratio} = (x+y)/(p-q);$

An error occurs only when p and q is equal.

Ex. Divide Overflow.

❖ **Program testing:**

Testing Process May include the following two stages:

- 1) *Human testing* .
- 2) *Computer Based testing*.

- ✓ **Human testing** is an effective error detection process and is done before the computer based testing. human testing include code inspection by the programmer, code inspection by the test group and review by the peer group.
- ✓ **Computer Based** testing involves compiler testing and runtime testing. Compiler testing detects all the undiscovered syntax errors. Run time error produce all the run time error messages such as Null pointer Assignment, Divide overflow and Stack Overflow.

When program is free from such error then program is error free and it will tested on sample data. After obtaining the result from sample data result are calculated manually. When manual result and computerised result match the program is said to tested and are ready for implementation.

❖ **Program debugging:**

Debugging is the process of isolation and correcting the errors. One simple method of debugging is to place print statements throughout the program to display the values of variable. it displays the dynamics of program and allows us to examine and compare the information at various points . Once the location of an errors is identified and the errors corrected and debugging statement may be removed.

❖ **Documentation**

A correct program is of little use unless it is supported by full documentation and it is the duty of a Programmer to ensure that this is maintained and supplied to any user. Documentation means writing the programme details on paper.

Documentation of program is designed to fulfill two function:

to enable a user to operate the program correctly and to enable another person to understand the program so that if necessary it may be modified or corrected by someone other than the programmer who first wrote it.

❖ **Implementation**

Implementation basically means taking the software/programs developed to the particular site where it is required, loading them into the computer and initiating the execution of the program.

Program Efficiency

Two critical resources of a computer system are execution time and memory. The efficiency of a program is measured in terms of these two resources. Efficiency can be improved with good design and coding practices.

❖ **Execution Time:**

The execution time is directly tied to the efficiency of the algorithm selected; however, certain coding techniques can considerably improve the execution efficiency. The following are some of the techniques which could be applied while coding the program.

- ✓ 1. Select the fastest algorithm possible.
- ✓ 2. Simplify arithmetic and logical expressions.
- ✓ 3. Use fast arithmetic operation, whenever possible.
- ✓ 4. Carefully evaluate loops to avoid any unnecessary calculation within the loop.
- ✓ 5. If possible, avoid the use of multi-dimensional arrays.
- ✓ 6. Use pointers for handling arrays and strings.

However, remember the following while attempting to improve efficiency.

- ✓ Analyse the algorithm and various parts of the program before attempting any efficiency changes.
- ✓ Make it work before making it faster.
- ✓ Keep it right while trying to make it faster.
- ✓ Do not sacrifice clarity for efficiency.

❖ **Memory Requirement**

Memory restriction in the microcomputer environment is a real concern to the programmer. It is therefore desirable to take all necessary steps to compress memory requirements.

- ✓ 1. Keep the program simple. This is key to memory efficiency.
- ✓ 2. Use an algorithm that is simple and requires less steps.
- ✓ 3. Declare arrays and strings with correct sizes.
- ✓ 4. When possible, limit the use of multi-dimensional arrays.
- ✓ 5. Try to evaluate and incorporate memory compression features.

- ✓ available with the language.

Structured Programming

In some conventional Languages there are some statements which causes lot of difficulty in program maintenance. Languages like BASIC, FORTRAN and COBOL uses GOTO statement control which moves the program flow in desired fashion. GOTO statement makes a program more unstructured because the flow of program moves forwards and backwards in a code this makes program so much confuse to understand the coding. Structured programming encourages the use of programming technique in which the use of goto statement is reduce.

- ✓ -The basic idea behind structured programming is that program can be broken down into more or less independent groups of statement.
- ✓ -There is no size restriction on how many statements can be in a group.
- ✓ -The main approach in structured programming is that it execute a group of instruction. it doesn't execute a single statement like
- ✓ GOTO Statements
- ✓ -Using Structured programming approach each group can be tested separately, so this approach allow the user to easily find out errors and debug the programs.
- ✓ -The structure of this approaches divided into three types:
- ✓ sequence, decision and loop

In sequence structure there must be a starting and ending point. After starting the sequence the program statements are executing one after another until all the statements in the sequence have been executed.

The decision Structure allows to define certain conditions in a group. The final structure is loop. All loops must have a condition where the execution get terminate, like While loop, do.. while loop, for loop etc.

❖ Top Down Approach

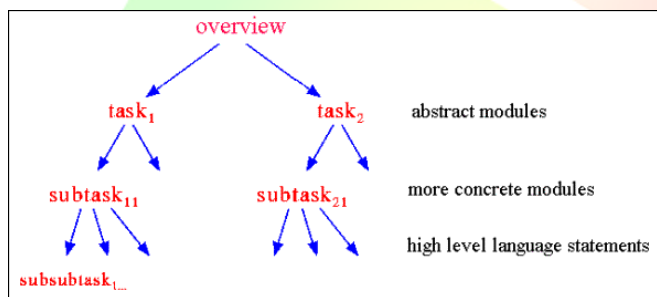
In this approach large programs are divide into small programmes, and these small programs known as module. In general, its a good approach to decomposing the solution of program in hirarchical structure. This task of the top-down design is to cut the whole problem into number of independent constitute task(module), and then cut the task into smaller subtasks, and so on.

In this approach, you begin writing the main modules. then the modules at the next level are written. this procedure is continued until all the modules are written. Meaning that the Top level functions are designed first, assuming certain

task to its lower level functions. The actual details of lower level functions are not considered until that level is reached. Thus the design of functions proceeds from top to bottom.

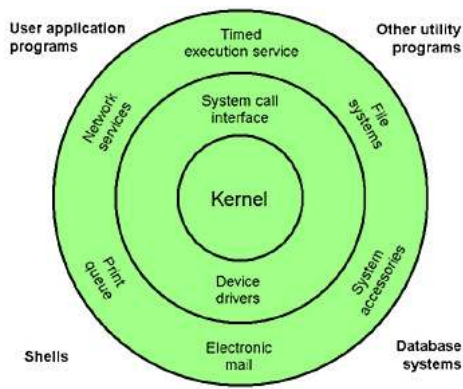
Advantage.

- ✓ a) The most important modules are written and tested first.
- ✓ b) It is easier for the user or the customer to see the progress being made in the project.
- ✓ c) Testing and debugging are easier and more efficient.
- ✓ d) The use of computer resources is more evenly distributed over the entire project.
- ✓ e) The implementation is normally smoother and shorter.
- ✓ f) It is easier to detect and correct time delays and cost overruns.



❖ Bottom Up Approach

Bottom up approach is an alternative to the top down approach. As the name implies the bottom level modules are written first. To test these modules, Dummy modules are written first. After the lower level modules are written, tested and debugged, the next lowest modules are written, tested and debugged. This process continues until all the modules have been completed. This approach is exactly opposite of what is done in the top down approach. This coding is useful because it encourages the reuse of a lot of code that may be available for small functions.



Design of Operating System is the best Example of bottom to top approach

Introduction To C Programming

Powerful features, simple syntax, and portability make C a preferred language among programmers for business and industrial applications. Portability means that C programs written for a computer with a particular kind of processor, say Intel, can be executed on computers with different processors such as Motorola, Sun Sparc, or IBM with little or no modification.

C language is widely used in the development of operating systems. An Operating System(OS) is a software(collection of programs) that controls the various functions of a computer. Also it makes other programs on your computer work. For example, you cannot work with a word processor program, such as Microsoft Word, if there is no operating system installed on your computer. Windows, Unix, Linux, Solaris, and MacOS are some of the popular operating systems.

❖ **Origin of C**

Dennis M. Ritchie, a systems engineer at Bell Laboratories, New Jersey developed C in the early 1970's. Although designed for the Unix operating system, it soon proved itself a powerful, general purpose programming language

❖ **History of C**

The *milestones* in C's development as a language are listed below:

- UNIX developed c. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language ``B" a second attempt. c. 1970.
- A totally new language ``C" a successor to ``B". c. 1971
- By 1973 UNIX OS almost totally written in ``C".

❖ **Where is C useful?**

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091

C is ability to communicate directly with hardware makes it a powerful choice for system programmers. In fact, popular operating systems such as Unix and Linux are written entirely in C. Additionally, even compilers and interpreters for other languages such as FORTRAN, Pascal, and BASIC are written in C. However, C scope is not just limited to developing system programs. It is also used to develop any kind of application, including complex business ones. The following is a partial list of areas where C language is used:

Embedded Systems

Game Programming

Systems Programming

Artificial Intelligence

Industrial Automation

Computer Graphics

Image Processing

Space Research

❖ What kind of language is C?

C is a structured programming language, which means that it allows you to develop programs using well-defined *control structures* (you will learn about *control structures* in the articles to come), and provides *modularity* (breaking the task into multiple sub tasks that are simple enough to understand and to reuse).

C is often called a **middle-level language** because it combines the best elements of low-level or machine language with high-level languages.

❖ Why you should learn C?

You should learn C because:

- *C is simple.*
- *There are only 32 keywords so C is very easy to master. Keywords are words that have special meaning in C language.*
- *C programs run faster than programs written in most other languages.*
- *C enables easy communication with computer hardware making it easy to write system programs such as compilers and interpreters.*
 - *Small size*
 - *Extensive use of function calls*
 - *Loose typing -- unlike PASCAL*
 - *Structured language*
 - *Low level (BitWise) programming readily available*
 - *Pointer implementation - extensive use of pointers for memory, array, structures and functions.*

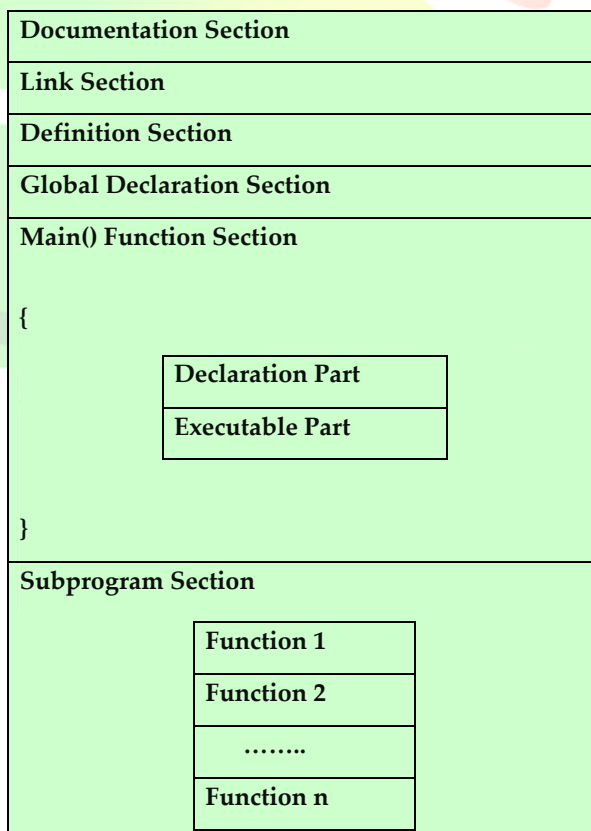
C has now become a widely used professional language for various reasons.

- *It has high-level constructs.*
- *It can handle low-level activities.*
- *It produces efficient programs.*

It can be compiled on a variety of computers

C Program Structure

A C program basically has the following form:



1. **Documentation Section:** This section consists of a set of comments lines giving the name of the program, the author and other details which the programmer would like to use later.
2. **Link Section:** This section provides instruction to the compiler to link functions from the system library.
3. **Definition section:** This section defines all symbolic constants.
4. **Global Section:** Here those variables are declared which are used in more than one function.
5. **main() Function Section:** Every program must have one main () function section. It contains declaration part and executable part.
6. **Subprogram Section:** It contains all the user-defined functions that are called in the main function.

The Simplest C Program: What's Happening?

```
#include <stdio.h>
int main()
{
    printf("This is output from my first program!\n");
    return 0;
}
```

This C program starts with **#include <stdio.h>**. This line **includes** the "standard I/O library" into your program. The standard I/O library lets you read input from the keyboard (called "standard in"), write output to the screen (called "standard out"), process text files stored on the disk, and so on. It is an extremely useful library. C has a large number of standard libraries like stdio, including string, time and math libraries.

The line **int main()** declares the main function. Every C program must have a function named **main** somewhere in the code.. At run time, program execution starts at the first line of the main function.

- In C, the { and } symbols mark the beginning and end of a block of code. In this case, the block of code making up the main function contains two lines.
- The **printf** statement in C allows you to send output to standard out (for us, the screen). The portion in quotes is called the **format string** and describes how the data is to be formatted when printed. The format string can contain string literals such as "This is output from my first program!," symbols for carriage returns (\n), and operators as placeholders for variables.
- The **return 0;** line causes the function to return an error code of 0 (no error) to the shell that started execution.

Identifiers

Identifiers, as the name suggests, are used to identify or refer to the names of variables, symbolic constants, functions and arrays. There are certain rules regarding identifier names, they are :

- Identifier names must be a sequence of letters and digits and must begin with a letter.
- The underscore character (_) is also permitted in identifiers. It is usually as a link between two words in long identifiers.
- Names should not be the same as a keyword.
- C is a case sensitive (i.e. upper and lower case letters are treated differently). Thus the names price, Price and PRICE denote different identifier.

Valid examples are :

City	Age	basic_pay	result
date_of_birth	Mark	num1	num2

The following identifiers are invalid :

<i>Invalid Identifiers</i>	<i>Reason For Invalidity</i>
Basic Pay	Blank space is not allowed
1price	Must start with a letter
\$amount	Special characters is not allowed
break	break is a keyword

Constants

Constants refer to fixed values that do not change during the execution of a program. Figure below shows the various types of constants available in C :

❖ Integer Constants

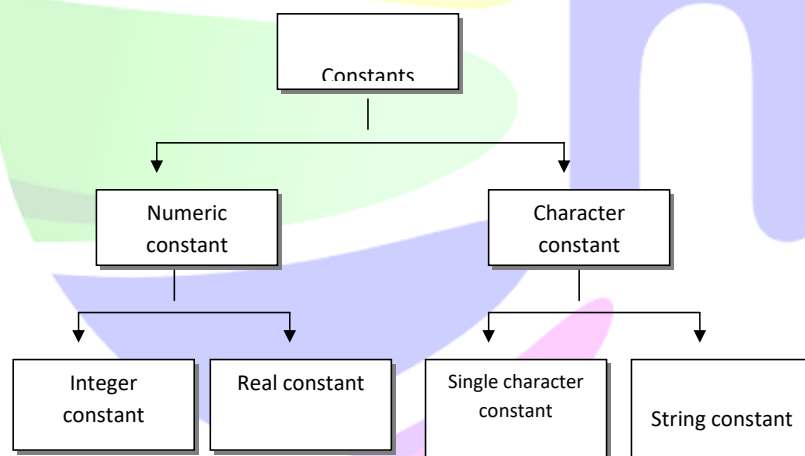
Integer constants are whole numbers without any fractional part.

Rules for constructing integer constants are :

- ✓ *An integer constant must have at least one digit and must not have a decimal point.*
- ✓ *An integer constant may be positive or negative. If no sign is preceded an integer constant, it is assumed to be positive.*
- ✓ *Embedded spaces, commas and non-digit characters are not allowed.*

There are three types of integer constants namely :

- ✓ *Decimal*
- ✓ *Octal*
- ✓ *Hexadecimal*



(i) Decimal Integer Constants

Decimal integer constants can consist any combinations of digits from 0 to 9, preceded by an optional + or – sign.

Valid examples are :

0 +47 179 -240 22099

Invalid examples are :

<i>Invalid Decimal Integer Constants</i>	<i>Reason For Invalidity</i>
15 750	Blank space is not allowed
20,000	Comma is not allowed
\$1000	\$ symbol is not allowed
12.55	Decimal point is not allowed

(ii) Octal Integer Constants

Octal integer constants can consist of any combinations of digits from 0 to 7. However, the first digit must be zero (0), in order to identify the constant as an octal number. For instance, decimal integer 8 will be written as 010 as octal integer.

Valid examples are :

0 047 0179 0240

(iii) Hexadecimal Integer Constants

Hexadecimal integer constants can consists of any combination of digits from 0 to 9 and letters from A to F (either uppercase or lowercase). A hexadecimal integer constant must begin with either 0x or 0X. For instance, decimal integer 12 will be written as 0XC as hexadecimal integer.

Valid examples are :

0x 0X2 0x7A 0xbcd

Thus, the number 12 will be written either 12 (as decimal), 014 (as octal) and 0xC (as hexadecimal).

Note : The larger integer value that can be stored is machine dependent.

<i>Machine Type</i>	<i>Largest Integer Value</i>
16 bit machines	32,767
32 bit machines	2,147,483,647

❖ Real Constants

Real constants are also called as floating point constants. Real constants are numbers having fractional parts. These may be written in one of the two forms :

- (i) Fractional form (ii) Exponential form

(i) Fractional Form

The fractional form consists of a series of digits representing the whole part followed by a decimal point and series of digits representing the floating part. The whole part or fractional part can be omitted but not both. The decimal point cannot be omitted.

Rules for constructing real constants expressed in fractional form are :

1. A real constant must have at least one digit and it must have a decimal point.
2. A real constant could be either positive or negative (default sign is positive).
3. Embedded spaces, commas and non-digit characters are not allowed.

Valid examples are :

0.0002 -0.96 179.47 +31.79
+2 -.47 179. .99

Invalid examples are :

<i>Invalid Fractional Form</i>	<i>Reason For Invalidity</i>
12,000.50	Comma is not allowed
31	No decimal point
12.30.45	Two decimal point

\$1000.75	\$ symbol is not allowed
15 750.25	Blank space is not allowed

(ii) Exponential Form

The exponential (or scientific) form of representation of real constants is usually used if the value of the constant is either too small or too large. In exponential form of representation, the real constant is represented in two parts namely, mantissa and exponent. The syntax is :

mantissa e exponent [or]

mantissa E exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter e separating the mantissa and exponent can be written in either lowercase or uppercase.

For example, 3.5 can be written as $0.35 \times 10 = 0.35E1$ where the mantissa part is 0.5 (the part appearing before E) and the exponent part is 1 (the part appearing after E).

Rules for constructing real constants expressed in exponential form are :

1. The mantissa and the exponent should be separated by a letter e (either uppercase or lowercase).
2. The mantissa and exponent part may have a positive or negative sign (default sign is positive).
3. The mantissa and exponent part must have at least one digit.
4. Embedded spaces, commas are not allowed.

Valid examples are :

0.31e2 12e-2 3.1e+5
3.18E4 -1.2E-2

Invalid examples are :

<i>Invalid Exponential Form</i>	<i>Reason For Invalidity</i>
12,000e2	Comma is not allowed

3.1e 2	Blank space is not allowed
3.1E+2.4	Exponent must be an integer

Note : Floating point constants are normally represented as double precision quantities. However, the suffixes f or F may be used to force single precision and l or L to extend double precision further.

❖ Single Character Constants

A single character constant or a character constant consist of a single character enclosed within apostrophes.

Rules for constructing single character constants are :

1. A character constant is a single alphabet, a single digit, a single special symbol or a blank space.
2. The maximum length of a character constant can be one character.

Valid examples are :

'B' 'V' '+' '\$' ''

Invalid examples are :

'abc' '123'

❖ String Constants

A string constant is a sequence of characters enclosed within double quotation marks. Technically, a string is an array of characters. The compiler automatically places the null character (`\0`) at the end of each such string, so program can conveniently find the end.

Rules for constructing string constants are :

1. *The string constant may be letters, numbers, special characters and blank spaces.*
2. *Every string constant ends up with a null character, which is automatically assigned.*

Valid examples are :

"India"

"2002"

"WELCOME"

"*.*"

"B"

"\$120"

Note :

1. A character constant 'B' is not the same as the string constant that contains the single character "B". The former is the single character and the latter a character string consisting of character B and \0 (null).
2. A character constant has an equivalent integer value, whereas single character string constant does not have an equivalent integer value. It occupies two bytes, one for the ASCII code of B and another for the null character with a value 0, which is used to terminate strings.

Variables

A quantity, which may vary during the execution of a program, is called as variable. It is a data name that may be used to store a data value. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.

Rules for constructing a variable are :

- ✓ A variable name is any combination of alphabets, digits or underscores.
- ✓ The first character in the variable name must be an alphabet.
- ✓ No comma or blank spaces are allowed within the variable name.
- ✓ No special characters other than an underscore (_) can be used in a variable name.
- ✓ A variable name cannot be a keyword.
- ✓ Variable names are case sensitive, i.e., uppercase and lowercase letters are treated as distinct and different. Hence, MARK, mark and Mark are three separate names.
- ✓ A variable name should not be of length more than 31 characters.

Valid examples are :

AVERAGE	height	subject1	n1
City	male_sum	fact	n

Invalid examples are :

<i>Invalid Variable Name</i>	<i>Reason For Invalidity</i>
char	char is a keyword
price\$	\$ sign is illegal
group one	Blank space is not allowed
786	First character must be an alphabet

Note :

1. Some compilers permit underscore as the first character.
2. There is no limit on the length of the variable, but some compilers recognize only first eight characters. For example, the variable name dataname1 and dataname2 mean the same thing to the compiler.

Declaration Of Variable

data_type V1, V2, ... Vn ;

Data Types

1. Primary (or fundamental) data types
2. User-defined data types
3. Derived data types
4. Empty data set

<i>Data Type</i>	<i>Size</i>	<i>Range</i>
char	1 byte	-128 to 127
Int	2 bytes	-32,768 to 32,767
float	4 bytes	3.4e-38 to 3.4e+38
double	8 bytes	1.7e-308 to 1.7e+308

❖ Integer Data Type

<i>Type</i>	<i>Length</i>	<i>Range</i>
Unsigned int (or) unsigned short int	16 bits	0 to 65,535
int (or)		

short int (or) signed int (or) signed short int	16 bits	-32,768 to 32,767
Unsigned long (or) Unsigned long int	32 bits	0 to 4,294,967,295
long (or) long int (or) signed long int	32 bits	-2,147,483,648 to 2,147,483,647

❖ Floating point Data Type

<i>Type</i>	<i>Length</i>	<i>Range</i>
float	32 bits	3.4e-38 to 3.4e+38
double	64 bits	1.7e-308 to 1.7e+308
long double	80 bits	3.4e-4932 to 1.1e+4932

❖ Character Data Type

<i>Type</i>	<i>Length</i>	<i>Range</i>
unsigned char	8 bits	0 to 255
char (or) signed char	8 bits	-128 to 127

Operators

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operator
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Comma operator
9. sizeof operator

❖ Arithmetic Operators

<i>Operator</i>	<i>Meaning</i>
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division or remainder after division

❖ Relational Operators

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

❖ Logical Operators

<i>Operator</i>	<i>Meaning</i>
&&	Logical AND
	Logical OR
!	Logical NOT

❖ Assignment Operator

=, +=, -=, *=, /=, %=

❖ Increment and Decrement Operators

++, --

❖ Conditional Operators

exp1 ? exp2 : exp3 ;

❖ Bitwise Operators

<i>Operator</i>	<i>Meaning</i>
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

Evaluation Of Expressions

<i>Algebraic Expression</i>	<i>Equivalent C Expression</i>
$3.14x r^2$	$3.14 * r * r$
$\frac{a}{bc}$	$a / (b * c)$
$5a^2 + 3a + 1$	$5 * a * a + 3 * a + 1$
$s(s-a)(s-b)(s-c)$	$s * (s - a) * (s - b) * (s - c)$

Precedence Of Operators

<i>Operator</i>	<i>Description</i>	<i>Associativity</i>	<i>Rank</i>
() [] -> and .	Function call Array element reference Structure operators	Left to Right	1
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation One's complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to Right	3
+ -	Addition Subtraction	Left to Right	4
<< >>	Left shift Right shift	Left to Right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right	6
= !=	Equality Inequality	Left to Right	7
&	Bitwise AND	Left to Right	8
^	Bitwise XOR	Left to Right	9
	Bitwise OR	Left to Right	10
&&	Logical AND	Left to Right	11

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

	Logical OR	Left to Right	12
?:	Condition expression	Right to left	13
= *= /= %= += -= &= ^= = <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to Right	15

Escape Sequences (Backslash Character Constants)

<i>ASCII Value</i>	<i>Escape Sequences</i>	<i>Meaning</i>
000	\0	Null
007	\a	Audible alter (bell)
008	\b	Backspace
009	\t	Horizontal tab
010	\n	New line
011	\v	Vertical tab
012	\f	Form feed
013	\r	Carriage return
034	\"	Double quote
039	\'	Single quote
063	\?	Question mark
092	\\	Backslash

Type Conversions in Expressions:

C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion.

If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is higher type.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during final assignment.

1. *float to int* cause truncation of the fractional part.
2. *double to float* caused rounding of digits.
3. *long int to int* causes dropping of the excess higher order bits

❖ Casting a Value:

There are instances when we want to force a type conversion in a way that is different from automatic conversion.

Example

```
ratio = female_number/male_number
```

Since `female_number` and `male_number` are declared as integers in the program, the decimal part of the result of division would be lost and `ratio` would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating as shown below:

```
ratio = (float) female_number/male_number
```

The process of such a local conversion is known as casting a value. The general form of cast is:

(type-name) expression

Basic I/O Functions

1. Reading a Character:

➤ **getchar()**

syntax is:

```
variable_name=getchar();
```

Example

```
char name;
```

```
name = getchar();
```

will assign the character 'H' to the variable name when we press the key H on the keyboard.

➤ **getch()**

getch() is also used to read a character from input device. getch() displays the character without accepting the carriage return.

➤ **getche()**

getche() reads a character from standard input device but the character is not displayed on the screen and also carriage return is not accepted.

2. Writing a Character

This function writes a character to the standard output device.

syntax

```
putchar(variable_name);
```

Where variable_name is type char variable containing a character. This statement displays character contained in the variable-name at the terminal.

Example:

```
answer='Y';  
putchar(answer);
```

Will display the character 'Y' on the screen.

Formatted input/output Function:

❖ **printf ()**

The function is defined as follows:

int printf(char *format, arg list ...) --

prints to stdout the list of arguments according specified format string. Returns number of characters printed.

The **format string** has 2 types of object:

- **ordinary characters** -- these are copied to output.
- **conversion specifications** -- denoted by % and listed in Table

<i>Format Specifier</i>	<i>Type of Argument</i>	<i>Input</i>
%c	Character	Reads a single character
%d or %i	Integer	Reads a decimal integer
%e or %E or %f or %g or %G	Floating point	Reads a floating point value
%hd or %hi	Short integer	Reads decimal short integer
%hu	Short integer	Reads decimal unsigned short integer
%ld or %li	Long integer	Reads decimal long integer
%le or %lf or %lg	Double	Reads signed double
%Le or %Lf or %Lg	Long double	Reads signed long double
%lo	Long integer	Reads an octal long integer
%lu	Long integer	Reads decimal unsigned long integer
%lx	Long integer	Reads hexadecimal long integer

%o	Octal integer	Reads an unsigned octal integer
%s	Sequence of characters	Reads a string
%u	Integer	Reads an unsigned decimal integer
%x or %X	Hexadecimal integer	Reads a unsigned hexadecimal integer

❖ Between % and format char we can put:

- (minus sign) -- left justify.

integer number -- field width.

m.d -- m = field width, d = precision of number of digits after decimal point or number of chars from a string.

So: `printf("%-2.3f\n", 17.23478);`

The output on the screen is:

17.235

and:

`printf("VAT=17.5%%\n");`

...outputs:

VAT=17.5%

Example:

`#include <stdio.h>`

`int main()`

`{`

`int a, b, c;`

`a = 5;`

`b = 7;`

`c = a + b;`

`printf("%d + %d = %d\n", a, b, c);`

`return 0;`

`}`

Here is an explanation of the different lines in this program:

- The line `int a, b, c;` declares three integer variables named **a**, **b** and **c**. Integer variables hold whole numbers.
- The next line initializes the variable named **a** to the value 5.
- The next line sets **b** to 7.
- The next line adds **a** and **b** and "assigns" the result to **c**.

The computer adds the value in **a** (5) to the value in **b** (7) to form the result 12, and then places that new value (12) into the variable **c**. The variable **c** is assigned the value 12. For this reason, the = in this line is called "the assignment operator."

- The **printf** statement then prints the line "5 + 7 = 12." The **%d** placeholders in the printf statement act as placeholders for values. There are three %d placeholders, and at the end of the printf line there are the three variable names: **a**, **b** and **c**. C matches up the first %d with a and substitutes 5 there. It matches the second %d with b and substitutes 7. It matches the third %d with c and substitutes 12. Then it prints the completed line to the screen: 5 + 7 = 12. The +, the = and the spacing are a part of the format line and get embedded automatically between the %d operators as specified by the programmer.

❖ **Scanf ()**

This function is defined as follows:

int scanf(char *format, args....) -- reads from stdin and puts input in address of variables specified in args list. Returns number of chars read.

Format control string similar to printf

Note: The ADDRESS of variable or a pointer to one is required by scanf.

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

We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.

```
char string[80];
scanf("%s",string);
```

The simplest application of **scanf** looks like this:

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

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b.

The scanf function uses the same placeholders as printf:

You **MUST** put **&** in front of the variable used in scanf. It is easy to forget the & sign, and when you forget it your program will almost always crash when you run it.

❖ **Library Functions:**

C provides a rich set of built-in functions called as library functions. It helps in making programming easy. These are already created functions.

If we want to use mathematical functions like sin, cos , etc we need to include math.h Similarly we can include other library functions as required:

```
string.h
graphics.h
math.h
stdio.h
conio.h
```



Control Statements

INTRODUCTION

1. Sequential control structure
2. Selective control structure
3. Iterative control structure

❖ Sequential Control Structure

The normal flow of control of all programs is sequential. In sequential structure, a sequence of programs statements are executed one after another in the order in which they are placed. Both selection and repetition statements allows allow the programmer to alter the normal sequential flow of control.

Sequential programming can also be called linear programming. The sequential programs are non-modular in nature. That is, reusability of code is not possible. Thus, they are difficult to maintain and understand. Examples of sequence control structure statements are, the program will have statements that are placed sequentially and there is no decision involved in the process. Also, the program does not require a specific task to be repeated over and over again.

❖ **Selective Control Structure (or) Decision Control Structure**

The selective structure allows the usual sequential order of execution to be modified. It consists of a test for a condition followed by alternative paths that the program can follow. The program selects one of the alternative paths depending upon the result of the test for condition. Examples of selective control structures statements are :

1. Simple if statement
2. if . . . else statement
3. Nested if . . . else statement
4. else if ladder
5. switch . . . case . . . default statement

❖ **Iterative Control Structure (or) Loop Control Structure**

The iterative structure provides the ability to go back and repeat a set of statements. Iterative structure is otherwise referred to as repetitive structure. Examples of iterative control structure statements are :

1. while statement
2. do . . . while statement
3. for statement

if Statements

C allows decisions to be made by evaluating a given expression as true or false. Such an expression involves the relational and logical operators. Depending on the outcome of the decision, program execution proceeds in one direction or another. The C statement that enables these tests to be made is called the if statements.

The if statements may be implemented in different forms depending on the complexity of conditions to be tested. They are :

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

✓ **Simple if Statement**

The simple if statement is used to specify conditional execution of program statement or a group of statements enclosed in braces. The syntax is :

```

if (test condition)
{
    statement-block ;
}
statement-x ;

```

When an if statement is encountered, test condition is evaluated first and if it is true, the statement-block will be executed. If the test condition is false, the statement-block will be skipped and the execution will jump to the statement-x.

When the test condition is true, both the statement-block and the statement-x are executed in sequence. The test condition is always enclosed within a pair of parenthesis. The statement-block may be a single statement or a group of statements.

Table below shows the various expressions that are used as conditions inside an if statement :

Conditional Expression	Meaning	For e.g., Value of a	For e.g., Value of b	Result
a == b	a is equal to b	5	5	True
		5	3	False
a != b	a is not equal to b	5	3	True
		5	5	False
a < b	a is less than b	3	5	True
		5	3	False
a > b	a is greater than b	5	3	True
		3	5	False
a <= b	a is less than or equal to b	3	5	True
		3	3	True
		5	3	False
a >= b	a is greater than or equal to b	5	3	True
		5	5	True
		3	5	False

Two or more conditions may be combined in an if statement using a logical AND operator (&&) or a logical OR operator (||). It can compare any number of variables in a single if statement.

Table below shows the various expressions that are used as conditions inside an if statement :

Conditional	Meaning	For e.g., Value	For e.g.,	For e.g.,	
-------------	---------	-----------------	-----------	-----------	--

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

<i>Expression</i>		<i>of a</i>	<i>Value of b</i>	<i>Value of c</i>	<i>Result</i>
$((a > b) \&\& (b > c))$	a is greater than b AND b is greater than c	30	20	10	True
		30	10	20	False
		10	30	20	False
		10	20	30	False
$((a > b) (b > c))$	a is greater than b OR b is greater than c	30	20	10	True
		30	10	20	True
		10	30	20	True
		10	20	30	False

Note : There is only one statement in the if block, the braces are optional. But if there is more than one statement you must use the braces.

✓ if . . . else Statement

Sometimes, we can execute one group of statements if the condition is true and another group of statements if the condition is false, in such a situation, the if . . . else statement can be used.

The if . . . else statement is an extension of simple if statement. The syntax is :

```

if (test condition)
{
    true-block-statement(s);
}
else
{
    false-block-statement(s);
}
statement-x;

```

If the test condition is true, then the true-block-statement(s) are executed. If the test condition is false, then the false-block-statement(s) are executed. In either case, either true-block-statement(s) or false-block-statement(s) will be executed, not both.

❖ Nested if . . . else Statement

We can write an entire if . . . else construct within either the body of an if statement or the body of an else statement. This is called nesting of ifs. The syntax is :

```

if (test condition 1)

```

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
 Compiled By: Sandeep Sappal
 Mob. : 9301123085, Ph. : 4062091

```

{
    if (test condition 2)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
}
else
{
    statement-3;
}
statement-x;

```

If the test condition 1 is false, the statement-3 will be executed. If the test condition 1 is true, it continues to perform the test condition 2. If the test condition 2 is true, the statement-1 will be executed. If the test condition 2 is false, the statement-2 will be executed and then the control is transferred to the statement-x.

❖ **else if Ladder**

Generally, the case where the statements in the if part of an if . . . else statement is another if statement tends to be confusing and best avoided. However an extremely useful construction occurs when the else part of if statement contains another if . . . else statement. This construction is called as an else if ladder or chain. The syntax is :

```

if (test condition 1)
{
    statement-1;
}
else if (test condition 2)
{
    statement-2;
}
. . . . .
else if (test condition n)
{
    statement-n;
}

```

```

else
{
    default-statement ;
}
statement-x ;

```

Each condition is evaluated in order and if any condition is true, the corresponding statement is executed and the remainder of the chain is terminated and the control is transferred to the statement-x. The final else statement containing the default statement will be executed if none of the previous n conditions are not satisfied.

switch Statement

The control statement which allows us to make a decision from the number of choices is called a switch, or more correctly a switch . . . case . . . default, since these three keywords go together to make up the control statement.

The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with the case is executed. The syntax is :

```

switch (expression)
{
    case value-1 :
        statement-1 ;
        break ;
    case value-2 :
        statement-2 ;
        break ;
    .      .      .
    .      .      .

```

```

        . . .
        default :
            default-statement ;      }
statement-x ;

```

Where the expression is an integer expression or characters. value-1, value-2 are constants or constant expression (valuable to an integral constant) and are known as case labels. Each of these values should be unique with a switch statement. statement-1, statement-2 are statement lists and may contain one or more statements. There is no need to put braces around these blocks. The keyword **case** is followed by an integer or a character constant. Each constant in each case must be different from all others and the case labels end with a colon (:).

When the switch is executed, the value is computed for expression, the list of possible constant expression values determined from all case statements is searched for a match. If a match is found, execution continues after the matching case statement and continues until a break statement is encountered or the end of statement is reached.

The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch. The default is an optional case. If a match is not found and the default statement prefix is found within switch, execution continues at this point. Otherwise, switch is skipped entirely and the control goes to the statement-x.

Note : At the end of every case, there should be a break statement. Otherwise, it will result in causing the program execution to continue into the next case whenever case gets executed.

goto Statement

The goto statement is used to alter the normal sequence of program execution by unconditionally transferring control to some part of the program. The syntax is :

```
goto label ;
```

Where label is an identifier used to label the target statement to which the control would be transferred. Control may be transferred to any other statement within the current function. The target function must be labeled followed by a colon. The syntax is :

```
label : statement ;
```

while Statement

The while statement is a general repetition statement that can be used in a number of programming environments. It is an entry controlled loop statement, since the test condition

is tested before the start of the loop execution. This loops statement is used when the number of passes are not known in advance. The syntax is :

```
while(test condition)
{
    body of the loop ;
}
statement-x ;
```

Where the body of the loop may have one or more statements. The braces are need only if the body contains two or more statements. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body of the loop is executed once again. This process of repeated execution of the body of the loop continues until the test condition becomes false and the control is transferred out of the loop. On exit, the program continues with the statement-x which is immediately after the body of the loop.

Note : The test condition specified in the while loop should eventually become false at one point of the program, otherwise the loop will become an infinite loop.

do . . . while Statement

There is a minor difference between the working of while and do . . . while loops. This difference is the place where the condition is tested. The while test condition before executing any of the statements within the while loop. The do . . . while tests the condition after having executed the statements within the loop. This means that do . . . while would execute its statements at lease once, even if the condition fails for the first time itself. The while, on the other hand will not execute its statement if the condition fails for the first time.

do . . . while is an exit controlled loop statement, since the test condition is performed at the end of the body of the loop and therefore the body of the loop is executed unconditionally for the first time. The syntax is :

```
do
{
    body of the loop ;
} while(test condition) ;
statement-x ;
```

Where the body of the loop may have one or more statements. On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, then the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement-x that appears immediately after the while statement.

for Statement

The for loop is another entry controlled loop that provides a more concise loop control structure. The syntax is :

```
for(exp1 ; exp2 ; exp3)  
{  
    body of the loop ;  
}
```

Before the first iteration, <exp1> is evaluated. This is usually used to initialize variables for the loop that is used to set the loop control variable. The <exp2> is a relational expression that determines when the loop should terminate. After each iteration of the loop, <exp3> is evaluated. This is usually used to increment or decrement the loop counters. The body of the loop is executed repeatedly till the condition in <exp2> is satisfied. The expressions must be separated by a semicolon. All the expressions are optional. If <exp2> is left out, it is assumed to be 1 (i.e. True).

Arrays

Arrays

- ❖ The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- ❖ An array is a list of a finite number *n* of *homogeneous* data elements (i.e., data elements of the same type) such that:
 - a) The elements of the array are referenced respectively by an *index* consisting of *n* consecutive numbers.
 - b) The elements of the array are stored respectively in successive memory locations.

Operations of Array

- ❖ Two basic operations in an array are *storing* and *retrieving (extraction)*

Storing: A value is stored in an element of the array with the statement of the form,

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

Data[i] = X ; Where I is the valid index in the array

And X is the element

Extraction : Refers to getting the value of an element stored in an array.

X = Data [i], Where I is the valid index of the array and X is the element.

Array Representation

- ✓ The number *n* of elements is called the *length* or *size* of the array. If not explicitly stated we will assume that the index starts from 0 and end with n-1.
- ✓ In general, the length (range) or the number of data elements of the array can be obtained from the index by the formula,
$$\text{Length} = \text{UB} - \text{LB} + 1$$
- ✓ Where UB is the largest index, called the Upper Bound, and LB is the smallest index, called Lower Bound, of the array.
- ✓ If LB = 0 and UB = 4 then the length is,
$$\text{Length} = 4 - 0 + 1 = 5$$
- ✓ The elements of an array A may be denoted by the subscript notation (or bracket notation),
$$A[0], A[1], A[2], \dots, A[N]$$
- ✓ The number K in A[K] is called a *subscript* or an *index* and A[K] is called a *subscripted variable*.
- ✓ Subscripts allow any element of A to be referenced by its relative position in A.
- ✓ If each element in the array is referenced by a single subscript, it is called single dimensional array.
- ✓ In other words, the number of subscripts gives the dimension of that array.

Declaration of arrays:

Like any other variable arrays must be declared before they are used. The general form of declaration is:

type variable-name[size];

The type specifies the type of the elements that will be contained in the array, such as int float or char and the size indicates the maximum number of elements that can be stored inside the array for ex:

float height[50];

Declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. In C the array elements index or subscript begins with number zero.

Initialization of arrays:

We can initialize the elements in the array in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

type array_name[size]={list of values};

The values in the list are separated by commas, for example the statement

int number[3]={0,0,0};

Will declare the array size as an array of size 3 and will assign zero to each element if the number of values in the list is less than the number of elements, then only that many elements are initialized. The remaining elements will be set to zero automatically.

In the declaration of an array the size may be omitted, in such cases the compiler allocates enough space for all initialized elements. For example the statement

int counter[]={1,1,1,1};

Will declare the array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

```
/* Program to count the no of positive and negative numbers*/
#include<stdio.h>
void main()
{
    int a[50],n,count_neg=0,count_pos=0,i;
    printf("Enter the size of the arrayn");
    scanf("%d",&n);
    printf("Enter the elements of the arrayn");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(i=0;i<n;i++)
    {
        if(a[i]<0)
            count_neg++;
        else
            count_pos++;
    }
    printf("There are %d negative numbers in the arrayn",count_neg);
    printf("There are %d positive numbers in the arrayn",count_pos);
}
```

Two-dimensional Arrays

- ✓ A two-dimensional $m \times n$ array A is a collection of $m \times n$ data elements such that each element is specified by a pair of integers (such as I, J), called subscripts, with the property that,

I Represent rows & J Represent Cols

- ✓ The element of A with first subscript i and second subscript j will be denoted by, **$A[i,j]$ or $A[i][j]$** (*c language*)
- ✓ Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications; hence two-dimensional arrays are sometimes are called *matrix arrays*.
- ✓ There is a standard way of drawing a two-dimensional $m \times n$ array A where the elements of A form a rectangular array with m rows and n columns and where the element $A[i][j]$ appears in *row i* and *column j* .
- ✓ A *row* is a horizontal list of elements, and a *column* is a vertical list of elements.

Example:

<i>Columns</i>			
	0	1	2
0	$A[0][0]$	$A[0][1]$	$A[0][2]$

<i>Rows</i>	1	A[1][0] A[1][1] A[1][2]
	2	A[2][0] A[2][1] A[2][2]

- ✓ The two-dimensional array will be represented in memory by a block of m*n sequential memory locations.
- ✓ Specifically, the programming languages will store the array either
 1. Column by column, i.e. *column-major order*, or
 2. Row by row, i.e. *row-major order*.

Multi dimensional Arrays:

Often there is a need to store and manipulate two dimensional data structure such as matrices & tables. Here the array has two subscripts. One subscript denotes the row & the other the column.

The declaration of two dimension arrays is as follows:

data_type array_name[row_size][column_size];

int m[10][20]

Here m is declared as a matrix having 10 rows(numbered from 0 to 9) and 20 columns(numbered 0 through 19). The first element of the matrix is m[0][0] and the last row last column is m[9][19]

Initialization of multidimensional arrays:

Like the one dimension arrays, 2 dimension arrays may be initialized by following their declaration with a list of initial values enclosed in braces

Example:

int table[2][3]={0,0,0,1,1,1};

Initializes the elements of first row to zero and second row to 1. The initialization is done row by row. The above statement can be equivalently written as

int table[2][3]={{0,0,0},{1,1,1}}

Example:

#include<stdio.h>
main(){

int mat[10][10],i,j;


```

for(i=0;i<2;i++)
{ for(j=0;j<2;j++){
scanf("%d",&mat[i][j]);
}}
for(i=0;i<2;i++)
{ for(j=0;j<2;j++){
printf("%d",mat[i][j]);}}

}

```

Strings

Handling Of Character String

A string is a sequence of characters. Any sequence or set of characters defined within double quotation symbols is a constant string. In c it is required to do some meaningful operations on strings they are:

- ✓ Reading string displaying strings
- ✓ Combining or concatenating strings
- ✓ Copying one string to another.
- ✓ Comparing string & checking whether they are equal
- ✓ Extraction of a portion of a string

Strings are stored in memory as ASCII codes of characters that make up the string appended with '\0' (ASCII value of null). Normally each character is stored in one byte, successive characters are stored in successive bytes. **The last character is the null character having ASCII value zero.**

Character	m	y		a	g	e		i	s
-----------	---	---	--	---	---	---	--	---	---

ASCII Code	77	121	32	97	103	10	32	105	115
Character		2		(t	w	o)	\0
ASCII Code	32	50	32	40	116	119	41	0	0

❖ Initializing Strings

Following the discussion on characters arrays, the initialization of a string must be the following form which is simpler to one dimension array.

```
char month1[]={'j','a','n','u','a','r','y'};
```

Then the string month is initializing to January. This is perfectly valid but C offers a special way to initialize strings. The above string can be initialized char month1[]="January"; The characters of the string are enclosed within a pair of double quotes. The compiler takes care of string enclosed within a pair of a double quotes. The compiler takes care of storing the ASCII codes of characters of the string in the memory and also stores the null terminator in the end.

```
/*String.c string variable*/
#include <stdio.h>
main()
{
```

```

char month[15];
printf("Enter the string");
gets(month);
printf("The string entered is %s", month);
}

```

In this example string is stored in the character variable month the string is displayed in the statement.

```
printf("The string entered is %s", month);
```

It is one dimension array. Each character occupies a byte. A null character (\0) that has the ASCII value 0 terminates the string. The figure shows the storage of string January in the memory recall that \0 specifies a single character whose ASCII value is zero.

j	a	n	u	a	r	y	\0
---	---	---	---	---	---	---	----

Character string terminated by a null character '\0'.

*A string variable is any valid C variable name & is always declared as an array.
The general form of declaration of a string variable is*

```
Char string_name[size];
```

The size determines the number of characters in the string name.

Example:

```

char month[10];
char address[100];

```

The size of the array should be one byte more than the actual space occupied by the string since the compiler appends a null character at the end of the string.

❖ Reading Strings from the terminal:

The function scanf with %s format specification is needed to read the character string from the terminal.

Example:

```
char address[15];  
scanf("%s",address);
```

Scanf statement has a draw back it just terminates the statement as soon as it finds a blank space, suppose if we type the string new york then only the string new will be read and since there is a blank space after word "new" it will terminate the string.

Note that we can use the scanf without the ampersand symbol before the variable name.

In many applications it is required to process text by reading an entire line of text from the terminal.

The function getchar can be used repeatedly to read a sequence of successive single characters and store it in the array.

We cannot manipulate strings since C does not provide any operators for string. For instance we cannot assign one string to another directly.

For example:

```
String="xyz";  
String1=string2;
```

Are not valid. To copy the chars in one string to another string we may do so on a character to character basis.

❖ Writing strings to screen:

The printf statement along with format specifier %s to print strings on to the screen. The format %s can be used to display an array of characters that is terminated by the null character for example printf("%s",name); can be used to display the entire contents of the array name.

❖ Arithmetic operations on characters:

We can also manipulate the characters as we manipulate numbers in c language. When ever the system encounters the character data it is automatically converted into a integer value by the system. We can represent a character as a interface by using the following method.

```
X='a';  
printf("%d\n",x);
```

Will display 97 on the screen. Arithmetic operations can also be performed on characters for example `x='z'-1;` is a valid statement. The ASCII value of 'z' is 122 the statement therefore will assign 121 to variable x.

It is also possible to use character constants in relational expressions for example `ch>'a' && ch <= 'z'` will check whether the character stored in variable ch is a lower case letter. A character digit can also be converted into its equivalent integer value suppose on the expression `a=character-'1'`; where a is defined as an integer variable & character contains value 8 then `a= ASCII value of 8 ASCII value '1'=56-49=7`.

We can also get the support of the c library function to converts a string of digits into their equivalent integer values the general format of the function in `x=atoi(string)` here x is an integer variable & string is a character array containing string of digits.

String operations (string.h)

C language recognizes that string is a different class of array by letting us input and output the array as a unit and are terminated by null character. C library supports a large number of string handling functions that can be used to carry out many of the string manipulations such as:

- ✓ *Length (number of characters in the string).*
- ✓ *Concatenation (adding two or more strings)*
- ✓ *Comparing two strings.*
- ✓ *Substring (Extract substring from a given string)*
- ✓ *Copy (copies one string over another)*

To do all the operations described here it is essential to include string.h library header file in the program.

❖ strlen() function:

This function counts and returns the number of characters in a string. The length does not include a null character.

Syntax

`n=strlen(string);`

Where n is integer variable. Which receives the value of length of the string.

Example

```
length=strlen("Hollywood");
```

The function will assign number of characters 9 in the string to a integer variable length.

```
/*writr a c program to find the length of the string using strlen() function*/
```

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

```
include < string.h >
```

```
void main()
```

```
{char name[100];
```

```
int length;
```

```
printf("Enter the string");
```

```
gets(name);
```

```
length=strlen(name);
```

```
printf("\nNumber of characters in the string is=%d",length); }
```

❖ *strcat() function:*

when you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The strcat() function joins 2 strings together. It takes the following form

```
strcat(string1,string2)
```

string1 & string2 are character arrays. When the function strcat is executed string2 is appended to string1. the string at string2 remains unchanged.

Example

```
strcpy(string1,"sri");
```

```
strcpy(string2,"Bhagavan");
```

```
printf("%s",strcat(string1,string2);
```

From the above program segment the value of string1 becomes sribhagavan. The string at str2 remains unchanged as bhagawan.

❖ *strcmp function:*

In c you cannot directly compare the value of 2 strings in a condition like

```
if(string1==string2)
```

Most libraries however contain the strcmp() function, which returns a zero if 2

strings are equal, or a non zero number if the strings are not the same. The syntax of strcmp() is given below:

```
strcmp(string1,string2)
```

String1 & string2 may be string variables or string constants. String1, & string2 may be string variables or string constants some computers return a negative if the string1 is alphabetically less than the second and a positive number if the string is greater than the second.

Example:

strcmp("Newyork","Newyork") will return zero because 2 strings are equal.

strcmp("their","there") will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.

strcmp("The", "the") will return 32 which is the numeric difference between ASCII "T" & ASCII "t".

❖ *strcmpi() function*

This function is same as strcmp() which compares 2 strings but not case sensitive.

Example

strcmpi("THE","the"); will return 0.

❖ *strcpy() function:*

C does not allow you to assign the characters to a string directly as in the statement name="Robert";

Instead use the strcpy() function found in most compilers the syntax of the function is illustrated below.

```
strcpy(string1,string2);
```

Strcpy function assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

```
strcpy(Name,"Robert");
```

In the above example Robert is assigned to the string called name.

❖ **strlwr () function:**

This function converts all characters in a string from uppercase to lowercase.

syntax

strlwr(string);

For example:

strlwr("EXFORSYS") converts to Exforsys.

❖ **strrev() function:**

This function reverses the characters in a string.

Syntax

strrev(string);

For ex strrev("program") reverses the characters in a string into "margrop".

❖ **strupr() function:**

This function converts all characters in a string from lower case to uppercase.

Syntax

strupr(string);

For example strupr("exforsys") will convert the string to EXFORSYS.

```
/* Example program to use string functions*/
#include <stdio.h>
#include <string.h>
void main()
{
    char s1[20],s2[20],s3[20];
    int x,l1,l2,l3;
    printf("Enter the strings");
    scanf("%s%s",s1,s2);
    x=strcmp(s1,s2);
    if(x!=0)
    {printf("\nStrings are not equal\n");
    strcat(s1,s2);
    }
    else
    printf("\nStrings are equal");
}
```



```
strcpy(s3,s1);
l1=strlen(s1);
l2=strlen(s2);
l3=strlen(s3);
printf("\ns1=%s\t length=%d characters\n",s1,l1);
printf("\ns2=%s\t length=%d characters\n",s2,l2);
printf("\ns3=%s\t length=%d characters\n",s3,l3);
}
```

What is pointer in c programming?

Pointer is a user defined data type which creates special types of variables which can hold the address of primitive data type like **char**, **int**, **float**, **double** or user defined data type like function, pointer etc. or derived data type like array, structure, **union**, **enum**.

Examples:

```
int *ptr;
int (*ptr)();
int (*ptr)[2];
```

In c programming every variable keeps two type of value.

1. Contain of variable or value of variable.
2. Address of variable where it has stored in the memory.

(1) Meaning of following simple pointer declaration and definition:

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

Explanation:

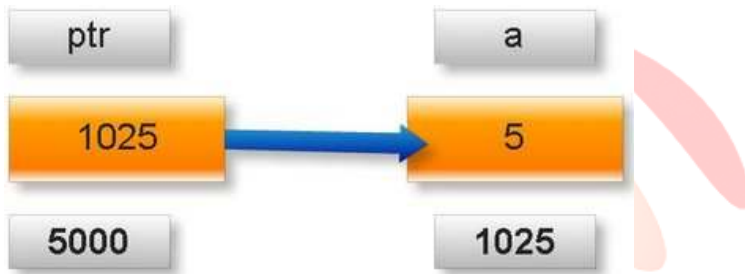
About variable a:

1. Name of variable : a
2. Value of variable which it keeps: 5
3. Address where it has stored in memory : 1025 (assume)

About variable ptr:

4. Name of variable : ptr
5. Value of variable which it keeps: 1025
6. Address where it has stored in memory : 5000 (assume)

Pictorial representation:



Note: A variable where it will be stored in memory is decided by operating system. We cannot guess at which location a particular variable will be stored in memory.

(2) Meaning of following pointer declaration and definition:

```

int a=50;
int *ptr1;
int **ptr2;
ptr1=&a;
ptr2=&ptr1;
  
```

Explanation:

About variable a:

1. Name of variable : a
2. Value of variable which it keeps: 50
3. Address where it has stored in memory : 5000 (assume)

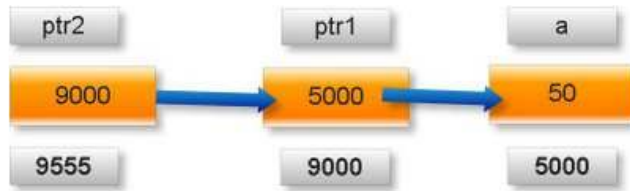
About variable ptr1:

4. Name of variable : ptr1
5. Value of variable which it keeps: 5000
6. Address where it has stored in memory : 9000 (assume)

About variable ptr2:

7. Name of variable : ptr2
8. Value of variable which it keeps: 9000
9. Address where it has stored in memory : 9555 (assume)

Pictorial representation of above pointer declaration and definition:



Note:

* is known as indirection operator which gives content of any variable.

& is known as reference operator which gives address where variable has stored in memory.

❖ Pointers :: Pointers and const Type Qualifier

- The const type qualifier can make things a little confusing when it is used with pointer declarations.
- The below example is from Weiss pg. 132:
- `const int * const ip; /* The pointer *ip is const and what it points at is const */`
- `int * const ip; /* The pointer *ip is const */`
- `const int * ip; /* What *ip is pointing at is const */`
- `int * ip; /* Nothing is const */`

As you can see, you must be careful when specifying the const qualifier when using pointers.

❖ Arithmetic operation with pointer in c programming

Address arithmetic is a method of calculating the address of an object with the help of arithmetic operations on pointers and use of pointers in comparison operations. Address arithmetic is also called pointer arithmetic.

According to C language standards, the result address must remain strictly within the bounds of a single array object (or just after it).

Adding or subtracting from a pointer moves it by a multiple of the size of the data type it points to. For example, assume we have a pointer to an array of 4-byte integers. Incrementing this pointer will increment its value by 4 (the size of the element). This effect is often used to increment a pointer to point at the next element in a contiguous array of integers.

Rule 1:

Address + Number = Address

Address - Number = Address

Address++ = Address

Address-- = Address

++Address = Address

--Address = Address

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

If we will add or subtract a number from an address result will also be an address.

(1)What will be output of following c program?

```
void main(){
    int d=67;
    int *ptr=&d;
    ptr=ptr+1;
    printf("%u",ptr);
}
```

If d has address 1000 then

Output: 1002

(2)What will be output of following c program?

```
void main(){
    double d=7.89;
    double *p=(double *)1000;
    p=p+1;
    printf("%u",p);
}
```

If d has address 1000 then

Output: 1008

❖ Uses Of Pointer

- ✓ Pointer are used for saving memory space.
- ✓ Use of pointer, assigns the memory space and also releases it. this concept helps in making the best use of the available memory (dynamic memory allocation).
- ✓ With pointers, data manipulation is done with address, so the execution time is faster.
- ✓ Two-dimensional and multi-dimensional array representation is easy with pointer.
- ✓ Concept of pointer is used in data structures such as linked list.

❖ The & And * Operator

The '&' is the address operator, the address variable

Ex-

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

```
main()
```

```
{    int a=5;
```

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

```
printf("vaule of a=%d\n",a);
```

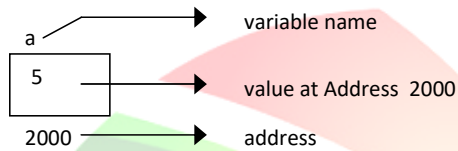
```
printf("address of a=%u\n",&a);
```

the '%u' is used for obtaining the address.

Output;

value of a=5

address of a=2000



❖ Declaration Of Pointer

Let us see, how we use the pointer in the expression. The & operator represents the address of variable if we want to give the address of variable to another variable then we can write as –

```
b=&a;
```

here 'b' is the variable which contains the address of the variable 'a'.



2000 3000

here value of 'a' is 5 and 'b' has the value 2000 which is address of 'a'.

when a pointer variable is declared then an asterisk (*) symbol should precede the variable name here b is a pointer type variable which points to the variable a . Hence it must be declared as –

```
int a=5;
```

```
int *b;
```

Here 'b' is an int type of pointer . Value at address contained in 'b' is an int the address of any variable is a whole number. Pointer variable contains the address of any variable , hence pointer variable always contains the whole number. Similarly we can declare character and floating type pointer-

char *p;

float *q;

char *p means 'p' contains the address of the variable, which is of character type variable .float *q contains the address of the variable which is of floating type .

Example

```
#include<stdio.h>
main ()
{
    Int a=5;
    Int *b;
    b= &a;
    printf("value of a =%d\n",a);
    printf("value of a =%d\n",*(&a));
    printf("value of ab=%d\n",*b);
    printf("address of a =%d\n",&a);
    printf("address of b =%d\n",b);
    printf("address of b=%d\n",&b);
    printf("value of b =address of %u\n",b);
```

<i>a</i>
5

<i>b</i>
2000

2000

300

out put

value of a= 5

value of a= 5

value of a= 5

address of a=2000

address of a= 2000

address of b= 3000

value of b=address of a=2000

Arrays and Pointers

❖ Arrays with Single Dimension

In C there is a very close connection between pointers and arrays. In fact they are more or less one and the same thing! When you declare an array as:

```
int arr[10];
```

you are in fact declaring a pointer **arr** to the first element in the array. That is, **arr** is exactly the same as **&arr[0]**. The only difference between **arr** and a *pointer variable* is that the array name is a constant pointer - you cannot change the location it points at. When you write an expression such as **arr[i]** this is converted into a pointer expression that gives the value of the appropriate element. To be more precise, **arr[i]** is exactly equivalent to ***(arr+i)** i.e. the value pointed at by **arr + i**. In the same way ***(arr+ 1)** is the same as **arr[1]** and so on.

Array Access	Pointer Equivalent
arr [0]	*arr
arr [2]	*(arr + 2)
arr [n]	*(arr + n)

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the

values in the array, but not the array itself, so statements like `arr ++` are illegal, but `arr[n] ++` is legal.

Since an array is like a pointer, we can pass an array to a function, and modify elements of that array without having to worry about referencing and de-referencing. Since the array is implemented as a hidden pointer, all the difficult stuff gets done automatically.

Access one Dimension array using pointer(dereferencing):

Here is a simple program that illustrates the relationship between array elements and their addresses

If **arr** is a one dimensional array, then the address of the first array element can be expressed as either `&arr[0]` or simply as `arr`.

```
#include<stdio.h>
void main()
{int arr[]={3,5,8,9},i;
for(i=0;i<=3;i++)
{
printf("Value:%u , Address:%u\n",*(arr+i),(arr+i));
} }
Value:3 , Address:8694
Value:5 , Address:8696
Value:8 , Address:8698
Value:9 , Address:8700
```

Pointer to An Array

A pointer to an array is a single pointer that holds the address of an array of variables.

```
#include<stdio.h>
void main()
{int a[]={3,5,6,7},i;
int *ptr=a;//pointer to array
for(i=0;i<=3;i++)
{ printf("value:%d,address:%u\n",ptr[i],ptr+i);
```


}}

Output:

value:3,address:8692

value:5,address:8694

value:6,address:8696

value:7,address:8698

Arrays with Multidimension

Suppose there is matrix of 3x3 i.e three rows and each contains three integer

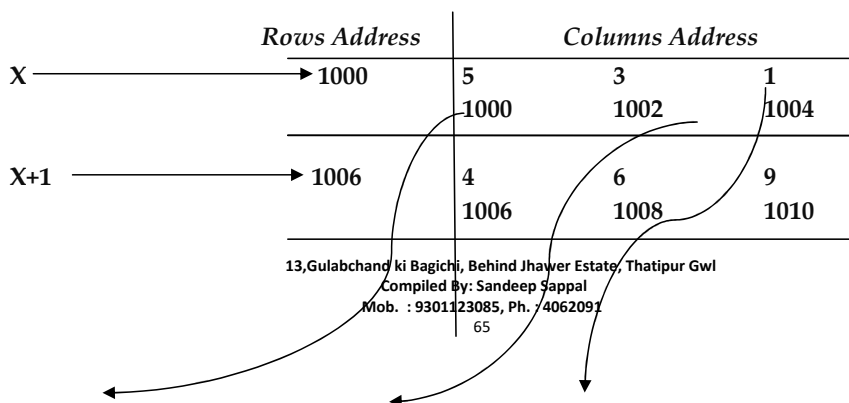
For Example

Int X[][3]={5,3,1,4,6,9,8,12,10};

Then size of each row is 6 bytes (because size of int is 2 and each row contains 3 elements)

And the cols size is 2.

Values	5	3	1	4	6	9	8	12	10
Address	1000	1002	1004	1006	1008	1010	1012	1014	1016
Row 0			Row 1			Row 2			



X+2	→	1012	8	12	10
			1012	1014	1016

*x or *(x+0)+0 *(x+0)+1 *(* (x+0)+2)

In above example **x** is a two-dimensional integer array having 3 rows and 3 columns. The item in row 1, column 1 can be accessed by writing either

X[1][1]
or
*** (* (x + 1) + 1)**

The second form requires some explanation. First, note that **(x + 1)** is a pointer to row 1. Therefore the object of this pointer, *** (x + 1)**, refers to the entire row. Since row 1 is a one-dimensional array, *** (x + 1)** is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence, **(* (x + 1) + 1)** is a pointer to element 1 (i.e., the second element) in row 2. The object of this pointer, *** (* (x + 1) + 1)**, therefore refers to the item in column 1 of row 1, which is **x[1][1]**.

Example:

```
#include<stdio.h>
void main()
{ int x[][3]={5,3,1,4,6,9,8,12,10};
  printf("Row address %u\n",x);
  printf("Address 0 row 0 col:%u\n",*(x+0)+0);
  printf("Value 0 row 0 col:%u\n",*(* (x+0)+0));
  for(int i=0;i<=2;i++)
  { for(int j=0;j<=2;j++)
    {
      printf("Address:%u,",*(x+i)+j);
      printf("Value:%u\n",*(* (x+i)+j));
    }
  }
}
```

Output

Row address 1000
 Address 0 row 0 col:1000
 Value 0 row 0 col:5

Address:1000,Value:5
 Address:1002,Value:3

Address:1004,Value:1
 Address:1006,Value:4
 Address:1008,Value:6
 Address:1010,Value:9
 Address:1012,Value:8
 Address:1014,Value:12
 Address:1016,Value:10

Pointer & Strings

There is very close relationship between string and pointer because string always work on address means string constant first directly stored in memory and return the base address. Thus we can directly assign String in pointer variable.

Example

Char x[]="New Delhi";

Or

Char *x="New Delhi";

Both are valid statement.

Value	N	e	w		D	e	l	h	i	NULL
Address	8002	8003	8004	8005	8006	8007	8008	8009	8010	8011

printf("%u",x);

print base address 8002

printf("%s",x);

read string from base address 8002 to NULL Pointer i.e New Delhi

printf("%u",x+4);

print address 8006

printf("%s",x+4);

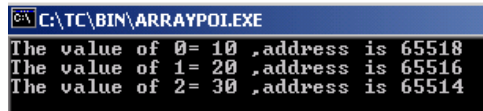
```
print Delhi
printf("%c", *x);
value at address 8002 i.e. N
printf("%c", *(x+2));
value at address 8004 i.e. w
```

Array of pointers

An array of pointers is a series of pointers that can each hold a memory address

```
#include <stdio.h>
#include <conio.h>
main() {
    clrscr();
    int *array[3];
    int x = 10, y = 20, z = 30;
    int i;
    array[0] = &x;
    array[1] = &y;
    array[2] = &z;
    for (i=0; i<3; i++) {
        printf("The value of %d= %d ,address is %u\t \n", i, *(array[i]),
            array[i]);
    }
    getch();
    return 0;
}
```

Output will be displayed as:



```
C:\TC\BIN\ARRAYPOLEXE
The value of 0= 10 ,address is 65518
The value of 1= 20 ,address is 65516
The value of 2= 30 ,address is 65514
```

Array of String using Pointers

The declaration of an array of character pointers is an extremely useful extension to single string pointer declarations. In the example below, A two element array of character pointers where each element is a pointer to a character. We have initialized the array with two elements by giving index numbers. In C, format specifier %s is used with the **printf** to print the string.

Here is the code:

```
#include <stdio.h>
#include <conio.h>
void main() {
    clrscr();
    char *array[2];
    array[0]="Hello";
    array[1]="World";
    printf("The Array of String is = %s,%s\n", array[0], array[1]);
    getch();
}
```

Output:

```
C:\TC\BIN\ARRAYOF.EXE
The Array of String is = Hello,World
```

Dynamic memory allocation

The process of allocating memory at run time is known as dynamic memory allocation. Although c does not inherently have this facility there are four library routines which allow this function.

Many languages permit a programmer to specify an array size at run time. Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program. But c inherently does not have this facility but supports with memory management functions, which can be used to allocate and free memory during the program execution. The following functions are used in c for purpose of memory management.

Function	Task
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free	Frees previously allocated space

realloc	Modifies the size of previously allocated space.
---------	--

Memory allocations process:

According to the conceptual view the program instructions and global and static variable in a permanent storage area and local area variables are stored in stacks. The memory space that is located between these two regions is available for dynamic allocation during the execution of the program. The free memory region is called the heap. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a null pointer.

Allocating a block of memory(malloc())

A block memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast-type*)malloc(byte-size);
```

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with **size byte-size**.

Example:

```
x=(int*)malloc(100*sizeof(int));
```

//Example malloc

```
#include <malloc.h>
```

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

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

```
int main() {
```

```
    int* array;
```

```
    int n, i;
```

```
    printf("Enter the number of elements: ");
```

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

```
    array = (int*) malloc(n*sizeof(int));
```

```
    for (i=0; i<n; i++) {
```

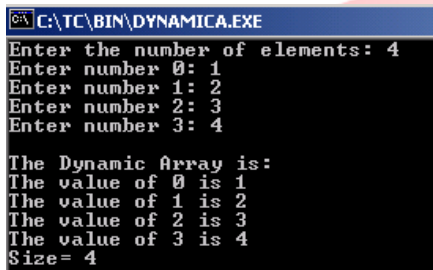
```
        printf("Enter number %d: ", i);
```

```
        scanf("%d", &array[i]);
```

```

}
printf("\nThe Dynamic Array is: \n");
for (i=0; i<n; i++) {
    printf("The value of %d is %d\n", i, array[i]);
}
printf("Size= %d\n", i);
getch();
return 0;
}

```



```

C:\TC\BIN\DYNAMICA.EXE
Enter the number of elements: 4
Enter number 0: 1
Enter number 1: 2
Enter number 2: 3
Enter number 3: 4
The Dynamic Array is:
The value of 0 is 1
The value of 1 is 2
The value of 2 is 3
The value of 3 is 4
Size= 4

```

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int.

Allocating multiple blocks of memory (calloc())

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of calloc is:

```
ptr=(cast-type*) calloc(n,elem-size);
```

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

Releasing the used space (free())

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

free(ptr);

ptr is a pointer that has been created by using malloc or calloc.

To alter the size of allocated memory: (realloc())

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is :

realloc

It is often useful to be able to grow or shrink a block of memory. This can be done using realloc which returns a pointer to a memory region of the specified size, which contains the same data as the old region pointed to by pointer (truncated to the minimum of the old and new sizes). If realloc is unable to resize the memory region in place, it allocates new storage, copies the required data, and frees the old pointer. If this allocation fails, realloc maintains the original pointer unaltered, and returns the null pointer value. The newly allocated region of memory is uninitialized (its contents are not predictable). The function prototype is
void *realloc(void *pointer, size_t size);

Need For User-Defined Functions

When program grew larger program maintaining becomes difficult. If a program is divided into functional parts these each part may be independently coded and later combined into a single unit. These subprograms called functions are much easier to understand, debug, and test. ***Functions can be called and used whenever required. This saves both time and space.***

❖ **This approach clearly results in a number of advantages.**

- ✓ It facilitates top-down modular programming
- ✓ The length of a source program can be reduced by using function at appropriate places.
- ✓ As mentioned earlier, it is easy to locate and isolate a faulty function for further investigations.
- ✓ A function may be used by many other programs.

Example:

```
#include<stdio.h>
int Sum (int,int); //function prototype
void main()
{ int a=10,b=20,c;
```

//CALLING FUNCTION

```
c=Sum(a,b); //jump to Called Function here a and b are actual arguments  
printf("%d",c);  
}
```

//CALLED FUNCTION

```
int sum(int a, int b) //a and b are formal arguments  
{  
    int k;  
    k=a+b;  
    return(k); //return a Value to calling function  
}
```

Calling Function :- Calling function c=Sum(a,b) call the Called Function as in Above example.

How Function Can be Declare:(Function prototype)

Syntax

```
<Return type> <Function Name> (<Argument 1,2,3..>)  
{  
-----  
-----  
}
```

Example: int Sum (int,int);

Return type *Function Name* *Arguments*

- ❖ **Function Name:** A function name follow the same rules of formation as other variables name in C.
- ❖ **Return Type :** A function may or may not be send back any value to the calling function. If it does, it is done through the return statement. The called fuction can only return one value per call.
Return type Can be declare as int, float , char etc.if it decalre as int, float, char etc. type, Called function returns a value according to its type to calling function.if return type is void no value is return by Called Function.

Example:

```
return(a);
```

- ❖ **Arguments:** Arguments are used to Transfer the Values from One function to another function. The argument list contains Valid variable names separated by commas. The list must be surrounded by parentheses.

Example

```
Power(a,b);
```

| _____ *Argument list.*

The arguments variables receive values from the calling function, thus providing a communication from the calling function to the called function.

Category Of Functions

- ✓ 1) Functions with no argument and no return values
- ✓ 2) Functions with Arguments and no return values
- ✓ 3) Functions with arguments and return values

❖ **Functions with no argument and no return values**

When a function has no arguments, it does not receive any data from the calling function. Similarly, When it does not return a value, the calling function does not receive any data from the called function.

Example:

```
void Message (void);  
main()  
{  
    Message(); // Calling function  
}
```

//Called Function

```
void Message (void)  
{ printf( "Welcome to PS-SOFTECH");  
}
```

❖ **Functions with Arguments and no return values**

When a function has arguments, it receives the data from the calling function. Similarly, When it does not return a value, the calling function does not receive any data from the called function.

Example:

```
#include<stdio.h>
void Sum (int,int); //function prototype

void main()
{ int a=10,b=20,c;
  //CALLING FUNCTION
  Sum(a,b);    //jump to Called Function here a and b are actual arguments

}
//CALLED FUNCTION
void sum(int a, int b) //a and b are formal arguments
{ int k;
  k=a+b;
  printf("%d",c);
}
```

❖ Functions with arguments and return values

When a function has arguments, it receives the data from the calling function. Similarly, When return type is declared, the calling function receives a data from the called function.

Example:

```
#include<stdio.h>
int Sum (int,int); //function prototype

void main()
{ int a=10,b=20,c;
  //CALLING FUNCTION
  c=Sum(a,b); //jump to Called Function here a and b are actual arguments
  printf("%d",c);
}
//CALLED FUNCTION
int sum(int a, int b) //a and b are formal arguments
{ int k;
  k=a+b;
  return(k); //return a Value to calling function
}
```

❖ Actual Arguments and Formal Arguments

Data or parameters or arguments which define in calling function are the **actual arguments** and the parameters which define in called function are

treated as **formal arguments**. The actual arguments should match in number, type and order.

The Value of actual arguments are assigned to the formal arguments on one to one basis, starting with the first argument.

Example:

```
int Sum (int,int); //function prototype
main()
{ int a=10,b=20,c;
  //calling function contains actual argument
  c=Sum( a, b);
}
int sum(int a,int b) // called function contains formal arguments
{
  int k;
  k=a+b;
  return(k); }
```

Call By Value Call By Reference

Arguments can generally be passed to functions in one of the two ways:

- ✓ sending the values of the arguments
- ✓ sending the address of the arguments

In the first method the 'value' of each actual argument in the calling function is copied into corresponding formal argument of the called function. With this method changes made to the formal argument in the called function have no effect on the values of the actual argument in the calling function.

The following program illustrates the 'Call by Value'.

```
main ()
{
  int a=10;
  int b=20;
  printf("%d,%d\n",a,b);
  swapv(a,b);
  printf("%d,%d\n",a,b);
}
swapv(int x,int y)
{
  int t;
  t=x;
```

```
x=y;  
y=t;  
}
```

The output of the above program will be:

10,20

10,20

Note that values of a and b remain unchanged even after exchanging the value of x and y.

In the second method (call by reference) the addresses of actual argument in the calling function are copied into formal argument of the called function. This means that using the formal arguments in the called function we can make changes in the actual arguments of the calling function. the following program illustrates this fact.

```
main()  
{  
  int a=10, b=20;  
  printf("%d,%d\n",a,b);  
  swapr(a,b);  
  printf("%d,%d\n",a,b);}
```

```
void swapr(int*x,int*y)  
{  
  int t;  
  t=*x;  
  *x=*y;  
  *y=t;  
}
```

The out of the above program would be:

10,20

20,10

Pass Array Into A Function

One can also pass array as a parameter in function

```
#include <stdio.h>
void printarr(int a[]) {
    int i;
    for(i = 0;i<5;i++) {
        printf(" %d\n",a[i]);
    }
}
main() {
    int a[5];
    int i;
    for(i = 0;i<5;i++) {
        a[i]=i;
    }
    printarr(a);//send address as parameter
}
```

Output

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091

0
1
2
3
4

The scope and lifetime of variables in function.

Variables in C differ in behaviour from those in most other languages. As mentioned earlier, a variable in C can have any one of the four storage classes.

- ✓ 1 Automatic variables
- ✓ 2 External variables
- ✓ 3 Static variables
- ✓ 4 Register variables

❖ Automatic Variables

Properties

Storage	<i>Memory.</i>
Default initial value	<i>An unpredictable value, which is often called a garbage value.</i>
Scope	<i>Local to the block in which the variable is defined.</i>
Life	<i>Till the control remains within the block in which the variable is defined.</i>

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are therefore

private (or local) to the function in which they are declared . Because of this property , automatic variables are also referred to as local or internal variables. One important feature of automatic variable is that their value cannot be changed accidentally by what happens in some other function in the program.

❖ External Variables

Properties:

<i>Storage</i>	<i>Memory.</i>
<i>Default initial value</i>	<i>Zero.</i>
<i>Scope</i>	<i>Global.</i>
<i>Life</i>	<i>As long as the program's execution doesn't come to an end.</i>

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Unlike local variables , global variables can be accessed by any function in the program .External variables are declared outside a function .

❖ Static Variables

Properties

<i>Storage</i>	<i>Memory.</i>
<i>Default initial value</i>	<i>Zero.</i>
<i>Scope</i>	<i>Local to the block in which the variable is defined.</i>
<i>Life</i>	<i>Value of the variable persists between different function calls.</i>

As the name suggests , the value of static variable persists until the end of the program .A variable can be declared static using the keyword static .A static variable may be either an internal type or an external type , depending on the place of declaration.Internal static variables are those which are declared inside a function .The scope of internal static variables extends upto the function in which they are defined.Therefore , internal static variables are similar to auto variables , except that they remain in existence(alive) throughout the remainder of the program .

Therefore, internal static variables can be used to retain values between function calls . An external static variable is declared outside of all functions and is available to all the functions in that program .The difference between a static external variable and a simple external variable is that the static external variables are

available only within the file where it is defined while the simple external variable can be accessed by other files.

❖ Register Variables.

Properties

Storage	<i>CPU registers.</i>
Default initial value	<i>Garbage value.</i>
Scope	<i>Local to the block in which the variable is defined.</i>
Life	<i>Till the control remains within the block in which the variable is defined.</i>

We can tell the compiler that a variable should be kept in one of the machine registers, instead of keeping in the memory since a register's access is much faster than a memory access, keeping the frequently accessed variable in the register will lead to faster execution of programs. This is done as follows:

```
register int cout;
```

Since only a few variables can be placed in the register, it is important to carefully select the variable.

Example:

```
/* Demonstrating Global variables */
#include <stdio.h>
/* These are global variables and can be accessed by functions from this point on */
int value1, value2, value3;

int add_numbers( void )
{
    auto int result;
    result = value1 + value2 + value3;
    return result;
}

main()
{
    auto int result;
    value1 = 10;
    value2 = 20;
    value3 = 30;
    result = add_numbers();
    printf("The sum of %d + %d + %d is %d\n",
```

```
        value1, value2, value3, final_result);  
    }
```

Sample Program Output

The sum of 10 + 20 + 30 is 60

Example:

/ example program illustrates difference between static and automatic variables */*

```
#include <stdio.h>  
/* ANSI function prototypes */  
void demo( void )  
{  
    auto int avar = 0;  
    static int svar = 0;  
    printf("auto = %d, static = %d\n", avar, svar);  
    ++avar;  
    ++svar; }  
void main()  
{  
    int i;  
    while( i < 3 ) {  
        demo();  
        i++;  
    }  
}
```

Sample Program Output

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091
83

auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2

Introduction Recursion

Recursion: Recursion is a process by which a function calls itself repeatedly. Until some specified conditions has been satisfied. This process is used for repetitive computations in which each action is started in terms of a previous result.

Some Important Point About Recursion

1. There must be a terminating condition for problems, which you want to solve with recursion this condition is called the base case for that problem
2. Reversal in the order of execution is the characteristic of every recursive problem. That is when a recursive program calls are not executed immediately. They are pushed onto stack as long as the terminating condition is encountered. as soon as recursive condition is encountered the recursive calls which they are pushed onto stack, are removed in reverse order (i.e. reverse order in which they are pushed onto stack) and executed one-by-one. It means that stack will be empty.

3. Each time a new recursive call is made ,a new memory space is allocated to each automatic (and registers variables if used) variables used by the recursive routine

Disadvantages of Recursion

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and executed time.
4. According to some computer professionals recursive does not offer any concrete.

Advantages over non-recursive procedures/function.

1. If proper precautions are not taken, recursion may result is non-terminating iterating
2. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

Types of Recursion=> the recursive functions are characterized based on.

- 1) Whether the function calls itself or not (direct or indirect recursion).
- 2) Whether there are pending operations at each recursive call(tail-recursive or not).
- 3) The shape of the calling pattern whether pending operations are also recursive(linear or free-recursive).

Direct Recursion: A function is directly recursive is it contains an explicit call to itself. For example the function.

```
void call(int x)
{
    If(x!=0)
    { printf("%d",x);
```

```

call(x-1);}
}

```

Includes a call to itself. so its directly recursive. The recursive call will occur for positive value of x.

Indirect Recursion A function myfun() is indirectly recursive if it contains a call to another function which ultimately calls myfun()

The following pair of function is indirectly recursive they call each other they are also known as mutually recursive function.

```

void myfun(int x)
{if(x<=0)
{printf("%d",x)
more(x);}
}
void more(int y)
{
printf("%d",y);
myfun(y-1);
}

```

Tail recursion: A recursive function is said to be tail recursive if there are no pending operation to be performed on return from a recursive call. Tail recursive function are often said to “return the value of the last recursive call as the value of the function.”

```

/* Calculate factorial by using recursion */
#include <stdio.h>
#include <conio.h>

```

```

int fact(int k)
{
if(k==0)
return 1;
else
return k*fact(k-1);
}

```

```

void main()

```

```

{
int n;
clrscr();
printf("\n Enter a number :");
scanf("%d",&n);
printf("\n Factorial value=%d",fact(n));
getch();
}

```

Output:

Enter a number :6
Factorial value=720

Iteration	Recursion
<ol style="list-style-type: none"> 1. It is a process of executing a statement or a set of statement repeatedly, until some specified condition is specified. 2. Iteration involves four clear cut steps initialization, condition executed and updating. 3. Any recursive problem can be solved iteratively. 4. Iterative counterpart of a problem is more efficient terms of memory utilization and executed speed. 	<ol style="list-style-type: none"> 1. Recursion is the technique of defining anything in terms of itself. 2. There must be an exclusive if statement inside the recursive function, specifying stopping condition. 3. Not all problems have recursive solution. 4. Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

Structures

1. Definition:

struct : Declares a structure, an object consisting of multiple data items that may be of different types.

2. DEFINING A STRUCTURE:

Syntax:

struct tag

optional

{

Don't forget the

data-type member 1;

data-type member 2;

.....

data-type member m;

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

};

Here, *struct* is the required keyword; *tag* (optional) is a name that identifies structures of this type; and member1, member2, ..., member m are individual member declarations.

- The individual members can be ordinary variables, pointers, arrays, or other structures.
- A storage class cannot be assigned to an individual member, and individual members can not be initialized within a structure type declaration.

3. DECLARING STRUCTURE VARIABLES:

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

storage-class struct tag variable1, variable2, ..., variable n;

where storage-class is an optional storage class specifier, struct is a required keyword, tag is the name that appeared in the structure declaration and variable1, variable2, ..., variable n are structure variables of type tag.

Example:

```
struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
};
```

Here, regno, name, dept and year are the members of the student structure. And this is the definition of the datatype. So, no memory will be allocated at this stage. The memory will be allocated after the declaration only. Structure variables can be declared as following methods:

a) Normal way of declaration

```
struct student s1, s2;
```

b) It is possible to combine the declaration of the structure composition with that of the structure variables, as shown below:

```
struct student  
{  
    int regno;  
    char name[20];  
    char dept[10];  
    int year;  
} s1, s2;
```

c) If we are going to declare all the necessary structure variables at definition time then we can create them without the tag, as shown below:

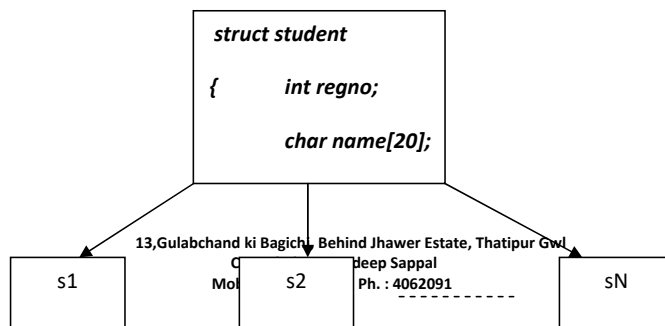
```
struct  
{  
    int regno;  
    char name[20];  
    char dept[10];  
    int year;  
} s1, s2;
```

Since there is no tag name, additional variables can not be generated other than this location. i.e. can't create new variables with this structure in the local functions. If we want we have to redefine the structure variable once again.

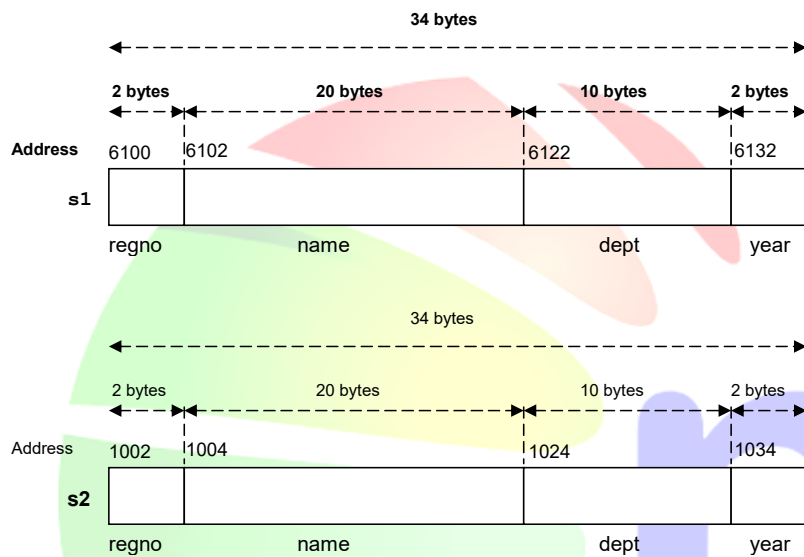
d) If we use the *typedef* in front of the *struct* keyword then the tag name alone can be used in other places whenever you want to use the student data type.

```
typedef struct student  
{  
    int regno;  
    char name[20];  
    char dept[10];  
    int year;
```

```
}; student s1, s2; /* here the struct keyword is not needed because of typedef */
```



The size of each of these variables is 34 bytes because the size of the student datatype is 34 bytes. And the memory will be allocated for each variable as follows:



4. INITIALIZING STRUCTURE VARIABLES:

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas.

The general form is,

storage-class struct tag variable = {value1, value2, ...,value n};

A structure variable, like an array, can be initialized only if its storage class is either external or static.

Example:

```
static struct student s1 = { 340, "Kumara Vel", "CSE", 3};
static struct student s2 = {533, "Sankari", "CSE", 4};
```

5. STORING VALUES INTO THE MEMBERS OF THE STRUCTURE VARIABLES:

a) Values may be stored by assignment operation.

```
s1.regno = 500;
strcpy(s1.name, "Surya");
strcpy(s1.dept, "CSE");
s1.year = 3;
```

b) also the scanf statement may be used to give values through the keyboard.

```
scanf("%d", &s1.regno);
scanf("%s", s1.name);
scanf("%s", s1.dept);
scanf("%d", &s1.year);
```

OR

```
scanf("%d%s%s%d", &s1.regno, s1.name, s1.dept, &s1.year);
```

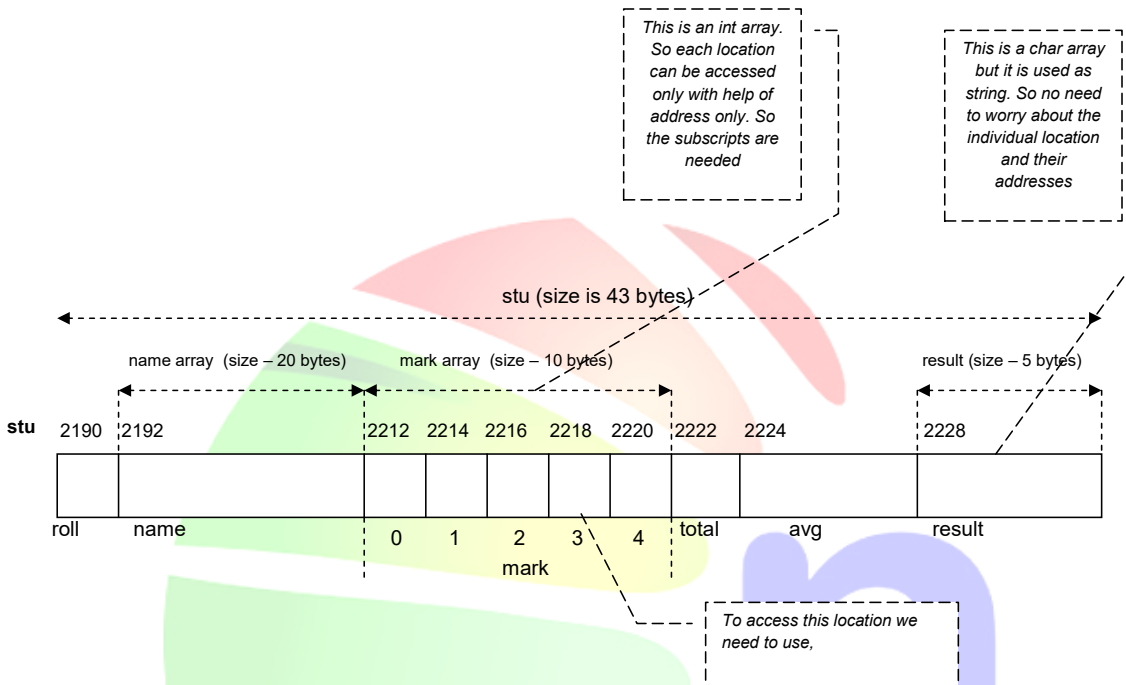
6. ARRAYS IN THE STRUCTURE:

The derived data types like array can be included in the structure as a member.

Example:

```
struct student
{
    int roll;
    char name[20];
    int marks[5];
    int total;
    float avg;
    char result[5];
}stu;
```

In memory it would be stored as given below:



7. NESTED STRUCTURES:

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

Example:

struct date

```
{
    int day;
    int month;
    int year;
};
```

struct bill

```
{
    int cno;
```

struct bill

```
{
    int cno;
    char name[20];
    float amt;
    struct date
```

```
{
```



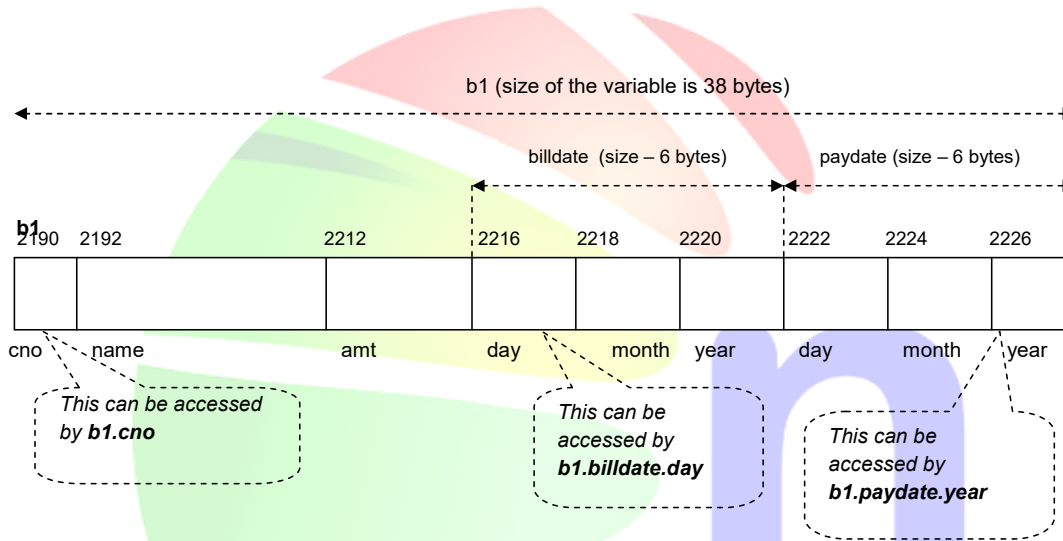
```

char name[20];
float amt;
struct date billdate;
struct date paydate;

```

}b1, b2;

The second structure bill now contains another structure, date, as one of its members. The structure may look like as follows:



8. PROCESSING STRUCTURES:

Consider the following structure:

```

struct student
{
    int regno;
    char name[20];
    char dept[10];
    struct date
    {
        int day;
        int month;
        int year;
    } bday;
}

```

```

    int marks[5];
    int year;
} s1;

```

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

structure_variable.member

where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure. The period (.) separates the variable name from the member name. It is a member of the highest precedence group, and its associativity is left to right.

Example:

s1.regno, s1.name, s1.dept, s1.year

A nested structure member can be accessed by writing

structure_variable.member.submember;

Example:

s1.bday.day, s1.bday.month, s1.bday.year

where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure. Similarly, if a structure is an array, then an individual array element can be accessed by writing

structure-variable.member[expression];

Example:

s1.mark[0], s1.mark[1], s1.mark[2], s1.mark[3], s1.mark[4]

10. POINTERS TO STRUCTURES:

The address of a given structure variable can be obtained by using the & operator. Pointers to structures, like all other pointer variables may be assigned addresses. The following statements illustrate this concept.

Example:

```

struct student
{

```

```

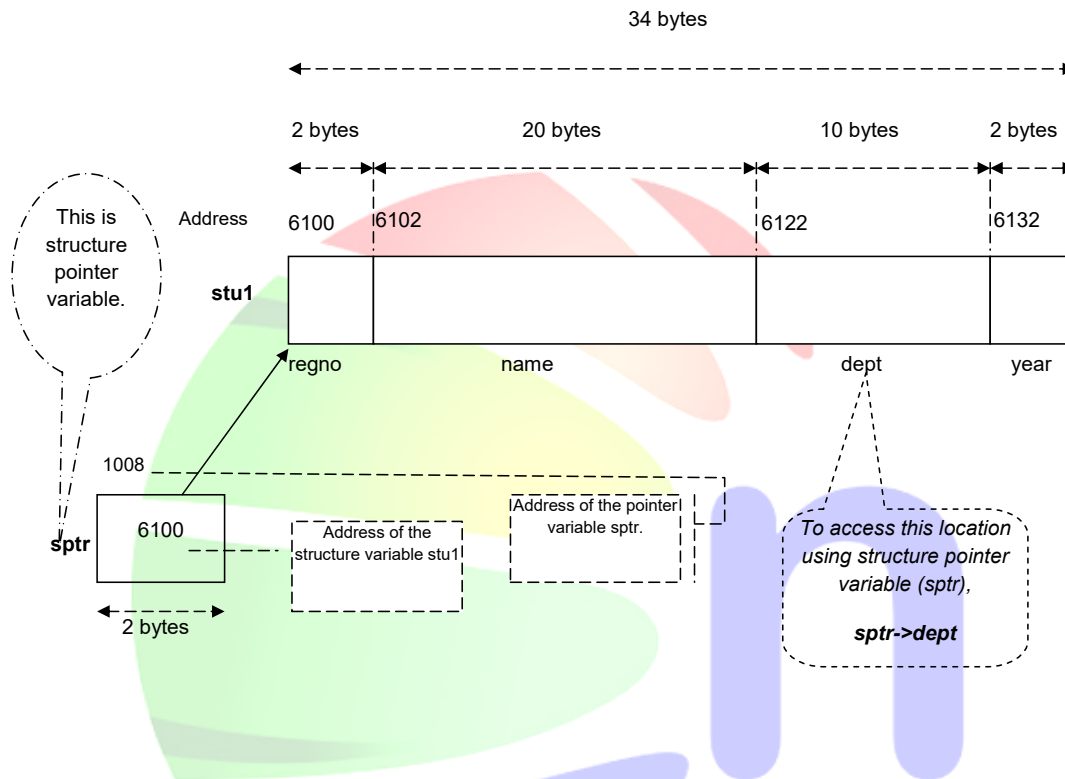
    int regno;
    char name[20];
    char dept[10];
    int year;

```

This is structure variable

This is structure pointer variable
 13, Gulabchand ki Bagichi, Behind Shalwar Estate, Thakipur Gwl
 Compiled By: Sandeep Kumar
 Mob.: 9301123085, Ph.: 4062091

```
};
struct student stu, *sptr;
sptr = &stu;
```



Access to members of the structure is shown below:

```
printf("Student Registration Number : %d\n", sptr->regno);
printf("Student Name : %s\n", sptr->name);
printf("Department Name : %s\n", sptr->dept);
printf("Year of Study : %d\n", sptr->year);
```

❖ Passing Structure to Function

```
#include<stdio.h>
struct student
{
```

```
int rollno;
```



```

        char name[20];
        int year;
    };
    void display( struct student s)
    {
        printf("Roll No:%d\n",s.rollno);
        printf("Name:%s\n",s.name);
        printf("Year:%d\n",s.year);

    }
    void main()
    { struct student s;

        printf("Enter Roll no:");
        fflush(stdin);
        scanf("%d",&s.rollno);
        printf("Enter Name:");
        fflush(stdin);
        gets(s.name);
        printf("Enter Year:");
        fflush(stdin);
        scanf("%d",&s.year);
        printf("Result is:\n");
        display(s); //passing structure to function
    }

```

Output:

```

Enter Roll no:12
Enter Name:Monu
Enter Year:1987
Result is:
Roll No:12
Name:Monu
Year:1987

```

❖ Self-Referential Structure

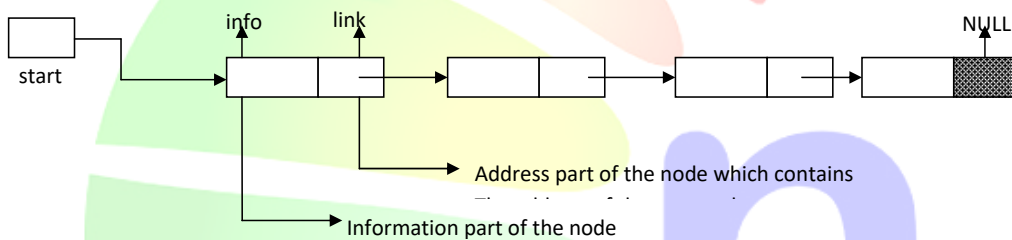
13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
 Compiled By: Sandeep Sappal
 Mob. : 9301123085, Ph. : 4062091
 97

Explain with an example the self-referential structure.

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

```
struct node {  
  
    int data;  
  
    struct node *link;  
  
};
```

In the above example, the listnode is a self-referential structure – because the *link is of the type struct node.



```
/* Program of single linked list*/  
#include <stdio.h>  
#include <malloc.h>  
struct node  
{  
    int info;  
    struct node *link;  
}*start;  
  
void create_list(int data)  
{  
    struct node *q,*tmp;  
    tmp= malloc(sizeof(struct node));  
    tmp->info=data;  
    tmp->link=NULL;
```

```

        if(start==NULL) /*If list is empty */
            start=tmp;
        else
        { /*Element inserted at the end */
            q=start;
            while(q->link!=NULL)
                q=q->link;
            q->link=tmp;
        }
    }
}
/*End of create_list()*/

```

```

void display()
{

```

```

    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;}

```

```

    q=start;
    printf("List is :\n");

```

```

while(q!=NULL)
{
    printf("%d ", q->info);
    q=q->link;
}
printf("\n");
}
/*End of display() */

```

```

void main()
{

```

```

    int choice,n,m,position,i;
    start=NULL;
    while(1)
    {
        printf("1.Create List\n");
        printf("2.Display\n");
        printf("3.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

```

```

switch(choice)
{
    case 1:
        printf("How many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("Enter the element : ");
            scanf("%d",&m);
            create_list(m);
        }
        break;

    case 2:
        display();
        break;

    case 3:
        exit();
    default:
        printf("Wrong choice\n");
}
/*End of switch */
/*End of while */
/*End of main()*/

```

Unions

Unions are a concept borrowed from structures and therefore follow the same syntax as structure. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```

union item
{
    int m;
    float x;
    char c;
}code;

```

In above Example Size of Union is 4 Byte Because in union memory location allocate according to largest member specified.



The C preprocessor

The C preprocessor is exactly what its name implies. It is a program that processes our source program before it is passed to the compiler. Preprocessor commands (often known as directives) form what can almost be considered a language within C language.

❖ Features of C Preprocessor

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:

- ✓ *Macro expansion*
- ✓ *File inclusion*

- ✓ Conditional Compilation
- ✓ Miscellaneous directives

Let us understand these features of preprocessor one by one.

❖ Macro Expansion

Have a look at the following program.

```
#define UPPER 25
main()
{
    int i;
    for ( i = 1 ; i <= UPPER ; i++ )
        printf ( "\n%d", i );
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main()** through the statement,

```
#define UPPER 25
```

This statement is called ‘macro definition’ or more commonly, just a ‘macro’. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

```
#define PI 3.1415
main()
{
    float r = 6.25 ;
    float area ;
    area = PI * r * r ;
    printf ( "\nArea of circle = %f", area ) ;
}
```

UPPER and PI in the above programs are often called ‘macro templates’, whereas, 25 and 3.1415 are called their corresponding ‘macro expansions’.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the #define directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces

the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

❖ Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )  
main()  
{  
float r1 = 6.25, r2 = 2.5, a ; a = AREA ( r1 ) ;  
printf ( "\nArea of circle = %f", a ) ;  
a = AREA ( r2 ) ;  
printf ( "\nArea of circle = %f", a ) ;  
}
```

Here's the output of the program...

Area of circle = 122.656250

Area of circle = 19.625000

❖ Macros versus Functions

In a macro call the preprocessor replaces the macro template with its macro expansion. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

This brings us to a question: when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.

Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

❖ File Inclusion

An external file containing function or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions this is achieved by the preprocessor directive.

#include "filename"

where filename is the name of the file containing the required definitions or functions At this point the preprocessor inserts the entire contents of filename into the source code of the program when the filename is included within the double quotation marks the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <FILENAME>

Without double quotation marks. in this case the file is searched only in the standard directories.

❖ Token Concatenation

Token concatenation, also called token pasting, is one of the most subtle — and easy to abuse — features of the C macro preprocessor. Two arguments can be 'glued' together using ## preprocessor operator; this allows two tokens to be concatenated in the preprocessed code.

#define MYCASE(item,id)

case id:

item##_##id = id;

break;

❖ Conditional Compilation

✓ **#ifndef statement : Conditional Compilation Directives (C Preprocessor)**

What it does ?

- These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression
- Block is Called as Conditional Group

Syntax :

```
#ifndef MACRONAME
Statement_block;
```



```
#endif
```

Explanation :

1. If the MACRONAME specified after `#ifndef` is not defined previously in `#define` then `statement_block` is followed otherwise it is skipped
2. We say that the conditional *succeeds* if *MACRO* is not defined, *fails* if it is not.

Live Example 1 :

```
#include "stdio.h"
void main()
{
    #ifndef NUM
        #define MAX 20 // Define another macro if MACRO NUM is defined
    #endif
    printf("MAX number is : %d", MAX);
}
```

Output :

```
MAX Number is 20
```

Live Example 2 :

```
#include "stdio.h"
#define MAX 90
void main()
{
    #ifndef MAX
        #define MIN 90
    #else
        #define MIN 100
    #endif
}
```

```
printf("MIN number : %d",MIN);  
}
```

Output :

```
MIN number : 100
```

Rules :

1. The MACRONAME inside of a conditional can include preprocessing directives.
2. They are executed only if the conditional succeeds.
3. You can nest conditional groups inside other conditional groups, but they must be completely nested.
4. You cannot start a conditional group in one file and end it in another.

✓ **#ifdef statement : Conditional Compilation Directives (C Preprocessor)**

What it does ?

- These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression
- Block is Called as Conditional Group

Syntax :

```
#ifdef MACRONAME  
    Statement_block;  
#endif
```

Explanation :

1. If the MACRONAME specified after #ifdef is defined previously in #define then statement_block is followed otherwise it is skipped
2. We say that the conditional *succeeds* if MACRO is defined, *fails* if it is not.

Live Example 1 :

```
#include "stdio.h"  
#define NUM 10  
void main()  
{
```

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091
106

```
#ifndef NUM
    #define MAX 20 // Define another macro if MACRO NUM is defined
#endif
printf("MAX number is : %d",MAX);
}
```

Output :

MAX Number is 20

Live Example 2 :

```
#include "stdio.h"
void main()
{
    #ifdef MAX #define MIN 90
    #else
        #define MIN 100
    #endif
    printf("MIN number : %d",MIN);
}
```

Output :

MIN number : 100

Rules :

1. The MACRONAME inside of a conditional can include preprocessing directives.
2. They are executed only if the conditional succeeds.
3. You can nest conditional groups inside other conditional groups, but they must be completely nested.
4. You cannot start a conditional group in one file and end it in another.

#if statement : Conditional Compilation Directives (C Preprocessor)

What it does ?

- These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression

Syntax :

```
#if Expression
Statement 1
Statement 2
.
.
Statement n
#endif
```

✓ **#else statement : Conditional Compilation Directives (C Preprocessor)**

What it does ?

- These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression

Syntax :

```
#if Expression 1
Statement_block1

#else Expression 2
Statement_block2
#endif
```

Explanation :

1. Expression allows only constant expression
2. Result of the Expression is TRUE , then Block of Statement between #if and #endif is followed
3. Result of the Expression is FALSE , then Block of Statement between #if and #endif is skipped
4. Result of the Expression is Non-Zero , then Block of Statement between #if and #endif is followed
5. Result of the Expression is Zero , then Block of Statement between #if and #endif is skipped

6. *#endif* is the end of *#if* statement

Live Example 1 :

```
#include "stdio.h"
#define NUM 10
void main()
{
    #if ((NUM%2)==0)
        printf("\nNumber is Even");
    #endif
}
```

Output :

Number is Even

Live Example 2 :

```
#include "stdio.h"
#define NUM 11
void main()
{
    #if ((NUM%2)==0)
        printf("\nNumber is Even");
    #else
        printf("\nNumber is Odd");
    #endif
}
```

Output :

Number is Odd

✓ **#elif statement : Conditional Compilation Directives (C Preprocessor)**

What it does ?

- These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression

Syntax :

```
#if Expression1
    Statement_block 1;

#elif Expression2
    Statement_block 2;
#elif Expression3
    Statement_block 3;
#else
    Statement_block 4;
#endif
```

Explanation :

1. Expression allows only constant expression
2. #elif directive means "else if"
3. #elif Establishes an if-else-if chain for multiple compilation options.
4. Result of the Expression is TRUE , then Block of Statement between #if and first #elif is compiled , then it jumps to #endif.
5. Result of the Expression is FALSE , then Corresponding #elif condition is tested , if true the the block followed by that elif is Compiled otherwise it checks for Next condition followed by next elif statement
6. #endif is the end of #if statement

Live Example 1 :

```
#include "stdio.h"
#define NUM 10
void main()
{
    #if (NUM == 0)
        printf("\nNumber is Zero");
    }
```

```
#elif (NUM > 0)
    printf("\nNumber is Positive");
#else
    printf("\nNumber is Negative");
#endif
}
```

Output :

Number is Positive

Live Example 2 :

```
#include "stdio.h"
#define MARKS 71
void main()
{
    #if (MARKS >= 70)
        printf("\nDistinction");
    #elif ((MARKS >= 60) && (MARKS < 70))
        printf("\nFirst Class");
    #elif ((MARKS >= 40) && (MARKS < 60))
        printf("\nSecond Class");
    #else
        printf("\nFail");
    #endif
}
```

Output :

Distinction

Miscellaneous Directives

There are two more preprocessor directives available, though they are not very commonly used. They are:

- ✓ #undef
- ✓ #pragma

❖ *#undef Directive*

On some occasions it may be desirable to cause a defined name to become 'undefined'. This can be accomplished by means of the **#undef** directive. In order to undefine a macro that has been earlier **#defined**, the directive, **#undef** macro template can be used. Thus the statement, **#undef PENTIUM** would cause the definition of PENTIUM to be removed from the system. All subsequent **#ifdef PENTIUM** statements would evaluate to false.

❖ **#pragma Directive**

This directive is another special-purpose directive that you can use to turn on or off certain features. Pragas vary from one compiler to another. There are certain pragmas available with ANSI C & Microsoft C compiler that deal with formatting source listings and placing comments in the object file generated by the compiler. Turbo C/C++ compiler has got a pragma that allows you to suppress warnings generated by the compiler. Some of these pragmas are discussed below.

(a) **#pragma startup** and **#pragma exit**: These directives allow us to specify functions that are called upon program startup (before **main()**) or program exit (just before the program terminates). Their usage is as follows:

```
void fun1();
void fun2();
#pragma startup fun1
#pragma exit fun2
main()
{ printf( "\nInside main" );
}
void fun1()
{ printf( "\nInside fun1" );
}
void fun2()
{ printf( "\nInside fun2" );
}
```

And here is the output of the program.

*Inside fun1
Inside main
Inside fun2*

Note that the functions **fun1()** and **fun2()** should neither receive nor return any value. If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called.

- (b) **#pragma warn:** This directive tells the compiler whether or not we want to suppress a specific warning. Usage of this pragma is shown below.

```
#pragma warn -rvl /* return value */
#pragma warn -par /* parameter not used */
#pragma warn -rch /* unreachable code */
int f1()
{
    int a = 5 ;
}
void f2 ( int x )
{
    printf ( "\nInside f2" );
}
int f3()
{
    int x = 6 ;
    return x ;
    x++ ;
}
void main()
{
    f1() ;
    f2 ( 7 ) ;
    f3() ;
}
```

If you go through the program you can notice three problems immediately. These are:

- (a) Though promised, **f1()** doesn't return a value.
- (b) The parameter **x** that is passed to **f2()** is not being used anywhere in **f2()**.
- (c) The control can never reach **x++** in **f3()**.

If we compile the program we should expect warnings indicating the above problems. However, this does not happen since we have suppressed the warnings using the **#pragma** directives. If we replace the '-' sign with a '+' then these warnings would be flashed on compilation. Though it is a bad practice to suppress warnings, at times it becomes useful to suppress them. For example, if you have written a huge program and are trying to compile it, then to begin with you are more interested in locating the errors, rather than the warnings. At such times you may suppress the warnings. Once you have located all errors, then you may turn on the warnings and sort them out.

Predefined Pre-Processor Names

❖ **The following names are predefined within the pre-processor:**

`__LINE__`

The current source file line number, a decimal integer constant.

`__FILE__`

The 'name' of the current source code file, a string literal.

`__DATE__`

The current date, a string literal. The form is

Apr 21 1990

where the month name is as defined in the library function `asctime` and the first digit of the date is a space if the date is less than 10.

`__TIME__`

The time of the translation; again a string literal in the form produced by `asctime`, which has the form "hh:mm:ss".

`__STDC__`

The integer constant 1. This is used to test if the compiler is Standard-conforming, the intention being that it will have different values for different releases of the Standard.

A common way of using these predefined names is the following:

```
#define TEST(x) if(!(x))\
    printf("test failed, line %d file %s\n",\
        __LINE__, __FILE__)
```

File Handling

Use to Store Data Permanently in Secondary Storage Device(disk)

Memory is volatile and its contents would be lost once the program is terminated. So if we need the same data again it would have to be either entered through the keyboard again or would have to be regenerated programmatically. Obviously both these operations would be tedious. At such times it becomes necessary to store the

data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a 'file' on the disk.

❖ File Operations

There are different operations that can be carried out on a file. These are:

- ✓ Creation of a new file
- ✓ Opening an existing file
- ✓ Reading from a file
- ✓ Writing to a file
- ✓ Moving to a specific location in a file (seeking)
- ✓ Closing a file

❖ File operation functions in C:

Function Name	Operation
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the beginning of the file

❖ File Pointer

Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer**.

Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable.

You declare these file pointer variables as follows:

```
FILE *fopen(), *fp1, *fp2, *fp3;
```

fopen() Function

Use to open a File in various modes

Syntax

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

Open Specified file in Specified mode and returns a FILE pointer on success or NULL pointer on failure.

❖ File Opening Modes

There several modes in which we can open a file. Following is a list of all possible modes in which a file can be opened. The tasks performed by **fopen()** when a file is opened in each of these modes are also mentioned.

"r" Searches file. If the file is opened successfully **fopen()** loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened **fopen()** returns NULL.

Operations possible –
reading from the file.

"w" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible –
writing to the file.

"a" Searches file. If the file is opened successfully **fopen()** loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to

open file.

Operations possible -
adding new contents at the
end of file.

"r+"

Searches file. If is opened
successfully **fopen()** loads
it into memory and sets up
a pointer which points to
the first character in it.
Returns NULL, if unable to
open the file.

❖ Closing the File

When we have finished reading and writing from the file, we need to close it. This is done using the function **fclose()** through the statement,

```
fclose ( fp );
```

❖ fgetc() function

<stdio.h>

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

Get character from stream

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced by one character to point to the next character.

fgetc and getc are equivalent, except that the latter one may be implemented as a macro.

Parameters

stream

Pointer to a FILE object that identifies the stream on which the operation is to be performed.

Return Value

The character read is returned as an int value.

If the End-of-File is reached or a reading error happens, the function returns EOF and the corresponding error or eof indicator is set. You can use either ferror or feof to determine whether an error happened or the End-Of-File was reached.

Example

```

/* fgetc example: money counter */
#include <stdio.h>
int main ()
{
    FILE * pFile;
    int c;
    int n = 0;
    pFile=fopen ("myfile.txt","r");
    if (pFile==NULL) perror ("Error opening file");
    else
    {
        do {
            c = fgetc (pFile);
            if (c == '$') n++;
        } while (c != EOF);
        fclose (pFile);
        printf ("The file contains %d dollar sign characters ($).\n",n);
    }
    return 0;
}

```

This program reads an existing file called myfile.txt character by character and uses the n variable to count how many dollar characters (\$) does the file contain.

❖ fputc() function

<stdio.h>

int fputc (int character, FILE * stream);

Write character to stream

Writes a character to the *stream* and advances the position indicator.

The character is written at the current position of the *stream* as indicated by the internal position indicator, which is then advanced one character.

Parameters

character

Character to be written. The character is passed as its int promotion.

stream

Pointer to a FILE object that identifies the stream where the character is to be written.

Return Value

If there are no errors, the same character that has been written is returned.

If an error occurs, EOF is returned and the error indicator is set (see ferror).

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

Example

```
/* fputc example: alphabet writer */  
#include <stdio.h>
```

```
int main ()  
{  
    FILE * pFile;  
    char c;  
  
    pFile = fopen ("alphabet.txt","w");  
    if (pFile!=NULL)  
    {  
        for (c = 'A' ; c <= 'Z' ; c++)  
        {  
            fputc ( (int) c , pFile );  
        }  
        fclose (pFile);  
    }  
    return 0;  
}
```

output:

*This program creates a file called alphabet.txt and writes
ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.*

❖ fgets() function

<stdio.h>

```
char * fgets ( char * str, int num, FILE * stream );
```

Get string from stream

Reads characters from *stream* and stores them as a C string into *str* until (*num*-1) characters have been read or either a newline or a the End-of-File is reached, whichever comes first.

A newline character makes fgets stop reading, but it is considered a valid character and therefore it is included in the string copied to *str*.

A null character is automatically appended in *str* after the characters read to signal the end of the C string.

Parameters

str

Pointer to an array of chars where the string read is stored.

num

Maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as *str* is used.

stream

Pointer to a FILE object that identifies the stream where characters are read from.

To read from the standard input, `stdin` can be used for this parameter.

Return Value

On success, the function returns the same *str* parameter.

If the End-of-File is encountered and no characters have been read, the contents of *str* remain unchanged and a null pointer is returned.

If an error occurs, a null pointer is returned.

Use either ferror or feof to check whether an error happened or the End-of-File was reached.

Example

```
/* fgets example */
#include <stdio.h>
void main()
{ FILE * pFile;
  char mystring [100];
  pFile = fopen ("myfile.txt" , "r");
  if (pFile == NULL) perror ("Error opening file");
  else {
    fgets (mystring , 100 , pFile);
    puts (mystring);
    fclose (pFile);  }
```

This example reads the first line of myfile.txt or the first 100 characters, whichever comes first, and prints them on the screen.

❖ fputs() function

<stdio.h>

```
int fputs ( const char * str, FILE * stream );
```


Write string to stream

Writes the string pointed by *str* to the *stream*.

The function begins copying from the address specified (*str*) until it reaches the terminating null character ('\0'). This final null-character is not copied to the stream.

Parameters

str

An array containing the null-terminated sequence of characters to be written.

stream

Pointer to a FILE object that identifies the stream where the string is to be written.

Return Value

On success, a non-negative value is returned.

On error, the function returns EOF.

Example

```
/* fputs example */  
#include <stdio.h>
```

```
int main ()  
{  
    FILE * pFile;  
    char sentence [256];  
    printf ("Enter sentence to append: ");  
    fgets (sentence,255,stdin);  
    pFile = fopen ("mylog.txt","a");  
    fputs (sentence,pFile);  
    fclose (pFile);  
    return 0;  
}
```

This program allows to append a line to a file called mylog.txt each time it is run.

❖ fread() function

<stdio.h>

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);

Read block of data from stream

Reads an array of *count* elements, each one with a size of *size* bytes, from the *stream* and stores them in the block of memory specified by *ptr*.

The position indicator of the stream is advanced by the total amount of bytes read.
The total amount of bytes read if successful is (*size * count*).

Parameters

ptr

Pointer to a block of memory with a minimum size of (*size*count*) bytes.

size

Size in bytes of each element to be read.

count

Number of elements, each one with a size of *size* bytes.

stream

Pointer to a FILE object that specifies an input stream.

Return Value

The total number of elements successfully read is returned as a size_t object, which is an integral data type.

If this number differs from the *count* parameter, either an error occurred or the End Of File was reached.

You can use either ferror or feof to check whether an error happened or the End-of-File was reached.

❖ fwrite() function

<stdio.h>

```
size_t fwrite ( const void * ptr, size_t size, size_t count, FILE * stream );
```

Write block of data to stream

Writes an array of *count* elements, each one with a size of *size* bytes, from the block of memory pointed by *ptr* to the current position in the *stream*.

The position indicator of the stream is advanced by the total number of bytes written.

The total amount of bytes written is (*size * count*).

Parameters

ptr

Pointer to the array of elements to be written.

size

Size in bytes of each element to be written.

count

Number of elements, each one with a size of *size* bytes.

stream

Pointer to a FILE object that specifies an output stream.

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

Return Value

The total number of elements successfully written is returned as a size_t object, which is an integral data type.

If this number differs from the *count* parameter, it indicates an error.

❖ fprintf() Function

```
int fprintf ( FILE * stream, const char * format, ... );
```

Write formatted output to stream

Writes to the specified *stream* a sequence of data formatted as the *format* argument specifies. After the *format* parameter, the function expects at least as many additional arguments as specified in *format*.

Parameters

stream

Pointer to a FILE object that identifies the stream.

format

C string that contains the text to be written to the stream.

Example

```
/* fprintf example */
#include <stdio.h>
void main ()
{
    FILE * pFile;
    int n;
    char name [100];
    pFile = fopen ("myfile.txt","w");
    for (n=0 ; n<3 ; n++)
    {   puts ("please, enter a name: ");
        gets (name);
        fprintf (pFile, "Name %d [%-10.10s]\n",n,name);
    }
    fclose (pFile);}
```

This example prompts 3 times the user for a name and then writes them to myfile.txt each one in a line with a fixed length (a total of 19 characters + newline).

Two format tags are used:

%d : Signed decimal integer

%-10.10s : left aligned (-), minimum of ten characters (10), maximum of ten characters (.10), String (s) .

Assuming that we have entered John, Jean-Francois and Yoko as the 3 names, myfile.txt would contain:

myfile.txt

Name 1 [John]
Name 2 [Jean-Franc]
Name 3 [Yoko]

❖ fscanf() Function

<stdio.h>

int fscanf (FILE * stream, const char * format, ...);

Read formatted data from stream

Reads data from the *stream* and stores them according to the parameter *format* into the locations pointed by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format tag within the *format* string.

Example

```
/* fscanf example */  
#include <stdio.h>
```

```
int main ()  
{  
    char str [80];  
    float f;  
    FILE * pFile;  
  
    pFile = fopen ("myfile.txt", "w+");  
    fprintf (pFile, "%f %s", 3.1416, "PI");  
    rewind (pFile);  
    fscanf (pFile, "%f", &f);  
    fscanf (pFile, "%s", str);  
    fclose (pFile);  
    printf ("I have read: %f and %s \n", f, str);  
    return 0;  
}
```

❖ fseek() function

<stdio.h>

int fseek (FILE * stream, long int offset, int origin);

Reposition stream position indicator

Sets the position indicator associated with the *stream* to a new position defined by adding *offset* to a reference position specified by *origin*.

The End-of-File internal indicator of the stream is cleared after a call to this function, and all effects from previous calls to ungetc are dropped.

When using `fseek` on text files with *offset* values other than zero or values retrieved with ftell, bear in mind that on some platforms some format transformations occur with text files which can lead to unexpected repositioning.

On streams open for update (read+write), a call to `fseek` allows to switch between reading and writing.

Parameters

stream

Pointer to a FILE object that identifies the stream.

offset

Number of bytes to offset from *origin*.

origin

Position from where *offset* is added. It is specified by one of the following constants defined in `<stdio.h>`:

SEEK_SET Beginning of file or 0

SEEK_CUR Current position of the file pointer or 1

SEEK_END End of file or -1

Return Value

If successful, the function returns a zero value.

Otherwise, it returns nonzero value.

Example

```
/* fseek example */
#include <stdio.h>
```

```
int main ()
{
    FILE * pFile;
    pFile = fopen ( "example.txt" , "w" );
    fputs ( "This is an apple." , pFile );
    fseek ( pFile , 9 , SEEK_SET );
    fputs ( " sam" , pFile );
    fclose ( pFile );
    return 0;
}
```

After this code is successfully executed, the file example.txt contains:
This is a sample.

❖ **ftell() function**

<stdio.h>

```
long int ftell ( FILE * stream );
```

Get current position in stream

Returns the current value of the position indicator of the *stream*.

For binary streams, the value returned corresponds to the number of bytes from the beginning of the file.

For text streams, the value is **not** guaranteed to be the exact number of bytes from the beginning of the file, but the value returned can still be used to restore the position indicator to this position using fseek.

Parameters

stream

Pointer to a FILE object that identifies the stream.

Return Value

On success, the current value of the position indicator is returned.

If an error occurs, -1L is returned, and the global variable `errno` is set to a positive value. This value can be interpreted by perror.

```
/* ftell example : getting size of a file */  
#include <stdio.h>
```

```
int main ()  
{  
    FILE * pFile;  
    long size;  
  
    pFile = fopen ("myfile.txt","rb");  
    if (pFile==NULL) perror ("Error opening file");  
    else  
    {  
        fseek (pFile, 0, SEEK_END);  
        size=ftell (pFile);  
        fclose (pFile);  
        printf ("Size of myfile.txt: %ld bytes.\n",size);  
    }  
}
```

```
return 0;
}
```

This program prints out the size of myfile.txt in bytes.

❖ rewind() function

<stdio.h>

```
void rewind ( FILE * stream );
```

Set position indicator to the beginning

Sets the position indicator associated with *stream* to the beginning of the file.

A call to rewind is equivalent to:

```
fseek ( stream , 0L , SEEK_SET );
```

except that, unlike fseek, rewind clears the error indicator.

On streams open for update (read+write), a call to rewind allows to switch between reading and writing.

Parameters

stream

Pointer to a FILE object that identifies the stream.

Return Value

none

```
/* rewind example */
```

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

```
int main ()
```

```
{
```

```
    int n;
```

```
    FILE * pFile;
```

```
    char buffer [27];
```

```
    pFile = fopen ("myfile.txt","w+");
```

```
    for ( n='A' ; n<='Z' ; n++)
```

```
        fputc ( n, pFile);
```

```
    rewind (pFile);
```

```
    fread (buffer,1,26,pFile);
```

```
    fclose (pFile);
```

```
    buffer[26]='\0';
```

```
    puts (buffer);
```

```
    return 0;
```

```
}
```

A file called *myfile.txt* is created for reading and writing and filled with the alphabet. The file is then rewinded, read and its content is stored in a buffer, that then is written to the standard output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

❖ putw () function

syntax

```
int putw ( int integer , FILE * stream );  
<stdio.h>
```

Write an integer to a stream.

Writes a binary *integer* of type **int** to the *stream* and increases its position indicator to point to the next character.

putw assumes no alignment in the *stream*.

Notice that the length of **int** type depends on systems and compilers so this may cause some porting problems.

Parameters.

integer

integer to be written.

stream

Pointer to an open file.

Return Value.

On success the integer value written is returned.

On error EOF is returned

Example.

```
/* putw/getw example */  
#include <stdio.h>
```

```
main()  
{  
    FILE * pFile;  
    int i;  
    pFile = fopen ("myfile.bin","wb+");  
    putw (20,pFile);  
    rewind (pFile);  
    i=getw (pFile);  
    printf ("The double of %d is %d.\n", i, i*2);  
    return 0;
```



```
}
```

This example creates a binary file called myfile.bin and stores a numeric value on it. Then this value is retrieved and its double is printed. This is done to emphasize that the type used is of type int and not a string.

❖ getw () function

```
int getw ( FILE * stream );
```

stdio.h

Get the next int value from a stream.

Returns the next value of type **int** from the *stream* and increases the position indicator of the *stream* to point the next character.

getw assumes no alignment in the *stream* and is not intended for use with files opened in text mode.

Notice that the length of **int** type depends on systems and compilers so this may cause some porting problems.

Parameters.

stream

Pointer to an open file.

Return Value.

On success the integer read is returned.

If the End Of File is reached or there has been an error reading, the function returns an EOF character.

Because **EOF** is a valid integer, use feof() or ferror() to verify if an error has occurred or the End-Of-File has been reached.

Example.

```
/* putw/getw example */
```

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

```
main()
```

```
{
```

```
FILE * pFile;
```

```
int i;
```

```
pFile = fopen ("myfile.bin","wb+");
```

```
putw (20,pFile);
```

```
rewind (pFile);
```

```
i=getw (pFile);
```

```
printf ("The double of %d is %d.\n", i, i*2);
```

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl

Compiled By: Sandeep Sappal

Mob. : 9301123085, Ph. : 4062091

```

return 0;
}

```

This example creates a binary file called myfile.bin and stores a numeric value on it. Then this value is retrieved and its double is printed. This is done to emphasize that the type used is of type int and not a string.

*/*Example program for using getw and putw functions*/*

```

#include< stdio.h >
main()
{
FILE *f1,*f2,*f3;
int number I;
printf("Contents of the data file\n");
f1=fopen("DATA","w");
for(I=1;I< 30;I++)
{
scanf("%d",&number);
if(number==-1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)/* Read from data file*/
{
if(number%2==0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("\nContents of the odd file\n");
while(number=getw(f2))!=EOF)
printf("%d%d",number);
printf("\nContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
}

```

```
fclose(f2);  
fclose(f3);  
}
```



Enumerated Data Types:

Its a user define data types. it defines as follows:

Syntax

enum identifier {value1, value2, valuen};

the identifier is user define enumerated data type which can be use to declare variables of that type

enum identifier v1,v2,...vn;

The compiler automatically assign integer beginning with 0 , it assign 0 to value1, 1 to value 2 and so on. the value can also override by assigning value to enumerated constants.

ex.

enum day{monday, tuesday, wednesday};

by default compiler give 0 to monday , 1 to tuesday , 2 to wednesday.

But one can override the value also

for example

enum day{monday=1, tuesday, wednesday};

now compiler give 1 to monday , 2 to tuesday , 3 to wednesday.

*/*Enum example in C*/*

#include <stdio.h>

int main()

{

enum Days{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};

Days TheDay;

int j = 0;

printf("Please enter the day of the week (0 to 6)\n");

scanf("%d",&j);

TheDay = Days(j);

if(TheDay == Sunday || TheDay == Saturday)

printf("Hurray it is the weekend\n");

else

printf("Curses still at work\n");

return 0;

}

Typedef Declarations

A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. You can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that

you have declared. Typedef names allow you to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

The following statements declare LENGTH as a synonym for **int** and then use this **typedef** to declare length, width, and height as integer variables:

```
typedef int LENGTH;
```

```
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, **typedef** can be used to define a class type (structure, union, or C++ class).

For example:

```
typedef struct {  
    int scruples;  
    int drams;  
    int grains;  
    } WEIGHT;
```

The structure WEIGHT can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

Command Line Arguments

What is a command line argument ? It is a parameter supplied to a program when the program is invoked . This parameter may represent a filename the program

should process. PROGRAM - Main can take two arguments called argc and argv and the information contained in the command line is passed on to the program through these arguments, when main is called up by the system. The variable argc is an argument counter that counts the number of argument on the command line. The argv is an argument vector and represent an array of character pointers that point to the command line argument. The size of this array will be equal to the value of argc.

Example - 1

```
#include <stdio.h>

main( int argc, char *argv[] )
{
    if( argc == 2 )
        printf("The argument supplied is %s\n", argv[1]);
    else if( argc > 2 )
        printf("Too many arguments supplied.\n");
    else
        printf("One argument expected.\n");
}
```

*Note that *argv[0] is the name of the program invoked, which means that *argv[1] is a pointer to the first argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one. Thus for n arguments, argc will be equal to n + 1. The program is called by the command line,*

```
myprog argument1
```

Example - 2

```
//will print all arguments supply on command prompt
#include <stdio.h>
main(int argc, char *argv[])
{
    int i;
    for(i = 0; i < argc; i++)
        printf("arg %d: %s\n", i, argv[i]);
    return 0;
}
```

Math Functions (Header File math.h)

➤ abs

Syntax:

```
#include <cstdlib>
int abs( int num );
```

The abs() function returns the absolute value of *num*

➤ **atan**

Syntax:

```
#include <cmath>
double atan( double arg );
```

The function atan() returns the arc tangent of *arg*, which will be in the range $[-\pi/2, +\pi/2]$.

Related topics:

➤ **ceil**

Syntax:

```
#include <cmath>
double ceil( double num );
```

The ceil() function returns the smallest integer no less than *num*. For example,

```
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

➤ **cos**

Syntax:

```
#include <cmath>
double cos( double arg );
```

The cos() function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of cos() is in the range [-1,1]. If *arg* is infinite, cos() will return NAN and raise a floating-point exception.

➤ exp

Syntax:

```
#include <cmath>
double exp( double arg );
```

The exp() function returns e (2.7182818) raised to the *arg*th power.

➤ fabs

Syntax:

```
#include <cmath>
double fabs( double arg );
```

The function fabs() returns the absolute value of *arg*.

Related topics:

➤ floor

Syntax:

```
#include <cmath>
double floor( double arg );
```

The function floor() returns the largest integer not greater than *arg*. For example,

```
y = 6.04;
x = floor( y );
```

would result in x being set to 6.0.

➤ fmod

Syntax:


```
#include <cmath>
double fmod( double x, double y );
```

The fmod() function returns the remainder of x/y .

➤ labs

Syntax:

```
#include <stdlib>
long labs( long num );
```

The function labs() returns the absolute value of *num*.

Related topics:

➤ log

Syntax:

```
#include <cmath>
double log( double num );
```

The function log() returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of x to an arbitrary base b , you can use:

```
double answer = log(x) / log(b);
```

➤ log10

Syntax:

```
#include <cmath>
double log10( double num );
```

The log10() function returns the base 10 (or common) logarithm for *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

➤ pow

Syntax:

```
#include <cmath>
double pow( double base, double exp );
```

The `pow()` function returns *base* raised to the *exp*th power. There's a domain error if *base* is zero and *exp* is less than or equal to zero. There's also a domain error if *base* is negative and *exp* is not an integer. There's a range error if an overflow occurs.

Related topics:

➤ **sin**

Syntax:

```
#include <cmath>
double sin( double arg );
```

The function `sin()` returns the sine of *arg*, where *arg* is given in radians. The return value of `sin()` will be in the range $[-1,1]$. If *arg* is infinite, `sin()` will return NAN and raise a floating-point exception.

➤ **sqrt**

Syntax:

```
#include <cmath>
double sqrt( double num );
```

The `sqrt()` function returns the square root of *num*. If *num* is negative, a domain error occurs.

➤ **tan**

Syntax:

```
#include <cmath>
double tan( double arg );
```

The `tan()` function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, `tan()` will return NAN and raise a floating-point exception.

Some Useful C Character Functions(ctype.h)

<u>Function</u>	<u>Return non-zero if true else zero</u>
int isalpha(c);	c is a letter.
int isupper(c);	c is an upper case letter.
int islower(c);	c is a lower case letter.
int isdigit(c);	c is a digit [0-9].
int isalnum(c);	c is an alphanumeric character (c is a letter or a digit);
int isspace(c);	c is a SPACE, TAB, RETURN, NEWLINE, FORMFEED, or vertical tab character.
int ispunct(c);	c is a punctuation character (neither control nor alphanumeric).

int toupper(int c); convert character c to upper case (leave it alone if not lower)

int tolower(int c); convert character c to lower case (leave it alone if not upper)

Some Useful Library Functions(stdlib.h)

➤ **Usage of atoi():**

int atoi (const char * str);

Parameters:

C string str interpreting its content as a integer. White-spaces are discarded until the first non-whitespace character is found.

Return value:

The function returns the converted integral number as an int value, if successful. If no valid conversion could be performed then an zero value is returned. If the value is out of range then INT_MAX or INT_MIN is returned.

Source code example of atoi():

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

    char buffer [256];
    printf ("Enter a number: ");
    gets (buffer);
    i = atoi (buffer);
    printf ("The value entered is %d.",i);
    return 0;}
```

Output of the atoi example program above:

Enter a number:12

The value entered is 12.

➤ **Usage of atof():**

double atof (const char * str);

Parameters:

C string str interpreting its content as a floating point number. White-spaces are discarded until the first non-whitespace character is found. A valid floating point number for atof is formed by: plus or minus sign, sequence of digits, decimal point and optional exponent part (character 'e' or 'E')

Return value:

The function returns the converted floating point number as a double value, if successful.

If no valid conversion could be performed then an zero value is returned.

Source code example of atof():

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{double a,b;
    char buffer [256];
    printf ( "Input: " );
    gets (buffer);
    a = atof (buffer);
    b = a/2;
    printf ( "a= %f and b= %f\n" , a, b );
    return 0;
}
```

Output of the atof example program above:

```
Input: 12
a= 12.000000 and b= 6.000000
```

➤ **Usage of exit():**

void exit (int status);

On call the process will terminate normally. It will perform regular cleanup as normal for a normal ending process.

Parameters:

A status value returned to the parent process.

Return value:

The argument status is returned to the host environment.
Normally you say 1 or higher if something went wrong and 0 if everything went ok.
For example exit(0) or exit (1).

Source code example of exit():

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{int * buffer;

    /*get a initial memory block*/
    buffer = (int*) malloc (10*sizeof(int));
    if (buffer==NULL)
    {
        printf("Error allocating memory!");
        exit (1);
    }

    free (buffer);
    return 0;
}
```

➤ Usage of atol():

long int atol (const char * str);

Parameters:

C string str interpreting its content as a integer. White-spaces are discarded until the first non-whitespace character is found.

Return value:

The function returns the converted integral number as an int value, if successful and returns it as a long int. If no valid conversion could be performed then an zero value is returned.

If the value is out of range then LONG_MAX or LONG_MIN is returned.

//Source code example of atol():

13, Gulabchand ki Bagichi, Behind Jhaver Estate, Thatipur Gwl
Compiled By: Sandeep Sappal
Mob. : 9301123085, Ph. : 4062091
141

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{long int long_int;
  char buffer [256];
  printf ("Enter a long number: ");
  gets ( buffer );

  long_int = atol (buffer);
  printf ("The value entered is %d.\n",long_int);

  return 0;
}
```

Output of the atol example program above:

```
Enter a long number: 256000
The value entered is 256000.
```