Chapter 1

Fundamentals and History of C

Introduction to C

C has emerged as the most widely used programming language for software development. C language supports the powerful low level features like pointer, memory allocation, bit manipulations etc. The features of C language make it possible to se the language for system programming like the development of compiler, interpreter, operating system, system utilities etc.

History of C:

C language was developed by Dennis Ritchie & Ken Thompson at Bell Laboratories (now part of AT & T) at USA. In 1968, Bell and MIT were doing a joint project on MULTICS (multiplexing information and computing service). Operating system for multi-user-time sharing- system. They are BCPL (Basic combined programming language) developed by Martin Richards at Cambridge University. Around the same time

The language called 'B' was written by Ken Thompson at Bell Laboratories. Denis Ritchie inherited the features of 'B' and BCPL, added some features of his own and developed 'c'.

Computer Definition:

A computer is a programmable machine. It allows the user to store all sorts of information and then 'process' that information, or data, or carry out actions with the information, such as calculating numbers or organizing words.

ADVANTAGES/FEATURES OF C

C language has become the language of choice of two decades among system programmers and application programmers. Reason for its popularity can be summarized as follows:

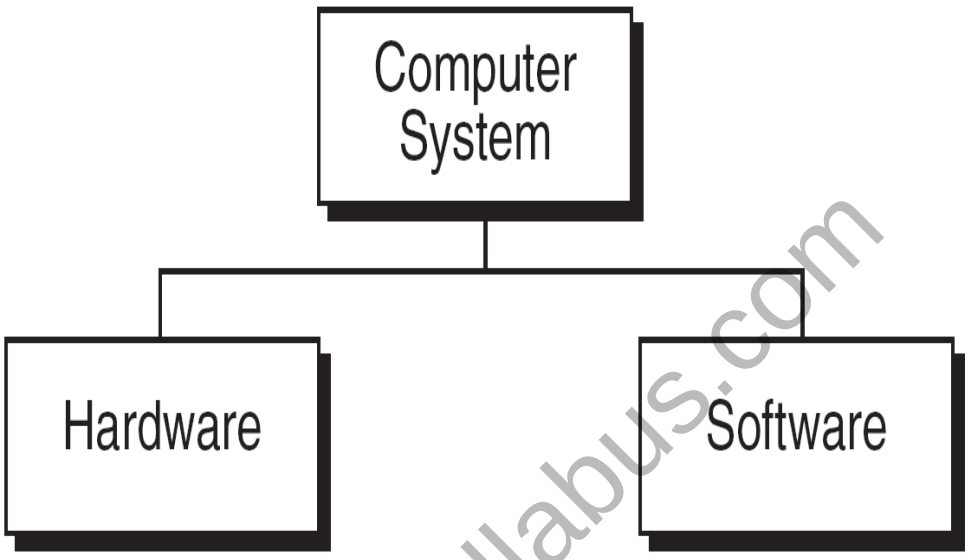
1. **Powerful and flexibility:**
   The power and popular UNIX as was written in C. The complier and interpreter for FORTAN, PASCAL, LISP, and BASIC are written in C.
2. **Portability:**
   C program written in one system can be run on other system with little modifications.
3. **Efficiency:**
   The program written in C language is highly efficient like assembly language in speed and memory management.
4. **Programmer oriented:**
   It has the flexible control structure and gives access to hardware and enables to manipulate individual bits of memory.

5. **Modularity:**

C program can be modularizing for step wise refinement. The complex program can be modularize into simple programs.

Computer Systems:

A computer is a system made of two major components :Hardware and Software



Hardware:

- The hardware are the parts of computer itself including the Central Processing Unit (CPU) and related microchips and micro-circuitry, keyboards, monitors, case and drives (floppy, hard, CD, DVD, optical, tape, etc...).
- Other extra parts called peripheral components or devices include mouse, printers, modems, scanners, digital cameras and cards (sound, colour, video) etc... Together they are often referred to as a personal computers or PCs.
- Central Processing Unit (CPU) - Though the term relates to a specific chip or the processor a CPU's performance is determined by the the rest of the computers circuitry and chips.
- Keyboard - The keyboard is used to type information into the computer or input information.
- Disk Drives - All disks need a drive to get information off - or read - and put information on the disk - or write. Each drive is designed for a specific type of disk whether it is a CD, DVD, hard disk or floppy. Often the term 'disk' and 'drive' are used to describe the same thing but it helps to understand that the disk is the storage device which contains computer files - or software - and the drive is the mechanism that runs the disk.
- Mouse - Most modern computers today are run using a mouse controlled pointer. Generally if the mouse has two buttons the left one is used to select objects and

text and the right one is used to access menus. If the mouse has one button (Mac for instance) it controls all the activity and a mouse with a third buttons can be used by specific software programs.

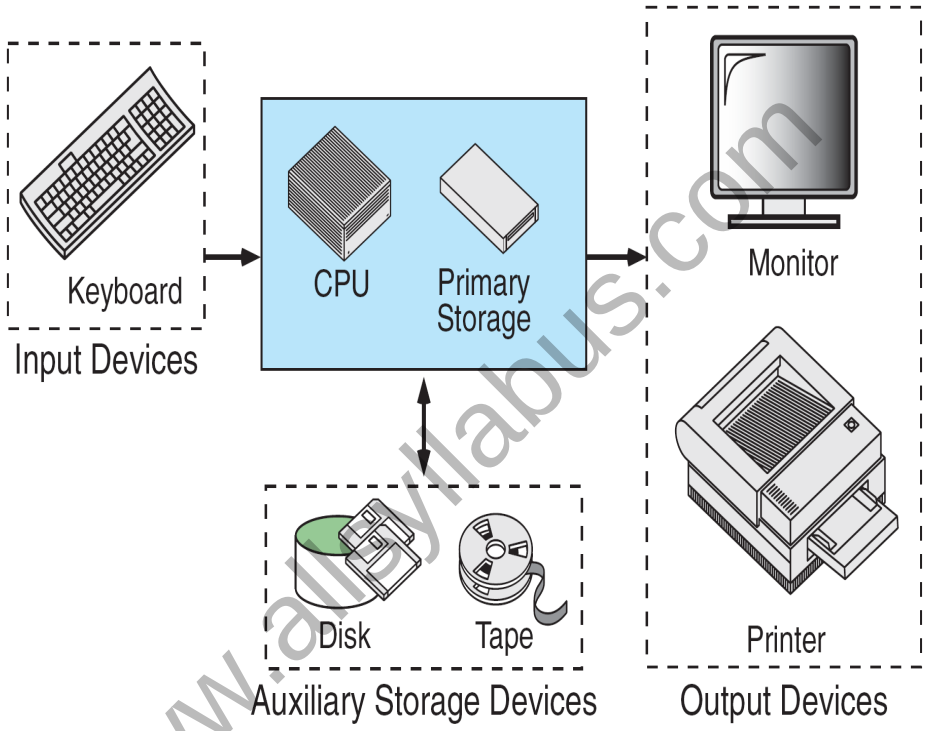
One type of mouse has a round ball under the bottom of the mouse that rolls and turns two wheels which control the direction of the pointer on the screen. Another type of mouse uses an optical system to track the movement of the mouse.

- Monitors - This Visual Display Unit (VDU) shows information on the screen when you type. This is called outputting information. When the computer needs more information it will display a message on the screen, usually through a dialog box. Monitors come in many types and plus sizes from the simple monochrome (one colour) screen to full colour screens.

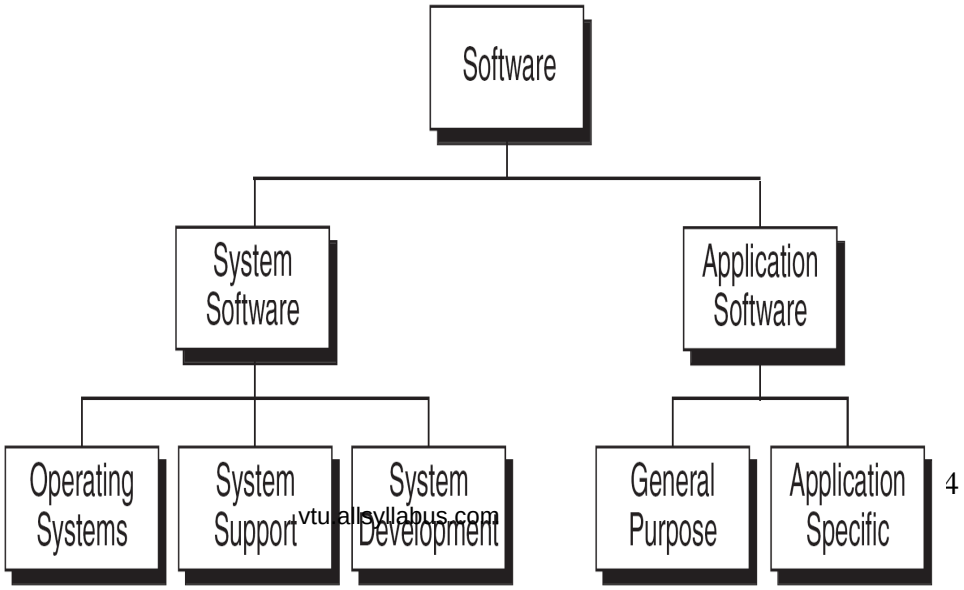
Most desktop computers use a monitor with a cathode tube and most notebooks use a liquid crystal display (LCD) monitor. To get the full benefit of today's software with full colour graphics and animation, computers need a color monitor with a display or graphics card.

- Printers - The printer takes the information on your screen and transfers it to paper or a hard copy. There are many different types of printers with various levels of quality. The three basic types of printer are; dot matrix, inkjet, and laser.
- Scanners- Scanners allow you to transfer pictures and photographs to your computer. A scanner 'scans' the image from the top to the bottom, one line at a time and transfers it to the computer as a series of bits or a bitmap. You can then take that image and use it in a paint program, send it out as a fax or print it.
- Memory - Memory can be very confusing but is usually one of the easiest pieces of hardware to add to your computer. It is common to confuse chip memory with disk storage. An example of the difference between memory and storage would be the difference between a table where the actual work is done (memory) and and a filing cabinet where the finished product is stored (disk). To add a bit more confusion, the computer's hard disk can be used as temporary memory when the program needs more than the chips can provide.
- Random Access Memory or RAM is the memory that the computer uses to temporarily store the information as it is being processed. The more information being processed the more RAM the computer needs.

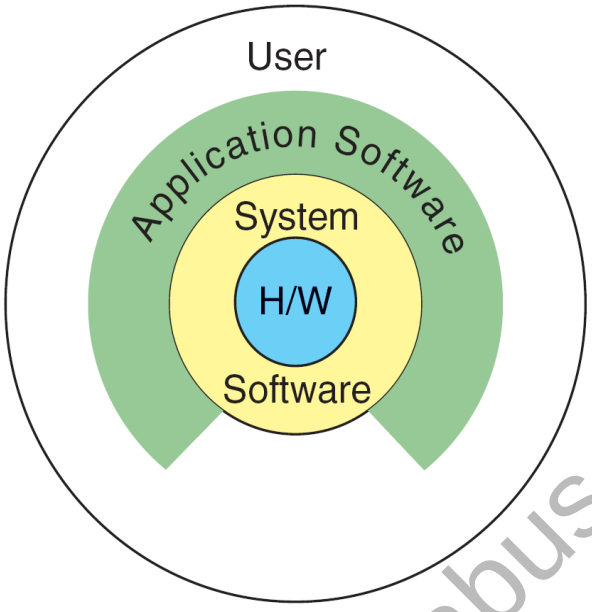
Basic hardware Components:



Types of software :



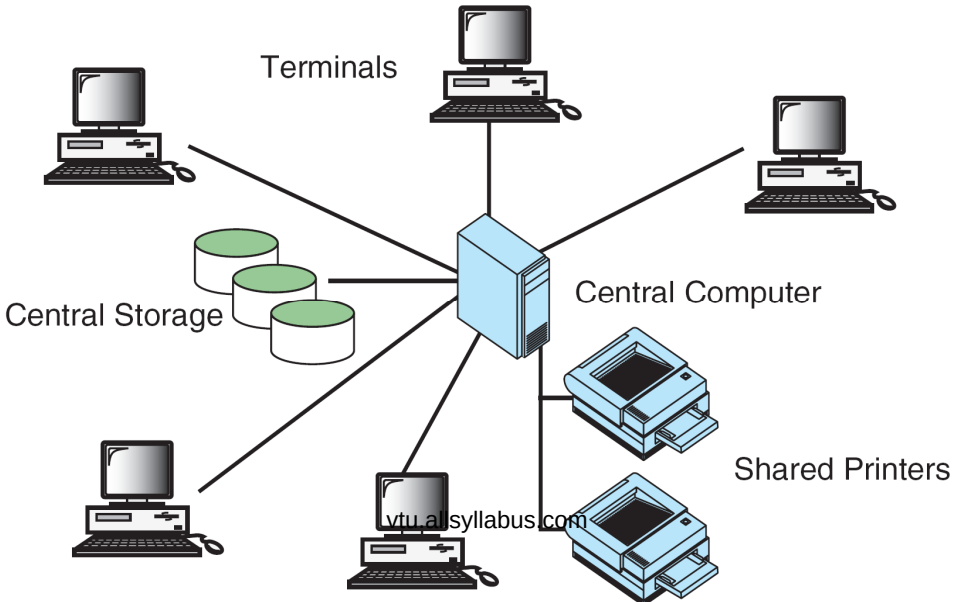
Relationship between system software and application software:



System Software: It helps in running the computer hardware and the computer system. System software is a collection of operating systems; devise drivers, servers, windowing systems and utilities. System software helps an application programmer in abstracting away from hardware, memory and other internal complexities of a computer.

Application Software: It enables the end users to accomplish certain specific tasks. Business software, databases and educational software are some forms of application software. Different word processors, which are dedicated for specialized tasks to be performed by the user, are other examples of application software.

Time sharing environment:



Time-sharing is an approach to interactive computing in which a single computer is used to provide apparently simultaneous interactive general-purpose computing to multiple users by sharing processor time. So basically, time sharing is for multi-user computer systems.

Time-sharing developed out of the realization that while any single user was inefficient, a large group of users together were not. This was due to the pattern of interaction; in most cases users entered bursts of information followed by long pause, but a group of users working at the same time would mean that the pauses of one user would be used up by the activity of the others. Given an optimal group size, the overall process could be very efficient. Similarly, small slices of time spent waiting for disk, tape, or network input could be granted to other users.

Implementing a system able to take advantage of this would be difficult. Batch processing was really a methodological development on top of the earliest systems; computers still ran single programs for single users at any time, all that batch processing changed was the time delay between one program and the next. Developing a system that supported multiple users at the same time was a completely different concept; the "state" of each user and their programs would have to be kept in the machine, and then switched between quickly. This would take up computer cycles, and on the slow machines of the era this was a concern. However, as computers rapidly improved in speed, and especially in size of core memory in which users' states were retained, the overhead of time-sharing continually decreased, relatively.

Problem Solving Method:

1. Recognize and understand the problem.
2. Accumulate facts.
3. Select appropriate theory.
4. Make necessary assumptions.
5. Solve the problem.
6. Verify results.

Algorithm and Flowchart:
Algorithm: An algorithm is just a detailed sequence of simple steps that are needed to solve a problem
Algorithms may be presented ..
1: in words
2:as a flowchart
3:in structured code

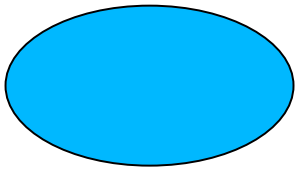
Flowchart: is an graphical representation of an algorithm.

5 steps in using computer as a problem solving tool

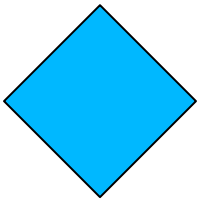
1. Develop an algorithm and a flowchart.
2. Write the program in a computer language
3. Enter the program into the computer.
4. Test and debug the program.
5. Run the program, input data, and get the result.

Basic Symbols:

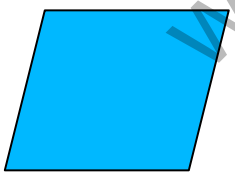
1.Stop/Start



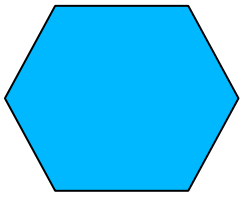
2.Question, Decisions (Use in Branching)



3. Input/Output

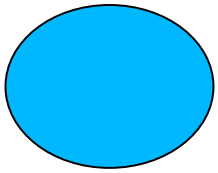


4. Preparation



5. Refers to separate flow char

6. Connecter (connects one part of the flowchart to another).



7.Process, Instruction

Example for algorithm

Finding the sum of two numbers:

Variables:
A:First Number
B:second Number
C:sum(A+B)

Algorithm:
Step1:Start
Step2:Input A
Step3:Input B
Step4:Calculate C=A+B
Step5:Output C
Step 6:Stop

Elements of C Program

System libraries
Header files
Application Source
Compiler
Linker
Creating C program
C program structure:
C program can be written using combinations of three control structures i.e. sequential, selection and repetitive. All C programs are made up of one or more function, each performing a particulars task. Every program has a special function named main(). It is special because the execution of any program starts from main functions.

General

```
Comments
Preprocessor directives
Global variables
main() function
{
        local variables
statements
--------------
--------------
}
function1()          other functions
{
        local variables
statements
--------------
--------------
}
```

```
/*Example number 1*/
#include<stdio.h>
main()
{
        Printf ("This is first program \n ");

return 0;
}
```

Notes:
Every program statements must end with a semi colon (;).
All programming statements must lie within curly braces.
Comment statements doesn't compile by the computer.

Comments /*……..*/
In C, /* is called opening comment mark and */ is closing comment mark. The C compiler ignores everything between the opening comment mark and closing comment mark. The comment is used for program documentation.

#include directory:
In C, #include is a preprocessor directive that tells the C preprocessor to look for a file and place the file in the location where #include indicates. The preprocessor includes the header file such as stdio.h, conio.h, string.h etc.

Header file:
The file that are included by the #include directive such as stdio.h is called header file. The header files are always placed at the start of a C program.

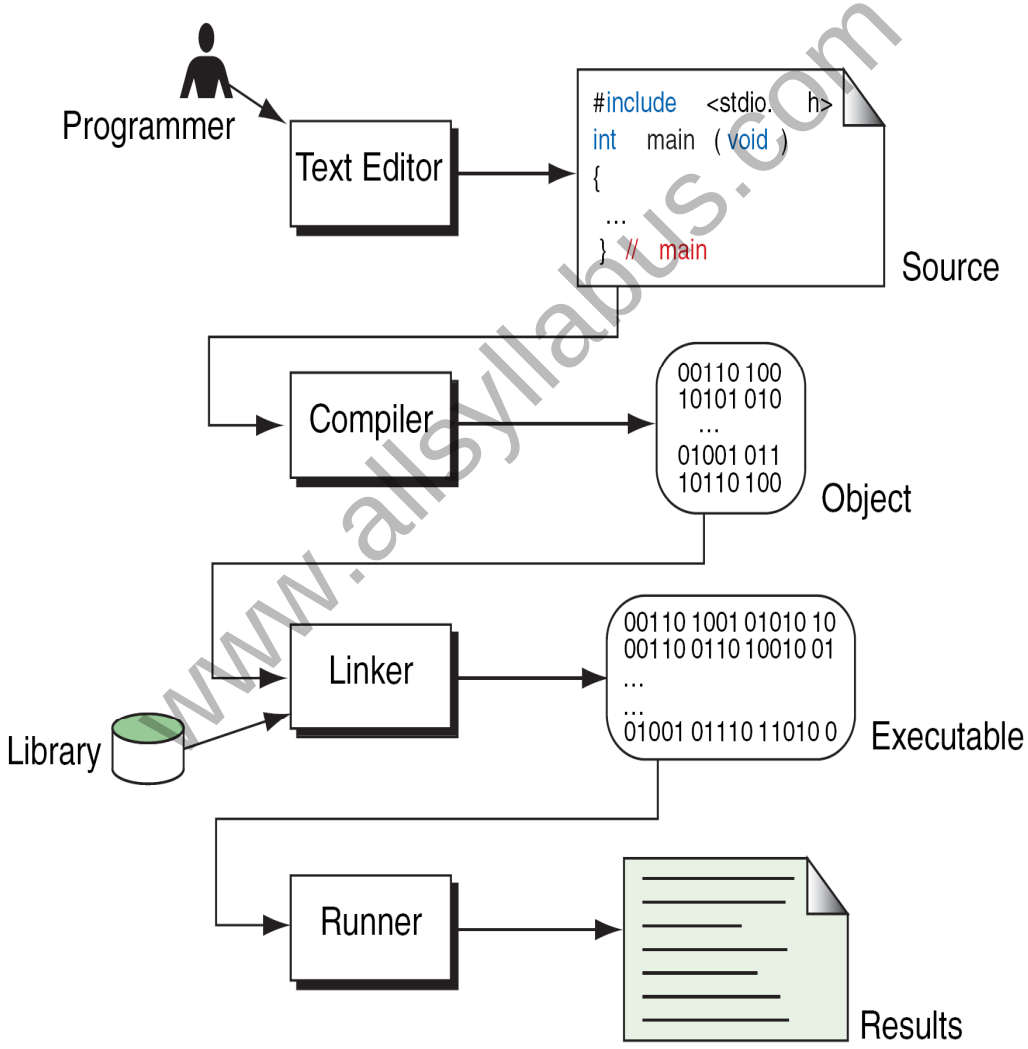
Angular <> braces:
In C, the angular bracket asks the C preprocessor to look for a header file in directory other than the current one. If we want to let the C preprocessor look into current directory first for header file before it starts to work, we can use double quotes to surround the name of the header file. Normally, the header file are saved in a sub directory called include.

Main function:
Every C program must have one and only one main functions. The main function can be put anywhere but the execution of a program starts with main function. The main function starts and end with curly braces {}

Printf functions and new line character:
The printf function is used to print the character string. The new line character which is usually sufficient at the end of the message, tells the system to move the cursor to the beginning of next line.

Return statements:
All function in C can return value. The main function itself returns an integer value. So, return 0 indicates that 0 is returned from main function and the program is terminated normally. We can return values other than 0 to tell the system that there is an error.

Building a C Program



Example

```
#include <stdio.h> /*preprocessor statements*/
#include<conio.h>
```

```
int main()
{
printf("Hello");
return 0;
}
```

This will print Hello on screen when executed

```
#include<stdio.h>
#include<conio.h>
void add(int x,int y)/*defined function*/
{
   int result;
   result=x+y;
   printf("sum of %d and %d is %d\n",x,y,result);
Void main()
 {
  clrscr();
  add(10,20); /*function calling*/
  add(30,20);
  getch();
}
```

Compiling:

1.The source file stored on the disk must be translated into machine language, this is the job of compiler.
2.The C Compiler is actually two separate programs : Preprocessor and the translator.

Running C program:
On linux platform
Create a file with extension .c for example consider a file hello.c
Preprocessing :assembly code cc -S hello.c
Compilation :a binary file cc -c hello.s
Linking: a.out or hello, an executable file cc hello.o
Cc -o hello hello.o
Loading (dynamic linking) and execution :./hello
./a.out
output : Hello

Identifiers and Keywords:
An identifiers can be defined as the name of the variable, function, arrays, structures, constants etc, are created by the programmer. They are the fundamental requirements of any programming language.

Keywords:
Keywords are reserved identifiers and they cannot be used as names for the program variables. The keywords are also called as reserved words. The meaning of the keywords already given to the compiler. There are 32 keywords available in C.

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

Delimiters:
Delimiters are used for syntactic meaning in C. These are as given below-

:	colon	used for label
;	semicolon	end of statements
( )	parenthesis	used in expression
{ }	curly braces	used for block of statements
[]	square bracket	used for array
#	hash	preprocessor directives
,	comma	variables delimiter

Variable / constant:
A variable is a named data storage location in computers memory. By using a variable name we are referring to the data stored in the location. A value that changes during the execution of the program is called variable.

Rules of variables declaration in C:
The variable name is a combination of alphabets, digits, underscore etc.
The first character in the variable name must be an alphabet.
A keyword cannot be used as variable name
No comma or blanks are allowed in variable name.
No special symbols ($, #) other than underscore can be used in variable name.
Uppercase & lowercase letters are distinct i.e. case sensitive
A variable name can't start with digit.

The maximum length of variable name should be 8 characters long in DOS based program & in ANSI C it supports up to 31 characters long.

Constants:

Identifier which doesn't change value during the execution of a program is called constant. The value associated with the storage always be constant.
C has 4 basic types of constant:
Integer constant
Floating point constant
Character constant
String constant

Integer constant
An integer constant has only numbers from 0 to 9. Examples are: 0, 111, 6598 [Invalid- 23,360 256.00 23-56]

Floating point constant
A floating point constant is a numeric constant with a decimal point are an exponent or both.
Eg: 0.5 488.33 1.56 e +9 9e-5
Invalid: 1,000.0 0.58 e+12.6

Character constant
The single character which can be stored in a storage space is called character constant. A character constant always denotes with single quotes.
Eg: 'A' 'C' 'X'
Invalid: "XY" 'ABC'

String Constant
The group of characters which represents with double quotes is a string constant. C allocates less memory for string constant for a string, it always manages by using arrays.
Eg: "PRIME" "COLLEGE"

Invalid: ‘PRIME’ COMPUTER

Data Types:
A data types defines a set of values that a variable can store along with a set of operation that can be performed on the variable. There are basically two types of data types:

1. Fundamental
2. Derived

Fundamental data types:
The fundamental data types are char, int, float and double

Data type	Memory required		
Char	1byte		
Int	2byte		
Float	2byte		
Double	4byte		

Derived data types:
Derived data types are derived from the fundamental data type. Some of the derived data types are short int, long int, long double etc.Char

Data type	Size	range
Signed char	1	-128 to 127
Unsigned char	1	0 to 255
Short signed int	2	-32768 to 32768
Short unsigned int	2	0 to 65535
Signed int	1	-128 to 127
Unsigned int	1	0 to 255
Long signed int	4	-2147483648 to +2147483648
Long unsigned int	4	0 to 4294967295
Float	4	3.4e-38 to 3.4e+38
Double	8	1.7e-308 to 1.7e +308
Long double	10	3.4e -4932 to 1.1e + 4932

KeywordsDefining Data:

Data can be defined in the following format:

Data definition	**data type**	**size**	**value assigned**
Char a, c;	char	1	-
Char a=2;	char	1	2
Int count=10;	int	2	10
Float num	float	4	-

Input / Output:

The important aspect of C programming language is its ability to handle input and output (I /O). Some of the inputs, output function are printf, scanf, getchar, putchar etc

Printf function

The printf function is used to display the value to the output devices It moves data from the computer's memory to the standard output devices.

Syntax: printf("format specifier" , arg1, arg2, ….arg n);

Formatspecifier control string

Arg 1. arg 2 variables or values

Format Specifier:

%d	int
%f	float
%c	char
%s	string

Examples:

```
#include<stdio.h>
main()
{
        float num1;
        int  num2=10, num3;
        num1=1.33;
        num3=5;
        printf("\n the value of num1 is %f " ,  num1);
        printf("\n the value of num2 is %d " ,  num2);
        printf("\n the value of num3 is %d " ,  num3);

        return 0;
}
```

Output:

```
the value of num1 is 1.33
the value of num2 is 10
the value of num3 is 5
```

scanf function:

The scanf function gets data from the standard input device and stores it in the computer memory.

Sysntax: scanf(" format specifier", & arg1, & arg2,……..,& argn) ;

Format specifier is the control string to denote the data type of variable

& arg1, & arg2,……..,& argn are the variables which stores values.

```
/*Example of scanf function*/
#include<stdio.h>
void main()
{
        int num1, num2, total;
        printf(" \n Enter the first numer");
```

```
        scanf(" %d" , &num1);
        printf(" \n Enter the second numer");
        scanf(" %d" , &num2);
        total=num1+num2;
        printf("\n Sum of two number= %d", total);
}
```

output:
Enter the first numer 10
Enter the second numer 20
Sum of two number = 30

Format specifier:
Format specifiers are the character string with % sign followed with a character. It specifies the type of data that is being processed. It is also called *conversion specifier*. When data is being output or input it must be specify with identifier(variable) and their format specifier.

c ➔ a single character
d ➔ a decimal integer
f ➔ a floating point number
e ➔ a floating point number
g ➔ a floating point number
h ➔ a sort integer
lf ➔ long range of floating point number (for double data types)
o ➔ an octal integer
x ➔ a hexadecimal integer
I ➔ a decimal, octal or hex decimal integer
S ➔ a string
U ➔ an unsigned decimal integer

Escape sequence:
Escape sequence are special character denoted by a backslash (\) and a character after it. It is called escape sequence because it causes an 'escape' from the normal way for characters are interpreted. For example if we use '\ n', then the character is not treated as n but it is treated as a new line. Some of the common escape sequence characters are:

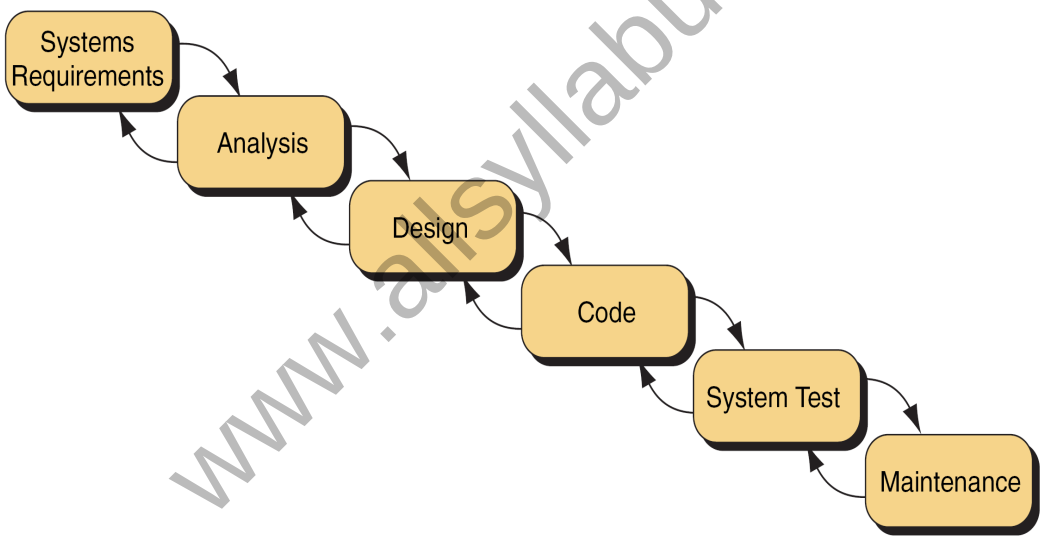
Escape sequence	Meaning
\a	Bell \ beep
\n	new line
\t	tab horizontal
\b	move the character to left one space
\r	return / enter key
\\	backslash
\'	single quote

\" double quote

getchar() function:
The getchar function is a part of input / output function of C programming language. It returns a single character from the standard input device. When it is transform to an integer it returns the ASCII value of the variable
Syntax: getchar()

putchar() function:
The putchar function is also part of input output library function of C language. It prints only a single character to a output device.
Syntax: putchar(variable)

System Development:
We have seen the steps that are necessary to build a program.
In this ,We discuss how we go about developing a program
This critical process determines the overall quality and success of our program.
If we carefully design each program using good structured development techniques, our program will be efficient, error free and easy to maintain.



Explanation:

the simplest software development life cycle model is the waterfall model, which states that the phases are organized in a linear order. A project begins with feasibility analysis. On the successful demonstration of the feasibility analysis, the requirements analysis and project planning begins.

The design starts after the requirements analysis is done. And coding begins after the design is done. Once the programming is completed, the code is integrated and testing is

done. On succeeful completion of testing, the system is installed. After this the regular operation and maintenance of the system takes place. The following figure demonstrates the steps involved in waterfall life cycle model.
With the waterfall model, the activities performed in a software development project are requirements analysis, project planning, system design, detailed design, coding and unit testing, system integration and testing. Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and beginning of the others. Some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation. Validation means confirming the output of a phase is consistent with its input (which is the output of the previous phase) and that the output of the phase is consistent with overall requirements of the system.
The consequences of the need of certification is that each phase must have some defined output that can be evaluated and certified. Therefore, when the activities of a phase are completed, there should be an output product of that phase and the goal of a phase is to produce this product. The outputs of the earlier phases are often called intermediate products or design document. For the coding phase, the output is the code. From this point of view, the output of a software project is to justify the final program along with the use of documentation with the requirements document, design document, project plan, test plan and test results.
Another implication of the linear ordering of phases is that after each phase is completed and its outputs are certified, these outputs become the inputs to the next phase and should not be changed or modified. However, changing requirements cannot be avoided and must be faced. Since changes performed in the output of one phase affect the later phases, that might have been performed. These changes have to made in a controlled manner after evaluating the effect of each change on the project.This brings us to the need for configuration control or configuration management.

The certified output of a phase that is released for the best phase is called baseline. The configuration management ensures that any changes to a baseline are made after careful review, keeping in mind the interests of all parties that are affected by it. There are two basic assumptions for justifying the linear ordering of phase in the manner proposed by the waterfall model.
For a successful project resulting in a successful product, all phases listed in the waterfall model must be performed anyway.
Any different ordering of the phases will result in a less successful software product.

Software engineering :
Software engineering is the establishment and use of sound engineering methods and principles to obtain software that is reliable and that works on real machines.
This definition, from the first international conference on software engineering in 1969, was proposed 30 years after the first computer was built. During this period, software was more of an art then a science.

Common Errors:

1Forgetting to put a break in a switch statement.
2.Using = instead of ==
3.Scanf errors:
 i:Forgetting to put an ampersand(&) on arguments
 ii:Using the wrong format for operand.
4.Size of arrays.
5.Integer division
6.Loop errors.
7.Not using prototypes.
8.Not initializing pointers.
9.String Errors
    i:Confusing character and string constants.
    ii:Comparing strings with ==.
    iii:Not null terminating strings.
    iv:Not leaving room for the null terminator.
10.Input /output errors
    i:Using getc(), getchar(), etc. incorrectly.
    ii:Using feof() incorrectly.
    iii:Leaving characters in the input buffer.
11.Using keywords as identifiers in the program.

Chapter 2

Structure of a C program

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /.

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

Assignment Statement

The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like

y = (m * x) + c

This assignment will save the value of the expression in vabriable y.

Arithmetic operators

Here are the most common arithmetic operators

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulo Reduction (Remainder from integer division)

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.

Here are some arithmetic expressions used within assignment statements.

velocity = distance / time;

force = mass * acceleration;

count = count + 1;

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

Shorthand	Equivalent
`i++;` or `++i;`	`i = i + 1;`
`i--;` or `--i;`	`i = i - 1;`

These operations are usually very efficient. They can be combined with another expression.

```
x = a * b++;   is equivalent to   x = a * b;
                                  b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

```
x = --i * (a + b);   is equivalent to   i = i - 1;
                                        x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - to ensure that your programs stay readable.

Another shorthand notation is listed below

Shorthand	Equivalent
`i += 10;`	`i = i + 10;`
`i -= 10;`	`i = i - 10;`
`i *= 10;`	`i = i * 10;`
`i /= 10;`	`i = i / 10;`

Type conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size, so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.

There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;

- The variable is too small to hold the value. In this case it will be corrupted (this is bad).
- The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.

Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting which temporarily disguises a value as a different type.

eg. The function sqrt finds the square root of a double.

int i = 256;

int root;

root = sqrt((double) i);

The cast is made by putting the bracketed name of the required type just before the value. (double) in this example. The result of sqrt((double) i); is also a double, but this is automatically converted to an int on assignment to root.

Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

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

Note that == is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

x == y

i > 10

a + b != c

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement or a for or a while loop. These will be introduced in a later chapter.

Logical Connectors

These are the usual And, Or and Not operators.

Symbol	Meaning
&&	And
||	Or
!	Not

They are frequently used to combine relational operators, for example

x < 20 && x >= 10

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
    printf("Not Acceptable !!\n");
```

Precedence of C operators

The following table shows the precedence of operators in C. Where a statement involves the use of several operators, those with the lowest number in the table will be applied first.

	Description	Represented By
1	Parenthesis	() []
1	Structure Access	. ->
2	Unary	! ~ ++ -- - * &
3	Mutiply, Divide, Modulus	* / %
4	Add, Subtract	+ -
5	Shift Right, Left	>> <<
6	Greater, Less Than, etc	> < =
7	Equal, Not Equal	== !=
8	Bitwise AND	&
9	Bitwise Exclusive OR	^
10	Bitwise OR	|
11	Logical AND	&&
12	Logical OR	||
13	Conditional Expression	?:
14	Assignment	= += -= etc
15	Comma	,

Summary

Three types of expression have been introduced here;

- Arithmetic expressions are simple, but watch out for subtle type conversions. The shorthand notations may save you a lot of typing.
- Comparison takes two numbers and produces a logical result. Comparisons are usually found controlling if statements or loops.
- Logical connectors allow several comparisons to be combined into a single test. Lazy evaluation can improve the efficiency of the program by reducing the amount of calculation required.

C also provides bit manipulation operators. These are too specialized for the scope of this course.

Chapter 3
FUNCTIONS

What is Function in C Language?

A function in C language is a block of code that performs a specific task. It has a name and it is reusable i.e. it can be executed from as many different parts in a C Program as required. It also optionally returns a value to the calling program

So function in a C program has some properties discussed below.

Every function has a unique name. This name is used to call function from "main()" function. A function can be called from within another function.

- A function is independent and it can perform its task without intervention from or interfering with other parts of the program.

3. A function performs a specific task. A task is a distinct job that your program must perform as a part of its overall operation, such as adding two or more integer, sorting an array into numerical order, or calculating a cube root etc.

6. A function returns a value to the calling program. This is optional and depends upon the task your function is going to accomplish. Suppose you want to just show few lines through function then it is not necessary to return a value. But if you are calculating area of rectangle and wanted to use result somewhere in program then you have to send back (return) value to the calling function.

C language is collection of various inbuilt functions. If you have written a program in C then it is evident that you have used C's inbuilt functions. Printf, scanf, clrscr etc. all are C's inbuilt functions. You cannot imagine a C program without function.

Structure of a Function:

A general form of a C function looks like this:

```
<return type> FunctionName (Argument1, Argument2, Argument3……)
{
Statement1;
Statement2;
Statement3;
}
```

An example of function.

```
int sum (int x, int y)
{
int result;
result = x + y;
```

```
return (result);
}
```

Advantages of using functions:

There are many advantages in using functions in a program they are:

- It makes possible top down modular programming. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
- The length of the source program can be reduced by using functions at appropriate places.
- It becomes uncomplicated to locate and separate a faulty function for further study.
- A function may be used later by many other programs this means that a c programmer can use function written by others, instead of starting over from scratch.
- A function can be used to keep away from rewriting the same block of codes which we are going use two or more locations in a program. This is especially useful if the code involved is long or complicated.

Types of functions:
A function may belong to any one of the following categories:

- Functions with no arguments and no return values.
- Functions with arguments and no return values.
- Functions with arguments and return values.
- Functions that return multiple values.
- Functions with no arguments and return values.

Example of a simple function to add two integers.

```
#include<stdio.h>
#include<conio.h>
void add(int x,int y)
{
int result;
result = x+y;
printf("Sum of %d and %d is %d.\n\n",x,y,result);
}
void main()
{
```

```
clrscr();
add(10,15);
add(55,64);
getch();
}
```

Program Output

- Sum of 10 and 15 is 25
- Sum of 55 and 64 is 119.

Output of above program.

Explanation

Before I explain, let me give you an overview of above c program code. This is a very simple program which has only function named “add()”. This “add()” function takes two values as arguments, adds those two values and prints the result.

Line 3-8 is a function block of the program. Line no. 3 is the header of function, void is return type of function, add is function name and (int x, int y) are variable which can hold integer values to x and y respectively. When we call function, line no. “12, 13, 14”, we need to send two integer values as its argument. Then these two values get stored in variable x and y of line no. 3. Now we have two values to perform addition; in line no. 5 there is an integer declaration named “result”. This integer will store the sum of x and y (please see line no. 6). Line no. 7 simply prints the result with message.

Now imagine the same program without using function. We have called “add()” function three times, to get the same output without using function we have to write Line no. 6 & 7 three time. If you want to add more value later in the program then again you have to type those two lines. Above example is a small and simple program so it does not appear great to use function. But assume a function consist 20 – 30 or more lines then it would not be wise to write same block of code wherever we need them. In such cases functions come handy, declare once, use wherever you want.

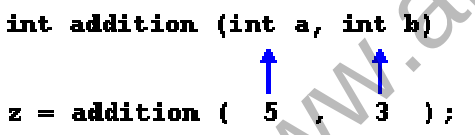
Calling Functions:

Call by Value and Call by Reference

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

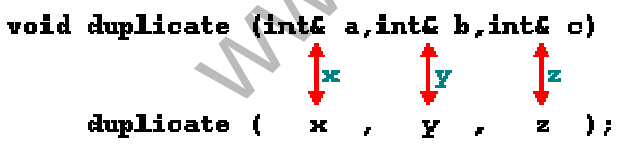
```
// passing parameters by reference
#include <iostream>
using namespace std;

```

```
6
7
8  void duplicate (int& a, int& b, int& c)
9  {
10
11   a*=2;
12   b*=2;
13
14   c*=2;
15 }
16
17
18 int main ()
   {
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
   }
```

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



o explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Default values in parameters:
When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

```
6
5
```

As we can see in the body of the program there are two calls to function divide. In the first one:

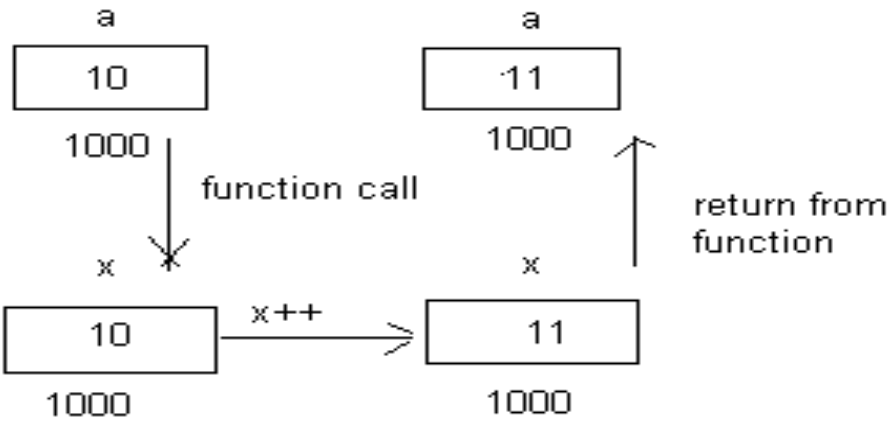
```
divide (12)
```

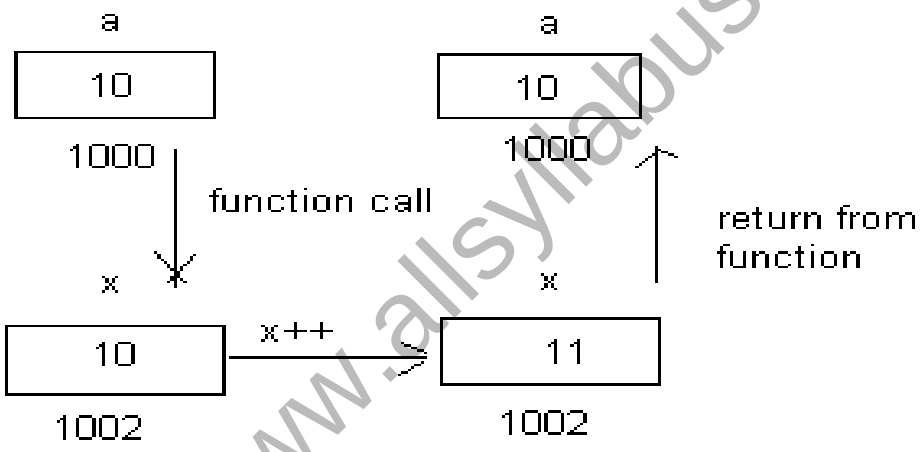
we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).





Call by value => copying value of variable in another variable. So any change made in the copy will not affect the original location.

Call by reference => Creating link for the parameter to the original location. Since the address is same, changes to the parameter will refer to original location and the value will be over written.

Call by reference Passes original argument Changes in function effect original,

Only used with trusted functions.

Calling a Function

The call to a function in C simply entails referencing its name with the appropriate arguments. The C compiler checks for compatibility between the arguments in the calling sequence and the definition of the function.

Library functions are generally not available to us in source form. Argument type checking is accomplished through the use of header files (like stdio.h) which contain all the necessary information. For example, as we saw earlier, in order to use the standard mathematical library you must include math.h via the statement

#include < math.h>

at the top of the file containing your code. The most commonly used header files are
< stdio.h> -> defining I/O routines

< ctype.h> -> defining character manipulation routines

< string.h> -> defining string manipulation routines

< math.h> -> defining mathematical routines

< stdlib.h> -> defining number conversion, storage allocation
and similar tasks

< stdarg.h> -> defining libraries to handle routines with variable
numbers of arguments

< time.h> -> defining time-manipulation routines

In addition, the following header files exist:
< assert.h> -> defining diagnostic routines

< setjmp.h> -> defining non-local function calls

< signal.h> -> defining signal handlers

< limits.h> -> defining constants of the int type

< float.h> -> defining constants of the float type

Random Number Generation:

- rand function
  - Load <stdlib.h>
  - Returns "random" number between 0 and RAND_MAX (at least 32767)

- i = rand();
- Pseudorandom
  - Preset sequence of "random" numbers
  - Same sequence for every function call

- Scaling
  - To get a random number between 1 and n
    - 1 + (rand() % n)
  - rand() % n returns a number between 0 and n - 1
  - Add 1 to make random number between 1 and n
    - 1 + (rand() % 6)

    number between 1 and 6.

The scope and lifetime of variables in functions:

The scope and lifetime of the variables define in C is not same when compared to other languages. The scope and lifetime depends on the storage class of the variable in c language the variables can be any one of the four storage classes:

1.AutomaticVariables
2.Externalvariable
3.Staticvariable
4.Registervariable.

The scope actually determines over which part or parts of the program the variable is available. The lifetime of the variable retains a given value. During the execution of the program. Variables can also be categorized as local or global. Local variables are the variables that are declared within that function and are accessible to all the functions in a program and they can be declared within a function or outside the function also.

Automatic variables

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined by default all variable declared without any storage specification is automatic. The values of variable remains unchanged to the changes that may happen in other functions in the same program and by doing this no error occurs.

```
/* A program to illustrate the working of auto variables*/
#include
voidmain()
{
intm=1000;
function2();
printf("%d\n",m);
}

function1()
{
intm=10;
printf("%d\n",m);
}
function2()
{
intm=100;
function1();
printf("%d\n",m);
}
```

A local variable lives through out the whole program although it accessible only in the main. A program with two subprograms function1 and function2 with m as automatic variable and is initialized to 10,100,1000 in function 1 function2 and function3 respectively. When executes main calls function2 which in turns calls function1. When main is active m=1000. But when function2 is called, the main m is temporarily put on the shelf and the new local m=100 becomes active. Similarly when function1 is called both previous values of m are put on shelf and latest value (m=10) become active, a soon as it is done main (m=1000) takes over. The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables:

Variables which are common to all functions and accessible by all functions of aprogram are internal variables. External variables can be declared outside a function.

Example

```
intsum;
floatpercentage;
main()
{
…..
…..
}
function2()
{
….
… .
}
```

The variables sum and percentage are available for use in all the three functions main, function1, function2. Local variables take precedence over global variables of the same name.

For example:

```
inti=10;
voidexample(data)
intdata;
{
inti=data;
}

main()
{
example(45);
}
```

In the above example both the global variable and local variable have the same name as i.
The local variable i take precedence over the global variable. Also the value that is stored in integer i is lost as soon as the function exits.

A global value can be used in any function all the functions in a program can access the global variable and change its value the subsequent functions get the new value of the global variable, it will be inconvenient to use a variable as global because of this factor every function can change the value of the variable on its own and it will be difficult to get back the original value of the variable if it is required.

Global variables are usually declared in the beginning of the main program ie., before the main program however c provides a facility to declare any variable as global this is possible by using the keyword storage class extern. Although a variable has been defined after many functions the external declaration of y inside the function informs the compiler that the variable y is integer type defined somewhere else in the program. The external declaration does not allocate storage space for the variables. In case of arrays the definition should include their size as well. When a variable is defined inside a function as extern it provides type information only for that function. If it has to be used in other functions then again it has to be re-declared in that function also.

Example:

```
main()
{
intn;
out_put();
externfloatsalary[];
……
…..
out_put();
}

voidout_put()
```

```
{
externfloatsalary[];
intn;
….
…..
}
float salary[size];
```

A function when its parameters and function body are specified this tells the compiler to allocate space for the function code and provides type info for the parameters. Since functions are external by default we declare them (in calling functions) without the qualifier extern.

Multi-file programs:

Programs need not essentially be limited into a single file, multi-file programs is also possible, all the files are linked later to form executable object code. This approach is very useful since any change in one file does not affect other files thus eliminating the need for recompilation of the entire program. To share a single variable in multiple programs it should be declared, as external variables that are shared by two or more files are obviously global variables and therefore we must declare them accordingly in one file and explicitly define them with extern in other file. The example shown below illustrates the use of extern declarations in multi-file programs.

File1.c

```
main()
{
externintj;
intk;
}

function1()
{
intz;
…
….
}
```

file2.c

```
function2()
{
int                                                                    k;
}
function3()
{
int                                                                    num;
…
….
}
```

the function in main file1 reference the variable j that is declared as global in file 2. Here function1() cannot access j if the statement extern int k is places before main then both the functions could refer to j. this can also be achieved by using extern int j statement inside each function in file1.

The extern specifier tells the compiler that the following variables types and names have already been declared elsewhere and no need to create storage space for them. It is the responsibility of the linker to resolve the reference problem. It is important to note that a multi-file global variable should be declared without extern in one of the files.

Static variables:

The value given to a variable declared by using keyword static persistes until the end of theprogram.
A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to stat in the example shown below x is incremented to 1. because x is static, this value persists and therefore the next call adds another 1 to x giving it a value of 2. The value of x becomes 3 when third call is made. If we had declared x as an auto then output would here been x=1 all the three times.

```
main()
{
intj;
for(j=1;j&lt;3;j++)
stat();
}
stat();
```

```
{
staticintx=0;
x=x+1;
printf("x=%d\n",x);
}
```

Register variables:

A variable is usually stored in the memory but it is also possible to store a varible in the compilers register by defining it as register variable. The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster.

This is done as follows:

```
register int count;
```

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be:

5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
```

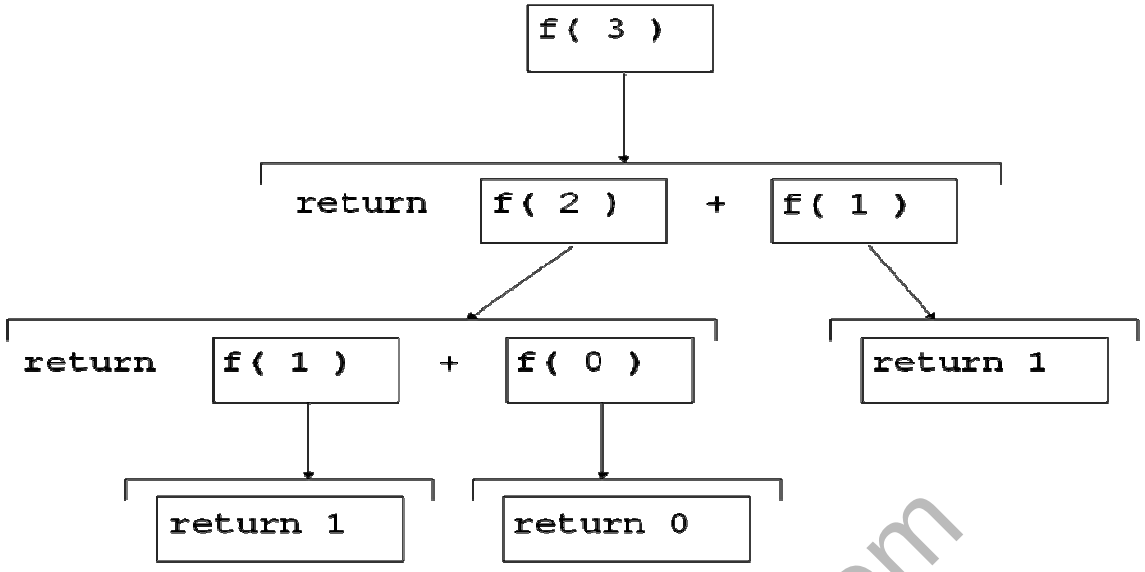
```
Please type a number: 9
9! = 362880
```

```
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return (1);
}


int main ()
{
  long number;
  cout << "Please type a number: ";
  cin >> number;
  cout << number << "! = " << factorial
(number);
  return 0;
}
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.



Tips and common errors

1.Several possible errors related to passing parameters.

- It is a compiler error if the types in the prototype declaration and function definition are incompatible.
- It is a compiler error to have a different number of actual parameters in the function call then there are in the prototype statement.
- It is logic error if you code the parameters in the wrong order. Their meaning will be inconsistence
- In the called program.

2.It is compiler error to define local variables with the same identifiers as formal parameters.

3.Using void return with a function that excepts a return value or using a return value with a function that excepts a void return is a compiler error.

4.Each parameters type must be individually specified: you cannot use multiple definitions like in variables.

5.Forgetting the semicolon at the end or a function prototype statement is a compiler error. Similarly , using a semicolon at the end of the header in a function definition is a compiler error.

6.It is most likely a logic error to call a function from within itself or one of its called functions.

7.It is a compiler error to attempt to define a function within the body of another function .

8.It is a run time error to code a function call without the parentheses, even when function has no parameters.

9.It is a compiler error if the type of data in the return statement does not match the function return type.

10.It is logic error to call srand every time you call rand.

Chapter 4

Selection: Making Decisions, Repetitions

A program consists of a number of statements which are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.

Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

This section will discuss the use of control statements in C. We will show how they can be used to write powerful programs by;

- Repeating important sections of the program.
- Selecting between optional sections of a program.

The if else Statement

This is used to decide whether to do something at a special point, or to decide between two courses of action.

The following test decides whether a student has passed an exam with a pass mark of 45

```
if (result >= 45)
    printf("Pass\n");
else
    printf("Fail\n");
```

It is possible to use the if part without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If is is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{       printf("Passed\n");
        printf("Congratulations\n")
}
else
{       printf("Failed\n");
        printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
        printf("Passed: Grade A\n");
```

```
else if (result >= 60)
    printf("Passed: Grade B\n");
else if (result >= 45)
    printf("Passed: Grade C\n");
else
    printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

The switch Statement

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

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{    switch(number) {
    case 0 :
```

```
            printf("None\n");
            break;
        case 1 :
            printf("One\n");
            break;
        case 2 :
            printf("Two\n");
            break;
        case 3 :
        case 4 :
        case 5 :
            printf("Several\n");
            break;
        default :
            printf("Many\n");
            break;
        }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

Loops

C gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loops is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.

The while Loop

The while loop repeats a statement until the test at the top proves false.

As an example, here is a function to return the length of a string. Remember that the string is represented as an array of characters terminated by a null character '\0'.

```
int string_length(char string[])
{       int i = 0;

        while (string[i] != '\0')
                i++;

        return(i);
}
```

The string is passed to the function as an argument. The size of the array is not specified, the function will work for a string of any size.

The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't null, the index is incremented and the test is repeated.

The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
}
while (input_value != 1 && input_value != 0)
```

The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialisation of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

The example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{       float total = 0.0;
        int i;

        for(i = 0; i < count; i++)
                total += array[i];

        return(total / count);
}
```

The for loop ensures that the correct number of array elements are added up before calculating the average.

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialisation or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behaviour to be specified simply and quickly.

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

Chapter 5

ARRAYS

The meaning of an array:

A group of related data items that share a common name is called an array. For example, we can define an array name marks to represent a set of marks obtained by a group of students. A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

Example,

Marks[7]

Represents the marks of the 7^{th} student. The complete set of values is referred to as an array, the individual values are called elements. The arrays can be of any variable type.

One-dimensional array:

When a list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or one dimensional array.

In C language ,single-subscripted variable xi can be represented as

x[1],x[2],x[3]……x[n]

The subscripted variable xi refers to the ith element of x. The subscript can begin with number 0. For example, if we want to represent a set of five numbers, say (57,20,56,17,23), by a array variable num, then we may declare num as follows

int num[5];

And the computer reserves five storage locations as shown below:

Num[0]

Num[1]

Num[2]

Num[3]

Num[4]

The values can be assigned as follows:

```
Num[0]=57;
Num[1]=20;
Num[2]=56;
Num[3]=17;
Num[4]=23;
```

The table below shows the values that are stored in the particular numbers.

Num[0]

Num[1]

Num[2]

Num[3]

Num[4]

57

20

56

17

23

Two dimensional arrays:

There are certain situations where a table of values will have to be stored. C allows us to define such table using two dimensional arrays.

Two dimensional arrays are declared as follows:

Type array_name [row_size][column_size]

In c language the array sizes are separated by its own set of brackets.

Two dimensional arrays are stored in memory as shown in the table below. Each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

	Column0	Column1	Column2
	[0][0]	[0][1]	[0][2]
Row 0	210	340	560
	[1][0]	[1][1]	[1][2]
Row 1	380	290	321
	[2][0]	[2][1]	[2][2]
Row2	490	235	240
	[3][0]	[3][1]	[3][2]
Row3	240	350	480

Declaration and initialization of arrays

The arrays are declared before they are used in the program. The general form of array declaration is

type variable_name[size];

The type specifies the type of element 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.

Example:

float weight[40]

Declares the weight to be an array containing 40 real elements. Any subscripts 0 to 39 are valid.

Similarly,

int group1[11];

Decalres the group1 as an array to contain a maximum of 10 integer constants.

The C language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold.

For example:

char text[10];

Suppose we read the following string constant into the string variable text.

"HOW ARE YOU"

Each character of the string is treated as an element of the array text and is stored in the memory as follows.

'H'

'O'

'W'

'A'
'R'
'E'

'Y'
'O'
'U'
'\o'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element text[11] holds the null character '\o' at the end. When declaring character arrays, we must always allow one extra element space for the null terminator.

Initialization of arrays

The general form of initialization of arrays is:

static type array-name[size]={ list of values};

The values in the list are separated by commas.

For example, the statement below shows

static int num[3]={2,2,2};

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

For example:

static float num1[5]={0.1,2.3,4.5};

Will initialize the first three elements to 0.1,2.3 and 4.5 and the remaining two elements to zero. The word static used before type declaration declares the variable as a static variable.

In some cases the size may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

static int count[ ]= {2,2,2,2};

Will declare the counter array to contain four elements with initial values 2.

Character arrays may be initialized in a similar manner. Thus, the statement

static char name[ ]={ 'S 'W,'A,'N}

Declares the name to be an array of four characters, initialized with the string "SWAN"

There certain draw backs in initialization of arrays.

There is no convenient way to initialize only selected elements.

There is no shortcut method for initializing a large number of array elements.

Program to read and write two dimensional array

```
#include<stdio.h>
main()
{
        int a[10][10];
         int i,j,row,col;
        printf("\n Input row and column  of a matrix:");
        scanf("%d %d",  &row,&col);
        for(i=0; i<row;i++)
           for(j=0;j<col;j++)
          scanf("%d", &a[i][j]);

for(i=0;i<row;i++)
{
        for(j=0;j<col;j++)
        printf("%5d", a[i][j]);
```

```
printf("\n");
}
```

Program showing one-dimensional array

```
main()

{
        int i;
        float a[10],value1,total;
 printf("Enter 10 Real numbers\n");
 for(i=0;i<10;i++)
{
        scanf("%f", &value);
        x[i]=value1;
}

total=0.0;
for(i=0;i<10;i++)
total=total+a[i]*a[i];

printf("\n");
for(i=0;i<10;i++)
printf("x[%2d]= %5.2f\n", i+1, x[i]);
printf("\ntotal=%.2f\n", total);
}
```

Programming examples:

1) Program to print multiplication tables

```
#define R1   4
#define  C1   4

main()
{
        int row,col,prod[R1][C1];
        int i,j;
        printf(" MULTIPLICATION TABLE \n\n");
        printf("   ");

        for(j=1;j<=C1;j++)
         printf("%4d",j);
         printf("\n");
          printf("----------------------------------------\n");
         for(i=0;i<R1;i++)
          {
                row=i+1;
                printf("%2dl", R1);
         for(j=1;j<=C1;j++)
           {
                col=j;
                prod[i][j]=row*col;
                printf("%4d", prod[i][j]);
            }
                printf("\n");
         }
        }
```

Output

MULTIPLICATION TABLE

```
        1  2   3   4
  ---------------------------------------
1   |     1   2   3    4
2   |     2   4   6    8
3   |     3    6   9   12
4   |     4    8   12  16
```

2) Program to show swapping of numbers.

```
#include<stdio.h>

main()
{
        int x[10], i,j,temp;
        for(i=0;i<=9;++i)
        scanf("%d", &x[i]);
for(j=0;j<=8;j+=2)
    {
temp=x[j];
        x[j]=x[j+1];
   x[j+1]=temp;
}

for(i=0;i<=9;++i)
printf("%d", x[i]);
printf("\n");
}
```

3) Program to sort a list of numbers using bubble sort:

```
#define N 10
```

```
    main()
{
    int i,j,k;
float a[N],t;
printf("Enter the number of items\n");
scanf("%d", &n);
printf("Input %d values \n", n);
for(i=1; i<=n ;;i++)
  scanf("%f", &a[i]);
for(i=1;i<=n-1;i++)
{
   for(j=1;j<=n-i; j++)
      {
         if)a[j]<=a[j+1])
{
     t=a[j];
     a[j]=a[j+1];
     a[j+1]=t;
}
else
    continue;
}
}
for(i=1;i<=n;i++)
 printf("%f", a[i]);
}
```

4) Program to calculate standard deviation

```
#include<math.h>
#define MAX 100
```

```
main()
{
    int i,n;
float val[MAX],deviation,sum,ssqr,mean,var,stddev;
sum=ssqr=n=0;
printf("Input values: input-1 to end\n");
for(i=1;i<MAX;i++)
{
    scanf("%f", &val[i]);
    if(val[i]==-1)
      break;
    sum+=val[i];
    n+=1;
}

mean=sum/(float)n;
for(i=1;i<=n;i++)
{
    deviation=val[i]-mean;
    ssqr+=deviation*deviation;
}
 var=ssqr/(float)n;
stddev=sqrt(var);
printf("\n Number of items:%d\n",n);
printf("Mean: %f \n", mean);
printf("Standard deviation: %f\n", stddev);
}
```

5) Program to find the largest and smallest of numbers

```
#include<stdio.h>
main()
{
    int I,s, largcount,smcount;
    float num[30],lar,small;

printf("\n size of array (MAX 30): \t");
scanf("%d", &size);

printf("\n Array elements:\t");

for(i=0;i<size;i++)
    scanf("%f", &num[i]);

for(i=0;i<size;i++)
    printf("%f", &num[i]);

lar=small=num[0];
larcount=smcount=1;
for(i=1;i<size;i++)
{
    if(num[i]>lar)
{
    lar=num[i];
    larcount=i+1;
}

elseif(num[i]<small)
{
    small=num[i];
    smcount=i+1;
```

```
}
}
printf("\n Largest value is % f found at %d", lar,larcount);
printf("\n Smallest value is  %f found at %d ", small, smcount);
}
```

Chapter 6

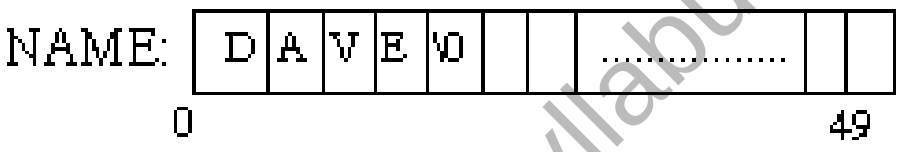
C STRINGS

What are C stings?

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0 (or \0).

For example:

char name[50] = “DAVE”;



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	H	I	S		\n		a		S	T	R	I	N	G	.	\0

printf("This is a string value. Beep! Beep! \7\7");

Initializing Strings

Following the discussion on characters arrays, the initialization of a string must 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 part 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 complier 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 nullcharacter for example printf(“%s”,name); can be used to display the entire contents of the array name.

C String header

The named string.h (<cstring> header in C++) is used to work with C strings. Confusion or programming errors arise when strings are treated as simple data types. Specific functions have to be employed for comparison and assignment such as strcpy for assignment instead of the standard = and strcmp instead of == for comparison.

Functions included in <string.h>

strlen()

The "strlen()" function gives the length of a string, not including the NULL character at the end:

```
/* strlen.c */

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

void main()
{
  char *t = "XXX";
  printf( "Length of <%s> is %d.\n", t, strlen( t ));
}
```

This prints:

```
Length of <XXX> is 3.
```

strcpy()

The "strcpy" function copies one string from another. For example:

```
/* strcpy.c */

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

void main()
{
  char s1[100],
       s2[100];
  strcpy( s1, "string 2" );
  strcpy( s2, "string 1" );

  puts( "Original strings: " );
  puts( "" );
  puts( s1 );
  puts( s2 );
  puts( "" );

  strcpy( s2, s1 );

  puts( "New strings: " );
  puts( "" );
  puts( s1 );
  puts( s2 );
}
```

This will print:

```
Original strings:

string 1
```

```
string 2

New strings:

string 1
string 1
```

Please be aware of two features of this program:

- This program assumes that "s1" has enough space to store the final string. The "strcpy()" function won't bother to check, and will give erroneous results if that is not the case.
- A string constant can be used as the source string instead of a string variable. Using a string constant for the destination, of course, makes no sense.

These comments are applicable to most of the other string functions.

strncpy()

There is a variant form of "strcpy" named "strncpy" that will copy "n" characters of the source string to the destination string, presuming there are that many characters available in the source string. For example, if the following change is made in the example program:

```
strncpy( s2, s1, 5 );
```

-- then the results change to:

```
New strings:

string 1
string
```

Notice that the parameter "n" is declared "size_t", which is defined in "string.h".

strcat()

The "strcat()" function joins two strings:

```
/* strcat.c */
```

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

void main()
{
  char s1[50],
       s2[50];
  strcpy( s1, "Tweedledee " );
  strcpy( s2, "Tweedledum" );
  strcat( s1, s2 );
  puts( s1 );
}
```

This prints:

```
Tweedledee Tweedledum
```

strncat()

There is a variant version of "strcat()" named "strncat()" that will append "n" characters of the source string to the destination string. If the example above used "strncat()" with a length of 7:

```
strncat( s1, s2, 7 );
```

-- the result would be:

```
Tweedledee Tweedle
```

Again, the length parameter is of type "size_t".

strcmp()

The "strcmp()" function compares two strings:

```
/* strcmp.c */

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

#define ANSWER "blue"
```

```
void main()
{
  char t[100];
  puts( "What is the secret color?" );
  gets( t );
  while ( strcmp( t, ANSWER ) != 0 )
  {
    puts( "Wrong, try again." );
    gets( t );
  }
  puts( "Right!" );
}
```

The "strcmp()" function returns a 0 for a successful comparison, and nonzero otherwise. The comparison is case-sensitive, so answering "BLUE" or "Blue" won't work.

There are three alternate forms for "strcmp()":

strncmp()

- A "strncmp()" function which, as might be guessed, compares "n" characters

in the source string with the destination string: "strncmp(s1, s2, 6)".

stricmp()

- A "stricmp()" function that ignores case in comparisons.

strnicmp()

- A case-insensitive version of "strncmp" called "strnicmp".

strchr()

The "strchr" function finds the first occurrence of a character in a string. It returns a pointer to the character if it finds it, and null if not. For example:

```
/* strchr.c */

#include <stdio.h>

#include <string.h>
```

```
void main()
{
  char *t = "MEAS:VOLT:DC?";
  char *p;
  p = t;
  puts( p );
  while(( p = strchr( p, ':' )) != NULL )
  {
    puts( ++p );
  }
}
```

This prints:

```
MEAS:VOLT:DC?
VOLT:DC?
DC?
```

The character is defined as a character constant, which C regards as an "int". Notice how the example program increments the pointer before using it ("++p") so that it doesn't point to the ":" but to the character following it.

strrchr()

The "strrchr()" function is almost the same as "strchr()", except that it searches for the *last* occurrence of the character in the string.

strstr()

The "strstr()" function is similar to "strchr()" except that it searches for a string, instead of a character. It also returns a pointer:

```
char *s = "Black White Brown Blue Green";
...
puts( strstr( s, "Blue" ) );
```

strlwr() and strupr()

The "strlwr()" and "strupr()" functions simply perform lowercase or uppercase conversion on the source string. For example:

```
/* casecvt.c */

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

void main()
{
  char *t = "Die Barney die!";
  puts( strlwr( t ) );
  puts( strupr( t ) );
}
```

-- prints:

```
die barney die!
DIE BARNEY DIE!
```


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);
}
```

Tips and common errors:

1.Do not confuse characters and string constants: the character constant is enclosed in a single quotes, and the string constant is enclosed in double quotes.

2.Remember to allocate memory space for the string delimiter when declaring and defining an array of char to hold a string.

3.String are manipulated with string functions, not operators.

4.The header file <string.h> is required for string functions.

5.The standard string functions requires a delimited string. You cannot use them on array of char that is not terminated with a delimiters.

6.Do not confuse string array and string pointers.

7.Passing a character to a functions when a string is required is another common error. this is most likely to occur with the formatted input and output functions, in which case, it is logical error. Passing a character in place of a string when a prototype header is declared is a compile error.

8.Using the address operator for a parameter in the "scanf" function with a string is a coding error.

9.It is a compile error to assign a string to a character, even when the character is part of string.

10.Using the assignment operator with strings instead of a function call is a compile error.

11.Since strings are built in an array structure, they may be accessed with indexes and pointers. When accessing individual bytes, it is a logic error, to access beyond the end of the data structures.