Programming in C and Data Structures

CODE:	**15PCD13/23**	**IA Marks: 20**
Hrs/Week:	**04**	**Exam Hrs: 03**
Total Hrs:	**50**	**Exam Marks:80**

Objectives:

The objectives of this course is to make students to learn basic principles of Problem solving, implementing through C programming language and to design & develop programming skills.

MODULE I

INTRODUCTION TO C LANGUAGE: Pseudocode solution to problem, Basic concepts of a C program, Declaration, Assignment & Print statement, Types of operators and expressions, Programming examples and exercise.

Text 1: Chapter 2 . **Text 2: 1**.1, 1.2,1.3. **10Hours**

MODULE II

BRANCHING AND LOOPING: Two way selection (if, if-else, nested if-else, cascaded if-else), switch statement, ternary operator? Go to, Loops (For, do-while, while) in C, break and continue, programming examples and exercises.

Text 1: Chapter 3**.** **Text 2:** 4.4. **10 Hours**

MODULE III

ARRAYS, STRINGS AND FUNCTIONS:

ARRAYS AND STRINGS: Using an array, Using arrays with Functions, Multi-Dimensional arrays. String: Declaring, Initializing, Printing and reading strings, strings manipulation functions, strings input and output functions, arrays of strings, programming examples and Exercises.

Text 1: 5.7, **Text 2**: 7.3, 7.4, chapter 9

FUNCTIONS: Functions in C, Argument Passing – call by value, Functions and program structure, location of functions, void and parameter less Functions, Recursion, programming examples and exercises.

Text 1: 1.7, 1.8, Chapter 4. **Text 2:** 5.1 to 5.4. **10 Hours**

MODULE IV

STRUCTURES AND FILE MANAGEMENT: Basic of structures, structures and Functions, Arrays of structures, structure Data types, type definition, Defining, opening and closing of files, Input and output operations, programming examples and exercises.

Text 1: 6.1 to 6.3. **Text 2**: 10.1 to 10.4, Chapter 11. **10 Hours**

MODULE V

POINTERS AND PREPROCESSORS: Pointers and address, pointers and functions arguments, pointers and arrays, address arithmetic, character pointer and functions, pointers to pointer ,Initialization of pointers arrays, Dynamic allocations methods, Introduction to Preprocessors, Complier control Directives, programming examples and exercises.

Text 1: 5.1 to 5.6, 5.8. **Text 2: 12.2,** 12.3**, 13.1 to 13.7**.

Introduction to Data Structures: Primitive and non primitive data types, Definition and applications of Stacks, Queues, Linked Lists and Trees.

Text 2 : 14.1, 14.2, 14.11, 14.12, 14.13, 14.15, 14.16, 14.17, 15.1. **08 Hours + 04 Hours**

Course Outcomes: On completion of this course, students are able to

- **Achieve Knowledge of design and development of problem solving skills.**
- **Understand the basic principles of Programming in C language**
- **Design and develop modular programming skills.**
- **Effective utilization of memory using pointer technology**
- **Understands the basic concepts of pointers and data structures.**

TEXT BOOK:

1. Brain W. Kernighan and Dennis M. Richie: The C programming Language, 2nd Edition, PHI, 2012.
2. Jacqueline Jones & Keith Harrow: Problem Solving with C, 1st Edition, Pearson 2011.

Reference Books:

1. Vikas Gupta: Computer Concepts and C Programming, Dreamtech Press 2013.
2. R S Bichkar, Programming with C, University Press, 2012.
3. V Rajaraman: Computer Programming in C, PHI, 2013.

Table Of Content

Functions in C,
Argument Passing – call by value, Functions and
program structure,
location of functions,
void and parameter less Functions,
Recursion

programming examples and exercises.

MODULE IV

STRUCTURES AND FILE MANAGEMENT **82**

Basic of structures,
Structures and Functions,
Arrays of structures,
Structure Data types,
Type definition,
Defining, opening and closing of files
Input and output operations
Programming examples and exercises

MODULE V

POINTERS AND PREPROCESSORS **92**

Pointers and address
Pointers and functions arguments
Pointers and arrays, address arithmetic
Character pointer and functions,
Pointers to pointer
Initialization of pointers arrays
Dynamic allocations methods
Introduction to Preprocessors
Complier control Directives

Programming examples and exercises.

Introduction to Data Structures:

Primitive and non primitive data types

Definition and applications of Stacks

Queues, Linked Lists and Trees

MODULE I

INTRODUCTION TO C LANGUAGE

Pseudo-code solution to problem,

The process of transforming the description of a problem into the solution of that problem by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques, and tools.

A few problems examples:

1. Calculating a tip
2. Double checking a grocery receipt
3. Automating LaTeX to HTML conversion

The Software Development Method

Also known as the *software life cycle*

Starting from the problem statement:

1. Requirements specification --- removal of ambiguity from the problem statement (acquiring expertise)
2. Analysis --- a detailed determination of the problem inputs and outputs
3. Design --- the development of a logically-ordered set of steps, whose application to the problem input produces the problem output
4. Implementation --- the creation of a working program from the design
5. Testing and verification --- of the working program
6. Documentation
    1. The problem being solved
    2. Who is responsible?
    3. The design
    4. The particulars of the implementation

Algorithm Design and Representation

Algorithm:

A sequence of a finite number of steps arranged in a specific logical order which, when executed, produces the solution for a problem.

An algorithm must satisfy the following requirements:

1. Input --- usually required
2. Output
3. Unambiguousness --- computers don't accept ambiguity
4. Generality --- solves a class of problems
5. Correctness --- correctly solve the given problem
6. Finiteness --- termination
7. Efficiency --- recognition of finite computing resources: CPU cycles, memory

Pseudocode:

A semiformal, English-like language with a limited vocabulary that can be used to design and describe algorithms.

- Meta-programming language
- Algorithm representation

Pseudocode Structural Elements

C. Bohm and G. Jacopini proved in 1966 than pseudocode required only three structural elements

The Sequence Control Structure
A series of steps or statements that are executed in the order they are written in an algorithm.

Example:

```
print "What is your name?"
read name
print "How old are you, ", name, "?"
read age
let birthYear = CURRENT_YEAR - age
```

begin/ end pair grouping:

```
begin
  let amountDue = overDue + currentBilling + penalty
  print "You owe: ", amountDue
end
```

The Selection Control Structure

The alternatives of two courses of action only one of which is taken depending on the outcome of a condition, which is either true or false.

```
if condition
  then_part
else
  else_part
end_if
```

Structure of then_part, else_part:

- A single statement
- A set of statements enclosed by begin/ end

```
if payment is overdue
  begin
    let amountDue = pastDue + currentBilling + penalty
    print "You owe: ", amountDue
  end
else
  print You owe: , currentBilling
end_if
```

Alternative nested if-else structure element: else_if

```
if grade < 60
  print "F"
else_if grade < 70
  print "D"
else_if grade < 80
  print "C"
else_if grade < 90
  print "B"
else
  print "A"
end_if
```

The Repetition Control Strucutre

Specifies a block of one or more statements that are repeatedly executed until a condition is satisfied.

```
while condition
  loop_body
end_while
```

Structure of loop_body

```
let sum = 0
while there are input numbers to sum
  begin
    print "Next number: "
    read number
    let sum = sum + number
  end
end_while
print "The sum is: ", sum
```

Basic concepts of a C program,

C language Preliminaries

Introduction: C is a programming language developed at AT & T' s Bell laboratories of USA in 1972. It was designed and written by a system programmer Dennis Ritchie. The main intention was to develop a language for solve all possible applications. C language became popular because of the following reasons.

1. C is a robust language , which consists of number of built-in functions and operators can be used to write any complex program
2. Programs written in c are executed fast compared to other languages.
3. C language is highly portable
4. C language is well suited for structured programming.
5. C is a simple language and easy to learn.

Fundamentals of Problem Solving

Executing a C program

Executing a program written in C involves a series of steps. These are

1. Creating the program.
2. Compiling the program.
3. Linking the program with functions that are needed from the C library.
4. Executing the program.

Introduction to C Language

Structure of a C program:

The basic structure of a C program is shown below

Documentation Section

Link Section

Definition Section

Global Declaration Section

main() Function Section

{

Declaration Part

Executable Part

}

Subprogram section

Function 1

Function 2

.

.

.

Function n

The documentation section consists of a set of comment lines giving the name of the program, the name author and other details which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section contains all symbolic constants. There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section. Every C program must have one main() function section. This section contains two parts declaration part and executable part. The declaration part declares all the variables used in the executable part. There should be at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is logical end of the program. All statements in the declaration and executable parts end with a

semicolon. The subprogram section contains all the user-defined functions that are called in the main function. The main function is very important compared to other sections.

Character Set

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The characters in C are grouped into the following categories.

1. Letters
2. Digits
3. Special characters
4. White Spaces

Letters	Digits
Uppercase A…Z	All decimal digits 0…9
Lowercase a….z	

Special characters

, comma	& ampersand
. period	^ carat
; semicolon	*asterisk
: colon	-minus sign
? question mark	+ sign
‘ apostrophe	< opening angle bracket
! exclamation mark	(or less than sign)
| vertical bar	> closing angle bracket
/ slash	(or greater than sign)
\ backslash	(left parenthesis
~ tilde	) right parenthesis
_ underscore	[left bracket
$ dollar sign	] right bracket
% percent sign	{ left brace
# number sign	} right brace

White Spaces

Blank Space

Horizontal tab

Carriage return

New line

Form feed

Identifiers:

In c language every word is classified into either keyword or identifier. All keywords have fixed meanings and these meanings cannot be changed. These serve as basic building blocks for program statements. All keywords must be written in lowercase. The list of all ANSI C keywords are listed below

ANSI 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

Identifiers refer to the names of variables, functions and arrays. These are user defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two word in long identifiers.

Integer Constants:

An integer constant refers to a sequence of digits. There are three types of integers, namely decimal, octal and hexadecimal. Decimal integers consist of a set of digits 0 through 9, preceded by an optional – or + sign. Some examples of decimal integer constants are

123

-431

0

34567

+678

Spaces, commas, and non-digit characters are not permitted between digits. For example

15 750

20,000

Rs 1000

are illegal numbers.

An octal integer constant consists of any combination of digits from the set 0 through 7 with a leading 0. Some examples are:

037

0

0435

0567

A sequence of digits preceded by 0x is considered as hexadecimal integer. They may also include alphabets A through F or a through F. The letters A through F represent the numbers 10 through 15. The examples for hexadecimal integers are:

0x2

0x9F

0xbcd

0x

Floating-point constants:

Integer numbers are inadequate to represent quantities that vary continuously, such as distances ,heights ,temperatures ,prices and so on. These quantities are represented by numbers containing fractional parts like 23.78. Such numbers are called floating-point constants or real constants. Examples for floating-point constants are given below.

213.

.95

-.71

+.5

A real number may also be expressed in exponential(or specific notation). For example the value 213.45 may be written as 2.1345e2 in exponential notation. e2 means multiply by 102. The general form is

mantissa e exponent

Character Constants:

A character constant contains a single character enclosed within a pair of single quote marks. Examples of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the number 5. The last constant is a blank space. Character constants have integer values known as ASCII values. For Example, the statement

printf ("%d", 'A');

would print the number 65,the ASCII value of the letter a. Similarly, the statement

printf("%c", 65)

would give the output as letter 'A'. It is also possible to perform arithmetic operations on character constants.

Backslash Character Constants:

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for new line character. The below table gives you a list of backslash constants.

Backslash Character Constants

Constant	Meaning
'\a'	audible alert(bell)
'\b'	back space
'\f'	form feed
'\n'	new line character
'\r'	carriage return

'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
' \" '	double quote
'\?'	question mark
'\\'	backslash mark
'\0'	null character

Note that each one of them represents one character, although they consist of two characters. These character combinations are called escape sequences.

String constants:

A string constant is a sequence of characters enclosed in double quotes. The letters may be numbers, special characters and blank space.

Examples are given below "THANK YOU"

"2345"

"?....."

"7+8-9"

"X"

Remember that a character constant 'X' is not equivalent to the single character string constant("X"). A single character string constant does not have an equivalent integer value as a single character constant. These type of constants are used in programs to build meaningful programs.

Meaning of variables:

A variable is a data name that may be used to store a data value. Unlike the constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples are given below.

Average

Height

Total

Counter_1

Rules for defining variables:

Variable names may consist of letters, digits, and the underscore(_) character, subject to the rules given below:

1. The variables must always begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, the length should not be normally mare than eight characters. Since first eight characters are treated as significant by many compilers.
3. Uppercase and lowercase are significant. That is ,the variable Rate is not the same as rate or TOTAL.
4. The variable name should not be a keyword.
5. White space is not allowed.

Some examples are given below:

Abhi	Value	I_rate
Mumbai	s1	ph_value
Rate	sum1	distance

The examples given below are invalid:

345	(rate)
%	56 nd

Declaration of variables:

Identifiers which are used as variable names should be prefixed as integer or float by the following declaration should appear at the beginning of a program before the variable names are used.

type _name variable name……variable name;

The type_name is always a reserved word. The type_name available for variable names storing numbers are int for integers and float for floating point numbers. Valid examples are given below:

int n, height ,count ,digit;

float rate, average , y_coordinate,p1;

When a variable name is declared then a memory location is identified and given this name.

The following declarations of variables are invalid:

Float a ,b ,c ; (comma after float is not valid)

Int :x; (: is not valid)

Real x, y;(real is not the correct type_name)

Fundamental data types:

C language has varieties of data types. Storage representations and machine instructions differ from machine to machine. The variety of data types available allow the programmer to select the appropriate to the needs of the application as well as the machine. The fundamental or primary data types are integer(int), character(char), floating point(float), and double-precision floating point(double). Many of them also has extended data types such as long, double, short, unsigned and signed. The range of basic four types are given below:

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

Char, int, float and double are all keywords and therefore their use is reserved. They may not be used as names of variables. Char stands for "character" and int stands for "integer". The keywords short int, long int and unsigned int may be and usually are, shortened to just short, long, and unsigned, respectively. Also, double and long float are equivalent, but double is the keyword usually used.

Integer Types:

Integers are whole numbers with a range of values supported by a particular machine. Generally integers occupy one word of storage. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767. A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647. C has three classes of integer storage, namely short int, int, and long int in both unsigned and signed forms. For example, short int represents fairly small integer.

Values and requires half the amount of storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always

positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535. We declare long and unsigned integers to increase the range of values.

Floating point types:

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword float. The type double can be used when the accuracy provided by a float number is not sufficient. A double data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. To extend the precision further, we may use long double which 80 bits..

Character types:

A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits(one byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned chars have values between 0 and 255, signed chars have values from -128 to 127.

Size and range of data types on a 16-bit machine

Type	Size(bits)	Range
Char or signed char	8	-128 to 127
Unsigned char	8	0 to 255
Int or signed int	16	-32,768 to 32,767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed Long int	32	-2,147,483,648 to 2,147,483,647
Unsigned long int	32	0 to 4,294,967,295
Float	32	3.4E-38 to 3.4+38
Double	64	1.7E-308 to 1.7E+308
Long double	80	3.4E-4932to 1.1E+4932

Input and Output Functions:

Scanf functions:-

The function scanf() is used to read data into variables from the standard input, namely a keyboard. The general format is:

Scanf(format-string, var1,var2,……varn)

Where format-string gives information to the computer on the type of data to be stored in the list of variables var1,var2……varn and in how many columns they will be found

For example, in the statement:

Scanf("%d %d", &p, &q);

The two variables in which numbers are used to be stored are p and q. The data to be stored are integers. The integers will be separated by a blank in the data typed on the keyboard.

A sample data line may thus be:

456 18578

Observe that the symbol &(called ampersand) should precede each variable name. Ampersand is used to indicate that the address of the variable name should be found to store a value in it. The manner in which data is read by a scanf statement may be explained by assuming an arrow to be positioned above the first data value. The arrow moves to the next data value after storing the first data value in the storage location corresponding to the first variable name in the list. A blank character should separate the data values. The scanf statement causes data to be read from one or more lines till numbers are stored in all the specified variable names.No that no blanks should be left between characters in the format-string. The symbol & is very essential in front of the variable name. If some of the variables in the list of variables in the list of variables in scanf are of type integer and some are float, appropriate descriptions should be used in the format-string.

For example:

Scanf("%d %f %e", &a , &b, &c);

Specifies that an integer is to be stored in a, float is to be stored in b and a float written using the exponent format in c. The appropriate sample data line is:

485 498.762 6.845e-12

Printf function:

The general format of an output function is

Printf(format-string, var1,var2…..varn);

Where format-string gives information on how many variables to expect, what type of arguments they are , how many columns are to be reserved for displaying them and any character string to be printed. The printf() function may sometimes display only a message and not any variable value. In the following example:

printf("Answers are given below");

The format-string is:

Answers are given below

And there are no variables. This statement displays the format-string on the video display and there are no variables. After displaying, the cursor on the screen will remain at the end of the string. If we want it to move to the next line to display information on the next line, we should have the format-string:

printf("Answers are given below\n");

In this string the symbol \n commands that the cursor should advance to the beginning of the next line.

In the following example:

printf("Answer x= %d \n", x);

%d specifies how the value of x is to be displayed. It indicates the x is to be displayed as a decimal integer. The variable x is of type int. %d is called the conversion specification and d the conversion character . In the example:

printf("a= %d, b=%f\n", a, b);

the variable a is of type int and b of type float or double. % d specifies that a is to be displayed as an integer and %f specifies that, b is to be displayed as a decimal fraction. In this example %d and %f are conversion specifications and d, f are conversion characters. Example to indicate how printf() displays answers.

```
/*Program illustrating printf()*/
# include<stdio.h>
main()
{
    int a= 45,   b= 67
    float x=45.78 , y=34.90
printf("Output:\n");
```

```
printf("1,2,3,4,5,6,7,,8,0\n");
printf("\n");
printf("%d, %d,,%f ,%f \n" , a,b,x,y);
printf("\n");
}
```

Output:

1234567890

45,67,45.78,34.90

Example for illustrating scanf and printf statements:

```
/* Program for illustrating use of scanf and printf statements */
#include<stdio.h>
main()
{
 int a,b,c,d;
float x,y,z,p;
scanf("%d %o %x %u", &a, &b ,&c ,&d);
printf("the first four data displayed\n");
printf(("%d %o %x %u \n", a,b,c,d);
scanf("%f %e %e %f", &x, &y, &z, &p);
printf("Display of the rest of the data read\n");
printf("%f %e %e %f\n", x,y,z,p);
printf("End of display");
}
```

Input:

-768 0362 abf6 3856 -26.68 2.8e-3 1.256e22 6.856

Output:

The first four data displayed

-768 362 abf6 3856

Display of the rest of the data read

-26.680000 2.800000 e-03 1.256000e+22 6.866000

End of display

Formatted input and output using format specifiers:

The function scanf is the input analog of printf, providing many of the same conversion facilities in the opposite direction.

int scanf (char *format, …..)

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments. The format argument is described below; the other arguments, each of which must be a pointer, indicate where the corresponding converted input to be stored. scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found . On end of file EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted. A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that scanf will read across line boundaries to find its input, since new lines are white space. (White space characters are blank, tab, new line, carriage return, vertical tab and form feed).

The general syntax is

int printf (char *format, arg1,arg2……..)

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed. The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications each of which causes conversion and printing of the next successive argument to printf. Each

conversion specification begins with a % and ends with a conversion character. Between the % and the conversion character there may be in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left or right, to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to printed from a string, or the number of digits after the decimal point of a floating point value, or the minimum number of digits for an integer.
- An h if the integer is to be printed as a short, or l if as a long.

The putchar function:

Single characters can be displayed using the C library function putchar. This function is complementary to the character input function getchar. The putchar function, like getchar, is a part of the standard C I/O library. It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character type variable. It must be expressed as an argument to the function, enclosed in parentheses, following the word putchar.

The general syntax is

putchar(character variable)

where character variable refers to some previously declared character variable.

A C program contains the following statements

```
Char c;
……
putchar(c);
```

C programs:

1) Program to demonstrate printf statement

```
#include<stdio.h>
main()
{
  printf("hello, world\y");
```

```
printf("hello, world\7");
printf("hello, world\?");
}
```

2) Program to convert farenheit to Celsius

```
#include<stdio.h>
main()
{
   float fahr, Celsius;

        printf(" enter the value for farenheit\n");
        scanf(" %f", &fahr);
        Celsius=(5.0/9.0)*fahr-32.0;
        printf("%f  %f \n", fahr,Celsius);
}
```

3) Program to depict interactive computing using scanf function.

```
#include<stdio.h>
main()
{
    int number;
    printf("enter an integer number\n");
    scanf("%d",  &number);
    If (number<100)
     {
      printf("Your number is smaller than 100\n\n");
       else
       printf("Your number contains more than two digits\n");
}
```

Output

Enter an integer number 54

Your number is smaller than 100

Enter an integer number 108

Your number contains more than digits

4) Program to depict interactive investment program

```
#include<stdio.h>
main()
{
  int year,period;
 float amount,inrate,value;
printf("Input amount , interest rate and period \n\n");
scanf("%f %f %d", &amount, &inrate,&period);
printf("\n");
year=1;
while(year<=period)
{
     value amount + inrate*amount;
     printf("%2d Rs. %8.2f\n", year, value);
     amount=value;
     year=year+1;
}
}
```

5) Program to calculate the average of a set of N numbers

```
#define N 10
main()
{
   int count;
    float sum, average,number;
sum=0;
count=0;
while(count<N)
```

```
{
   scanf("%f", &number);
   sum=sum+number;
count=count+1;
}
average= sum/N;
printf("N=%d Sum= %f", N, sum);
printf("Average=%f", average);
}
```

6) Program to convert days to months and days

```
#include<stdio.h>
main()
{
   int months,days;
   printf("enter days \n");
   scanf("%d", &days);
   months=days/30;
   days=days%30;
   printf("Months = %d  Days= %d", months,days);
}
```

Types of operators and expressions,

Arithmetic operators:

C provides all the basic arithmetic operators. The operators +,-,* and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus

*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division produces the remainder of an integer division.

Examples are:

a - b	a + b
a * b	a / b
a % b	-a * b

Here a and b are variables and are known as operands. The modulo divison operator % cannot be used on floating point data.

Arithmetic expressions:

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. Expressions are evaluated using an assignment statement of the form

Variable=expression;

The table below shows the algebraic expression and C language expression

Algebraic expression	**C expression**
a x b-c	a*b-c
(m + n) (x + y)	(m + n) *(x + y)
(a b)/c	a*b/c
3x2+2x+1	3*x*x+2*x+1
x/y +c	x/y + c

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the precious value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

```
x=a*b-c;
y=b/c*a;
z=a-b/c+d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a ,b ,c and d must be defined before they are used in the expressions.

Modes of expression:

There are three different modes of expression.

1. Integer Arithmetic
2. Real Arithmetic
3. Mixed-mode Arithmetic

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic. This mode of expression always yields an integer value. The largest integer value depends on the machine, as pointed out earlier

Example:

If a and b are integers then for a=14 and b=4

We have the following results:

a - b=10

a + b = 18

a * b = 56

a / b = 3

a % b = 2

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is, 6/7=0 and -6/-7=0

but -6/7 may be zero -1 (Machine dependent)

Similarly, during modulo division , the sign of the result is always the sign of the first operand(the dividend). That is

-14 % 3 =-2

-14 % -3= -2

14 % -3=2

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If x, y, and z are floats, then we will have:

x=6.0/7.0=0.857143

y= 1.0/3.0 =0.333333

z= -2.0/3.0= -0.666667

The operator % cannot be used with real operands.

Mixed- mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number.

Thus

15/10.0=1.5

where as

15/10=1

Arithmetic operators precedence:-

In a program the value of any expression is calculated by executing one arithmetic operation at a time. The order in which the arithmetic operations are executed in an expression is based on the rules of precedence of operators.

The precedence of operators is :

Unary (-) FIRST

Multiplication(*) SECOND

Division(/) and (%)

Addition(+) and Subtraction(-) LAST

For example, in the integer expression –a *b/c+d the unary- is done first, the result –a is multiplied by b, the product is divided by c(integer division) and d is added to it. The answer is thus:

-ab/c+d

All the expressions are evaluated from left to right. All the unary negations are done first. After completing this the expression is scanned from left to right; now all *, / and % operations are executed in the order of their appearance. Finally all the additions and subtractions are done starting from the left of the expression..

For example, in the expression:

Z=a + b* c

Initially b*c is evaluated and then the resultant is added with a. Suppose if want to add a with b first, then it should be enclosed with parenthesis , is shown below

Z = (a + b) * c

Use of parentheses:

Parentheses are used if the order of operations governed by the precedence rules are to overridden.In the expression with a single pair of parentheses the expression inside the parentheses is evaluated FIRST. Within the parentheses the evaluation is governed by the precedence rules.

For example, in the expression:

a * b/(c+d * k/m+k)+a

the expression within the parentheses is evaluated first giving:

c+dk/m+k

After this the expression is evaluated from left to right using again the rules of precedence giving

ab/c+dk/m+k +a

If an expression has many pairs of parentheses then the expression in the innermost pair is evaluated first, the next innermost and so on till all parentheses are removed. After this the operator precedence rules are used in evaluating the rest of the expression.

((x * y)+z/(n*p+j)+x)/y+z

xy,np+j will be evaluated first.

In the next scan

Xy+z/np+j +x

Will be evaluated. In the final scan the expression evaluated would be:

(Xy+ z/np+j+x)/y +z

Increment and Decrement operators:-

The increment operator ++ and decrement operator – are unary operators with the same precedence as the unary -, and they all associate from right to left. Both ++ and – can be applied to variables, but no to constants or expressions. They can occur in either prefix or postfix position, with possibly different effects occurring. These are usually used with integer data type.

The general syntax is:

++variable|--variable| variable++| variable—

Some examples are

++count -kk index++ unit_one--

We use the increment and decrement statements in for and while extensively.

Consider the following example

```
m=5;
y=++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m=5;
y=m++;
```

then the value of y would be 5 and m would 6. A prefix operator first adds to 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Similar is the case, when we use ++(or--) in the subscripted variables. That is, the statement a[i++]=10; is equivalent to

```
a[i]=10;
i=i+1;
```

The increment and decrement operators can be used in complex statements. Example

```
m=n++ -j+10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation.

Relational operators:

We often compare two quantities and depending on their relation, to take certain decisions. For example, we may compare the age of two persons, or the price of two items, and

so on. These comparisons can be done with the help of relational operators. C supports six relational operators in all. These operators and their meanings are shown below

Relational Operators

Operator	Meaning
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and has the following form: ae- 1 relational operator ae-2. ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables or combination of them.

Given below are some examples of simple relational expressions and their values:

4.5<= 10 TRUE

4.5< 10 FALSE

-35>= 0 FALSE

10< 7+5 TRUE

a+b == c+d TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators. Relational expressions are used in decision statements such as, if and while to decide the course of action of a running program.

Logical operators:

In addition to the relational operators . C has the following three logical operators.

&&	logical AND
||	logical OR
!	logical NOT

The logical operators && and || are used when we want to test more than one condition and make decisions.

Example:

a>b && x == 10

An expression of this kind which combines two or more relational expression is termed as a logical expression or a compound relational expression. Like the simple relational expressions , a logical expression also yields a value of one or zero, according to the truth table shown below. The logical expression given above is true only if a>b is true and x==10 is true. If either (or both) of them are false, the expression is false.

Truth Table

Op-1	op-2	Value of the expression	
Op-1 && op-2	op-1 || op2		
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

If(age>55 && salary <1000)

If (number<0 || number>100)

Relational and logical expressions:

We have seen that float or integer quantities may be connected by relational operators to yield an answer which is true of false. For example the expression,

Marks>=60

Would have an answer true if marks is greater than or equal to 60 and false if marks is less than 60. The result of the comparison (marks>=60) is called a logical quantity. C provides a facility to combine such logical quantities by logical operators to logical expressions. These logical expressions are useful in translating intricate problem statements.

Example :

A university has the following rules for a student to qualify for a degree with Physics as the main subject and Mathematics as the subsidiary subject:

He should get 50 percent or more in Physics and 40 percent or more in Mathematics.

If he gets less than 50 percent in Physics he should get 50 percent or more in Mathematics. He should get atleast 40 percent in Physics.

If he gets less than 40 percent in Mathematics and 60 percent or more in Physics he is allowed to reappear in an examination in Mathematics to qualify.

In all the other cases he is declared to have failed.

A Decision Table for Examination Results

Chemistry marks	>=50	>=35	>= 60	Else
Physics Marks	>=35	>=50	<35	
Pass	x	x	-	-
Repeat Physics	-	-	x	-
Fail	-	-	-	x

```
/*This program implements above rules*/
include<stdio.h>
main()
{
unsigned int roll_no, physics_marks, chem_marks;
while(scanf("%d %d %d", &roll_no,&physics_marks, &chem_marks)!=EOF)
{
If(((chem_marks>=50 &&(physics_marks>=35)) || ((chem_marks>=40)) &&(physics_marks>=50)))
Printf("%d %d %d  Pass\n", roll_no, chem_marks, physics_marks);
Else If (( chem_marks>=60)) && physics_marks<35))
Printf("%d %d %d Repeat Physics\n", roll_no,physics_marks,chem_marks);
Else
Printf("%d %d %d Failed \n", roll_no, physics  _marks, chem_marks);
}
}/*End while*/
}/*End main*/
```

Precedence of relational operators and logical operators:

Example:

(a>b *5) &&(x<y+6)

In the above example, the expressions within the parentheses are evaluated first. The arithmetic operations are carried out before the relational operations. Thus b*5 is calculated and after that a is compared with it. Similarly y+6 is evaluated first and then x is compared with it .

In general within parentheses:

The unary operations, namely, -,++,--,! (logical not) are performed first.

Arithmetic operations are performed next as per their precedence.

After that the relational operations in each sub expressions are performed, each sub expression will be a zero or non _ zero. If it is zero it is taken as false else it is taken as true.

These logical values are now operated on by the logical operators.

Next the logical operation && is performed next.

Lastly the evaluated expression is assigned to a variable name as per the assignment operator.

The conditional operators:

An operator called ternary operator pair "?:" is available in C to construct conditional expressions of the form.

exp1? exp2: exp3;

where exp1,exp2, and exp3 are expressions.

The operator ?; works as follows: exp1 is evaluated first. If it is nonzero(true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions(either exp2 or exp3) is evaluated.

For example, consider the following statements

```
x=3;
y=15;
z=(x>y)?x:y;
```

In this example, z will be assigned the value of b. This can be achieved using the if..else statements as follows:

```
If (x>y)
  z=x;
else
  z=b;
```

Bitwise operators:

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. where the filename is the name containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of the 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 directory and then in the standard directories.

Bitwise Operators

Operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	One's Complement

The Comma Operator

C has some special operators. The comma operator is one among them. This operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression.

For example, the statement

Value=(a=2, b=6 ,a+b);

First assigns the value 2 to a, then assigns 6 to b, and finally assigns 8(i.e 2+6) to value. The comma operator has the lowest precedence of all operators,hence the parentheses are necessary.

Some examples are given below:

In for loops:

For(a=1, b=10;a<=b; a++, b++)

In while loops:

While(c=getchar(), C!='10')

Exchanging values:

T=x, x=y, y=t;

The precedence of operators among themselves and across all the set of operators:

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. The operator at the higher level of precedence are evaluated first.

Operator	Description
+	Unary plus
-	Unary minus
++	Increment
--	Decrement
!	Logical negation
~	One's Complement
&	Address
size of(type)	type cast conversion
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
<<	left shift
>>	Right shift
<	less than
<=	less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equality
!=	Inequality
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND

||	Logical OR
?:	Conditional expression
=	Assignment operators
*= /= %=	
+= -= &=	
^= |=	
<<= >>=	
,	Comma operator

The associatively of operators:

The operators of the same precedence are evaluated either from left to right or from right to left depending on the level. This is known as the associatively property of an operator.

The table below shows the associatively of the operators:

Operators	Associativity
() [ ] →	left to right
~ ! –(unary)	left to right
++ -- size of(type)	
&(address)	left to right
*(pointer)	
* / %	
<< >>	left to right
<<= >>=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
=+ =- *= /= %= &= ^= |= <<= >>=	right to left
,(comma operator)	left to right

Evaluation of expressions involving all the above type of operators:

The following expression includes operators from six different precedence groups.

Consider variables x, y , z as integer variables.

Z+=(x0>0 && x<=10) ? ++x : x/y;

The statement begins by evaluating the complex expression

(x>0 && x<=10)

If this expression is true, the expression ++x is evaluated. Other wise, the a/b is evaluated. Finally, the assignment operation(+=) is carried out, causing the value of c to be increased by the value of the conditional expression. If for example x, y, and z have the values 1,2,3 respectively, then the value of the conditional expression will be 2(because the expression ++a will be evaluated), and the value of z will increase to 5(z=3+2). On the other hand, if x,y and z have the values 50,10,20 respectively, then the value of the conditional expression will be 5(because the expression x/y will be evaluated) and the value of z will increase to 25(z=20+5).

MODULE II

BRANCHING AND LOOPING

Two way selections (if, if-else, nested if-else, cascaded if-else)

An expression such as x = 0or i++or printf(...)becomes a statement when it is followed by a semicolon, as in

x = 0; i++;

printf(...);

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal. Braces {and }are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another.

The if-elsestatement is used to express decisions. Formally the syntax is

```
if (expression)
statement1
else
statement2
```

where the else part is optional. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement 1 is executed. If it is false (expression is zero) and if there is an else part, statement 2 is executed instead. Since an if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

if (expression)

instead of

if (expression!= 0)

Sometimes this is natural and clear; at other times it can be cryptic. Because the elsepart of an if-elseis optional, there is an ambiguity when an else if omitted from a nested ifsequence. This is resolved by associating the else with the closest previous else-less if. For example, in

```
if (n > 0)
if (a > b)
z = a;
else
z = b;
```

the else goes to the inner if, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
if (a > b)
z = a;
}
else
z = b;
```

The ambiguity is especially pernicious in situations like this

```
if (n > 0)
for (i = 0; i < n; i++)
if (s[i] > 0) {
printf("...");
return i;
}
else /* WRONG */
printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the else with the inner if. This kind of bug can be hard to find; it's a good idea to use braces when there are nested ifs. By the way, notice that there is a semicolon after z = ain

```
if (a > b)
z = a;
```

else

z = b;

This is because grammatically, a statement follows the if, and an expression statement like ``z = a;'' is always terminated by a semicolon.

The construction

if (expression)

statement

else if (expression)

statement

else if (expression)

statement

else if (expression)

statement

else

statement

occurs so often that it is worth a brief separate discussion. This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces. The last else part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

else

statement

can be omitted, or it may be used for error checking to catch an ``impossible'' condition. To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. The elements of v must be in increasing order. The function returns the position (a number between 0 and n-1) if x occurs in v, and -1 if not. Binary search first compares the input value xto the middle element of the array v. If x is less than the middle value,

searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare xto the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```
/* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
    int binsearch(int x, int v[], int n)
    {
        int low, high, mid;
        low = 0;
        high = n - 1;
        while (low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
        high = mid + 1;
        else if (x > v[mid])
        low = mid + 1;
        else /* found match */
        return mid;
    }
    return -1; /* no match */
```

The fundamental decision is whether xis less than, greater than, or equal to the middle element v[mid]at each step; this is a natural for else-if. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside.) Write a version with only one test inside the loop and measure the difference in run-time.

Switch

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constantinteger values, and branches accordingly.

```
switch (expression) {
case const-expr: statements
case const-expr: statements
default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order. we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of if ... else if ... else. Here is the same program with a switch:

```
#include <stdio.h>
main() /* count digits, white space, others */
{
int c, i, nwhite, nother, ndigit[10];
nwhite = nother = 0;
for (i = 0; i < 10; i++)
ndigit[i] = 0;
while ((c = getchar()) != EOF) {
switch (c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9': ndigit[c-'0']++; break;
case ' ':
case '\n':
case '\t': nwhite++;break;
default: nother++;break;
}
}
printf("digits =");
for (i = 0; i < 10; i++)
```

```
printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",
nwhite, nother);
return 0;
}
```

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. Break and return are the most common ways to leave a switch. A break statement can also be used to force an immediate exit from while, for, and do loops, Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a break to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall throughs should be used sparingly, and commented. As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

ternary operator?

Go to, Loops (For, do-while, while) in C,

We have already encountered the while and for loops. In

```
while (expression)
statement
```

the expression is evaluated. If it is non-zero, statement is executed and expression is reevaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.

The for statement

```
for (expr1; expr2; expr3)
statement is equivalent to
expr1;
```

```
while (expr2) {
statement
expr3;
}
```

except for the behaviour of continue, which is described in Section 3.7.

Grammatically, the three components of a forloop are expressions. Most commonly, expr1and expr3 are assignments or function calls and expr2 is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If expr1or expr3 is omitted, it is simply dropped from the expansion. If the test, expr2, is not present, it is taken as permanently true, so

```
for (;;) {
...
}
```

is an ``infinite'' loop, presumably to be broken by other means, such as a break or return. Whether to use while or for is largely a matter of personal preference. For example, in

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
; /* skip white space characters */
```

there is no initialization or re-initialization, so the while is most natural. The for is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop. This is most obvious in

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

which is the C idiom for processing the first n elements of an array, the analog of the Fortran DO loop or the Pascal for. The analogy is not perfect, however, since the index variable i retains its value when the loop terminates for any reason. Because the components of the forare arbitrary expressions, for loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a for, which are better reserved for loop control operations.

```
#include <ctype.h>
/* atoi: convert s to integer; version 2 */
```

```
int atoi(char s[])
{
int i, n, sign;
for (i = 0; isspace(s[i]); i++) /* skip white space */
;
sign = (s[i] == '-') ? -1 : 1;
if (s[i] == '+' || s[i] == '-') /* skip sign */
i++;
for (n = 0; isdigit(s[i]); i++)
n = 10 * n + (s[i] - '0');
return sign * n;
}
```

The standard library provides a more elaborate function strtolfor conversion of strings to long integers. The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
/* shellsort: sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
int gap, i, j, temp;
for (gap = n/2; gap > 0; gap /= 2)
for (i = gap; i < n; i++)
for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
temp = v[j];
v[j] = v[j+gap];
v[j+gap] = temp;
}
```

```
}
```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from n/2by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by gap and reverses any that are out of order. Since gap is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the for makes the outer loop fit in the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma ``,'', which most often finds use in the for statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a for statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function reverse(s), which reverses the string sin place.

```
#include <string.h>
/* reverse: reverse string s in place */
void reverse(char s[])
{
int c, i, j;
for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
c = s[i];
s[i] = s[j];
s[j] = c;
}
}
```

The commas that separate function arguments, variables in declarations, etc., are not comma operators, and do not guarantee left to right evaluation. Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the for loop in reverse, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in reverse, where the exchange can be thought of a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
c = s[i], s[i] = s[j], s[j] = c;
```

Loops - Do-While

The while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once. The syntax of the

do is

```
do
statement
while (expression);
```

The statement is executed, then expression is evaluated. If it is true, statement is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, do-while is equivalent to the Pascal repeat-until statement. Experience shows that do-while is much less used than while and for. Nonetheless, from time to time it is valuable, as in the following function itoa, which converts a number to a character string (the inverse of atoi). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```
/* itoa: convert n to characters in s */
void itoa(int n, char s[])
{
int i, sign;
if ((sign = n) < 0) /* record sign */
n = -n; /* make n positive */
i = 0;
do { /* generate digits in reverse order */
s[i++] = n % 10 + '0'; /* get next digit */
} while ((n /= 10) > 0); /* delete it */
if (sign < 0)
s[i++] = '-';
s[i] = '\0';
reverse(s);
```

}

The do-whileis necessary, or at least convenient, since at least one character must be installed in the array s, even if nis zero. We also used braces around the single statement that makes up the body of the do-while, even though they are unnecessary, so the hasty reader will not mistake the while part for the beginning of a while loop.

Break and continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately. The following function, trim, removes trailing blanks, tabs and newlines from the end of a string, using a breakto exit from a loop when the rightmost non-blank, non-tab, non newline is found.

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
int n;
for (n = strlen(s)-1; n >= 0; n--)
if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
break;
s[n+1] = '\0';
return n;
}
```

Strlen returns the length of the string. The forloop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when n becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters. The continue statement is related to break, but less often used; it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration. As an example, this fragment processes only the non-negative elements in the array a; negative values are skipped.

```
for (i = 0; i < n; i++)
if (a[i] < 0) /* skip negative elements */
continue;
... /* do positive elements */
```

The continue statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

Go to and labels

C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto statement is never necessary, and in practice it is almost always easy to write code without it. Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... ) {
...
if (disaster)
goto error;
}
...
error:
/* clean up the mess */
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places. A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the goto. The scope of a label is the entire function. As another example, consider the problem of determining whether two arrays a and b have an element in common. One possibility is

```
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
if (a[i] == b[j])
```

```
        goto found;
        /* didn't find any common element */
        ...
        found:
        /* got one: a[i] == b[j] */
...
```

Code involving a gotocan always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```
        found = 0;
        for (i = 0; i < n && !found; i++)
        for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
        found = 1;
        if (found)
        /* got one: a[i-1] == b[j-1] */
        ...
        else
        /* didn't find any common element */
```

With a few exceptions like those cited here, code that relies on goto statements is generally harder to understand and to maintain than code without gotos. Although we are not dogmatic about the matter, it does seem that goto statements should be used rarely, if at all

MODULE III

ARRAYS, STRINGS AND FUNCTIONS

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]	57
Num[1]	20
Num[2]	56
Num[3]	17
Num[4]	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 [0][0]	Column1 [0][1]	Column2 [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
Row3	[3][0]	[3][1]	[3][2]
	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.

1. There is no convenient way to initialize only selected elements.

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

Reading Writing and manipulation of above types of arrays.

Program to read and write two dimensional arrays.

```
#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:

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("%2d|", 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
```

STRINGS

String variable:

A string is an array of characters. Any group of characters defined between double quotation marks is called a constant string.

Example:

"Good Morning Everybody"

Character strings are often used to build meaningful and readable programs.

A string variable is any valid C variable name and is always declared as an array.

Declaring and initializing string variables:

The general form of string variable is

char string_name[size];

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

Some examples are:

```
char   state[10];
char   name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a null character('\0') at the end of the string. Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
Static char state[10]=" KARNATAKA";
Static char state[10]={'K','A','R','N','A','T','A','K','A'\O'};
```

The reason that state had to be 10 elements long is that the string KARNATAKA contains 10 characters and one element space is provided for the null terminator. C also permits us to initialize a character array without specifying the number of elements.

For example, the statement

```
static char string[ ]    ={'H', 'E', 'L', 'L', 'O' \O};
```

Defines the array string as a six element array.

Reading and writing strings:

To read a string of characters input function scanf can be used with %s format specification.

Example:

```
char add[20];
Scanf("%s", add);
```

Note that unlike previous scanf calls, in the case of character arrays, the &(ampersand) is not required before the variable name. The scanf function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Program to read a series of words using scanf function

```
main()
{
```

```
    char text1[50],text2[50],text3[50],text4[50];
    printf("Enter   text:\n");
    scanf("%s  %s", text1,text2);
    scanf("%s", text3);
    scanf("%s", text4);
    printf("\n");
    printf("text1= %s\n text2=%s\n", text1,text2);
    printf("text3=  %s\n text4= %s\n", text3,text4);
}
```

Writing strings:

The printf function with %s can be used to display an array of characters that is terminated by the null character.

Example:

```
printf("%s",   text);
```

Can be used to display the entire contents of the array name. We can also specify the precision with which the array is displayed. For example, the specification

%12.4

indicates that the first four characters are to printed in a field width of 12 columns.

Program to illustrate writing strings using %s format

```
main()
{
        static char state[15]= "MADHYA PRADESH";
        printf("\n \n");
        printf("----------------------------------\n");
        printf("%13s\n", state);
        printf("%5s\n", state);
        printf("%15.6s \n", state);
        printf("%15.0s\n", state);
        printf("%.3s\n", state);
```

}

String functions:

C library supports a large number of string functions. The list given below depicts the string functions

Function	Action
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string with another
strlen()	finds the length of a string.

String Concatenation :strcat() function:

The strcat function joins two strings together. The general form is

strcat(string1,string2);

string1 and string2 are character arrays. When the function strcat is executed. String2 is appended to string1. It does so by removing the null character at the end of string1 and placing string2 from there. The string at string2 remains unchanged.

Example:

Text1= VERY \0

Text2= GOOD\0

Text3= BAD\0

Strcat(text1,text2);

Text1= VERY GOOD\0

Text2= GOOD\0

Strcat(text1,text3);

Text1= VERY BAD

Text2= BAD

We must make sure that the size of string1 is large enough to accommodate the final string. Strcat function may also append a string constant to string variable.

For example:

strcat(text1,"GOOD");

C permits nesting of strcat functions. The statement

strcat(strcat(string1,string2),string3);

Is allowed and concatenates all the three strings together. The resultant string is stored in string1.

String comparison/strcmp() function:

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal. The general form is :

strcmp(string1,string2);

String1 and string2 may be string variables or string constants.

Examples are:

strcmp(name1,name2);

strcmp(name1, "ABHI");

strcmp("ROM", "RAM");

We have to determine whether the strings are equal, if not which is alphabetically above.

String copying/strcpy() function:

The strcpy() function works almost like a string-assignment operator. The general format is

strcpy(string1,string2);

It copies the contents of string2 to string1. string2 may be a character variable or a string constant.

For example, the statement

strcpy(city , "BANGALORE");

Will assign the string "BANGALORE" to the string variable city. The statement strcpy(city1,city2); will assign the contents of the string variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2.

Finding the length of a string/strlen();

This function counts and returns the number of characters in a string.

The general syntax is n=strlen(string);

Where n is an integer variable which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character. Implementing the above functions without using string functions:

String concatenation:

We cannot assign one string to another directly,we cannot join two strings together by the simple arithmetic addition. The characters from string1 and string2 should be copied into the string3 one after the other. The size of the array string3 should be large enough to hold the total characters.

Program to show concatenation of strings:

```
main()
{
        int i,j,k;
        static char first_name={“ATAL”};
        static char sec_name={“RAM”};
        static char  last_name={“KRISHNA”};
        char name[30];

        for(i=0;first_name[i]!=’\0’;i++)
            name[i]=first_name;
        for(i=0;second_name[j]!=’\0’; j++)
            name[i+j+1]=sec_name[j];
             name[i+j+1] =’ ‘;
        for(k=0;last_name[k]!=’\0’;k++)
             name[i+j+k+2]=last_name[k];
             name[i+j+k+2]=’\0’;
              printf(“\n \n”);
               printf(“%s \n”, name);
}
```

Output

ATAL RAM KRISHNA

String comparison:

Comparison of two strings cannot be compared directly. It is therefore necessary to compare the strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character.
The following segment of a program illustrates this,

```
----------------------------
i=0;
while(str1[i]==str2[i] && str1[i]!='\0'
        && str2[i]!='\0')
        i=i+1;
  if(str1[i]=='\0' && str2[i]=='\0')
        printf("strings are equal\n");
    else
        printf("strings are not equal\n");
--------------------------------
```

String copying:

Program to show copying of two strings:

```
main()
 {
        char string1[80],string2[80];
        int j;
        printf("Enter a string\n");
        printf("?");
        scanf("%s", string2);
```

```
        for(j=0;string2[i]!='\0';j++)
          string1[j]=string2[j];
          string1[j]='\0';
          printf("\n");
          printf("%s\n",string1);
          printf("Number of characters=%d\n", j);
}
```

Program to find the length of a string:

```
#include<stdio.h>
main()
{       char line[80],character
        int c=0,i;
        printf("Enter the text\n");
        for(i=0;line[i];!='\0';i++)
{           character=getchar();
            line[i]=character;
            c++;
  }
   printf("The length of the string \n", c);
}
```

Arithmetic operations on characters:

We can manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation , we may write it as an integer. For example:

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

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables.

For example:

```
y='z'-1;
```

Is a valid statement. In ASCII , the value of 'z' is 122 and therefore , the statement will assign the value 121 to the variable Y. We may also use character constants in relational expressions. For example:

```
ch>='a' && ch<='z'
```

Would test whether the character contained in the variable ch is an lower-case letter. We can convert a character digit to its equivalent integer value using the following relationship :

```
y=character –'0';
```

Where y is defined as an integer variable and character contains the character digit.

Example: Let us assume that the character contains the digit '7', then,

```
y=ASCII value of '7'-ASCII value of '0'
  = 55-48
  =7
```

C library has a function that converts a string of digits into their integer values. The function takes the form

```
y=atoi(string);
```

y is an integer variable and string is a character array containing a string or digits

Consider the following example:

```
num="1974"
year=atoi(num);
```

Num is a string variable which is assigned the string constant "1974". The function atoi converts the string "1974" to its numeric equivalent 1974 and assigns it to the integer variable year.

Programming examples:

Program to sort strings in alphabetical order:

```
#define ITEMS 10
#define MAX    25
main()
{
        char str [ITEMS][MAX], dum[MAX];
        int i=0;j=0;
        printf("Enter names of %d items \n", ITEMS);
        while(i<ITEMS)
        scanf("%s", str[i++]);
        for(i=1;i<ITEMS;i++)
        {
        for(j=1;j<=ITEMS-i;j++)
         {
           if(strcmp(string[j-1],string[j])>0)
                 strcpy(dummy,string[j-1]);
                 strcpy(str[j-1],str[j]);
                 strcpy(str[j],dummy);
        }
        }
        for(i=0;i<ITEMS;i++)
          printf("%s", str[i]);
}
```

Program to show string handling functions:

```
        #include<string.h>
        main()
        {
          char s1[20],s2[20],s3[20];
           int y,len1,len2,len3;
           printf("\n Enter two string constants\n");
           printf("?");
           scanf("%s %s", s1,s2);
```

```
    x=strcmp(s1,s2);
     If(y!=0)
        {
          printf("\n\n Strings are not equal\n");
           strcat(s1,s2);
        }
   else
        printf("\n\n Strings are equal\n");
        strcpy(s3,s1);
        len1=strlen(s1);
        len2=strlen(s2);
        len3=strlen(s3);
        printf("\n s1= %s length= %d character \n",s1,len1);
        printf("\n s1= %s length= %d character \n",s2,len2);
         printf("\n s1= %s length= %d character \n",s3,len3);
        }
```

Program to convert lowercase characters in to upper case characters:

```
#include<stdio.h>
main()
{
        char text[85];
        int i=0;
        printf("Enter a line of text in lowercase:\t");
        scanf("%[^\n]",text);
        printf("%s",text);
        printf("\n Converted to uppercase text is :\t");
        while(text[i]!='0')
          printf("%c", toupper(text[i]));
           i++;
        }
              printf("\n");
```

}

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes. C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system. Function declaration and definition is the area where the ANSI standard has made the most changes to C. possible to declare the type of arguments when a function is declared. The syntax of function declaration also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically. The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized. The C preprocessor has also been enhanced. New preprocessor facilities include a more complete set of conditional compilation directives, a way to create quoted strings from macro arguments, and better control over the macro expansion process. To begin with, let us design and write a program to print each line of its input that contains a particular ``pattern'' or string of characters. (This is a special case of the UNIX program grep.) For example, searching for the pattern of letters ``ould'' in the set of lines

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */
int getline(char line[], int max)
int strindex(char source[], char searchfor[]);
char pattern[] = "ould"; /* pattern to search for */
/* find all lines matching pattern */
main()
```

```
{
char line[MAXLINE];
int found = 0;
while (getline(line, MAXLINE) > 0)
if (strindex(line, pattern) >= 0) {
printf("%s", line);
found++;
}
return found;
}
/* getline: get line into s, return length */
int getline(char s[], int lim)
{
int c, i;
i = 0;
while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
s[i++] = c;
if (c == '\n')
s[i++] = c;
s[i] = '\0';
return i;
}
/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
int i, j, k;
for (i = 0; s[i] != '\0'; i++) {
for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
;
if (k > 0 && t[k] == '\0')
return i;
```

```
}
return -1;
}
```

Each function definition has the form return-type function-name(argument declarations)

```
{
declarations and statements
}
```

Various parts may be absent; a minimal function is dummy() {}which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, int is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split. The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return: return expression;

The expression will be converted to the return type of the function if necessary. Parentheses are often used around the expression, but they are optional. The calling function is free to ignore the returned value. Furthermore, there need to be no expression after return; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution ``falls off the end" of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its ``value" is certain to be garbage. The pattern-searching program returns a status from main, the number of matches found. This value is available for use by the environment that called the program

Functions Returning Non-integers

First, atofitself must declare the type of value it returns, since it is not int. The type name precedes the function name:

```
#include <ctype.h>
/* atof: convert string s to double */
```

```
double atof(char s[])
{
double val, power;
int i, sign;
for (i = 0; isspace(s[i]); i++) /* skip white space */
;
sign = (s[i] == '-') ? -1 : 1;
if (s[i] == '+' || s[i] == '-')
i++;
for (val = 0.0; isdigit(s[i]); i++)
val = 10.0 * val + (s[i] - '0');
if (s[i] == '.')
i++;
for (power = 1.0; isdigit(s[i]); i++) {
val = 10.0 * val + (s[i] - '0');
power *= 10;
}
return sign * val / power;
}
```

Second, and just as important, the calling routine must know that atofreturns a non-int value. One way to ensure this is to declare atof explicitly in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded with a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>
#define MAXLINE 100

/* rudimentary calculator */
main()
{
double sum, atof(char []);
```

```
char line[MAXLINE];
int getline(char line[], int max);
sum = 0;
while (getline(line, MAXLINE) > 0)
printf("\t%g\n", sum += atof(line));
return 0;
}
```

The declaration

double sum, atof(char []); says that sum is a double variable, and that atof is a function that takes one char[]argument and returns a double. The function atof must be declared and defined consistently. If atof itself and the call to it in main have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) atof were compiled separately, the mismatch would not be detected, atof would return a double that main would treat as an int, and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

sum += atof(line)

If a name that has not been previously declared occurs in an expression and is followed by a left parentheses, it is declared by context to be a function name, the function is assumed to return an int, and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

The structure of the program is thus a loop that performs the proper operation on each operator and operand as it appears:

```
while (next operator or operand is not end-of-file indicator)
if (number)
push it
else if (operator)
pop operands
do operation
```

push result

else if (newline)

pop and print top of stack

else

error

The operation of pushing and popping a stack are trivial, but by the time error detection and recovery are added, they are long enough that it is better to put each in a separate function than to repeat the code throughout the whole program. And there should be a separate function for fetching the next input operator or operand. The main design decision that has not yet been discussed is where the stack is, that is, which routines access it directly. On possibility is to keep it in main, and pass the stack and the current stack position to the routines that push and pop it. But main doesn't need to know about the variables that control the stack; it only does push and pop operations. So we have decided to store the stack and its associated information in external variables accessible to the push and pop functions but not to main. Translating this outline into code is easy enough. If for now we think of the program as existing in one source file, it will look like this:

```
#includes
#defines
function declarations for main
main() { ... }
external variables for push and pop
void push( double f) { ... }
double pop(void) { ... }
int getop(char s[]) { ... }
routines called by getop
```

Later we will discuss how this might be split into two or more source files. The function main is a loop containing a big switch on the type of operator or operand; this is a more typical use of switch than the one

```
#include <stdio.h>
#include <stdlib.h> /* for atof() */
#define MAXOP 100 /* max size of operand or operator */
```

```
#define NUMBER '0' /* signal that a number was found */
int getop(char []);
void push(double);
double pop(void);
/* reverse Polish calculator */
main()
{
int type;
double op2;
char s[MAXOP];
while ((type = getop(s)) != EOF) {
switch (type) {
case NUMBER:
push(atof(s));
break;
case '+': push(pop() + pop());
        break;
case '*':push(pop() * pop());
        break;
case '-':op2 = pop();
        push(pop() - op2);
        break;
case '/':op2 = pop();
        if (op2 != 0.0)
        push(pop() / op2);
        else
                printf("error: zero divisor\n");
        break;
case '\n': printf("\t%.8g\n", pop());
        break;
default:
```

```
                printf("error: unknown command %s\n", s);
        break;
}
}
return 0;
}

#define MAXVAL 100 /* maximum depth of val stack */
int sp = 0; /* next free stack position */
double val[MAXVAL]; /* value stack */
/* push: push f onto value stack */
void push(double f)
{
if (sp < MAXVAL)
val[sp++] = f;
else
printf("error: stack full, can't push %g\n", f);
}
/* pop: pop and return top value from stack */
double pop(void)
{
if (sp > 0)
return val[--sp];
else {
printf("error: stack empty\n");
return 0.0;
}
}
```

A variable is external if it is defined outside of any function. Thus the stack and stack index that must be shared by push and pop are defined outside these functions. But main itself does not refer to the stack or stack position - the representation can be hidden. Let us now turn to the

implementation of getop, the function that fetches the next operator or operand. The task is easy. Skip blanks and tabs. If the next character is not a digit or a hexadecimal point, return it. Otherwise, collect a string of digits (which might include a decimal point), and return NUMBER, the signal that a number has been collected.

```
#include <ctype.h>
int getch(void);
```

Because +and *are commutative operators, the order in which the popped operands are combined is irrelevant, but for -and /the left and right operand must be distinguished.

In

push(pop() - pop()); /* WRONG */ the order in which the two calls of pop are evaluated is not defined. To guarantee the right order, it is necessary to pop the first value into a temporary variable as we did in main.

```
void ungetch(int);
/* getop: get next character or numeric operand */
int getop(char s[])
{
int i, c;
while ((s[0] = c = getch()) == ' ' || c == '\t')
;
s[1] = '\0';
if (!isdigit(c) && c != '.')
return c; /* not a number */
i = 0;
if (isdigit(c)) /* collect integer part */
while (isdigit(s[++i] = c = getch()))
;
if (c == '.') /* collect fraction part */
while (isdigit(s[++i] = c = getch()))
;
```

```
s[i] = '\0';
if (c != EOF)
ungetch(c);
return NUMBER;
}
```

What are getchand ungetch? It is often the case that a program cannot determine that it has read enough input until it has read too much. One instance is collecting characters that make up a number: until the first non-digit is seen, the number is not complete. But then the program has read one character too far, a character that it is not prepared for.

The problem would be solved if it were possible to ``un-read" the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read. Fortunately, it's easy to simulate ungetting a character, by writing a pair of cooperating functions. getchdelivers the next input character to be considered; ungetch will return them before reading new input. How they work together is simple. ungetchputs the pushed-back characters into a shared buffer -- a character array. getchreads from the buffer if there is anything else, and calls getcharif the buffer is empty. There must also be an index variable that records the position of the current character in the buffer. Since the buffer and the index are shared by getchand ungetchand must retain their values between calls, they must be external to both routines. Thus we can write getch, ungetch, and their shared variables as:

```
#define BUFSIZE 100
char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */
int getch(void) /* get a (possibly pushed-back) character */
{
return (bufp > 0) ? buf[--bufp] : getchar();
}
void ungetch(int c) /* push character back on input */
{
if (bufp >= BUFSIZE)
printf("ungetch: too many characters\n");
```

```
else
buf[bufp++] = c;}
```

Static Variables

The variables spand valin stack.c, and buf and bufpin getch.c, are for the private use of the functions in their respective source files, and are not meant to be accessed by anything else. The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External static thus provides a way to hide names like buf and bufpin the getch ungetch combination, which must be external so they can be shared, yet which should not be visible to users of getch and ungetch. Static storage is specified by prefixing the normal declaration with the word static. If the two routines and the two variables are compiled in one file, as in

```
static char buf[BUFSIZE]; /* buffer for ungetch */
static int bufp = 0; /* next free position in buf */
int getch(void) { ... }
void ungetch(int c) { ... }
```

then no other routine will be able to access bufand bufp, and those names will not conflict with the same names in other files of the same program. In the same way, the variables that push and pop use for stack manipulation can be hidden, by declaring sp and valto be static. The external static declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared static, however, its name is invisible outside of the file in which it is declared.

The static declaration can also be applied to internal variables. Internal static variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal static variables provide private, permanent storage within a single function.

Register Variables

A register declaration advises the compiler that the variable in question will be heavily used. The idea is that register variables are to be placed in machine registers, which may result in

smaller and faster programs. But compilers are free to ignore the advice. The register declaration looks like

register int x;

register char c;

and so on. The registerdeclaration can only be applied to automatic variables and to the formal parameters of a function. In this later case, it looks like

```
f(register unsigned m, register long n)
{
register int i;
...
}
```

In practice, there are restrictions on register variables, reflecting the realities of underlying hardware. Only a few variables in each function maybe kept in registers, and only certain types are allowed. Excess register declarations are harmless, however, since the word register is ignored for excess or disallowed declarations.

Initialization has been mentioned in passing many times so far, but always peripherally to some other topic. This section summarizes some of the rules, now that we have discussed the various storage classes. In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic andregister variables have

undefined (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int x = 1;

char squota = '\'';

long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptionally before the program begins execution. For automatic and register

variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

```
int binsearch(int x, int v[], int n)
{
int low = 0;
int high = n - 1;
int mid;
...
}
```

instead of

```
int low, high, mid;
low = 0;
high = n - 1;
```

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely amatter of taste. We have generally used explicit assignments, because initializers in declarations are harder to see and further away from the pointof use. An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas. For example, to initialize an array days with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case. If there are fewer initializers for an array than the specified size, the others will be zero for external, static and automatic variables. It is an error to have too many initializers. There is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the preceding values as well. Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char pattern = "ould";
```

is a shorthand for the longer but equivalent

char pattern[] = { 'o', 'u', 'l', 'd', '\0' }; In this case, the array size is five (four characters plus the terminating '\0').

MODULE-IV

Structures and File Managements

Array is used to represent a group of data items that belongs to same data-type, such as int or float. However, if we want to represent a collection of data items of different data-types using a single name, then we cannot use an array. For that C supports constructed data type known as structure.

"A structure is a used-defined data type, which contains group of multiple different data type related variable."

We can define the structure by following format:

```
struct tag_name  {
datatype element_1;
datatype element_2;
…
datatype element_n;
};
```

In above syntax, 'struct' keyword declares a structure holds multiple different data type variable. These variables are known as the **structure elements** or **members of structure**. Each member may have same or different data type. The name of the structure is known as **tagname**. In above format there is no any declaration is made but it defines only the format of structure variable so it is known as **structure template**. Structure variable can be declared as following way:

```
struct tag_name      variable_1, variable_2, … , variable_n;
```

```
e.g.        struct student {                        // structure template
int rollno;
char name[100];
char address[100];
```

```
};
struct student s1,s2,s3;                // structure declaration
```

It is also allows both structure declaration and template creation in one statement:

```
struct      tag_name {
datatype    member_1;
datatype    member_2;
datatype    member_3;
…
datatype    member_n;
}       variable_1, variable_2, variable_3;
```

Here both the task, one is structure template definition and variable declaration is made in one statement. 'variable_1', 'variable_2', 'variable_3' are structure variable after declaration.

During defining structure we have to consider following points:

1. The structure template is terminated with semicolon (;);
2. During structure template definition, no any memory location is made for structure elements.
3. The total size of structure variable is equal to sum of individual size of structure elements.

We can define structure out of any function or inside function. When we define structure out of any function the structure definition is global and can be used by any other function.

Accessing structure elements for assignment, print, and read

We can access structure elements by using **member operator '.'**, which also known as **dotoperator** or **period operator**.

```
e.g.    struct student {                    // structure template
int rollno;
char name[100];
char address[100];
```

```
};

struct student s1;                // structure declaration

main( ) {
s1.rollno = 1;
strcpy (s1.name, "Shantilal");
strcpy (s1.address, "Ahmedabad");
}
```

Here we have assign the rollno, name, and address of student by using dot operator. Similarly we can read structure elements and print it.

```
scanf(" %d", &s1.rollno);
scnaf(" %s", s1.name);
printf(" %d\n", s1.rollno);
printf(" %s\n", s1.name);
```

Initialization of structure: -

We can initialize structure as like as array by specifying list of values in curly bracket. The format of initialization is:

```
struct  variable_1= { list of values separated by comma };
 e.g.        struct student {                        // structure template
int rollno;
char name[100];
char address[100];
};

main( ) {
struct student s1={1,"Shantilal", "Ahmedabad" };          // Initialization
}
```

In above example we have initialized the structure elements by separating with comma in curly bracket. We assigned values 1, Shantilal, Ahmedabad to rollno, name, and address respectively.

But we cannot initialize the structure element during structure definition.

```
e.g.        struct student {
int rollno=0;              // Gives compilation error
}
```

ANSI C standard allows the initialization of **auto** storage class variable but non ANSI compiler allows only initialization of **static** and **extern** storage class variable. So for that we must write static or extern before declaration.

```
static   struct  variable_1= { list of values separated by comma };
// For non ANSI compiler
```

Array of Structure

As we known that structure is a group of related different data type variables. Sometimes we need multiple variables of structure for that we can use the array of structure. For example we have structure of student. We want the list of students of entire class but the one variable of structure can represent only one student. So to represent entire class variable we have to use array of student structure.

We can declare the array of structure as simply as other data type array. And same way to other data type we can use this variable with the subscript.

```
struct  arrayname[size];
e.g.        struct student {                    // structure template
int rollno;
char name[100];
char address[100];
};

main( ) {
struct student s[10];
int i;
printf("Enter 10 student data : \n");
```

```
for(i=0 ; i < 10 ; i ++) {
scanf("%d", &s[i].rollno);
scanf("%s", s[i].name);
scanf("%s", s[i].address);
}
}
```

We can initialize the array of structure as like as the two dimensional array:

struct structurename variablename[size] = {{list of values of 1st element},
{list of values of 2nd element} , …. }

e.g. struct student s[2] = {{1, "Shantilal", "Ahmedabad"}, {2, "Mulji","Bhadara"}};

Structure Data Type

Object conepts was derived from Structure concept. You can achieve few object oriented goals using C structure but it is very complex.

Following is the example how to define a structure.

```
struct student {

      char firstName[20];
      char lastName[20];
      char SSN[9];
      float gpa;
    };
```

Now you have a new datatype called student and you can use this datatype define your variables of student type:

struct student student_a, student_b; or an array of students as

struct student students[50];

Another way to declare the same thing is:

```
struct {
   char firstName[20];
   char lastName[20];
   char SSN[10];
```

```
    float gpa;
} student_a, student_b;
```

All the variables inside an structure will be accessed using these values as student_a.firstNamewill give value of firstName variable. Similarly we can aqccess other variables.

SSN : 2333234
GPA : 2009.20

Type Definition

There is an easier way to define structs or you could "alias" types you create. For example:

typedef struct{

char firstName[20];

char lastName[20];

char SSN[10];

float gpa;

}student;

Now you can use student directly to define variables of student type without using struct keyword. Following is the example:

student student_a;

You can use typedef for non-structs:

typedef long int *pint32;

pint32 x, y, z;

x, y and z are all pointers to long ints.

Defining Opening and Closing of files

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. For Example:

FILE *fp;

To open a file you need to use the fopen function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

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

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

w - open for writing (file need not exist)

a - open for appending (file need not exist)

r+ - open for reading and writing, start at beginning

w+ - open for reading and writing (overwrite file)

a+ - open for reading and writing (append if file exists)

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:

```
FILE *fp;
fp=fopen("/home/tutorialspoint/test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

To close a function you can use the function:

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

fclose returns zero if the file is closed successfully.

An example of fclose is:

```
fclose(fp);
```

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument.

Try out following example:

```
#include <stdio.h>
main()
{
  FILE *fp;
  fp = fopen("/tmp/test.txt", "w");
  fprintf(fp, "This is testing...\n");
  fclose(fp;);
}
```

This will create a file test.txt in /tmp directory and will write This is testing in that file.

Here is an example which will be used to read lines from a file:

```
#include <stdio.h>
main()
{
  FILE *fp;
  char buffer[20];
  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buffer);
  printf("Read Buffer: %s\n", %buffer );
  fclose(fp;);
}
```

It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input. The fgetc function, which takes a file pointer, and returns an int, will let you read a single character from a file:

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

The fgetc returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constnat that indicates that you've reached the end of the file.

The fputc function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:

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

Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary I/O

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

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

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

Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

Input and Output Operations

Input : In any programming language input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Output : In any programming language output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

Here we will discuss only one input function and one putput function just to understand the meaning of input and output. Rest of the functions are given into C - Built-in Functions

printf() function

This is one of the most frequently used functions in C for output. (we will discuss what is function in subsequent chapter.).

Try following program to understand printf() function.

```
#include <stdio.h>
main()
{
  int dec = 5;
  char str[] = "abc";
  char ch = 's';
  float pi = 3.14;
  printf("%d %s %f %c\n", dec, str, pi,  ch);
}
```

The output of the above would be:

```
5 abc 3.140000 c
```

Here %d is being used to print an integer, %s is being usedto print a string, %f is being used to print a float and %c is being used to print a character.

A complete syntax of printf() function is given in C - Built-in Functions

scanf() function

This is the function which can be used to to read an input from the command line.

MODULE V

Pointers and Pre Processors

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps. As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>
int main ()
{
  int  var1;
  char var2[10];
  printf("Address of var1 variable: %x\n", &var1  );
  printf("Address of var2 variable: %x\n", &var2  );
  return 0;
}
```

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

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

What Are Pointers?

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

type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

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

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

How to use Pointers?

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

```
#include <stdio.h>
int main ()
{
  int  var = 20;   /* actual variable declaration */
  int  *ip;        /* pointer variable declaration */
  ip = &var;  /* store address of var in pointer variable*/
  printf("Address of var variable: %x\n", &var  );
  /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );
```

```
  /* access the value using the pointer */
  printf("Value of *ip variable: %d\n", *ip );
  return 0;
}
```

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

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

NULL Pointers in C

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

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

```
#include <stdio.h>
int main ()
{
  int  *ptr = NULL;
  printf("The value of ptr is : %x\n", ptr  );
   return 0;
}
```

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

```
The value of ptr is 0
```

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

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

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

if(!ptr) /* succeeds if p is null */

C Pointers in Detail:

Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:

Concept	Description
C - Pointer arithmetic	There are four arithmetic operators that can be used on pointers: ++, --, +, -
C - Array of pointers	You can define arrays to hold a number of pointers.
C - Pointer to pointer	C allows you to have pointer on a pointer and so on.
Passing pointers to functions in C	Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
Return pointer from functions in C	C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

Pointers and functional arguments

One of the best things about pointers is that they allow functions to alter variables outside of there own scope. By passing a pointer to a function you can allow that function to read *and write* to the data stored in that variable. Say you want to write a function that swaps the values of two variables. Without pointers this would be practically impossible, here's how you do it with pointers:

Example swap_ints.c

```
#include <stdio.h>

int swap_ints(int *first_number, int *second_number);

int
main()
{
  int a = 4, b = 7;
```

```
printf("pre-swap values are: a == %d, b == %d\n", a, b)

  swap_ints(&a, &b);

  printf("post-swap values are: a == %d, b == %d\n", a, b)

  return 0;
}

int
swap_ints(int *first_number, int *second_number)
{
  int temp;

  /* temp = "what is pointed to by" first_number; etc... */
  temp = *first_number;
  *first_number = *second_number;
  *second_number = temp;

  return 0;
}
```

As you can see, the function declaration of **swap_ints()** tells GCC to expect two pointers (address of variables). Also, the *address-of* operator (**&**) is used to pass the address of the two variables rather than their values. **swap_ints()** then reads

Pointers and arrays

One of the best things about pointers is that they allow functions to alter variables outside of there own scope. By passing a pointer to a function you can allow that function to read *and write* to the data stored in that variable. Say you want to write a function that swaps the values of two variables. Without pointers this would be practically impossible, here's how you do it with pointers:

Example 5-2. swap_ints.c

```
#include <stdio.h>
int swap_ints(int *first_number, int *second_number);
int
main()
{
```

```
  int a = 4, b = 7;
  printf("pre-swap values are: a == %d, b == %d\n", a, b)
  swap_ints(&a, &b);
  printf("post-swap values are: a == %d, b == %d\n", a, b)
  return 0;
}
Int swap_ints(int *first_number, int *second_number)
{
  int temp;
  /* temp = "what is pointed to by" first_number; etc... */
  temp = *first_number;
  *first_number = *second_number;
  *second_number = temp;
  return 0; }
```

As you can see, the function declaration of **swap_ints()** tells GCC to expect two pointers (address of variables). Also, the *address-of* operator (**&**) is used to pass the address of the two variables rather than their values. **swap_ints()** then reads

Address Arithmetic

C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

ptr++

Now, after the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual

value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < MAX; i++)
  {
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

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

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  /* let us have array address in pointer */
  ptr = &var[MAX-1];
  for ( i = MAX; i > 0; i--)
  {
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

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

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;
  while ( ptr <= &var[MAX - 1] )
  {
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* point to the previous location */
    ptr++;
    i++;
  }
  return 0;
}
```

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

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Character Pointer and Functions

C Constant Pointer and Pointer to Constant

As a developer, you should understand the difference between constant pointer and pointer to constant.

C Constant pointer

A pointer is said to be constant pointer when the address its pointing to cannot be changed.

Lets take an example:

char ch, c;

char *ptr = &ch

ptr = &c

In the above example we defined two characters (‘ch’ and ‘c’) and a character pointer ‘ptr’. First, the pointer ‘ptr’ contained the address of ‘ch’ and in the next line it contained the address of ‘c’. In other words, we can say that Initially ‘ptr’ pointed to ‘ch’ and then it pointed to ‘c’. But in case of a constant pointer, once a pointer holds an address, it cannot change it. This means a constant pointer, if already pointing to an address, cannot point to a new address. If we see the example above, then if ‘ptr’ would have been a constant pointer, then the third line would have not been valid.

A constant pointer is declared as :

<type-of-pointer> *const <name-of-pointer>

For example :

```
#include<stdio.h>
int main(void)
{
   char ch = 'c';
   char c = 'a';
   char *const ptr = &ch; // A constant pointer
   ptr = &c; // Trying to assign new address to a constant pointer. WRONG!!!!
   return 0;
}
```

When the code above is compiled, compiler gives the following error :

$ gcc -Wall constptr.c -o constptr

constptr.c: In function ‘main’:

constptr.c:9: error: assignment of read-only variable ‘ptr’

So we see that, as expected, compiler throws an error since we tried to change the address held by constant pointer.

Now, we should be clear with this concept. Lets move on.

C Pointer to Constant

This concept is easy to understand as the name simplifies the concept. Yes, as the name itself suggests, this type of pointer cannot change the value at the address pointed by it.

Lets understand this through an example :

```
char ch = 'c';
char *ptr = &ch
*ptr = 'a';
```

In the above example, we used a character pointer ‘ptr’ that points to character ‘ch’. In the last line, we change the value at address pointer by ‘ptr’. But if this would have been a pointer to a constant, then the last line would have been invalid because a pointer to a constant cannot change the value at the address its pointing to.

A pointer to a constant is declared as :

```
const <type-of-pointer> *<name-of-pointer>;
```

For example :

```
#include<stdio.h>
int main(void)
{
   char ch = 'c';
   const char *ptr = &ch; // A constant pointer 'ptr' pointing to 'ch'
   *ptr = 'a';// WRONG!!! Cannot change the value at address pointed by 'ptr'.

   return 0;
}
```

When the above code was compiled, compiler gave the following error :

$ gcc -Wall ptr2const.c -o ptr2const

ptr2const.c: In function ‘main’:

ptr2const.c:7: error: assignment of read-only location ‘*ptr’

So now we know the reason behind the error above ie we cannot change the value pointed to by a constant pointer.

C Pointer to Pointer

We have used or learned pointer to a data type like character, integer etc. But in this section we will learn about pointers pointing to pointers. As the definition of pointer says that its a special variable that can store the address of an other variable. Then the other variable can very well be a pointer. This means that its perfectly legal for a pointer to be pointing to another pointer. Lets suppose we have a pointer ‘p1′ that points to yet another pointer ‘p2′ that points to a character ‘ch’. In memory, the three variables can be visualized as : So we can see that in memory, pointer p1 holds the address of pointer p2. Pointer p2 holds the address of character ‘ch’. So ‘p2′ is pointer to character ‘ch’, while ‘p1′ is pointer to ‘p2′ or we can also say that ‘p2′ is a pointer to pointer to character ‘ch’.

Now, in code ‘p2′ can be declared as :

char *p2 = &ch;

But ‘p1′ is declared as :

char **p1 = &p2;

So we see that ‘p1′ is a double pointer (ie pointer to a pointer to a character) and hence the two *s in declaration.

Now,

‘p1′ is the address of ‘p2′ ie 5000

‘*p1′ is the value held by ‘p2′ ie 8000

‘**p1′ is the value at 8000 ie ‘c’

```
#include<stdio.h>
int main(void)
{
   char **ptr = NULL;
```

```
    char *p = NULL;
    char c = 'd';
    p = &c;
    ptr = &p;
    printf("\n c = [%c]\n",c);
    printf("\n *p = [%c]\n",*p);
    printf("\n **ptr = [%c]\n",**ptr);
    return 0;
}
```

Here is the output :

```
$ ./doubleptr
 c = [d]
 *p = [d]
 **ptr = [d]
```

Introduction to Preprocessors

Preprocessor Compiler Control

You can use the cc compiler to control what values are set or defined from the command line. This gives some flexibility in setting customised values and has some other useful functions. The -D compiler option is used. For example:

```
cc -DLINELENGTH=80 prog.c -o prog
```

has the same effect as:

```
  #define LINELENGTH 80
```

Note that any #define or #undef **within** the program (prog.c above) **override** command line settings.

You can also set a symbol without a value, for example:

```
  cc -DDEBUG prog.c -o prog
```

Here the value is assumed to be 1.

The setting of such flags is useful, especially for debugging. You can put commands like:

```
#ifdef DEBUG
print("Debugging: Program Version 1\");
```

```
#else
print("Program Version 1 (Production)\");
#endif
```

Also since preprocessor command can be written anywhere in a C program you can filter out variables etc for printing ***etc.*** when debugging:

```
x = y *3;
#ifdef DEBUG
print("Debugging: Variables (x,y) = \",x,y);
#endif
```

The -E command line is worth mentioning just for academic reasons. It is not that practical a command. The -E command will force the compiler to stop after the preprocessing stage and output the current state of your program. Apart from being debugging aid for preprocessor commands and also as a useful initial learning tool (try this option out with some of the examples above) it is not that commonly used.

A data type is a classification of data, which can store a specific type of information. Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain.

The term "data type" and "primitive data type" are often used interchangeably. Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types. Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "lastname" and define it as a string data type. The variable will then store data as a string of characters.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data.

A **stack** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the **top** of the stack. A stack is a dynamic, constantly

changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place. The last element inserted into the stack is the first element deleted-**last in first out list (LIFO).** After several insertions and deletions, it is possible to have the same frame again.

Primitive Operations

When an item is added to a stack, it is **push**ed onto the stack. When an item is removed, it is **pop**ped from the stack.

Given a stack s, and an item i, performing the operation push(s,i) adds an item i to the top of stack s.

push(s, H);

push(s, I);

push(s, J);

Operation pop(s) removes the top element. That is, if i=pop(s), then the removed element is assigned to i.

pop(s);

Because of the push operation which adds elements to a stack, a stack is sometimes called a **pushdown list.** Conceptually, there is no upper limit on the number of items that may be kept in a stack. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the **empty stack.** Push operation is applicable to any stack. Pop operation cannot be applied to the empty stack. If so, underflow happens. A Boolean operation empty(s), returns TRUE if stack is empty. Otherwise FALSE, if stack is not empty.

Queues:

A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.

When we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed

• New people must enter the queue at the rear. **Push**, although it is usually called an **enqueue** operation.

• When an item is taken from the queue, it always comes from the front. **pop**, although it is usually called a **dequeue** operation.

What is Queue?

• Ordered collection of elements that has two ends as front and rear.

• Delete from front end

• Insert from rear end

• A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

Queue Operations

• Queue Overflow

• Insertion of the element into the queue

• Queue underflow

• Deletion of the element from the queue

• Display of the queue

Linked Lists

During implementation, overflow occurs. No simple solution exists for more stacks and queues. In a sequential representation, the items of stack or queue are implicitly ordered by the sequential order of storage.

If the items of stack or queue are explicitly ordered, that is, each item contained within itself the address of the next item. Then a new data structure known as linear linked list. Each item in the list is called a node and contains two fields, an information field and a next address field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a pointer.The null pointer is used to signal the end of a list. The list with no nodes – empty listor null list. The notations used in algorithms are:If p is a pointer to a node, node(p) refers to the node pointed to by p. Info(p) refersto the information of that node. next(p) refers to next address portion. If next(p) is notnull, info(next(p)) refers to the information portion of the node that follows node(p) inthe list.

A linked list (or more clearly, "singly linked list") is a data structure that consists of a sequence of nodes each of which contains a reference (i.e., a link) to the next node in the sequence.

A linked list whose nodes contain two fields: an integer value and a link to the next node. Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

Trees

A tree is a finite set of one or more nodes such that: (i) there is a specially designated node called the root; (ii) the remaining nodes are partitioned into n 0 disjoint sets T_1, ...,T_n where each of these sets is a tree. T_1, ...,T_n are called the subtrees of the root. A tree structure means that the data is organized so that items of information are related by branches. One very common place where such a structure arises is in the investigation of genealogies.

AbstractDataType tree{

instances

A set of elements:

(1) empty or having a distinguished root element

(2) each non-root element having exactly one parent element operations

root()

degree()

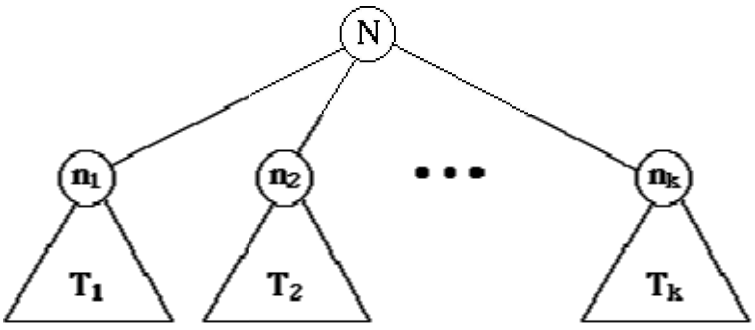
child(k)

}

Some basic terminology for trees:

- Trees are formed from nodes and edges. Nodes are sometimes called vertices. Edges are sometimes called branches.
- Nodes may have a number of properties including value and label.
- Edges are used to relate nodes to each other. In a tree, this relation is called "parenthood."
- An edge {a,b} between nodes a and b establishes a as the parent of b. Also, b is called a child of a.
- Although edges are usually drawn as simple lines, they are really directed from parent to child. In tree drawings, this is top-to-bottom.
- **Informal Definition**: a tree is a collection of nodes, one of which is distinguished as "root," along with a relation ("parenthood") that is shown by edges.
- **Formal Definition**: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.

1. A single node is a tree. It is "root."
2. Suppose N is a node and T_1, T_2, ..., T_k are trees with roots n_1, n_2, ...,n_k, respectively. We can construct a new tree T by making N the parent of the nodes n_1, n_2, ..., n_k. Then, N is the root of T and T_1, T_2, ..., T_k are subtrees.



The tree T, constructed using k subtrees

More terminology

- A node is either internal or it is a leaf.
- A leaf is a node that has no children.
- Every node in a tree (except root) has exactly one parent.
- The degree of a node is the number of children it has.
- The degree of a tree is the maximum degree of all of its nodes.
- **Paths and Levels**

- **Definition**: A path is a sequence of nodes n_1, n_2, ..., n_k such that node n_i is the parent of node n_{i+1} for all 1 <= i <= k.
- **Definition**: The length of a path is the number of edges on the path (one less than the number of nodes).
- **Definition**: The descendents of a node are all the nodes that are on some path from the node to any leaf.
- **Definition**: The ancestors of a node are all the nodes that are on the path from the node to the root.
- **Definition**: The depth of a node is the length of the path from root to the node. The depth of a node is sometimes called its level.
- **Definition**: The height of a node is the length of the longest path from the node to a leaf.
- **Definition**: the height of a tree is the height of its root.