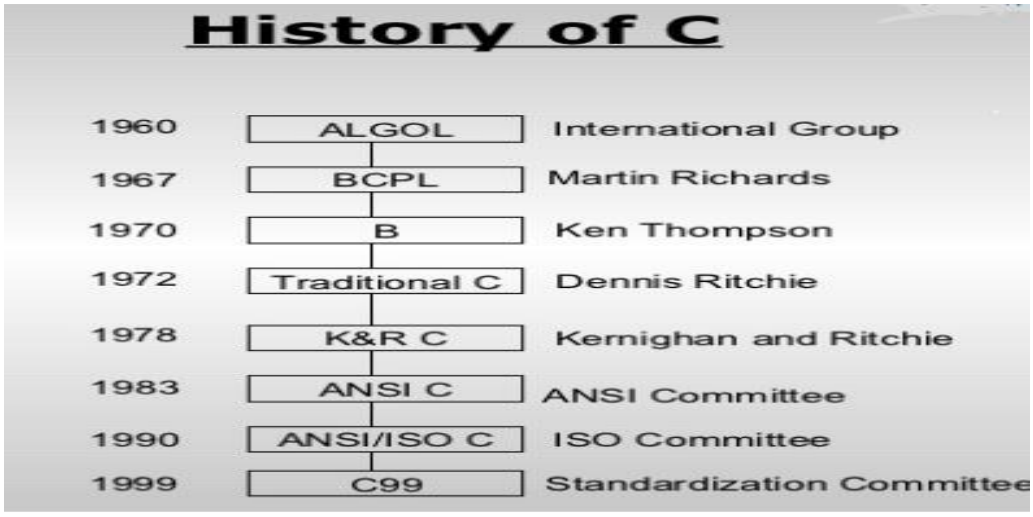
PROGRAMMING IN C -7BCE1C1

UNIT-I

1.1 History of c
1.2 Importance of 'C' language
1.3 Basic Structure of C Program
1.4 programming Style:
.1.5 Character set
1.6 C Tokens
    1.6.1. Keywords and Identifiers
    1.6.2. Constants
    1.6.3. Strings
    1.6.4. Special Symbols
    1.6.5. Operators in C
1.7 Variables
1.8 Data types
1.9. Declaration of variables
1.10. Defining symbolic constants
1.11. Declaring variable as constants
1.12. Overflow and underflow of data
1.13. Operators
    1.13.1. Arithmetic Operators
    1.13.2. Relational Operators
    1.13.3. Logical Operators
    1.13.4. Assignment Operators
    1.13.5. Increment and Decrement Operators
    1.13.6. Conditional Operators
    1.13.7. Bitwise Operators
    1.13.8. Special Operators
1.14. Expressions
1.15. Evaluation of expression
1.16. Precedence of arithmetic Operator
1.17. Type conversion in expressions
1.18. Operator precedence and Associativity
1.19. Mathematical function

1.1 History of c



1.2 Importance of 'C' language

1. It is robust language whose rich setup of built in functions and operator can be used to write any complex program.

2. Program written in C are efficient due to several variety of data types and powerful operators.

3. The C compiler combines the capabilities of an assembly language with the feature of high level language. Therefore it is well suited for writing both system software and business package.

4. There are only 32 keywords; several standard functions are available which can be used for developing program.

5. C is portable language; this means that C programs written for one computer system can be run on another system, with little or no modification.

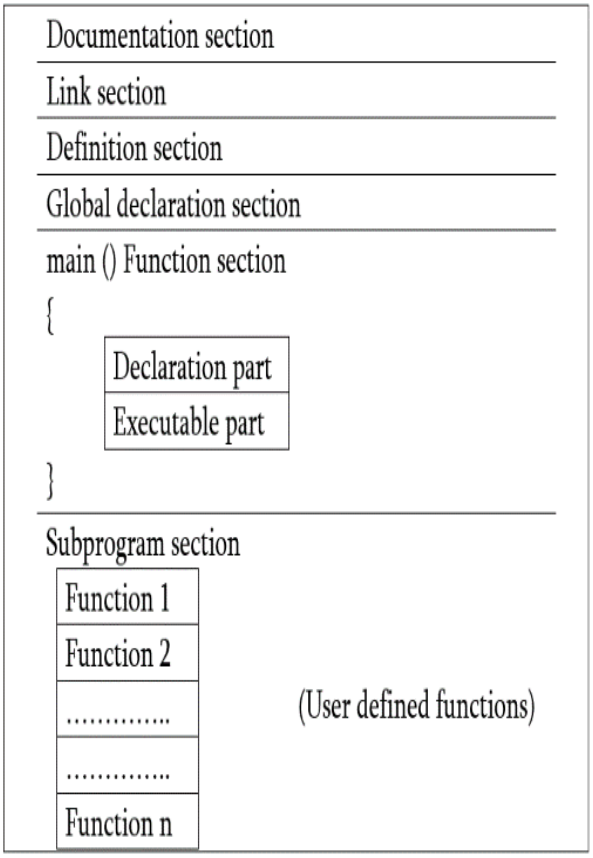
6. C language is well suited for structured programming, this requires user to think of a problems in terms of function or modules or block. A collection of these modules make a program debugging and testing easier.

7. C language has its ability to extend itself. A c program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions. In India and abroad mostly people use C programming language because it is easy to learn and understand.

1.3 Basic Structure of C Program

The components of the basic structure of a C program consists of 6 parts

1. **Document section**-The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.
2. **Preprocessor/link Section**-This part of the code is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.
3. **Definition section-**In this section, we define different constants. The keyword define is used in this part.
4. **Global declaration section**-This part of the code is the part where the global variables are declared. All the global variable used are declared in this part. The user-defined functions are also declared in this part of the code.
5. **Main function**-Every C-programs needs to have the main function. Each main function contains 2 parts. A declaration part and an Execution part. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.
6. **User-defined function section**-All the user-defined functions are defined in this section of the program.



1.4 programming Style:

1. It has to use lowercase letters. Uppercase only used for symbolic constant.

2. Braces used for group the statement together.

3.Group the multiple line in one statement

4.Comments are used for understand the program logic.

1.5 Character set

C language contains the following set of characters...

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

Letters

C language supports all the alphabets from the English language. Lower and upper case letters together support 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

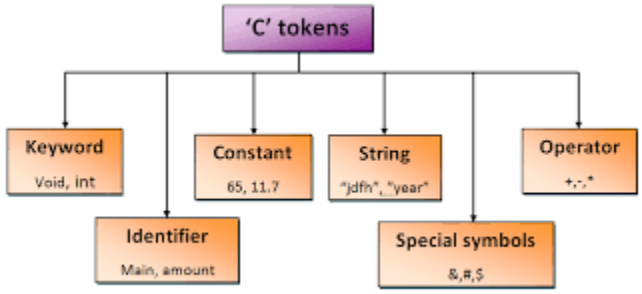
Special characters

C language supports a rich set of special characters- camma,period semicolon colan,question mark ect(**~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; : ' " / ? . > , < \ |**)

white spaces, backspaces,harizantal tab,carriage return,newline,form feed.

1.6 C Tokens

The smallest individual unit are known as tokens. C has six types of tokens as shown in fig.



1.6.1. Keywords and Identifiers

All keywords have fixed meanings and cannot be changed. Keywords are **pre-defined** or **reserved words** in c. there are total 32 keywords in C.

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

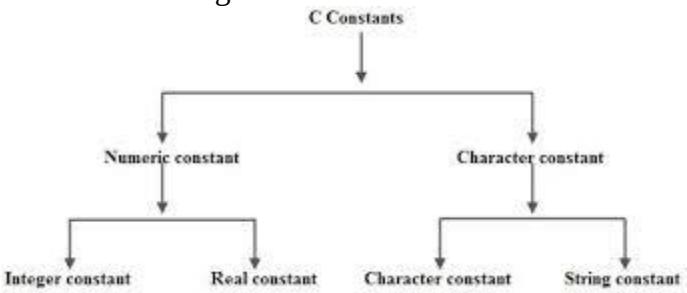
Identifiers: Identifiers refer to the name of a variable, function and arrays.

Rules for identifiers:

- First Character must be an alphabet (or underscore).
- must consist of only letters, digits, or underscore.
- It should be up to 31 characters long.
- Cannot use a keyword.
- must not contain white space.

1.6.2. Constants

Constants: the fixed values do not change during the execution time. Several types constants as in fig



1. Integer constants

2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

1.6.3. Strings

The string constants are a sequence of characters enclosed in double quotes. A string is an array of characters ended with a null character (\0). This null character indicates that string has ended. Strings are always enclosed with double quotes (“ “).

Let us see how to declare String in C language −

- char string[20] = {‘h’,’e’,’l’,’l’,’o’, ‘\0’};
- char string[20] = “WELCOME”;
- char string [] = “WELCOME”;

1.6.4. Special Symbols

Some special backslash character constants are used in Output functions. These characters combination is known as escape sequences.

Constants	Meaning
‘\a’	Audible alert (bell)
‘\b’	Back space
‘\f’	Form feed
‘\n’	New line
‘\r’	Carriage return
‘\t’	Horizontal tab
‘\v’	Vertical tab
‘\’’	Single quote
‘\”’	Double quote
‘\?’	Question mark
‘\\’	Backslash
‘\0’	Null character

1.6.5. Operators in C

Operators is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands.

1.7 Variables

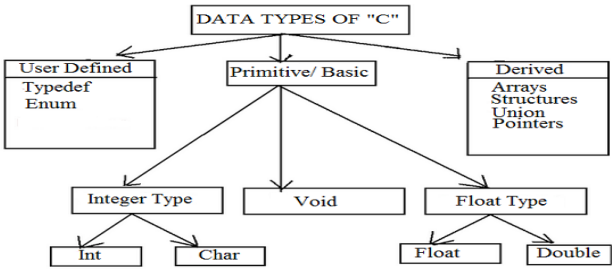
A variable is a data name that may be used to store a data value. the values can be changed during the execution time.

Rules for variable names

- The first letter must be an letters(or underscore).
- Variable name has length of 31 characters.
- Uppercase and lowercase are significant
- It should not be a keyword
- White space is not allowed

1.8 Data types

Data types are the keywords, which are used for assigning a type to a variable.



- Primary (Fundamental)datatypes
- Derived data types
- User defined data types

Primary (Fundamental)datatypes

1. **Integer(int)-**Integer are whole numbers with a range. Integer occupy one word of storage. C has three class of integer storage that is short int, int, long int all in signed and unsigned forms. The storage size of int data type is 2 or 4 or 8 byte. int (2 byte) can store values from -32,768 to +32,767.int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.

2. Character(char)- Character data type allows a variable to store only one character.Storage size of character data type is 1. We can store only one character using character data type.“char” keyword is used to refer character data type.For example, ‘A’ can be stored using char datatype. You can’t store more than one character using char data type.Please refer C – Strings topic to know how to store more than one characters in a variable.
3. Floating point(float)-Float data type allows a variable to store decimal values. Storage size of float data type is 4.We can use up-to 6 digits after decimal using float data type.For example, 10.456789 can be stored in a variable using float data type
4. Double -precision floating point(double)- Double data type is also same as float data type .The range for double datatype is from 1E–37 to 1E+37. A double data type number uses 64 bits (8 bytes) It gives a precision of up to 14 digits.

5. Void - The void has no values. This is usually used to specify the type of functions. The type of function is said to be void when it does not return any value to the calling function. It can also play a role of a generic type, meaning that it can represent any of the other standard types.

Derived data types

The derived data type consists of Arrays (used to manage large numbers of objects of the same type) , Functions (does a specific job), Pointers (represents the address and type of a variable or a function).

User defined data types

Structures (Different data items that make up a logical unit are grouped together as a structure data type)

- Unions (A union permits references to the same location in memory to have different types) and
- Enumeration (used to define variables that can only be assigned certain possible values throughout the program)

1.9. Declaration of variables

After designing suitable variable names, we must declare them to the compiler. The declaration does 2 things:

- It tells the compiler what the variable is
- It specifies what type of data the variable will hold

Primary Data Type Declaration

A variable can be used to store a value of any data type **Syntax**

```
data-type v1,v2,…vn;
```

where v1,v2…vn are the names of variables. **Example** int count; double ratio

1.10. Defining symbolic constants

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. A constant is 3.142, for "pi" value. The "pi" value can be used for many purposes in a program. We face two problems in the subsequent use of such programs.

- Problem in modification of the program
- Problem in understanding the program

Assignment of such constants to a symbolic name frees us from these problems. The symbolic constant can be defined as follows:

```
#define symbolic_name value of constant

// Example

#define STRENGTH 100

#define PASS_MARK 50
```

Symbolic names are sometimes called constant identifiers. The following are the rules for declaring symbolic constants,

- Symbolic names have the same form as variable names.
- no blank space between '#' and the word 'define'
- # must be the first character in the line
- a blank space is must between the #define and the symbolic name
- after definition, the symbolic name should not be assigned to any other value within the program
- symbolic names are not declared for data types
- #define must not end with semicolon (;)
- #define may appear anywhere in the program but before it is referenced in the program

1.11. Declaring variable as constants

The need of certain values of variables to remain constant during the execution of a program. This can be done with by declaring the variable with the qualifier const at the time of initialization.

Ex: const int class_size=40;

(value of the int variable class_size cannot be modified by the program).

1.12. Overflow and underflow of data

- Problem of data overflow occurs when the value of a variable is either too big or too small for the data to hold
- In floating point conversions, the values are rounded off to the number of significant digits
- An overflow normally results in the largest possible value a machine can hold and underflow results in zero
- The overflow problem may occur if the data type does not match the value of the constant
- 'C' does not provide any warning or indication of integer overflow. It simply gives incorrect results. So care should be taken to define correct data types for handling I/O values

1.17.Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators require some data to operate on and such data is called *operands*. Operators in C can be classified into following categories:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

1.17.1. Arithmetic Operators

C programming language provides all basic arithmetic operators. These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus. **The '/' is integer division which only gives integer part as result after division. '%' is modulo division which gives the remainder of integer division as result.**

Operators	Meanings
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

1.17.2. Relational Operators

These operators are used to compare the value of two variables.

Operators	Meanings
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

1.17.3. Logical Operators

These operators are used to perform logical operations on the given two variables. Logical operators are used when more than one conditions are to be tested and based on that result, decisions have to be made. An expression which combines two or more relational expressions is known as logical expression.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

1.17.4. Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. The most useful assignment operator in C is '='. C also has a set of following shorthand assignment operators.

var op = exp;

is the same as the assignment

var = var op exp;

where *var* is a variable, *op* is arithmetic operator, exp is an expression. In this case, 'op=' is known as shorthand assignment operator.

Operator	Meaning
=	a=b
a+=b	a=a+b
a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b

1.17.5. Increment and Decrement Operators

programming allows the use of ++ and – operators which are increment and decrement operators respectively. Both the increment and decrement operators are unary operators. The increment operator ++ adds 1 to the operand and the decrement operator – subtracts 1 from the operand. The general syntax of these operators are:

Operator	Description	Example
++	increment	a++ (post increment) ++a (pre increment)
--	decrement	a-- (post decrement) --a (pree decrement)

Increment Operator: *m++ or ++m*;

Decrement Operator: *m--or --m*;

In the example above, *m++* simply means *m=m+1;* and *m--* simply means *m=m-1;*

Increment and decrement operators are mostly used in for and while loops.

++m and *m++* performs the same operation when they form statements independently but they function differently when they are used in right hand side of an expression.

++m is known as prefix operator and *m++* is known as postfix operator. A prefix operator firstly adds 1 to the operand and then the result is assigned to the variable on the left whereas a postfix operator firstly assigns value to the variable on the left and then increases the operand by 1. Same is in the case of decrement operator.

For example,

```
X=10;
Y=++X;
```

1.17.6. Conditional Operators

Conditional operators return one value if condition is true and returns another value is condition is false.The operator pair "?" and ":" is known as conditional operator. These pair of operators are ternary operators. The syntax is:

expression1 ? expression2 : expression3 ;

This syntax can be understood as a substitute of if else statement.

Consider an if else statement as:

```
if  (a > b)
x = a ;
else
x = b ;
```

Now, this if else statement can be written by using conditional operator as:

```
x = (a > b) ? a : b ;
```

1.17.7. Bitwise Operators

In C programming, bitwise operators are used for testing the bits or shifting them left or right.

Operators	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

1.17.8. Special Operators

C programming supports special operators like comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and ->).

i). Comma Operator

The comma operator can be used to link the related expressions together. A comma linked expression is evaluated from left to right and the value of the right most expression is the value of the combined expression.

For example:

```
x = (a = 2, b = 4, a+b)
```

In this example, the expression is evaluated from left to right. So at first, variable a is assigned value 2, then variable b is assigned value 4 and then value 6 is assigned to the variable x. Comma operators are commonly used in for loops, while loops, while exchanging values.

ii). Sizeof() operator

The sizeof operator is usually used with an operand which may be variable, constant or a data type qualifier. This operator returns the number of bytes the operand occupies. Sizeof operator is a compile time operator. The sizeof operator is usually used to determine the

length of arrays and structures when their sizes are not known. It is also used in dynamic memory allocation.

x = sizeof (a);
y = sizeof(float);

1.18. Expressions

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of mathematical expressions written in proper syntax of C are:

Algebraic expression	*C expression*
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2+2x+1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right)+c$	x/y+c

1.19. Evaluation of expression

Expressions are evaluated using an assignment statement of the form:

Variable=expression;

The expression evaluated first and the result then replaces the previous value of the variable on the left hand side. All the variables used in the expression must be assigned values before evaluation attempted. Examples of evaluation statement are

```
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 statement are used in program, the variable a,b,c and d must be defined before they are used in the expressions.

1.20. Precedence of arithmetic Operator

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

High priority: * / %

Low priority: + -

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered.

- Suppose, we have an arithmetic expression as: x = 9 – 12 / 3 + 3 *2 – 1. This expression is evaluated in two left to right passes as:

First Pass
Step 1: x = 9-4 + 3 * 2 – 1
Step 2: x = 9 – 4 + 6 – 1
Second Pass
Step 1: x = 5 + 6 – 1
Step 2: x = 11 – 1
Step 3: x = 10

- But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example, x = 9 – 12 / (3 + 3) * (2 – 1)

When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passes as:

First Pass
Step 1: x = 9 – 12 / 6 * (2 – 1)
Step 2: x= 9 – 12 / 6 * 1
Second Pass
Step 1: x= 9 – 2 * 1
Step 2: x = 9 – 2
Third Pass
Step 3: x= 7

- There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.For example, we have an expression as:

x = 9 – ((12 / 3) + 3 * 2) – 1.The expression is now evaluated as:

First Pass:
Step 1: x = 9 – (4 + 3 * 2) – 1
Step 2: x= 9 – (4 + 6) – 1
Step 3: x= 9 – 10 -1
Second Pass
Step 1: x= - 1 – 1

Step 2: x = -2

Operator Precedence The precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to any of these levels. The operators at higher level of precedence are evaluated first. The number of evaluation steps is equal to the number of operators in the arithmetic expression.

The rules for evaluation of expression are as follows:

1. First, parenthesized sub expressions from left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub-expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
4. The associativity rule is applied when to or more operators of the same precedence appear in a subexpression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parentheses assume highest priority.

1.17. Type conversion in expressions

Implicit Type Conversion

C automatically converts any intermediate values to the proper type so that expression can be evaluated without losing any significance. This automatic conversion is known as implicit type conversion. During evaluation, if the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds. And the result is always of the higher type.

For example: 5/2 = 2

5/2.0 = 2.5 (Implicit type conversion)

5.0/2 = 2.5 (Implicit type conversion)

Explicit Type Conversion

There are instances when we want to force a type conversion in a way that is different from the automatic conversion. This problem is solved by converting locally one of the variables to the higher type.

For Example: 5/2 = 2 (float)

5/2 = 2.5 (Explicit type conversion)

5/ (float)2 = 2.5 (Explicit type conversion)

Explicit type conversion is also known as type casting. There are instances when we want to force a type conversion in a way that is different from the automatic conversion. This problem is solved by converting locally one of the variables to the higher type

1.18. Operator precedence and Associativity

The operators of the same precedence are evaluated either from 'Left-to-Right' or from 'Right-to-Left', depending on the priority (or precedence) level. This is known as the associativity property of operator. Precedence rules decide the order in which different operators are applied. Associativity rule decides the order in which multiple occurrences of the same level operator are applied. The precedence and associativity of some of the operators is as follows:

Table 3.8 *Summary of C Operators*

Operator	Description	Associativity	Rank
() []	Function call Aray element reference	Left to right	1
+ − ++ − − ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ −	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== !=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= * = /= %= += — &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

1.19. Mathematical function

Mathematical function is frequently used in analysis of real-life problems. Most of the c compilers support these basic math functions. Table list some standard math functions.

Table 3.9 *Math functions*

Function.	*Meaning*
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x >= 0

PROGRAMMING IN C -7BCE1C1

UNIT II

2.1. MANAGING I/O OPERATIONS

Managing i/o as we all know the three essential functions of a computer are reading, processing and writing data. Majority of the programs take data as input, and *display the processed data* after known as result

I/O operations are useful for a program to interact with users. stdlib is the standard C library for input-output operations. While dealing with input-output operations in C, two important streams play their role. These are:

1. Standard Input (stdin)
2. Standard Output (stdout)

Standard input or stdin is used for taking input from devices such as the keyboard as a data stream. Standard output or stdout is used for giving output to a device such as a monitor. For using I/O functionality, programmers must include stdio header-file within the program.

2.2. READING AND WRITING A CHARACTER

Reading a Character

The easiest and simplest of all I/O operations are taking a character as input by reading that character from standard input (keyboard). getchar() function can be used to read a single character. This function is alternate to scanf() function.

Syntax:

```
var_name = getchar();
```

Writing a Character

Similar to getchar() there is another function putchar() which is used to write characters, but one at a time.

Syntax:

```
putchar(var_name);
```

2.3. FORMATTED INPUT, OUTPUT

Formatted Input

It refers to an input data which has been arranged in a specific format. This is possible in C using scanf(). We have already encountered this and familiar with this function.

Syntax:

```
scanf("control string", arg1, arg2, ..., argn);
```

The field specification for reading integer inputted number is:%w sd .Here the % sign denotes the conversion specification; w signifies the integer number that defines the field width of the number to be read. d defines the number to be read in integer format.

Input data items should have to be separated by spaces, tabs or new-line and the punctuation marks are not counted as separators.

Formatted output

The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.

Syntax:

```
printf("control string", arg1, arg2, ..., argn);
```

In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.

Reading and Writing Strings in C

There are two popular library functions gets() and puts() provides to deal with strings in C.

gets:

The char *gets(char *str) reads a line from stdin and keeps the string pointed to by the str and is terminated when the new line is read or EOF is reached. The declaration of gets() function is:

Syntax:

```
char *gets(char *str);
```

Where str is a pointer to an array of characters where C strings are stored.

puts:

The function - int puts(const char *str) is used to write a string to stdout, but it does not include null characters. A new line character needs to be appended to the output. The declaration is:

Syntax:

```
int puts(const char *str)
```

where str is the string to be written in C.

Decision Making And Branching

'C' language processes decision making capabilities supports the flowing statements known as control or decision making statements

1. If statement
2. switch statement
3. conditional operator statement
4. Goto statement

If Statement : The if statement is powerful decision making statement and is used to control the flow of execution of statements The If statement may be complexity of conditions to be tested

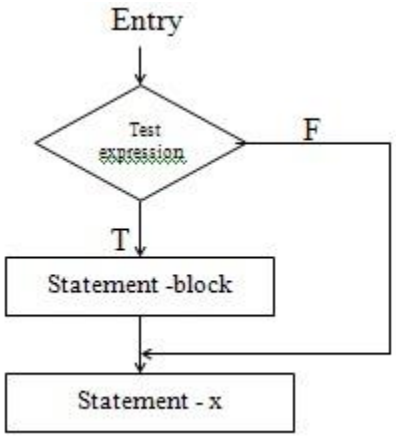
(a) Simple if statement
(b) If else statement
(c) Nested If-else statement
(d) Else –If ladder

Simple If Statement : The general form of simple if statement is

```
If(test expression)
{
statement block;
        }   statement-x ;
```

The statement -block may be a single statement or a group of statement if the test expression is true the statement block will be executed. Otherwise the statement -block will be skipped and the execution will jump to the statement –X. If the condition is true both the statement –block sequence .

Flow chart :



Ex : If(category = sports)
{ marks = marks + bonus marks;
} printf("%d",marks);

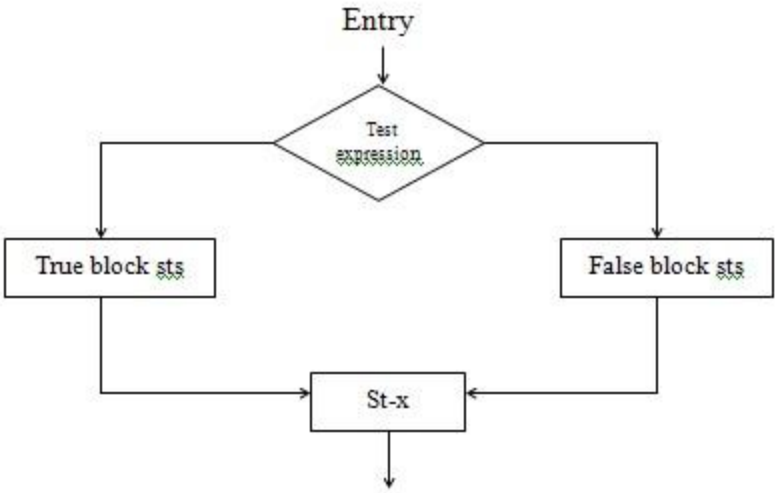
If the student belongs to the sports category then additional bonus marks are added to his marks before they are printed. For other bonus marks are not added .

If –Else Statement : The If statement is an extension of the simple If statement the general form is

```
        If (test expression)
        {
          true-block statements;
        }
else
        {
         false-block statements;
        }
 statement – x;
```

If the test expression is true then block statement are executed, otherwise the false –block statement are executed. In both cases either true-block or false-block will be executed not both.

Flow chart :



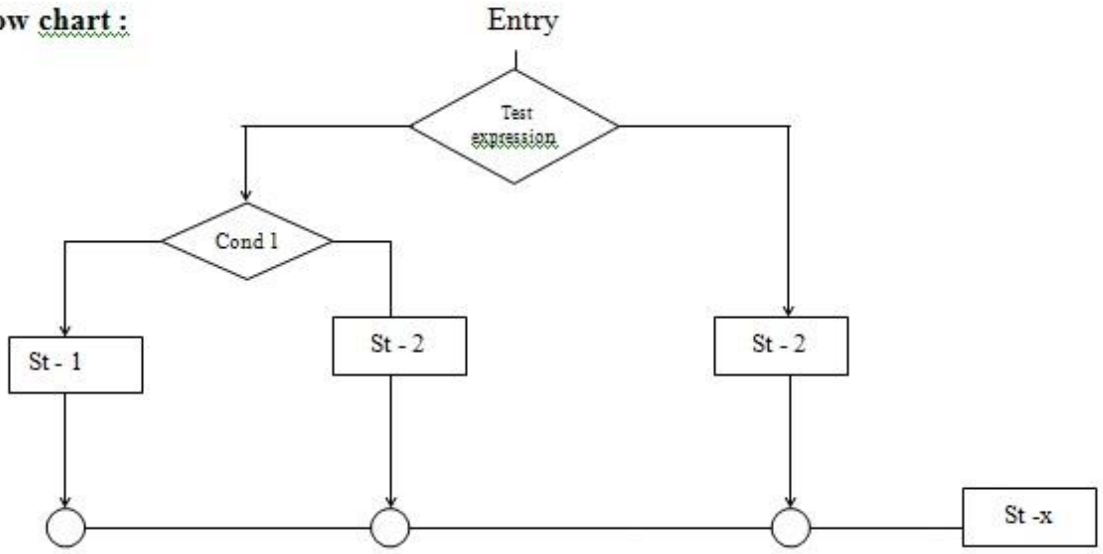
Ex : If (code == 1)
boy = boy + 1;
else
girl = girl + 1;
st-x;

Here if the code is equal to '1' the statement *boy=boy+1;* Is executed and the control is transfered to the *statement st-n*, after skipping the else part. If code is not equal to '1' the statement *boy =boy+1;* is skipped and the statement in the else part *girl =girl+1;* is executed before the control reaches the statement st-n.

Nested If –else statement : When a series of decisions are involved we may have to use more than one if-else statement in nested form of follows .

If(test expression)
{ if(test expression)
{ st –1;
}
else
{ st – 2;
}else
{
st – 3;
}
}st – x;

Flow chart :



If the condition is false the st-3 will be executed otherwise it continues to perform the nested If –else structure (inner part). If the condition 2 is true the st-1 will be executed otherwise the st-2 will be evaluated and then the control is transferred to the st-x

Some other forms of nesting If-else

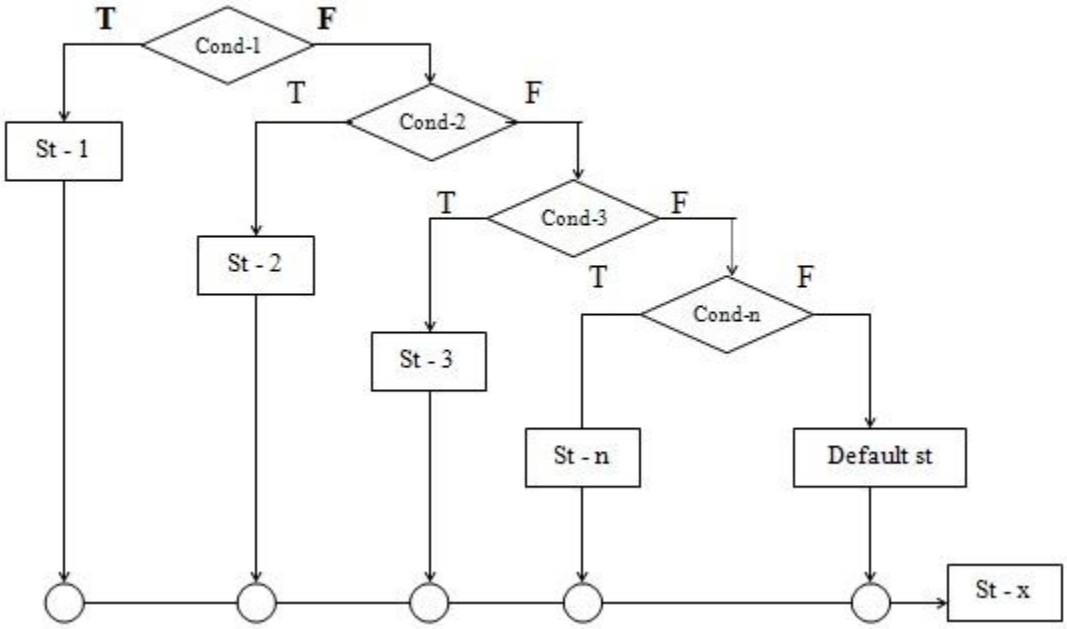
```
If ( test condition1)
{  if (test condition2)
st –1 ;
} else
if (condition 3)
{ if (condition 4)
st – 2;
}st – x;
```

Else-If ladder : A multi path decision is charm of its in which the statement associated with each else is an If. It takes the following general form.

```
        If (condition1)
St –1;
Else  If (condition2)
St –2;
Else if (condition 3)
St –3;



Else
```

Default – st;
St –x;

Flow chart : Entry



This construct is known as the wise-If ladder. The conditions are evaluated from the top of the ladder to down wards. As soon as a true condition is found the statement associated with it is executed and the control the is transferred to the st-X (i.e.., skipping the rest of the ladder). when all the n-conditions become false then the final else containing the default – st will be executed.

Ex : If (code = = 1) Color = "red";
Else if (code = = 2) Color = "green"
Else if (code = = 3) Color = "white";
Else Color = "yellow";

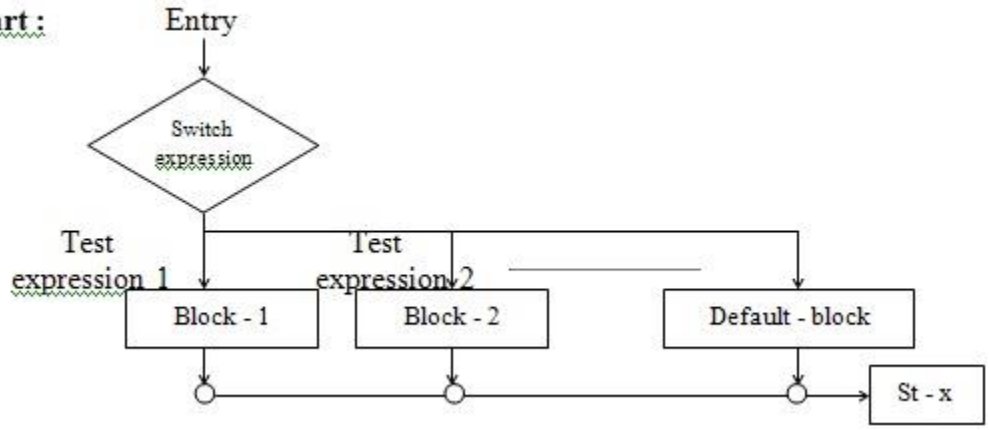
If code number is other than 1,2 and then color is yellow.

Switch Statement : Instead of else –if ladder, 'C' has a built-in multi-way decision statement known as a switch. The general form of the switch statement is as follows.

Switch (expression)
{
case value1 : block1;
break;

```
case value 2  : block 2;
                break;


default       :  default block;
                 break;
}
st – x;
```

Flow Chart :



The expression is an integer expression or character value1, value-2---- are constants or constant expressions and also known as case lables. Each of the values should be a unit within a switch and may contain zero or more statements.

When the switch is executed the value of the expression is successively compared against the values value-1,value-2------- If a case is found whose value matches with the of the expression then the block of statements that follows the case are executed .

The break statement at the end of each block signals the end a particular case and causes an exist from the switch statement transfering the control to the st-x following the switch. The default is an optional case . If will be executed if the value of the expression doesn't match with any Of the case values then control goes to the St-x.

Ex :

```
        switch (number)
        {
        case 1 : printf("Monday");
                 break;
        case 2 : printf("Tuesday");
                 break;
        case 3 : printf("Wednesday");
                 break;
        case 4 : printf("Thursday");
                 break;
```

```
case 5 : printf("Friday");
         break;
        default : printf("Saturday");
         break;
}
```

The Conditional (? :) Operator : These operator is a combinations of question and colon and takes three operands this is also known as conditional operator. The general form of the conditional operator is as follows

Conditional expression? Expression 1:expression2

The conditional expression is evaluated first If the result is non-zero expression is evaluated and is returns as the value of the conditional expression, Otherwise expression2 is evaluated and its value is returned.

Ex : flag = (x<0) ? 0 : 1

It's equalent of the If-else structure is as follows

```
        If ( x<0)
Flag = 0;
        Else
                Flag = 1;
```

Goto Statement : The goto statement is used to transfer the control of the program from one point to another. It is something reffered to as unconditionally branching. The goto is used in the form

Goto label;

Label statement : The label is a valid 'C' identifier followed by a colon. we can precode any statement by a label in the form

Label : statement ;

This statement immediately transfers execution to the statement labeled with the label identifier.

Ex :

```
i = 1;
bc : if(1>5)
goto ab;
printf("%d",i);
++i;
gotobc;
ab : {
printf("%d",i); }
```

Output :

```
1
2
3
4
5
```

Decision making looping

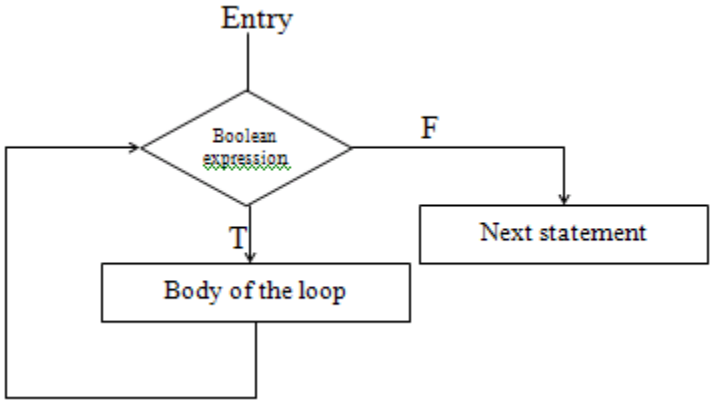
The 'C' language provides three loop constructs for performing loop operations they are

1. The while statement
2. Do-while statement
3. The for statement

The While Statement : This type of loop is also called an entry controlled, is executed and if is true then the body of the loop is executed this process repeated until the boolean expression becomes false. Ones it becomes false the control is a transferred out the loop. The general form of the while statement is

```
While (boolean expression)
{
        body of the loop;
}
```

Flow chart:



Ex :

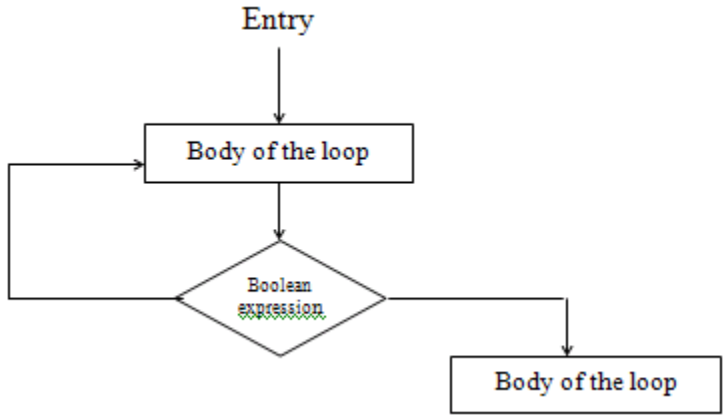
```
i = 1;
While(I<=5)
{   printf("%d",i);
i++; }
```

In the above example the loop with be executed until the condition is false

Do-While Statement : This type of loop is also called an exist controlled loop statement. i.e.., The boolean expression evaluated at the bottom of the loop and if it is true then body of the loop executed again and again until the boolean expression becomes false. Ones it becomes false the control is do-while statement is

```
Do
{
        body of the loop ;
}
while ( boolean expression)
```

Flowchart:



Ex :

```
i = 1;
Do
{
printf("%d",i);
i++;
}
While(i<=5)
```

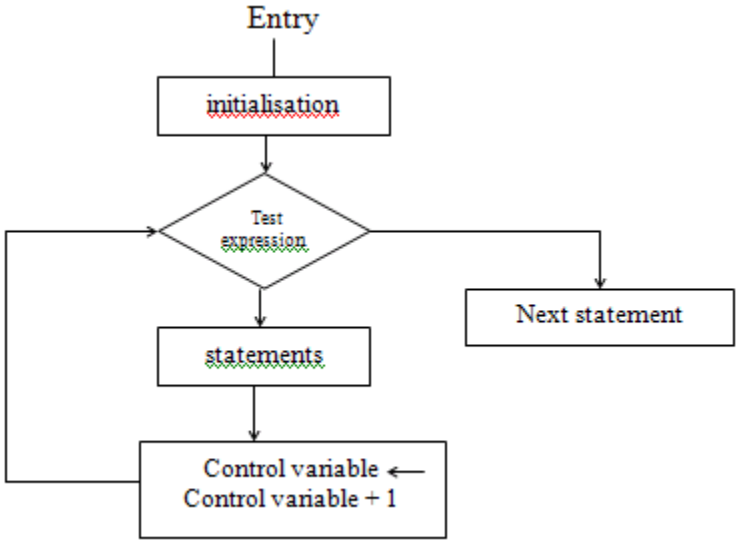
While statement	**Do-while statement**
1. It is an entry controlled loop.	1. It is an exit controlled looping.
2. If boolean expression is false then the body of the loop never Boolean expression is either true or false	2. The body of the loop executed atleast ones even be executed if the

The For Statement : The for loop is another entry control led loop that provides a more concise loop control structure the general form of the for loop is

```
For ( initialisation; test condition; increment)
{
        body of the loop;
}
```

Where initialization is used to initialize some parameter that controls the looping action, 'test condition' represents if that condition is true the body of the loop is executed, otherwise the loop is terminated After evaluating information and the new value of the control variable is again tested the loop condition. If the condition is satisfied the body of the loop is again executed it this process continues until the value of the control variable false to satisfy the condition.

Flow chart :



Ex : for (I=1; I<=5; I++)
{
printf("%d",i);
}

Output : 1 2 3 4 5

Jumping From The Loops :

Break Statement : The break statement can be accomplish by using to exist the loop. when break is encountered inside a loop, the loop is immediately exited and the program continues with the statement which is followed by the loop. If nested loops then the break statement inside one loop transfers the control to the next outer loop.

Ex :
```
for (I=1; I<5; I++)
{
if ( I == 4)
break;
printf("%d",i);
}
```

Output : 1 2 3

Continue Statement : The continue statement which is like break statement. Its work is to skip the present statement and continues with the next iteration of the loop .

Ex :
```
for (I=1; I<5; I++)
{
if ( I == 3)
continue;
printf("%d",i);
}
```

Output : 1 2 4 5

In the above example when I=3 then the continue statement will rise and skip statement in the loop and continues for the next iteration i.e.., I=4.

7BCE 1C1 – PROGRAMMING IN C

UNIT -3:

CHAPTER 7: ARRAYS

- **One Dimensional Arrays**
- **Declaration of One Dimensional Arrays**
- **Initialization of One Dimensional Arrays**
- **Two Dimensional Arrays**
- **Multi Dimensional Arrays**
- **Dynamic Arrays**

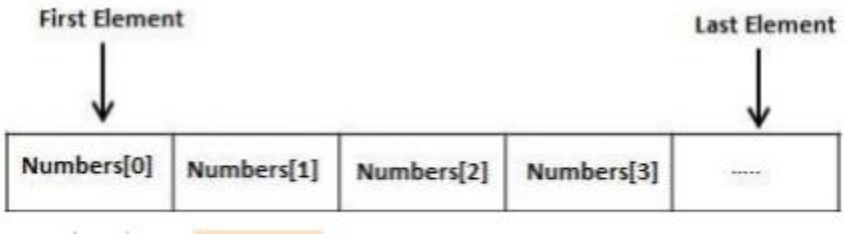
CHAPTER 8: Strings

- **Declaring and Initializing String Variables**
- **Reading Strings from Terminal**
- **Writing Strings to Screen**
- **String Handling Functions**

CHAPTER 7: ARRAYS

What is Array?

- C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



One Dimensional Arrays

- A one-dimensional array (or **single dimension array**) is a type of linear array. Accessing its elements involves a **single subscript** which can either represent a row or column index.
- As an example consider the C declaration

  int anArrayName[10];

- which declares a one-dimensional array of ten integers. Here, the array can store ten elements of type int . This array has indices starting from zero through nine.

- ⇨ For example, the xpressions **anArrayName[0]** and **anArrayName[9]** are the **first** and **last** elements respectively.

Declaration of One Dimensional Arrays

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

type arrayName [arraySize];

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

double balance[10];

- ⇨ Now balance is a variable array which is sufficient to hold up-to 10 double numbers.
- ⇨ Rules for Declaring One Dimensional Array:
    - An array variable must be declared before being used in a program.
    - The declaration must have a **data type** (int, float, char, double, etc.), **array name**, and **subscript**.
    - The subscript represents the size of the array. If the size is declared as 10, programmers can store 10 elements.
    - An array index always starts from 0. For example, if an array variable is declared as s[10], then it ranges from 0 to 9.
    - Each array element stored in a separate memory location.

Initialization of One Dimensional Arrays

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

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

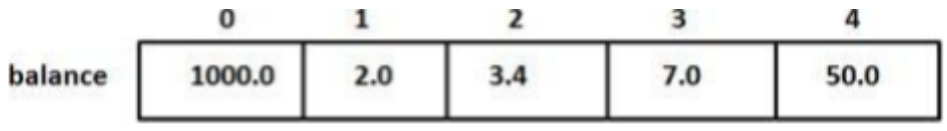
- The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:
  - If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

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

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

balance[4] = 50.0;

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



Accessing Array Elements

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

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable.

⇨ Following is an example which will use all the above mentioned three concepts viz. **declaration, assignment and accessing arrays**:

```
#include <stdio.h>
 int main ()
{
        int n[ 10 ]; /* n is an array of 10 integers */
         int i,j;
        /* initialize elements of array n to 0 */
         for ( i = 0; i < 10; i++ )
         {
                n[ i ] = i + 100; /* set element at location i to i + 100 */
        }
         /* output each array element's value */
        for (j = 0; j < 10; j++ )
        {
                printf("Element[%d] = %d\n", j, n[j] );
         }
 return 0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
```

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

Two-Dimensional Arrays

- The simplest form of the multidimensional array is the two-dimensional array. A twodimensional array is, in essence, a list of one-dimensional arrays.
- To declare a twodimensional integer array of size x, y you would write something as follows:

type arrayName [x][y];

- Where type can be any valid C data type and arrayName will be a valid C identifier. A two dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimentional array a, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

⇨ Two dimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
                  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
                  {8, 9, 10, 11}   /*initializers for row indexed by 2 */
               };
```

⇨ The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

⇨ An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

⇨ The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>
 int main ()
{
        /* an array with 5 rows and 2 columns*/
        int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
        int i, j;
        /* output each array element's value */
        for ( i = 0; i < 5; i++ )
```

```
        {
                for ( j = 0; j < 2; j++ )
                {
                        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
                }
        }
return 0;
}
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

- As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Multi-dimensional Arrays

- In C programming, you can create an **array of arrays**. These arrays are known as multidimensional arrays. Here is the general form of a multidimensional array declaration:

type name[size1][size2]...[sizeN];

⇨ For example, the following declaration creates a three dimensional 2 , 3, 2 integer array:

int test[2][3][2];

⇨ Example C program for multi dimensional array,

```
// C Program to store and print 12 values entered by the user
#include <stdio.h>
int main()
{
    int test[2][3][2];
    printf("Enter 12 values: \n");
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {  scanf("%d", &test[i][j][k]);  }
        }
    }
    // Printing values with proper index.
    printf("\nDisplaying values:\n");
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
            }
        }
    }
    return 0;
}
```

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

```
Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

Displaying Values:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

Dynamic Arrays

- As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as **static memory allocation** and the array is called as **static array.**
- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming. The memory of an array allocated at run time is called **dynamic memory allocation** and that type of array is called as **dynamic array.**
- To allocate memory dynamically, using **pointer variables** or using library functions are **malloc(), calloc(), realloc()** and **free()** are used. These functions are defined in the **<stdlib.h>** header file.

CHAPTER 8: Strings

What is a String?

- **String** is nothing but a **collection of characters in a linear sequence**. 'C' always treats a string a single data even though it contains whitespaces. A single character is defined using single quote representation. A string is represented using double quote marks.

Example: "Welcome to the world of programming!"

- 'C' provides standard library <string.h> that contains many functions which can be used to perform complicated string operations easily.

Declaring and initializing a String Variables

- A string is a simple array with char as a data type. 'C' language does not directly support string as a data type. Hence, to display a string in 'C', you need to make use of a character array.
- The general syntax for declaring a variable as a string is as follows,

```
char string_variable_name [array_size];
```

- The classic string declaration can be done as follow:

```
char string_name[string_length] = "string";
```

- The size of an array must be defined while declaring a string variable because it used to calculate how many characters are going to be stored inside the string variable. Some valid examples of string declaration are as follows,

```
char first_name[15];//declaration of string variable
```

```
char last_name[15];
```

- ⇨ The above example represents string variables with an array size of 15. This means that the given character array is capable of holding 15 characters at most. The indexing of array begins from 0 hence it will store characters from a 0-14 position. **The C compiler automatically adds a NULL character '\0' to the character array created.**
- ⇨ Let's study the initialization of a string variable. Following example demonstrates the initialization of a string variable,

```
char first_name[15] = "ANTHONY";
char first_name[15] = {'A','N','T','H','O','N','Y','\0'}; // NULL character '\0'
char string1 [6] = "hello";
/* string size = 'h'+'e'+'l'+'l'+'o'+"NULL" = 6 */
char string2 [ ] = "world";
/* string size = 'w'+'o'+'r'+'l'+'d'+"NULL" = 6 */
char string3[6] = {'h', 'e', 'l', 'l', 'o', '\0'} ; /*Declaration as set of characters ,Size 6*/
```

- ⇨ In string3, the NULL character must be added explicitly, and the characters are enclosed in single quotation marks.
- ⇨ 'C' also allow us to initialize a string variable without defining the size of the character array. It can be done in the following way,

```
char first_name[ ] = "NATHAN";
```

- ⇨ The **name of a string** acts as a **pointer** because it is basically an array.

String Input: Read a String from Terminal

- ⇨ When writing interactive programs which ask the user for input, C provides the **scanf(), gets(),** and **fgets()** functions to find a line of text entered from the user.

scanf() function:

⇨ When we use **scanf()** to read, we use the "%s" format specifier without using the "&" to access the variable address because an array name acts as a pointer. For example:

```
#include <stdio.h>
int main()
{
char name[10];
int age;
printf("Enter your first name and age: \n");
scanf("%s %d", name, &age);
printf("You entered: %s %d",name,age);
}
```

Output:

```
Enter your first name and age:
John_Smith 48
```

⇨ The problem with the scanf function is that it never reads an entire string. It will halt the reading process as soon as whitespace, form feed, vertical tab, newline or a carriage return occurs. Suppose we give input as "Guru99 Tutorials" then the scanf function will never read an entire string as a whitespace character occurs between the two names. The scanf function will only read Guru99.

gets() function:

⇨ In order to read a string contains spaces, we use the **gets()** function. Gets ignores the whitespaces. It stops reading when a newline is reached (the Enter key is pressed).For example:

```
#include <stdio.h>
int main()
{
char filename[25];
printf("Enter your full name: ");
gets(full_name);
```

```
printf("My full name is %s ",full_name);
return 0;
}
```

Output:

```
Enter your full name: Dennis Ritchie
My full name is Dennis Ritchie
```

fgets() function:

⇨ Another safer alternative to gets() is **fgets()** function which reads a specified number of characters. For example:

```
#include <stdio.h>
int main()
{
        char name[10];
        printf("Enter your  name plz: ");
        fgets(name, 10, stdin);
        printf("My name is %s ",name);
        return 0;
}
```

Output:

```
Enter your name plz: Carlos
My name is Carlos
```

⇨ The fgets() arguments are :

- the string name,
- the number of characters to read,
- stdin means to read from the standard input which is the keyboard.

String Output: Print/Display a String to Screen

⇨ C provides the **printf(), puts(),** and **fputs()** functions to displaying a string on an output device.

printf() function:

⇨ The standard printf function is used for printing or displaying a string on an output device. The format specifier used is **%s**

Example,

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

⇨ String output is done with the **fputs()**, **puts()** and **printf()** functions.

fputs() function:

⇨ The fputs() needs the name of the string and a pointer to where you want to display the text. We use stdout which refers to the standard output in order to print to the screen. For example:

```
#include <stdio.h>
int main()
{
        char town[40];
        printf("Enter your town: ");
        gets(town);
        fputs(town, stdout);
        return 0;
}
```

Output:

```
Enter your town: New York
New York
```

puts() function

⇨ The puts function prints the string on an output device and moves the cursor back to the first position. A puts function can be used in the following way,

```
#include <stdio.h>
int main()
{
        char name[15];
        gets(name);       //reads a string
        puts(name);       //displays a string
        return 0;
}
```

⇨ The syntax of this function is comparatively simple than other functions.

String Handling Functions (The string library)

⇨ The standard 'C' library provides various functions to manipulate the strings within a program. These functions are also called as string handlers. All these handlers are present inside **<string.h>** header file.

Function	Purpose
strlen()	This function is used for finding a length of a string. It returns how many characters are present in a string excluding the NULL character.
strcat(str1, str2)	This function is used for combining two strings together to form a single string. It Appends or concatenates str2 to the end of str1 and returns a pointer to str1.
strcmp(str1, str2)	This function is used to compare two strings with each other. It returns 0 if str1 is equal to str2, less than 0 if str1 < str2, and greater than 0 if str1 > str2.

⇨ Lets consider the program below which demonstrates string library functions:

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

```
{
	//string initialization
	char string1[15]="Hello";
	char string2[15]=" World!";
	char string3[15];
	int val;

	//string comparison
	val= strcmp(string1,string2);
	if(val==0)
	{
		printf("Strings are equal\n");
	}
	else
	{
		printf("Strings are not equal\n");
	}

	//string concatenation
	printf("Concatenated:%s",strcat(string1,string2));

	//string length
	printf("\nLengthstring1 :%d",strlen(string1));
	printf("\nLengthstring2: %d",strlen(string2));

	//string copy
	printf("\nCopied:%s\n",strcpy(string3,string1));
	//string1 is copied into string3

	return 0;
}
```

Output:

```
Strings are not equal
Concatenated string:Hello World!
Length of first string:12
Length of second string:7
Copied string is:Hello World!
```

⇨ Other important library functions are:

- ✓ **strncmp(str1, str2, n) :**it returns 0 if the first n characters of str1 is equal to the first n characters of str2, less than 0 if str1 < str2, and greater than 0 if str1 > str2.

- ✓ **strncpy(str1, str2, n):** This function is used to copy a string from another string. Copies the first n characters of str2 to str1
- ✓ **strchr(str1, c):** it returns a pointer to the first occurrence of char c in str1, or NULL if character not found.
- ✓ **strrchr(str1, c):** it searches str1 in reverse and returns a pointer to the position of char c in str1, or NULL if character not found.
- ✓ **strstr(str1, str2):** it returns a pointer to the first occurrence of str2 in str1, or NULL if str2 not found.
- ✓ **strncat(str1, str2, n):** Appends (concatenates) first n characters of str2 to the end of str1 and returns a pointer to str1.
- ✓ **strlwr() :**to convert string to lower case
- ✓ **strupr() :**to convert string to upper case
- ✓ **strrev() :** to reverse string

Review Questions:

1. Define Array.
2. What do you meant by one dimensional array?
3. How to declare an array in C Program?
4. How to access the array elements?
5. Explain the initialization of two dimensional arrays.
6. Discuss multi dimensional array with example.
7. Write a C program to add two matrices.
8. Define static array.
9. What are the possible ways available in C to create dynamic arrays?
10. What are the functions we need to dynamic memory allocation?
11. State the difference of static and dynamic array.
12. What is String?
13. How to bring the string variable in C program?
14. How can you read a string from terminal?

15. What can we do to display a string on screen?

16. Explain various string handling functions with appropriate C program.

PROGRAMMING IN C-7BCE1C1

UNIT IV

CHAPTER 9: USER DEFINED FUNCTIONS

- Need multifunction programs
- Elements of user defined functions
- Definition
- Return values and their types
- Function calls,declaration
- Category of function-all types of arguments and return values
- Nesting of functions
- Scope visibility and life time of variables

CHAPTER 10: STRUCTURES AND UNIONS

- Defining a structure
- Declaring a structure variable
- Accessing structure members
- Initialization-copying and comparing
- Operation on individual members
- Array of structures
- Array within structures
- Structures within structures
- Structures and functions
- Unions
- Size of structures
- Bit fields

CHAPTER – 9

Need Multifunction program :

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. Thus a program, which has been written using a number of functions, is treated as a multi-function program.

- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
- The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.
- A function can also be referred as a method or a sub-routine or a procedure, etc.

Elements of user-defined function in C programming

There are multiple parts of user defined function that must be established in order to make use of such function.

- **Function declaration or prototype**
- **Function call**
- **Function definition**
- **Return statement**

Function declaration or prototype

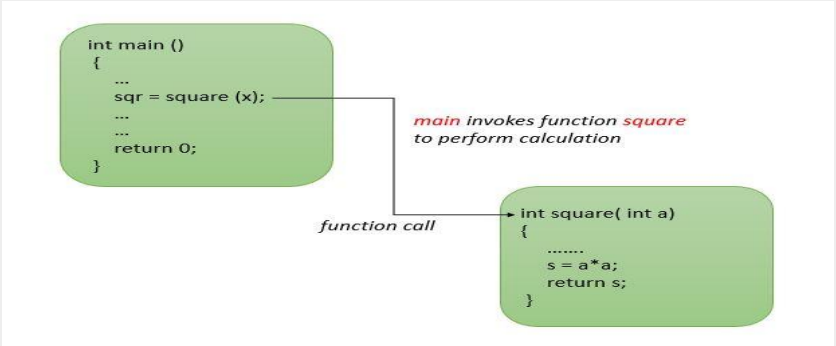
```
int square(int a);   //function prototype
```

- Here, int before function name indicates that this function returns **integer value** to the caller while int inside parentheses indicates that this function will recieve an integer value from caller.

Function Call

Here, function square is called in main

```
sqr = square (x);    //function call
```



Function definition

A function definition provides the actual body of the function.

Syntax of function definition

```
return_value_typefunction_name (parameter_list)

{

  // body of the function

}
```

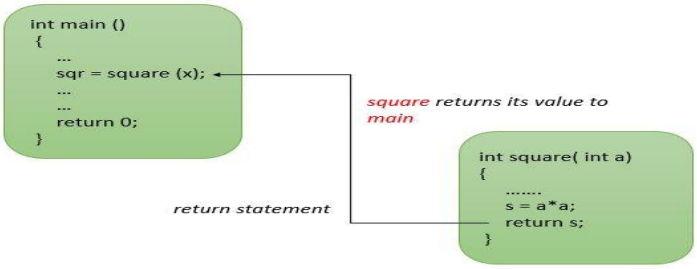
- It consists of a function header and a function body. The function_name is an identifier.
- The return_value_type is the data type of value which will be returned to a caller.
- Some functions performs the desired task without returning a value which is indicated by void as a return_value_type.
- All definitions and statements are written inside the body of the function.

.

Return statement

Return statement returns the value and transfer control to the caller.

```
return s;  //returns the square value s
```



There are three ways to return control.

- **return;**
  The above return statement does not return value to the caller.
- **return expression;**
  The above return statement returns the value of expression to the caller.
- **return 0;**
  The above return statement indicate whether the program executed correctly.

Defining a Function

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

```
return_typefunction_name( parameter list )
 {
   body of the function
}
```

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

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

  /* local variable declaration */
int result;

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

  return result;
}
```

Function Declarations

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts −
  →return_typefunction_name(parameter list);

For the above defined function max(), the function declaration is as follows

int max(int num1, int num2);

→Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

- Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.
- Calling a Function
- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its

function-ending closing brace is reached, it returns the program control back to the main program.

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

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
int a = 100;
int b = 200;
int ret;

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

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

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
int result;

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

   return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

Return values and their types

Return statement returns the value and transfer control to the caller.

```
return s;  //returns the square value s
```

There are three ways to return control.

- **return;**
  The above return statement does not return value to the caller.
- **return expression;**
  The above return statement returns the value of expression to the caller.
- **return 0;**
  The above return statement indicate whether the program executed correctly.

---

function calls

Function Arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function

Sr.No.	Call Type & Description
1	**Call by value** • This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Call by reference** • This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses call by value **to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.**

Category of Functions

- A function depending an whether the arguments are present or not and whether a value is returned or not, may belong to one of following categories

1. Function with no return values, no arguments
2. Functions with arguments, no return values
3. Functions with arguments and return values
4. Functions with no arguments and return values.

function has no arguments.

- It does not receive any data from the calling function. Similarly, it doesn't return any value. The calling function doesn't receive any data from the called function. So, there is no communication between calling and called functions.

function has some arguments .

- It receives data from the calling function, but it doesn't return a value to the calling function. The calling function doesn't receive any data from the called function. So, it is one way data communication between called and calling functions.
  Eg: Printing n Natural numbers

```
01 #include<stdio.h>
02 #include<conio.h>
03 void nat( int);
04 void main()
05 {
```

```
06   int n;
07   clrscr();
08   printf("\n Enter n value:");
09   scanf("%d",&n);
10   nat(n);
11   getch();
12 }
13
14 void nat(int n)
15 {
16   int i;
17   for(i=1;i<=n;i++)
18   printf("%d\t",i);
19 }
```

Output:
Enter n value: 5
1 2 3 4 5
Note:
In the main() function, n value is passed to the nat() function. The n value is now stored in the formal argument n, declared in the function definition and subsequently, the natural numbers upto n are obtained.

Functions with arguments and return values

functions has some arguments and it receives data from the calling function. Simillarly, it returns a value to the calling function. The calling function receives data from the called function. So, it is two-way data communication between calling and called functions.
Eg:

```
01 #include<stdio.h>
02 #include<conio.h
   >
03 int fact(int);
04 void main(
   )
05 {
06   int n;
07   clrscr();
08   printf("\n Enter n:");
09   scanf("%d",&n);
10   printf("\n Factorial of the number : %d", fact(n));
11   getch();
12 }
13
14 int fact(int n)
15 {
16   int i,f;
17   for(i=1,f=1;i<=n;i++)
18   f=f*i;
19   return(f);
20 }
```

Output:

Enter n: 5
Factorial of the number : 120

Functions with no arguments and return values.

In this category, the functions has no arguments and it doesn't receive any data from the calling function, but it returns a value to the calling function. The calling function receives data from the called function. So, it is one way data communication between calling and called functions.

Eg:

view source

print?

```
01 #include<stdio.h>
02 #include<conio.h>
03 int sum();
04 void main()
05 {
06   int s;
07   clrscr();
08   printf("\n Enter number of  elements to be added :");
09   s=sum();
10   printf("\n Sum of the elements :%d",p);
11   getch();
12 }
13
14 int sum()
15 {
16   int a[20], i, s=0,n;
17   scanf("%d",&n);
18   printf("\n Enter the elements:");
19   for(i=0;i< n; i++)
20   scanf("%d",& a[i]);
21   for(i=0;i< n; i++)
22   s=s+a[i];
23   return s;
24 }
```

Nesting of functions

- In some applications, we have seen that some functions are declared inside another function. This is sometimes known as nested function, but actually this is not the nested function. This is called the lexical scoping. Lexical scoping is not valid in C because the compiler is unable to reach correct memory location of inner function.

- Nested function definitions cannot access local variables of surrounding blocks. They can access only global variables. In C there are two nested scopes the local and the global. So nested function has some limited use. If we want to create nested function like below, it will generate error.

Example

```
#include<stdio.h>

main(void){

  printf("Main Function");

  intmy_fun(){

    printf("my_fun function");

    // defining another function inside the first function.

    int my_fun2(){

      printf("my_fun2 is inner function");

    }

  }

  my_fun2();

}
```

Output

```
text.c:(.text+0x1a): undefined reference to `my_fun2'
```

But an extension of GNU C compiler allows declaration of the nested function. For this we have to add auto keyword before the declaration of nested function.

Example

```
#include<stdio.h>

main(void){

  autointmy_fun();

  my_fun();

  printf("Main Function\n");

  intmy_fun(){

    printf("my_fun function\n");

  }

  printf("Done");

}
```

Output

```
my_fun function
Main Function
Done
```

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion(){
  recursion();/* function calls itself */
}

int main(){
  recursion();
}
```

- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.
- Number Factorial
- The following example calculates the factorial of a given number using a recursive function –

```
#include<stdio.h>

unsignedlonglongint factorial(unsignedinti){

if(i<=1){
return1;
}
returni* factorial(i-1);
}

int  main(){
inti=12;
printf("Factorial of %d is %d\n",i, factorial(i));
return0;
}
```

- When the above code is compiled and executed, it produces the following result −
- Factorial of 12 is 479001600

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

```
#include<stdio.h>

intfibonacci(inti){

if(i==0){
return0;
}

if(i==1){
return1;
}
returnfibonacci(i-1)+fibonacci(i-2);
}

int  main(){

inti;

for(i=0;i<10;i++){
printf("%d\t\n",fibonacci(i));
}
```

```
return0;
}
```

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

```
0
1
1
2
3
5
8
13
21
34
```

Passing array to function

- Just like variables, array can also be passed to a function as an argument . In this guide, we will learn how to pass the array to a function using call by value and call by reference methods.

1. Function call by value in C
2. Function call by reference in C

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include<stdio.h>
voiddisp( charch)
{
printf("%c ", ch);
}
int main()
{
chararr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
for (int x=0; x<10; x++)
 {
/* I'm passing each element one by one using subscript*/
disp (arr[x]);
  }

return0;
}
```

Output:

```
a b c d e f g h i j
```

Passing array to function using call by reference

- When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```
#include<stdio.h>
voiddisp( int *num)
{
printf("%d ", *num);
}
```

```
int main()
{
intarr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
for (inti=0; i<10; i++)
    {
/* Passing addresses of array elements*/
disp (&arr[i]);
    }

return0;
}
```

Output:

```
1234567890
```

How to pass an entire array to a function as an argument?

- In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:
- Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that **arr** is equivalent to the **&arr[0]**.

```
#include<stdio.h>
voidmyfuncn( int *var1, int var2)
{
            /* The pointer var1 is pointing to the first element of
             * the array and the var2 is the size of the array. In the
             * loop we are incrementing pointer so that it points to
             * the next element of the array on each increment.
             *
             */
for(int x=0; x<var2; x++)
   {
printf("Value of var_arr[%d] is: %d \n", x, *var1);
/*increment pointer for next element fetch*/
      var1++;
   }
}

int main()
{
intvar_arr[] = {11, 22, 33, 44, 55, 66, 77};
myfuncn(var_arr, 7);
return0;
}
```

Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

Scope

- Scope is defined as the area in which the declared variable is ‘available’. There are five scopes in C: program, file, function, block, and prototype. Let us examine a dummy program to understand the difference (the comments indicate the scope of the specific variable):

```
1    void foo() {}
2    // "foo" has program scope
3    static void bar() {
```

```
4       // "bar" has file scope
5       printf("hello world");
6       inti;
7       // "i" has block scope
8     }
9     void baz(int j);
10    // "j" has prototype scope
11    print:
      // "print" has function scope
```

- The foo function has program scope. All non-static functions have program scope, and they can be called from anywhere in the program. Of course, to make such a call, the function needs to be first declared using extern, before being called, but the point is that it is available throughout the program.

- The function bar has file scope — it can be called from only within the file in which it is declared. It cannot be called from other files, unlike foo, which could be called after providing the external declaration of foo.

- The label print has function scope. Remember that labels are used as a target for jumps using goto in C. There can be only one print label inside a function, and you can write a goto print statement anywhere in the function, even before the label appears in the function. Only labels can have function scope in C.

- The variable i has block scope, though declared at the same level/block as print. Why is that so? The answer is, we can define another variable with the same name i inside another block within the bar function, whereas it is not possible for print, since it is a label.

- The variable j has prototype scope: you cannot declare any other parameter with the same name j in the function baz. Note that the scope of j ends with the prototype declaration: you can define the function baz with the first argument with any name other than j.

Lifetime

➢ The lifetime of a variable is the period of time in which the variable is allocated a space (i.e., the period of time for which it "lives"). There are three lifetimes in C: static, automatic and dynamic. Let us look at an example:

```
1   int foo() {
2     static int count = 0;
3     // "count" has static lifetime
4     int * counter = malloc(sizeof(int));
5     // "counter" has automatic lifetime
6     free(counter);
7     // malloc'ed memory has dynamic lifetime
8   }
```

- In this code, the variable count has a static lifetime, i.e., its lifetime is that of the program. The variable counter has an automatic lifetime — its life is till the function returns; it points to a heap-allocated memory block — its life remains till it is explicitly deleted by the program, which is not predictable, and hence it has a dynamic lifetime.

Visibility

Visibility is the “accessibility” of the variable declared. It is the result of hiding a variable in outer scopes. Here is a dummy example:

```
inti;
// the "i" variable is accessible/visible here
void foo() {
   inti;
   // the outer "i" variable
   // is not accessible/visible here
   {
      inti;
      // two "i" variables at outer scopes
      // are not accessible/visible here
   }
   // the "i" in this block is accessible/visible
   // here and it still hides the outer "i"
}
// the outermost "i" variable
//is accessible/visible here
```

CHAPTER – 10

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly structure is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag]
{

  member definition;
  member definition;
  ...
  member definition;
```

```
} [one or more structure variables];
```

- The structure tag is optional and each member definition is a normal variable definition, such as inti; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books
{
            char  title[50];
            char  author[50];
            char  subject[100];
            intbook _id;
} book;
```

Declaration of Structure Variable

- Just as we declare variables of type int, char etc, we can declare variables of structure as well.Suppose, we want to store the roll no., name and phone number of three students. For this, we will define a structure of name 'student' (as declared above) and then declare three variables, say 'p1', 'p2' and 'p3' (which will represent the three students respectively) of the structure 'student'.

```
struct student
{
          introll_no;
  char name[30];
  intphone_number;
};
main()
{
  struct student p1, p2, p3;
}
```

Here, p1, p2 and p3 are the variables of the structure 'student'.

We can also declare structure variables at the time of defining structure as follows.

```
struct student
{
  introll_no;
  char name[30];
  intphone_number;
}p1, p2, p3;
```

- Now, let's see how to enter the details of each student i.e. roll_no, name and phone number.
- Suppose, we want to assign a roll number to the first student. For that, we need to access the roll number of the first student. We do this by writing

- p1.roll_no = 1;
- This means that use dot (.) to use variables in a structure. p1.roll_no can be understood as roll_no of p1.
- If we want to assign any string value to a variable, we will use strcpy as follows.
- strcpy(p1.name, "Brown");

Accessing Structure Members

- To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program −

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
intbook_id;
};

int main( ) {

struct Books Book1;        /* Declare Book1 of type Book */
struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
```

```
   return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

How to initialize a structure variable?

C language supports multiple ways to initialize a structure variable. You can use any of the initialization method to initialize your structure.

- Initialize using dot operator
- Value initialized structure variable
- Variant of value initialized structure variable

Initialize structure using dot operator

- In C, we initialize or access a structure variable either through dot `.` or arrow `->` operator. This is the most easiest way to initialize or access a structure.

Example:

```
// Declare structure variable
struct student stu1;

// Initialize structure members
stu1.name  = "Pankaj";
stu1.roll  = 12;
stu1.marks = 79.5f;
```

Value initialized structure variable

- The above method is easy and straightforward to initialize a structure variable. However, C language also supports value initialization for structure variable. Means, you can initialize a structure to some default value during its variable declaration.

Example:

```
// Declare and initialize structure variable
struct student stu1 = { "Pankaj", 12, 79.5f };
```

Invalid initialization:

```
// Declare and initialize structure variable
struct student stu1 = { 12, "Pankaj", 79.5f };
```

The above code will throw compilation error. Since the order of member type in structure is character array, integer finally float. But, we aren't initializing the structure variable in the same order.

Variant of value initialized structure variable

- The above approach may suit all needs. In addition, C language supports flexibility to initialize structure members in any order. I know this sounds bit confusing. As, just now I said C will throw error if you try to initialize members in different order of declaration.

This approach is an extension of above. Here, you can specify member name along with the value.

Example:

```
// Declare and initialize structure variable
struct student stu1 = {
   .roll  = 12,
   .name  = "Pankaj",
   .marks = 79.5f
};
```

Structure default initialization

- Default initialization of a variable considered as good programming practice.However, C doesn't support any programming construct for default structure initialization. You manually need to initialize all fields to 0 or `NULL`.
- Initializing all fields to `NULL` is bit cumbersome process. Let's do a small hack to initialize structure members to default value, on every structure variable declaration.

Example:

```
// Define macro for default structure initialization
#define NEW_STUDENT { "", 0, 0.0f }

// Default initialization of structure variable
struct student stu1 = NEW_STUDENT;
```

Program to declare, initialize and access structure

```
/**
 * How to declare, initialize and access structures in C language
 */

#include <stdio.h>

// Macro for default student structure initialization
#define NEW_STUDENT { "", 0, 0.0f }
[

// Student structure type declaration
struct student
{
```

```
   char    name[100];
int     roll;
   float   marks;
};

intmain()
{
   // Declare structure variable with default initialization
struct student stu1 = NEW_STUDENT;

   // Read student details from user
printf("Enter student name: ");
gets(stu1.name);

printf("Enter student roll no: ");
scanf("%d", &stu1.roll);

printf("Enter student marks: ");
scanf("%f", &stu1.marks);


   // Print student details
printf("\n\nStudent details\n");
printf("Name : %s\n",   stu1.name);
printf("Roll : %d\n",   stu1.roll);
printf("Marks: %.2f\n", stu1.marks);

   return 0;
}
```

Output

```
Enter student roll no: 12
Enter student marks: 79.5

Student details
Name : Pankaj Prakash
Roll : 12
Marks: 79.50
```

Copying and Comparing Structure Variables

- Two variables of the same structure type can be copied the same way as ordinary variables.If e1 and e2 belong to the same type, then the following statement is valid. e1 = e2, and e2 = e1; However, the statements that are shown here:e1 < e2; and e1 != e2; are not permitted.C language doesn't permit any logical operations on structure variables.

We can compare two structure variables but comparison of members of a structure can only be done individually.

Write a program to illustate the comparison of structure variables	
#include<stdio.h> #include<conio.h> struct class {	

int number; char name[20];
float marks;
};
main()
{
int x;
//Declaring and initializing structures of 'class' type
struct class student2 = {2, "gita", 78.00};
struct class student3;
student3 = student2; // Copying student2 to student3
if ((student3.number = student2.number) && (student3.marks = student2.marks)) // verifying results of copy
{
printf("\n student2 and student3 are equal");
printf("%d %s %f\n", student3.number, student3.name, student3.marks);
}
else
printf("\n student2 and student3 are different");
}
	student3 = student2; This will copy values of members of student2 to corresponding members of student3

Output:
student2 and student 3 are equal.

Operations on struct variables in C

- In C, the only operation that can be applied to *struct* variables is assignment. Any other operation (e.g. equality check) is not allowed on *struct* variables.
  For example, program 1 works without any error and program 2 fails in compilation.

Program 1

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p2
  printf(" p2.x = %d, p2.y = %d", p2.x, p2.y);
  getchar();
  return 0;
}
```

Program 2

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

```
struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p2
  if (p1 == p2)  // compiler error: cannot do equality check for
                 // whole structures
  {
    printf("p1 and p2 are same ");
  }
  getchar();
  return 0;
}
```

Arrays Of Structures

- Usually, we need a number of records of any kind of structure we declare. Suppose we need a class record consisting of student's name, roll number, age for 60 students. It is difficult to declare 60 structure variables.

- In order to overcome this difficulty we declare an array structure variables. This implies that we store the information of 60 students under a same structure variable name but withdifferent subscript values.

struct class student[60];

These can be accessed as follows:

student[i].sname, student[i].sno, student[i].age.

Write a program to compue the monthly pay of 100 employees using each employee's name, basic pay, DA is computed as 52% of basic salary.

```
01 #include<stdio.h>
02 #include<conio.h>
03 void main()
04 {
05   struct employee
06   {
07     char ename[20];
08     int bp;
09     float da;
10     float gs;
11   }emp[100];
12   int i;
13   for(i=0;i<100;i++)
```

```
14   {
15     printf("\n Enter Emp name:");
16     gets(emp[i].ename);
17     printf("\n Enter Emp Basic pay:");
18     scanf("%d",&emp[i].bp);
19     emp[i].da= 0.52*emp[i].bp;
20     emp[i].gs= emp[i].bp+emp[i].da;
21   }
22   clrscr();
23   for(i=0;i<100;i++)
24   {
25     printf("\n Emp Name      : %s",emp[i].ename);
26     printf("\n Emp Basic pay : %d",emp[i].bp);
27     printf("\n Emp DA        : %f",emp[i].da);
28     printf("\n Emp GS        : %f",emp[i].gs);
29   }
30   getch();
31 }
```

Array within a Structure

- A structure is a data type in **C/C++** that allows a group of related variables to be treated as a single unit instead of separate entities. A structure may contain elements of different data types – int, char, float, double, etc. It may also contain an array as its member. Such an array is called an array within a structure. An array within a structure is a member of the structure and can be accessed just as we access other elements of the structure.

- Below is the demonstration of a program that uses the concept of the array within a structure. The program displays the record of a student comprising the *roll number*, *grade*, and *marks* secured in various subjects. The marks in various subjects have been stored under an array called *marks*. The whole record is stored under a structure called a *candidate*.

```
// C program to demonstrate the
// use of an array within a structure
#include <stdio.h>

// Declaration of the structure candidate
struct candidate {
    introll_no;
    char grade;

    // Array within the structure
    float marks[4];
};
// Function to displays the content of
// the structure variables
void display(struct candidate a1)
{

    printf("Roll number : %d\n", a1.roll_no);
    printf("Grade : %c\n", a1.grade);
```

```
    printf("Marks secured:\n");
    inti;
    intlen = sizeof(a1.marks) / sizeof(float);

    // Accessing the contents of the
    // array within the structure
    for (i = 0; i<len; i++) {
        printf("Subject %d : %.2f\n",
               i + 1, a1.marks[i]);
    }
}
// Driver Code
int main()
{
    // Initialize a structure
    struct candidate A
        = { 1, 'A', { 98.5, 77, 89, 78.5 } };

    // Function to display structure
    display(A);
    return 0;
}
```

Output:

```
Roll number : 1

Grade : A

Marks secured:

Subject 1 : 98.50

Subject 2 : 77.00

Subject 3 : 89.00

Subject 4 : 78.50
```

C Nested Structure(Structure within structure)

- Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.
- The structure variables can be a normal structure variable or a pointer variable to access the data. You can learn below concepts in this section.

1. Structure within structure in C using normal variable
2. Structure within structure in C using pointer variable

1. STRUCTURE WITHIN STRUCTURE IN C USING NORMAL VARIABLE:

- This program explains how to use structure within structure in C using normal variable. “student_college_detail’ structure is declared inside “student_detail” structure in this program. Both structure variables are normal structure variables.
- Please note that members of “student_college_detail” structure are accessed by 2 dot(.) operator and members of “student_detail” structure are accessed by single dot(.) operator.

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

structstudent_college_detail
{
    intcollege_id;
    char college_name[50];
```

```
};

structstudent_detail
{
   int id;
   char name[20];
   float percentage;
   // structure within structure
   structstudent_college_detailclg_data;
}stu_data;

int main()
{
   structstudent_detailstu_data = {1, "Raju", 90.5, 71145,
                                   "Anna University"};
   printf(" Id is: %d \n", stu_data.id);
   printf(" Name is: %s \n", stu_data.name);
   printf(" Percentage is: %f \n\n", stu_data.percentage);

   printf(" College Id is: %d \n",
              stu_data.clg_data.college_id);
   printf(" College Name is: %s \n",
              stu_data.clg_data.college_name);
   return 0;
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Anna University
```

STRUCTURE WITHIN STRUCTURE (NESTED STRUCTURE IN C) USING POINTER VARIABLE:

- This program explains how to use structure within structure in C using pointer variable. “student_college_detail’ structure is declared inside “student_detail” structure in this program. one normal structure variable and one pointer structure variable is used in this program.
- Please note that combination of .(dot) and ->(arrow) operators are used to access the structure member which is declared inside the structure.

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

structstudent_college_detail
{
   intcollege_id;
   char college_name[50];
};

structstudent_detail
{
   int id;
   char name[20];
```

```
    float percentage;
    // structure within structure
    structstudent_college_detailclg_data;
}stu_data, *stu_data_ptr;

int main()
{
  structstudent_detailstu_data = {1, "Raju", 90.5, 71145,
                                  "Anna University"};
    stu_data_ptr = &stu_data;

    printf(" Id is: %d \n", stu_data_ptr->id);
    printf(" Name is: %s \n", stu_data_ptr->name);
    printf(" Percentage is: %f \n\n",
                      stu_data_ptr->percentage);

    printf(" College Id is: %d \n",
                      stu_data_ptr->clg_data.college_id);
    printf(" College Name is: %s \n",
                    stu_data_ptr->clg_data.college_name);

    return 0;
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Anna University
```

Structures and Functions in C

- Like all other types, we can pass structures as arguments to a function. In fact, we can pass, individual members, structure variables, a pointer to structures etc to the function. Similarly, functions can return either an individual member or structures variable or pointer to the structure.

Let's start with passing individual member as arguments to a function.

Passing Structure Members as arguments to Function

We can pass individual members to a function just like ordinary variables.The following program demonstrates how to pass structure members as arguments to the function.

```
1 #include<stdio.h>
2
3 /*
4 structure is defined above all functions so it is global.
5 */
6
7 struct student
  {
```

```
 8 char name[20];
 9 introll_no;
10 int marks;
11 };
12
13 voidprint_struct(char name[], introll_no, int marks);
14
15 intmain()
16 {
17 struct student stu= {"Tim", 1, 78};
18 print_struct(stu.name, stu.roll_no, stu.marks);
19 return0;
20 }
21
22 voidprint_struct(char name[], introll_no, int marks)
23 {
24 printf("Name: %s\n", name);
25 printf("Roll no: %d\n", roll_no);
26 printf("Marks: %d\n", marks);
27 printf("\n");
28 }
29
```

Expected Output:

```
1 Name: Tim
2 Roll no: 1
3 Marks: 78
```

How it works:

- In lines 7-12, a structure student is declared with three members namely `name`, `roll_no` and `marks`.
- In line 14, a prototype of function `print_struct()` is declared which accepts three arguments namely name of type pointer to `char`, `roll_no` of type `int` and `marks` is of type `int`.
- In line 18, a structure variable `stu` of type `struct student` is declared and initialized.
- In line 19, all the three members of structure variable `stu` are passed to the `print_struct()` function. The formal arguments of `print_struct()` function are initialized with the values of the actual arguments.
- From lines 25-27, three `printf()` statement prints `name`, `roll_no` and `marks` of the student.
- The most important thing to note about this program is that `stu.name` is passed as a reference because name of the array is a constant pointer. So the formal argument of `print_struct()` function i.e name and `stu.name` both are pointing to the same array. As a result, any changes made by the function `print_struct()` will affect the original array. We can verify this fact by making the following amendments to our program.

1. In the main function add the following line after the call to print_struct() function.

2. printf("New name: %s", stu.name);

3. In print_struct() function add the following two lines just before the last printf() statement.

```
printf("\nChanging name ... \n");
strcpy(name, "Jack");
```

Now our program should look like this:

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

/*
structure is defined above all functions so it is global.
*/

struct student
{
char name[20];
introll_no;
int marks;
};

voidprint_struct(char name[], introll_no, int marks);

intmain()
{
struct student stu= {"Tim", 1, 78};
print_struct(stu.name, stu.roll_no, stu.marks);

printf("New name: %s", stu.name);

return0;
}

voidprint_struct(char name[], introll_no, int marks)
{
printf("Name: %s\n", name);
printf("Roll no: %d\n", roll_no);
printf("Marks: %d\n", marks);

printf("\nChanging name ... \n");
strcpy(name, "Jack");

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

Expected Output:

```
Name: Tim
Roll no: 1
Marks: 78

Changing name ...

New name: Jack
```

This verifies the fact that changes made by print_struct() function affect the original array.

Passing Structure Variable as Argument to a Function

- In the earlier section, we have learned how to pass structure members as arguments to a function. If a structure contains two-three members then we can easily pass them to function but what if there are 9-10 or more members ? Certainly passing 9-10 members is a tiresome and error-prone process. So in such cases instead of passing members individually, we can pass structure variable itself

The following program demonstrates how we can pass structure variable as an argument to the function.

```
1 #include<stdio.h>
2
3 /*
4 structure is defined above all functions so it is global.
5 */
6
7 struct student
8 {
9 char name[20];
10 introll_no;
11 int marks;
12 };
13
14 voidprint_struct(struct student stu);
15
16 intmain()
17 {
18 struct student stu= {"George", 10, 69};
19 print_struct(stu);
20 return0;
21 }
22
23 voidprint_struct(struct student stu)
24 {
25 printf("Name: %s\n", stu.name);
26 printf("Roll no: %d\n", stu.roll_no);
27 printf("Marks: %d\n", stu.marks);
28 printf("\n");
29 }
```

Expected Output:

```
1 Name: George
2 Roll no: 10
3 Marks: 69
```

How it works:

- In lines 7-12, a structure `student` is declared with three members namely: `name`, `roll_no` and `marks`.
- In line 14, the prototype of function `print_struct()` is declared which accepts an argument of type `struct student`.
- In line 18, a structure variable `stu` of type `struct student` is declared and initialized.
- In line 19, `print_struct()` function is called along with argument `stu`. Unlike arrays, the name of structure variable is not a pointer, so when we pass a structure variable to a function, the formal argument of `print_struct()` is assigned a copy of the original structure. Both structures reside in

different memory locations and hence they are completely independent of each other. Any changes made by function print_struct() doesn't affect the original structure variable in the main() function.

- The printf() statements from lines 25-27 prints the details of the student.

Passing Structure Pointers as Argument to a Function

Although passing structure variable as an argument allows us to pass all the members of the structure to a function there are some downsides to this operation.

1. Recall that a copy of the structure is passed to the formal argument. If the structure is large and you are passing structure variables frequently then it can take quite a bit of time which make the program inefficient.
2. Additional memory is required to save every copy of the structure.

The following program demonstrates how to pass structure pointers as arguments to a function.

```
1 #include<stdio.h>
2
3 /*
4 structure is defined above all functions so it is global.
5 */
6
7 struct employee
8 {
9 char name[20];
10int age;
11chardoj[10]; // date of joining
12char designation[20];
13};
14
15voidprint_struct(struct employee *);
16
17intmain()
18{
19struct employee dev = {"Jane", 25, "25/2/2015", "Developer"};
20print_struct(&dev);
21
22return0;
23}
24
25voidprint_struct(struct employee *ptr)
26{
27printf("Name: %s\n", ptr->name);
28printf("Age: %d\n", ptr->age);
29printf("Date of joining: %s\n", ptr->doj);
30printf("Age: %s\n", ptr->designation);
31printf("\n");
32}
```

Expected Output:

```
1Name: Jin
2Age: 25
3Date of joining: 25/2/2015
```

4Age: Developer

How it works:

- In lines 7-13, a structure employee is declared with four members namely name, age, doj(date of joining) and designation.
- In line 15, the prototype of function print_struct() is declared which accepts an argument of type pointer to struct student.
- In line 19, a structure variable dev of type struct employee is declared and initialized.
- In line 20, print_struct() is called along with along with the address of variable dev. The formal argument of print_struct() is assigned the address of variable dev. Now ptr is pointing to the original structure, hence any changes made inside the function will affect the original structure.
- The printf() statements from lines 27-30 prints the details of the developer.
- The downside of passing structure pointer to a function is that the function can modify the original structure. If that is what you intentionally want that's fine. However, if don't want functions to modify original structure use the const keyword. Recall that const keyword when applied to a variable makes it read-only.
- Let's rewrite the previous program using const keyword.

```
1 #include<stdio.h>
2
3 /*
4 structure is defined above all functions so it is global.
5 */
6
7 struct employee
8 {
9 char name[20];
10int age;
11chardoj[10]; // date of joining
12char designation[20];
13};
14
15voidprint_struct(conststruct employee *);
16
17intmain()
18{
19struct employee dev = {"Jane", 25, "25/2/2015", "Developer"};
20print_struct(&dev);
21
22return0;
23}
24
25voidprint_struct(conststruct employee *ptr)
26{
27printf("Name: %s\n", ptr->name);
28printf("Age: %d\n", ptr->age);
29printf("Date of joining: %s\n", ptr->doj);
30printf("Age: %s\n", ptr->designation);
```

```
31
32//ptr->age = 11;
33
34printf("\n");
35}
```

- Now even though we are passing a structure pointer to print_struct() function, any attempt to modify the values of the structure will result in compilation error. Try commenting out code in line 32 and see it yourself.

Unions

- A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

- To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union[union tag]{
  member definition;
  member definition;
...
  member definition;
}[one or more union variables];
```

- The **union tag** is optional and each member definition is a normal variable definition, such as inti; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
Union Data
{
        inti;
        float f;
        char str[20];
} data;
```

- Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.
- The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because

this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

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

unionData{
inti;
float f;
charstr[20];
};

int main(){

unionDatadata;

printf("Memory size occupied by data : %d\n",sizeof(data));

return0;
}
```

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

Memory size occupied by data : 20

Accessing Union Members

- To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

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

unionData{
inti;
float f;
charstr[20];
};

int main(){

unionDatadata;

data.i=10;
data.f=220.5;
strcpy(data.str,"C Programming");

printf("data.i : %d\n",data.i);
printf("data.f : %f\n",data.f);
printf("data.str : %s\n",data.str);

return0;
}
```

- When the above code is compiled and executed, it produces the following result −
- data.i : 1917853763
- data.f : 4122360580327794860452759994368.000000
- data.str : C Programming
- Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.
- Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

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

unionData{
inti;
float f;
charstr[20];
};

int main(){

unionDatadata;

data.i=10;
printf("data.i : %d\n",data.i);

data.f=220.5;
printf("data.f : %f\n",data.f);

strcpy(data.str,"C Programming");
printf("data.str : %s\n",data.str);

return0;
}
```

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

data.i : 10
data.f : 220.500000
data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

Size of structures

sizeof operator in C

- The sizeof for a struct is not always equal to the sum of sizeof of each individual member. This is because of the padding added by the compiler to avoid alignment issues. Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.

- Different compilers might have different alignment constraints as C standards state that alignment of structure totally depends on the implementation.

Let’s take a look at the following examples for better understanding:

- **Case 1:**

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate
// size of struct
#include <stdio.h>

int main()
{

    struct A {

        // sizeof(int) = 4
        int x;
        // Padding of 4 bytes

        // sizeof(double) = 8
        double z;

        // sizeof(short int) = 2
        short int y;
        // Padding of 6 bytes
    };

    printf("Size of struct: %ld", sizeof(struct A));

    return 0;
}
```

Output:

```
Size of struct: 24
```

- The red portion represents the padding added for data alignment and the green portion represents the struct members. In this case, **x** (int) is followed by **z** (double), which is larger in size as compared to **x**. Hence padding is added after **x**. Also, padding is needed at the end for data alignment.

- **Case 2:**

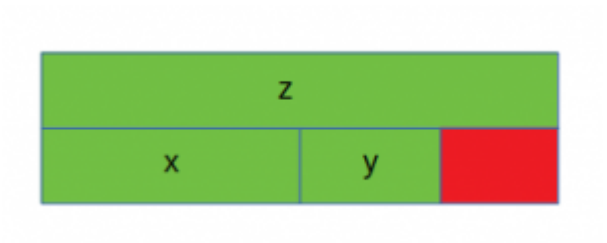
filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate
// size of struct
#include <stdio.h>
```

```
int main()
{

   struct B {
      // sizeof(double) = 8
      double z;

      // sizeof(int) = 4
      int x;

      // sizeof(short int) = 2
      short int y;
      // Padding of 2 bytes
   };

   printf("Size of struct: %ld", sizeof(struct B));

   return 0;
}
```

Output:

```
Size of struct: 16
```



In this case, the members of the structure are sorted in decreasing order of their sizes. Hence padding is required only at the end.

- **Case 3:**

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate
// size of struct
#include <stdio.h>

int main()
{

   struct C {
      // sizeof(double) = 8
      double z;

      // sizeof(short int) = 2
      short int y;
      // Padding of 2 bytes

      // sizeof(int) = 4
      int x;
   };
```

```
    printf("Size of struct: %ld", sizeof(struct C));

    return 0;
}
```

Output:

```
Size of struct: 16
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include −

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting :bit length after the variable. For example −

```
structpacked_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

- Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my_int.
- C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next wo**rd.**

EXPECTED QUESTION:-

2 Mark Question:

1. Differentiate a structure from an array.
2. What is recursion?
3. What is prototype?
4. What is known as bitfield?
5. What is union?
6. What is the feature of " union"?

5 Mark Question

1. Discuss the function with arguments and return values along with example.
2. How to declare the structure variable and discuss on how to access its members?
3. Compare structure and union.
4. Explain scope and visibility of a variable.
5. Write a c program to declare result of an examination using structure for ten students.

6. Write a c program to find out factorial of given number using function

10 Mark Question :

1. Write a detailed note on function in C.
2. Write a c program using structures to handle a student detail.
3. Explain user defined functions? What is passing argument by value and by reference.
4. Explain pass by reference and pass by value in parameters of a function.
5. Write a c program using structure to declare result of an examination for ten students.

Unit- V

POINTERS

1. Pointers

- The Address of a Variables.
- Declaring, initialization of pointer variable
- Accessing Variable through its variable
- Chain of Pointers
- Pointer Increments & Scale Factor
- Pointer and Character strings
- Pointers as Function Argument in C
- Pointer and structure

2. Files

- 2.1. Defining, opening, closing files in c
- 2.2. Input and output operation files in c
- 2.3. Error Handling During I/O Operation
- 2.4.Command Line Argument in C

1. Pointers

Introduction

- The pointer in C language is a variable which stores the address of another variable.
- This variable can be of type int, char, array, function, or any other pointer.
- The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.
- Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;
int* p = &n; // Variable p of type pointer is pointing to
```

the address of the variable n of type integer.

1.1 The Address of a variable

- A program being executed by a processor has two major parts - the code and the data.
- The code section is the code you've written and the data section holds the variables you're using in the program.
- All code and variables are loaded into memory (usually RAM) and the processor executes the code from there.
- Each segment (usually a byte) in the memory has an address - whether it holds code or variable - that's the way for the processor to access the code and variables.
- For example, consider a simple program of adding two numbers
- When the program will run the processor will save these two numbers in two different memory locations.

- Adding these numbers can be achieved by adding the contents of two different memory locations.

- A memory location where data is stored is the address of that data. In C address of a variable can be obtained by prepending the character & to a variable name.
- Try the following program where a is a variable and &a is its address:

Example:1

```
#include <stdio.h>

int main()
{
   int a = 55;

   printf("The address of a is %p", &a);

   return 0;
}
```

Output:

```
The address of a is 0x7fffa3c0d4ec
```

Example 2

```
#include <stdio.h>

void f(int p)
{
   printf("The address of p inside f() is    %p\n", &p);
}

void g(int r)
{
   printf("The address of r inside g() is    %p\n", &r);
   f(r);
}

int main()
{
```

```
    int a = 55;

    printf("The address of a inside main() is %p\n", &a);
    f(a);
    g(a);

    return 0;
}
```

Output

```
The address of a inside main() is 0x7ffc856fed5c
The address of p inside f() is    0x7ffc856fed3c
The address of r inside g() is    0x7ffc856fed3c
The address of p inside f() is    0x7ffc856fed1c
```

1.2 Declaring, initialization of pointer variable

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

Syntax

```
int *a;//pointer to int

char *c;//pointer to char
```

Example

```
#include<stdio.h>

int main(){

int number=50;

int *p;

p=&number;//stores the address of number variable
```

printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.

printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.

return 0;

}

Output

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

1.3. Accessing Variable through its variable

Steps:

- **Declare a normal variable, assign the value**
- **Declare a pointer variable with the same type as the normal variable**
- **Initialize the pointer variable with the address of normal variable**
- **Access the value of the variable by using asterisk (*) - it is known as dereference operator**

Example:

```
#include <stdio.h>
int main(void)
{
	//normal variable
	int num = 100;

	//pointer variable
	int *ptr;

	//pointer initialization
	ptr = &num;

	//pritning the value
	printf("value of num = %d\n", *ptr);

	return 0;
}
```

Output

```
value of num = 100
```

1.4. Chain of Pointers

- ✓ **It is possible to make a pointer to point another pointer, thus creating a chain of pointers.**
- ✓ **For example, in the following program, the pointer variable 'ptr2' contains the address of the pointer variable 'ptr1', which points to the location that contains the desired value.**
- ✓ **This is known as multiple indirections.**
- ✓ **A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.**
- ✓ **The declaration 'int **ptr2' tells the compiler that 'ptr2' is a pointer to a pointer of int type. Remember, the pointer 'ptr2' is not a pointer to an integer, but rather a pointer to an integer pointer.**
- ✓ **We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice.**

Example Program

```
#include <stdio.h>
// C program for chain of pointer

int main()
{
   int var = 10;

   // Pointer level-1
   // Declaring pointer to variable var
   int* ptr1;

   // Pointer level-2
   // Declaring pointer to pointer variable *ptr1
   int** ptr2;

   // Pointer level-3
   // Declaring pointer to double pointer **ptr2
```

```
    int*** ptr3;

    // Storing address of variable var
    // to pointer variable ptr1
    ptr1 = &var;

    // Storing address of pointer variable
    // ptr1 to level -2 pointer ptr2
    ptr2 = &ptr1;

    // Storing address of level-2 pointer
    // ptr2 to level-3 pointer ptr3
    ptr3 = &ptr2;

    // Displaying values
    printf("Value of variable "
        "var = %d\n",
        var);
    printf("Value of variable var using"
        " pointer ptr1 = %d\n",
        *ptr1);
    printf("Value of variable var using"
        " pointer ptr2 = %d\n",
        **ptr2);
    printf("Value of variable var using"
        " pointer ptr3 = %d\n",
        ***ptr3);

    return 0;
}
```

Output:

```
Value of variable var = 10
Value of variable var using pointer ptr1 = 10
Value of variable var using pointer ptr2 = 10
Value of variable var using pointer ptr3 = 10
```

1.5. Pointer Increments & Scale Factor

Pointers can be incremented like.

p1=p1+1;

p1=p2+2; &so on.

- **The expression like p1++;**
- **Will cause the pointer p1 to point to the next value of its type. For ex. If p1 is an integer pointer with an initial value, say 2800, then after with an initial value, the value of p1 will be 2902, & not 2801.**
- **i.e., when we increment a pointer, its value is incremented by the length of the data type that it points to . This length is called the scale factor.**
- **The no of bytes used to store various data types depends on the system & can be found by making use of the size of operator.**

- **Character 1 byte**
- **Integers 2 bytes**
- **Floats 4 bytes**
- **Long integers 4 bytes**
- **Double 8 bytes**

1.6. Pointers as Function Argument in C

- **Pointer as a function parameter is used to hold addresses of arguments passed during function call.**
- **This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable.**

Example Program

Swapping two numbers using Pointer

```
#include <stdio.h>
void swap(int *a, int *b);

int main()
{
   int m = 10, n = 20;
   printf("m = %d\n", m);
   printf("n = %d\n\n", n);
   swap(&m, &n);    //passing address of m and n to the swap function
   printf("After Swapping:\n\n");
   printf("m = %d\n", m);
   printf("n = %d", n);
   return 0;
}

/*
   pointer 'a' and 'b' holds and
   points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
   int temp;
```

```
   temp = *a;
   *a = *b;
   *b = temp;
}
```

m = 10

n = 20

After Swapping:

m = 20

n = 10

1.7. Pointer as a character string

Pointer To Strings

- **A String is a sequence of characters stored in an array. A string always ends with null ('\0') character.**
- **Simply a group of characters forms a string and a group of strings form a sentence.**
- **A pointer to array of characters or string can be looks like the following:**

Example

```
#include <stdio.h>
int main()
{
char *cities[] = {"Iran", "Iraq"};
int i;
```

```
for(i = 0; i < 2; i++)

printf("%s\n", cities[i]);

return 0;

}
```

Output

```
Iran

Iraq
```

- ✓ **In the above pointer to string program, we declared a pointer array of character datatypes and then few strings like "Iran", "Iraq" where initialized to the pointer array (*cities[]).**
- ✓ **Note that we have not declared the size of the array as it is of character pointer type.**
- ✓ **Coming to the explanation, cities[] is an array which has its own address and it holds the address of first element (I (Iran)) in it as a value.**
- ✓ **This address is then executed by the pointer, i.e) pointer start reading the value from the address stored in the array cities[0] and ends with '\0' by default.**
- ✓ **Next cities[1] holds the address of (I (Iraq).This address is then executed by the pointer, i.e) pointer start reading the value from the address stored in the array cities[1] and ends with '\0' by default.**
- ✓ **As a result Iran and Iraq is outputted.**

1.8. Pointer and structure

- we can also have array of structure variables.
- And to use the array of structure variables efficiently, we use pointers of structure type.
- We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

Syntax:

```
struct name {
   member1;
   member2;
   .
   .
};

int main()
{
   struct name *ptr, Harry;
}
```

Here, ptr is a pointer to struct.

Example: Access members using Pointer

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

```
{
  int age;
  float weight;
};

int main()
{
   struct person *personPtr, person1;
   personPtr = &person1;

   printf("Enter age: ");
   scanf("%d", &personPtr->age);

   printf("Enter weight: ");
   scanf("%f", &personPtr->weight);

   printf("Displaying:\n");
   printf("Age: %d\n", personPtr->age);
   printf("weight: %f", personPtr->weight);
   return 0;
}
```

In this example, the address of person1 is stored in the personPtr pointer using personPtr = &person1;.

Now, you can access the members of person1 using the personPtr pointer.

2. Files

- **A file represents a sequence of bytes on the disk where a group of related data is stored.**
- **File is created for permanent storage of data. It is a readymade structure.**
- **In C language, we use a structure pointer of file type to declare a file.**

Syntax

```
FILE *fp;
```

- **C provides a number of functions that helps to perform basic file operations.**

Following are the functions,

Function	description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file

rewind() set the position to the beginning point

2.1. Defining, Opening a File or Creating a File, closing file

The fopen() function is used to create a new file or to open an existing file.

<u>General Syntax:</u>

***fp = FILE *fopen(const char *filename**

- **Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.**
- **filename is the name of the file to be opened and mode specifies the purpose of opening the file. Mode can be of following types,**

mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The fclose() function is used to close an already opened file.

General Syntax :

int fclose(FILE *fp);

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file.

This EOF is a constant defined in the header file stdio.h.

Input/Output operation on File

- **In the above table we have discussed about various file I/O functions to perform reading and writing on file.**
- **getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.**

Example program

```
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp = fopen("one.txt", "w");
   printf("Enter data...");
   while( (ch = getchar()) != EOF) {
      putc(ch, fp);
   }
```

```
   fclose(fp);
   fp = fopen("one.txt", "r");
   while( (ch = getc(fp)! = EOF)
   printf("%c",ch);
   // closing the file pointer
   fclose(fp);
   return 0;
}
```

2.2 Input and output operation files in c

Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>
struct emp
{
   char name[10];
   int age;
};
void main()
{
   struct emp e;
   FILE *p,*q;
   p = fopen("one.txt", "a");
   q = fopen("one.txt", "r");
   printf("Enter Name and Age:");
```

```
   scanf("%s %d", e.name, &e.age);
   fprintf(p,"%s %d", e.name, e.age);
   fclose(p);
   do
   {
      fscanf(q,"%s %d", e.name, e.age);
      printf("%s %d", e.name, e.age);
   }
   while(!feof(q));
}
```

In this program, we have created two FILE pointers and both are refering to the same file but in different modes.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

2.3. Error Handling During I/O Operation

While dealing with files, it is possible that an error may occur. This error may occur due to following reasons:

- **Reading beyond the end of file mark.**
- **Performing operations on the file that has not still been opened.**
- **Writing to a file that is opened in the read mode.**
- **Opening a file with invalid filename.**
- **Device overflow.**

Thus, to check the status of the pointer in the file and to detect the error is the file. C provides two status-enquiry library functions

feof() - The feof() function can be used to test for an end of file condition

Syntax

feof(FILE *file_pointer);

Example

if(feof(fp))

printf("End of file");

ferror() - The ferror() function reports on the error state of the stream and returns true if an error has occurred.

Syntax

ferror(FILE *file_pointer);

Example

if(ferror(fp)!=0)

printf("An error has occurred");

2.4. Command Line Argument in C

- **Command line argument is a parameter supplied to the program when it is invoked.**
- **Command line argument is an important concept in C programming.**
- **It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.**

Syntax:

int main(int argc, char *argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   }
   else
   {
      printf("argument list is empty.\n");
   }
```

```
    return 0;
}
```

- **Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument.**
- **If no argument is supplied, argc will be 1.**