
15CS31T - : PROGRAMMING WITH C

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT -1 : Introduction To 'C' Language

Brief history of C language

C was developed at Bell Laboratories in 1972 by Dennis Ritchie. Many of its principles and ideas were taken from the earlier language B and B's earlier ancestors BCPL and CPL. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both high level, machine independent programming and would still allow the programmer to control the behaviour of individual bits of information. The one major drawback of CPL was that it was too large for use in many applications. In 1967, BCPL (Basic CPL) was created as a scaled down version of CPL while still retaining its basic features. In 1970, Ken Thompson, while working at Bell Labs, took this process further by developing the B language. Finally in 1972, a co-worker of Ken Thompson, Dennis Ritchie, returned some of the generality found in BCPL to the B language in the process of developing the language we now know as C.

In 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C which became known as ANSI Standard C. Today C is in widespread use with a rich standard library of functions.

C programming language was developed by Dennis Ritchie in 1972.

C language is improved version of B language where B language is improved version of BCPL(Basic Combined programming Language)

The letter 'C' is the second letter of BCPL. Hence C stands for “combined”.

C is a high level language.

C can be used for developing application programs and OS and also for processing scientific and engineering data.

Features of C language

1. C supports variety of data types.
2. It supports powerful operators.
3. It has rich set of built-in functions.
4. It has 32 keywords/reserved words.
5. C is highly portable.
6. Allows user to add functions to C library.
7. C is faster than BASIC
8. C is case sensitive.
9. Highly flexible in developing application programs and OS.
10. It supports pointers.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

1.1 C character set.

Character set: The character set is the fundamental raw material of any language and they are used to represent information. Like natural languages, computer language will also have well defined character set, which is useful to build the programs.

Character Set Consists Of:

Types	Character Set
Lowercase Letters	a-z
Uppercase Letters	A to Z
Digits	0-9
Special Characters	!@#\$\$%^&*
White Spaces	Tab Or New line Or Space

Valid C Characters : Special Characters are listed below:

Symbol	Meaning
~	Tilde
!	Exclamation-mark
#	Number-sign
\$	Dollar-sign
%	Percent-sign
^	Caret
&	Ampersand
*	Asterisk
(Left-parenthesis

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

)	Right-parenthesis
—	Underscore
+	Plus-sign
	Vertical bar
\	Backslash
`	Apostrophe
—	Minus sign
=	Equal to sign
{	Left brace
}	Right brace
[Left bracket
]	Right bracket
:	Colon
”	Quotation mark
;	Semicolon
<	Opening angle bracket
>	Closing angle bracket
?	Question mark
,	Comma

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

.	Period
/	Slash

1.2 Variables and Identifiers:

1.2.1 Variables:

Variable in C Programming is also called as container to store the data. Variable name may have different data types to identify the type of value stored. Suppose we declare variable of type integer then it can store only integer values. Variable is considered as one of the building block of C Programming which is also called as identifier.

A Variable is a name given to the memory location where the actual data is stored.

Variables are those whose value may vary during program execution.

Variables are represented by symbolic names.

Example: num1, sum, total.

Examples for valid variable names:

Rama
Gptmbl_
_1947
key_12
pralaya12

Examples for invalid variable names

12_diode: invalid because first letter cannot be a digit.

nav@gmail: invalid because it contains a special character @.

continue: invalid because continue is a keyword.

Naveen kumar: invalid because it contains a special character(blank space).

1.2.2 Identifiers:

Identifiers are the names of variables denoting values, labels, functions, arrays etc.

Identifier must be unique. They are created to give unique name to identify it during the execution of the program. For example:

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
int money;  
  
double accountBalance;
```

Here, `money` and `accountBalance` are identifiers.

Also remember, identifier names must be different from keywords. You cannot use `int` as an identifier because `int` is a keyword.


Rules for writing an identifier:

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore. However, it is discouraged to start an identifier name with an underscore.
3. There is no rule on length of an identifier. However, the first 31 characters of identifiers are discriminated by the compiler.

Identifier Names


✱ Some correct identifier names are -

arena, s_count
marks40
class_one



✱ Some erroneous identifier names are -

1stsst
oh!god
start....end



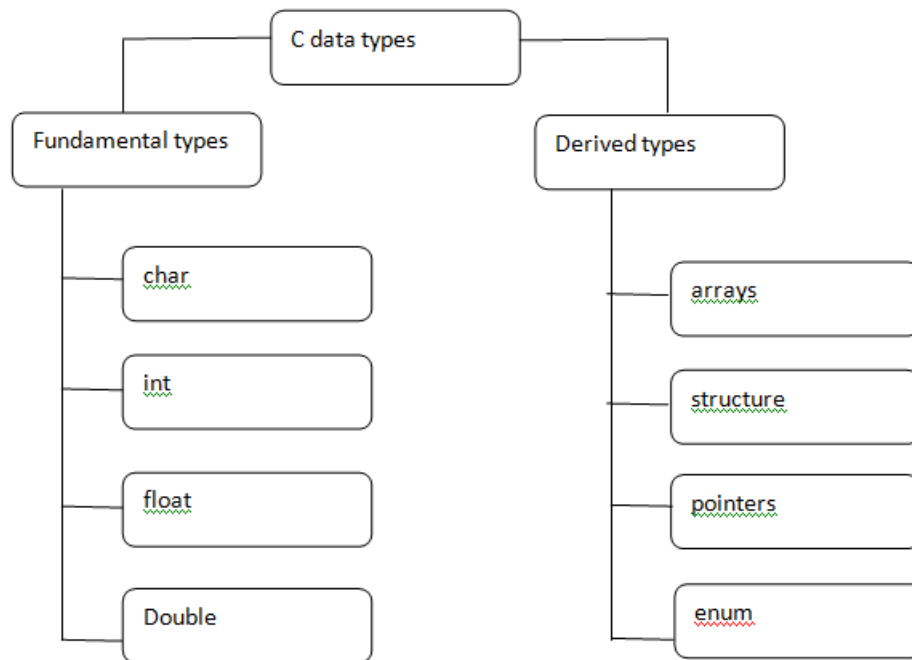
1.3 Built-in Data Types

Data types refers to the type and size of data associated with variables.

Built in data types are also known as fundamental or basic data types.

C data types can be divided into:

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus



The four built in data types are,

1. char
2. int
3. float
4. double

1.3.1 char

- The variable of char data type holds a character constant. A character constant is a character enclosed within a pair of single quotes.
- Char data type reserves 1 byte of memory for the variable storage.
- Example: char s1,s2;

Here s1 and s2 are variables of char data type.

A character denotes any alphabet, digit or special symbol used to represent information. Below figure shows the valid alphabets, numbers and special symbols allowed in C.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Alphabets	A, B,, Y, Z a, b,, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * () _ - + = \ { } [] : ; " ' < > , . ? /

Character Data Types		
Type	Size (in bytes)	Range
char	1	- 128 to 127
Signed char	1	- 128 to 127
unsigned char	1	0 to 255

1.3.2 int

Integers are whole numbers that can have both positive and negative values but no decimal values. Example: 0, -5, 10

- A variable of int data type holds integer numbers.
- int data type reserves 2 bytes of memory for the variable storage.
- Example: int n1, n2;

Here n1 and n2 are variables of int data type.

1.3.3 float

Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc.

- A variable of float data type holds real (fractional) numbers.
- float data types reserves 4 bytes of memory for the variable storage.
- Variable of float data type holds a number with six decimal digits of precision.
- Example: float length, height;

Here length and height are variables of float data type.

1.3.4 Double

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Double type variables can hold real numbers such as: 475.25, -78.85, 345.0 etc.

- A variable of double data type holds real or fractional numbers with high precision than float data type .
- In double data type the fractional numbers are stored in exponent form.
- Variable of double data type holds a number with ten decimal digits of precision.
- double data type reserves 8 bytes of memory for the variable storage.
- Example: double voltage, percentage;

Here voltage and percentage are variables of double data type.

Note: By default a real number is treated as a double.

Data types in C

Data Type	Memory Requirement	Range
char	1 byte	-128 to +127
short or int	2 bytes	-32768 to +32767
long	4 bytes	-2147483648 to +2147483647
float	4 bytes	-3.4e38 to +3.4e38
double	8 bytes	-1.7e308 to +1.7e308

1.4 Variable Definition, Declaration.

Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working.

- Variables are those whose value may vary during program execution.

A variable declaration indicates the data type of the variables and number of memory locations required for variable storage.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Any variable to be used in the program must be declared using declaration statement.

General syntax: `data_type variable_list;`

The data types can be int, float, double, char etc.

Examples for variable declarations.

1. `int marks1, marks2;`

2. `float average;`

Here int and float are data types.

marks1, marks2 and average are variables.

1.5 C Key Words-Rules & Guidelines for Naming Variables.

1.5.1 C Keywords or reserved words.

- Keywords are those whose meaning is already defined in C compiler.
- Keywords are also called as reserved words.
- Keywords are written in lower case.
- Keywords cannot be used as identifiers. Compilers show error message when keywords are declared as variables/ constants.
- They cannot store values
- C has 32 keywords.

Example : int, float, char ,if etc.

List of 32 keywords :

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

1.5.2 Rules and guidelines for naming variables:

- 1) A variable name can contain digits, alphabets and underscores.
- 2) The first character must be an alphabet or underscore.
- 3) Maximum number of characters in variable name is 8.
- 4) Variable names are case sensitive.
- 5) Variable name must not contain any special character except underscore.
- 6) Variable name must not be a reserved word.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

1.6 Arithmetic operators and Expressions.

An operator is a symbol which operates on a value or a variable. For example: `+` is an operator to perform addition.

C programming has wide range of operators to perform various operations.

1.6.1. Arithmetic operators

Arithmetic operators perform arithmetic operations like addition, subtraction, multiplication and division. Arithmetic operators are as shown in below.

Symbol used	Operations
<code>+</code> (plus)	Adds two numbers
<code>-</code> (minus)	Subtract two number
<code>*</code> (asterisk)	Multiplies two number
<code>/</code> (slash)	Divides two numbers
<code>%</code> (percent)	gives remainder of integer division

- These operators are called binary operators because they operate on two operands.
- `+`, `-`, `*`, `&`, `/` operators operate on int, float and double data type variables.

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20, then :

Operator	Description	Example
<code>+</code>	Adds two operands.	$A + B = 30$
<code>-</code>	Subtracts second operand from the first.	$A - B = 10$
<code>*</code>	Multiplies both operands.	$A * B = 200$
<code>/</code>	Divides numerator by de-numerator.	$B / A = 2$
<code>%</code>	Modulus Operator and remainder of after an integer division.	$B \% A = 0$

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

1.6.2. Arithmetic expressions

Arithmetic expression contains numeric variables and constants associated with arithmetic operators.

Ex; $2*x-3*y+z$

Modes of Arithmetic Expressions.

There are 3 types

1. Integer mode arithmetic expressions.
2. Real (floating – point) mode arithmetic expression.
3. Mixed mode arithmetic expression.

1. Integer mode arithmetic statement /expression.

This expression contains integer operands (constants or variable). An arithmetic operation between

integer operands results in integer value only .

Ex; int A,B,Sum;

Sum=A+B;

2. Real mode arithmetic statement/expression.

These expressions contain real type (floating –point or double data type) operands. An arithmetic operation between real operands results in real value only.

Ex; float a, b, sum;

sum =a+b;

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

3. Mixed mode arithmetic statement/expression

These expressions contain mixture of int and real data type operands. An arithmetic operation between integer and real operands results in real value only.

```
int total =10
```

```
float avg ,sum =55.6
```

```
avg = sum /total ;/*Avg gets real/fractional value*/
```

1.7 Constants and Literals.

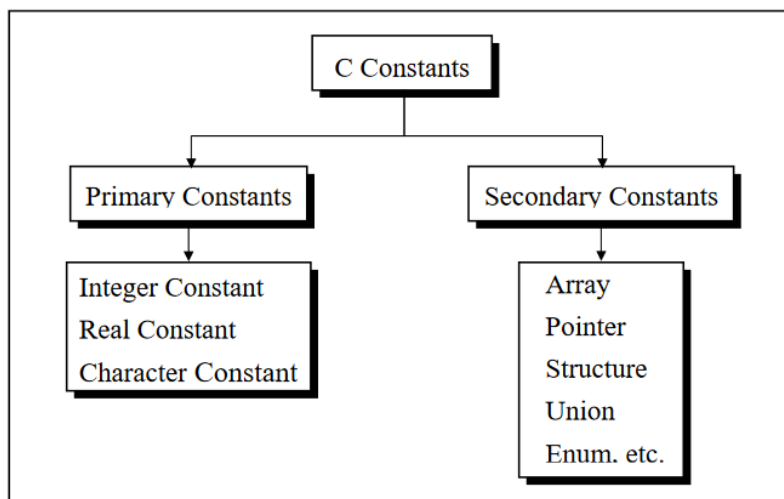
Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called *literals*. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string *literal*.

Types of C Constants

C constants can be divided into two major categories:

- (a) Primary Constants
- (b) Secondary Constants

These constants are further categorized as shown in Figure




1.8 Precedence and Order of Evaluation.

The pre-defined order in which the arithmetic operations are carried out in an arithmetic expression is called the *precedence of arithmetic operators*.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Hierarchy rules are given below.

- 
0. Zero => contents within parenthesis are evaluated first.
 1. First => unary minus operator(-)
 2. Second => multiplication, division and modulus operators(*,/,%)
 3. Third => addition, subtraction operators (+,-)
 4. Last => the assignment operator(=)

Priority	Operators	Description
1 st	* / %	multiplication, division, modular division
2 nd	+ -	addition, subtraction
3 rd	=	assignment

If the expression contains the operators of same hierarchy, then the expression is evaluated from left to right.

If the expression contains many parentheses, then contents of inner most parenthesis is evaluated first, the next innermost as second step and so on.

Example:

$$\begin{aligned}
 &\Rightarrow 2 + \underline{1 * 3} - 4 \% 3 * 1 + 16 / 2 \% 5 \\
 &\Rightarrow 2 + \underline{3} - \underline{4 \% 3} * 1 + 16 / 2 \% 5 \\
 &\Rightarrow 2 + 3 - \underline{1 * 1} + 16 / 2 \% 5 \\
 &\Rightarrow 2 + 3 - 1 + \underline{16 / 2} \% 5 \\
 &\Rightarrow 2 + 3 - 1 + \underline{8 \% 5} \\
 &\Rightarrow \underline{2 + 3} - 1 + 3 \\
 &\Rightarrow \underline{5 - 1} + 3 \\
 &\Rightarrow 4 + 3 \\
 &\Rightarrow 7
 \end{aligned}$$

1.9 Simple assignment statement

An **assignment statement** gives a value to a variable. For example,

$x = 5;$

gives x the value 5.

The value of a variable may be changed. For example, if x has the value 5, then the assignment statement

$x = x + 1;$

will give x the value 6.

The general syntax of an assignment statement is

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

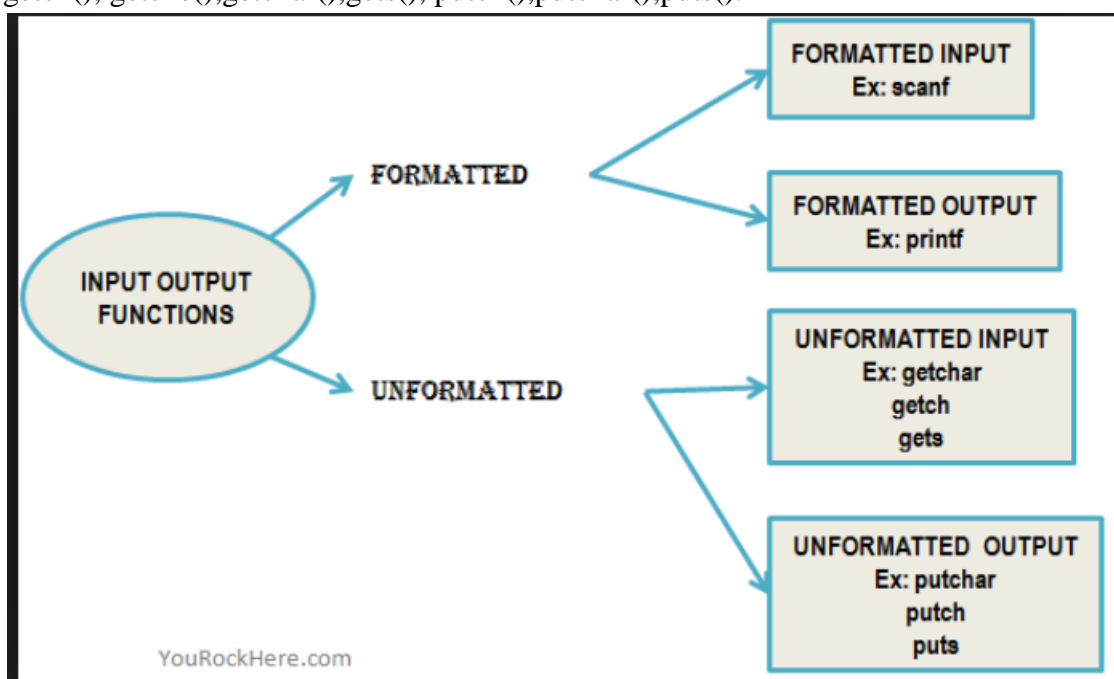
```
variable = expression;
```

1.10 Basic input/output statement

C supports standard built-in functions to read the data from input devices (like keyboard) and to display the data on output devices (like monitor).

These functions are classified as :

1. Formatted I/O functions
scanf() and printf()
2. Unformatted I/O functions
getch(), getche(), getchar(), gets(), putchar(), puts().



Format specifier/ Conversion specifiers:

Format specifiers indicate the data type of the value to be read or displayed when used in formatted I/O functions (scanf() and printf()).

The list of format specifiers and associated data types are listed below.

Format specifier	Data type
%d	decimal integer(signed)
%f	floating-point
%c	character constant
%s	string constant

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

%u	unsigned decimal integer
%o	unsigned octal number
%x	unsigned hexadecimal number
%ld	long decimal number
%e	Floating point number in exponent form

1.10.1 Formatted I/O function

These use format specifiers like (%d,%f,%c) to specify the data type of the variable to be read or displayed. scanf() and printf() are formatted I/O functions.

scanf() function:

- It is an input function
- It is used to read the value from keyboard.
- scanf stands for scan formatted or scan function.

The general form or general syntax

```
scanf("control string",arg1,arg2,arg3,.....argn);
```

As shown above scanf() function contains two parts :control string and argument list

Control string:

It contains format specifiers to indicate the data types of the values to be read. These format specifiers are %d,%f,%c etc.

Format specifiers are always included within double quotes.

Argument list

This contains variables names into which the read values are stored.

The data type of format specifiers and variables must match.

Example:

```
1)scanf("%d", &num1);
```

Here one integer is read from keyboard and stored in the variable num1.

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

Here two integer values are read and stored in the variables a and b.

printf() function

- It is an output function
- It is used to display the values or text message on monitor.
- printf stands for print formatted or print function.

The general form or general syntax

```
printf("control string",arg1,arg2,arg3,.....argn);
```

As shown above printf() function contains two parts :control string and argument list

Control string:

It may contain format specifiers or any text message or both. These format specifiers are %d,%f,%c etc.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Format specifiers and text messages are always included within double quotes.

Argument list

This contains variables names whose values are to be displayed.

The data type of format specifiers and variables must match.

Example:

```
1)printf("Hardwork is the key to success");
```

output

Hardwork is the key to success

```
2) float a=56.565
```

```
Printf(" The area of the circle is=%f",a);
```

output

The area of the circle is=56.565000

1.10.2 Unformatted I/O function

These do not use format specifiers .The default data type is char. These are used for reading or displaying the value of character constant or string constant. Examples: getch(), getche(), getchar()/fgetchar(), putchar() and puts().

getch()

- It is an unformatted input function.
- It is used to read a character constant from keyboard.
- It does not echo typed character on the screen.

Ex: char p;
p=getch();

getche()

- It is an unformatted input function.
- It is used to read a character constant from keyboard.
- It echoes typed character on the screen.

Ex: char p;
p=getche();

getchar()/fgetchar()

- It is an unformatted input function.
- It is used to read a character constant from keyboard.
- It echoes typed character on the screen.

Ex: char p;
p=getche();

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

gets()

- It is an unformatted input function.
- It is used to read a character string from keyboard.
- End of the string is indicated by pressing ENTER key

putch()

- It is an unformatted output function.
- It is used to display a character constant on monitor.

Example: `putch('g');`/* it will display **g** on the monitor*/

puts()

- It is an unformatted output function
- It is used to display a sequence of characters(string) on monitor.

Example: `puts(" Hi! Good morning");`

It will display **Hi! Good morning** on the monitor

1.11 Simple 'C' programs.

1. /*Program to find sum of two variables*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;                                //declaration of variables
    printf("enter 1st integer value\n");
    scanf("%d",&a);
    printf("enter 2nd integer value\n");
    scanf("%d",&b);
    c=a+b;
    printf("sum of given values is %d\n", c);
    getch();
}
```

Output:

```
enter 1st integer value
10
enter 2nd integer value
20
sum of given values is 30
```

2. /*Program to swap the values of 2 variables without using third variable*/

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

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
void main()
{
clrscr();
int A, B;
printf("Enter value of A and B \n");
scanf("%d%d", &A,&B);
printf("Before swapping A=%d and B=%d \n",A,B);
A=A+B; /*formula*/
B=A-B;
A=A-B;
printf("After swapping A=%d and B=%d \n", A, B);
getch();
}
```

Output

Enter value of A and B

-34

47

Before swapping A=-34 and B=47

After swapping A=47 and B=-34

3.write a C program to read 3 integers and to print sum and average of three integer numbers.

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

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

```
void main()
```

```
{
```

```
clrscr();
```

```
int a,b,c,sum;
```

```
float average;
```

```
printf("Enter any 3 integer numbers\n");
```

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

```
sum=a+b+c;
```

```
average=sum/3;
```

```
printf("The sum of 3 numbers =%d",sum);
```

```
printf("The average of three numbers=%f",average);
```

```
getch();
```

```
}
```

Output

Enter any 3 integer numbers

5

20

12

The sum of 3 numbers =37

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

The average of three numbers=12.333333

4. Write a C language to accept temperature in Fahrenheit and to convert it in to centigrade.

Formula: $C = (F - 32) / 1.8$

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
float fah_temp,cent_temp;
printf("Enter the temperature value in Fahrenheit \n");
scanf("%f", &fah_temp);
cent_temp=(fah_temp-32)/1.8;
printf("Equivalent temperature value in centigrade =%f",cent_temp);
getch();
}
```

Output

Enter the temperature value in Fahrenheit

68.0

Equivalent temperature value in centigrade =20.000000

1.12 Algorithms – Definition and Characteristics.

An algorithm is a finite set of instructions for solving a problem.

Characteristics of an Algorithm

Design

- Each step of an algorithm must be exact, precisely and ambiguously described.
- It must terminate, i.e. it contains a finite number of steps.
- It must be effective, i.e., produce the correct output.
- It must be general, i.e., to solve every instance of the problem.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

1.13 Simple algorithms.

1. Algorithm to find sum of 2 numbers.

Step 1. Start
Step 2. Read n_1 .
Step 3. Read n_2 .
Step 4. $\text{Sum} = n_1 + n_2$.
Step 5. Print sum.
Step 6. Stop.

2. Algorithm to find sum and average of 3 numbers.

Step 1. Start
Step 2. Read n_1, n_2, n_3 .
Step 3. $\text{Sum} = n_1 + n_2 + n_3$.
Step 4. $\text{avg} = (n_1 + n_2 + n_3) / 3$.
Step 5. Print sum.
Step 6. Print avg.
Step 7. Stop.

1.14 Flow chart – Type of flow chart.

A flowchart is a pictorial representation of an algorithm in which steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows.

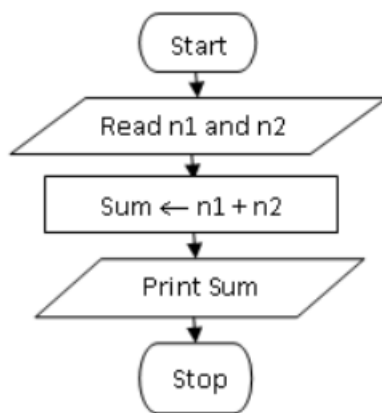
Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Types of Flowchart

Document Flowchart	shows controls over a document-flow through a system
Data Flowchart	shows controls over a data-flow in a system
System Flowchart	shows controls at a physical or resource level
Program Flowchart	shows the controls in a program within a system

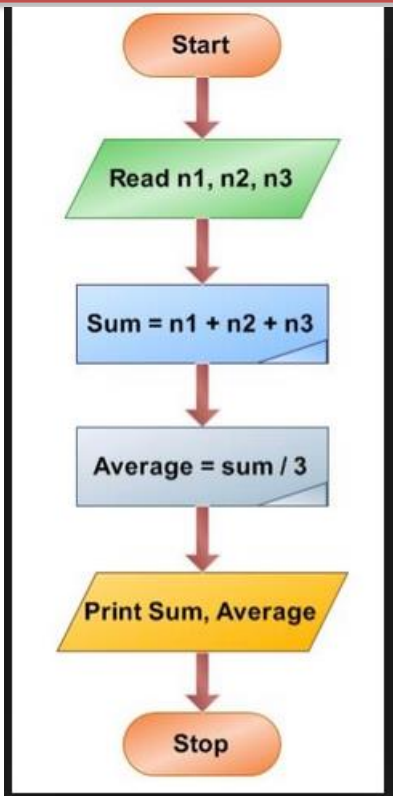
1.15. Simple flow charts

1.15.1 Flowchart to find sum of two numbers.



1.15.2 Flowchart to find sum and average of three numbers.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus



Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT -2 : Decision making- Branching and Looping

2.1 C Operators

1. Conditional Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Bitwise Operators
7. Special Operators
8. Arithmetic operators

2.1.1 Conditional Operators: A ternary operator pair “ ? : ” is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

where exp1,exp2 and exp3 are expressions.

The ? : Operators works as follows

- exp1 is evaluated first.
- If it is true, then the exp2 is evaluated and becomes the value of the expression.
- If exp1 is false then exp3 is evaluated and becomes the value of the expression.

Example: $a=10, b=15;$

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

In this example x is assigned the value of b expression $a>b$ is evaluated to false.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Write a c program to find smallest & largest of three numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int a,b,c,small,large;
    clrscr();
    printf("Enter values of a,b, and c\n");
    scanf("%d%d%d",&a,&b,&c);
    small=(a<b)?a:b;
    small=(small<c)?small:c;
    printf("The smallest number is %d\n",small);
    large=(a>b)?a:b;
    large=(large>c)?large:c;
    printf("The largest number is %d\n",large);
    getch();
}
```

2.1.2 Relational operators

Relational operators are used to compare the relationship between two operands (variables or constants). C provides 6 relational operators as given below.

SYMBOL	MEANING	EXAMPLE int p=5;
<	Less than	p<10; true
<=	Less than or equal to	p<=10; true
>	Greater than	p>10; false
>=	Greater than or equal to	p>=10; false
==	Equal to	p==10; false
!=	Not equal to	p!=10; true

The relational expression evaluates to 1 if the relation is TRUE and evaluates to 0 if the relation is FALSE.

The hierarchy level of relational operators is given below

FIRST	<, <=, >, >=.
SECOND	==, !=

The relational operators have lower precedence level than the arithmetic operators.

2.1.3 Logical operators.

C provides three logical operators as given below

Symbol	Meaning	Example
&&	Logical AND	marks>35 && marks<60
	Logical OR	marks==35 marks >35
!	Logical NOT	!(Marks<35)

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

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

- The result of Logical AND is 1 when both the conditions are TRUE otherwise 0.
- The result of Logical OR is 0 when both the conditions are FALSE otherwise 1
- NOT operator logically negates the given value of the expression

Note: NOT operator has higher precedence than arithmetic and rational operators

2.2 Decision Making and Branching

Decision making helps us to change the order of execution of the statements based on certain conditions. Decision making capabilities are supported by following statements

1. if statement
 - i) simple if
 - ii) if...else
 - iii) Nested if...else
 - iv) else if ladder
2. Switch statement
3. Conditional operator statement
4. goto statement

2.2.1 Simple if statement

General form/Syntax:

```
if (test expression)
{
    Statement-block;
}
Statement-x;
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

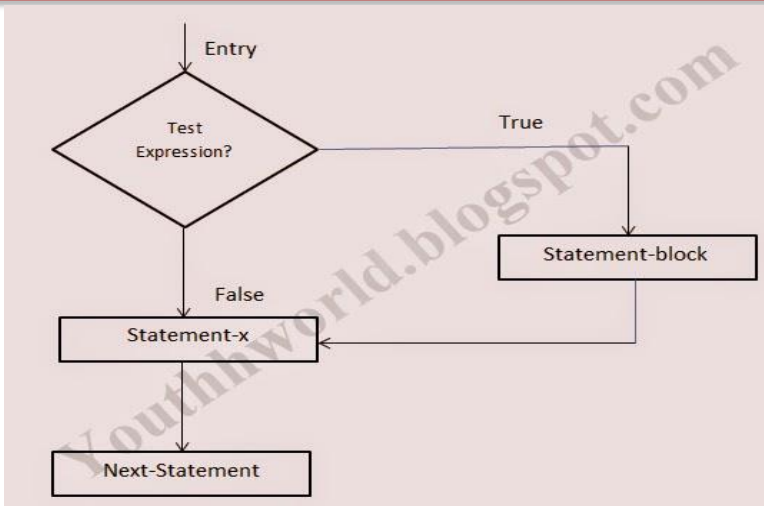


Fig 2.1 Flow chart of simple if control

where test-expression can be a logical or relational or arithmetic expression that results in either TRUE or FALSE

Working/Execution:

1. First, evaluates the test expression.
2. If the result is TRUE , statement block is executed and continues execution with statement-x
3. If it is FALSE, statement- block is skipped from execution and continues execution with the statement-x.

Example:

```

if(a>10)
{
  x=100;
}
Y=50;
  
```

if the value of a is greater than 10 then x is assigned value 100 and y is 50. If a is less than 10 only y is assigned 50.

2.2.2 if else Statement:

General Form /Syntax:

if(test expression)

```

{
  Statement_1;
}
else
{
  Statement_2;
}
  
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
}  
statement-x;
```

Working/Execution:

If the test expression is true, then the statement₁ is executed; otherwise, statment2 is executed. In both the cases the control is transferred subsequently to the statement-x.

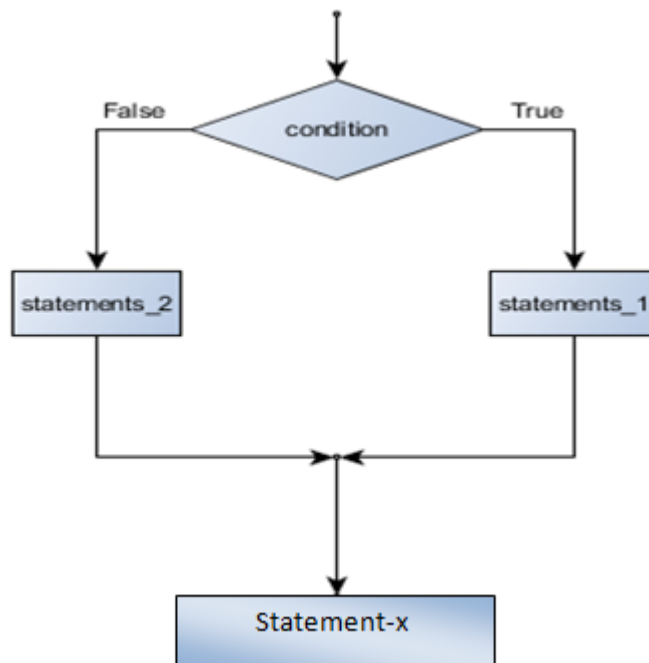


Fig 2.2. Flow chart of if else statement

Example:

```
if(a>b)  
{  
    printf("%d",a);  
}  
else  
{  
    printf("%d",b);  
}  
printf("thank you");
```

If the test expression $a > b$ is evaluated to true then the value of "a" is printed. If it is evaluated to false then value of b is printed. In both the cases "thank you" statement is printed.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

/*Write a C program to find whether the number is positive or negative */

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num;
    clrscr();
    printf("Enter any integer number \n");
    scanf("%d",&num);
    if(num>0)
        printf("Given number is Positive \n");
    else
        printf("Given number is Negative \n");
    getch();
}
```

Output

Enter any integer number

-33

Given number is Negative

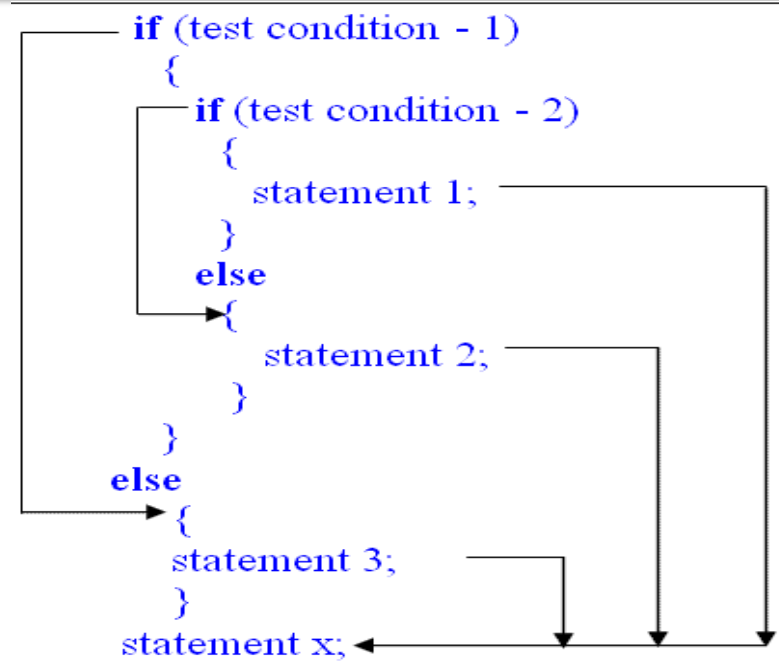
2.2.3 Nested if else Statement:

Nested if...else statement can be used to choose an option from more than two alternatives.

When an if statement is placed in another if statement, it is called nested if statement.

General Form /Syntax:

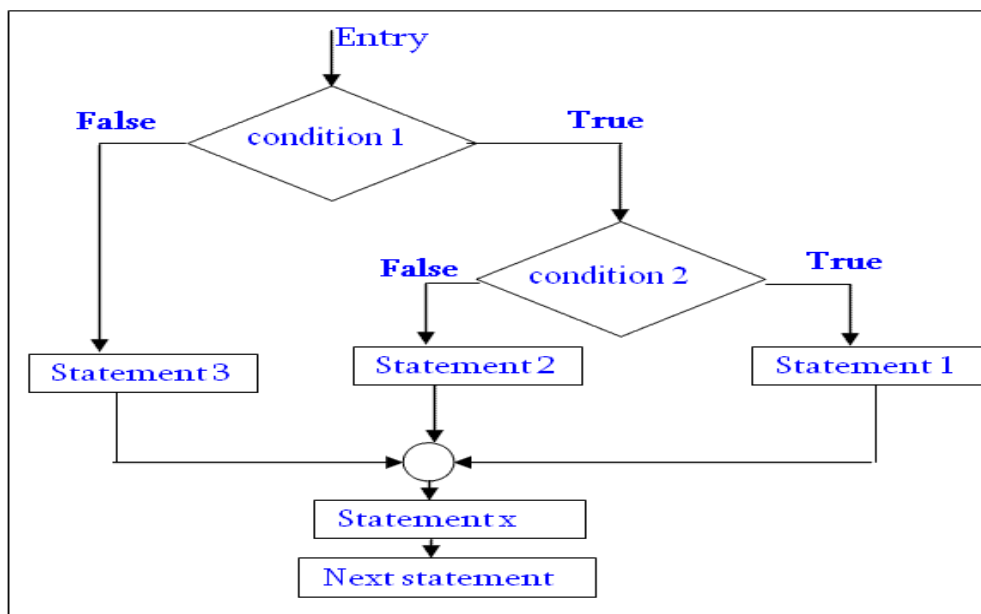
Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus



Where test condition-1 can be a logical or relational or arithmetic expression that results in TRUE or FALSE.

Working/Execution:

1. Evaluates the condition-1.
2. If the condition-1 is TRUE, evaluates the condition-2, if condition-2 is true then executes statement 1 otherwise executes statement 2. control is jumped to statement x
3. If the condition-1 is FALSE statement 3 is executed followed by statement-x execution.



Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Fig 2.3. Flow chart for nest if else statement

Example:

```

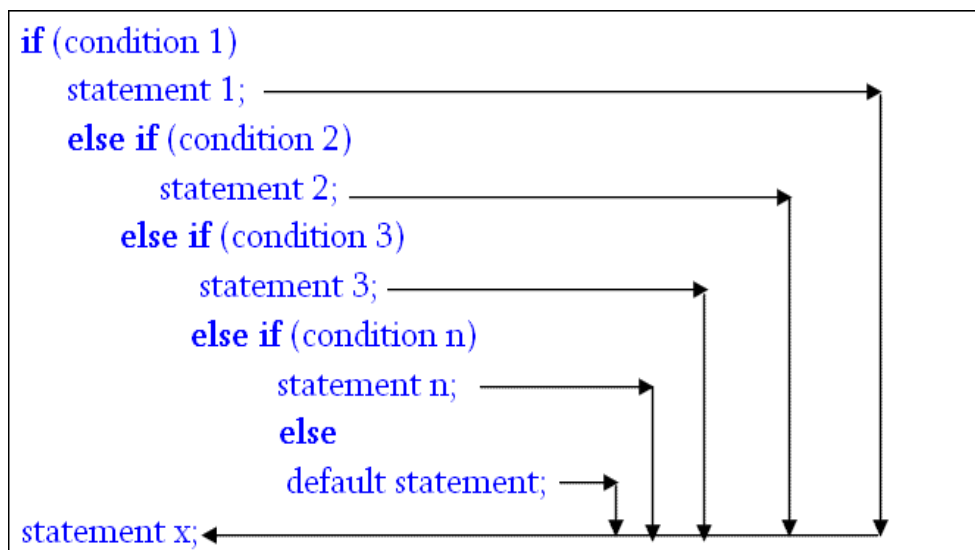
if(n>=0)
{
    if(n==0)
    {
        printf("The given number is zero \n")
    }
    else
    {
        printf("The given number is positive \n");
    }
}
else
    printf("The give n number is negative \n");

```

2.2.4 else if ladder

else if ladder can be used to choose an option from more than two alternatives. else if ladder is multiple branch decision statement.

General form/Syntax:



where test-expressions can be a logical or relational or arithmetic expressions which result in TRUE or FALSE.

Working/execution

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

1. The conditions are evaluated from the top to downwards, as soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement x skipping the rest of the ladder.
2. When all the n conditions become false then final else containing default statement will be executed.

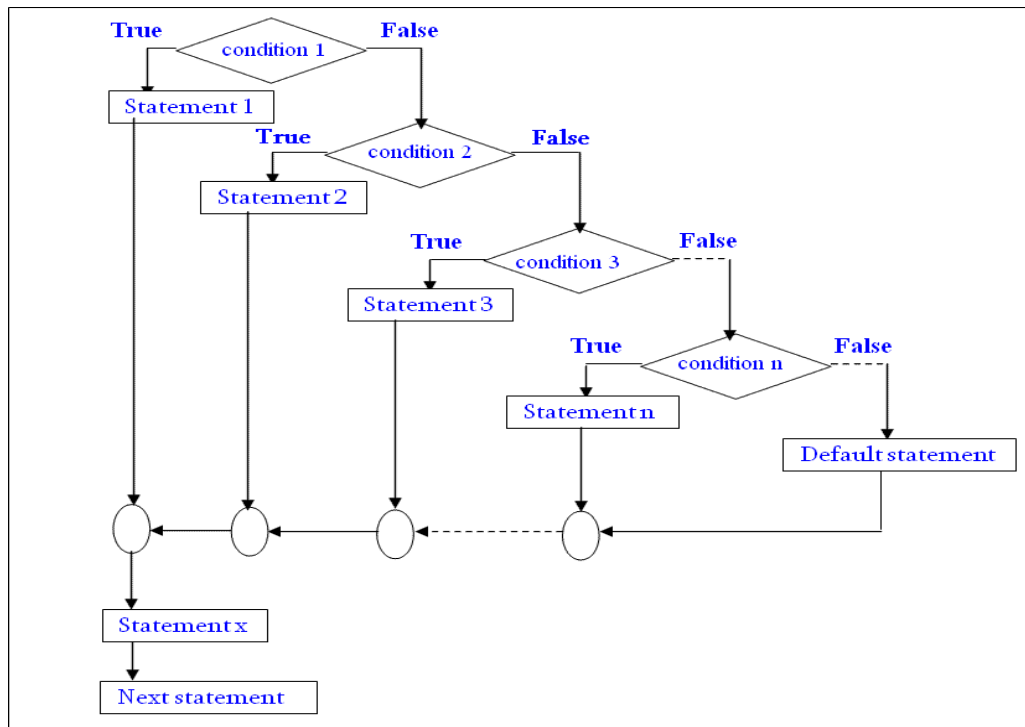


Fig 2.4. Flow chart for else-if ladder statement

Example: `if(a>b && a>c)`

```

    largest=a;
  else if(b>a && b>c)
    largest=b;
  else
    largest=c;

```

2.2.5 The switch Statement

Switch statement is multiple branch decision statement. The switch statement tests the value of a given variable(expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

General form/Syntax:

```

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

```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus


```

                                break;
                                ....
                                ....

                                default:
                                    default-block;
                                }
                                Statement-x;

```

where expression can be an integer constant or characters. Value-1,value-2 are constants or constant expressions and are known as case labels. switch statement is multiple branch decision statement.

Working:

1. Evaluates the expression.
2. The resulted integer value is compared with case constants.
3. If match is found , the statements associated with that particular case are executed.
4. If no match is found , statements under default are executed.
5. When break statement is encountered, the switch is terminated.

Note: Each case statement must have break statement.

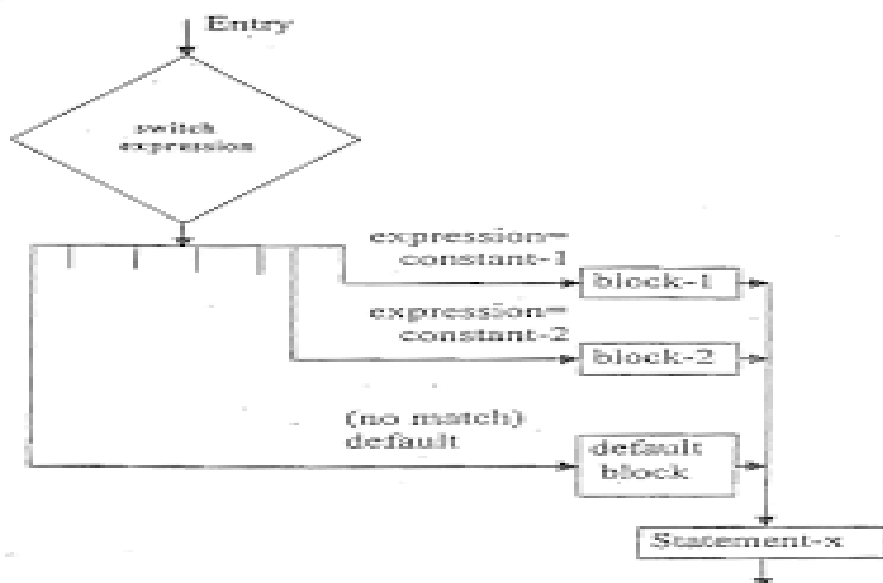


Fig 2.5 Flow chart for switch statement

Example for switch statement:

```

switch(letter)
{
    case 'a': printf("The given character is a vowel \n");
              break;
    case 'e': printf("The given character is a vowel \n");
              break;
    case 'i': printf("The given character is a vowel \n");
              break;
    case 'o': printf("The given character is a vowel \n");
              break;
}

```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```

        case 'u': printf("The given character is a vowel \n");
                   break;
        default: printf("The given character is not a vowel \n");
    }

```

```

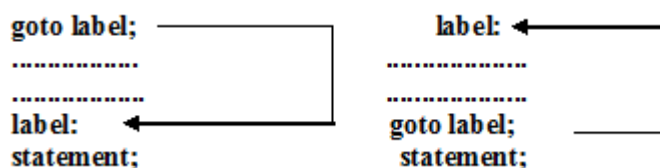
/*Write a C program to find whether given character is vowel or not*/
#include<stdio.h>
#include<conio.h>
void main()
{
    char letter;
    clrscr();
    printf("Enter any character \n");
    scanf("%c",&letter);
    switch(letter)
    {
        case 'a': printf("The given character is a vowel \n");
                   break;
        case 'e': printf("The given character is a vowel \n");
                   break;
        case 'i': printf("The given character is a vowel \n");
                   break;
        case 'o': printf("The given character is a vowel \n");
                   break;
        case 'u': printf("The given character is a vowel \n");
                   break;
        default: printf("The given character is not a vowel \n");
    }
    getch();
}

```

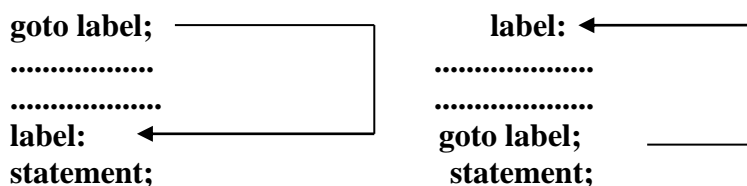
2.2.6 The goto Statement and Labels

- To branch unconditionally from one point to another in a program goto statement is used.
- goto requires a label to identify the place where the branch is to be made.
- Label is a valid variable name, and must be followed by a colon (:).
- Label is placed immediately before the statement where the control is to be transferred.

General format/syntax:



Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Forward jumpBackward jump

Label can be anywhere either before or after the goto label.
Note: goto breaks the normal sequential execution of the program.

Types of jumps

1. Backward jump: If the label is before the **goto** statement, a loop is formed and set of statements will be executed repeatedly.
2. Forward jump: If the label is placed after the **goto** statement, then some statements are skipped.

Note: There are two types of control transfer statements:

1. Unconditional transfer control statements: goto ,break.
2. Conditional transfer control statements: if,for,while,do...while.

2.3 Looping

Looping statements are used to execute a statement or group of statements repeatedly for a specified number of times or as long as the condition is true/ until the condition is false.

A **loop** is a particular area of a program where some executable statements are written which gets executed by testing one or more conditions. So, in looping, a sequence of statements is executed until some conditions for termination are satisfied.

A program loop therefore consists of two segments; the body of the loop and the control statement.

1. '**body of the loop**' consists of set of statements and the other
2. '**Control statement**' .The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure can be classified into two types; entry controlled and exit controlled.

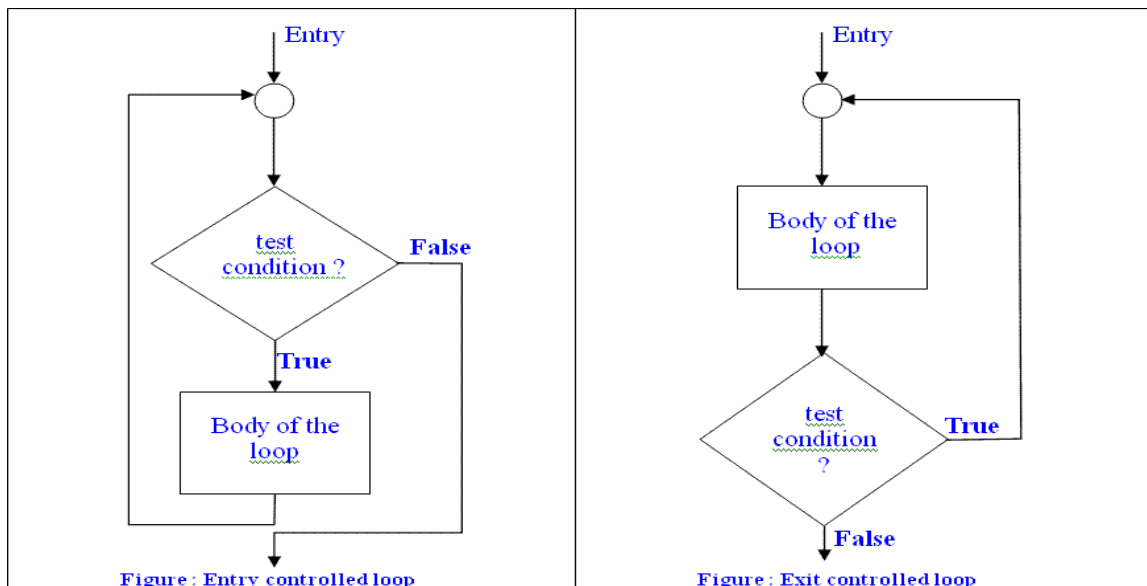
1. **Entry-controlled loops:**-In Entry controlled loop the test condition is checked first and if that condition is true than the block of statement in the loop body will be executed

Example: while loop and for loop.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

2. **Exit controlled loop:** -In exit controlled loop the body of loop will be executed first and at the end the test condition is checked, if condition is satisfied then body of loop will be executed again.

Example:do-whileloop.



C programming language provides three constructs for performing loop operations.

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

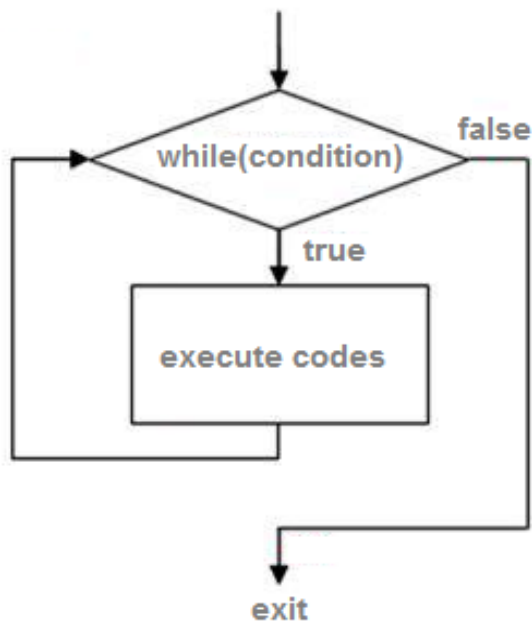
Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements for fixed number of time.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body. This ensures that the loop body is run at least once

2.3.1 While loop:

The while is an entry controlled looping statement, it makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. While is used to repeat a block of statements until condition is true. The general form of **while** statement is:

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

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



The **test-condition** is evaluated and if the condition is **true**, then the body of loop is executed. After execution of the body, the test-condition is once again evaluated and if it is **true**, the body is executed once again. This process is repeated, execution of the body continues until the test-condition finally becomes **false** and the control is transferred out of the loop. If the condition is initially false, the statement will not be executed.

Write a C program to print the natural numbers less than 10 using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i ; // declaration statement
    clrscr();
    i=0 ; // initialization statement
    while ( i<10) // loop statement
    {
        printf ( "%d\t" , i ) ;
        i++;    // increment statement
    }
    getch();
}
```

Output: 0 1 2 3 4 5 6 7 8 9

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

In the above example the first one is a declaration statement, which declares that **i** is an integer type variable. The second one is an initialization statement which initializes the loop variable **i** to zero. Next one is the while loop. The loop will be executed until the value of the loop variable **i** is less than or equal to 9. As soon as the value of **i** is 10 the outcome of the conditional statement of while loop will be false and the loop will be terminated. The execution of while loop for each execution can be tabulated as below –

Execution Number	Value of conditional statement	Result	printf("%d", -c)	value of c (c++)
1	0<10	True	0	
2	1<10	True	1	
3	2<10	True	2	
4	3<10	True	3	
5	4<10	True	4	
6	5<10	True	5	
7	6<10	True	6	
8	7<10	True	7	
9	8<10	True	8	
10	9<10	True	9	
11	10<10	False	will not execute	will not execute

Fig - Execution of while loop for example 3.5

2.3.2 Do-while Loop

In a do-while loop body of the loop is executed before the test is performed. At the end of the loop, test condition in a while statement is evaluated. If the test expression is **true**, the program continues to evaluate the body of the loop once again. This process continues as long as the test expression is true. When the test expression becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

General Syntax:

```
do
{
    Body of the loop;
}
while(test-condition);
```

Note: This is only Basic information for students. Please refer "Reference Books" prescribed as per syllabus

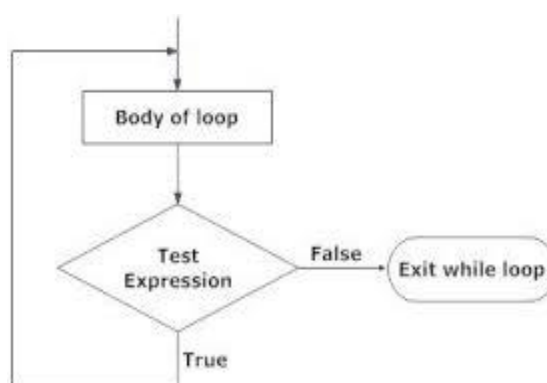


Figure: Flowchart of do...while loop

Note: The test-condition is evaluated at the bottom of the loop, the do...while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

Example:

```

do
{
    printf("Enter positive integer \n");
    scanf("%d", &num);
}
while(num<=0); /*loop repeats until positive number is entered*/
  
```

2.3.3 Compare while and do....while loop

Sl no	while loop	do...while loop
1	General structure: <pre> while (test condition) { body of the loop; } </pre>	General structure: <pre> do { body of the loop; } while(test-condition); </pre>
2	Entry condition or Entry-controlled looping construct	Exit condition or Exit-controlled looping construct
3	It is a pre-test loop	It is a post-test loop
4	Body of the loop will not be executed at all if the test-expression is evaluated to FALSE at the first time itself	Body of the loop will be executed at least once even if the test-expression is evaluated to FALSE for the first time itself
5	While is a counter-controlled loops.	Do .while is sentinel- controlled loops.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

2.3.4 for loop:

For loop is an entry-controlled looping statement. The general form of the **for** statement is:

```
for (initial value; test-condition; increment/ decrement statement )  
{  
    body of the loop  
}
```

for structure consist of three different statements separated by semicolon (;).

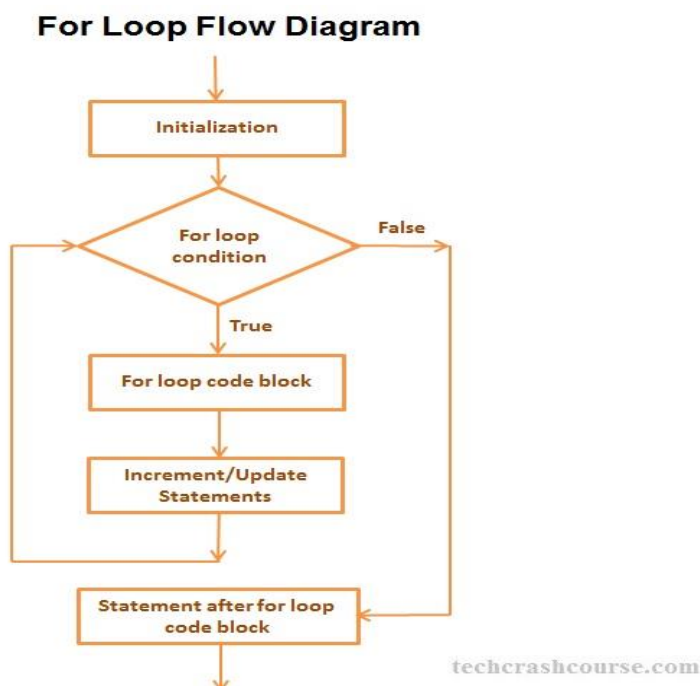
Step 1: first initial value is executed

Step 2: then condition is checked if it is true then control enters in the loop and body of the loop is executed.

Step 3: then increment or decrement of the variable is done.

Step 4: again it checks the condition if it is true , enters the loops, executes the statements

This loop continues unless condition is false.



Write a C program to print the numbers from 1 to 10 using for loop

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

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus


```
{
    int i;
    clrscr();
    for ( i = 1 ; i <=10 ; i++ )
        printf ( " %d\t ", i );
    getch();
}
```

OUTPUT:

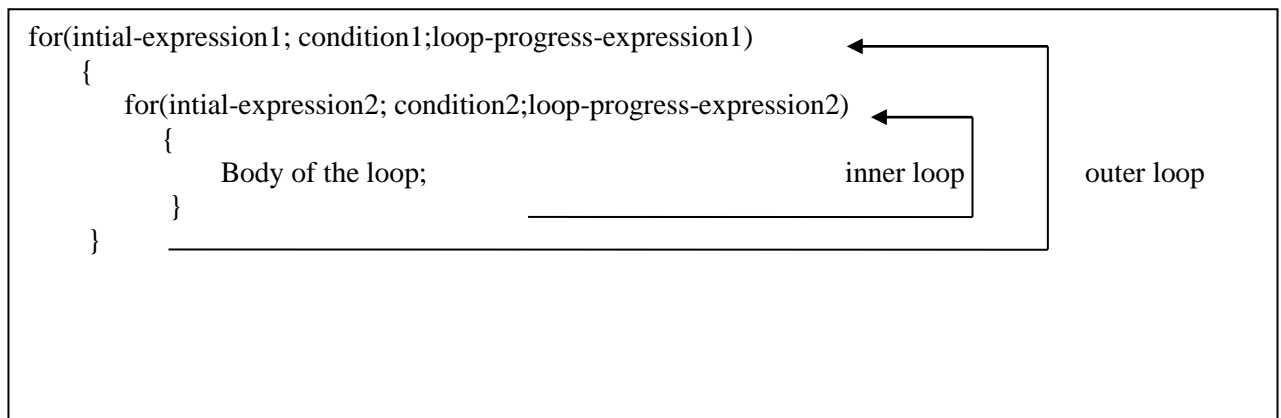
1 2 3 4 5 6 7 8 9 10

In the above example the first loop variable **i** is initialized to one, then check the condition, if the value of **i** is less than 10, condition becomes true and body of the loop is executed, the control will execute the printf statement which will print the value 1. Next the loop variable **i** is incremented by 1. Thus the current value of loop variable **i** is now 2. Again check the condition whether $2 < 10$. Since the condition is true, the printf statement will be executed and prints 2. Thus the process is repeated until the loop variable **i** becomes 11.

2.3.4.1 Nesting of for loops

If a for loop is placed in another for loop, then it is called a nested for loop.

General form:



Step 1: first initialization is done for expression1 and condition1 is checked

Step 2: if condition1 is true then control enters the inner for loop and expression2 is initialized, condition2 is checked.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Step 3: if condition2 is true, then body of the loop is executed. Loop progress-expression2 is either incremented/decremented and body of the loop is continue to execute until the loop end condition2 becomes false.

Step 4: When the condition2 is false, control jumps to the outer for loop to increment/decrement loop-progress- expression1. If loop-end condition1 is true then control enters the inner for loop to initialize expression2 once again

Step 5: if the condition1 is false the control comes out of the outer for loop only when loop-end-condition1 is false.

Example: for(i=0,count=0;i<10;++i)
 {
 for(k=1;k<10;++k)
 {
 printf("%d",++count);
 }
 }

Write a C program to generate right triangle

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

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

```
void main()  
{  
  int n, i, j;  
  clrscr();  
  printf("Enter number of rows\n");  
  scanf("%d",&n);  
  for ( i = 1 ; i <= n ; i++ )  
  {  
    for( j = 1 ; j <= i ; j++ )  
      printf(" %d",i);  
    printf("\n");  
  }  
  getch();  
}
```

OUTPUT

Enter number of rows

5

1

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

2 2

3 3 3

4 4 4 4

5 5 5 5 5

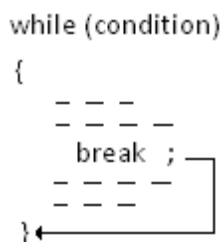
2.3.5 Jumps in Loops (Break and Continue)

Sometimes it is desirable to skip a part of the loop (continue) or to leave the loop (break) as soon as certain condition occurs.

Exiting from a loop can be achieved by using **break** statement.

- ✓ The break statement in a switch statement causes the control to terminate switch statement and the statement following switch statement will be executed.
- ✓ When break statement is used inside a loop (such as for/while/do-while), the control comes out of the loop and the statement following the loop will be executed. The statements after break statement are skipped.
- ✓ The break statement is used to terminate the loop when a specific condition is reached.
- ✓ If break appears in the inner loop of a nested loop, the control only comes out of the inner loop.

```
while (condition)
{
    _ _ _ _ _
    break ;
    _ _ _ _ _
}
```



Program to illustrate break statement

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1; i<=5 ; i++)
    {
        if(i==3)
            break;
        printf("%d\t",i);
    }
    getch();
}
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
}
```

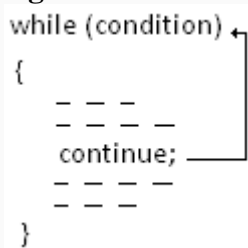
Output:

```
1      2
```

Continue Statement:

During execution of a loop, it may be necessary to skip part of the loop based on some condition.

The working structure of 'continue' is similar as that of the break statement but difference is that it cannot terminate the loop. It causes the loop to be continued with next iteration after skipping any statements in between. Continue statement tells the compiler, simply “skips statements and continue with the next iteration”.

Figure :***Program to illustrate continue statement***

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1; i<=5; i++)
    {
        if(i==3)
            continue;
        printf("%d\t",i); // 3 is omitted
    }
    getch();
}
```

Output:

```
1      2      4      5
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Break	Continue
1. When break is executed, the statements following break are skipped and causes the loop to be terminated.	1. When continue statement is executed, the statements following continue are skipped and causes the loop to be continued with the next iteration
2. Syntax is: break;	2. Syntax is: continue;
3. For example, <pre>for (i=1;i<=5;i++) { if(i==3)break; printf("%d\t",i); } </pre> Output 1 2	3. For example, <pre>for (i=1;i<=5;i++) { if(i==3) continue; printf("%d\t",i); } </pre> Output 1 2 4 5
<pre> graph TD Start(()) --> CC[conditional code] CC --> Cond{condition} Cond -- "If condition is true" --> CC Cond -- "If condition is false" --> Break{break} Break --> End((())) </pre>	<pre> graph TD Start(()) --> CC[conditional code] CC --> Cond{condition} Cond -- "If condition is true" --> Continue{continue} Continue --> CC Cond -- "If condition is false" --> End((())) </pre>

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT – 3 : Functions

3.1 Definition of Function:

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

Function naming conventions in c programming:

Rule 1. Name of function includes alphabets, digits and underscore.

Rule 2. First character of name of any function must be either alphabets or underscore.

Rule 3. Name of function cannot be any keyword of c program.

Rule 4. Name of function cannot be global identifier.

Rule 5. Name of function cannot be exactly same as of name of other function or identifier within the scope of the function.

Rule 6. Name of function is case sensitive.

Uses of c functions:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

3.2 Standard Library of C functions:

C Library functions are inbuilt functions in C programming.

The prototype and data definitions of the functions are present in their respective header files, and must be included in your program to access them.

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

```
#include <stdio.h>
int main()
{
    // If you use printf() function without including the <stdio.h>
    // header file, this program will show an error.
    printf("Catch me if you can.");
}
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

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

List of the most common libraries and a brief description of the most useful functions they contain follows:

1. **stdio.h: I/O functions:**

- a. **getchar()** returns the next character typed on the keyboard.
- b. **putchar()** outputs a single character to the screen.
- c. **printf()** as previously described.
- d. **scanf()** as previously described.

2. **string.h: String functions**

- a. **strcat()** concatenates a copy of `str2` to `str1`.
- b. **strcmp()** compares two strings.
- c. **strcpy()** copies contents of `str2` to `str1`.

3. **ctype.h: Character functions**

- a. **isdigit()** returns non-0 if `arg` is digit 0 to 9.
- b. **isalpha()** returns non-0 if `arg` is a letter of the alphabet.
- c. **isalnum()** returns non-0 if `arg` is a letter or digit.
- d. **islower()** returns non-0 if `arg` is lowercase letter.
- e. **isupper()** returns non-0 if `arg` is uppercase letter.

4. **math.h: Mathematics functions**

- a. **acos()** returns arc cosine of `arg`.
- b. **asin()** returns arc sine of `arg`.
- c. **atan()** returns arc tangent of `arg`.
- d. **cos()** returns cosine of `arg`.
- e. **exp()** returns natural logarithm.
- f. **fabs()** returns absolute value of `num`.
- g. **sqrt()** returns square root of `num`.

5. **time.h: Time and Date functions**

- a. **time()** returns current calendar time of system.
- b. **difftime()** returns difference in secs between two times.
- c. **clock()** returns number of system clock cycles since program execution.

6. **stdlib.h: Miscellaneous functions**

- a. **malloc()** provides dynamic memory allocation, covered in future sections.

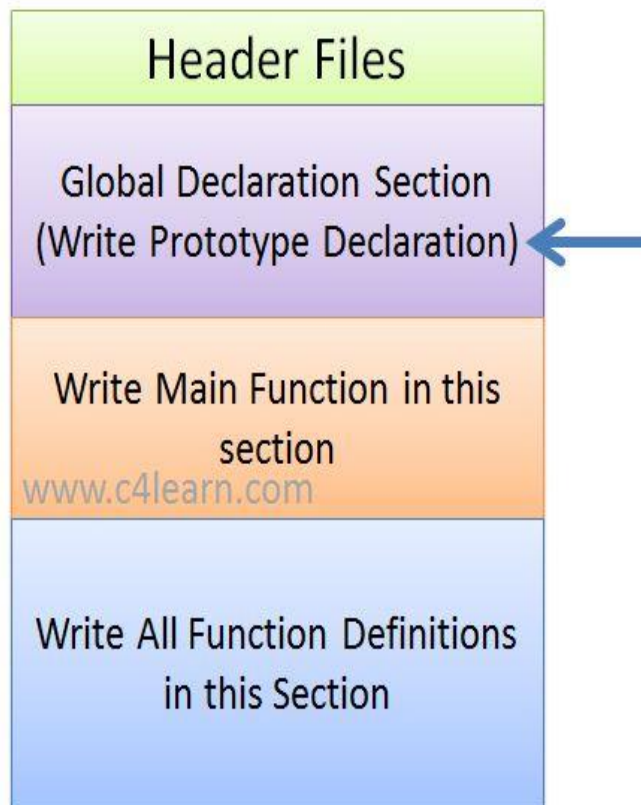
Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

- b. **rand()** as already described previously.
- c. **srand()** used to set the starting point for rand().

3.3 Function prototype:

A function prototype declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual declaration.

Pictorial representation



Syntax:

```
return_type function_name ( type arg1, type arg2..... );
```

prototype declaration comprised of three parts i.e name of the function, return type and parameter list.

[Examples of prototype declaration:](#)

Function with two integer arguments and integer as return type is represented using below syntax.

```
int sum(int,int);
```

Function with integer argument and integer as return type is represented using below syntax.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus


```
int square(int);
```

In the below example we have written function with no argument and no return type.

```
void display(void);
```

In below example we have declared function with no argument and integer as return type.

```
int getValue(void);
```

Important Points :

1. Our program starts from main function. Each and every function is called directly or indirectly through main function.
2. Like variable we also need to declare function before using it in program.
3. In C, declaration of function is called as prototype declaration.
4. Function declaration is also called as function prototype.

Below are some of the important notable things related to prototype declaration:

1. It tells name of function, return type of function and argument list related information to the compiler.
2. Prototype declaration always ends with semicolon.
3. Parameter list is optional.
4. Default return type is integer.

Positioning function declaration:

1. If function definition is written after main then only we write prototype declaration in global declaration section.
2. If function definition is written above the main function then ,no need to write prototype declaration.

Case 1: Function definition written before main.

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

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
void displayMessage() {  
    printf("www.c4learn.com");  
}  
  
void main() {  
    displayMessage();  
}
```

Case 2: Function definition written after main.

```
#include<stdio.h>  
  
//Prototype Declaration  
void displayMessage();  
  
void main() {  
    displayMessage();  
}  
  
void displayMessage() {  
    printf("www.c4learn.com");  
}
```

If we write prototype declaration then:

1. Prototype declaration tells compiler that we are going to define this function somewhere in the program.
2. Compiler will have prior information about function.
3. As compiler have prior information ,during function calling compiler looks forward in the program for the function definition.

If we don't write prototype declaration then:

1. Compiler don't have any reference of Function.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

2. Compiler don't have prior information of that function.
3. Compiler gets confused and interpret it as unknown reference and throws error.

3.4 Formal parameter list:

Parameters are values that are passed into a [function](#) .

Note :

1. Parameter Means **Values Supplied to Function so that Function can Utilize These Values.**
2. Parameters are Simply Variables.
3. Difference between Normal Variable and Parameter is that **“These Arguments are Defined at the time of Calling Function”.**
4. Syntactically We can pass **any number of parameter to function.**
5. Parameters are Specified **Within Pair of Parenthesis .**
6. These Parameters are **Separated by Comma (,)**

Example: Display(a,b,c,d,e);

- **Parameter** : The names given in the function definition are called Parameters.
- **Argument** : The values supplied in the function call are called Arguments.

There are two categories:

- *Actual parameters* are parameters as they appear in function calls.
- *Formal parameters* are parameters as they appear in function declarations.

In C Programming Function Passing Parameter is Optional. We can Call Function Without Passing Parameter.

1. Function Call Without Passing Parameter :

```
add(a,b);
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

1. Here Function add() is Called and 2 Parameters are Passed to Function.
2. a,b are two Parameters.

2. Function Call Without Passing Parameter :

Display();

- Parameter Written In Function Definition is Called “Formal Parameter”.

```
Void main()
{
    Int num1;
    Display(num1);
}

Void display(int para1)
{
    -----
}
```

- Para1 is “**Formal Parameter**”

Actual Parameter :

- Parameter Written In Function Call is Called “Actual Parameter”.

```
Void main()
{
    Int num1;
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
Display(num1);  
  
}  
  
Void display(int para1)  
  
{  
  
-----  
  
}
```

- num1 is “Actual Parameter”

3.5 Return Type:

The **return type** defines the data type of the value returned from a subroutine or function, the return type must be explicitly specified when declaring a function.

In the example:

```
int sum(int, int);
```

the return type is int.

Various mechanisms are used for the case where a subroutine does not return any value, e.g., a return type of void is used.

```
void display()
```

Function accepts argument and it return a value back to the calling Program thus it can be termed as Two-way Communication between calling function and called function.

Example programme:

```
#include<stdio.h>  
float calculate_area(int);  
void main()  
{
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
int radius;
float area;
printf("\nEnter the radius of the circle : ");
scanf("%d",&radius);
area = calculate_area(radius);
printf("\nArea of Circle : %f ",area);
}
float calculate_area(int radius)
{
    float areaOfCircle;
    areaOfCircle = 3.14 * radius * radius;
    return(areaOfCircle);
}
```

Output :

```
Enter the radius of the circle : 2
Area of Circle : 12.56
```

Explanation :

1. In the above program we can see that inside main function we are calling a user defined calculate_area() function.
2. We are passing integer argument to the function which after area calculation returns floating point area value.

3.6 Function call:

Definition for **function call** - a call that passes control to a subroutine, after the subroutine is executed control returns to the next instruction in main program.

Syntax: Calling Function in C

```
function_name(Parameter1 ,Parameter2 ,.....Parameter n);
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Calling a Function:

1. Call a C function just by **writing function name with opening and closing round brackets followed with semicolon.**
2. If we have to supply parameters then we can write parameters inside pair of round brackets.
3. Parameters are optional.

Call Function without Passing Parameter :

```
display();
```

Passing 1 Parameter to function :

```
display(num);
```

Passing 2 Parameters to function :

```
display(num1,num2);
```

Calling Function in C: Sample Code

```
#include<stdio.h>
#include<conio.h>
int sum(int,int);
void main()
{
    int a,b,c;
    printf("\nEnter the two numbers : ");
    scanf("%d%d",&a,&b);
    c = sum(a,b);           //calling function
    printf("\nAddition of two number is : %d",c);
    getch();
}

int sum (int num1,int num2)
{
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
int num3;  
num3 = num1 + num2 ;  
return(num3);  
}
```

Output :

Enter the two numbers : 12 12
Addition of two number is : 24

Analysis of Code : Calling Function

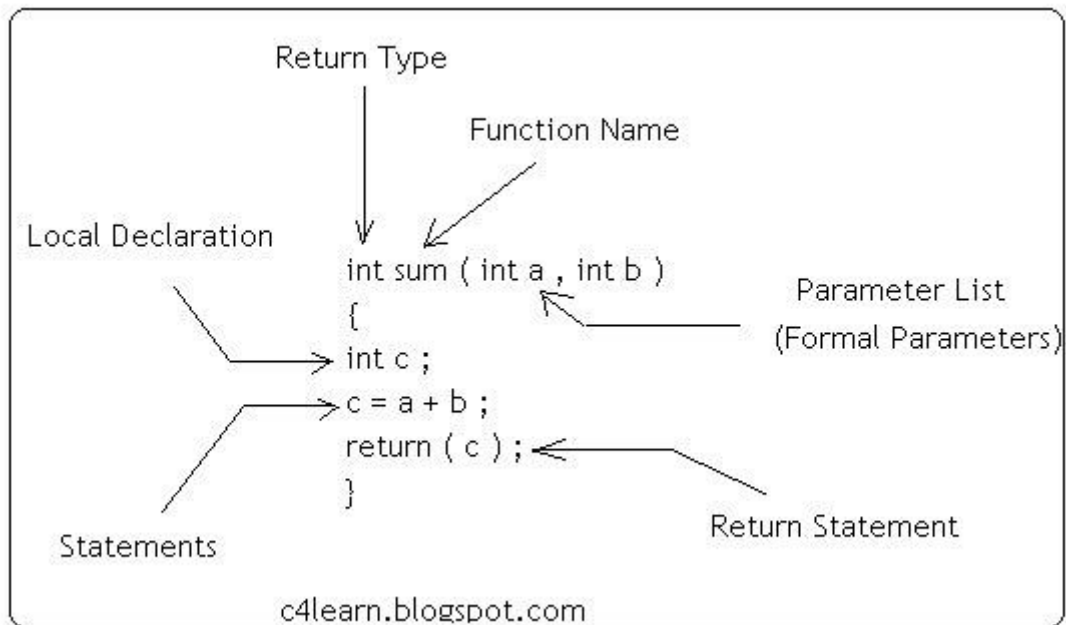
1. In the above example **sum(a,b);** is function call .
2. a,b are **parameters passed to function 'sum'**
3. Function call should be made by **ending Semicolon.**

3.7 Block structure:

```
return-type function-name(parameters)  
{  
  declarations  
  statements  
  return value;  
}
```

Example:

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus



Different Sub parts of Above Syntax:

return-type:

1. Return Type is Type of value returned by function.
2. Return Type may be “Void” if function is not going to return a value.
3. If return type for a function is not specified, it defaults to int.

function-name:

1. It is Unique Name that identifies function.
2. All Variable naming conversions are applicable for declaring valid function name.

Parameters:

1. Comma-separated list of types and names of parameters.
2. Parameter injects external values into function on which function is going to operate.
3. Parameter field is optional.
4. If no parameter is passed then no need to write this field.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Value:

1. It is value returned by function upon termination.
2. Function will not return a value if return-type is void.

Note : The return statement forces the function to return immediately though function is not yet completed

3.8 Passing arguments to a Function: call by value.

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called **argument**.

Two Ways of Passing Argument to Function in C Language :

1. Call by Value.
2. Call by Reference.

Call by Value :

- The default parameter passing mechanism is Call by value.
- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Example program1:

```
#include<stdio.h>
void interchange(int number1,int number2)
{
    int temp;
    temp = number1;
    number1 = number2;
    number2 = temp;
}
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
int main()
{
    int num1=50,num2=70;
    interchange(num1,num2);
    printf("\nNumber 1 : %d",num1);
    printf("\nNumber 2 : %d",num2);

    return(0);
}
```

Output :

```
Number 1 : 50
Number 2 : 70
```

Explanation : Call by Value

1. While Passing Parameters using call by value , **xerox copy of original parameter is created** and passed to the called function.
2. Any update made inside method will not affect the **original value of variable in calling function**.
3. In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.
4. As their scope is limited to only function so they **cannot alter the values inside main function**.

Example Programme2:

```
#include<stdio.h>
int sum(int n1,int n2)
{
    return(n1+n2);
}
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
int main()
{
int num1=11,num2=22;
int result;
result = sum(num1,num2);
printf("\nResult : %d",result);
return(0);
}
```

Explanation of Example :

Function Name	sum
Return Type	Integer
Calling Function	main
Parameter Passing Method	Pass by Value
No of Parameters Passed to Function	2
Actual Parameter 1	num1
Actual Parameter 2	num2
Formal Parameter 1	n1
Formal Parameter 2	n2
Function Will Return	n1+n2
Returned Value will be Catch in	result

- Refer this url link for functions:

<https://youtu.be/x8APkeKR6r0>

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT - 4 : Arrays

Introduction

We have been using variable names in C, that allow us to store a single data entity at any given time, thus representing one memory location of appropriate size in bytes. such variables are called scalar variables or unsubscripted variables.

But a situation may arise, where in a group of similar data items have to be processed collectively ! In such a case array , a derived data type comes to your help. An array is used to store and access a list of data values under a single variable name called array variable or subscripted variable.

4.1 What is an Array?

An array can be defined as “a homogeneous collection of data elements”.

OR

An array can be defined as “a set of ordered data values or group of elements that have a common variable name and all elements of an array belong to same datatype which are identified separately from one another by a subscript or an index”.

Advantages and Disadvantages of arrays.

Advantages:

1. Single variable name can be used for similar data elements.
2. Since array elements are stored in consecutive memory locations, it is easy to access and process array elements.
3. Two dimensional arrays are used to represent matrices.
4. Arrays can be used to implement stacks, queues etc.
5. Simple and easy to create

Disadvantages:

1. It is difficult to operate with multi dimensional array.
2. Memory size of an array cannot be changed during run time.
3. Inserting data in array is difficult.
4. Deleting data in array is difficult.
5. The array size should be known in prior to reserve memory locations.

4.2 Declaring an Array:

An Array definition specify the array variable name, its datatype for array elements and the size of that array. The size indicate maximum number of data elements array can contain.

The general syntax of an array declaration is as follows:

datatype arrayname[size];

Example

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Consider the following declaration:

```
int marks [ 5 ];
```

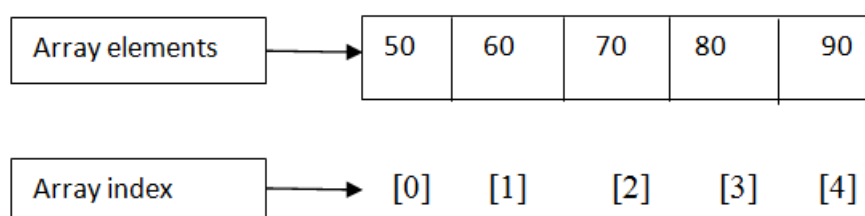
- Each int datatype value is 2 bytes.

Note: By default array name holds the address of the first element of an array.

4.3 Initializing an Array

You cannot assign a whole array to another array. Instead individual array elements can be assigned. Their value may be changed and / or initialized to new values.

For Example Consider an array that is initialized to five integer values,
`int marks[5]={ 50,60,70,80,90};`



3.4 One dimensional Arrays :

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

One dimensional Arrays can be declared as follows:

```
Data_type array_name[size];
```

Example: `int marks[5];`

One dimensional Arrays can be initialized to values during declaration.

General form:

```
Data type array_name[size] = {initial values};
```

The initial values are included in pair of braces and separated by comma(,).

Example: `int marks[5]={ 10,20,25,16,5};`

It Assigns values as given below.

```
marks[0]=10 =>1st element  
marks[1]=20 =>2nd element  
marks[2]=25 =>3rd element  
marks[3]=16 =>4th element  
marks[4]=5  =>5th element
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

If all the elements are initialized, then no need to specify the size of array. Thus above example can also be written as:

```
int marks[ ]={10,20,25,16,5};
```

It is not compulsory to initialise all the elements of array.

i.e. `int marks[5]={10,20};` /*initialises only first 2 elements out of 5 and the remaining elements are initialised to 0 by default*/

Example program:

/* Write a C program to accept 10 numbers ,store them in an array and display the array elements */

```
#include<stdio.h>
Main( )
{
    int i,num[10];
    printf("enter 10 elements to the array num ");
    For( i=0 ;i< 10; i+ +)
        Scanf("%d",&num[i]);
    Printf("Array elements are as follows ");
    For (i=0 ;i<10; i+ +)
        Printf("num [%d]= %d",i ,num[i] );
```

OUTPUT

Enter 10 elements to the array num

10 20 30 40 50 60 70 80 90 11

Array elements are as follows

Num [0] =10

Num [1] = 20

Num [2] =30

Num [3] = 40

Num [4] = 50

Num [5] = 60

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Num [6] = 70

Num [7] = 80

Num [8] = 90

Num [9] = 11

4.4 Array manipulation

Array variables can be manipulated like usual variables. i.e

1. Values can be assigned to array locations.
Example: marks[3]=12;
2. Arithmetic operations can be performed on array elements
Example: marks[4]=marks[2]+4;
3. Array elements can be used in relational and logical expressions.
Example : if(marks[3]>10)
4. Array elements can be used in loops.
Example: for(i=0;i<10;i++)
marks[i]=IA[i]+att[i];
5. Array elements can be used in I/O functions for read/write operations.
printf(“%d”,num[i]);
6. Sorting and searching can be performed on array elements.

4.4 Finding the largest/smallest element in array.

/* C program to accept 10 numbers into an array and to find out biggest and smallest of them.*/

```
#include< stdio.h>
main( )
{
    Int num[10],i ,big,sma;

    printf(“ Enter 10 elements to array num” );

    for( i=0 ; i<10 ;I + +)

        scanf(“%d”,&num[i]);

    big=num[0];
    sma=num[0];

    for( i=1; i< 10; i++)
    {
        if( num[i] > big)
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
        big=num[i];  
        if (num[i]< sma )  
            sma=num[i];  
    }  
    Printf(" Largest =%d ",big);  
    Printf (" Smallest =%d",sma);  
    getch( );  
}
```

Output

Enter 10 elements to array num

10
20
30
40
50
60
70
80
90
100
Largest=100
Smallest=10

4.5 Searching & Sorting of element from an array.

4.5.1 Searching

Finding a particular element in an array is called searching.

Linear search: It is also called sequential searching. Here searching a particular element takes place element by element, one after another until desired element is found. If element is found, the search is successful, otherwise search fails.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus



```

/*Write a C program to search for given number in an array*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int num[100],snum, i , location=0;
    clrscr();
    printf("Enter ten integer values\n");
    for(i=0;i<10;++i)
        scanf("%d",&num[i]);
    printf("Enter number to be searched\n");
    scanf("%d",&snum);
    for(i=0;i<N;++i)
        if(snum==num[i])
        {
            location=i+1;
            break;
        }
    if(location==0)
        printf("The given number is not found");
    else
        printf("The given number is found at location %d",location);
    getch();
}

```

OUTPUT:

```

Enter 10 values
12
-25
32
11
45
25

```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

69
55
18
21
Enter number to be searched
25
The given number is found at location 6

4.5.2 Sorting

Arranging the array elements in an ascending order or descending order is called sorting. Bubble sorting algorithm is most widely used for sorting.

Refer video for bubble sort
<https://youtu.be/lyZQPjUT5B4>

```
/*Write a C program to sort 10 numbers in ascending order using bubble sort algorithm.*/  
#include<stdio.h>  
void main( )  
{  
    int num[10];  
    int i,k,exchange;  
    int last=9,temp;  
    printf("Enter any 10 integer numbers\n");  
    for (i=0;i<10;i++)  
        scanf("%d",&num[i]);  
    for(i=0;i<9;i++)  
    {  
        exchange=0;  
        for(k=0;k<last;k++)  
            if(num[i]>num[i+1])  
            {  
                temp=num[i];  
                num[i]=num[i+1];  
                num[i+1]=temp;  
                exchange=exchange+1;  
            }  
        if(exchange==0)  
            break;  
        else  
            last--;  
    }  
    printf("Numbers in ascending order are: \n");  
    for(i=0;i<10;i++)  
        printf("%d\n",num[i]);  
    getch();  
}
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Output

Enter any 10 integer numbers

22

66

99

01

55

44

88

77

33

11

Numbers in ascending order are:

01

11

22

33

44

55

66

77

88

99

4.8 Declaring & Initialization of Two dimensional arrays

It is an array in which each element is referred by two subscripts. First subscript specifies row position and second subscript specifies column position.

General form: **data-type array-name[row-size][column size];**

Example : `int matrix[3][3];`

4.8.1 DECLARATION OF TWO-DIMENSIONAL ARRAY.

The arrays should be declared before they are used in the program.

The general form of two-dimensional array declaration is

Data-type array-name[row-size][column-size];

Where data-type can be int, float, char or double.

array-name is the name of array variable.

row-size and column-size denotes number elements in row and column respectively.

As shown above array declaration specifies array variable name, data type of array elements and number of rows and columns.

Example 1:

```
int a[2][2];
```

It declares a two-dimensional array containing 2 rows and 2 columns.

Example 2:

```
float amount[3][2];
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

It declares a two-dimensional array containing 3 rows and 2 columns.

4.8.2 Initialization of two-dimensional array.

Two-dimensional array variables can be initialized to values during declaration as shown below.

General form:

Data type	array_name	[row-size][column-size]	= {list of initial
------------------	-------------------	--------------------------------	---------------------------

The initial values are written row by row and included in pair of braces and separated by comma(,).

Example: `int matrix[2][2]={ 1,2,3,4};`

The above example can also be written as:

`int matrix[2][2]={ { 1,2} ,{3,4}};`

Assigns values as given below.

`marks[0][0]=1 =>row1, column1`

`marks[0][1]=2 =>row1,column2`

`marks[1][0]=3 =>row2,column1`

`marks[1][1]=4 =>row2,column2`

If all the elements are initialized, then no need to specify the size of array. Thus above example can also be written as:

`int matrix[][]={ { 1,2} ,{3,4}};`

It is not compulsory to initialise all the elements of array.

i.e. `int matrix[2][2]={ 1,2};/*initialises only first row*/.`

	Column 0	Column 1	Column 2
Row 0	<code>arr[0][0]</code>	<code>arr[0][1]</code>	<code>arr[0][2]</code>
Row 1	<code>arr[1][0]</code>	<code>arr[1][1]</code>	<code>arr[1][2]</code>
Row 2	<code>arr[2][0]</code>	<code>arr[2][1]</code>	<code>arr[2][2]</code>

→ Column Index

→ Row Index

→ Array Name

4.9 Addition/Multiplication of two matrices.

4.9.1 Addition of two matrices

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

$$A = \begin{pmatrix} 5 & 10 & 20 \\ 8 & 6 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 8 & 5 \\ 2 & 9 & 3 \end{pmatrix}$$

Addition of two matrixes:

$$A + B = \begin{pmatrix} 5+3 & 10+8 & 20+5 \\ 8+2 & 6+9 & 5+3 \end{pmatrix} = \begin{pmatrix} 8 & 18 & 25 \\ 10 & 15 & 8 \end{pmatrix}$$

/*C program to add two matrices*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int m1[3][3],m2[3][3],sum[3][3],i,k;
clrscr();
printf("Enter 9 elements of first matrix\n");
for(i=0;i<3;++i)
for(k=0;k<3;++k)
scanf("%d",&m1[i][k]);
printf("Enter 9 elements of second matrix\n");
for(i=0;i<3;++i)
for(k=0;k<3;++k)
scanf("%d",&m2[i][k]);
for(i=0;i<3;++i)
for(k=0;k<3;++k)
sum[i][k]=m1[i][k]+m2[i][k];
printf("The sum of two matrices is\n");
for(i=0;i<3;++i)
{
for(k=0;k<3;++k)
printf("\t%d",sum[i][k]);
printf("\n");
}
getch();
}
```

OUTPUT:

Enter 9 elements of first matrix

```
1    2    3
4    5    6
7    8    9
```

Enter 9 elements of second matrix

```
9    8    7
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

6 5 4
3 2 1

The sum of two matrices is

10 10 10
10 10 10
10 10 10

4.9.2 Multiplication of two matrices

The screenshot shows a web-based matrix multiplication tool. It has two input sections for 'Matrix 1' and 'Matrix 2', both set to '3x3'. Matrix 1 contains the values 6, 2, 3 in the first row; 4, 5, 6 in the second row; and 7, 8, 9 in the third row. Matrix 2 contains the values 1, 2, 3 in the first row; 4, 5, 6 in the second row; and 7, 8, 9 in the third row. Below the matrices, the calculation process is shown: the first row of the result is calculated as (6*1)+(2*4)+(3*7)=35, (6*2)+(2*5)+(3*8)=46, and (6*3)+(2*6)+(3*9)=57. The second row is (4*1)+(5*4)+(6*7)=66, (4*2)+(5*5)+(6*8)=81, and (4*3)+(5*6)+(6*9)=96. The third row is (7*1)+(8*4)+(9*7)=102, (7*2)+(8*5)+(9*8)=126, and (7*3)+(8*6)+(9*9)=150. The final result matrix is displayed as a 3x3 grid with these values.

Matrix 1	x	Matrix 2																		
Matrix Type <input type="text" value="3x3"/>		Matrix Type <input type="text" value="3x3"/>																		
<table border="1"> <tr><td>6</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td></tr> </table>	6	2	3	4	5	6	7	8	9	x	<table border="1"> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td></tr> </table>	1	2	3	4	5	6	7	8	9
6	2	3																		
4	5	6																		
7	8	9																		
1	2	3																		
4	5	6																		
7	8	9																		
<table border="1"> <tr> <td>$(6*1)+(2*4)+(3*7)$</td> <td>$(6*2)+(2*5)+(3*8)$</td> <td>$(6*3)+(2*6)+(3*9)$</td> </tr> <tr> <td>$(4*1)+(5*4)+(6*7)$</td> <td>$(4*2)+(5*5)+(6*8)$</td> <td>$(4*3)+(5*6)+(6*9)$</td> </tr> <tr> <td>$(7*1)+(8*4)+(9*7)$</td> <td>$(7*2)+(8*5)+(9*8)$</td> <td>$(7*3)+(8*6)+(9*9)$</td> </tr> </table>			$(6*1)+(2*4)+(3*7)$	$(6*2)+(2*5)+(3*8)$	$(6*3)+(2*6)+(3*9)$	$(4*1)+(5*4)+(6*7)$	$(4*2)+(5*5)+(6*8)$	$(4*3)+(5*6)+(6*9)$	$(7*1)+(8*4)+(9*7)$	$(7*2)+(8*5)+(9*8)$	$(7*3)+(8*6)+(9*9)$									
$(6*1)+(2*4)+(3*7)$	$(6*2)+(2*5)+(3*8)$	$(6*3)+(2*6)+(3*9)$																		
$(4*1)+(5*4)+(6*7)$	$(4*2)+(5*5)+(6*8)$	$(4*3)+(5*6)+(6*9)$																		
$(7*1)+(8*4)+(9*7)$	$(7*2)+(8*5)+(9*8)$	$(7*3)+(8*6)+(9*9)$																		
<table border="1"> <tr><td>35</td><td>46</td><td>57</td></tr> <tr><td>66</td><td>81</td><td>96</td></tr> <tr><td>102</td><td>126</td><td>150</td></tr> </table>			35	46	57	66	81	96	102	126	150									
35	46	57																		
66	81	96																		
102	126	150																		

/* C program to multiply two 3x3 matrices*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int M1[3][3],M2[3][3],prod[3][3],i,k,m;
clrscr();
printf("Enter 9 elements of first matrix\n");
for(i=0;i<3;++i)
for(k=0;k<3;++k)
scanf("%d",&M1[i][k]);
printf("Enter 9 elements of second matrix\n");
for(i=0;i<3;++i)
for(k=0;k<3;++k)
scanf("%d",&M2[i][k]);
for(i=0;i<3;++i)
for(k=0;k<3;++k)
{
prod[i][k]=0;
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus


```

for(m=0;m<3;++m)
prod[i][k]=prod[i][k]+M1[i][m]*M2[m][k];
}
printf("The product of two matrices is\n");
for(i=0;i<3;++i)
{
for(k=0;k<3;++k)
printf("\t%d",prod[i][k]);
printf("\n");
}
getch();
}

```

output

Enter 9 elements of 1st matrix

```

1    2    3
4    5    6
7    8    9

```

Enter 9 elements of 2nd matrix

```

9    8    7
6    5    4
3    2    1

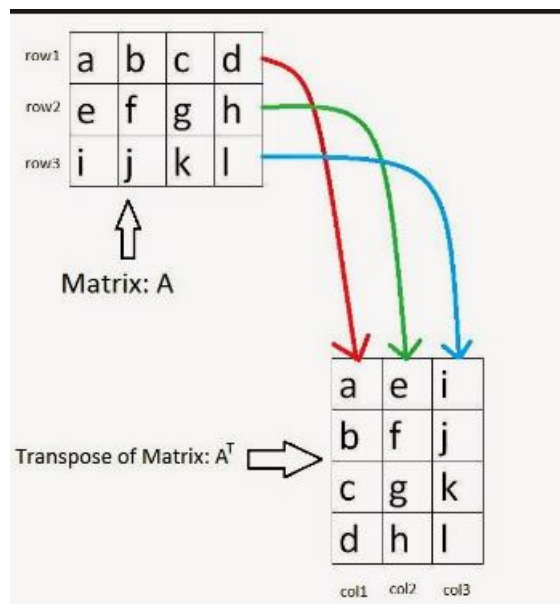
```

The product of 2 matrices is

```

18   30   36
72   75   72
126  120  108

```

Transpose of a square matrix:

/* C program to find transpose of a given matrix*/

#include<stdio.h>

#include<conio.h>

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
void main()
{
int A[3][3],tra[3][3],i,k;
clrscr();
printf("Enter 9 elements of matrix\n");
for(i=0;i<3;++i)
for(k=0;k<3;++k)
scanf("%d",&A[i][k]);
for(i=0;i<3;++i)
for(k=0;k<3;++k)
tra[i][k]=A[k][i];
printf("Transpose of given matrix is\n");
for(i=0;i<3;++i)
{
for(k=0;k<3;++k)
printf("\t%d",tra[i][k]);
printf("\n");
}
getch();
}
```

output

```
Enter 9 elements of matrix
1      -2      3
4       5     -6
7      -9      8
Transpose of given matrix is
      1      4      7
     -2      5     -9
      3     -6      8
```

4.11 Null terminated strings as array of characters.

A **null-terminated string** is a character string stored as an array containing the characters

and terminated with a null character (`'\0'`, called NUL in ASCII).

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' }; //valid character array initialization
```

```
char name[10]={ 'L', 'e', 's', 's', 'o', 'n', 's', '\0' }; //valid initialization
```

```
char name[13]="StudyTonight"; //valid character array initialization
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Remember that when you initialize a character array by listings all its characters separately then you must supply the `'\0'` character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3]="hell"; //Illegal
```

```
char str[4];
```

```
str="hell"; //Illegal
```

To initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The above declares an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character `'\0'` at the end.

Therefore, the array of char elements called myword can be initialized with a null-terminated sequence of characters by either one of these two statements:

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character (`'\0'`), which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically

The string "hello world" contains 12 characters including `'\0'` character which is automatically added by the compiler at the end of the string.

4.12 Arrays as function arguments.

Array can be passed as an argument to a function.

Consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows,

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

```
/* function declaration */
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
double getAverage(int arr[], int size);

void main ()
{
    /* an int array with 5 elements */
    int balance[5] = { 1000, 2, 3, 17, 50 };
    double avg;

    /* array is passed as a function argument */
    avg = getAverage( balance, 5 );

    /* output the returned value */
    printf( "Average value is: %f ", avg );
}
```

```
double getAverage(int arr[], int size) {

    int i;
    double avg;
    double sum = 0;
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = sum / size;
    return avg;
}
```

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

```
Average value is: 214.400000
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT – 5 : STRINGS

5.1 Definition of string

A collection of one or more characters enclosed within a pair of double quotes is called a string.

Example:

1. "shoot at sight"
2. "1st year EC"
3. "12\$"
4. "b"

As shown in examples, string can have combination of alphabets, numbers and special characters.

String are terminated by null terminator (`\0`). Null terminator denotes end of the string.

5.2 Declaring and Initialing String Variables

5.2.1 Declaration

C does not support strings as a data type. It allows us to represent strings as character arrays.

The general form of declaration of a string variable is

char string_name [string-size];

The size determines the number of characters in the variable string_name.

Example: **char city[10];**

When the compiler assigns character string to a character array, it automatically supplies a null character (`'\0'`) at the end of the string. Therefore the size should be equal to the maximum number of characters in the string plus 1.

5.2.2 Initialization

Like numeric arrays, character arrays can be initialized when they are declared.

1. `Char city[9]="NEW YORK";`

The string "NEW YORK" contains 8 characters (including space) and one memory location is provided for the null character.

Example:

1. `char name[15]="karnataka";`

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

2. `char result[5]="pass"`
3. `char result[5]={ 'p','a','s','s','\0' };`

In Examples 1 and 2 C compiler automatically inserts null terminator at the end. In example 3 specifying null character is compulsory

It is not compulsory to specify the size of the string if all the elements are initialised.

i.e. `char result[]="pass";`

5.3 Reading and writing strings from variables

5.3.1 Reading

Strings can be read into string variable by using built in functions such as:

1. `scanf()`
2. `getchar()`
3. `gets()` and
4. `strcpy()`.

5.3.1.1 scanf() function reads character string upto the appearance of blank space .

Example: `char city[50];`
`scanf("%s",city);`

if the following line of text is typed in at the terminal,

NEW YORK

Then only the string "NEW" will be read into the array **city**, since the blank space after the word "NEW" will terminate the reading of a string.

The `scanf` function automatically terminates the string that is read with a null character. In case of character arrays `%s` is used to read a character string and ampersand (&) is not required before the variable name.

We can also specify the field width using the form `%ws` in the `scanf` statement for reading a specified number of characters from the input string. Example:

`scanf("%ws",name);`

1. the width **w** is equal or greater than the number of characters typed in. The entire string will be stored in the string variable.
2. The width **w** is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider the following statements

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
char name[10];
```

```
scanf("%5s",name);
```

the input string RAM will be stored as

R	A	M	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

The input KRISHNA will be stored as

K	R	I	S	H	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading line of Text

scanf with %s and %ws can read only read string without whitespaces.

5.3.1.2 getchar() function can be repeatedly used to read successive single characters from the input and place them into a character array. Thus the entire line of text can be read and stored in an array.

The getchar() function can take the following form

```
char ch;
```

```
ch=getchar();
```

The below program can read a line of text(maximum 80characters)into a string **line** using getchar() function. Every time a character is read, it is assigned to its location in the string **line** and tested for newline character. When the newline character is read, the loop is terminated and newline character is replaced by null character to indicate the end of the string.

```
/*Example program to read a line of text using getchar()*/
```

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

```
void main()
```

```
{
```

```
    char line[81],character;
```

```
    int c;
```

```
    c=0;
```

```
    printf("Enter text. Press<Return> at end\n");
```

```
    do
```

```
    {
```

```
        character=getchar();
```

```
        line[c]=character;
```

```
        c++;
```

```
    }
```

```
    while(character!='\n');
```

```
    c=c-1;
```

```
    line[c]='\0';
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
printf("\n%s\n",line);  
}
```

Output:

Enter text.Press<Return> at end

Programming in C is interesting.

Programming in C is interesting

5.3.1.3 gets() function is used to read a string including blank spaces until enter key is pressed. This is simple function with one string parameter.

```
gets(str);
```

where **str** is a string variable. It reads characters into **str** from the keyboard until a new line character is encountered and then appends the null character to the string. It does not skip whitespaces.

Example: char name[50];
 gets(name);

5.3.1.4 strcpy() function copies a string into a string variable.

Example: char name[50];
 strcpy(name,"kolar");

5.3.2 Writing

Character string is displayed by using

1. printf() function
2. putchar() function
3. puts() function

5.3.2.1 printf() function: The format %s can be used to print strings to the screen.

Example 1: printf("%s", name);

Can be used to display entire contents of array **name**.

The specification %10.4s indicates that the first four characters are to be printed in a field width of 10 columns. If we include the minus sign in the specification (eg: %-10.4s), the string will be left justified.

```
/*Program to display the string under various format specifications*/
```

```
Void main()
```

```
{
```

```
Char country[15]="United Kingdom";
```

```
Printf("%15s\n",country);
```

```
Printf("%15.6s\n",country);
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus


```
Printf(“%-15.6s\n”,country) ;  
}
```

Output:

United Kingdom

United

United

5.3.2.2 putchar() function is used to output the values of character variables. It requires one parameter.

It takes the following form

```
char ch='A';
```

```
putchar(ch);
```

putchar() function prints only one character at a time. We can use this function inside a loop to output a string characters.

Example: `char name[6]="PARIS";`

```
for(i=0;i<5;i++)
```

```
    putchar(name[i]);
```

```
    putchar('\n');
```

5.3.2.3 puts() function prints the value of the string variable and it requires one parameter.

```
puts(str);
```

where str is a string variable containing a string value.

Example: `char name[]= "education is power";`

```
puts(name);
```

5.4 Arithmetic operations on characters

When a character constant/variable is used in an expression it is automatically converted into an integer value by the system. If the machine uses ASCII codes then

```
x='a';
```

```
printf(“%d\n”,x);
```

Display the number 97 on the screen, as ASCII code of a is 97.

Simple arithmetic manipulations can be performed on characters.

Example:

```
1. int x;
```

```
    x='A'+1;
```

```
    printf(“%c”,x);/*displays B on monitor*/
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
2. int y=65;  
   y++; /*x becomes 66*/  
   printf("%c",y);/* displays B on monitor*/
```

The c library supports a function that converts a string of digits into their integer values.

The function takes the form

```
x=atoi(string);
```

where x is an integer variable, string is a character array containing a string of digits.

Example:

```
number="1988";  
year=atoi(number);
```

1. Number is a string variable which contains string constant "1988".
2. The function **atoi** converts the string "1988" to numeric 1988 and assigns it to the integer variable year.

5.5 Putting Strings together

Strings cannot be added directly that is,

```
Str3=str1+str2;  
Str2=str1+"hello"; are invalid.
```

The characters from str1 and str2 should be copied into the str3 one after the other. The size of the array str3 should be large enough to hold the total characters.

5.6 Comparison of two Strings

C does not permits comparison of two strings directly. Statements such as

```
if((name1==name2)  
if(name=="ABC"))
```

are not permitted. It is therefore necessary to compare two strings character by character. The comparison is done until there is a mismatch or one of the strings terminate to null character.

Following segment of the program illustrate this

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

Note:

refer "Reference Books" prescribed as per syllabus

5.7 STRING HANDLING FUNCTIONS

C supports many string handling functions. These are included in string.h header file. They are:

1. strcat()
2. strcmp()
3. strcpy()
4. strlen()

5.7.1 The strcat () function - string (joining) concatenation:

This function is used to join two strings. This function takes two arguments of type string. This process of joining two strings is called string concatenation.

Syntax: **strcat(string1, string2);**

Here, string1 and string2 are character arrays. When strcat is executed, string2 is appended to string1. the contents of the string2 gets appended to the content of string1.

Example:

Part1=

V	E	R	Y		\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Part2=

G	O	O	D	\0		
0	1	2	3	4	5	6

Part3=

B	A	D	\0			
0	1	2	3	4	5	6

Execution of the statement strcat(part1,part2);

Will result in

Part1=

V	E	R	Y		G	O	O	D	\0			
0	1	2	3	4	5	6	7	8	9	10	11	12

G	O	O	D	\0		
0	1	2	3	4	5	6

V	E	R	Y		B	A	D	\0				
---	---	---	---	--	---	---	---	----	--	--	--	--

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Part2= 0 1 2 3 4 5 6 7 8 9 10 11 12

While the statement `strcat(part1,part3)` will result in

Part1=

Part3=

B	A	D	\0			
0	1	2	3	4	5	6

We must make sure that the size of string1(to which string2 is appended) is large enough to accommodate the final string

Strcat function may also append a string constant to a string variable.

`strcat(part1,"GOOD");`

`/*Write a C program to concatenate two string using strcat() function*/`

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void
{
char str1[100], str2[100];
clrscr();
printf("Enter the
gets(str1);

printf("Enter the
gets(str2);
strcat(str1, str2);

printf("String1 after concatenation
getch();
}
```

Output:

Enter the string1:

Govt

Enter the string2:

Polytechnic

String1 after concatenation is: Govt Polytechnic

5.7.2 The strcmp () function - string compare:

strcmp() function compares two strings to find out whether they are identical or not. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first.

The **strcmp()** function accepts two strings as arguments and returns the integer value, after comparison, as follows:

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

1. A value of **zero** if the first string and the second string are **identical (equal)**.
2. A **positive value** if the second string is **greater than** the first string alphabetically.
3. A **negative value** if the first string is **less than** the second string alphabetically.

Syntax:

c=strcmp(str1, str2);

Here, the contents of the string2 gets compared with the content of string1.

Example:

```
int c;
char str1[ ]="abcd";
char str2[ ]="efgh"
c=strcmp(str1,str2);/*c becomes -1(negative integer)*/
```

ASCII value of a(97) is subtracted with ASCII value of e(101) that is, 97-101 which is -4, a negative value.

/*Write a C program to compare two string using strcmp() function*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void
{
char str1[100], str2[100];
int Diff;
clrscr();
printf("Enter
gets(str1);

printf("Enter
gets(str2);
Diff=strcmp(str1, str2);
if(Diff==0)
```

```
printf("String1 and String2 are equal");
```

```
else if(Diff>0)
```

```
printf("String1 is Greater than String2");
```

```
else
```

Output 1:

```
Enter the string1:
Govt.
Enter the string2:
Govt.
String1 and String2 are
equal
```

Output 2:

```
Enter the string1:
Govt.
Enter the string2:
GOVT.
String1 is Greater than
String2
```

Output 3:

```
Enter the string1:
GOVT.
Enter the string2:
Govt.
String1 is lesser than
String2
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
printf("String1 is Lesser than String2");  
getch();  
}
```

5.7.3The strcpy() function - string copy:

This function copies the contents of one string into another string including '\0'. This function takes two arguments of type **string** and copies contents of second argument to first one.

Syntax:

strcpy(str1,str2);

Here, the contents of **str2** is copied to **str1** and contents of **str2** remains unchanged. Str2 may be a character array variable or a string constant. The total size of the string str1 should always be greater than or equal to the size of the string str2.

Example:

```
char str1[ ]="Hello";  
char str2[ ]="Hai"  
strcpy(str1,str2);/*str1 becomes "Hai" and str2 remains equal to "Hai"*/
```

/*Write a C program to copy the contents from one string to another using strcpy() function*/

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

```
void  
{  
char src[100], target[100];
```

```
clrscr();  
printf("Enter  
gets(src);  
strcpy(target, src);
```

```
printf("\n The target string after copied from source string is %s", target);
```

OUTPUT:

Enter the source string

Mysore

in()

the

source

string:");

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
getch();  
}
```

Here, the `strcpy()` function copies the characters (“Mysore”) from the string **src** and assigns it to an target string **target**.

5.7.4 The strlen () function - string length

This function counts and returns the number of characters present in a string. It counts all the characters upto ‘\0’ (Null) Character but except the ‘\0’ (Null) character.

Syntax:

```
m=strlen(str);
```

Here, the integer variable **m** stores the length of the string **str**.

```
Example: char name[ ]="diploma";  
         int n;  
         n=strlen(name);/* n becomes 7*/
```

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

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
void  
{  
char  
int  
clrscr();  
printf("Enter  
gets(str);  
length=strlen(str);  
  
printf("\n Number of characters present in the string is %d", length);  
  
getch();  
}
```

OUTPUT:

Enter the string

Mysore

Number of characters present in the string is 6

Here, the `strlen()` function counts the number of characters (6) in the string **str** and assigns it to an integer variable **length**.

5.8 Other String handling functions:

5.8.1 The strncpy() function:

This function copies the first **n** characters from one string (Source string) into another (Target String) string.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

Syntax: strncpy(target, source, n);

It takes 3 arguments; target string, source string and the no of characters that needs to be copied specified by n . If the source string is less than n, then entire characters along with '\0' character is copied to target string.

Example: strncpy(s1,s2,5);

This statement copies first 5 characters of the source string s2 into the target string s1. Null character should be placed explicitly at 6th position as shown below.

s1[6]='\0';

5.8.2 The strncmp () function:

It takes 3 arguments and has the following form

strncmp(s1,s2,n);

This function compares the left-most n characters of s1 to s2 and returns

1. 0 if they are equal.
2. Negative number if s1sub-string is less than the substring s2.
3. Positive otherwise.

5.8.3 The strncat () function:

This function concatenates the first n characters of s2 to the end of s1. It takes three arguments and has the following form.

strncat(string1, string2, n);

Example:

S1:	B	A	L	A	\0								
	0	1	2	3	4	5	6	7	8	9	10	11	12
S2:	G	U	R	U	S	W	A	M	Y	\0			
	0	1	2	3	4	5	6	7	8	9			

After

strncat(s1,s2,4); execution

S1	B	A	L	A	G	U	R	U	\0
	0	1	2	3	4	5	6	7	8

5.8.4 Strstr() function

It is a two parameter function that can be used to locate a sub-string in a string. This takes the following form

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus


```
strstr(s1,s2);
```

```
strstr(s1,"ABC");
```

The function strstr() searches the string s1 to see whether the string s2 is contained in s1. If yes, the function returns the position of first occurrence of the sub string. Otherwise, it returns a null pointer.

Table of string Handling Functions:

Following tables shows the list of string handling functions defined in <string.h> header file.

Functions	Description
strcat(str1, str2)	Appends the string str2 to string str1
strcmp(str1, str2)	Compare the string str1 with string str2
strcpy(str1, str2)	Copies the contents of string str2 to string str1
l=strlen(string)	Counts and returns the number of characters in a string.
strncpy(str1, str2, n)	Copies first n characters of string str2 to another string str1
strncmp(str1, str2, n)	Compares first n characters of strings str1 with string str2
strncat(str1, str2, n)	Appends the first n characters of string str2 to string str1
strstr(str1, str2)	Finds first occurrence of a given string str2 in string str1

The Pre-Preprocessor

5.9 Introduction

A unique feature of c language is the preprocessor. A program can use the tools provided by preprocessor to make his program easy to read, easy to modify, portable and more efficient.

The *preprocessor* is a program that processes the source code before it passes through the compiler.

The preprocessor operates under the control of *preprocessor directives*. Normally Preprocessor directives are placed in the source program before the **main ()** before the source

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions are taken then the source program is handed over to the compiler.

The preprocessor directives always begin with the symbol # (hash) and it does not require semicolon at the end.

For Example:

```
#include<stdio.h>
#define PI 3.14
```

A set of commonly used preprocessor directives and their functions is given in table 7.1.

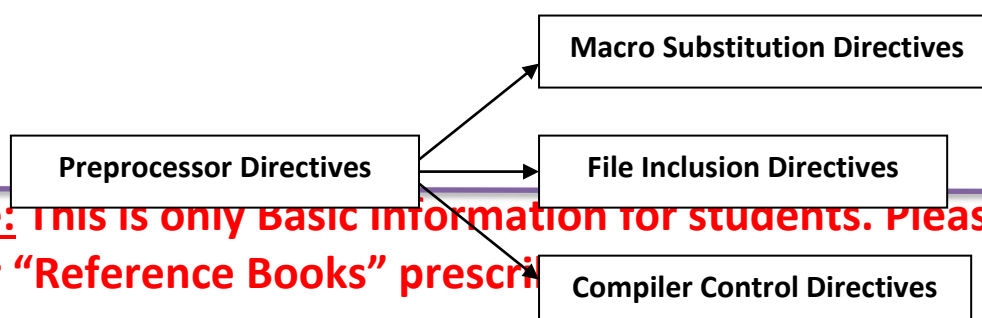
Directive	Function
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies a file to be included
#ifdef	Tests for macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether the macro is not def
#if	Tests a compile time condition
#else	Specifies alternatives when # if test fails

Table 7.1 Preprocessor directives

5.9.1 Types of Preprocessor Directives

The preprocessor directives can be divided into three categories

1. Macro substitution directives
2. File inclusion directives
3. Compiler control directives



Note: This is only basic information for students. Please refer "Reference Books" prescribed.

5.10 Macro Substitution Directives

The **macro substitution** is the process of replacing an **identifier** of a C program by a pre defined **string** composed of one or more tokens. This can be accomplished by the directive **#define** statement. This statement, usually known as a *macro definition* (or simply a *macro*) takes the following general form.

Syntax:

```
#define identifier string
```

If this statement is included in the program at the beginning, the preprocessor replaces every occurrence of the **identifier** in the source code by a string. The definition should start with the keyword **#define** and should followed by the *identifier* and a *string*, with at least one blank space between them. The string may be any text and identifier must be a valid c name.

For example, if you wish to use the word **ON** for the value 1 and the word **OFF** for the value 0, you could declare these two **#define** directives:

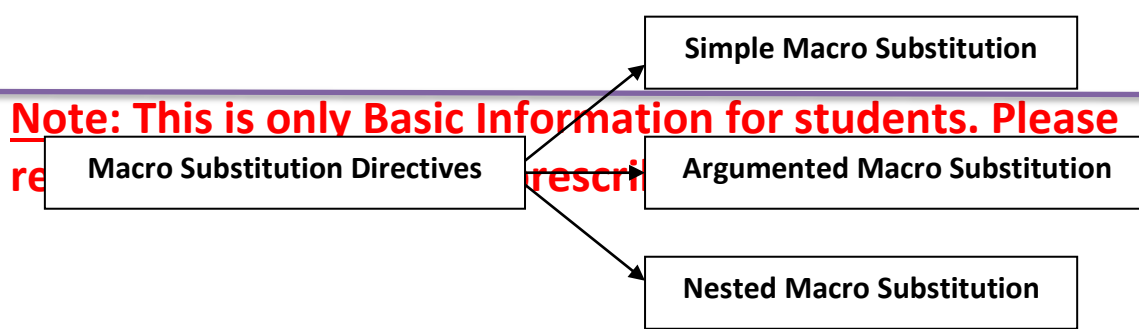
```
#define OFF    0
#define ON     1
```

Note: Notice that there is no semicolon in this statement. There may be any number of spaces between the identifier and the character sequence, but once the character sequence begins, it is terminated only by a newline.

5.10.1 Types of Macro Substitution

There are different forms of macro substitution. The most common forms are

1. Simple macro substitution
2. Argumented macro substitution
3. Nested macro substitution



5.10.1.1 Simple macro substitution

Simple macro substitution is commonly used to define constants.

For example, #define pi 3.1415926

Writing macro definition in capitals is a convention not a rule a macro definition can include more than a simple constant value it can include expressions as well. Following are valid examples:

```
#define AREA 12.36

#define MAX 100

# define COUNTRY "INDIA"
```

/* Write a program to illustrates simple macro substitution */

```
#include<stdio.h>
#include<conio.h>
#define pi 3.14
void main( )
{
    float r, area;
    clrscr( );
    printf("Enter the radius of a circle: ");
    scanf("%f",&r);
    area = pi*r*r;
    printf("Area of a circle = %.2f\n",area);
    getch( );
}
```

OUTPUT:

Enter the radius of a circle:5

Area of a circle = 78.50

/* Write a program to to illustrates simple macro substitution of literal text*/

```
#include<stdio.h>
#include<conio.h>
#define TEST if(x>y)
#define AND
#define PRINT printf("Very Good\n")
void main( )
{
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
clrscr( );  
TEST AND PRINT;  
}  
The preprocessor would translate TEST AND PRINT line to  
if(x>y) printf("Very Good\n");
```

5.10.1.2 Macros with Arguments

Like the functions, macros can have arguments. When a macro is called the preprocessor substitutes the string, replacing the formal parameters with the actual parameters.

Syntax:

```
#define Identifier(arg1, arg2 .... arg n) string
```

Where, **arg1, arg2, arg n** are the formal macro arguments (Like formal argument in function definition). There should be no space between macro identifier and left parenthesis.

There is a basic difference between simple replacement and replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a macro call.

A simple example of a macro with arguments is

```
#define CUBE (x) (x*x*x)
```

If the following statements appear later in the program,

```
volume=CUBE(side);
```

The preprocessor would expand the statement to

```
volume =(side*side*side)
```

/* Write a program to illustrate macro substitution with arguments */

```
#include<stdio.h>  
#include<conio.h>  
#define SQR(X) ((X) * (X)) // In ((X) * (X)), X is a formal parameter  
void main( )  
{  
    int n, square;  
    clrscr( );  
    printf("Enter the value of n: ");  
    scanf("%d",&n);  
    square = SQR(n); // n is an actual parameter  
    printf("Square of the number %d = %d", n, square);  
    getch( );  
}
```

OUTPUT:

Enter the value of n:10

Square of the number 10 = 100

5.10.1.3 Nesting of Macros

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

The process of defining one macro in the definition of another macro is called as Nested Macro Substitution.

For example,

```
# define SQUARE(x) ((x) * (x))
# define CUBE(x) (SQUARE(x) * (x))
# define SIXTH(x) (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro until no more macros appear in the text, For example the last definition, **SIXTH(x)** is expanded into

$(\text{SQUARE}(x) * (x)) * (\text{SQUARE}(x) * (x))$

SQUARE(x) is still a macro, it is further expanded into

$((((x) * (x)) * (x)) * ((x) * (x)) * (x)))$

which is finally evaluated to x^6

```
/* Write a program to illustrates the nesting of macros */
#include<stdio.h>
#include<conio.h>
#define SQR(X) ((X) * (X))
#define FOURTH(X) (SQR(X) * SQR(X))
void main( )
{
    int p, Square, Power4;
    clrscr( );
    printf("Enter the value of P\n");
    scanf("%d",&P);
    Square = SQR(P);
    Power4 = FOURTH(P);
    printf("Square of P = %d\n", Square);
    printf("P to the Power of 4 = %d\n", Power4);
    getch( );
}
```

OUTPUT:

Enter the value of P

3

Square of P = 9

P to the power of 4 = 81

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

}

In the above program, whenever the preprocessor finds the Identifier **SQR(P)**, it expands into **(X*X)** and whenever the preprocessor finds the Identifier **FOURTH(P)**, it expands into **(SQR(X) * SQR (X))**.

Undefining a Macro

A defined macro can be undefined using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

5.11 File Inclusion

File inclusion directives are the one form of preprocessor directives, which are used to include an external file (containing macros and function definition) into the current file (program).

Syntax:

#include "filename"

OR

#include <filename>

Where, **#include** is a *File inclusion directive*

Filename is the name of the file to be included that contains required definitions and functions.

File name can be written within a pair of angular brackets or double quotation marks as show below.

#include <stdio.h>

#include "cube.c"

When such statements appear in a program, the preprocessor inserts the entire code of function or macro to the current program.

1. **#include** <filename>

When the filename is written within a pair of angular brackets (< >), the preprocessor searches the specified file name only in the standard directories. The standard directories usually contain standard header files provided by the C compiler.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

For example <stdio.h>, <conio.h>, and <math.h> etc

Example: #include <stdio.h>

2. #include “filename”

When the filename is written within a pair of double quotes (“ ”), the preprocessor searches the specified file name in the current directory and then in the standard directories.

Example: #include “cube.c”

```
/*cube.c*/
cube(int x)
{
    int c;
    c = x*x*x;
    return(c);
}
```

/* Write a program to illustrates the inclusion of files */

```
#include<stdio.h>
#include<conio.h>
#include "cube.c"
main( )
{
    int n, cb;
    clrscr( );
    printf("Enter the value of n\n");
    scanf("%d",&n);
    cb = cube(n);
    printf("The cube of the number %d is %d", n, cb);
    getch( );
}
```

OUTPUT:

Enter the value of n

4

The cube of the number 4 is 64

Note: Nesting of included files is allowed. That is an included file can include another file, but a file cannot include itself. If an included file is not found, then an error is reported and compilation is terminated.

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

UNIT – 6 : Structures and Unions

6.1 Basics of Structures

Array can be used to represent a group of data items that belong to same data type. If we want to represent a collection of data items of different types using single name then structures are used, which is a mechanism of packing data of different types. Structures help us to organize complex data in a more meaningful way.

Example:

Time: seconds, minutes, hours

Book: author, title, price, year

6.1.1 Defining a Structure

Structure is a collection of data items of different types represented using single name. Variables in structure may be of same data type or of different data types. Each variable is called member of structure.

The **general format** of a structure definition is as follows:

```
struct tag-name
{
    Datatype    member 1;
    Datatype    member 2;
    .....
    Datatype    member n;
}; /*Semicolon is compulsory, which indicates the end of the structure
definition*/
```

where

struct is a keyword.

tag-name is the *name of the structure*

member 1, member 2...., member n are the *members* of the structures of similar or dissimilar data types.

semicolon (;) is used to terminate the structure definition.

Example:

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

The keyword **struct** declares a structure to hold the details of 4 data fields, namely **title**, **author**, **pages** and **price** which are called structure members.

- Book_bank is the name of the structure that is called structure tag or tag name. The tag name subsequently used to declare variables that have the tag's structure.

title	author	pages	price
(Array of characters)	(Array of characters)	(integer)	(float)
20 Bytes	15 Bytes	2 Bytes	4 Bytes

6.2 Declaration of Structure Variables:

A structure variable declaration includes following elements:

1. The keyword **struct**.
2. The structure **tag name**.
3. List of variable names separated by commas.
4. A terminating semicolon.

Syntax:

```
struct tag-name variable/s;
```

Example 1: `struct book_bank book1,book2,book3;`

Declares **book1**, **book2** and **book3** as variables of type **struct book_bank**.

Members of the structure themselves are not variables. Hence the structure definition does not reserve any memory space for the members (fields). The memory will be allocated only when structure variables are created.

It is possible to combine the definition and declaration of the structure variable in a statement as follows:

Example 2:

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
}book1,book2,book3;
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

The use of tag-name (structure name) is optional here. So the above two examples can be rewritten as follows.

Example 2:

```
struct
{
    char title[20];
    char author[15];
    int pages;
    float price;
}book1,book2,book3;
```

But this approach is not recommended as we cannot use it for future declarations.

6.3 Structure Initialization:

Structure variable can be initialized at compile time. The **compile –time initialization** of a structure variable must have the following elements:

1. The keyword **struct**
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

Example 2:

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
}book1={"introduction to c","balagurusamy",548,260.0};
```

This assigns "introduction to c" to book1.title,"balagursamy"to book1.author, 548 to book1.pages and 260.0 to book1.price.

Another method of initialization is

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

Void main()

{

book1={"introduction to c","balagurusamy",548,260.0};

.....

.....

}

C does not support initialization of individual structure members within the template.

Rules to be followed while initializing structure members and are shown below:

- i) We cannot initialize individual members inside the structure definition (or structure template).

For example

```
struct book_bank
{
    char title[20]= {"introduction to c"};
    char author[15]= {"balagurusamy"};
    int pages=548;
    float price=260;
};
```

is invalid.

- ii) The order of values enclosed in braces must match the order of members in the structure definition. **For example,**

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
}book1={"introduction to c","balagurusamy",548,260.0};
```

Here, titl=="introduction to c", author="balagurusamy", pages=548, price=260.0

- iii) It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.

- iv) The uninitialized members will be assigned default values as follows.

Zero (0) for integer and floating point numbers.

'\0' for characters and strings.

For example

```
struct book_bank book1={"introduction to c","balagurusamy" };
```

Here, pages and price are initialized to **zero** by default.

- v) During initialization, the members of initializes should not exceed the number of members. It leads to syntax error.

- vi) During initialization, there is no way to initialize members in the middle of a structure without initializing the previous members. For example,

```
struct student S1 = {1, 75.00}; is invalid
```

refer "Reference Books" prescribed as per syllabus

6.4 Structure Assignment:

The process of assigning the values of one structure variable to another structure variable of same type by using assignment operator (=) is called **structure assignment**.

We can also assign the values of one structure variable to another structure variable, member by member. When the numbers of members are very large, it is better to assign the values at once as shown below. Two variables of the same structure type can be copied the same way as ordinary variables.

/*Write a C program to assign the values from one structure variable to another */

```
#include<stdio.h>
#include<conio.h>
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
};

void main()
{
    struct book_bank book2,book3;
    clrscr();
    struct student book1={ introduction to c","balagurusamy",548,260.0};
    strcpy(book2.title,book1.author);
    strcpy(book2.author,book1.author);
    book2.pages=book1.pages;
    book2.price=book1.price;
    book3= book1;

    printf("book1 details\n");
    printf("-----\n");
    printf("title=%s\nauthor=%s\npages=%d\nprice=%f\n",book1.title,book1.author,book1.pages
    ,book1.price);
    printf("book2 details\n");
    printf("-----\n");
```

} **/*Copying member by**

/*Copying all the member at once

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
printf("title=%s\nauthor=%s\npages=%d\nprice=%f\n",book2.title,book2.author,book2.pages,book1.price);
printf("book3 details\n");
printf("-----\n");
printf("title=%s\nauthor=%s\npages=%d\nprice=%f\n",book3.title,book3.author,book3.pages,book3.price);
    getch();
}
```

Output:

Book1 details

title=introduction to c

author=Balagurusamy

pages=548

price=260.0

Book2 details

title=introduction to c

author=Balagurusamy

pages=548

price=260.0

Book3 details

title=introduction to c

author=Balagurusamy

pages=548

price=260.0

Accessing the members of a structure:

The members of a structure themselves are not variables they should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' This is also known as a **dot operator** or **period operator**.

A member of a structure can be accessed by specifying the **variable name** followed by a **period (also called dot)** which in turn is followed by the **member name**.

Syntax:

```
structure variable name.member name;
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

For example: **book1.price**

is the variable representing the price of book1 and can be treated like any other ordinary variable. We can assign the members of book1 in the following way

```
Strcpy(book1.title,"introduction to c");
```

```
Strcpy(book1.author,"balagurusamy");
```

```
Book1.pages=548;
```

```
Book1.price=260.0;
```

We can also use scanf to give values through the keyboard

```
scanf("%s",book1.title);
```

```
scanf("%s",book1.author);
```

 are valid input statements.

By specifying **book1.title**, we can access the title "introduction to c".

6.5 Nested Structures (Structure within structure):

Structures within a structure means *nesting of structures*. Following structure defined to store the information about the salary of employees.

Example:

```
struct salary
{
    Char name[10];
    char department[20];
    int basic_pay;
    int da;
    int hra;
    int city_allowance;
}employee;
```

This structure defines name, department, basic_pay and 3 kinds of allowances. We can group all the items related to allowance together and declare them under substructure as shown below

```
struct salary
{
    Char name[10];
    char department[20];
    int basic_pay;
    struct
    {
        int da;
        int hra;
        int city_allowance;
    }allowance;
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
} employee;
```

The **salary** structure contains a **member** named allowance, which itself is a structure with 3 members. The members contained in the inner structure namely da, hra and city_allowance can be referred to as

```
employee.allowance.da  
employee.allowance.hra  
employee.allowance.city_allowance
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outermost to innermost) with the member using the dot operator.

We can use tag names to define inner structures

```
struct pay  
{  
    int da;  
    int hra;  
    int city_allowance;  
};  
  
struct salary  
{  
    Char name[10];  
    char department[20];  
    int basic_pay;  
    struct pay allowance;  
    struct pay arrears;  
};  
Struct salary employee[100];
```

Pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

/*Write a C program to illustrate the use of Nested Structures */

```
#include<stdio.h>  
#include<conio.h>  
struct salary  
{  
  
    char name[20];  
    char department[15];  
    int basic;  
    struct pay  
    {  
  
        int da;  
        int hra;
```

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus


```
        int city_allowance;
    }allowance;
}emp;
void main( )
{
    printf("Enter the Employee name\n");
    scanf("%s",emp.name);
    printf("Enter the department \n");
    scanf("%s",emp.department);
    printf("enter the basic");
    scanf("%d",&emp.basic);
    printf("Enter the Salary Information\n");
    printf("Enter da \n");
    scanf("%d",&emp.allowance.da);

    printf("Enter  hra\n");
    scanf("%d",&emp.allowance.hra);

    printf("Enter city_allowance\n");
    scanf("%d",&emp.allowance.city_allowance);

    printf("-----\n");
    printf("The Employee Information is");
    printf("Name = %s\n",emp.name);

    printf("Department= %s\n",emp.department);

    printf("Basic Salary = %d\n",emp.basic);

    printf("Dearness Allowance = %d\n",emp.allowance.da);

    printf("House Rent Allowance = %d\n", emp.allowance.hra);

    printf("city allowance=%d",emp.allowance.city_allowance);

}
```

Output:

```
Enter the Employee name
John Smith
Enter department
Research
Enter the basic
50000
Enter the Salary Information
Enter da
25000
Enter hra
15000
Enter city_Allowance
10000

-----
The Employee information is
Name = John Smith
Department=Research
Basic Salary = 50000
Dearness Allowance = 25000
House Rent Allowance =1 5000
City allowance=10000
```

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

6.6 Structures and arrays:**6.6.1 Array of structures:**

For example, if we want to analyze the marks obtained by a class of students, we can use a template that describes **student name and marks** obtained in various subjects and then declare **all the students as structure variables**. In such cases we may declare an array of structures, each element of the array representing a structure variable.

```
struct class student[100];
```

Defines an array called student, that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration

```
struct marks
{
    int subject1;
    int subject2;
    int subject3;
};
main()
{
    struct marks student[3]={45,68,81},{75,53,69},{57,36,71}};
    .....
}
```

This declares the student as an array of 3 elements student[0], student[1], and student[2] where each element of student array is a structure variable with three members. Thus, the members can be initialized as follows

```
student[0].subject1=45;
student[1].subject1=68;
....
....
student[2].subject1=71;
```

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array student is represented as shown below in figure 10.1

student[0].subject1	45
student[0].subject2	68
student[0].subject3	81
student[1].subject1	75
student[1].subject2	53
student[1].subject3	69

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

student[2].subject1	57
student[2].subject2	36
student[2].subject3	71

Fig.10.1 The array student inside the memory

6.6.2 Arrays within structures:

C allows us to use arrays (Single and multi dimensional) within structures. One of the most common applications of this type is in setting up an array of characters inside a structure. Similarly we can use single or multi dimensional arrays of type int or float. For example, the following declaration is valid

Example: struct marks
 {
 int number;
 float subject[3];
 }student[2];

In the above example, the member subject contains 3 elements **subject[0]**, **subject[1]**, **subject[2]** and these elements can be accessed by using appropriate subscripts.

For example, **student[1].subject[2]** is used to access the marks scored in third subject by the second student.

6.7 Structures and functions:

Like any other variables, structure variables can be passed as arguments to the functions. This can be done using the following ways.

- Passing individual structure members.
- Passing entire structure.
- Passing by pointers

Case 1:

Just like an ordinary variables, the structure members can be passed individually.

For Example:

/*Write a C program to illustrate the passing individual members of the structure as an arguments to the function*/

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct student
    {
        int reg_num;
        char name[20];
        int class;
        int age;
    };
    struct student stud = {12345, "Savitha", 2, 16};
    show(stud.reg_no, stud.name, stud.class, stud.age);
}

show(int roll_num, char name, int sem, int age) /*Function to display structure variables*/
{
    printf("Reg.No = %d\nName = %s\nClass = %d\nAge = %d",roll_num, name, sem, age);
    getch( );
}
```

Output:

Reg.No = 12345

Name = Savitha

This method is inefficient and unmanageable when the number of structure elements are very large. So it is better to pass the entire structure variable as an argument.

Note: It is possible to pass few members of the structure as an argument to the function.

Case 2:

This approach is very efficient when the number of structure elements are very large . In this approach, a copy of the entire structure is passed as argument to the **called function**. *But any changes to the members of the structure within the called function will not affect the original structure.* Hence, it is necessary for the function to return the entire structure back to the calling function. Syntax of function call is shown below

Syntax:

function_name(structure_variable_name);

General form of called function is

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

```
data-type function_name(struct _type st_name)
{
    .....
    .....
    return(expression);
}
```

/*Write a C program to illustrate the passing an entire structure variable as an arguments to the function*/

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

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

```
main( )
```

```
{
```

```
    struct student
```

```
    {
```

```
        int reg_num;
```

```
        char name[20];
```

```
        int class;
```

```
        int age;
```

```
    };
```

```
    struct student stud = {12345, "Savitha", 2, 16};
```

```
    show(stud);
```

```
}
```

```
show(struct student stud1)
```

/*Function to display structure variables*/

```
{
```

```
    printf("Reg.No = %d\nName = %s\nClass = %d\nAge = %d",reg_no, name, class, age);
```

```
    getch( );
```

```
}
```

Case 3:

Pointers are used to pass the structure as an argument which is referred as call by reference parameter passing method. Here, the address location of the structure is passed to the called function. The function can access indirectly the original structure, any changes made to the structure members with in the function are directly reflected on original structure. Hence there is no need to return the entire structure back to the calling function.

6.8 UNIONS

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

A union is a collection of heterogeneous elements. That is, it is a group of elements. Each element is of a different type. **Defining Unions are similar to structures except the keyword struct.** However, there is a major difference in the way the structure member and union members are stored. *Each member within a structure is assigned its own memory location. But the union members share a common memory location.* Thus, unions can handle only one member at a time.

Unions are chosen for applications involving multiple members, where values need to be assigned to all of the members at any one time.

Syntax:

```
union tag-name
{
    Datatype member 1;
    Datatype member 2;
    .....
    Datatype member n;
}; /*Semicolon is compulsory, which indicates the end of the union definition*/
```

Where, **union** is a *keyword*, which indicates that the union is defined

tag-name is the *name of the union (Union name)*

member 1, member 2....Member n are the *members* or *fields* of the structures of similar and/or dissimilar datatypes.

Semicolon (;) is used to terminate the union definition.

Example: **union student**

```
{
    int reg_num;
    char name[20];
    float average;
};
```

Where, **union** is a *keyword*, which indicates that the union is defined

student is the *tag_name of the union*

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus

reg_num, name, average are the *members* or *fields* of the union of int, char, , float data types respectively.

Semicolon (;) is used to terminate the union definition.

Here, the union definition does not reserve any memory space for the members (fields).Memory allocation is done only when union variables is declared.

6.8.1 Declaration and accessing of a union:

Once the union is defined, it is necessary to declare the union variable.

Syntax:

```
union tag-name variable1,variable2....;
```

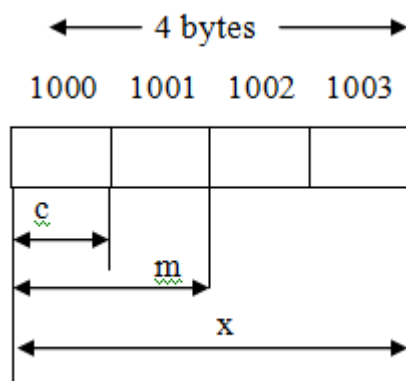
variable1, variable2... are the name of the union variable of type **tag-name**

Example: **union item**

```
{
    int m;
    float x;
    char c;
}code;
```

This declares variable **code** of type **union item**.

The union contains 3 members m,x and c with data types of type int, float and char respectively. Since only one location is allocated for a union variable **code**, **only one union member is accessed at a time**.



Sharing of a storage location by union members

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. The above figure shows that all the 3 members are sharing the same memory location of variable **x**, which is largest (that is 4 bytes) among 3 members.

Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

To access a union member following statements can be used

code.m

code.x

code.c are valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m=379;
```

```
code.x=7859.36;
```

```
printf(“%d”,code.m);
```

would produce erroneous output. Since union creates a storage location that can be used by any one of its members. As in the above statements **m** value is overwritten by **x**, if we try to print the value of **m** after overwriting then error will be generated.

6.9 Size of structures:

The sum of the size of the each individual members of the structure is called as size of the structure.

The unary operator **sizeof()** to tell us the size of the structure (or any variable). The **sizeof()** operator returns the size of its operand in bytes. The expression

sizeof(struct x)

will returns the number of bytes required to hold all the members of the structure x.

If y is a structure variable of type **struct x** then

sizeof(y) will produce the same result.

/*Write a program to find the size of the structure using sizeof() operator*/

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

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

```
void main( )
```

```
{
```

```
    struct student
```

```
    {
```

```
        int reg_num;
```

```
        char name[20];
```

```
        float average;
```

Output:

The Size of the structure student is 26

Bytes

Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus


```

    }stud;
    printf("The Size of the structure student is %d Bytes", sizeof(stud));
    getch( );
}

```

reg_num (Integer)	name (Array of characters)	average (Float)	
2 Bytes	20 Bytes	4 Bytes	= 26 Bytes

Note:

The size of the structure may vary from one platform to another platform based on the size of primitive (basic) data type on those platforms.

For Example: Integer is 2 bytes in some platform and 4 bytes in some

6.9.1 Difference between structures and unions:

The syntax of defining a structure and union is same except the keywords struct and union. The way the structure and unions access their members using dot operator is same. A structure can be a member of union, and a union can be a member of structure. Even though some similarities exist, there are some differences between the two. The difference between the structures and unions lie in the way their members are stored and initialized.

Structure	union
1. The keyword struct is used to define a structure	1. The keyword union is used to define a union.
2. The compiler will allocate the memory for each member and so size of memory allocated is greater than or equal to the sum of sizes of its members.	2. The size of the memory allocated by the compiler is the size of the member that occupies the largest space.
3. Each member within a structure is assigned its own unique storage area.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order i.e., given any member, the members following this will be assigned addresses in increasing order. This indicates	4. The address is same for all the members of a union. This indicates that every member begins at offset zero.

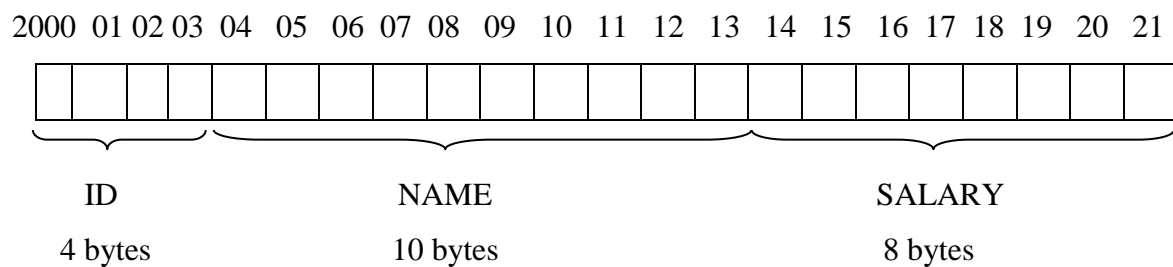
Note: This is only Basic Information for students. Please refer "Reference Books" prescribed as per syllabus

that memory for each member will start at different offset values.	
5. Individual members can be accessed at a time.	5. Only one member can be accessed at a time.

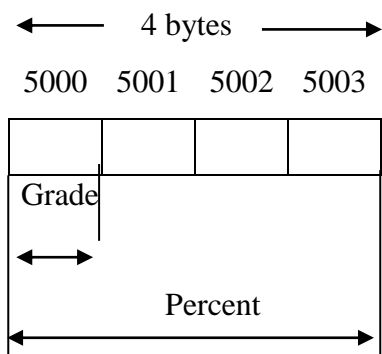
Memory mapping for structure is as follows (In General):

Base address

a ↓



Memory mapping for union is as follows:



Note: This is only Basic Information for students. Please refer “Reference Books” prescribed as per syllabus