CP MODULE 1

Introduction to C Language: Preprocessor directives, header files, data types and qualifiers. Operators and expressions. Data input and output, control statements.

INTRODUCTION TO C

C language was originally developed in the 1970s by Dennis Ritchie at Bell Telephone Laboratories.

BCPL and B languages are the ancestors of C language.

Properties of C language

- *C is a general-purpose, structured programming language.*
- *C is procedure oriented language.*
    - *C language program is composed of procedures or functions.*
- C is used to write very concise source programs,
- Instructions in C language resemble algebraic expressions.
- C language is Englike like. It uses keywords such as **if, else, for, do** and **while**
- C language has a relatively small instruction set.
    - But actual implementations of C include extensive library functions that can enhance the basic instructions.
    - Features and capabilities of the language can easily be extended by the user by writing additional library functions
- C programs are and efficient, fast highly portable
    - C programs written on one computer can be run on another computer with little or no alteration

Structure of a C Program

- Every C program consists of one or more modules called functions.
- One of the functions must be called main() function.
- The program will always begin by executing the main function,
- The main() function calls other functions. Any other function definitions must be defined separately, either ahead of or after main.

Each function must contain:

1. A ***function heading***, which consists of the function name, followed by an optional list of arguments, enclosed in parentheses.
2. A *list of* ***argument declarations***,if arguments are included in the heading.
3. A ***compound statement***, which comprises the remainder of the function.

GENERAL FORMAT OF C PROGRAM

```
Preprocessor directives
Function prototypes
Global variable declarations
main()
{
    Local variable declarations
    Statements
    Function calls
    Executable statements
}
Other function definitions
```

The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as ***parameters***.)
Each compound statement is enclosed within a pair of braces, i.e., { }.
The braces may contain one or more elementary statements (called expression statements) and other compound statements.
Thus compound statements may be nested, one within another.
Each expression statement must end with a semicolon (;)

Comments (remarks) are non-executable statements. They may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., /* this is a comment */).
Use of comments: comments are helpful

- In identifying the program's principal features
- in explaining the underlying logic of various program features.
- For documentation

Comments can be single line comment or multi line comment
A statement is a single line comment if it starts with //
Example
//a=10;

Multiline comments sare used for making more than one statement as comments.
Multi line comment begins with /* and it ends with */. I.e Every statement within /* and */ are treated as comments(t executable statements)
Example
/* a=10;
b=2;*/

These statements are not executed.

Preprocessor directives

The C preprocessor is a collection of special statements, called directives, that are executed at the beginning of the compilation process.

- Preprocessor statements starts with symbol #
    - E.g. for preprocessor directives are **#include ,#define, #if, #elif, #else, #endif, #ifdef, #ifndef, #line** and **#undef.**
- The preprocessor includes *three special operators:* **defined, #, and ##.**
- Preprocessor directives usually appear at the beginning of a program.
    - But they can be written anywhere in the program.
- The "stringizing" operator # allows a formal argument within a macro definition to be converted to a string.
    - If a formal argument in a macro definition is preceded by this operator, the corresponding actual argument will automatically be enclosed in double quotes.
- The "token-pasting" operator ## causes individual items within a macro definition to be concatenated, to form a single item.

E.g. for preprocessor statements:

The following statement help to include the contents of the header file stdio.h within the program.

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

E.g.

```
#define NULL 0
```

This statement will define the symbolic constant NULL to zero.

E.g. if symbolic constant FOREGROUND has already been defined, the symbolic constant BACKGROUND will represent the value 0. Otherwise,FOREGROUND and BACKGROUND will represent the values 0 and 7, respectively. This is represented using the following preprocessor statements:

```
 #if defined(FOREGR0UND)

        #define BACKGROUND 0
#else
        #define FOREGROUND 0
        #define BACKGROUND 7
#endif
```

The below preprocessor statements are same as above code.

```
#ifdef FOREGROUND
#define BACKGROUND 0
#else
#define FOREGROUND 0
#define BACKGROUND 7
#endif
```

The #undef directive “undefmes” a symbolic constant or a macro identifier; i.e., it negates the effect of a #define directive.

The directive #ifdef is equivalent to #if defined( ). Similarly, the directive #ifndef is equivalent to #if defined ( ) , i.e., “if not defined.”

Header files

Machine-dependent features can be provided as library functions, or as character constants or macros that can be included within the header files.

C places the required library-function declarations in special source files, called **header files.** Most C compilers include several header files. Each header file contains declaration of functions that are related. Header file uses the extension **.h**.

Syntax of # include to include header file is

#include <filename>

where filename represents the name of a special file.

- If filename is system defined file it should be enclosed in angular brackets <>.
    - E.g stdio.h is a system defined file

.#include <stdio.h>

- If the file is user defined it can be enclosed in double quotes.
    - E.g. Let sample.h is a user defined header file it can be included using

# include”sample.h”

For example:

- stdio.h is a header file containing declarations for input/output routines(functions).
    - printf(),scanf()
- math.h contains declarations for certain mathematical functions
    - pow(),sqrt()
- ctype.h
    - contains function declarations such as toupper() tolower()
- The required header files must be merged with the source program during the compilation process.
    - This is accomplished by placing one or more #include statements at the beginning of the source program.
    - E.g. if we are using mathematically functions, we have to write the following in the program

      #include math.h

#include <stdio. h> statement help to include the contents of the header file stdio.h within the program.

C fundamentals

Basic elements used to construct C statements are:
C character set, identifiers and keywords, data types, constants, variables and arrays, declaration, expression and statements

C character set

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters

+	-	*	/	=	%	&	#
!	?	^	"	'	~	\	|
<	>	(	)	[	]	{	}
:	;	.	,	_	*(blank space)*		

Identifiers and keywords

Identfiers are names that are given to various program elements, such as variables, functions and arrays.

- Identifiers consist of letters and digits. Underscore _ is also treated as a letter.
- first character must be a letter or underscore
- Case sensitive. Upper- and lowercase letters are not interchangeable
- Keywords cannot be used as identifiers

E.g for valid identifiers.

X	Y12	sum_1	_ t e m p e r a t u r e	
names	area	tax_rate	TABLE	table

Here table and TABLE are treated as different identifiers

The following names are not valid identifiers for the reasons stated.

4th	The first character must be a letter.
"X"	Illegal characters "
order-no	Illegal character (-).
error flag	Illegal character (blank space).

An identifier can be arbitrarily long. Some implementations of C recognize only the first eight characters.

Keywords

Keywords are reserved words. They have standard, predefined meanings in C.
The standard keywords are
auto extern sizeof break floatn static case for struct char goto switch const if typedef int union continue default long unsigned do register void double return volatile else short while enum signed

Keywords are all lowercase. They cannot be used as identifiers.

DATA TYPES & QUALIFIERS

C supports several different types of data, each of which may be represented differently within the computer's memory.
The basic data types are
Memory requirements for each data type may vary from one C compiler to another.

Data Type	*Description*	*Memory Requirement*
int	integer quantity	2 bytes or one word (varies from one compiler to another)
char	single character	1 byte
float	floating-point number (i.e., a number containing a decimal point and/or an exponent)	4 bytes (1 word)
double	double-precision floating-point number (i.e., more significant figures, and an exponent which may be larger in magnitude)	8 bytes(2 words)

Qualifiers help to improve the features of basic data type.
Data type qualifiers are
short
long
signed
Unsigned
Integer data types
Integer quantities can be defined as int, short int, long int, signed int or unsigned int.

- If **short int** and int both have the same memory requirements (e.g., 2 bytes), then
  - ***long int*** will generally have double the requirements (e.g., 4 bytes).
- if int and ***long int*** both have the same memory requirements (e.g., 4 bytes) then
  - ***short*** int will generally have half the memory requirements (e.g., 2 bytes).

	short int	**long int**
Space occupied in memory	2 bytes	4 bytes
Speed	Programs run faster	Run slow

Range	-32,768 to 32,767	-2147483648 to -2147483647
Format(control)string	%d or %i	%ld or %li
E.g.	short int a;	long int b;

When integer variable is defined with no qualifiers, the default is short signed int
It means
int a;
is same as
short signed int a;

An***unsigned int***has the same memory requirements as an ordinary int.

- In the case of an ordinary ***int*** (or a short int or a long int), the *leftmost bit is reserved for the sign.*
- With an ***unsigned int***, all of the bits are used to represent the numerical value.
    - Thus, an unsigned int can be approximately twice as large as an ordinary int (negative values are not permitted because it is unsigned)

	int	Unsigned int
Space occupied in memory	2 bytes	2 bytes
Range	-32,768 to 32,768	0 to 65,535
Format(control)string	%d or %i	%u

By default unsigned int is short unsigned int.

long signed int	long unsigned int
-2147483648 to 2147483647	0 to 4294967295

Character types
Char type is to represent individual characters.
Occupies ***1 byte*** in memory

signed char	unsigned char
-127 to 128	0 to 255

FLOAT AND DOUBLE TYPES

	float	**double**
Space occupied in memory	4 bytes	8 bytes

Range	$3.4e^{-38}$ to $3.4e^{38}$	$1.7e^{-308}$ to $1.7e^{308}$
Format(control)string	%f	%lf

Some compilers permit the qualifier long to be applied to float or to double, e.g., long float, or long double.
Every identifier that represents a number or a character within a C program must be associated with one of the basic data types before the identifier appears in an executable statement. This is accomplished via a type declaration.

Other data types are void and enum.

CONSTANTS
There are four basic types of constants in C. They are :
integer constants,floating-point constants, character constants and string constants

Integer and floating-point constants represent numbers. They are often referred to as ***numeric-type constants***.
The following rules apply to all numeric-type constants.
1. Commas and blank spaces cannot be included within the constant.
2. The constant can be preceded by a minus (-) sign if desired. (Actually the minus sign is an operator that changes the sign of a constant to negative)
3. The value of a constant cannot exceed specified minimum and maximum bounds.
(These bounds will vary from one C compiler to another.)
Integer Constants
An integer constant is an integer-valued number. Thus it consists of a sequence of digits. Integer constants can be written in three different number systems: decimal (base lO), octal (base 8) and hexadecimal (base 16).
A **decimal integer constant** can consist of any combination of digits taken from the set 0 through 9.
If the constant contains two or more digits, the ***first digit must be something other than 0.***
Some valid decimal integer constants are.

0	***1***	***743***	***5280***	***32767***	***9999***

The following decimal integer constants are **invalid**.

12,245	***illegal character (,).***
36.0	***illegal character (.).***
10 20 30	***illegal character (blank space).***
123-45-6789	***illegal character (-).***
0900	***the first digit cannot be a zero. If number has 2 or more digits***

An ***octal integer constant*** can consist of any combination of digits taken from the set 0 through 7. However the ***first digit must be 0***,in order to identify the constant as an octal number.
Somel **valid** octal integer constants are.

0	01	0743	077777

Some invalid octal integer constants are:

743	Does not begin with 0.
05280	Illegal digit (8).
0777.777	Illegal character (.).

A ***hexadecimal integer constant*** must begin with either Ox or OX. It can then be followed by any combination of digits taken from the sets 0 through 9 and a through f (either upper- or lowercase). The letters a through f (or A through F) represent the (decimal) quantities 10 through 15, respectively.

Some **valid** hexadecimal integer constants are:

0x	0x1	OX7FFF	Oxabcd

Some **invalid** hexadecimal integer constants are:

OX12.34	Illegal character (.).
OBE38	Does not begin with Ox or OX.
Ox. 4bff	Illegal character (.).
OXDEFG	Illegal character (G).

Typical maximum value for most personal computers and many minicomputers is 32767 decimal (equivalent to 77777 octal or 7FFF hexadecimal), which is 2^{15}- 1.

Unsigned and Long Integer Constants

An *unsigned integer* constant can be identified by appending the letter **U** (either upper- or lowercase) to the end of the constant.

An *unsigned long integer* may be specified by appending the letters **UL** to the end of the constant. The letters may be written in either upper- or lowercase. However, the U must precede the L.

Constant	Number System
50000U	decimal (unsigned)
123456789L	decimal (long)
123456789UL	decimal (unsigned long)
0123456L	octal (long)
077777711	octal (unsigned)
OX50000U	hexadecimal (unsigned)
OXFFFFFUL	hexadecimal (unsigned long)

Floating-Point Constants

A floating-point constant is a base- 10 number that contains either a decimal point or an exponent (or both).

Some **valid** floating-point constants are:

0.	1 .	0.2	827.602
50000.	0.000743	12.3	315.0066
2 E-8	0.006e-3	1.6667E+8	.12121212e12

Some **invalid** floating-point constants are:

1	Either a decimal point or an exponent must be present.

1,000.0	Illegal character (,).
2E+10.2	The exponent must be an integer quantity (it cannot contain a decimal point).
3E 10	Illegal character (blank space) in the exponent.

If an exponent is present, its effect is to shift the location of the decimal point to the right, if the exponent is positive, or to the left, if the exponent is negative.
If a decimal point is not included within the number, it is assumed to be positioned to the right of the last digit.
The number 1.2 **x 10**$^{-3}$would be written as **1 .2E-3** or **1 .2e-3.** This is equivalent to **0.12e-2,** or **12e-4,** etc.

E.g The quantity **3** x 10^{5}can be represented in C by any of the following floating-point constants.

300000.	**3e5**	**3e+5**	**3E5**	**3.Oe+**
.3e6	**0.3E6**	**30E4**	**30. E+4**	**300e3**

0.0 (which is less than either **3.4E-38** or **1 .7E-308)** is a valid floating-point constant.

Character Constants
A ***character constant*** is a single character, enclosed in apostrophes (i.e., single quotation marks).
Some character constants are shown below.

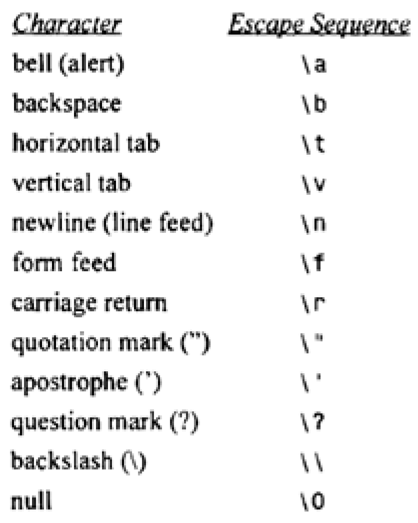
' A '   ' X'   '3'   '?'   ' '

The constant ' 'consists of a blank space, enclosed in apostrophes.

Most computers, and virtually all personal computers, make use of the ASCII (i.e., American Standard Code for Information Interchange) character set, in which each individual character is numerically encoded with its own unique 7-bit combination (hence a total of 2^7 = **128** different characters).

Character	ASCII value
A	65
B	66
Z	90
a	97
z	122
0	32
10	42

Escape Sequences
Certain nonprinting characters, as well as the backslash (\) and the apostrophe ('), can be expressed in termsof ***escape sequences.*** An escape sequence always begins with a backward slash and is followed by one ormore special characters.
For example, a line feed **(LF),** which is referred to as a ***newline*** in C, can be represented as \n.



Character	*Escape Sequence*
bell (alert)	\a
backspace	\b
horizontal tab	\t
vertical tab	\v
newline (line feed)	\n
form feed	\f
carriage return	\r
quotation mark (")	\"
apostrophe (')	\'
question mark (?)	\?
backslash (\\)	\\\\
null	\0

String Constants
A ***string constant*** consists of any number of consecutive characters (including none), enclosed in (double) quotation marks.
Some string constants are shown below.

"green"	"Washington, **D.C.** 20005”	"270-32-3456"
"$19.95"	"THE CORRECT ANSWER **IS:”**	**“2*** (**I+3**)/J "
“ ”	"Line l\nLine 2\nLine **3"**	“”

VARIABLES AND ARRAYS
Variable

- **A *variable*** is an identifier that is used to represent some specified type of information in a program.
- A variable is an identifier that is used to represent a single data item; i.e., a numerical quantity or a character constant etc.
- The data item must be assigned to the variable at some point in the program.
    - o The data item can then be accessed later in the program by referring to the variable name.
- A given variable can be assigned different data items(values) at various places within the program.
    - o Thus, the information represented by the variable can change during the execution of the program.
- The datatype associated with the variable cannot change.

E.g.
int a, b, c;
char d;
These statements are called type declarations.
Here a , b and c are integer variables
d is character variable
Thus a , **b** and **c** will each represent **an** integer-valued quantity, and **d** will represent a single character.

a = **3;**
b = **5;**
c = a + b ;
d = ‘p’ **;**
These are assignment statements.
Here the integer quantity **3** is assigned(stored) to **a,** ***5*** is assigned to **b,**and the quantity represented by the sum a + **b** (i.e., **8)** is assigned to **c.** The character **'p'** is then assigned to **d.**

Array
The ***array*** is another kind of variable that is used extensively in C.

- An array is an identifier that refers to a***collection*** of data items that all have the same name.
- The data items must all be of the same type (e.g., allintegers, all characters, etc.).
- The individual data items are represented by their corresponding ***array-elements***(i.e., the first data item is represented by the first array element, etc.).
- The individual array elements aredistinguished from one another by the value that is assigned to a ***subscript.***

Integer array is a collection of integers.
Character array is a collection of characters.
Suppose that **x** is a 10-element array. The first element is referred to **as x** [0] ,the second **as x** [1] ,andso on. The last element will be **x** [9].
E.g
int x[10];
This is the declaration of an integer array x of sixe 10
Thus, the value of the subscript for the **first** element is 0, the value of the subscript for the second element is 1, and so on. For *an* n-element array, the subscripts always range from *0* to n-1
.

DECLARATIONS

- **A** ***declaration*** associates a group of variables with a specific data type.
- All variables must be declared before they can appear in executable statements.
- **A** declaration consists of a data type, followed **by** one or more variable names, ending with a semicolon.

E.g.
int a, b, *c ;*
float r1,r2;
char flag, text[80];
Thus, **a, b** and c are declared to be integer variables, **r1** and **r2** are floating-point variables, **flag** is a char-type variable and **text** is an 80-element, char-type array.

These declarations can be also written as

int a;
int b;

int c;
float r1;
float r2;
char flag;
char text[80];

Integer-type variables can be declared to be ***short integer*** for smaller integer quantities, or ***long integer*** for
larger integer quantities.
short int a, **b,** c;
long int r, **s,** t;
int **P,** q;

The above declarations could have been written as
short **a, b,** c;
long r, **s,** t ;
int **P,** 9;
Floating-point variables can be declared to be ***double precision*** by using the type indicator **double** or
long float rather than float .
char t[ ] = "California";
This declaration will cause t to be an 11-element character array. The first **10** elements will represent the 10characters within the word California, and the 1lth element will represent the null character **(\0)**which is automaticallyadded at the end of the string.

char t[11] = "California";
where the size of the array is explicitly specified. In such situations it is important, however, that the size be specifiedcorrectly.

If the size is too small
E.g.
char t[10] = "California";
the characters at the end of the string (in this case, the null character) will be lost.

If the size is too large, e.g.,
char t[20] = "California";
the extra array elements may be assigned zeros, or they may be filled with meaningless characters.

EXPRESSIONS

- An ***expression*** represents a **single data item**, such as a number or a character.
- The expression may consist of a **single entity**, such as a constant, a variable, an array element or a reference to a function.
- It may also consist of some **combination of such entities,** interconnected by one or more ***operators.***
- Expressions can also represent logical conditions that are either true or false.

- In C the conditions ***true*** and ***false*** are represented by the integer values **1** and ***0,***respectively.

Some expressions are shown below.
a + b
x = y
c = a + b
x <= y
x == Y
++i
The first expression involves use of the ***addition operator*** (+). The second expression involves the ***assignment operator*** (=). In this case, the expression causes the value represented by **y** to be assigned to **x.** In the third line, the value of the expression **a**+**b)** is assigned to the variable **c.**
The last expression ++I causes the value of the variable I to be increased by 1 (i.e., incremented). Thus, the expression isequivalent to
i = i + 1

STATEMENTS
A ***statement*** causes the computer to carry out some action. There are three different classes of statements in C.
They are ***expression statements, compound statements*** and ***control statements.***
An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Some expression statements are
a = **3;**
c = a + b ;
++i;
printf ("Area = **%d", a) ;**
;
The first two expression statements are assignment-type statements. Each causes the value of the expression on the right of the equal sign to be assigned to the variable on the left.
The third expression statement is an incrementing-type statement, which causes the value of I to increase by 1.
The fourth expression statement causes the printf function to be evaluated.It prints Area= and the value of variable a.OUTPUT IS Area=3

The last expression statement ; does nothing, since it consists of only a semicolon. It is simply a mechanism forproviding **an** empty expression statement in places where this type of statement is required. Consequently, it is called a***null statement.***

A **compound statement** consists of several individual statements enclosed within a pair of braces { }
The individual statements may themselves be expression statements, compound statements or controlstatements.
A compound statement does ***not*** end with a semicolon.

E.g

{

a=2;

b=3;

c=a+b

}

Control statements are used to create special program features, such **as** logical tests, loops and branches. They uses **if, else, for, while, do while.**

SYMBOLIC CONSTANTS

A symbolic constant is a name that substitutes for a sequence of characters.
The characters may represent anumeric constant, a character constant or a string constant. Thus, a symbolic constant allows a name to appearin place of a numeric constant, a character constant or a string.
A symbolic constant is defined by writing
#define *name text*
where ***name*** represents a symbolic name, typically written in uppercase letters, and ***text*** represents thesequence of characters that is associated with the symbolic name.
Here ***text*** does not end with a semicolon, since a symbolic constant definition is not a true C statement. Moreover, if ***text*** were to end with a semicolon, this semicolon would be treated **as** though it were a part of the numeric constant, character constant or string constant that is substituted for the symbolic name.
Usually symbolic names are written in uppercase.

# define TRUE 1
# define PI 3.14
Suppose that the program contains the statement
area = PI * **radius** * **radius;**
During the compilation process, each occurrence of a symbolic constant will be replaced by its corresponding text. Thus,the above statement will become
area = 3.141593 * radius * radius;

OPERATORS AND EXPRESSIONS

There are different types of operators:
arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operator
These operators can be used to form expressions in C.
The data items that operators act upon are called **operands**.
Some operators require two operands, are called **binary operators**., Some operators act upon only one operand. They are called unary operators

Operands can be an expressions or single variables.

ARITHMETIC OPERATORS
There are 5

Operator	*Purpose*
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after integer division

The %operator is sometimes referred to as the modulus operator.
There is <u>*no exponentiation*</u> operator in C. There is a library function called pow() to perform exponentiation.
Operands can be integer quantities, floating-point quantities or characters.
The remainder operator (**%**) requires that ***both operands be integers*** and the ***second operand be nonzero.*** Division operator (/) requires that the second operand be nonzero.

If a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient. Otherwise result will be truncated quotient.

Suppose integer variables a=10 and b=3
Result of a+b is 13
a/b is 3
a%3 is 1

Suppose floating point variables c=12.5 and d=2.0
Result of c+d is 14.5
c/d is 6.25
c%d is not allowed. Error. ***Operands of % operator should be integers.***

Suppose that c 1 and c2 are character-type variables that represent the characters P and **T,** respectively.
Characters **P** is encoded as (decimal) **80, T** is encoded as **84,** and **5** is encoded as 53 in the ASCII character set,

Expression	*Value*
c1	80
c1 + c2	164
c1 + c2 + 5	169
c1 + c2 + '5'	217

The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C assign the sign of the first operand to the remainder.
Suppose a=11 and b=-3

Then result of
a/b is -3
a%b is 2

Suppose a=-11 and b=3
Then result of
a/b is -3
a%b is -2

Suppose a=-11 and b=-3
Then result of
a/b is 3
a%b is -2

The equation a = ((a / **b) * b)** + (a % **b) is always true in C.**
RULES

- If **both operands are floating-point types** whose precisions differ (e.g., a float and a double), the *lower precision operand will be converted* to the precision of the other operand, and *the result will be expressed in this higher precision.*
    - Operation between a **float** and a **double** will result in a **double;**
    - a **float** and a **long double** will result in a **long double;**
    - a **double** and a **long double** will result in a **long double.**
- If ***one operand is a floating-point type*** (e.g., float , double or long double)and the *other* is a char or an int (short int or long int), the char or int will be *converted to the floating-point type* and the result will be expressed as such.
    - Operation between an **int** and a **double** will result in a **double.**
- If *neither operand is a floating-point type* but ***one is a long int*** **,** the other will be converted to **long int** and the result will be **long int .**
    - **O**peration between a **long int** and an **int** will result in **along int .**
    - If neither operand is a floating-point type or a **long int ,** then both operands will be converted to **int** (if necessary) and the result will be **i n t .** Thus, an operation between a **short int** and an **int** will result inan **int .**

TYPE CASTING
The value of an expression can be converted to a different data type.
To do so, the expression must be preceded by the name of the desired data type, enclosed in parentheses, i.e.,

(data type) expression

This type of construction is known **as** a ***cast.***

Example i=12.5 can be converted into interger using (int)12.5

Suppose i=7 and is floating point value 8.5

(i+f)%4 is illegal because i+f is not integer

This can be solved using

((int)(i+f)) %4

Similarly ((int)(f)) % 2 also is valid. The result is 1

OPERATOR PRECEDENCE

High precedence	*/ %
Low precedence	+ -

Order in which consecutive operations with same precedence are carried out is called associativity.

Operators are Left to right associative. i.e. they are evaluated from left to right.

E.g. **a - b / c * d**

Suppose floating point numbers a,b,c,d are 1.,2.,3.,4. Respectively

1.- [(2. / **3.) x 4.]** = **1.** - [0.666666. . . **x 4.]** = **1.** - 2.666666 … = -1.666666 …

(a-b)/(c-d)

(1 . - 2 .) / (3 . - 4.) = - l . / 12.=-0.08333333

UNARY OPERATORS

Class of operators that act upon a single operand to produce a new value are known as ***unary operators.***
E.g. unary minus – which is used to negate a number
The ***increment operator,*** ++,
the ***decrement operator,*** --
sizeof operator
cast operator denoted as **(type)** where *type* is the data type to which it is to be converted

Negative number is an expression, consisting of the unary minus operator, followed by a positive numeric constant.

Minus sign can be followed by a numerical operand which may be an integer constant, a floating-point constant, a numeric variable or an arithmetic expression.
Example

```
-743        -0X7FFF     -0.2          -5E-8
-root1      -(x + y)    -3 * (x + y)
```

The increment operator ++ causes its operand to be increased by 1, whereas the decrement operator -- causes its operand to be decreased by 1.
*The **operand** used with each of these operators must be a **single variable**.*

The increment and decrement operators can be written before or after the operand, but meaning will change accordingly.
If the operator precedes the operand(**Pre-increment/Pre decrement)** (e.g., **++i)** then the operand will be altered(incremented by 1) in value ***before*** it is utilized for its intended purpose within the program.
E.g

```
i=1;
a=++i;
```

Here i is set as 1
The value of i is increased by 1, so value of i becomes 2. Then value of i is stored in a.
So after this value of a will be 2. Value of i will be 2
If the operator ***follows*** the operand(**Post-increment/ post decrement)** (e.g., **i++),** then the value of the operand will be altered ***after*** it is utilized.
E.g

```
i=1;
a=i++;
```

Here i is set as 1
Here the value of i is stored in a. Then i increases by 1
So after this value of a will be 1. Value of i will be 2

The **sizeof** this operator allows a determination of the number of bytes allocated to various types of data items
Unary operators have a higher precedence than arithmetic operators
Associativity of the unary operators is ***right to left***

RELATIONAL AND LOGICAL OPERATORS

There are 4

Operator	*Meaning*
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

These operators all fall within the same precedence group, which is *lower than the arithmetic and unary operators*. The associativity of these operators is left to right.

Thera are two equality operators

= = ***equal to***
!= not equal to
The equality operators fall into a separate precedence group*, beneath the relational operators.* These operators also have a left-to-right associativity.
These six operators are used to form ***logical expressions***, which represent conditions that are either **true(1)** or f**alse(0)**. The resulting expressions will be **of** type integer. true 1 .false 0
E.g
m=1;
n=2;
x=2;

m==x result is 0
n==x result is 1

C contains **two *logical operators***(also called *logical connectives)*

Operator	*Meaning*
&&	and
||	or

These operators are referred to as ***logical and***and ***logical or,*** respectively.
They are used to combine logical expressions.

- The result of a *logical* ***and*** operation will be **true** only if **<u>both operands are true</u>**,
- the result of a *logical* ***or***operation will be **true** if **either operand is true** or if **both operands are true**.
    - In other words, the result of a*logical or* operation will be **false** only if <u>both operands are false</u>.

***<u>Any</u>* <u>nonzero value, (not just 1), is interpreted as true.</u>**

Suppose
i=7;
f=5.5;
c='w';
ASCII value of 'w' is 119

Expression	*Interpretation*	*Value*
`(i >= 6) && (c == 'w')`	true	1
`(i >= 6) || (c == 119)`	true	1
`(f < 11) && (i > 100)`	false	0
`(c != 'p') || ((i + f) <= 10)`	true	1

Logical and has a **higher precedence** than *logical or.*

The associativity is left to right.
C also includes the **unary operator ! t**hat negates the value of a logical expression; i.e., it causes an expression that is originally true to become false, and vice versa. This operator is referred to as **the *logical negation* (or *logical not*)**operator

Suppose f=6
!(f>5) is false 0

ASSIGNMENT OPERATORS

There are several different assignment operators in C.
The most commonly used assignment operator is =.

identifier = expression
Here ***identifier*** generally represents a variable, and *expression* represents a constant, a variable or a more complex expression.

=	= =
= is the assignment operator. It is used to assign a value to an identifier, whereas	= = is the equality operator. It is used to determine(check) if two expressions have the same value.
a=2; this store value 2 in variable a	a==2 This compares whether a is equal to 2. If yes result is 1. Otherwise result is 0

Assignment expressions are often referred to as ***assignment statements,*** since they are usually written as
complete statements.
If the *two operands* in an assignment expression are of ***different data types***, then the value of the expression on the right (i.e., the right-hand operand) will automatically be converted to the type of the identifier on the left.
Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example:

- **A** floating-point value may be truncated if assigned to an integer identifier.
  int i;
  i=3.3
  Here value of i will be 3

  i = -3.9;
  value of i will be -3
  i='w';
  Here value of i will be ASCII value of w that is 119

- **A** double-precision value may be rounded if assigned to a floating-point (single-precision) identifier.
- An integer quantity may be altered if assigned to a shorter integer identifier or to a character identifier
  (some high-order bits may be lost).

Multiple assignments of the form
Identifier1 = ***Identifier2= 3***=….= ***expression***
are permissible in C.

The multiple assignment
Identifier1 = ***Identifier2***= ***expression***
is equivalent to
Identifier1 = (***Identifier2=expression)***
and so on, with right-to-left nesting for additional multiple assignment
E.g.
int a,b,c;
a=b=c=2;

There are five additional assignment operators: +=, -= , *=, /= **and** %=.
expression 1 += ***expression 2***
is equivalent to
expression 1 = ***expression 1*** + ***expression 2***

Similarly, the assignment expression
expression 1 -= ***expression 2***
is equivalent to
expression 1 = ***expression I*** - ***expression 2***

Assignment operators have **a lower precedence** than any of the other operators. the assignment operations have a **right to- left associativity.**

THE CONDITIONALOPERATOR ? :

Simple conditional operations can be carried out with the ***conditional operator*** (**?** :).
An expression that makes use of the conditional operator is called a ***conditional expression.***
Such an expression **can** be written in place of the more traditional **if -else** statement.
A conditional expression is written in the form

expression 1 ? ***expression 2*** : ***expression 3***

- Here expression I is evaluated first.
  - If expression 1is true (i.e., if its value is nonzero), then expression 2 is evaluated, and that result becomes the value of the conditional expression.
  - if expression 1is false (i.e., if its value is zero), then expression 3 is evaluated, and that result becomes the value of the conditional expression.

E.g.

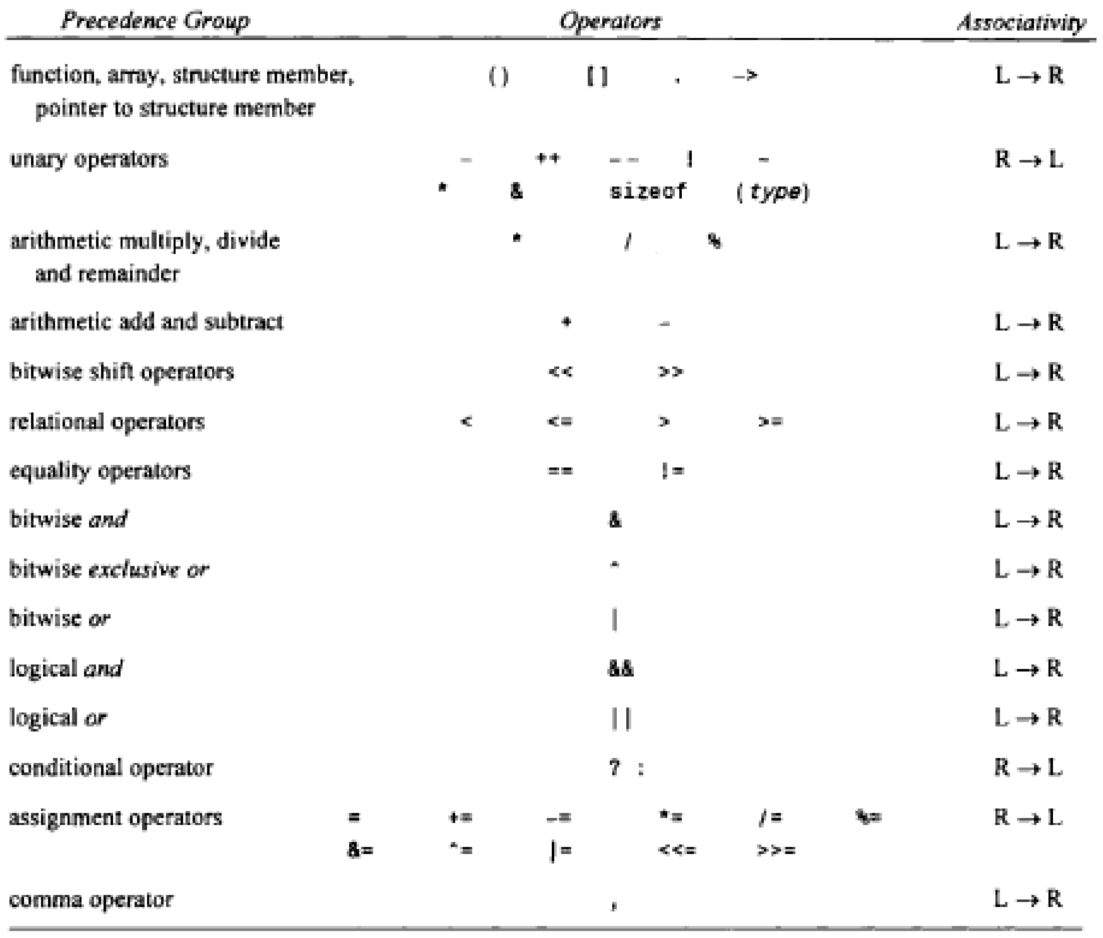
```
int x,y,z;
x=3;
y=5;
z=(x>y)?x:y;
```

Here conditional expressions frequently appear on the right-hand side of a simple assignment statement. The resulting value of the conditional expression is assigned to the identifier on the left.

Here expression 1 is (x>y)
Expression 2 is x
Expression3 is y
If x>y is **true**(1) then z=x will be done.   (expression 2 will be evaluated)
If x>y is **false**(0)then z=y will be done.   (expression 3 will be evaluated)

Here x>y is evaluated. Since 3 is not greater than 5, is not greater than so expression 3 is evaluated.
So z contains value of y(4)

OPERATOR PRECEDENCE TABLE



Precedence Group	Operators	Associativity
function, array, structure member, pointer to structure member	( ) [ ] . ->	L → R
unary operators	- ++ -- ! ~ * & sizeof (*type*)	R → L
arithmetic multiply, divide and remainder	* / %	L → R
arithmetic add and subtract	+ -	L → R
bitwise shift operators	<< >>	L → R
relational operators	< <= > >=	L → R
equality operators	== !=	L → R
bitwise *and*	&	L → R
bitwise *exclusive or*	^	L → R
bitwise *or*	|	L → R
logical *and*	&&	L → R
logical *or*	||	L → R
conditional operator	? :	R → L
assignment operators	= += -= *= /= %= &= ^= |= <<= >>=	R → L
comma operator	,	L → R

DATA INPUT AND OUTPUT

C language has a collection of library functions, which includes a number of input/output functions.

getchar, putchar, scanf, printf , gets and puts.

- **getchar** and **putchar**, help to transfer*single characters*in and out of the computer.

- **scanf** and **printf** permit the transfer of single characters, numericalvalues and strings;
- **gets** and **puts** facilitate the input and output of ***strings***.

The program begins with the preprocessor statement **#include <stdio. h>.** This statement causes the contents ofthe header file **stdio.h** to be included within the program. The header file supplies required information to the libraryfunctions**.**

SINGLE CHARACTER INPUT -THE getchar FUNCTION

Single characters can be entered into the computer using the C library function **getchar.**

The **getchar** function is a part of the standard C I/O library. It returns a single character from a standardinput device (typically a keyboard).

character_variable = **getchar** ( ) ;

where ***character_variable*** refers to some previously declared character variable.

E.g.

```
#include<stdio.h>
main()
{
char c;
c = getchar();
}
```

The first statement declares that **c** is a character-type variable. The second statement causes a single character to be entered from the standard input device (usually a keyboard) and then assigned to **c.**

If an ***end-of-file***condition is encountered when reading a character with the **getchar** function, the value of the symbolic constant **EOF** will automatically be returned.(Usualy EOF is -1)

The **getchar** function can also be used to read multicharacter strings, by reading one character at a time within a multipass loop.

SINGLE CHARACTER OUTPUT -THE putchar FUNCTION

Single characters can be displayed (Le, written out of the computer) using the C library function **putchar.**

This function is opposite to the character input function **getchar.**

The **putchar** function, like **getchar,** is a part of the standard C **I/O** library.

It transmits a single character to a standard output device (typically a **computer** monitor) and displays it.

The function reference is :

putchar (***character_variable)***

where***character_variable*** refers to some previously declared character variable.

E.g.

```
#include<stdio.h>
main()
{
char c;
c='a';
putchar(c);
}
```

The first statement declares that **c** is a character-type variable. The second statement set 'a' as value of character variable c. The third statement causes the current value of **c** to be transmitted to the standard output device (e.g., a computer monitor) and it is displayed.

The **putchar** function can output a string constant by storing the string within a onedimensional,character-type array.

ENTERING INPUT DATA -THE scanf FUNCTION

Input data can be entered into the computer from a standard input device by means of the C library function ***scanf.***

This function can be used to enter any combination of numerical values, single characters and strings.

The function **returns** the number **of** data items that have been entered successfully.

The ***scanf*** function is written **as**

scanf(contro1_string, argl, arg2, . . . , ***argn)***

where ***control_string*** refers to a string containing certain required formatting information.***argl,arg2,*** . . . ***argn*** are arguments that represent the individual input data items.

- The control string consists of individual groups of characters
- one character group for each input dataitem.
- Each character group must begin with a percent sign (%).

- A single character groupwill consist of the percent sign %, followed by a ***conversion character*** which indicates the type of thecorresponding data item.

Within the control string, ***multiple character groups can be contiguous***, or they can be ***separated by whitespace characters*** (i.e., blank spaces, tabs or newline characters).

If whitespace characters are used to separate multiple character groups in the control string, then *all consecutive whitespace characters in the input data will be read but ignored*.

Conversion Character	*Meaning*
c	data item is a single character
d	data item is a decimal integer
e	data item is a floating-point value
f	data item is a floating-point value
g	data item is a floating-point value
h	data item is a short integer
i	data item is a decimal, hexadecimal or octal integer
o	data item is an octal integer
s	data item is a string followed by a whitespace character (the null character \0 will automatically be added at the end)
u	data item is an unsigned decimal integer
x	data item is a hexadecimal integer
[. . .]	data item is a string which may include whitespace characters (

Control string %c is for scanning(inputting) characters. If we want to input integer control string %d is used in scanf function.

E.g

```
#include<stdio.h>
main()
{
int a;
scanf("%d",&a);
}
```

The arguments are written **as** variables or arrays, whose types should match the corresponding character groups in the control string.

- ***Each variable name(*** used as argument in scanf***)must be preceded by an ampersand*** (&). (The arguments are actually pointers that indicate where the data items are stored in the computer's memory)

- Array names (used as argument in scanf) should ***not*** begin with an ampersand.

E.g.

```
#include <stdio.h>
main( )
{
char item(20];
int partno;
float cost;
scanf("%f%d%s",&cost , &partno,item);
}
```

Here item is an array, so in scanf it is not prefixed with &. Cost is floating point variable. Partno is integer variable. Here there is whitespace between control strings %f ,%d and %s

The **scanf** function *can also be written with no whitespace characters in the control string*. This is also valid. E.g.

scanf("%f%d%s", &cost , &partno,item);

This scanf contains three character groups. The first character group,**%f**indicates that the first argument **(&cost)** represents a floating-point value. The second character group, **%d,** indicates that thesecond argument **(&partno)** represents a decimal integer value, and the third character group,**%s,** indicates that the third argument **(item)** represents a string.

The data items to be entered(inputted) must correspond to the arguments in the **scanf** function in number, in type and in order. Numeric data items are written in the sameform as numeric constants. octal values need not be preceded by a ***0,***and hexadecimalvalues need not be preceded by **Ox** or **OX.** Floating-point values must include either a decimal point or anexponent (or both).

If two or more data items are to be entered,

- *they must be separated by whitespace characters.*
- *Or wecan enter items in different lines by pressing enter key. (Enter key is a new line character \n . It is also considered as whitespace).*

If the control string begins by reading a character-type data item,

- it is a good idea to*precede the first conversion character with a blank space*,
    - o because this helps the scanf function to ignore anyextraneous characters that may have been entered earlier.

E.g.

```
#include <stdio.h>
main( )
{

char item[20];
int partno;
float cost;
scanf("  %s %d %f', item, &partno, &cost);
}
```

The following data items could be entered from the standard input device(keyboard) when the program is executed.

fastener 12345 0.05	**Fastener** **12345** **0.05**	**fastener 12345** **0.05**

The blank space that precedes **%s**ignores any previously entered extraneous characters from being assigned to **item.**.

The actual data items are numeric values, single characters or strings, or some combination thereof. They are entered from a standard input device (typicallya keyboard).

s-type conversion character help to acceptcharactersthat *terminate* with a white space character (space, new line etc.). Therefore, a string that ***includes*** whitespace characters cannot be entered by using %s, because it will consider only string up to space. The contents

E.g.

```
#include <stdio.h>
main( )
{
char a[10];
scanf(" %s",a);}
```

If we enter the following input through keyboard:-

Hello how are you

The variable a only stores *Hello. Because when it reads space it considers that string*

To solve this,*(to make scanf accept strings with whitespace characters)* the s-type conversion character within the control string is replaced by a sequence of charactersenclosed in square brackets, denoted**as** [. . .]. Whitespace characters and other allowed characters may be included within thebrackets.

The string will *terminate*, once an input character that <u>does not match any of the characters within the brackets is encountered</u>.**A** null character **(\0)**will then automatically be added to the end of the string.

```
#include <stdio.h>
main( )
{
char line[ 8 0 ] ;
scanf ( "%[ ABCDEFGHIJKLMNOPQRSTUWXYZ]" , line ) ;
}
```

Here inside the square bracket, space and all upper case letters are written. So *when any input other than these is encountered, the string will terminate*.

If the string

NEW YORK CITY

is entered from the standard input device when the program is executed, the entire string will be assigned to the array lin**e**since the string is comprised entirely of uppercase letters and blank spaces.

If the string enters is

New York City

here, then only the single letter **N** would be assigned to **line ,** since the first lowercase letter (in this case, **e)** would beinterpreted **as**the first character beyond the string(because only uppercase letters and space are written inside [].Allothers will be treated as not part of string.i.e. they are delimiters)

If a **Circumflex** (i.e., ^) precede the characters within the square brackets, then when we enter a string through keyboard,<u>characters will be readas long as each input character ***does not*** match one of the characters enclosedwithin the brackets</u>.

i.e The string will *terminate*, once an input character that <u>matches any of the characters within the brackets is encountered</u>.

The first character in the string thatmatches the characters in square brackets will be considered as delimiter then the string is terminated.

If the characters within the brackets are simply the circumflex followed by a newline character, then thestring entered from the standard input device can contain any ASCII characters except the newline character(line feed). Thus, the user may enter whatever he or she wishes and then press the **Enter** key. The **Enter**key will issue the newline character, thus signifying the end of the string.

```
#include <stdio.h>
main( )
{
char x[80] ;
scanf (“%^[  \ n ] ” , x ) ;
}
```

Here circumflex(^) is used with square bracket. Inside the square bracket, \n (newline) is written. So *when input input is read all characters upto newline(enter key) is stored in x. When newline (enter key is pressed)(\n) is encountered, the string will terminate.*

The blank space preceding %[^\n] , ignore any unwanted characters that may have been entered previously.

When the scanf function is executed, a string of undetermined length (but not more than **79**characters) will beentered from the standard input device and assigned to x .

This can accept a line of characters.

If we enter the input :

Hello how are you

Then x contains the string *Hello how are you*

SettingFIELD WIDTH in scanf

The consecutive nonwhitespace characters in a data item arecollectively called a***field***. We can set a limit on field width by specifying a maximumfield width for that data item. An unsigned integer within the control string, between the percent sign (%) andthe conversion character shows the maximum field width of item.

- The data item *may contain fewer characters* than the specified field width.
- Any characters that *extend beyondthe specified field width will not be read.*

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

```
main( )
{
int a,b,c;
scanf ( "%3d %4d %2d" , &a, &b,&c) ;
}
```

Here integer variables***a*** can have maximum 3 field width(digits), b can have maximum 4 digits and can have maximum 2 digits. .

Suppose the input data items are entered as

1 2 3

Then the following assignments will result:

a = 1 , b = 2 , c = 3

If the data had been entered **as**

123 45 789

Then the assignments would be

a = 123, b = 45, c = 78

Now suppose that the data had been entered **as**

123456789

Then the assignments would be

a = 123, b = 4567, c = 89

Suppose that the data had been entered **as**

1234 5678 9

The resulting assignments would now be

a = 123, b = 4, c = 56

E.g.

```
#include <stdio.h>
main ( )
{
int i;
float x;
char c;
```

scanf("%3d %5f %c", &i,**&x, &c);**
}
If the data items are entered **as**

10 256.875 **T**

when the program is executed, then 10 will be assigned to i, **256.8** will be assigned to **x** and the character 7 will beassigned to c.

Usage of single-letter *prefix in scanf*

Most versions of C allow certain conversion characters within the control string to be preceded by asingle-letter ***prefix,*** which indicates the length of the corresponding argument.
For example, **an** 1(lowercase **L)** is used to indicate either a signed or unsigned long integer argument, or a double-precision argument. Similarly, an **h** is used to indicate a signed or unsigned short integer. L is long double.

```
#include <stdio.h>
main ( )
{
short ix,iy;
long lx,ly;
double dx ,dy;
scanf ("hd %ld %lf , &ix, &lx , &dx) ;
scanf("%3ho %71x %151e", &iy, &ly, &dy);
}
```

The control string %hd in the first **scanf** function indicates that the first data item will be assigned to a short decimal integervariable(denoted by single letter prefix h), the second will be assigned to a long decimal integer variable, and the third will be assigned to a doubleprecisionvariable.
The control string in the second **scanf** function indicates that the first data item iy will have a maximumfield width of **3** characters and it will be assigned to a short octal integer variable, the second data item ly will have amaximum field width of **7** characters and it will be assigned to a

long hexadecimal integer variable, and the third data item dywill have a maximum field width of **15** characters and it will be assigned to a double-precision variable.

E.g

```
#include <stdio.h>
main ( )
{
short ix ,iy;
long lx,ly;
double dx,dy;
scanf ("hd %D %f I' , &ix, &lx, &dx) ;
scanf (%3ho %7X %15e", &iy, &1y, &dy);
}
```

Here uppercase conversion characters (in the **scanf** functions) indicate long integers.

Assignment *suppression in scanf function*

It is possible to skip over a data item, without assigning it to the variable or array. To do so, the % sign within the appropriate control group is followed by an asterisk (*). This feature is referred to as assignment ***suppression.***

E.g.

```
#include <stdio.h>
main( )
{
char item[20);
int partno;
float cost;
scanf(" %s %*d %f",item, &partno, &cost);
}
```

Since the asterisk symbol* is present in the second character group corresponding to partno, its value(second value) is not assigned to variable partno.

If the we input values as follows

fastener 12345 0.05

then **fastener** will be assigned to **item** and 0.05 will be assigned to **cost.** But second value**12345** will not be assigned to**partno** because of the asterisk, which is interpreted **as** an assignment suppression character.

Integer quantity **12345** will be read into the computer along with the other data items, even though it isnot assigned to its corresponding variable.

<u>Methods to skip over whitespace characters when we input strings through scanf function</u>

If the control string contains <u>multiple character groups without interspersed whitespace characters</u>, thensome care must be taken with c-type conversion. In such cases a *whitespace character within the input data*

will be interpreted as a data item. To **<u>skip over such whitespace characters</u>** and read the next non-whitespace

character, the conversion group **%1s** should be used. %is will not consider whitespace character, but it will read next non whitespace character.

E.g.

```
include <stdio.h>
main ( )
{
char c l , c2, c3;
scanf(" %c%c%c", &cl, &c2, &c3);
}
```

If the input data consisted of

a b c

(with blank spaces between the letters), then the following assignments would result:

cl = **a, c2** = <blankspace>, **c3** = **b**

Bit if the **scanf** function were written **as**

scanf(" %c%ls%ls", &cl, &c2, &c3)

and if we input as follows:

a b c

then the following are the assignments

cl = a, c2 = b, c3 = c

Methods to skip whitespace in string input:-

- **Use** the conversion group **%1s** in scanf function.

  scanf(" %c%ls%ls", &cl, &c2, &c3)
- We can write the **scanf** function with blank spaces separating the **%c** terms

  scanf (' **%c %c %c", &cl, &c2, &c3);**
- We can write**scanf** function without space

  scanf(" %c%c%c", &cl, &c2, &c3)

  but write the input data **as**consecutive characters without blanks; i.e., **abc**

  Variable c1 will contain value a. c2 contains b and c3 contains c.

Unrecognized characters in control string in scanf function

Unrecognized characters within the control string are expected to be matched by the same characters in the input data. Such input characters will be read into the computer, but not assigned to **an** identifier.Execution of the **scanf** function will terminate if that matching character is not found.

E.g.

```
#include <stdio.h>
main ( )
{
int i;
float x;
scanf("%d a %f", &i, &x);
}
```

Here ‘a’ is unrecognized character in control string.

If the input data consist of

1 a 2.0

then the decimal integer **1** will be read in and assigned to i,the character **a** will be read in but ignored, andthe floating-point value **2.0**will be read in and assigned to **x.**

On the other hand, if the input were entered simply **as**

1 2.0

then the **scanf** function assigns 1 to variable i. But since 'a' is not entered(found) after first input,the execution would stop. Therefore, next variable**x** would automatically represent the value ***0.***

WRITING OUTPUT DATA -THE prntf FUNCTION

Output data can be written from the computer onto a standard output device using the library function

printf function declared inside stdio.h

printf function can be used to output any combination of numerical values, single characters and strings.

The **printf** function moves data from the computer's memory to the standard output device(monitor or output screen), whereas the **scanf** function enters data from the standard input device(keyboard) and stores it in the computer's memory.

printf function is written as

printf(*control string, arg7, arg2, . . . , argn****);***

control string refers to a string that contains formatting information.

arg7, arg2, . . . , argn are arguments that represent the individual output data items.

The arguments can be written **as** constants, single variable or array names, or more complex expressions.

The arguments in a printf function*do **not** represent memory addresses* and therefore are ***not*** preceded by ampersands.

The control string consists of individual groups of characters, with one character group for each output

data item. Each character group must begin with a percent sign (%).

An individual character group will consist of the percent sign, followed by a ***conversion character*** indicating the type of the corresponding data item.

In character group %d, character d is the conversion character which represent integer.

Multiple character groups can be :

- *contiguous, or*
- *they can be separated by other characters, including whitespace characters.*

Conversion Character	Meaning
c	Data item is displayed as a single character
d	Data item is displayed as a signed decimal integer
e	Data item is displayed as a floating-point value with an exponent
f	Data item is displayed as a floating-point value without an exponent
g	Data item is displayed as a floating-point value using either e-type or f-type conversion depending on value. Trailing zeros and trailing decimal point will not be displayed.
i	Data item is displayed as a signed decimal integer
o	Data item is displayed as an octal integer, without a leading zero
s	Data item is displayed as a string
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer, without the leading 0x

```
#include <stdio.h>
#include <math.h>
main()
{
float i= 2.0, j = 3.0;
printf ( " %f %f %f %f", i,j, i + j , sqrt( i + j ) ) ;
}
```

The first two arguments within the printffunction are single variables, the third argument is an arithmeticexpression, and the last argument is a function reference that has a numeric expression **as** an argument.

Output of this program is:

```
2.000000 3.000000 5.000000 2.236068
```

```
#include <stdio.h>
main ( )
{
char item[20];
int partno;
float cost;
printf("Enter the item \n");
scanf("%s",item);
printf("Enter the part number \n");
scanf("%d",&partno);
printf("Enter the cost \n");
scanf("%f",&cost);
printf ( "item=%s part number=%d cost=%f",item, partno, cos t ) ;
}
```

OUTPUT

```
Enter the item
Nut
Enter the part number
101
Enter the cost
3.5
item=Nut part number=101 cost=3.500000
```

E.g.

```
#include <stdio.h>
main ( )
{
int a,b;
float c;
printf("Enter  a b c values");
scanf("%d%d%f",&a,&b,&c);
printf("Result is %d%d%f",a,b,c);
}
```

OUTPUT is:

```
Enter a b c values
12 23 45
Result is 12 23 45
```

The f -type conversion and the e-type conversion are both used to output floating-point values.

However,

the e-type causes an exponent to be included in the output, whereas the former does not.

```
#include <stdio.h>
main() / * display floating-point output 2 d i f f e r e n t ways */
{
double x = 5000.0, y = 0.2;
printf("%f%f%f%f\n\n" , x, y, x*y, x/y);
printf("%e %e %e %e", x, y, x*y, x/y);
}
```

OUTPUT is

```
5000.000000 0.200000 1000.000000 25000.000000
5.000000e+03 2.000000e-01 1.000000e+03 2.500000e+04
```

In the printf function, s-type conversion is used to output a string that is terminated **by** the null character (***0).*** Whitespace characters **may** be included within the string.

Minimum field width in printf function

Minimum field width can be specified *by preceding the conversion character by an unsigned integer.*

- If the number of characters in the corresponding data item is *less than the specified field width*, then the data item will be preceded by enough leading blanks to fill the specified field.
- **If** the number of characters in the data item *exceeds the specified field width,*then additional space will be allocated to the data item,so that the entire data item will be displayed.

  This is just the opposite of the field width indicator in the **scanf**function, which specifies a m***aximum*** field width.

```
#include <stdio.h>
main ( )          /* minimum field width specifications */
.(
int i= 12345;
float x = 345.678;
printf ("%3d %5d %8d\n\n", i, i, i);
printf("%3f %10f %13f\n\n", x , x , x ) ;
printf("%3e %12e %16e", x, x, x ) ;
}
```

Notice the double newline characters in the first two **printf** statements. They will cause the lines of output to be double
spaced, **as** shown below.
When the program is executed, the following output is generated.

```
12345 12345 12345
345.678000 345.678000 345.678000
3.546780e+02 3.546780e+02     3.546780e+02
```

Here i stores number 12345 of field width 5.
The first line of output displays a decimal integer using three different minimum field widths (three characters, fivecharacters and eight characters). The entire integer value is displayed within each field, even if the field width is too small(**as** with the first field in this example).
The second value in the first line is preceded by one blank space. This is generated by the blank space separating thefirst two character groups within the control string.
The third value is preceded by four blank spaces. One blank space comes from the blank space separating the lasttwo character groups within the control field. The other three blank spaces fill the minimum field width, which exceedsthe number of characters in the output value (the minimum field width is eight, but only five characters are there in 12345 so remaining space is 3 blank spaces before 12345).
Field width of **345.678000 is 10.**
Field width of **3.546780e+02 is 12.**
E.g.

```
#include <stdio.h>
main ( ) / * minimum field width specifications */
.(
int i = 12345;
float x = 345.678;
printf("%3d %5d %8d\n\nn, 1, i, 1);
printf("%3g %l0g %13g\n\n",x, x, x ) ;
printf("%3g %13g %16gn, x, x, x ) ;}
```

The conversion character *<u>g suppress trailing zeroes after decimal point.</u>*

Execution of this program causes the fo**llowing output to be displayed.**

```
12345 12345 12345
345.678 345.678345.678
345.678 345.678 345.678
```

Field width of 345.678 is 8.

<u>Specifying precision in printf function</u>

It is also possible to *<u>specify the maximum number of decimal places</u>* for a floating-point value, or the maximum number of characters for a string. This specification is known as ***precision.***

The precision is an unsigned integer that is always preceded by **a** decimal point. **If** a minimum field width is specified in addition to the precision (as is usually the case), then the precision specification follows the field width specification. Both of these integer specifications precede the conversion character.

A floating-point number will be ***rounded*** if **it** must be shortened to conform to a precision specification.

```
#include <stdio.h>
main() /* display a floating-point number with several different precisions */
{
float x = 123.456;
printf("%7f %7.3f %7.lf\n\n", x, x, x ) ;
printf ("%12e %12.5e %12.3e", x, x, x ) ;
}
```

The field width of **123.456 is 7**

The field width of **1.23456e+03**

When this program is executed, the following output is generated.

```
123.456000 123.456 123.5
1.234560e+03  1.23456e+02     1.235e+02
```

Minimum field width specification need not necessarily accompany the precision specification. We can specify the precision without the minimum field width, though the precision must still be precededby a decimal point.

```
#include <stdio.h>
main() /* display a floating-point number with several different precisions */
{
float x = 123.456;

printf ( " %f %.3f % . lf \ n \ n " , x, x, x ) ;
printf ( " %e %.5e %.3e", x , x , x);
}
```

Execution of this program produces the following output.

```
123.456000 123.456 123.5
1.234560e+02 1.23456e+02 1.235e+02
```

In the case of **string**, the minimum field width is interpreted in the same manner as with a numerical quantity; i.e., leading blanks will be added if the string is shorter than the specified field width, and additional space will be allocated if the string is longer than the specified field width. Hence, the field width specification will not prevent the entire string from being displayed.

The precision specification will determine the maximum number of characters that can bedisplayed. **If** the precision specification is less than the total number of characters in the string, the excessright-most characters will not be displayed. This will occur even if the minimum field width is larger than theentire string, resulting in the addition of leading blanks to the truncated string.

```
#include <stdio.h>
main( )
{
char z [ l 2 ] ;
printf("%l0s %15s %15.5s %.5s", z,z,z,z ) ;
}
```

Now suppose that the string **hexadecimal (field width is 11)**is assigned to the character array **z .** When the program is executed, thefollowing output will be generated.

hexadecimal hexadecimal hexad hexad

The first string is shown in its entirety, even though this string consists of 11 characters but the field width specification is
only 10 characters.
The second string is paddedwith four leading blanks to fill out the 15-character minimum; hence, the second string is ***rightjustified*** within its field.
The third string consists of only five nonblank characters because of the five-character precision specification; however,
10 leading blanks are added to fill out the minimum field width specification, which is 15 characters.
The last string alsoconsists of five nonblank characters. Leading blanks are not added, however, because there is no minimum field width specification.

```
#include <stdio.h>
main ( )
{
short a, b;
long c, d;
printf("%5hd %6hx %8lo %lu", a, b, c, d);
}
```

The control string indicates that the first data item will be a short decimal integer, the second will be a short hexadecimal integer, the third will be a long octal integer, and the fourth will be a long unsigned (decimal) integer.

```
#include <stdio.h>
main ( ) / * use of uppercase conversion characters */
{
int a = Ox80ec;
float b = 0.3e-12;
printf("%4x %10.2e\n\n", a, b);
printf("%4X %10.2E", a, b);
}
```

Notice that the first printfstatement contains lowercase conversion characters, whereas the second printfstatement
contains uppercase conversion characters.
When the program is executed, the following output is generated.

```
80ec 3.00e-13

80EC 3.00E-13
```

The first quantity on each line is a hexadecimal number. Note that the letters **ec** (which are a part of the hexadecimalnumber) are shown in lowercase on the first line, and in uppercase on the second line.
The second quantity on each line is a decimal floating-point number which includes an exponent. Notice that theletter **e,** which indicates the exponent, is shown in lowercase on the first line and uppercase on the second.

Usage of flag in printf

In addition to the field width, the precision and the conversion character, each character group within the control string can include a **flag**, which affects the appearance of the output. The flag must be placed immediately after the percent sign (%).

Flag	*Meaning*
–	Data item is left justified within the field (blank spaces required to fill the minimum field width will be added *after* the data item rather than *before* the data item).
+	A sign (either + or –) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
0	Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item. (*Note:* Some compilers consider the zero flag to be a part of the field width specification rather than an actual flag. This assures that the 0 is processed last, if multiple flags are present.)
' ' (*blank space*)	A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.
# (*with* o- *and* x-*type conversion*)	Causes octal and hexadecimal data items to be preceded by 0 and 0x, respectively.
# (*with* e-, f- *and* g-*type conversion*)	Causes a decimal point to be present in all floating-point numbers, even if the data item is a whole number. Also prevents the truncation of trailing zeros in g-type conversion.

```
#include <stdio.h>
main ( ) /* use of flags with integer and floating-point numbers */
{
int i = 123;.
float x = 12.0, y = -3.3;
printf(":%6d %7.0f %lO.le:\n\n", i, x, y);
printf (" :%-6d %-7.0f %-lO.le: \n\n" 1, x , y) ;
printf(":%+6d %+7.0f %+lO.le:\n\n', 1, x, y);
printf(":%-+6d %-+7.0f %-+lO.le:\n\n", 1, x, y);
printf(":%7.0f %#7.0f %7g %#7g:", x, x, y, y);
}
```

When the program is executed, the following output is produced. (The colons indicate the beginning of the first field and the end of the last field in each line.)

```
:   123      12   -3.3e+00:

:123    12      -3.3e+00  :

:  +123     +12   -3.3e+00:

:+123   +12     -3.3e+00  :

:     12     12.    -3.3 -3.30000:
```

The first line illustrates how integer and floating-point numbers appear without any flags. Each number is right justified within its respective field. The second line shows the same numbers, using the same conversions, with a - flag included within each character group. Note that the numbers are now left justified within their respective fields. The third line shows the effect of using a + flag. The numbers are now right justified, **as** in the first line, but each number (whether positive or negative) is preceded by an appropriate sign.

The fourth line shows the effect of combining a - and a + flag. The numbers are now left justified and preceded by an appropriate sign. Finally, the last line shows two floating-point numbers, each displayed first without and then with the # flag. Note that the effect of the flag is to include a decimal point in the number **12.** (which is printed with f-type conversion), and to include the trailing zeros in the number **-3.300000**(printed with g-type conversion).

```
#include <stdio.h>
main() / * use of flags with unsigned decimal, octal and hexadecimal numbers */
{
int i = 1234, j = 01777, k = OxaO8c;
printf(":%8u %80 %8x:\n\n", i, j, k ) ;
printf(":%-8u %-80 %-8x:\n\n", i, j, k ) ;
printf ( " :%#8u %#80 %#8X: \n\n", i, j, k ) ;
printf ( " :%08u %080 %08X: \n\n" , i, j , k) ;
}
```

Execution of this program results in the following output. (The colons indicate the beginning of the first field and the end of the last field in each line.)

```
:    1234     1777     a08c:

:1234     1777     a08c    :

:    1234    01777   0XA08C:

:00001234 00001777 0000A08C:
```

The first line illustrates the display of unsigned integer, octal and hexadecimal output without any flags. The numbers are right justified within their respective fields. The second line shows what happens when you include a – flag within each character group. Now the numbers are left justified within their respective fields.

In the third line we see what happens when the # flag is used. This flag causes the octal number **1777** to be preceded by a 0(appearing as **01 777),** and the hexadecimal number to be preceded by

OX (Le., **OXAO8C).** Notice that the unsigneddecimal integer **1234** is unaffected by this flag. Here hexadecimal number now contains uppercase characters, since the conversion character was written in uppercase **(X).**

The last line illustrates the use of the 0 flag. This flag causes the fields to be filled with leading **OS** rather than leading blanks. We again see uppercase hexadecimal characters, in response to the uppercase conversion character **(X).**

THE gets AND puts FUNCTIONS

C contains a number of other library functions that permit some form of data transfer into or out of the computer.

gets and **puts** functions, facilitate the transfer of strings between the computer and the standard input/output devices.

Each of these functions accepts a single argument. The argument must be a data item that represents a string. (e.g., a character array). The string may include whitespace characters. In the case of **gets,** the string will be entered fiom the keyboard, and will terminate with a newline character (i.e., the string will end when the user presses the **Enter** key).

The **gets** and **puts** functions offer simple alternatives to the use of **scanf** and **p r i n t f** for reading and displaying strings, as illustrated in the following example

```
#include <stdio.h>
main( ) / * read and w r i t e a l i n e o f t e x t * /
{
char x[80] ;
gets(x);
puts(x);
}
```

This program utilizes gets and puts, rather than scanf and printf , to transfer the line of text into and out of the computer.

On the other hand, the scanf and printf functions in the earlier program can be expanded to include additional data items, whereas the present program cannot.

CONTROL STATEMENTS

- A realistic C program needs a logical test be carried out at some particular point within the program.
    - One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as ***branching***.
- There is also a special kind of branching, called ***selection***,
    - in which one group of statements is selected from several available groups
- The program may require that a group of instructions be executed *repeatedly*, until some logical condition has been satisfied. This is known as ***looping***.
- Most **control statements** contain *expression statements or compound statements, including embedded compound statements*
    - An *expression statement* consists of an expression, followed by a semicolon
    - A *compound statement* consists of a sequence of two or more consecutive statements enclosed in braces ({ and }).
        - The enclosed statements can be expression statements, other compound statements or control statements.

CONTROL STATEMENTS

- BRANCHING **if-else** statement
- LOOPING **while** statement, **do-while** statement, **for** statement,
- **switch** statement
- **break** statement
- **continue** statement
- **go to** statement

BRANCHING: THE if -else STATEMENT

- The if - else statement is used to carry out a *logical test*(check whether a condition is true or false)
- and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).
- The *else* portion of the if - else statement is optional.

Thus, in its simplest general form, the statement can be written as

if (expression) statement

- The expression must be placed in parentheses.
- This statement will be executed only if the expression has a nonzero value (i.e., if expression is true).
- If the expression has a value of zero (i.e., if expression is false), then the statement will be ignored. The statement can be either simple or compound.

The general form of an if statement which includes the else clause is

if (expression) statement1 else statement2

If the expression has a nonzero value (i.e., if expression is true), then statement1 will be executed. Otherwise (i.e., if expression is false), statement2will be executed.

It is possible to nest (i.e., embed) if - else statements, one within another. There are several different forms that nested if -else statements can take. The most general form of two-layer nesting is

```
if e7 if e2 s7

          else s2

else if e3 s3

          else s4
```

where e I, e2 and e3 represent logical expressions and s 7, s2,s3and s4 represent statements. Now, one complete if - else statement will be executed if e7 is nonzero (true), and another complete if - else statement will be executed if e 7 is zero (false). It is, of course, possible that s7, s2,s3and s4 will contain other if-else statements.

LOOPING: THE while STATEMENT

The while statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.

The general form of the while statement is

while (expression) statement

The statement inside while loop will be executed repeatedly, as long as the expression inside while is true (ie., as long expression has a nonzero value).

This statement can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition for the loop.

Suppose we want to display the consecutive digits 0, **1, 2,** . . . ,**9,** with one digit on each line. This can be done using   the following program.

```
#include <stdio.h>
main( )
 {
int x= 0;
while ( x<= 9) {
```

```
printf ( "%d\n", x ) ;
x++;

}

}
```

while statement is also called ENTRY CONTROLLED loop, because statements inside while loop is executed only if the condition inside the while loop is true. So control can enter while loop only if condition is satisfied.

while(1);

This will lead to infinite loop. 1 means true.

Suppose the program is executed

```
#include <stdio.h>
main( )
 {
int x= 0;
while ( x<= 9);
{
        printf ( "%d\n", x ) ;
        x++;

}

}
```

The output of above code is infinite loop(because 0<=9 and after while loop ; is written so nothing is done and execute infinitely)

while loops are generally used when the number of passes is ***not*** known in advance.

THE do - while STATEMENT

The general form of the **do - while** statement is
do *statement* while *(expression)*;

```
do

{

Statement;

Statement;
```

…….

}while(condition);

do-while statement is also called EXIT CONTROLLED loop, because statements inside while loop is executed first without checking condition initially. When control reaches the end of loop, condition is checked. At the end of each iteration only, the condition is checked. If condition is true, then statement in the loop is executed again. Otherwise the loop terminates and control out of the loop to the statement after the loop.

Suppose we want to display the consecutive digits 0, **1, 2,** . . . ,**9,** with one digit on each line. This can be done using the following program.

```
#include <stdio.h>
main( )
 {
int x= 0;
        do{
        printf ( "%d\n", x ) ;
        x++;

        } while ( x<= 9);

}
```

The following code also prints 0 to 9 in each line

```
#include <stdio.h>
main( )
 {
        int x= 0;
        do
        printf ( "%d\n", x++) ;
        while ( x<= 9);
}
```

The following programs will result in infinite loop:-

```
    (1) #include <stdio.h>
main( )
 {
        int x= 0;

        do{
        printf ( "%d\n", x ) ;
        x++;

        } while ( x>=0);
```

}

This will print numbers from 0 to infinite numbers in each line.

```
     (2) #include <stdio.h>
main( )
 {
int x= 0;

do{
printf ( "%d\n", x ) ;
} while ( x>=0);}
```

This will print 0 infinite number of times in each line

for STATEMENT

The general form of the **for** statement is
for (***expression* 1; *expression*** 2; ***expression 3) statements;***

for (***expression* 1; *expression*** 2; ***expression 3)***
{
statement;
Statement;
}

- where ***expression 1*** is used to **initialize** some parameter (called an ***index)*** that controls the looping action,
- ***expression2*** represents a **condition** that must be true for the loop to continue execution, and
- ***expression***3 is used to **alter the value of the paramete**r initially assigned by ***expression***

Typically, ***expression 1*** is an assignment expression, ***expression*** 2 is a logical expression and ***expression*3** is a unary expression or an assignment expression(value modification) e.g. i++ , i=i+2.

for statement is equivalent to

expression 1;
while (expression 2) {
statement;
expression 3;
}

The looping action will continue **as** long **as** the value of ***expression 2*** is not zero(true).

- **While** loops are generally used when the number of passes is ***not*** known in advance

- **for** loops are generally used when the number of passes ***is*** known in advance.

E.g.Suppose we want to display the consecutive digits 0, **1, 2,** . . . ,**9,** with one digit on each line.

```
#include <stdio.h>
main( )
 {
        int x= 0;
        for(x=0;x<=9;x++
        {
        printf ( "%d\n", x ) ;
        }

}
```

The following for statement is same as while loop:

```
for( ; x<=9; )

{

        printf ( "%d\n", x ) ;
        x++;
}
```

The following statements lead to infinite loop

```
for(;;;) ;

for(x=0; ;x++) ;

for( ; ;x++) ;
```

Working of for loop

E.g.

```
        #include <stdio.h>
        main( )
        {
        int x;

        for(x=0;x<=2;x++)
        {
        printf ( "%d\n", x ) ;
        }
        printf("stop"); }
```

- Here x is initialized to 0

- Then check x <=0. Here 0<=2
    - So statements inside for loop is executed.
    - **0** is printed in one line
- Then x increase by 1(x++)
    - x becomes 1
- Then check x <=0. Here 1<=2
    - So statements inside for loop is executed.
    - **1** is printed in next line
- Then x increase by 1(x++)
    - x becomes 2
- Then check x <=0. Here 2<=2
    - So statements inside for loop is executed.
    - **2** is printed in next line
- Then x increase by 1(x++)
    - x becomes 3
- Then check x <=0. Here 3 not <=2 So control go out of for loop
- ***stop*** is printed in next line

switch statement

If there are manyalternative conditions, it is complex if we use- if else if statement. Switch statement is most suitable for evaluating many number of alternative conditions.

The**switch** statement is a multiway decision making statement.

The **switch**statement helps to choose group of statements from several available groups of statements.

General form of switch statements is

switch(expression) statement

statement is a group of case statements. The switch statement is written as

switch(expression)

{

case firstvalue:

        statements;

        statements;

case secondvalue:

statements;

statements;

….

default:

statements;

statements;

}

The firstvalue, secondvalue etc along with case must be integer or character(character must be enclosed within single quotes)

- The expression inside switch can be a variable or any other expression, whose value is an integer constant or character constant.
- The resulting value of expression is then checked with values in case in order.
- If any of the values in case is same as the resulting value of expression, then statements inside that case statement is executed.
- If **case** statements does not end with break statement then remaining case values are also checked.
- If case statement ends with break then only statements inside that matching case statements(if any) are executed. Remaining case statements are not checked.
- If value of expression does not match any of the case values then if a **default** statement is present then the statements inside default statement is executed.
    - The **default** group may appear anywherewithin the **switch** statement-it need not necessarily be placed at the end.
    -

Eg. Print words of numbers from 1 to 5

```
#include <stdio.h>
main( )
          {
int x;

printf(“Enter a  number from 1 to 5”);

scanf(“%d”,&x);

switch(x)

{
```

```
case 1:
        printf(“one”);
        break;
case 2:
        printf(“Two”);
        break;
case 3:
        printf(“Three”);
        break;
case 4:
        printf(“Four”);
        break;
case 5:
        printf(“Five”);
        break;
}}
```