UNIT-II:

Introduction to C Programming- Identifiers, The main () Function, The printf () Function

Programming Style - Indentation, Comments, Data Types, Arithmetic Operations, Expression Types, Variables and Declarations, Negation, Operator Precedence and Associativity, Declaration Statements, Initialization. Assignment - Implicit Type Conversions, Explicit Type Conversions (Casts), Assignment Variations, Mathematical Library Functions, Interactive Input, Formatted Output, Format Modifiers.

Introduction to C Programming:

✓ Identifiers

Identifiers are names for entities in a C program, such as variables, arrays, functions, structures, unions and labels. An identifier can be composed only of uppercase, lowercase letters, underscore and digits, but should start only with an alphabet or an underscore.

An identifier is a string of alphanumeric characters that begins with an alphabetic character or an underscore character. Actually, an identifier is a user-defined word. There are 53 characters, to represent identifiers. They are 52 alphabetic characters (i.e., both uppercase and lowercase alphabets) and the underscore character. The underscore character is considered as a letter in identifiers. The underscore character is usually used in the middle of an identifier. There are 63 alphanumeric characters, i.e., 53 alphabetic characters and 10 digits (i.e., 0-9).

Rules for constructing identifiers

- The first character in an identifier must be an alphabet or an underscore and can be followed only by any number alphabets, or digits or underscores.
- They must not begin with a digit.
- Uppercase and lowercase letters are distinct. That is, identifiers are case sensitive.
- Commas or blank spaces are not allowed within an identifier.
- Keywords cannot be used as an identifier.
- Identifiers should not be of length more than 31 characters.
- Identifiers must be meaningful, short, quickly and easily typed and easily read.

Valid identifiers: total sum average _x y_ mark_1 x1
Invalid identifiers: 1x - begins with a digit char - reserved word
x+y - special character

Note: Underscore character is usually used as a link between two words in long identifiers.

Differentiate between Keywords words and identifiers

Keyword	Identifier
Predefined-word	User-defined word
Must be written in lowercase only	Can written in lowercase and uppercase
Has fixed meaning	Must be meaningful in the program
Whose meaning has already been explained to the C compiler	Whose meaning not explained to the C compiler
Combination of alphabetic characters	Combination of alphanumeric characters
Used only for it intended purpose	Used for required purpose
Underscore character is not considered as a letter	Underscore character is considered letter

✓ main() function

main() function is the entry point of any C program. It is the point at which execution of program is started. When a C program is executed, the execution control goes directly to the main() function. Every C program have a main() function.

Syntax:

```
void main()
{
 .....
 .....
}
```

In above syntax;

- **void:** is a keyword in C language, void means nothing, whenever we use void as a function return type then that function nothing return. here main() function no return any value.
- In place of void we can also use **int** return type of main() function, at that time main() return integer type value.
- **main:** is a name of function which is predefined function in C library.

Simple example of main()

Example

```
#include<stdio.h>
void main()
{
printf("This is main function");
}
```

Output: This is main function

✓ Printf()

Printf is a predefined function in "stdio.h" header file, by using this function, we can print the data or user defined message on console or monitor. While working with printf(), it can take any number of arguments but first argument must be within the double cotes (" ") and every argument should separated with comma (,) Within the double cotes, whatever we pass, it prints same, if any format specifies are there, then that copy the type of value. The scientific name of the monitor is called console.

Syntax:

printf("user defined message");

prinf("Format specifers",value1,value2,..);

Example of printf function

```
int a=10;
double d=13.4;
printf("%f%d",d,a);
```

scanf()

scanf() is a predefined function in "stdio.h" header file. It can be used to read the input value from the keyword.

Syntax

scanf("format specifiers",&value1,&value2,.....);

Example of scanf function

```
int a;
float b;
scanf("%d%f",&a,&b);
```

In the above syntax format specifier is a special character in the C language used to specify the data type of value.

Format specifier:

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%lf	Long double

The address list represents the address of variables in which the value will be stored.

Example:

```
int a;
float b;
scanf("%d%f",&a,&b);
```

In the above example scanf() is able to read two input values (both int and float value) and those are stored in a and b variable respectively.

Syntax:

```
double d=17.8;
char c;
long int l;
scanf("%c%lf%ld",&c&d&l);
```

Clrscr() and Getch() in C

clrscr() and getch() both are predefined function in "conio.h" (console input output header file).

Clrscr()

It is a predefined function in "conio.h" (console input output header file) used to clear the console screen. It is a predefined function, by using this function we can clear the data from console (Monitor). Using of clrscr() is always optional but it should be place after variable or function declaration only.

Example of clrscr()

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10, b=20;
int sum=0;
clrscr();  // use clrscr() after variable declaration
sum=a+b;
printf("Sum: %d",s);
getch();
}
```

Output: Sum: 30

Getch()

It is a predefined function in "conio.h" (console input output header file) will tell to the console wait for some time until a key is hit given after running of program.

By using this function we can read a character directly from the keyboard. Generally getch() are placing at end of the program after printing the output on screen.

Example of getch()

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

void main()
{
int a=10, b=20;
int sum=0;
clrscr();
sum=a+b;
printf("Sum: %d",s);
getch();  // use getch() befor end of main()
}
```

Output: Sum: 30

Programming Style:

Programming style is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

- ✓ **Indentation**:

In computer programming languages, indentation is used to format program (source code) to improve readability.

Example:

So we can see that Indentation is nothing but formatting the source of our program (by the help of tab/ spaces) in such a way that it increases readability.

Sample Code:

```
main() {
variable declaration;
for loop {
another for loop {
yet another for loop {
some work to be done;
another work;
```

```
}
again some work;
} }
```

Now lets have look at his indented

Sample Code with Indentation:

```
main() {
        variable declaration;
        for loop {
                another for loop {
                        yet another for loop {
                                some work to be done;
                                another work;
                        }
                again some work;
                }
        damn some more work;
        }
}
```

When we look at this code, we can clearly see that inside our main(), we have a nested for loop. It certainly is more readable, and understandable.

✓ Comments:

A **comment** is a note to yourself (or others) that you put into your source code. All comments are ignored by the compiler. They exist solely for your benefit. Comments are used primarily to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it. You can also use a comment to temporarily remove a line of code. Simply surround the line(s) with the comment symbols.

In C, the start of a comment is signaled by the **/*** character pair. A comment is ended by ***/**. For example, this is a syntactically correct C comment:

/* This is a comment. */

Comments can extend over several lines and can go anywhere except in the middle of any C keyword, function name or variable name. In C you can't have one comment within another comment. That is comments may not be nested. Lets now look at our first program one last time but this time with comments:

```
main() /* main function heading */
{
printf("\n Hello, World! \n"); /* Display message on */
} /* the screen */
```

Commenting

Commenting involves placing **Human Readable Descriptions** inside of computer programs detailing what the **Code** is doing. Proper use of commenting can make code maintenance much easier, as well as helping make finding bugs faster. Further, commenting is very important when writing functions that other people will use. Remember, well documented code is as important as correctly working code.

Where to Comment:

Comments should occur in the following places:

1. **The top of any program file.**

This is called the **"Header Comment"**. It should include all the defining information about who wrote the code, and why, and when, and what it should do. (See Header Comment below)

2. **Above every function**.

This is called the function header and provides information about the purpose of this "sub-component" of the program.

When and if there is only one function in a file, the function header and file header comments should be merged into a single comment. (See Function Header below)

3. **In line**

Any "tricky" code where it is not immediately obvious what you are trying to accomplish, should have comments right above it or on the same line with it.

✓ Data types:

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we use in our program. These data types have different storage capacities.

- C data types are defined as the data storage format that a variable can store a data to perform a specific operation.
- Data types are used to define a variable before to use in a program.
- Size of variable, constant and array are determined by data types.

There are four data types in C language. They are,

S.no	Types	Data Types
1	Basic data types	int, char, float, double
2	Enumeration data type	enum
3	Derived data type	pointer, array, structure, union
4	Void data type	void

BASIC DATA TYPES IN C:

✓ INTEGER DATA TYPE:

- Integer data type allows a variable to store numeric values.
- "int" keyword is used to refer integer data type.
- The storage size of int data type is 2 or 4 or 8 byte.
- int (2 byte) can store values from -32,768 to +32,767
- int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.

Note:

- We can't store decimal values using int data type.
- If we use int data type to store decimal values, decimal values will be truncated and we will get only whole number.
- In this case, float data type can be used to store decimal values in a variable.

✓ CHARACTER DATA TYPE:

- Character data type allows a variable to store only one character.
- Storage size of character data type is 1. We can store only one character using character data type.
- "char" keyword is used to refer character data type.
- 'A' can be stored using char datatype. You can't store more than one character using char data type.

✓ FLOATING POINT DATA TYPE:

Floating point data type consists of 2 types. They are,

1. Float 2. double

FLOAT:

- Float data type allows a variable to store decimal values.
- Storage size of float data type is 4.
- We can use up-to 6 digits after decimal using float data type.
- For example, 10.456789 can be stored in a variable using float data type.

DOUBLE:

- Double data type is also same as float data type which allows up-to 10 digits after decimal.
- The range for double datatype is from 1E–37 to 1E+37.

SIZEOF() FUNCTION IN C:

sizeof() function is used to find the memory space allocated for each C data types.

✓ MODIFIERS IN C:

- The amount of memory space to be allocated for a variable is derived by modifiers.
- Modifiers are prefixed with basic data types to modify (either increase or decrease) the amount of storage space allocated to a variable.
- There are 5 modifiers available in C language. They are,

1. Short 2. long 3. signed 4 .unsigned 5.long long

Below table gives the detail about the storage size of each C basic data type in 16 bit processor. Please keep in mind that storage size and range for int and float datatype will vary depend on the CPU processor (8,16, 32 and 64 bit).

S.No	C Data types	storage	Range
1	char	1	–127 to 127
2	int	2	–32,767 to 32,767
3	float	4	1E–37 to 1E+37 with six digits of precision
4	double	8	1E–37 to 1E+37 with ten digits of precision
5	long double	10	1E–37 to 1E+37 with ten digits of precision
6	long int	4	–2,147,483,647 to 2,147,483,647
7	short int	2	–32,767 to 32,767
8	unsigned short int	2	0 to 65,535
9	signed short int	2	–32,767 to 32,767
10	long long int	8	–(2power(63) –1) to 2(power)63 –1
11	signed long int	4	–2,147,483,647 to 2,147,483,647
12	unsigned long int	4	0 to 4,294,967,295
13	unsigned long long int	8	2(power)64 –1

2. ENUMERATION DATA TYPE IN C:

- Enumeration data type consists of named integer constants as a list.
- It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.
- Enum syntax in C:

  enum identifier [optional{ enumerator-list }];
- Enum example in C:

```
enum month { Jan, Feb, Mar }; or
/* Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default */
enum month { Jan = 1, Feb, Mar };
/* Feb and Mar variables will be assigned to 2 and 3 respectively by default */
enum month { Jan = 20, Feb, Mar };
/* Jan is assigned to 20. Feb and Mar variables will be assigned to 21 and 22 respectively by default */
```

3. DERIVED DATA TYPE IN C:

- Array, pointer, structure and union are called derived data type in C language.

4. VOID DATA TYPE IN C:

- Void is an empty data type that has no value.
- This can be used in functions and pointers.

✓ Expression Types:

C Programming Expression:

1. In programming, an expression is any legal combination of symbols that represents a value.
2. C Programming provides its own rules of Expression, whether it is legal expression or illegal expression. For example, in the C language x+5 is a legal expression.
3. Every expression consists of at least one operand and can have one or more operators.
4. Operands are values and Operators are symbols that represent particular actions.

Valid C Programming Expression:

C Programming code gets compiled firstly before execution. In the different phases of compiler, c programming expression is checked for its validity.

Expressions	Validity
a + b	Expression is valid since it contain + operator which is binary operator
+ + a + b	Invalid Expression

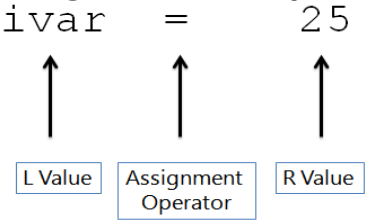
L-Value of Expression:

1. L-Value stands for **left value**
2. L-Value of Expressions refer to a memory locations
3. In any assignment statement L-Value of Expression must be a container(i.e. must have ability to hold the data)
4. Variable is the only container in C programming thus L Value must be any Variable.
5. L Value Cannot be Constant, Function or any of the available data type in C

Example of L-Value of Expression :

```
#include<stdio.h>
int main()
{
  int num;
  num = 5;
return(0);
}
```

Diagram Showing L-Value of Expression :



In the above expression, Constant value 5 is being assigned to a variable 'num'. Variable 'num' is called as storage region's , 'num' can considered as LValue of an expression.

R-Value of Expression:

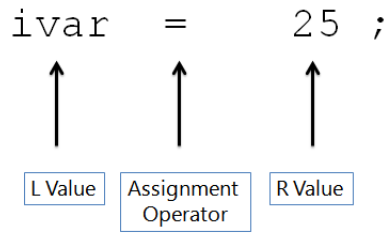
1. R Value stands for **Right value** of the expression.
2. In any **Assignment statement** R-Value of Expression must be anything which is capable of returning Constant Expression or Constant Value.
3. R Value Can be anything of following

Examples of R-Value of Expression	
Variable	Constant
Function	Macro
Enum Constant	Any other data type

Example of R-Value of Expression

```
#include<stdio.h>
int main()
{
int num;
num = 5;
return(0);
}
```

Diagram Showing Lvalue :



Types of Expression:

In Programming, different verities of expressions are given to the compiler. Expressions can be classified on the basis of Position of Operators in an expression.

Type	Explanation	Example
Infix	Expression in which Operator is in between Operands	a + b
Prefix	Expression in which Operator is written before Operands	+ a b
Postfix	Expression in which Operator is written after Operands	a b +

Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to A B C + * D / .

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication. Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C".

We can add (totally unnecessary) brackets to make this explicit: ((A (B C +) *) D /)

Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to / * A + B C D

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear: (/ (* A (+ B C)) D)

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in.

In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication). Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - Bdoes not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).

Examples:

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

✓ Negation: C 1's Compliment Operator

One's Compliment Operator in C

1. It is denoted by ~
2. Bit Pattern of the data can be Reversed using One's Compliment
3. It inverts each bit of operand .
4. One's Compliment is Unary Operand i.e Operates on 1 Argument

Original Number A	0000 0000 0011 1100
One'sCompliment	1111 1111 1100 0011
Zero's Are Changed to	1
One's Are Changed to	0

Syntax :

~Variable_Name

Example : Negation Operator in C Programming

```
#include<stdio.h>
int main()
{
int a=10;
printf("\nNegation of Number 1 : %d",~a);
return(0);
}
```

Output :

Negation of Number 1 : -11

✓ Operator precedence & associativity

C Programming supports wide range of operators. While Solving the Expression we must follow some rules. While solving the expression [a + b *c] , we should first perform Multiplication Operation and then Addition, similarly in order to solve such complicated expression you should have hands on Operator Precedence and Associativity of Operators.

Operator precedence & associativity table

Operator precedence & associativity are listed in the following table and this table is summarized in decreasing Order of priority i.e topmost operator has highest priority and bottommost operator has Lowest Priority.

Operator	Description	Associativity
( ) [ ] . -> ++ –	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ – + – ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right

^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Summary of operator precedence:

1. **Comma** Operator Has **Lowest Precedence**.
2. **Unary Operators** are Operators having **Highest Precedence**.
3. **Sizeof** is Operator not Function.
4. Operators sharing Common Block in the Above Table have Equal Priority or Precedence.
5. While Solving Expression, Equal Priority Operators are handled on the basis of FIFO[First in First Out] i.e Operator Coming First is handled First.

Assignment:

✓ Type Conversion

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called **type conversion.**

Type conversion in c can be classified into the following two types:

1) Implicit 2) Explicit

Implicit Type Conversion:

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as **implicit type conversion** or **type promotion**.

The compiler converts all operands into the data type of the largest operand.
The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

1. If either of the operand is of type long double, then others will be converted to long double and result will be long double.
2. Else, if either of the operand is double, then others are converted to double.
3. Else, if either of the operand is float, then others are converted to float.
4. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
5. Else, if either operand is long int then other will be converted to long int.
6. Else, if either operand is unsigned int then others will be converted to unsigned int.

Note:
It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Explicit Type Conversion:

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.The explicit type conversion is also known as **type casting**.

Type casting in c is done in the following form:

(data_type)expression;

where, *data_type* is any valid c data type, and *expression* may be constant, variable or expression.

For example,

```
x=(int)a+b*d;
```

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

1. All integer types to be converted to float.
2. All float types to be converted to double.
3. All character types to be converted to integer.

✓ Assignment variations;

In a program statement if we use a shorthand notation to perform an operation and an assignment at the same time then it is called as Assignment variation. C supports assignment operator with this shorthand or mixed notations.

Operator Name	Syntax	Meaning
Addition assignment	**a+=b**	**a=a+b**
Subtraction assignment	**a-=b**	**a=a-b**
Multiplication assignment	**a*=b**	**a=a*b**
Division assignment	**a/=b**	**a=a/b**

Modulo assignment	**a%=b**	**a=a%b**
Bitwise AND assignment	**a&=b**	**a=a&b**
Bitwise OR assignment	**a|=b**	**a=a|b**
Bitwise XOR assignment	**a ^=b**	**a=a ^b**
Bitwise left shift assignment	**a<<=b**	**a=a<<b**
Bitwise right shift assignment	**a>>=b**	**a=a>>b**

✓ Mathematical Library Functions:

In C programming language all mathematical operations are grouped as functions in the standard library.

Function	#include	What It Does
sqrt()	math.h	Calculates the square root of a floating-point value
pow()	math.h	Returns the result of a floating-point value raised to a certain power
abs()	stdlib.h	Returns the absolute value (positive value) of an integer
floor()	math.h	Rounds up a floating-point value to the next whole number
ceil()	math.h	Rounds down a floating-point value to the next whole number

All the functions listed, save for the abs() function, deal with floating-point values. The abs() function works only with integers.

math.h library functions

All C inbuilt functions which are declared in math.h header file are given below.

LIST OF INBUILT C FUNCTIONS IN MATH.H FILE:

- "math.h" header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.
- Click on each function name below for detail description and example programs.

S.no	Function	Description
1	**floor ( )**	This function returns the nearest integer which is less than or equal to the argument passed to this function.
2	**round ( )**	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer
3	**ceil ( )**	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
4	**sin ( )**	This function is used to calculate sine value.
5	**cos ( )**	This function is used to calculate cosine.
6	**cosh ( )**	This function is used to calculate hyperbolic cosine.
7	**exp ( )**	This function is used to calculate the exponential "e" to the x^{th} power.
8	**tan ( )**	This function is used to calculate tangent.
9	**tanh ( )**	This function is used to calculate hyperbolic tangent.
10	**sinh ( )**	This function is used to calculate hyperbolic sine.
11	**log ( )**	This function is used to calculates natural logarithm.
12	**log10 ( )**	This function is used to calculates base 10 logarithm.
13	**sqrt ( )**	This function is used to find square root of the argument passed to
14	**pow ( )**	This is used to find the power of the given number.
15	**trunc.(.)**	This function truncates the decimal value from floating point value

✓ Interactive Input:

A standard task in C programming is to get interactive input from the user; that is, to read in a number or a string typed at the keyboard. One method is to use the function scanf.

For example to get an integer,
we might use (#include <stdio.h> is implied):
int answer; /* an integer chosen by the user */
scanf ("%d", &answer);

This works fine if the user actually types a number, but is problematic if they make a mistake and type a letter instead. One can do better by checking the return value of scanf, which is zero if no conversions were assigned, but it is difficult to recover and get corrected input.

The C '**scanf**' method is a function located in a 'stdio' library. It is used to read a formatted input including a character, string, and numeric data from Standard Input (stdin), which is generally a keyboard. Though scanf is an extremely useful function in simpler programs, it does not effectively handle human input errors. This makes the C 'scanf' function a little unreliable and hence should only be used to implement simple programs where reliability is not a priority.

The 'format' argument of C 'scanf' function is a C character string, consisting of one or more of the following components:

- Format specifiers.
- Whitespace characters.
- Non-whitespace characters.

On successful execution of the C 'scanf' function, the total number of characters read from the input is returned. If it fails, a negative value or an 'EOF' is returned by the function.

C Scanf Function Format Specifiers

% [*] [width] [length_Modifiers] type_Specifier

The C 'scanf' function format is a combination of the parameters mentioned above. Every specifier is preceded by a percentage (%) sign. A brief description of the aforementioned format specifiers is given below:

- A '*' sign is an optional specifier, which specifies that user input is read from 'stdin' but is ignored and not saved.
- 'Width' is an optional specifier that indicates the maximum number from total input characters that can be read from 'stdin'.
- The 'length_Modifiers' propery is another optional item and is used to indicate the exact type for the user input such as long 'l' for which no explicit conversion letter exists.
- The 'type_Specifier' property is a conversion letter used to indicate the actual type of data that is read from 'stdin'. User input is converted into the provided type of data and assigned to the corresponding variable.

✓ Formatted Output

One of the most important and common tasks in every program is the printing of output. Programs use output to request input from a user, to display status messages, and to inform the user of the results of computations that the program performs.

For obvious reasons, the manner in which the program displays its output can have a deep effect on the usefulness and usability of a program. When a program prints its output in a neatly formatted fashion, the output is always easier to read and understand. As a result, being able to write programs easily that produce attractive output is an essential feature of most programming languages.

C has a family of library functions that provide this capability. All of these functions reside in the Stdio(**St**an**d**ard **i**nput **o**utput) library.

Although the library contains several functions for printing formatted output, it is likely that you will only use two of them with any frequency. One, the **printf** (short for "**print formatted**") function, writes output to the computer monitor.

Using printf to print messages To use the printf function, you must first insure that you have included the stdio library in your program. You do this by placing the C preprocessor directive
#include <stdio.h> at the beginning of your program.
***function_name* (*argument1*, *argument2*, ...);**
The "..." signifies that there may be more arguments. In fact, there may also be fewer arguments.

Example: Printf("sum of two numbers is, %d",a+b);

printf ("Please type an integer then press Enter: ");
When the computer executes this statement, the message will appear on the screen:
Please type an integer then press Enter

C uses *escape sequences* within a format string to indicate when we want printf to print certain special characters, such as the character that the Enter key produces. The escape character for a *newline* (which sends the cursor to the beginning of the next line on the screen) is \n. The backslash is called the *escape character* in this context and it indicates that the programmer wants to insert a special character into the format string. Without the backslash, printf would simply print the 'n'. You might guess that the 'n' is an abbreviation for "newline." C provides several escape sequences, but only a few are common.

Escape sequence	Action
\n	prints a newline
\b	prints a backspace (backs up one character)
\t	prints a tab character
\\	prints a backslash
\"	prints a double quote

If we alter the second example above as follows:

printf ("Searching, please wait...\n");

the screen will appear as before, except that now the cursor will be on the next line. Furthermore, if the program contains another printf statement later on, the next output will be printed on that same next line.

Searching, please wait...

|

✓ Format Modifiers:

A format specification can also include "modifiers" that can control how much of the item's value is printed and how much space it gets. The modifiers come between the `%' and the format-control letter. Here are the possible modifiers, in the order in which they may appear:

`-' The minus sign, used before the width modifier, says to left-justify the argument within its specified width. Normally the argument is printed right-justified in the specified width.

Thus, printf "%-4s", "foo"

Result: prints `foo '.

`**WIDTH'** This is a number representing the desired width of a field. Inserting any number between the `%' sign and the format control character forces the field to be expanded to this width.

The default way to do this is to pad with spaces on the left. For example, **printf "%4s", "foo"** prints ` foo'.

The value of WIDTH is a minimum width, not a maximum. If the item value requires more than WIDTH characters, it can be as wide as necessary. Thus, **printf "%4s", "foobar"** prints `foobar'.

Preceding the WIDTH with a minus sign causes the output to be padded with spaces on the right, instead of on the left.

`.PREC' This is a number that specifies the precision to use when printing. This specifies the number of digits you want printed to the right of the decimal point. For a string, it specifies the maximum number of characters from the string that should be printed.

The C library `printf"s dynamic WIDTH and PREC capability (for example, `"%*.*s"') is supported. Instead of supplying explicit WIDTH and/or PREC values in the format string, you pass them in the argument list. For example:

```
w = 5
p = 3
s = "abcdefg"
printf "<%*.*s>\n", w, p, s is exactly equivalent to

s = "abcdefg"
printf "<%5.3s>\n", s
```

Both programs output `<**abc>'. (We have used the bullet symbol "*" to represent a space, to clearly show you that there are two spaces in the output.)

Earlier versions of `awk' did not support this capability. You may simulate it by using concatenation to build up the format string, like so:

```
w = 5
p = 3
s = "abcdefg"
printf "<%" w "." p "s>\n", s
```

This is not particularly easy to read.
There are several format specifiers (Modifiers) - the one you use should depend on the type of the variable you wish to print out. Here are the common ones:

Format Specifier	Type
%d (or %i)	int
%c	char
%f	float
%lf	double
%s	string
%x	hexadecimal

- ✓ To display a number in scientific notation, use %e.
- ✓ To display a percent sign, use %%.
- ✓ %d is essentially the same as %i but I used %d from the very first day.
- ✓ LF stands for "long float".