C-02 Programming with C – Input and Output

2.1 printf( ) function with format specifiers:

Output sends a copy of the data in the computers memory to another location like a monitor, printer, disk etc. without erasing the data from the computer's memory or changing the way the data is stored. One such function which can do this job in C is the versatile **`printf()`**, which can **display all types of data** and can work with multiple arguments in one single parameter. In addition, **`printf()`** can **format the way the data appears**. In order to format all types of data and to display them, the **`printf()`** function parameter is divided into two parts. The **first part** is called the **control string** or the format string. The control string indicates where the data is to be displayed on the string. It includes **within double quotes** i.e. " " any literal text that needs to be displayed along with special characters called **format specifiers**. The format specifiers are *placeholders in the literal text* that indicate the type and positioning of the data.

Each specifier starts with a **%** sign followed by a letter indicating the data type as shown below.

Specifier	Data Type	Range	Bytes	Example
%d	Signed integer	-32768 to +32767	2	25
%ld	Long signed integer	-2147483648 to +2147483647	4	-2547896
%u	Unsigned integer	0 to 65535	2	55525
%lu	Long unsigned integer	0 to 4294967295	4	958478625
%f	Float	-3.4e38 to +3.4e38	4	12.051000
%lf	Double	-1.7e308 to +1.7e308	8	5.368e
%Lf	Long double	-1.7e4932 to +1.7e4932	10	-1.5e400
%e	Scientific notation	-3.4e38 to +3.4e38	4	1.205100e+01
%g	Shortest decimal /scientific notation	-3.4e38 to +3.4e38	4	12.051
%c	Character	0 to 255	1	C
%s	String	N.A.	N.A.	New String

Same number displayed using %f, %e & %g specifiers

The **second part** of the parameter is called the **data list**, containing the values, constants or variables that are to be displayed using the control string. The data list is **separated** from the control string by a comma. A comma separates each piece of data in the data list. When the compiler creates the object code, it substitutes the placeholders by the data in the list. The general format of a **`printf()`** function is:

```
printf("control string with placeholders", data1,data2,…);
```
Data List

displays the value contained in the variable *age*, over here

e.g.: **`printf("I am %d years old", age);`**

If the value contained in `age` is **17** the output of the above statement will be: **`I am 17 years old`**

The **`printf()`** function is thus actually passing two parameters to the library function – the **Control String** and the **Data List**. Any number of data can be passed to the **`printf()`** function, but a format specifier must be included for each item and moreover the **values in the data list must appear in the same order as their specifiers**. Thus the first item in the list is substituted for the first format specifier, the second item for the second format specifier and so on as shown below:

Program Line	Output	
`printf("The value of PI is: %f",3.1416);`	`The value of PI is: 3.1416`	
`printf("I am %d years old", age);`	`I am 17 years old`	*(if age = 17)*
`printf("The square of %d is %d", a ,a*a);`	`The square of 3 is 9`	*(if a = 3)*
`printf("%d by %d gives %f",div, dis, quo);`	`5 by 4 gives 1.250000`	*(if div = 5 & dis = 4)*
`printf("My initials are %c & %c", c1, c2);`	`My initials are H & R`	*(if c1='H' & c2='R')*
`printf("The ASCII of %c is %d", num, num);`	`The ASCII of A is 65`	*(if num = 65)*
`printf("%%d is used to format an int");`	`%d is used to format an int`	

In all the above examples, care has been taken such that the order in which the format specifiers appear, match with the data and data types of the data list. Thus in the example:

`printf("%d by %d gives %f", div, dis, quo);` → **`div`** and **`dis`** have to be **`int`** type variables and **`quo`** has to be a **`float`** type variable to match the declarations in the control string, so that the first **`%d`** will display the value in **`div`**, the second **`%d`** will display the value in `dis` and the third, **`%f`** will display the quotient **`quo`**. **In case of a mismatch between specifier type and data type, proper output will not be displayed**.

Program1:

```
/*Program to display the use of different format specifiers*/
#include<stdio.h>
main()
{
int dig1;
float dig2;
long result1;
double result2, result3;
printf("Enter an int <100 (number1): ");
scanf("%d",&dig1);
printf("Enter a float (number2): ");
scanf("%f",&dig2);
       result1=dig1*10000000.0+dig1*1000000.0;
       result2=dig2*10000000.0+dig2*1000000.0;
       result3=dig2*0.000001+dig2*0.0001;
printf("\nnumber1*10000000+number1*1000000 gives:...\n");
printf("With %%d specifier : %d\n",result1);
printf("With %%ld specifier : %ld\n",result1);
printf("With %%lu specifier : %lu\n\n",result1);
printf("number2*10000000+number2*1000000 gives:...\n");
printf("With %%e specifier : %e\n",result2);
printf("With %%g specifier : %g\n\n",result2);
printf("number2*0.000001+number2*0.0001 gives:...\n");
printf("With %%e specifier : %e\n",result3);
printf("With %%g specifier : %g\n",result3);
return(0);
}
```

Output of the above program with number1 = 89 and number2 = 9.2:

```
       Enter an int <100 (number1): 89
       Enter a float (number2): 9.2

       number1*10000000+number1*1000000 gives:...
       With %d specifier : 23232
       With %ld specifier : 979000000
       With %lu specifier : 979000000

       number2*10000000+number2*1000000 gives:...
       With %e specifier : 1.012000e+08
       With %g specifier : 1.012e+08

       number2*0.000001+number2*0.0001 gives:...
       With %e specifier : 9.292000e-04
       With %g specifier : 0.0009292
```

Program2:

```
/*Program to display the use of the %s specifier*/
#include<stdio.h>
#define COM1 "Thank You"
main()
{
char com2[]="Subhash";
printf("Using %%s specifier to print: %s\n",COM1);
printf("Using %%s specifier to print: %s\n",com2);
return(0);
}
```

Output of the above program:

```
       Using %s specifier to print: Thank You
       Using %s specifier to print: Subhash
```

In program1, result1 gives a long integer since its value exceeds the range of integers i.e. 32767. Hence it cannot be displayed using the **%d** specifier and gives a wrong result as can be seen from the first line of the output (i.e. 23232). When displayed using the **%ld** specifier, the correct result gets displayed.

Note: In the example:

```
printf("The ASCII of %c is %d", num, num);
```

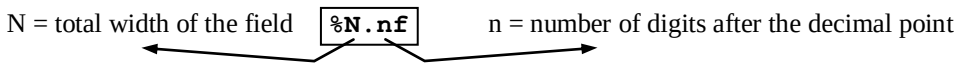
different specifiers are used to display the same integer variable num. Here the **%c** specifier, which displays a character, converts the integer represented by num to the corresponding ASCII character and the %d specifier displays the value contained in num. This statement can be used to display all the ASCII characters and their corresponding ASCII values using proper looping. **Thus we can change the displayed output by changing the format specifier for the same variable**.

Two more format specifiers of interest are the **%0** & the **%x** specifier which converts an integer type data to the corresponding **Octal** and **Hexadecimal** number respectively. For example:

```
printf("The octal no. for %d is %0",num, num); → The octal no. for 12 is 14
printf("The hex no. for %d is %x",num, num);   → The hex no. for 12 is C
```

Field Width Specifiers: The next thing that the format specifiers are used for are for formatting the way the data is displayed. This is done using the *Field Width Specifiers*. The spacing and the number of characters displayed are ontrolled by the field width specifiers. The most common use of width specifiers is to display currency value on the screen. Usually due to obvious reasons, a currency is assigned a float value and all float values are displayed with 6 digits after the decimal place which is useless when dealing with currencies i.e. Rs 25.580000 is something that is not desired. Depending on some compilers this may also be displayed as Rs. 25.579998. In such cases a field width specifier is used to customise and display the output as say Rs. 25.58.

The general format for a floating point field width specifier is:



N reserves the number of spaces allotted for a field. In case the data length (including the decimal part and the decimal point) is less than N then extra spaces will be put before the data to make the total as N. This does not imply that the full data will not be displayed if the data length is more than the specified length. In such case **C** ignores the field width specified and displays the whole data. The following examples clarify the use of field width specifiers. A marker to denote each column of text is printed, to trace the specified width.

Certain special symbols when inserted along with the field width specifiers can allign (justify) the data along certain lines or put extra characters along with the data. These are:

- **0** (zero): This causes leading zeroes to come in place of blank spaces before the data, if the specified field width is longer than the actual data width.
- – (minus): This causes the data item to be left justified within the specified field. Thus blank spaces to fill the remaining field width will be added after the data and not before it.
- + (plus): This causes a sign to precede each signed numerical data item i.e. depending upon the value of the data, either a positive or a negative sign will precede the data.
- # (hash): When using a e-,f- or g-type conversion this causes a decimal point to be present in all floating point numbers even if the data item is a whole number. It also prevents the truncation of trailing zeroes in g type conversion.

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
printf("Cost is Rs%f:",26.39);       → Cost is Rs26.390000:
printf("Cost is Rs%.2f:",26.39);     → Cost is Rs26.39:
printf("Cost is Rs%8.2f:",26.39);    → Cost is Rs   26.39:
printf("Cost is Rs%-8.2f:",26.39);   → Cost is Rs26.39   :
printf("Cost is Rs%5.0f:",26.39);    → Cost is Rs   26:
printf("Cost is Rs%08.2f:",26.39);   → Cost is Rs00026.39:
printf("Profit is Rs%+8.2f:",prof);  → Profit is Rs  +26.39:
printf("Profit is Rs%-+8.2f:",prof); → Profit is Rs+26.39  :
printf("Net charge is :%-+4d:",c);   → Net charge is :+26 :
printf("Net charge is :%#6.0f:",c);  → Net charge is :26.   :
```

```
printf("Net charge is :%#6.0f:",c);  → Net charge is :26.   :
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

2.2 Escape Sequences:

Apart from containing the format specifiers for formatting the text, the **printf()** function can also **control the way the cursor moves on the screen** and some other computer functions by using special codes called *Escape Sequences*.

Each code begins with a **backslash** "\" which identifies the character that follows as an escape sequence. When the compiler encounters the backslash, it does not display the next character but performs the function indicated by it. The different types of escape sequences along with examples are given below:

- The **newline** code: the sequence **\n** performs a **newline** command. There is no need to put any blank space between the escape sequence and the preceeding or following text.
  Example1: `printf("A\nB\nC");` **\n** inserts a line after displaying A on the screen
  Output1:
  ```
  A
  B
  C
  ```

- The **tab** code: the sequence **\t** moves the cursor to the next **preset tab** stop. The tab escape sequence is used to allign data or text along a particular vertical line.
  Example2: `printf("0\t1\t2\t3\t4");` → **\t** inserts a tab after displaying 2 on the screen
  Output2: `0  1  2  3  4`

  The same **printf()** function would have displayed the following with and without using the **\t** sequence:
  Ex3: `printf("01234");` → `01234`
  `printf("0\t1\t2\t3\t4");` → `0  1  2  3  4`

  Ex4:
  ```
  printf("S\tM\tT\tW\Tt\tF\tS");          → S   M   T   W   T   F   S
  printf("\t\t1\t2\t3\t4\5");             →         1   2   3   4   5
  printf("6\t7\t8\t9\t10\t11\12");        → 6   7   8   9   10  11  12
  printf("13\t14\t15\t16\t17\t18\19");    → 13  14  15  16  17  18  19
  ```

- The **return** code: the sequence **\r** performs a **carriage return**, moving the cursor to the **start of the same line** without moving down to the next line. Hence if anything is written after executing the \r sequence, it will overwrite the existing text.

  Example5: `printf("Left\rRight");`
  Output5: `Right` → Only the word `Right` will be displayed

- The **backspace** code: the sequence **\b** moves the cursor just **one position to the left**. It is non-destructive and **does not erase any character** as it moves to the left. If after a backspace command, a newline command is given, the cursor moves to the next line without inserting a blank line.

- The **formfeed** code: when output is sent to the printer then the **\f** sequence **ejects** the current page.

- Displaying certain **special characters**: to distinguish between a syntactical sequence and any special punctuation mark like **'**, **"**, \ etc. in the control string, the character to be displayed is combined with the escape sequence \ .
  Example5: `printf("Press \"ENTER\" to start");` → `Press "ENTER" to start`
  Example6: `printf("\\n inserts a new line");` → `\n inserts a new line`

2.3 The putchar( ) function:

The **printf()** function is a general purpose function to display any string with or without format specifiers along with data, onto the screen. Apart from **printf()** two other function which can display characters on the screen are the **putchar()** function and the **puts()** function.

The **putchar()** function displays a single character value on the monitor. The parameter of the **putchar()** function must be either a *char literal*, a *char constant* or a *char variable*. It is used instead of a **printf()** function at places, where only a single character needs to be displayed (viz. in prompts or options). **This is done so, as the putchar() function does not do any formatting and hence executes faster and requires less space**

to compile. When using literals, the character literal should be put inside single quotes as **'A'**. Examples of the **putchar()** funcion are given below:

1. `putchar('H');` Output (*using char literal)* → **H**

2.
```
#define TRUTH 'Y'
main()
{putchar(TRUTH);}
```
Output (*using char constant)* → **Y**

3.
```
main()
{   char initial;
    initial = 'J';
    putchar(initial);}
```
Output (*using char variable)* → **J**

4. `putchar('\n');` Outputs a *new line*, treating the escape sequence **\n** as a **single control character** and not as two separate characters **** and **n**

5. `putchar('\0');` Outputs a null character and is same as `putchar(' ');`

The **putchar()** function is defined in the **stdio** header file and is derived from the **putc()** function, which is used to send a data to a specified device such as a disk or a printer. The stdio header file contains information that uses **putc()** to perform the **putchar()** function. As the **put** functions can output to disk files, certain provisions are made for **putc()** and **putchar()** to accommodate codes that may not fit into a single memory space provided for a **char** type variable. Accordingly originally **putc()** was designed to take integer type variables as argument. The compiler internally converts the **int** type variable to a **char** type. **Hence with the putc() or putchar() function one can use either an int or a char as the argument** as shown below:

-
```
main()
   {     int initial;
         initial = 'S';
         putchar(initial);
   }
```
Output is → **S**

-
```
main()
   {     char initial;
         initial = 'S';
         putchar(initial);
   }
```
Output is → **S**

2.4 The puts( ) function:

Whereas the **putchar()** function displays a character on the screen, the **puts()** function is used to **display a text string** on the screen. Similar to **putchar(),** the **puts()** function displays unformatted string and hence is used for compact and faster code execution. Though formatting cannot be done, but the **escape sequences** like **\n** or **\t can be used** along with the string as shown below:

```
main()
{
char wish1[]="Good Morning Calcutta";
char wish2[]="Good\tMorning\tCalcutta";
clrscr();
puts(wish1);                                  Output is →
putchar('\n');                                          →
puts(wish2);                                            →
putchar('\n');                                          →
puts("Good\n\tMorning\n\t\tCalcutta");                   →
getch();
return(0);
}
```

newline

tab

```
Good Morning Calcutta

Good    Morning   Calcutta

Good
        Morning
                  Calcutta
```

Tab positions

2.5 Input using scanf( ) function:

Input is the process by which data is fed into the computer for use by the program. This data input to the computer is stored in a variable to be used when required by the program. The data input to a variable can change each time a new input is given to the same variable. We will discuss here the different functions that are available in C to input data from the keyboard – a standard input device. **Scanf()** is the general purpose input function which can be used to input all types of data from the keyboard. **Scanf() scan**s **f**ormatted characters from the

keyboard. The function scans the keyboard for any keystroke and then interprets the characters input, based on the format specifiers. One **scanf()** function can take different types of data all at the same time.

```
scanf("control string",data list);
```

Similar to the **printf()** function, the **control string** contains the format specifiers, which are known as **conversion characters** as they determine how the data is to be interpreted during input.

The **data list** contains the variable name *addresses* (and NOT just the variable names) at which the input data is going to be stored. The address of the variable is denoted by placing the pointer operator '**&**' before the variable name for *numeric* (i.e. **int**, **float**, **double** etc.) and *character* (i.e. **char**) type variables. The pointer operator points to the memory address at which the contents of the variable are stored. For example, for the variable **age**, **&age** refers to the memory address at which the content of the variable age is stored. Thus if the value of age is **25** and the value is stored in the memory address 60020, then age equals 25 and **&age** equals **60020**. Note that for string type variables no such pointer operators are required (as a string is a type of array of characters and will be discussed in detail later).

One important thing should be kept in mind while constructing the **scanf()** argument i.e. the **number and type of conversion characters in the control string should exactly match or correspond to the number and type of variable names in the data list**. The following examples will make the statement clear:

```
main()
{
int age, class;
char section;
char first_name[10], last_name[15]; → String type variables are declared through character
puts("Enter your First-Name, Last-Name, Age, Class, Section\n");
puts("(Separate each piece of data with a blank)\n");
scanf("%s%s%d%d%c",first_name,last_name,&age,&class,&section);
printf("Your Name is %s %s\n",first_name,last_name);
printf("Your age is %d years\n",age);
printf("You are in Class %d, Section %c\n",class,section);
return(0);
}
```

The output of the above program will be the following:

```
Enter your First-Name, Last-Name, Age, Class, Section
(Separate each piece of data with a blank)
Xytiz Saraf 17 11 C              → Data is entered through the keyboard separated by blanks
Your Name is Xytiz Saraf         → Data is printed out onto the monitor
Your age is 17 years
You are in Class 11, Section C
```

There are certain things to note about this program and the use of **scanf()**.

- Firstly, in the control string the conversion characters should be written without any blank or any character in-between them to avoid the chance for any compilation error. Thus **"%d%d%f"** is desirable than **"%d %d %f"**. (Extra characters can be put inside the input control string as will be shown later)
- Secondly, whenever a string is input for the **first_name** and **last_name**, we are formatting the input string by the two consecutive **%s** conversion characters. Next the **age** and **class** are input through the two **%d** specifiers and finally the **section** is input through the **%c** conversion character.
- Thirdly, the pointer operator '**&**' is used along with the variable names for the inputs **age (int)**, **class (int)**, **section (char)** while the strings **first_name** and **last_name** are directly assigned to the variable names.
- Finally, when more than one input is there in the **scanf()** function, then during input each piece of data should be separated by a blank(s) or a tab(s) or by an enter(s). **Scanf() ignores any such white-space character** or characters such as blank, tab or enter and considers as valid inputs only non-white-space characters. Thus each piece of data can be separated by more than one blank, tab or enter, with no effect on the input as long as the data types match with the conversion character types. Since **scanf()** ignores any white-space character, if a string is entered with more than one word and with blank spaces separating the words, then only the first word will be accepted and any subsequent words will be ignored by **scanf()**. Thus if the string "**Hello World**" is

entered, then only "**Hello**" will be assigned to the string variable. (**first_name** & **last_name** were entered separately due to this reason).

Be careful in specifying the order in which the conversion characters and the variable names are written. If the order does not match, then the proper data will not be input into the variables and accordingly the output will be garbage (do not forget the good old saying: *garbage in, is garbage out*). One more thing needs to be noted i.e. though during declaration a **char** type variable can be declared as an **int** type but during input the **%c** *specifier should be used in the* **scanf()** if the data is to be used as a **char** type later.

The following simple program can be used to see the effect of white-space characters and invalid data lists in **scanf()** by inputting different combinations of inputs and white-space characters (blanks, tabs, enters).

```
#include<stdio.h>
main()
{
int a, b;
char c[10];
puts("Enter 2 numbers and a string");
scanf("%d%d%s", &a, &b, c);
printf("Number 1 is :%d\nNumber 2 is :%d\nString is:%s", a, b, c);
return(0);
}
```

As discussed earlier, though it is not preferable to use characters other than the conversion characters in the control string of the **scanf()** function but **extra characters can be used** in the control string in the way as shown in the following C statements.

```
puts("Enter the rate of item1");
scanf("Rs.%f", &rate);
printf("The rate you entered is equal to Rs. %.2f", rate);
```

Output of the above program statements will be:

```
Enter the rate of item1                  → Prompt for rate of item1
Rs.29.50                                 → User types in the value as 'Rs.29.50'
The rate you entered is equal to Rs.29.50  → The value of rate used is 29.50
```

In the above example the literal string "**Rs.**" in the control string for **scanf()** indicates that during input the user should type the rate along with the currency sign i.e. **Rs.29.50** for example. This means that the **scanf()** function while interpreting the input stream will expect the "**Rs.**" string before the **float** input and will ignore it and convert whatever value is input *after* the string literal as per the **%f** conversion character as valid input. While using such string literals care should be taken so that the user types in exactly what has been put as the string literal, otherwise improper and unpredictable input will take place. Thus in the above example though the user makes the input as **Rs.29.50** but the value contained in the variable rate will be only **29.50**.

Let us now examine how the **scanf()** function reads the data during input. Data is read by the **scanf()** function from what is called the **input stream**, which means a series of characters being input from some source. In the case of **scanf()** it is the raw data that is input from the keyboard. Rather than take the whole data as it has been typed by the user and assign it to a variable, the **scanf()** function reads the data as a series of meaningless characters and uses the *conversion characters* to **convert** the raw data to the desired data types which agree with the conversion characters. During conversion it ignores all white-space characters and looks for the first non-white-space character which matches with the specifier, and assigns it to the respective variable. The assignment stops at the first non-matching character.

Important Note!: The **scanf()** function uses an area of the memory called the *buffer* to store the characters being typed into the input stream, before converting the data using the conversion characters. In case **scanf()** terminates early (as may be the case when dealing with char type inputs) some characters may remain in the buffer unused. It may so happen that the next input operation can start reading the characters still remaining in the buffer from the last input instead of reading the current input, thus causing an error in the data stored in the input variables. To avoid this problem the **fflush()** function is used which flushes out or clears any data that may remain in the buffer. The argument of the **fflush()** function is the *buffer name* that we want to clear. In this case the argument should be the buffer related to the standard input device i.e. the keyboard and which is designated by '**stdin**' (**st**andar**d in**put device). The following example will make the use of the **fflush()** function clear. We will be inputting data into two character type variables and one integer type variable using the **scanf()** function and again display them on the screen using **printf()**. The first program will not use the **fflush()** function while the second will use the same, all other parts remaining the same.

```
#include<stdio.h>
main()
{
char a, b;
int c=1;
puts("Enter Character1");
scanf("%c",&a);
puts("Enter Character2");
scanf("%c",&b);
puts("Enter a Number");
scanf("%d",&c);
printf("Character1 is %c",a);
printf("\nCharacter2 is %c",b);
printf("\nThe number is %d",c);
return(0);
}
```

When the above program is run, the first prompt will show and **scanf()** will wait for the user to type in a character. After entering the character, when the user presses the **ENTER** key (for **scanf()** to accept the data), the second prompt will be skipped and the third prompt will be displayed. The reason for skipping the second **scanf()** is that as the **%c** converter accepts only a single character, the **ENTER** key which represents a newline character **(\n)** is ignored by the first **scanf()** and remains in the input keyboard buffer. When the program encounters the second **scanf()**, it looks for an input in the memory buffer and finds the newline character already present and accepts the same as a valid character input, assigns it to the variable '**b**' and skips to the next program line. As a result the output of the above program will display the first character entered, then a blank line and finally the integer number entered. The blank line will correspond to the newline character **\n** that is stored in the variable '**b**' and displayed by the second **printf()** as the blank line. The output of the above program is shown below:

Enter Character1	→ the prompt for character1 is displayed
P	→ scanf( ) inputs the first character from the keyboard along with the ENTER
Enter Character2	→ the prompt for character2 is displayed but the user is not asked for the input
Enter a Number	→ the next prompt for the number is displayed
12	→ scanf( ) inputs the number from the keyboard
Character1 is p	→ printf( ) displays the first character input in 'a' along with the string literal
Character2 is	→ printf( ) displays only the string literal and executes the newline \n for 'b'
	→ the \n of the above printf( ) is displayed
The number is 12	→ the number in 'c' is displayed

The same program is written below using the **fflush()** function and the output is shown below the program. Here the **fflush()** after each **scanf()** for the characters, clears the memory buffer and hence no residual data is left behind for the next **scanf()** to accept and causes expected running of the program.

```
#include<stdio.h>
main()
{
char a, b;
int c;
puts("Enter Character1");
scanf("%c",&a);
fflush(stdin);
puts("Enter Character2");
scanf("%c",&b);
fflush(stdin);
puts("Enter a Number");
scanf("%d",&c);
printf("Character1 is %c",a);
printf("\nCharacter2 is %c",b);
printf("\nThe number is %d",c);
return(0);
}
```

The output of the above program will be:

```
Enter Character1   → the prompt for character1 is displayed
p                  → scanf() inputs the first character from the keyboard along with the ENTER
Enter Character2   → the prompt for character2 is displayed
q                  → scanf() inputs the second character along with the ENTER
Enter a Number     → the next prompt for the number is displayed
12                 → scanf() inputs the number from the keyboard
Character1 is p    → printf() displays the first character input in 'a' along with the string literal
Character2 is q    → printf() displays the second character input in 'b' along with string literal
The number is 12   → the number in 'c' is displayed
```

2.6 Input using gets( ) function:

Apart from the **scanf()** function there is the **gets()** function to input any type of unformatted text/string into a variable. The **gets()** function can be used to input any type of string data, even with blank spaces separating the words in the input stream. The argument of the **gets()** function contains the string variable name (*without any pointer operator* i.e. &) as shown in the example below:

```
#include<stdio.h>
main()
{
char name[21];
puts("Enter your name (<=20 characters)");  → Prompt for the name
gets(name);                                 → The gets() function waits for the user to
printf("\nYour name is %s", name);              type in the name
return(0);
}
```

The output of the above program will be:

```
Enter your name (<=20 characters)      → Prompt for name
Ion Roy                → User types in name as Ion Roy with a blank between the two words
Your name is Ion Roy                   → printf() prints the name as Ion Roy
```

If the same program was written using **scanf()** function, the program and the output would have been:

```
#include<stdio.h>
main()
{
char name[21];
puts("Enter your name (<=20 characters)");
scanf("%s",name);
printf("\nYour name is %s", name);
return(0); }
```

The output of the above program will be:

```
Enter your name (<=20 characters)      → Prompt for name
Ion Roy                → The user inputs the name as Ion Roy with a blank between the two words
Your name is Ion       → The output is Ion only as the scanf() has ignored the part after the blank
```

The **gets()** function waits for the user to input the data string. As the data is typed in character by character, the same gets echoed (i.e. displayed) on the screen. After writing in the string, only when the **ENTER** key is pressed, the typed characters get assigned to the variable name with the C compiler adding the null terminator **\0** to complete the string. As long as the **ENTER** key is not pressed if any character(s) is mistyped, one can use the backspace key to delete the unwanted characters and type back the correct ones. Only when the enter key is pressed will the string input be assigned to the variable.

2.7 Input using getch( ) function:

Similar to the **putch()** function which is used to output a character data, there is the **getch()** function to input any character data. One can input the character either as a `char` or an `int` type as had been explained during discussing the **putch()** function. Like other functions, the **getch()** function **does not appear at the beginning of a line** but is assigned to a variable with the '**=**' sign. One more thing to note about the **getch()** function is that

it does not take anything as its argument i.e. nothing is to be put inside the brackets following the `getch`. When the program encounters the **`getch()`** function, it stops and waits for some input from the keyboard. Since **`getch()`** expects a single character as input, whenever any character is typed into the keyboard (including a blank), the function does not wait for the user to press the **`ENTER`** key but automatically passes the control to the next program line and assigns the character typed to the variable. The character typed, is **not shown** or echoed on the monitor.

If a **`getch()`** function is used without assigning it to any variable, when the program encounters the function as usual it waits for some input from the keyboard. Whenever something is typed into the keyboard, the program passes on to the next line without assigning the character typed to any variable and the data entered is lost forever. The **`getch()`** function is used in this manner when someone wants to halt the program until the user wants to proceed further, as with the "Press any key to continue" type of comment. The following example illustrates the use of the **`getch()`** function both with and without a variable assignment.

```
#include<stdio.h>
#include<conio.h>
main()
{
int ascii;              → Variable ascii is declared as an int (same effect if declared as a char)
clrscr();
puts("Input any character to see its ASCII Code");     → Prompt to input any character
ascii=getch();          → The character typed is assigned to the variable ascii
printf("The ASCII value of %c is %d.",ascii, ascii);
puts("\nPress any key to continue");
getch();                → The program halts and waits for an input to proceed further
return(0);
}
```

2.8 Input using getche( ) function:

In the **`getch()`** function, whatever is entered through the keyboard is not displayed onto the monitor screen. To display or echo the character typed, onto the screen we use the **`getche()`** function (the 'e' for '**e**cho' is added at the end of the **`getch()`** function to derive the name). The following program section shows the use of the **`getche()`** function:

```
#include<stdio.h>
#include<conio.h>
main()
{
char prompt;            → Variable prompt is declared as a char
clrscr();
… … …
… … …
printf("\nEnter Male or Female (M/F):");
prompt=getche();        → The program halts and waits for an input to proceed further
printf("\nYou have typed %c as your sex", prompt);
… … …
return(0);
}
```

Output of the above section:

```
Enter Male or Female (M/F): M      → Prompt to enter M or F for Male or Female and
You have typed M as your sex         character typed in, is showed on the screen and
                                     displayed using printf()
```

The same program section if written using the **`getch()`** function would have shown as output the following:

```
Enter Male or Female (M/F):        → Prompt to enter M or F for Male or Female and
You have typed M as your sex         character typed in, is not showed on the screen but
                                     displayed using printf()
```

2.9 Input using getchar( ) function:

In the **getch()** and **getche()** functions, whatever is typed through the keyboard need not be entered by pressing the **ENTER** key but automatically gets entered. This is not the case with the **getchar()** function, where after typing in the character the same is to be followed by the **ENTER** key to assign the input to the variable. The **getchar()** function also **echos** the character typed, onto the screen.

Let us now write a **sample program** to see how a user-friendly program should be written, which when run will not cause any confusion on the part of the user. The program should indicate its purpose, the user inputs should be clearly preceded by proper prompts so that the user knows exactly what to type in and thus avoiding any input error.

Problem: If telephone calls made by a subscriber are less than 300, then charge per call is Rs. 0.80. If calls are between 301 to 600, then rate per call is Rs. 1.20 and if calls are more than 601, then rate is Rs. 1.50 per call. Write a C program to calculate and generate the telephone bill based on the number of calls made by a subscriber.

```
/************************************/
/*PROGRAM TO CALCULATE TELEPHONE BILL*/
/************************************/
/*program designed by Atlanta Banerjee: revision-0 dated 02/02/02*/
#include<stdio.h>
#include<conio.h>
main()
{
int year, total_calls, slab1, slab2=0, slab3=0;
float bill_amount;
char month[4], name[16];
clrscr();
puts("   ######################################################################");
puts("   # CALCULATION OF TELEPHONE BILL BASED ON RATE & NUMBER OF CALLS #");
puts("   ######################################################################");
puts("\nPlease enter your name (<=15 characters including blanks)");
gets(name);
puts("\nPlease enter the total number of calls made by you: ");
scanf("%d",&total_calls);
fflush(stdin);
puts("\nPlease enter the last month name (in 3 letters as Jan, Feb etc.)");
gets(month);
puts("\nPlease enter the year (as 2002)");
scanf("%d",&year);
if (total_calls>600)
       {bill_amount=300*0.80+300*1.20+(total_calls-600)*1.50;
        slab1=300;
        slab2=300;
        slab3=total_calls-600;}
if (total_calls>300&&total_calls<601)
       {bill_amount=300*0.80+(total_calls-300)*1.20;
        slab1=300;
        slab2=total_calls-300;}
if (total_calls<301)
       {bill_amount=total_calls*0.80;
        slab1=total_calls;}
printf("\nTelephone bill for %s for the month of %s,%d",name,month,year);
printf("\nNumber of calls @Rs.0.80/call is \t\t%4d",slab1);
printf("\nNumber of calls @Rs.1.20/call is \t\t%4d",slab2);
printf("\nNumber of calls @Rs.1.50/call is \t\t%4d",slab3);
printf("\n\t\t\t\t\t    --------");
printf("\nThe total number of calls made is equal to \t%4d",total_calls);
printf("\nThe bill amount is equal to Rs.%8.2f",bill_amount);
puts("\nPress ENTER to exit");
getch();
return(0);
}
```

The output of the above program is given below:

```
   ######################################################################
   # CALCULATION OF TELEPHONE BILL BASED ON RATE & NUMBER OF CALLS #
   ######################################################################

Please enter your name (<=15 characters including blanks)
Albert Einstein

Please enter the total number of calls made by you:
562

Please enter the last month name (in 3 letters as Jan, Feb etc.)
Jan

Please enter the year (as 2002)
2001

Telephone bill for Albert Einstein for the month of Jan,2001
Number of calls @Rs.0.80/call is                300
Number of calls @Rs.1.20/call is                262
Number of calls @Rs.1.50/call is                  0
                                              --------
The total number of calls made is equal to      562
The bill amount is equal to Rs.  554.40
Press ENTER to exit
```

---***---