Chapter 4

Console Input/Output

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Introduction
- Types of Console I/O Functions
- Unformatted Console I/O Functions
  - **Single Character Output:** putch(), putchar()
  - **Single Character Input:** getch(), getche(), getchar()
  - **String Output:** puts()
  - **String Input:** gets()
- Formatted Console I/O Functions
  - **Formatted Output:** printf()
  - **Formatted Input:** scanf()

Introduction

- Input and output are not a part of the C language itself. Dennis Ritchie wanted C to remain compact. With this intention he deliberately omitted everything related to I/O from the primary construct of this language. In C, all the I/O operations are carried out through the standard (built-in) I/O library functions.
- The built-in I/O library functions available in C can be classified into two broad categories:
  1. Console I/O Functions: Functions to receive input from the keyboard and write output to the monitor (In C, the keyboard is the standard input device, also referred to as `stdin` and the screen is the standard output device, also referred to as `stdout`. Together they are called the console).
  2. File I/O Functions: Functions to perform I/O operations on a file (stored on floppy disk or hard disk).

In this chapter, we would be discussing only Console I/O functions. File I/O functions would be discussed in the Chapter "Files & Command Line Arguments".

Types of Console I/O Functions

- C has many console I/O functions, classified as either unformatted or formatted. The chart shows the most common functions under each category.

Console I/O Functions

Unformatted Console I/O Functions

(“Unformatted” means the user can’t specify the format regarding how to read or how to print)

Type	Input	Output

char	getch()
	getche()
	getchar()
		putch()
		
		putchar()

int	---	---
float	---	---
double	---	---
string	gets()	puts()

Formatted Console I/O Functions

(“Formatted” means the user can specify the format regarding how to read or how to print)

Type	Input	Output
char		
int		
float	scanf()	printf()
double		
string		

- **[NOTE]: To use these functions, we have to *include* the header file `stdio.h` and `conio.h` in our program through the preprocessor directives:**

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

descriptions for `getch()` and `putch()` are present in `conio.h` while descriptions for others are in `stdio.h`.

In the following sections, we shall discuss these console I/O functions under each category in detail.

Unformatted Console I/O Functions

Single Character Output: `putch()`, `putchar()`

- **Function Prototypes (as in the library): Both have the same prototypes**

```
int putch(int);
```

```
int putchar(int);
```

- **Syntaxes (function calls to print): Both have the same syntax**

```
putch(character_variable_or_constant);
```

```
putchar(character_variable_or_constant);
```

- **Working: The working of both functions are exactly same. Both functions writes the character (an unsigned character) contented in** `character_variable_or_constant` **to the screen** `(stdout)`**.**

- **Return Value: On success, both functions returns the character written on the screen. If an error occurs, an end of file (EOF) is returned.**

Programming Example 1:

Our aim is to print 'a', 'b' and 'c' as follows:

```
a         b
c
```

The program:

```c
/* PR4_1.c: Program to demonstrate the use of putch() and putchar() */

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

void main()
{
    char ch = 'a';
    putch(ch);
    putch('\t');
    putchar('b');
    putchar('\n');
    putch('c');
}
```

Programming Example 2:

```
/* PR4_2.c: Program to verify the return type of putch() and putchar()*/

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

void main()
{
   char ch1, ch2;
   ch1 = putch('a');
   printf ("\nThe putch () has just printed: %c\n\n", ch1);

   ch2 = putchar('b');
   printf ("\nThe putchar () has just printed: %c\n\n", ch2);
}
```

Output

```
a
The putch() has just printed: a

b
The putchar() has just printed: b
```

Single Character Input: `getch()`, `getche()`, `getchar()`

- **Function Prototypes (as in the library):** All have the same prototypes

```
int getch(void);
```

```
int getche(void);
```

```
int getchar(void)
```

- **Syntaxes (function calls):** All have the same syntax

```
character_variable = getch();
or getch();  /* Called in isolation */
```

```
character_variable = getche();
or getche();  /* Called in isolation */
```

```
character_variable = getchar();
or getchar();  /* Called in isolation */
```

- **Working:** The workings are little bit different
  - `getch():`
    - It reads a single character as soon as it is typed from the keyboard `(stdin)` and assigns it to the `character_variable`. So, the user CAN'T see what is being typed.
    - When called in isolation, it just reads the character but don't assign it to any variable.
  - `getche():`
    - The letter 'e' means 'echo' (display). So $\text{getche()} = \text{getch()} + \text{echo}$.
    - It reads a single character as soon as it is typed from the keyboard `(stdin)`, echoes (displays) it and then assigns it to the `character_variable`. So, the user can see what is being typed.
    - When called in isolation, it just reads and displays the character but don't assign it to any variable.

- **getchar():**
  - Like getche(), it also reads a single character as soon as it is typed from the keyboard (stdin) and echoes (displays) it, but then it waits for the enter key to be pressed. Only after the enter key is pressed it assigns the character to the character_variable. So, the user can see what is being typed.
  - When called in isolation, it just reads and displays the character but don't assign it to any variable after pressing the enter key.

- Return Values: The return values of all the functions are same. All the functions, on success, returns the character typed from the keyboard (which is assigned to the character_variable). On error, the functions returns EOF.

■ Programming Example:

```
/* PR4_3.c: Program to demonstrate the use of getch(), getche() and getchar() */

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

void main()
{
   char ch1, ch2, ch3;

   printf("\n Enter a character: ");
   ch1 = getch(); /*Reads the character as soon as it is typed*/
   printf("\n You have just entered: ");
   putch(ch1);

   printf("\n\n Enter another character: ");
   ch2 = getche(); /*Reads and echoes the character as soon as it is typed*/
   printf("\n You have just entered: ");
   putch(ch2);

   printf("\n\n Enter another character: ");
   ch3 = getchar(); /* Reads and echoes the character as soon as it is typed, but then
                       requires the enter key to be pressed*/
   printf("\n You have just entered: ");
   putch(ch3);
}
```

Output

```
Enter a character:
You have just entered: 5

Enter another character: y
You have just entered: y

Enter another character: p
You have just entered: p
```

String Output: `puts()`

- **Function Prototype (as in the library):**

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

- **Syntax (function call to print):**

```
puts(string_variable_or_constant);
```

- **Working: It writes the string contented in `string_variable_or_constant` to the screen `(stdout)` up to but not including the null character. A newline character is appended to the output (i.e., the cursor moves the to the beginning of a new line after printing the string).**

- **Return Value: If successful, a non-negative value is returned. On error, the function returns EOF.**

Programming Example 1:

Our aim is to print the following:

```
Bikash is a MCA student.
His rollNo is 01.
```

The program:

```
/* PR4_4.c: Program to demonstrate the use of puts()*/

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

void main()
{
    char s1[] = "Bikash is a MCA student."; /*In C, a string is a character array
                                              terminating with NULL character*/
    puts(s1);
    puts("His rollNo is 01.");

    getch();
}
```

■ Programming Example 2:

```
/* PR4_5.c: Program to verify the return type of puts()*/

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

void main()
{
    int n;
    n = puts("Hi");
    printf ("\nStatus (Success = 0, Error = Nothing): %d\n\n", n);
}
```

Output

```
Hi
Status (Success = 0, Error = Nothing):0
```

String Input: `gets()`

- **Function Prototype (As in the library):**

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

- **Syntax (function call):**

```
gets(string_variable);
```

  Note how this syntax differs from that of `getch()`. In `getch()` the syntax was:

  `character_variable = getch();`

- **Working: `gets()` waits for the enter key to be pressed after typing a sequence of character. So, the user can see what is being typed. Once the enter key is pressed, `gets()` reads the sequence of characters typed from the keyboard `(stdin)` and assigns it to the `string_variable` (the buffer pointed to by `string_variable`) until the newline character (i.e., the enter key).**

- **Return Value: On success, it returns the string typed from the keyboard (which is assigned to the `string_variable`). On error, it returns a NULL.**

■ Programming Example:

```c
/* PR4_6.c: Program to demonstrate the use of gets()*/

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

void main()
{
   char s1[50];
   char s2[] = " is a MCA student.";
   puts("Enter the name: ");
   gets(s1);

   putchar('\n');
   puts(strcat(s1,s2)); /*strcat() function is used to concatenate two strings */
   puts("His rollNo is 01.");

   getch();
}
```

Output

```
Enter the name:
Bikas

Bikash is an MCA student.
His rollNo is 01.
```

Formatted Console I/O Functions

Formatted Output: `printf()`

- We have already used this function many times without getting into its details. In this section, we shall explore this library function in detail.
- `printf()` means "print formatted", i.e., the `printf()` function not only prints the outputs on the screen `(stdout)`, but it also controls how the outputs are displayed on the screen (according to the format provided).
- The `printf()` function can be used to print any data type or any combination of the data types.
- Function Prototype (As in the library):

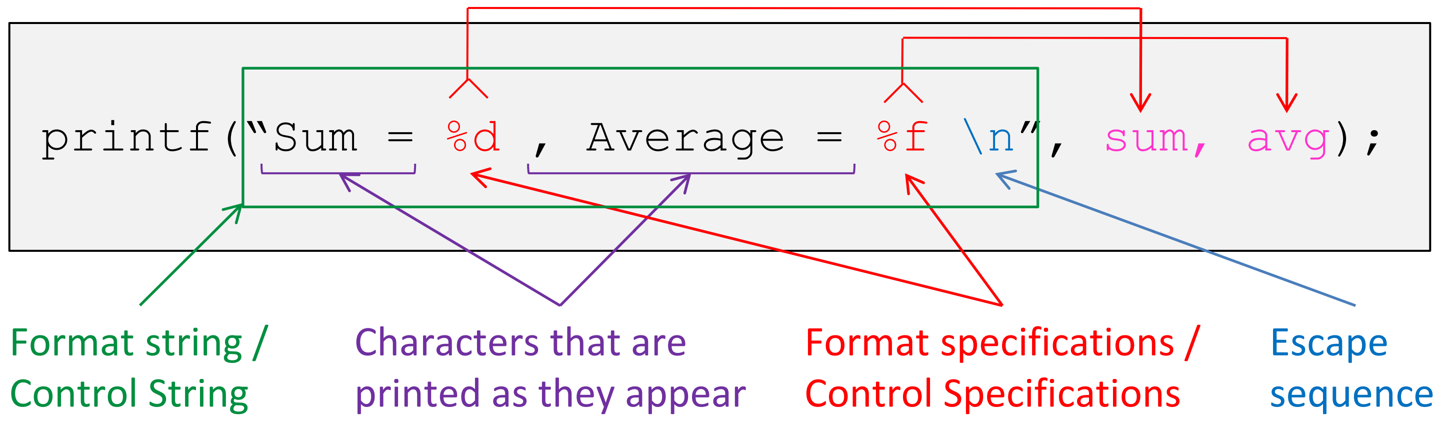
```
int printf (const char*, ...);
```

- **Syntax (function call to print):**

```
printf("format_string", variable_or_constant_list);
```

- **Here** `format_string` **is a string** (corresponds to `const char*` in the prototype). **It's called "format string" or "control string". It can contain:**
  1. **Characters that will be printed on the screen as they appear.**
  2. **Escape sequences (such as \n, \t, \b etc.)**
  3. **Format specifications (or, conversion specifications) that begin with a % sign and ends with a type specifier (such as d, f etc.).**
     - **The format specifications define how the output are displayed on the screen. The specifications are different for different data types.** *(More on format specifications are discussed soon).*
- `variable_or_constant_list` **is a set** (zero or more in number) **of variables or constants. They must match in number, order, and type with the format specifications.**

e.g.,



```
printf("Sum = %d , Average = %f \n", sum, avg);
```

Format string / Control String
Characters that are printed as they appear
Format specifications / Control Specifications
Escape sequence

- **Working: The `printf()` function works as follows:**
  - **It scans the format string from left to right and goes on printing it character by character on the screen `(stdout)` as long as it comes across a % or \ sign.**

    e.g., In the above example, `Sum =` is printed on the screen as it is.
  - **The moment it comes across a format specification** (that begins with a %) **it picks up the 1st variable/constant from "`variable_or_constant_list`" & prints its value in the specified format as per the format specification.**

    e.g., In the above example, the moment `%d` is met the variable `sum` is picked up and its value is printed as an integer.

- Similarly, when an escape sequence is met it takes the appropriate action.

  e.g., In the above example, the moment `\n` is met it places the cursor at the beginning of the next line.

- This process continues until the end of format string is reached, at which point the `printf()` function stops.

- **Return Value:** On success, it returns the total number of characters written on the screen. On error, a negative number is returned.

  e.g.,

  ```
  int n = printf("Hello");
  ```

  If the above statement is executed, the value of `n` will be assigned to 5.

More on the Format Specifications in `printf()`

- **Till now we have used the format specifications in its simplest form, i.e., with a % sign followed by a type specifier, for example %d, %f etc. Format specifications in `printf()` can contain a lot more.**
- **Format specification can contain, in order, the following components:**

```
%[flags][width][.precision]type_specifier
```

e.g.,

```
%-+20.4f
```

The components within the square brackets are optional.

All the components are explained next.

- **The % Sign:**
  - **It indicates the beginning of the format specification.**

■ Type Specifier:

- ➢ It indicates the end of the format specification.
- ➢ Type specifiers are different for different data types. Following is the list of type specifiers that can be used with `printf()`:

Type Specifiers for Character Data Types	
signed character unsigned character	`c`

The special character %	`%`
	e.g., to print 5% on the screen:
	`int n = 5;`
	`printf("%d%%", n);`

[NOTE]: Some other special characters like single quote (‘), double quote (“”), question mark (?), etc. can be printed with the help of escape sequences.

Type Specifiers for Integer Data Types		
signed integer (in decimal form)	d, i	Additionally, these specifiers may be prefixed with the following letters: • h: for interpreting the data type as short (e.g., %hd means short singed integer) • l: for interpreting the data type as long.
unsigned integer (in decimal form)	u	
unsigned integer (in octal form without a leading zero)	o	
unsigned integer (in hexadecimal form without a leading Ox or OX)	x (for printing with small a-f) X (for printing with capital A-F)	

Type Specifiers for Real (float & double) Data Types	
float (in the regular form i.e., [-]dddd.dddd)	`f`
float (in exponential form i.e., [-]d.dddd **e**[+/-]ddd)	`e` (the letter 'e' in the exponential form appears in small) `E` (the letter 'E' in the exponential form appears in capital)
float (either in regular form or in exponential form based on the precision ; in whichever form the precision is smaller)	`g` (if printed in exponential form the letter 'e' appears in small) `G` (if printed in exponential form the letter 'E' appears in small)

[Cont.]

double (in regular form i.e., signed double)	`lf`
double (in exponential form)	`le, lE`
double (either in regular form or in exponential form based on the precision ; in whichever form the precision is smaller)	`lg, lG`
long double (in regular form i.e., signed long double)	`Lf`
long double (in exponential form)	`Le, LE, LG`

Type Specifiers for String Data Type	
string	`s`

- **Flags: Following is the list of flags used with `printf()`:**

Flags	Description
–	Can be used with any data type. When used, the output is left-justified within the specified width (width is discussed next). Remaining field will be blank. If the - sign is absent the out put is right-justified within the specified width.
+	Can be used with only signed numeric items (i.e., signed integers or real numbers) When used, a + or - sign will precede the signed numeric item.
0	Can be used with only numeric items (i.e., integers or real numbers) When used, causes leading zeros to appear for the numeric item.
#	Can be used with only octal or hexadecimal integers. When used, causes the octal or hexadecimal number to be preceded with O or Ox
	Can be used with only real numbers in regular form (non exponential form). When used, causes the decimal point to be present in the real number, even if the number is a whole number.

[NOTE]: The flags are written in the same order as given above.

- **Width:**
  - It is a number that can be used with any data type. If used, it specifies the minimum field width (the converted argument will be printed in a field of at least this width). The output is by default right justified.
  - The width may be specified as *, in which case the value is obtained from the next argument (which must be an `int`) in the argument list.
- **Precession:**
  - It is a number that can be used with integers, real numbers, and string.
    - When used with an integer (d, i, u, o, x, X) indicates the minimum number of digits to be printed.
    - When used with a real number (f, e, g, lf, le, lg, Lf , Le, Lg) indicates the maximum no. of digits after the decimal point to be printed.
    - When used with a string (s type specifier) indicates the maximum number of characters to be printed.
  - Like width, the precession may also be specified as *, in which case the value is obtained from the next argument in the argument list.

- **Example 1 (Character Output):**

Here we have considered that `char z = 'a';`

Sl. No.	Printing Specifications	:Output:	Comment
1	`printf("%c", z);`	`:a:`	Normal output
2	`printf("%6c", z);`	`:     a:`	Output in a field width of 6. Right justified by default.
3	`printf("%-6c", z);`	`:a     :`	Output in a field width of 6. Left justified because of the - flag.
4	`printf("Have you secured 70%% marks in the exam\?");`	`:Have you secured 70% marks in the exam?:`	To display the special character % we have used the format specification %%
5	`printf("%c", 65);`	`:A:`	The format specification %c specifies that the argument 65 (which is an integer) should be display as a character. *The students are advised to examine what happens if a float or a string is specified to be displayed as a character.*

■ Example 2 (Integer Output):

Here we have considered that `int z = 9876;`

Sl. No.	Printing Specifications	:Output:	Comment
1	`printf(“%d”, z);`	`:9876:`	Normal output
2	`printf(“%6d”, z);`	`:  9876:`	Output in a field width of 6. Right justified by default.
3	`printf(“%-6d”, z);`	`:9876  :`	Output in a field width of 6. Left justified because of the - flag.
4	`printf(“%2d”, z);`	`:9876:`	The number is greater than the *minimum* field width. So, it overrides the width specification.
5	`printf(“%-+6d”, z);`	`:+9876 :`	Output in a field width of 6. Left justified because of the - flag. The + sign appears because of the + flag.
6	`printf(“%+06d”, z);`	`:+009876:`	Output in a field width of 6. Right justified by default. The + sign appears because of the + flag. Leading 0’s because of the 0 flag.

[Cont.]

Sl. No.	Printing Specifications	:Output:	Comment
7	`printf("%6.6d", z);`	`:009876:`	Output in a field width of 6. Right justified by default. Minimum no. of digits to print is 6, so two leading zeros.
8	`printf("%6d", ‘A’);`	`:    65:`	Output in a field width of 6. Right justified by default. The format specification %d specifies that the argument ‘A’ (which is a character) should be display as an integer.
9	`printf("%d", 97.5);`	`:97:`	The format specification %d specifies that the argument 97.5 (which is a double) should be display as an integer. So, the fractional part is omitted. *The students are advised to examine what happens if a string is specified to be displayed as an integer.*

■ Example 3 (Float/Double Output):

Here we have considered that `float z = 98.7654;`

Sl. No.	Printing Specifications	:Output:	Comment
1	`printf(“%f”, z);`	`:98.7654:`	Normal output
2	`printf(“%7.4f”, z);`	`:98.7654:`	The width is 7 and the precision is 4. So, the output is same as the normal output.
3	`printf(“%7.2f”, z);`	`:  98.77:`	The width is 7 and the precision is 2. So, the number of digits after the decimal point are truncated to two.
4	`printf(“%-7.2f”, z);`	`:98.77  :`	Same as above. Only left justified because of the - flag.
5	`printf(“%-+7.2f”, -z);`	`:-98.77 :`	Same as above. Only the sign appears because of the + flag.
6	`printf(“%*.2f”, 7, z);`	`:  98.77:`	Indirect width specification. Compare output 3.
7	`printf(“%*.*f”, 7, 2, z);`	`:  98.77:`	Indirect width specification. Compare output 3.

[Cont.]

Sl. No.	Printing Specifications	:Output:	Comment
8	`printf(“%e”, z);`	`:9.876540e+001:`	Normal output in exponential form.
9	`printf(“%11.2e”, z);`	`:  9.88e+001:`	The width is 10 and the precision is 2. So, the number of digits after the decimal point are truncated to two.
10	`printf(“%-11.2e”, z);`	`:9.88e+001  :`	Same as above. Only left justified.
11	`printf(“%10.4e”, -z);`	`:-9.8765e+001:`	The width is 10 which is less than the number of characters to be printed. So, it is overridden. The precision is 4. So, the number of digits after the decimal point are truncated to four.
12	`printf(“%f”, 91);`	`:91.0000:`	The format specification %f specifies that the argument 91 (which is an integer) should be displayed as a float.

- **Example 4 (String Output):**

  Here we have considered that `char z[] = “New Delhi 1100”;`

Sl. No.	Printing Specifications	:Output:
1	`printf(“%s”, z);`	`:New Delhi 1100:`
2	`printf(“%20s”, z);`	`:     New Delhi 1100:`
3	`printf(“%20.10s”, z);`	`:          New Delhi :`
4	`printf(“%-20.*s”, 10, z);`	`:New Delhi          :`
5	`printf(“%5s”, z);`	`:New Delhi 1100:`
6	`printf(“%.5s”, z);`	`:New D:`

- **Example 5 (Mixed Data Type Output):**

Sl. No.	Printing Specifications	:Output:
1	`printf(“%s has secured %d%% marks in C”, “Dev”, 95);`	`:Dev has secured 95% marks in C:`

Formatted Input: `scanf()`

- We have already used this function many times without getting into its details. In this section, we shall explore this library function in detail.
- `scanf()` means "scan formatted", i.e., the `scanf()` function not only reads the inputs from the keyboard `(stdin)`, but it also controls how the inputs are scanned / interpreted (according to the format provided).
- The `scanf()` function can be used to read any data type or any combination of the data types.
- Function Prototype (As in the library):

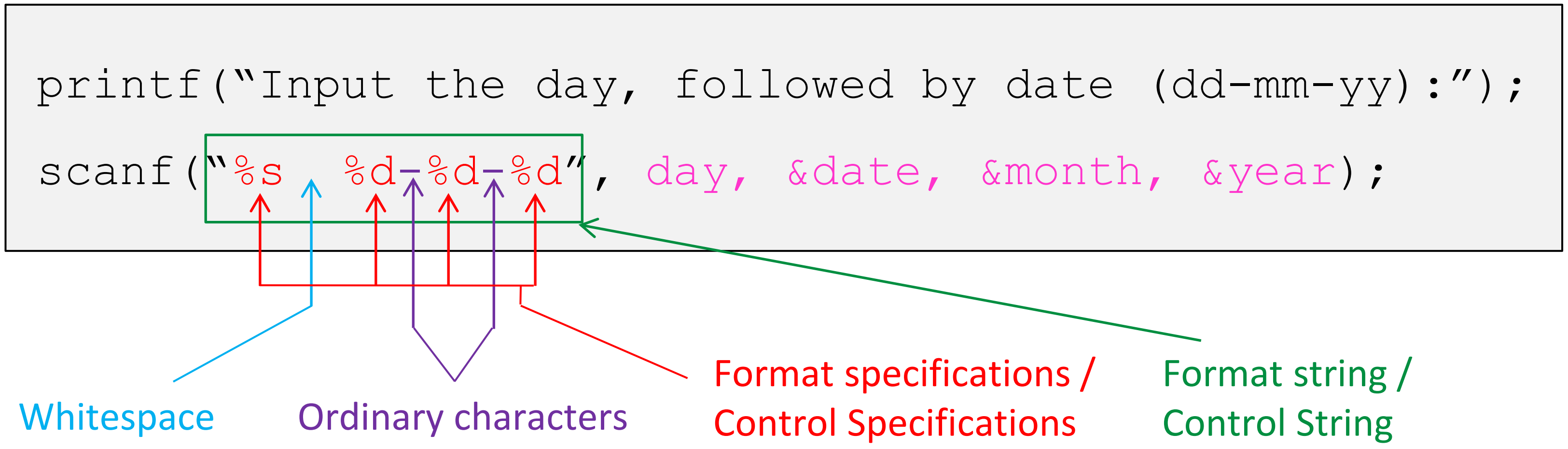
```
int scanf (const char*, ...);
```

- **Syntax (function call):**

```
scanf("format_string", address_list);
```

- **Here** `format_string` **is a string** (corresponds to `const char*` in the prototype). **It's called "format string" or "control string". It can contain:**
  1. **Whitespace characters (spaces, tabs, newlines) which are ignored. They are only added for the readability purpose.**
  2. **Ordinary characters (such as hyphen "-" etc.)**
  3. **Format specifications (or, conversion specifications) that begin with a % sign and ends with a type specifier (such as d, f etc.).**
     - **The format specifications define how the inputs should be interpreted. The specifications are different for different data types.** *(More on format specifications are discussed soon).*
- `address_list` **are the memory addresses of the variables where the scanned inputs are to be placed.**

e.g.,



```
printf("Input the day, followed by date (dd-mm-yy):");
scanf("%s  %d-%d-%d", day, &date, &month, &year);
```

Whitespace
Ordinary characters
Format specifications / Control Specifications
Format string / Control String

- **Working:** The `scanf()` function works as follows:
  - It first examines the format string from left to right (ignoring the whitespace characters), to determine the number, order, and type of *input data items*, also called *input fields* (data items like a string or an integer that corresponds to the format specifications %s, %d etc.) and *ordinary characters* (like a hyphen) it expects.

    e.g., In the previous example, the `scanf()` function after examining the format string expects, in order, the following items:

    - A string, which is the 1st input field (corresponds to %s).
    - An integer, which is the 2nd input field (corresponds to %d).
    - A hyphen, which is an ordinary character.
    - An integer, which is the 3rd input field (corresponds to %d).
    - A hyphen, which is an ordinary character.
    - And finally, an integer, which is the 4th input field (corresponds to %d).

- **Then it starts scanning the inputs item-by-item.**

[NOTE]: How does the `scanf()` *function able to separate the input fields?*

Meaning: Let us consider the previous example i.e.,

```
printf("Input the day, followed by date (dd-mm-yy):");
scanf("%s  %d-%d-%d", day, &date, &month, &year);
```

Some valid input instances are:

```
Input the day, followed by date (dd-mm-yy): Sunday 25-05-15
or
Input the day, followed by date (dd-mm-yy): Sunday 25- 05- 15
```

How does `scanf()` identifies the four input fields "Sunday", "25", "05", "15", in that order?

The approach taken by scanf():

`scanf()` assumes that an input field should be separated from its previous input field by a space, a tab or a newline (enter), an ordinary character (such as the hyphen) or a combination of these.

e.g., let us again consider our valid input instances:

```
Input the day, followed by date (dd-mm-yy): Sunday 25-05-15
or
Input the day, followed by date (dd-mm-yy): Sunday 25- 05- 15
```

Here, the 2nd input field (25) is (must be) separated form the 1st input field (Sunday) by a space. However, the 3rd input field (05) *may or may not* be separated from the 2nd input field (25) by a space because, the ordinary character hyphen immediately follows the 2nd input filed (25), as explicitly specified in the format string.

- If the scanned item happens to be an *input field*, the `scanf()` function interprets (converts) the scanned item according to the format specifications (%s, %d etc.) and stores it at an address, as given by the next address in the “`address_list`”.

  On the other hand, if the scanned item happens to be an *ordinary character*, then the `scanf()` function just matches the scanned item with the corresponding ordinary character but DOESN’T store it.

  For Example: In our case,

  - The 1st scanned item is an *input field*. So, it is (can be) interpreted as a string in accordance with %s and stored at the address “day”.
  - The 2nd scanned item is also an *input field*. It is interpreted as an integer in accordance with %d and stored at the address “&date”.
  - The 3rd scanned item is an *ordinary character (-)*. So, it is matched with the expected ordinary character (-), but not stored.
  - The process continues...

- The `scanf()` functions stops at the moment (and doesn't scan the inputs further) under three conditions:
  - When the scanned *input field* fails to be interpreted (converted) according to the format specification.
  - When the scanned *ordinary character* fails to match the expected ordinary character.
  - When the format string ends.

- Return Value: On success, it returns the total number of input fields scanned successfully. On error, a negative number is returned.

e.g.,

```
int n;
printf("Input the day, followed by date (dd-mm-yy):");
n= scanf("%s  %d-%d-%d", day, &date, &month, &year);
```

If the above statement is executed, the value of `n` will be assigned to 4.

More on the Format Specifications in `scanf()`

- Till now we have used the format specifications in its simplest form i.e., with a % sign followed by a type specifier, for example %d, %f etc. Format specifications in `scanf()` can contain a lot more.
- Format specification can contain, in order, the following components:

  `%[*][width]type_specifier` e.g., `%*8f`

  The components within the square brackets are optional.

  All the components are explained next.
- The % Sign:
  - It indicates the beginning of the format specification.
- The * Sign:
  - It can be used with any data type.
  - When present, indicates that the input field is to be read but ignored, i.e., it is not stored in the corresponding argument.

- **Width:**
  - It is a number that can be used with any data type.
  - When used, it specifies the maximum number of characters to be read from the current input field

Type Specifier:

- It indicates the end of the format specification.
- All the type specifiers that are applicable to `printf()` are also applicable for `scanf()`. There is just one extra addition for strings.

Type Specifiers for String Data Type	
`s`	We have already studied it in `printf()`
`[characters]` `[^characters]`	This is the new specifier. This is called a search set. `[characters]` means, ONLY the `characters` are allowed in the input string. i.e., During reading the input string, the `scanf()` function stops further scanning at the 1st encounter of a character that doesn't belong to `characters`. `[^characters]` does exactly the reverse. It means, ONLY the `characters` (that are after the "^") are NOT allowed in the input string (all other characters are allowed).i.e., During reading the input string, the `scanf()` function continues scanning until the 1st encounter of a character that belongs to `characters`.

Examples:

Sl. No.	Scanning Specifications, Outputs, and Comments

1	*Code:*
	`int n1;`
	`float f1;`
	`printf("Enter an integer followed by a float: ");`
	`scanf("%d %f", &n1, &f1);`
	`printf("Integer: %d\nFloat: %f\n\n", n1, f1);`
	
	Output:
	`Enter an integer followed by a float: 56 89.26`
	`Integer: 56`
	`Float: 89.2600`
	
	Comment:
	Normal scanning.

2

Code:

```
int n1, n2, n3;
printf("Enter three integers: ");
scanf("%d %*d %d", &n1, &n2, &n3);
printf("Integer 1: %d\nInteger 2: %d\nInteger 3: %d\n\n",
n1, n2, n3);
```

Output:

```
Enter three integers: 56 89 32
Integer 1: 56
Integer 2: 32
Integer 3: 5986478
```

Comment:

The second input field (89) is read but ignored because of the * character (%*d). So the 3rd input field (32) is stored in n2. n3 contains a garbage value.

3

Code:

```
int n1, n2;
printf("Enter two four-digit integers: ");
scanf("%2d %4d", &n1, &n2);
printf("Ingeter 1: %d\nInteger 2: %d\n", n1, n2);
```

Output:

```
Enter two four-digit integers: 5689 1234
Integer 1: 56
Integer 2: 89
```

Comment:

%2d specifies that the maximum number of characters to be read from the 1st input field is two i.e., the 1st input field consists of two characters . So n1 is assigned with 56. After this the 2nd input field starts, but a space indicates the end of an input field. So the 2nd input field becomes 89, which is stored in n2.

4

Code:

```
int n1, n2, n3;
printf("Enter a nine-digit integer: ");
scanf("%3d %4d %3d", &n1, &n2, &n3);
printf("Integer 1: %d\nInteger 2: %d\nInteger 3: %d\n\n",
n1, n2, n3);
```

Output:

```
Enter two four-digit integers: 123456789
Integer 1: 123
Integer 2: 4567
Integer 3: 89
```

Comment:

Behavior is similar to the previous output (Serial No. 3)

5

Code:

```
char s1[50];
printf(“Enter a string: ");
scanf("%s", s1);
printf("The string is: %s", s1);
```

Output 1:

```
Enter a string: Niroj
The string is: Niroj
```

Output 2:

```
Enter a string: Niroj Kumar Pani
The string is: Niroj
```

Comment:

Recall that scanf() assumes that the input fields should be separated by whitespace characters. So, the scanf() assumes “Niroj” to be the 1st input filed.

[NOTE]

If you want to read a string that contains spaces, there are two ways:

1. use `gets()` instead of `scanf()`.
2. use %[^\n] format specifier within `scanf()` [Refer examples 7-9.]
   e.g., In the above example, we can rewrite the `scanf()` as: `scanf(“%[^\n”, s1);`

6

Code:

```
char day[50];
int date, month, year;
printf("Input the day, followed by date (dd-mm-yy): ");
scanf("%s  %d-%d-%d", day, &date, &month, &year);
printf("Day: %s, Date: %d, Month: %d, Year: %d", day, date,
month, year);
```

Output 1:

```
Input the day, followed by date (dd-mm-yy): Sunday 25-12-12
Day: Sunday, Date: 25, Month: 12, Year: 12
```

Output 2:

```
Input the day, followed by date (dd-mm-yy): Sunday 25- 12- 12
Day: Sunday, Date: 25, Month: 12, Year: 12
```

Output 3:

```
Input the day, followed by date (dd-mm-yy): Sunday 25 12- 12
Day: Sunday, Date: 25, Month: 5648795, Year: 56412889
```

Comment:

The scanf() expects a hyphen immediately after the 2nd input field (25). In output 3 there is a mismatch. So it stops further scanning. So month and year displays the garbage values.

7

Code:

```
char mailID [50];
printf(“Enter a mailID: ");
scanf("%[a-z,@,.]", mailID);
printf("The mailID is: %s", mailID);
```

Output 1:

```
Enter a mailID: nirojpani@gmail.com
The mailID is: nirojpani@gmail.com
```

Output 2:

```
Enter a mailID: niroj123@gmail.com
The mailID is: niroj
```

Comment:

The search set [a-z, @, .] specifies that only the characters “a-to-z”, an “@” and a”.” are allowed in the input string (no other characters are allowed).
If those characters are entered all those characters are assigned to mailID.
If the input string contains any other character, the string will not be scanned further at the first encounter one of such character.

8

Code:

```
char name [50];
printf("Enter your name: ");
scanf("%[a-z, A-Z]", name);
printf("Your name is: %s", name);
```

Output 1:

```
Enter your name: Niroj Kumar Pani
Your name is: Niroj Kumar Pani
```

Output 2:

```
Enter your name: Niro123 Kumar Pani
Your name is: Niro
```

Comment:

The search set [a-z, A-Z] specifies that only the characters "a-to-z", "A-Z" and a space " " are allowed in the input string (no other characters are allowed).
If those characters are entered all those characters are assigned to mailID.
If the input string contains any other character, the string will not be scanned further at the first encounter one of such character.

9	*Code:*
	`char s1[50];`
	`printf(“Enter a string (not having small letters): ");`
	`scanf("%[^a-z]", s1);`
	`printf("The string is: %s", s1);`
	
	Output 1:
	`Enter a string (not having small letters): AC1@345`
	`The string is: AC1@345`
	
	Output 2:
	`Enter a string (not having small letters): 98@acAQR3`
	`The string is: 98@`
	
	Comment:
	The search set [^a-z] specifies only the characters “a-z” are NOT allowed in the input string (ALL other characters are allowed).
	The reading of the string will continue until the 1st encounter one of these characters.

End of Chapter 4