-Input & Output Operation-

Programming Fundamentals

Introduction

- Reading, processing, and writing of data are the three essential functions of a computer program.
- Most programs take some data as input and display the processed data, often known as information or results, on a suitable medium.
- When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line.
- When we say **Output**, it means to display some data on screen, printer, or in any file.
- C programming provides a set of built-in functions to read the given input and output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

- C programming treats all the devices as **files**.
- So devices such as the display are addressed in the same way as files and the following these files are automatically opened when a program executes to provide access to the keyboard and screen.
- The simplest of all input/output operations is reading a character from the 'standard input' unit (usually the keyboard) and writing it to the 'standard output' unit (usually the screen).
- **#include <stdio.h>** : standard input output header file

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

READING A CHARACTER

getchar(): Reading a single character (a valid char from c charset-256).

```
variable_name = getchar( );
```

where, variable_name is a valid C name that has been declared as char type.

1. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to getchar function.

2. Used on the RHS of an assignment statement, the character value of getchar is in turn assigned to the variable name on the left.

3. Example 1: To take user input for a **choice**

```
char answer;
answer = getchar();  →Y
if(answer == ‘Y’ || answer == ‘y’)
        printf(“\n\nMy name is BUSY BEE\n”);
else
        printf(“\n\nYou are good for nothing\n”);
}
```

Example 2:

The getchar function may be called successively to read the characters contained in a line of text.

The following program segment reads characters from keyboard one after another until the ‘Return’ key is pressed.

```
char character;
character = ‘ ‘;
while(character != ‘\n’) {
character = getchar();
}
```

- Note:

getchar() function accepts any character keyed in including (RETURN and TAB)

This could create problems when we use getchar() in a loop interactively.

A **dummy getchar()** or **fflush()** may be used to ‘eat’ the unwanted newline character.

Character Test Functions: To check the character types

Function	*Test*
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

Library: ctype.h
If true → a non zero value is returned.
If false, 0 is returned.

WRITING A CHARACTER

- Like getchar, there is an analogous function putchar for writing characters **one at a time to the terminal.** It takes the form as shown below:

```
putchar (variable_name);
```

where variable_name is a type char variable containing a character.

E.g. 1: char answer = ‘Y’;

putchar (answer); //displays value of ‘answer’ i.e. Y and not ‘Y’).

E.g. 2: putchar (‘\n’); //cause the cursor on the screen to move to the beginning of the next line.

FORMATTED INPUT

- Formatted input refers to an input data that has been arranged in a particular format.
- **scanf** : means scan formatted

- The general form of scanf is:

  scanf (“control string”, arg1, arg2, ... argn);

Control string/format string: field format in which data is to be entered.

It can have field specifications (%d, %c, %f etc) and an optional number for field width (%20f) and whitespaces (blank, tab, newline).

Arguments: specify the address of locations where the data is stored.

Note: Control string and arguments are separated by commas.

- E.g.: scanf(“%s%d%f”, **name**,&roll,&marks);

Suppose i/p: Mayra 12 76.2 o/p:??

a) Inputting Integer Numbers

%: indicates conversion specification follows,

w: integer number of the number to be read, and

d: known as data type character, indicates that the number to be read is in integer mode.

E.g.: **scanf (“%2d %5d”, &num1, &num2);**

Suppose:

i/p1: 50 31426 → num1: 50 num2: 31426

i/p2: 31426 50 → num1: 31 num2: 426 (50: assigned to first var in next scanf call)

Note: To avoid errors that may come as part of using field width, skip using them.

- Important points:

a) Input data items must be separated by spaces, tabs or newlines.

b) Punctuation marks do not count as separators.

c) If we enter decimal instead of integer, **scanf may ignore decimal part or skip reading further input. (try it)**

d) When the scanf reads a particular value, reading of the value will be terminated as soon as the #characters specified by field-width is reached or until a character that is invalid for input is encountered.

e) The data type character d (in scanf) may be preceded by ‘l’ (letter ell) to read long integers and h to read short integers.

f) Any non whitespace character is allowed between field specification.

e.g.: scanf(“%d-%d”, &a, &b);

accepts input like 123-456

and assigns 123 to a and 456 to b.

b) Inputting Real Numbers:

- Unlike integers, no need to specify field width for real numbers.
- Can use %f for reading both decimal and exponential notations.

e.g.: scanf("%f %*f %f", &x, &y); with the input data 12.34 1.45 99.12

x: 12.34 y: 99.12

*f: ignored

- If the number to be read is of double type: use %lf instead of %f
- Long double : %Lf

c) Inputting Character Strings:

%wc: used to input single character

%ws: used to input collection of characters (string)

e.g.: scanf("%2c %10s", &x, y); //Notice no & in front of string name

i/p: a qwerty → x:a, y:qwerty

Note: scanf terminates reading a string as soon as it encounters a blank space.

e.g.:scanf("%s", y);

i/p: Joe Smith y:?? Joe

- **Some conversion specifications w.r.t scanf:**

1. **%[characters]:** to denote exactly the characters allowed for scanf
2. **%[^characters]:** use circumflex (^) to denote characters not allowed for scanf

e.g.: Consider address is a string (declared)

scanf("%[a-z]", address); //allows only lowercase letters

i/p: rollnumber123

scanf("%[^a-zA-Z]", address); //both upper & lowercase alphabets not allowed

i/p: New Delhi 110002

3. Blank spaces can be included in the strings using %c[]

scanf("%[^\n]", address); //address= Joe Smith

d) Reading Mixed Data Types:

It is possible to use one scanf statement to input a data line containing mixed mode data

Ensure input data items match with the control specification.

e.g. printf(scanf (“%d %c %f %s”, &count, &code, &ratio, name));

will read the data 15 p 1.575 coffee

Note:

1. When an item is entered that does not match the type expected, the scanf function does not read any further and immediately returns the values read.

2. When a scanf function completes reading its list, it returns the value of number of items that are successfully read.

- Common format specifiers:

Code	*Meaning*
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

FORMATTED OUTPUT

- The **printf** statement provides certain features to control the alignment and spacing of print-outs on the terminals.
- Syntax: `printf("control string", arg1, arg2, ....., argn);`
- Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications: define output format (%d, %c, %f..)
3. Escape sequence characters such as \n, \t, and \b.

- The control string indicates how many arguments follow and what their types are.
- The arguments arg1, arg2, ..., argn are the variables whose values are formatted and printed according to the spec. of control string.
- The argument should match in number, type and order with format specifications.

- Simple format specification: `% w.p type-specifier`

a) Both w and p are optional.

b) **w**: integer number; total #columns for the output value, (min field width)

p: integer number; #digits after decimal point (in real numbers) OR #characters to be printed from a string

c) -: left justification

Note:

1) printf never supplies a newline automatically.
2) Use \n for newline

a) Output of Integer Numbers:

Format specification: **% w d**

w: minimum field width (If number> min field width specified, it will be printed in full width overriding the min width specification).

d: specifies that value to be printed is an integer

Note: For short int: use hd and for long int: use ld

Some Examples:

Format	*Output*
printf("%d", 9876)	9 8 7 6
printf("%6d", 9876)	_ _ 9 8 7 6
printf("%2d", 9876)	9 8 7 6
printf("%-6d",9876)	9 8 7 6 _ _
printf ("%06d", 9876)	0 0 9 8 7 6

1) - : minus After % can be used for left justification (ignore 0)

2) It is also possible to pad with zeros the leading blanks

b) Output of Real Numbers:

Format specification: `% w.p f`

w: indicates the minimum number of positions that are to be used for the display of the value

p: indicates the number of digits to be displayed after the decimal point (precision)

Note: Value is rounded-off to ‘p’ decimal places and then right justified.

Leading blanks and trailing zeros will appear as necessary.

The default precision is 6 decimal places.

The negative numbers will be printed with the minus sign.

The number will be displayed in the form: [-] mmm-nnn

Exponential form for format specification: **% w.p e**

It takes the form: [-] m.nnnne[±]xx

Some examples: (Suppose, y = 98.7654) 98.76=9.876x10 power 1

Format	*Output*
printf("%7.4f ",y)	9 8 . 7 6 5 4
printf("%7.2f",y)	9 8 . 7 7
printf("%-7.2f",y)	9 8 . 7 7
printf"%f",y)	9 8 . 7 6 5 4
printf("%10.2e",y)	9 . 8 8 e + 0 1
printf("%11.4e",-y)	– 9 . 8 7 6 5 e + 0 1
printf("%-10.2e",y)	9 . 8 8 e + 0 1
printf"%e",y)	9 . 8 7 6 5 4 0 e + 0 1

c) Printing of a Single Character:

Format: `%wc`

The character will be right justified in the field of ‘w’ columns.

For left justification, use minus symbol (-)

Default value of w=1 (a/b/1..)

d) Printing of Strings:

Format: `%w.ps`

w: field width,

p: #characters of the string to be displayed

- Some examples: Suppose string="NEW DELHI 110001"

Specification	Output																			
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
%s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W		D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
%.5s	N	E	W		D															
%-20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				

e) Mixed Data Output:

We can mix data types in one printf statement

e.g.: printf("%d %f %s %c", a, b, c, d);

Note: printf uses its control string to decide how many variables to be printed with what type.

Common format specifiers for printf:

Code	*Meaning*
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h for short integers

l for long integers or double

L for long double.

- Common output format flags:

Flag	*Meaning*
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
# (with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion.