High-Level and Low-Level Input/Output Functions in C Language

When you access files and devices using the following groups of functions:

1. High-level or Stream-level I/O
2. Low –level I/O

The following describes the differences between high-level and low-level I/O functions.  
  
**High-level or Stream-level I/O**

* Is more flexible and usually more convenient.
* Hides complexity from the programmer.
* When using the I/O functions high-level I/O is slower as compared to the low-level I/O.

**Low-level I/O**

* Provides direct access to files and devices.
* Is complex (Buffer management is to done by the programmer)
* When using I/O functions, low-level I/O is faster as compared to the high-level I/O.
* Uses a “file descriptor” to track the status of the file.

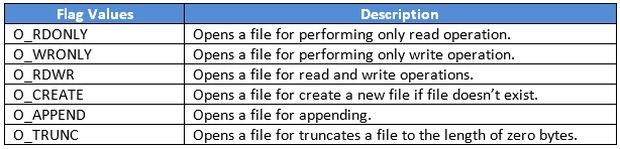
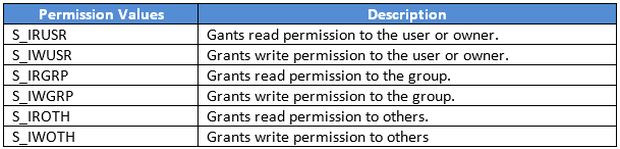
**File Descriptors**  
When you perform any read or write operations you need to open a file. You can create a new file to write to it, or you can use an existing file by discarding its previous contents. You need to specific permissions to be able to perform read or write operations.  
  
When you write to a file, the system checks for the existing permissions and if you have permission then the system returns a non-negative integer called a “file descriptor”. In low-level I/O functions a file descriptor is used to identity a file for all subsequent read or write operations.  
  
**Low-level I/O functions are used for:**

* Accessing files and devices directly.
* Reading binary files in large chunks.
* Performing I/O operations quickly and efficiently.

The low-level I/O system in C provides functions that can be used to access files and devices.

1. Open()
2. Close()
3. Read()
4. Write()

**The open() function**The open() function can be used to open an existing file or to create a new file. This function returns a file descriptor for the file name passed to it. If the file opens sucessfuly then the file position is set to zero position, signifying the beginning of the file.  
  
The open() function has the following syntax:  
  
**Int open (char \*filename, int flags, int perms);**  
  
In the preceeding syntax:

* The first parameter, \*filename is a string that contains the name of file you want to open.
* The second parameter, flags is an integer that specifies how the file is to be opened as the following table lists.  
    
  Flag Values Description  
  **Table 1:** Common Values of a flag
* The third parameter, perms, is the mode of permission with which the file is to be opened. There is a permission associated with files that control read, write and execute access for the owner of the file. This argument is used only when the file is to be created.  
    
    
  **Table 2:**Common Values of a permission

**The close() function**  
The close() function closes the file that was opened using the open() function. It takes the file descriptor as a parameter to close the file. If the file is closed successfully then the value returned is 0. In the case of an error, the close() function returns a value of 1.  
  
The close() function has the following syntax.  
  
**Int close (int filedesc);**  
  
**Example:** open() and close() function in C program:

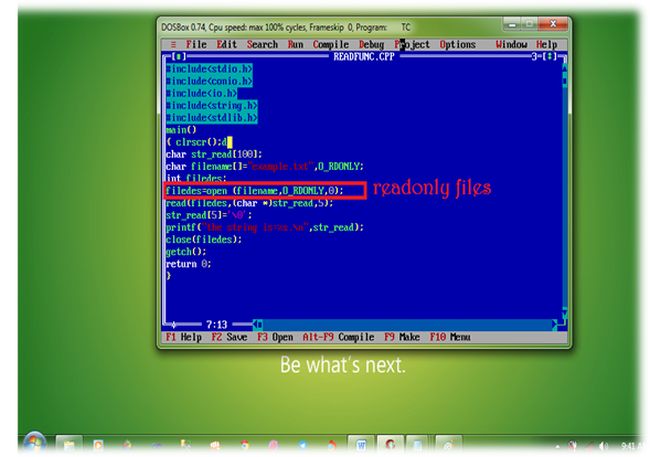
1. #include<stdio.h>
2. #include<conio.h>
3. #include<string.h>
4. #include<io.h>                                   /\*input/outpot header file\*/
5. #include<stdlib.h>
6. main()
7. {
8. clrscr();
9. char filename[]="example.txt",O\_WRONLY ,O\_CREATE;
10. int filedes, close\_err;
11. /\*open the file , filename for writing or create It if it does not exist\*/
12. filedes=open(filename , O\_WRONLY | O\_CREATE , 0);
13. **if**(filedes==-1)
14. {
15. printf("The file cannot be opened\n");
16. }
17. **else**
18. {
19. close\_err=close(filedes);
20. **if**(close\_err==-1)
21. {
22. printf("The file cannot be closed\n");
23. }
24. }
25. getch();
26. **return** 0;
27. }

**The read() function**  
The low-level I/O system defines the read() function for reading data from a file. The read operation starts from the current file position and terminates after all the required bytes have been read.  
  
The read() function has the following syntax:  
  
**Int read (int filedes, char \*buffer, int size);**  
  
In the preceding syntax:

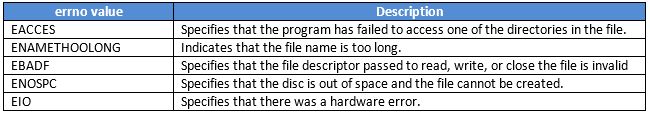
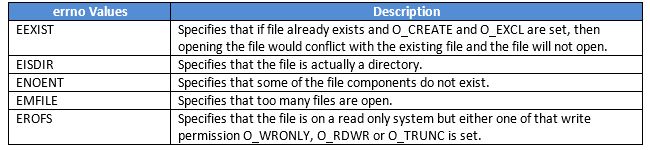
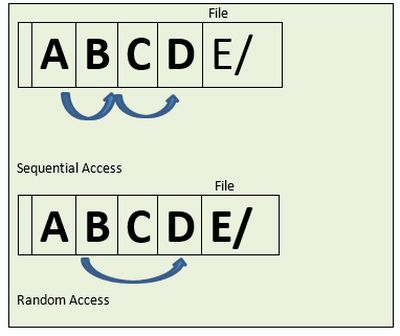
* The first argument is a file descriptor, that is obtained by opening the file.
* The second argument is a character array, where the data will be stored during the read operation.
* The third argument is the number of bytes to be transferred.

The read() function reads the number of bytes that are specified in the parameter size from the file with the descriptor filedes. The results are stored in the buffer. The function returns the number of bytes actually read. A value of zero indicates end of file (except if the value of the size argument is also zero). In the case of an error, the read() function returns -1.  
  
**Example**

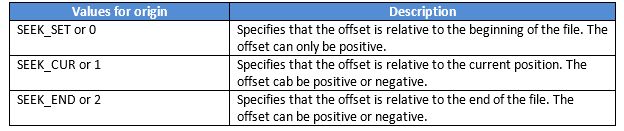
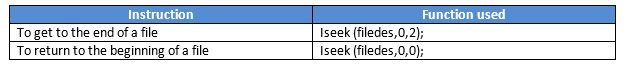
1. #include<stdio.h>
2. #include<conio.h>
3. #include<io.h>
4. #include<string.h>
5. #include<stdlib.h>
6. main()
7. {
8. char str\_read[100];
9. char filename[]="example.txt",O\_RDONLY;
10. int filedes;
11. filedes=open (filename,O\_RDONLY,0);
12. read(filedes,(char \*)str\_read,5);
13. str\_read[5]='\0';
14. printf("the string is=%s.\n",str\_read);
15. close(filedes);
16. getch();
17. **return** 0;
18. }

  
  
**The write() function**  
The write() function enables you to write contents to a file. The arguments passed to the write() function are similar to those of the read() function.  
  
The write() function has the following syntax:  
  
**Int write (int filedes, char \*buffer, int size);**  
  
The write() function picks the number of bytes specified in the parameter size from the buffer and writes it to the file with the descriptor fields. The function returns the number of bytes actually written. This may be the specified number of bytes, but can always be smaller. When the file size is large, the write() function should always be called in an iterating loop, until all the data is written. In the case of an error, the write() function returns -1.  
  
**Example**

1. #include<stdio.h>
2. #include<conio.h>
3. #include<io.h>
4. #include<string.h>
5. #include<stdlib.h>
6. main()
7. {  
      clrscr();
8. char str\_write[]="Krishna Rajput";
9. char filename []="example.txt",O\_WRONLY,O\_CREATE,O\_TRUNC;
10. int filedes,n;
11. // open the file and truncate if it exist.
12. filedes=open(filename,O\_WRONLY | O\_CREATE | O\_TRUNC,0);
13. // write 10 byte of data
14. write(filedes,(char \*)str\_write,10);
15. getch();
16. **return** 0;
17. }

**Error Handling**  
Some of the low-level I/O functions produce errors when accessing files. They return error flags, when they fail to perform the specified task. You can find these types of errors using the variable errno.  
  
The following table lists some values of errno that are common to the open() , close() , read()  and write() functions.  
  
  
**Common errno Values**  
  
Apart from the common values, there are certain errno values that are specific to the open() function. The following table lists the errno values that are specific to the open() function.  
  
  
**errno Values for the open() Function**  
  
There are certain errno values that are specifies to the write() function. The following table lists the errno values that are specifies to the open() function.  
  
errno Values for thee write Function  
**errno Values for thee write() Function**  
  
**Using Random Access in Files**  
The read and write on files are usually sequential in nature. Each input and output takes place in a file immediately after the previous one. Random access permits non-sequential file access so that a file can be read or written out of sequence.  
  
The sequence access and random access in files in shown in the following figure.  
  
  
  
In the proceeding figure, a user can move to segment C of the file, by either accessing A, B and C individually or by directly moving to segment C. The method of accessing file contents individually is called sequential access and the method of accessing file contents directly is called random access.  
  
**The lseek() Function**  
The lseek() function returns the file position, as measured in bytes from the beginning of the file. This function is similar to the high-level file routine fseek() except it accepts a file descriptor as an argument instead of a stream. The lseek() function does not require a read or write of the file for positioning it.  
  
The lseek() function has the following syntax:  
  
**Long lseek (int filedes, long offset, int origin);**  
  
In the preceding syntax:

* The first parameter is the file descriptor.
* The second parameter specifies the number of bytes to move the file position.
* The third parameter is a constant that specifies whether the offset is relative to the beginning of the file, current file position, or end of the file.

The following table lists the various values of this parameter:  
  
  
**The Offset Values**  
  
For example, a user specifies SEEK\_END as the value for the origin and sets the file position after the current end of the file and writes the data. In this case, the result will be a file that will contain zeroes up to the specified position after the end of the file. The blocks of zeroes are not actually written to disk, therefore the file takes up less space on disk. The following table lists some instructions for moving to the end or beginning of a file.  
  
  
**Moving within a File**  
  
**Example**

1. #include<stdio.h>
2. #include<io.h>
3. #include<string.h>
4. #include<conio.h>
5. main()
6. {
7. char str\_write[]="abcdef";
8. int str\_read[10];
9. char filename[]="example.txt",O\_RDWR,O\_CREATE,O\_TRUNC;
10. int filedes;
11. //open the file, Truncate if it exists.
12. filedes=open(filename, O\_RDWR | O\_CREATE | O\_TRUNC);
13. //write 5 bytes of data
14. write (filedes, (char \*)str\_write,5);
15. //seek the beginning of the file
16. lseek(filedes,(long) str\_read,5);
17. //Terminate the data we have read with a null character
18. str\_read[5]='\o';
19. printf("The string is =%s.\n",str\_read);
20. close(filedes);
21. getch();
22. **return** 0;
23. }

Using Formatted I/O (Input-Output) Functions With Example in C

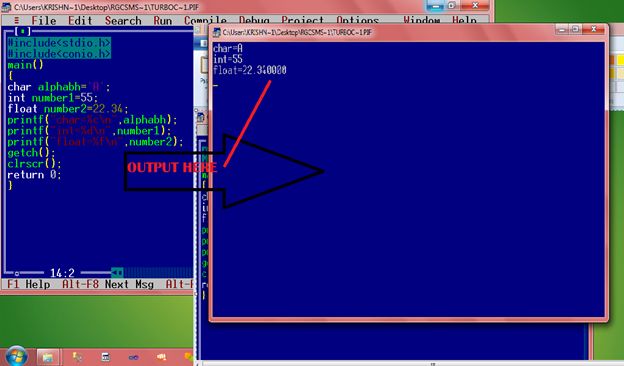
**Introduction To C**  
C provides standard functions scanf() and printf(), for performing formatted input and output .These functions accept, as parameters, a format specification string and a list of variables.  
  
The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.  
  
Now to discuss formatted output in functions.  
  
**Formatted Output**The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.  
 **Syntax:**  
printf (format, data1, data2,……..);  
  
In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.  
  
**Example:**printf (“%c”, data1);  
  
The character specified after % is called a conversion character because it allows one data type to be converted to another type and printed.  
  
See the following table conversion character and their meanings.

|  |  |
| --- | --- |
| Conversion Character | Meaning |
| d | The data is converted to decimal (integer) |
| c | The data is taken as a character. |
| s | The data is a string and character from the string , are printed until a NULL, character is reached. |
| f | The data is output as float or double with a default Precision 6. |

|  |  |
| --- | --- |
| Symbols | Meaning |
| \n | For new line (linefeed return) |
| \t | For tab space (equivalent of 8 spaces) |

**Example**  
printf (“%c\n”,data1);  
  
The format specification string may also have text.  
  
**Example**  
printf (“Character is:”%c\n”, data1);  
  
The text "Character is:" is printed out along with the value of data1.  
  
**Example with program**

1. #include<stdio.h>
2. #include<conio.h>
3. Main()
4. {
5. Char alphabh="A";
6. **int** number1= 55;
7. **float** number2=22.34;
8. printf(“**char**= %c\n”,alphabh);
9. printf(“**int**= %d\n”,number1);
10. printf(“**float**= %f\n”,number2);
11. getch();
12. clrscr();
13. retrun 0;
14. }
15. Output Here…
16. **char** =A
17. **int**= 55
18. flaot=22.340000

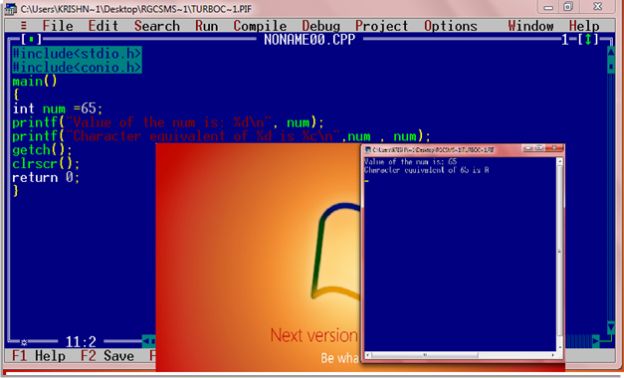
See the following image:  
  
  
  
**What is the output of the statement?**printf(“Integer is: %d; Alphabet is:%c\n”,number1, alpha);  
  
Where number1 contains 44 and alpha contains "Krishna Singh".  
  
Give the answer below.  
  
Between the character % and the conversion character, there may be:

* A minus sign: Denoting left adjustment of the data.
* A digit: Specifying the minimum width in which the data is to be output, if the data has a larger number of characters then the specified width occupied by the output is larger. If the data consists of fewer characters then the specified width, it is padded to the right or to the left (if minus sign is not specified) with blanks. If the digit is prefixed with a zero, the padding is done with zeros instead of blanks.
* A period: Separating the width from the next digit.
* A digit following the period: specifying the precision (number of decimal places for numeric data) or the maximum number of characters to be output.
* Letter 1: To indicate that the data item is a long integer and not an int.

|  |  |  |
| --- | --- | --- |
| Format specification string | Data | Output |
| |%2d| | 9 | |9| |
| |%2d| | 123 | |123| |
| |%03d| | 9 | |009| |
| |%-2d| | 7 | |7| |
| |%5.3d| | 2 | |002| |
| |%3.1d| | 15 | |15| |
| |%3.5d| | 15 | |0015| |
| |%5s| | “Output sting” | |Output string| |
| |%15s| | “Output sting” | |Output string| |
| |%-15s| | “Output sting” | |Output string| |
| |%15.5s| | “Output sting” | |Output string| |
| |%.5s| | “Output sting” | |Output| |
| |%15.5s| | “Output sting” | |Output| |
| |%f| | 87.65 | |87.650000| |
| |%.4.1s| | 87.65 | |87.71| |

Example based on the conversion character:

1. #include<stdio.h>
2. #include<conio.h>
3. main()
4. {
5. Int num=65;
6. printf(“Value of num **is** : %d\n:, num);
7. printf(“Character equivalent of %d **is** %c\n”, num , num);
8. getch();
9. clrscr();
10. rerurn o;
11. }

Output Here…  
char =A  
int= 55  
flaot=22.340000  
  
  
  
**Formatted Input**  
The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().  
  
**Syntax**scanf (format, num1, num2,……);  
The function scnaf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,….).  
  
For Example:  
  
scanf(“ %c %d”,&Name, &Roll No);  
  
**Note**: the data names are listed as &Name and &Roll No instead of Name and Roll No respectively. This is how data names are specified in a scnaf() function. In case of string type data names, the data name is not preceded by the character &.  
 **Example with program**  
Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.  
  
Solve here:

1. #include<stdio.h>
2. #include<conio.h>
3. main()
4. {
5. Int e\_num;
6. Float e\_wt;
7. printf (“Enter the Element No. and Weight of a Proton\n”);
8. scanf (“%d %f”,&e\_num, &e\_wt);
9. printf (“The Element No.**is**:”,e\_num);
10. printf (“The Weight of a Proton **is**: %f\n”, e\_wt);
11. getch();
12. **return** 0;
13. }

