Chapter: 6

**C's Console input/output & File input/output**

Details on #define, char, string i/o, printf, scanf, file i/o

**6.1 What is console? With brief intro :**

Getting data into the program is called input and sending the information out from the program is called output. Most of the compiler developers provide the common I/O functions with their implementations. Say for example same printf() function is used to print onto the console output device (monitor) but Turbo C has their way of printf() code to print on to the monitor and gcc has their way of printf() code to print on to the monitor. All the I/O functions are classified into

1. Console I/O
2. File I/O
3. Network I/O

Console I/O : A console refers to an interface through which users communicate with system programs of the operating system or with other console applications. The interactions consist of input operations from standard input device like keyboard and the text display on standard output ( usually on computer screen ). In C programming , in so many words: a console is a terminal from which you enter inputs and get outputs. Here the input device keyboard is called console input device and the output device monitor is called console output device. There are different I/O functions used to perform console I/O operations. These are basically classified into

1. Formatted functions : Formatted Console I/O functions are ***printf(), scanf(), sprint()*** and ***sscanf()*** functions used to print and read formatted data of almost any type.
2. Unformatted functions : C language provides number of unformatted functions like ***getchar(), getche(), getch(), putchar(), putch(), gets()*** and ***puts()*** to read and write characters on to the Console I/O.

***getchar():*** It is a line-buffered input function. Here the meaning of buffer is temporary storage. When we use ***getchar()***, all the characters we enter will be stored in the buffer until user shows confirmation by pressing enter. The characters stored in the buffer would be accessed by the ***getchar()*** one by one. It is generally used where a line of text need to be accessed character by character.

***getch() :*** Though it is also used to accept a character from the keyboard, functionally it is different from ***getchar()***. ***getch()*** doesn't echo (show) the typed character on the monitor and straightaway reads the character rather buffering.

***getche() :*** It is a similar function to ***getch()***, used to accept a character from the console input but slightly differs in its functionality. It reads the character directly without buffering but echoes (shows) the given character.

***putchar(), putch() :*** Both these functions are used to write a character onto console output. Though these accept integer as argument, we can also send a character as an argument. It returns the ASCII value of a printed character in return. It returns -1 (EOF) if anything goes wrong with the writing. ***putch()*** is not used in GCC.

***gets() :*** since ***scanf()*** could not access a line of text with spaces from the keyboard and store into the array, only the first word can be accepted and stored. The only way to accept a line of text from the keyboard including with spaces is ***gets().*** The ***gets()*** function accepts the address of any character array, stores the line of text accepted from the keyboard character by character and a terminating character ***'\0'*** is automatically added. It also returns the address of string accepted from the keyboard.

Take care while using gets(): Care must be taken while using gets() because it would not perform bound checking (checking the size) before storing the string into the character array. If user gives a string whose length is more than the size of array then it may overwrite other data rather giving error Hence ***gcc*** gives warning while compiling a C program with ***gets()***.

***puts() :*** It is used to print a string onto the console output; it accepts the address of character array or string as argument and prints character by character on the monitor until the end of character ***'\0'***. It returns -1 (EOF) if anything goes wrong while writing, returns a non-negative on successfully writing the data on to the monitor. One more good thing with the ***puts()*** is that, it automatically introduces a new line character.

**6.2 Macro substitution : #define**

The #define directive tells the preprocessor to perform a text substitution throughout your entire program. That is, it causes one sequence of characters to be replaced by another. This process is generally referred to as macro substitution. The general form of the #define statement is shown here:

**#define** macro-name character-sequence

Notice that this line does not end in a semicolon. Each time the macro-name is encountered in the program, the associated character-sequence is substituted for it. For example, consider this program:

#include<stdio.h>

#define MAX 100

**int** **main**(void){ **int** i;

**for**(i=O; i<MAX; i++) **printf**("%d", i);

**return** 0;}

When the identifier MAX is encountered by the preprocessor, 100 is automatically substituted. Actually the for loop to the compiler;

**for**(i=O; i<100; i++) **printf**("%d", i);

Keep in mind: At the time of the substitution, ***100*** is simply a string of characters composed of a **1** and two **0**'s. The preprocessor does not convert a numeric string into its internal binary format. This is left to the compiler.

1. The macro name can be any valid C identifier. But most programmers have adopted the convention of using uppercase for macro names. This makes it easy for identifying macro names in programs.
2. Preprocessor directives in general and #define in particular are not affected by C's code blocks. It can be defined outside of all functions or within a function, once it is defined, all code after that point may have access to it. For example,

**int** **main**(void){ #define MAX 100

**int** i;

**for**(i=O; i<MAX; i++) **printf**("%d", i);

**return** 0;}

1. Each preprocessor directive must appear on its own line. i.e. ***#define BIG 100 #define SMALL 0*** is not acceptable.
2. Macro substitutions are useful for two main reasons.
   1. First, to maintain a specific value : C library function's certain predefined values may vary between programming environments. Macro substitutions solve this problem.
   2. Second , to maintain programs – Easy to change values : For example, if you know that a value, such as an array size, is going to be used several places in your program, it is better to create a macro for this value. Then if you ever need to change this value, you simply change the macro definition. All references to it will be changed automatically when the program is recompiled.
3. Since a macro substitution is simply a text replacement, you can use a macro name in place of a quoted string. For example, the

#define FUN "Macro Substitutions are Fun"

**int** **main**(void){ **printf**(FUN); **return** 0; }

**6.3 Standard CONSOLE i/o : getchar(), putchar() & EOF**

EOF: when error occurs while reading/writing an input/output in the case of int returning type function , a negative value is returned (usually -1) it is called macro EOF. The EOF macro, defined in STDIO.H, stands for end-of-file. Since EOF is an integer value, to allow it to be returned, the responsible/corresponding function must return an integer.

The ANSI C standard defines these two functions prototype that perform character input and output, respectively;

***int getchar(void);*** /\*Return type is ***int***\*/

***int putchar(int ch);*** /\*Return type is ***int*** and also parameter type is ***int***\*/

They both use the header file STDIO.H.

int getchar(void); : Here getchar( ) is declared as returning an int. The getchar( ) function returns the next character typed on the keyboard. This character is read as an unsigned char convened to an int. Even though getchar( ) is declared as returning an int, but when program will assign this value to a char variable the high-order byte(s) of the integer is simply discarded. The reason that getchar( ) returns an integer is that when an error occurs while reading input, getchar( ) returns the macro EOF, which is a negative integer (usually -1). Since EOF is an integer value, to allow it to be returned, getchar( ) must return an integer.

* Many compilers implement getchar( ) in a line-buffered manner, which makes its use limited in an interactive environment.

int putchar(int ch); :Here putchar( ) is declared as returning an int , its parameter is declared to be of type int. The putchar( ) function outputs a single character to the screen . Although its parameter is declared to be of type int, this value is converted into an unsigned char by the function. Thus, only the low-order byte of ch is actually displayed. If the output operation is successful, putchar() returns the character written. If an output error occurs, EOF is returned.

Note

1. In the vast majority of circumstances, if an error occurs when reading from the keyboard, it means that the computer has ceased to function. Therefore, most programmers don't usually bother checking for EOF when using getchar( ). They just assume a valid character has been returned. Of course, there are circumstances in which this is not appropriate-for example, when I/O is redirected (as explained in Next Chapter).
2. If an output error occurs executing putchar( ), EOF is returned. For reasons similar to getchar( ), if output to the screen fails, the computer has probably crashed anyway, so most programmers don't bother checking the return value of putchar( ) for errors.
3. The reason to use putchar( ) rather than printf( ) with the %c specifier to output a character is that putchar( ) is faster and more efficient.
4. Line Buffering : getchar( ) is generally implemented using line buffering. When input is line buffered, no characters are actually passed back to the calling Program until the user presses ENTER. The following program demonstrates this:

#include <stdio.h>

#include <conio.h>

**int** **main**(void){ **char** ch;

**do** {ch = **getchar**( ) ; **putchar**('.') ;} **while**(ch != '\n');

**return** 0;}

Instead of printing a period between each character, what you will see on the screen is all the letters you typed before pressing ENTER, followed by a string of periods.

**do** {s\_ch = **getche**( ) ; **printf**(".") ;} **while**(s\_ch != '\r'); printing a period between each character.

1. /n and /r corresponds to std-char-functions and nonstd-char-functions : When entering characters using getchar( ), pressing ENTER will cause the newline character (\n) to be returned. However, when using one of the alternative non-standard functions, pressing ENTER will cause the carriage return character (\r) to be returned. Keep this difference in mind.

This program illustrates how ***getchar(), getche(), putchar(), printf()*** works: ***getchar()*** and ***getche(),*** uses different header files. In ***putchar()*** we used single quote ' and in printf() we used double quote ".

#include <stdio.h>

#include <conio.h>

**int** **main**(void){ **char** ch, s\_ch;

**do** {ch = getchar( ) ; putchar('.') ;} **while**(ch != '\n');

**printf**("\n\nHowever getche & printf works differently \n");

**do** {s\_ch = getche( ) ; printf(".") ;} **while**(s\_ch != '\r');

**return** 0;}

**6.4 NON-STANDARD CONSOLE FUNCTIONS : getche(), getch(), kbhit(), cprintf(), cscanf()**

Both ***getche()*** and ***getch()*** are non-standard functions of ***C***. Prototypes of ***getche()*** and ***getch()*** are :

***int getche(void);*** /\*Return type is ***int***\*/

***int getch(void);*** /\*Return type is ***int***\*/

Both functions use the header file CONIO.H and returns int. .

* The ***getche()*** function waits until the next keystroke is entered at the keyboard. When a key is pressed, ***getche()*** echoes(shows) it to the screen and then immediately returns the character. The character is read as an unsigned char and elevated to int. However, your routines (functions) can simply assign this value to a char value.
* The ***getch()*** function is the same as ***getche()***, except that the keystroke is not echoed(not showed) to the screen.
* Another very useful non-ANSI-standard function commonly supplied with a C compiler is ***kbhit()***. It has this prototype:

***int kbhit(void);*** /\*Return type is ***int***\*/

The ***kbhit()*** function also requires the header file CONIO.H. This function is used to determine whether a key has been pressed or not. When a key is pressed, kbhit() returns true (nonzero), but does not read the character (can be read it with ***getche()*** or ***getch()*** ). If no keystroke is pending, khhit( ) returns false (zero).

Problems of mixing standard I/O and non-standared I/O : For some compilers. the non-standard i/O functions such as ***getche()*** are not compatible with the standard i/O functions such as ***printf()*** or ***scanf().*** When this is the case, mixing the two can cause unusual program behavior. If mixing standard I/O and non-standared I/O is not compatible in your compiler, you may need to use non-standard versions of ***scanf()*** and/or ***printf(),*** too. These are called ***cprintf()*** and ***cscanf()*** . Both ***cprintf()*** and ***cscanf()*** use the CONIO.H header file.

* The ***cprintf()*** function works like ***printf()*** except that it does not translate the newline character (\n) into the carriage return, linefeed pair as does the ***printf()*** function. Therefore, It is necessary to explicitly output the carriage return (\r) where desired.
* The ***cscanf()*** function works like the ***scanf()*** function.

**6.5 Details on gets() and puts( )**

Their function prototypes of ***gets()*** and ***puts()*** are

***char \*gets(char \*str);*** /\*"pointer-function" returns ***char*** type & ***char*** type "pointer-parameter"\*/

***int puts(char \*str);*** /\*" function" returns ***int*** type & ***char*** type "pointer-parameter"\*/

These functions use the header file STDIO.H. Both do not perform bound checking (checking the size) before storing the string into the character array.

* The ***gets()*** function reads characters entered at the keyboard until a carriage return is read (i.e., until the user presses ENTER). It stores the characters in the array pointed to by str. The carriage return is not added to the string. Instead, it is converted into the null terminator. If successful, ***gets()*** returns a pointer to the start of str. If an error occurs, a null pointer is returned.
* The ***puts()*** function outputs the string pointed to by str to the screen. It automatically appends a carriage return, line-feed sequence (i.e. *there is no need to use new-line format* : each ***puts()*** output the corresponding string in different line). Example : **puts**("one"); **puts**("two") ; **puts**("three"); outputs the words one, two, and three on three separate lines. If successful, ***puts()*** returns a non-negative value. If an error occurs, EOF is returned.
  + The main reason to use ***puts()*** instead of ***printf(),*** to output a string is that ***puts()*** is much smaller and faster.

Note

* even though ***gets( )*** returns a pointer to the start of the string, it still must be called with a pointer to an actual array. For example, the following is wrong:

**char** \*p;

p = **gets**(p); /\* ***wrong!!!*** \*/

Here, there is no array defined into which ***gets()*** can put the string. This will result in a program failure.

**6.6 printf( ) : Details**

The ***printf()*** function has this prototype:

***int printf(char \*control-string, ... );***

* The periods indicate a variable-length argument list. The ***printf()*** function returns the number of characters output. If an error occurs, it returns a negative number. But if the console is not working, the computer is probably not functional anyway.
* The control string may contain two types of items: characters, to be output and format specifiers. The most used format specifiers are ***%c, %d, %f, %s, %u*** and ***%p***. The ***printf()*** format specifires are shown in the following table

|  |  |  |
| --- | --- | --- |
| Code | Format | Details |
| ***%c*** | Character |  |
| ***%d*** | Signed decimal integers |  |
| ***%i*** | Signed decimal integers | The %i command is the same as %d and is redundant. |
| ***%e*** | Scientific notation (lowercase 'e') | Display float using scientific notation in lowercase. Use L for long double. |
| ***%E*** | Scientific notation (uppercase 'E') | Display float using scientific notation in uppercase. Use L for long double. |
| ***%f*** | Decimal floating point |  |
| ***%g*** | Uses ***%e*** or ***%f***, whichever is shorter (lower case) | Use lowercase scientific/ normal whichever is shorter. Use L for long double. |
| ***%G*** | Uses ***%E*** or ***%f***, whichever is shorter (upper case) | Use uppercase scientific/normal whichever is shorter. Use L for long double. |
| ***%o*** | Unsigned octal | display an integer in octal format. |
| ***%s*** | String of characters |  |
| ***%u*** | Unsigned decimal integers |  |
| ***%x*** | Unsigned hexadecimal (lowercase letters) | display Hexadecimal letters ***'a'*** through ***'f'*** in lowercase. |
| ***%X*** | Unsigned hexadecimal (uppercase letters) | display Hexadecimal letters ***'a'*** through ***'f'*** in uppercase. |
| ***%p*** | Displays a pointer |  |
| ***%n*** | The associated argument is a pointer to an integer into which the number of characters written so far is placed. | The argument that matches the %n specifier must be a pointer to an integer. When the %n is encountered, ***printf()*** assigns the integer pointed to by the associated argument the number of characters output so far. |
| ***%%*** | Prints a % sign | Since all format specifires begins with percentage, so use %% |

Example 1 : This program prints the value 90 four different way: decimal, octal, lowercase hexadecimal, and uppercase hexadecimal. It also prints a floating-point number using scientific notation with a lowercase 'e' and an uppercase 'E' .

#include<stdio.h>

**int** **main**(void){

**printf** ("Decimal = %d, Octal = %o, Low-Hex = %x, Up-Hex = %X\n" , 90, 90, 90, 90);

**printf**("Low-sci = %e, Up-sci = %E\n", 99.231, 99.231); **return** 0;}

The output from this program is shown here:

90 132 5a 5A

9.92310e+01 9.92310E+01

**6.6.1 General form of format specifiers , minimum field width and precision specefier :**

Format specifiers : All format specifiers begin with %. A format specifier, also referred to as a format code , determines how its matching argument will be displayed. Format specifiers and their arguments are matched from left to right, and there must be as many arguments as there are specifiers.

* Minimum field-width : All specifiers "excluding %%, %p, %c" may have a minimum field-width specifier and/or a precision specifier associated with them. Both of these are integer quantities. If the item to output is shorter than the specified minimum field width, the output is padded with spaces, so that it equals the minimum width. However, if the output is longer than the minimum, output is nor truncated. The minimum-field-width specifier is placed after the % sign and before the format specifier. Eg : %10f tells printf() to output a double value of width 10.

The minimum-field-width specifier is especially useful for creating tables that contain columns of numbers that must line up. For example, this program prints 1000 random numbers in three columns.

#include<stdio.h>

#include<stdlib.h>

**int** **main**(void){**int** i;

**for**(i=0; i<100; i++)

**printf**("%10d %10d %10d\n", **rand**() , **rand**(), **rand**());

**return** 0;}

* Precision specifier : The precision specifier follows the minimum-field-width specifier. The two are separated by a period (i.e. "**.**"). The precision specifier affects different types of format specifiers differently. If it is applied to the ***%d, %i, %o, %u*** or ***%x*** specifiers, it determines how many digits are to be shown. Leading zeros are added if needed. When applied to ***%f***, ***%e***, or ***%E***, it determines how many digits will be displayed after the decimal point. For ***%g*** or ***%G***, it determines the number of significant digits. When applied to the ***%s***, it specifies a maximum field width (the maximum field specifire for ***scanf()*** and ***printf()*** are different . "**.**" is not used for ***scanf()***). If a string is longer than the maximum-field-width specifier, it will be truncated.

By default, all numeric output is right justified. To left justify output , put a minus sign directly after the % sign. The general form of a format specifier is shown here. Optional items are shown between brackets,

***%[-][minimum-field-width][.][precision] format-specifier***

for example , ***%15.2f*** tells ***printf()*** to output a double value using a field width of ***15***, with ***2*** digits after the decimal point.

Note

1. If you don't want to specify a minimum field width, you can still specify the precision. Simply put a period in front of the precision value.
2. The ***rand()*** function with STDLIB.H : Use C's standard library functions, ***rand()***, to generate the random numbers. The ***rand()*** function returns a random integer value each time it is called. It uses the header STDLIB.H.

**6.7 scanf( ) : Details**

The prototype for ***scanf()*** is:

***int scanf(char \*control-string, ... );***

The ***control-string*** consists mostly of format specifiers. However, it can contain other characters. The format specifiers determine how ***scanf()*** reads information into the variables pointed to by the arguments that follow the control string.

* The specifiers are matched in order, from left to right, with the arguments.
* There must be as many arguments as there are specifiers.

The ***scanf()*** function returns the number of fields assigned values. If an error occurs before any assignments are made, EOF is returned.

|  |  |  |
| --- | --- | --- |
| Code | Meaning | Details |
| ***%c*** | Read a single character |  |
| ***%d*** | Read a decimal integer |  |
| ***%i*** | Read a decimal integer |  |
| ***%e*** | Read a floating-point number |  |
| ***%f*** | Read a floating-point number |  |
| ***%g*** | Read a floating-point number |  |
| ***%o*** | Read an octal number | used to read an unsigned integer using octal bases |
| ***%s*** | Read a string |  |
| ***%x*** | Read a hexadecimal number | used to read an unsigned integer using hexadecimal bases |
| ***%p*** | Read a pointer | inputs a memory address using the format determined by the host environment |
| ***%n*** | Receives an integer value equal to  the number of characters read so far | assigns the number of characters input up to the point the ***%n*** is encountered to the integer variable pointed to by its matching argument, ***l*** or ***h*** applied for long or short . |
| ***%u*** | Read an unsigned integer |  |
| ***%[]*** | Scan for a set of characters |  |

scanset ***%[]*** : A scanset specifier is created by putting a list of characters inside square brackets, for example, here is a scanset specifier containing the letters 'ABC' :  ***%[ABC]*** */\*reads A,B,C from input\*/*

* When ***scanf()*** encounters a scanset, it begins reading input into the character array pointed to by the scanset's matching argument. It will only continue reading characters as long as the next character is part of the scanset.
* As soon as a character that is not part of the scanset is found, ***scanf()*** stops reading input for this specifier and moves on to any others in the control string.
* You can specify a range in a scanset using the **-** (hyphen), for example, this scanset specifies the characters 'A' through 'Z',

***%[A-Z]***

Technically, the use of the hyphen to specify a range is not specified by the ANSI C standard, but it is nearly universally accepted.

When the scanset is very large, sometimes it is easier to specify what is not part of a scanset, To do this, precede the set with a **^** , for example, ***%[^0123456789]*** When ***scanf()*** encounters this scanset, it will read any characters except the digits 0 through 9.

Here’s an example of a scanset that accepts both the upper- and lowercase characters (but no spaces).

**char** str[80];

**printf**("Enter letters, anything else to stop\n");

**scanf**("%[a-zA-Z]", str); **printf**(str);

Following code is slightly changed with a whitespace " " inside ***[]*** , it allows white space between characters to be read by ***scanf()***.

**char** edstr[80];

/\*Admits space\*/

**printf**("Enter letters with space, anything else to stop\n");

**scanf**("%[a-zA-Z ]", edstr); **printf**(edstr); */\*added white space inside []\*/*

* You could also specify punctuation symbols and digits inside **scanf**("%[a-zA-Z ]", edstr);, so that you can read virtually any type of string.

Use of **%\* :** you can suppress(neglect) the assignment of a field by putting an asterisk "**\***" immediately after the ***%*** sign. This can be very useful when inputting information that contains needless characters. (eg : input password containing "**-**" : ***abc-3nh-th7***). For example, given this ***scanf()*** statement

**int** first , second;

**scanf**( "%d%\*c%d", &first. &second);

inputting : 555-2345, then scanf() will read as first=555 and second=2345, the "**-**" will be skipped.

will cause ***scanf()*** to assign ***555*** to first, discard the ***-***, and assign ***2345*** to second. Since the hyphen is not needed, there is no reason to assign it to anything. Hence, no associated argument is supplied.

Notes

1. The specifiers ***%d, %i, %u, %x***, and ***%o*** (integer type specifiers) may be modified by the ***h*** when inputting into a short variable and by ***l*** when inputting into a long variable.
2. The specifiers ***%e. %f,*** and ***%g*** are equivalent. They all read floating-point numbers represented in either scientific notation or standard decimal notation . To read a double, modify them with ***l***. To read a long double, modify them with ***L***.
3. Limitation of ***scanf()***to read strings : You can use ***scanf()*** to read a string using the ***%s*** specifier, but when ***scanf()*** inputs a string, it stops reading that string when the first whitespace character is encountered (i.e. ***scanf()*** can read a word, not whole sentence). A whitespace character is either a space, a tab, or a newline, This means that you cannot easily use ***scanf()*** to read input like this into a string:

this is one string

Because there is a space after "this," ***scanf()*** will ,stop inputting the string at that point. This is why ***gets()*** is generally used to input strings.

1. Line-Buffering : As ***scanf()*** is generally implemented, it line-buffers input in the same way that ***getehar()*** often does. While. this makes little difference when inputting numbers, its lack of interactivity tends to make ***scanf()*** of limited value for other types of input.
2. You can specify a maximum field width for all specifiers except ***%c*** (for which a field is always one character) and ***%n*** (to which the concept does not apply). The maximum field width is specified as an unsigned integer, and it immediately precedes the format specifier character. For example, this limits the maximum length of a string assigned to str to 3 characters: ***scanf("%3s", str);***

**int** i, j;

**printf**("Enter an integer: "); **scanf**("%3d%d", &i, &j);

**printf**("%d %d", i, j);

This illustrates the maximum-field-width specifier: If ***12345*** is entered, ***i*** will be assigned ***123***, and ***j*** will have the value ***45***. The reason for this is that ***scanf()*** is told that ***i's*** field is only three characters long. The remainder of the input is then sent to ***j***.

1. If a space appears in the control string, then ***scanf()*** will begin reading and discarding whitespace characters until the first non-whitespace character is encountered. For example: This program lets the user enter a number followed by an operator followed by a second number, such as 12 + 4. It then performs the specified operation on the two numbers

#include<stdio.h>

**int** **main**(void){ **int** i, j; **char** op;

**printf**("Enter operation: "); **scanf**("%d %c %d", &i, &op, &j);

**switch**(op){ **case** '+': **printf**("%d", i+j); **break**;

**case** '-': **printf**("%d", i-j); **break**;

**case** '/': **if**(j) **printf**("%d",i/j); **break**;

**case** '\*': **printf**("%d", i\*j);}

**return** 0;}

Notice that the format for entering the information **scanf("%d %c %d", &i, &op, &j);** contains whitespace between two format specifires, this tricks admits whitespace between numbers and operation (here the white space is skipped by ***scanf()***)

1. If any other character appears in the control string, ***scanf()*** reads and discards all matching characters until it reads the first character that does not match that character .

#include<stdio.h>

**int** **main**(void){ **int** i, j;

**printf**("Enter a decimal number: "); **scanf**("%d.%d", &i, &j);

**printf**("left part: %d, right part: %d", i, j);

**return** 0;}

This program illustrates the effect of having non-whitespace characters in the control string. It allows you to enter a decimal value, but it assigns the digits to the left of the decimal point to one integer and those to the right of the decimal to another. The decimal point "**.**" between the two %d specifiers causes the decimal point in the number to be matched and discarded.

**6.8 STREAMS for file I/O in C**

The stream and the file : The C I/O system supplies a consistent interface to the programmer, independent of the actual I/O device being used. To accomplish this, C provides a level of abstraction between the programmer and the hardware. This abstraction is called a stream.

* The actual device providing I/O is called a file. A file is the actual physical entity that receives or supplies the data. As C defines the term file, it can refer to a disk file, the screen, the keyboard, memory, a port, a file on tape, and various other types of I/O devices. The most common form of file is, of course, the disk file.
* In C, disk I/O (like certain other types of I/O) is performed through a logical interface called a stream. All streams have similar properties, and all are operated on by the same I/O functions, no matter what type of file the stream is associated with. A stream is a logical interface to a file. Although files differ in form and capabilities, all streams are the same. The advantage to this approach is that to the programmer, one hardware device will look much like any other. The stream automatically handles the differences. A stream is linked to a file using an open operation. A stream is disassociated from a file using a close operation.
* There are two types of streams: text and binary.
* A text stream contains ASCII characters. When a text stream is being used, some character translations may take place. For example, when the newline character is output, it is usually converted into a carriage return, linefeed pair. For this reason, there may not be a one-to-one correspondence between what is sent to the stream and what is written to the file (ASCII-text).
* A binary stream may be used with any type of data. No character translations will occur, and there is a one-to-one correspondence between what is sent to the stream and what is actually contained in the file.

Current location : One final concept you need to understand is that of the current location. The current location, also referred to as the current position, is the location in a file where the next file access will occur. For example, if a file is 100 bytes long and half the file has been read, the next read operation will occur at byte 50, which is the current location.

**Details on STREAM**

Conceptually, the C program deals with a stream instead of directly with a file. A stream is an idealized flow of data to which the actual input or output is mapped. That means various kinds of input with differing properties are represented by streams with more uniform properties. The process of opening a file then becomes one of associating a stream with the file, and reading and writing take place via the stream.

Streams : Streams enable the device-independent input and output. Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported: text streams and binary streams.

text streams : A text stream consists of one or more lines. A line in a text stream consists of zero or more characters plus a terminating new-line character. That's what any program expects when it reads text, and that's what any program produces when it writes text. The string of characters that go into, or come out of a text stream may have to be modified to conform to specific conventions. This results in a possible difference between the data that go into a text stream and the data that come out (not one-one correspondent). For instance, in some implementations when a space-character precedes a new-line character in the input, the space character gets removed out of the output. In general, when the data only consists of printable characters and control characters like horizontal tab and new-line, the input and output of a text stream are equal.

binary streams : Compared to a text stream, a binary stream is pretty straight forward. A binary stream is an ordered sequence of characters that can transparently record internal data. Data written to a binary stream shall always equal the data that gets read out under the same implementation (one-one correspondence). Binary streams, however, may have an implementation-defined number of null characters appended to the end of the stream.

Note

1. In C all input and output is done with streams.
2. Stream is nothing but the sequence of bytes of data.
3. A sequence of bytes flowing into program is called input stream.
4. A sequence of bytes flowing out of the program is called output stream.
5. Use of Stream make I/O machine independent.
6. Predefined Streams:

|  |  |
| --- | --- |
| stdin | Standard Input |
| stdout | Standard Output |
| stderr | Standard Error |

1. Standard Input Stream Device :

* stdin stands for (Standard Input)
* Keyboard is standard input device .
* Standard input is data (Often Text) going into a program.
* The program requests data transfers by use of the read operation.
* Not all programs require input.

1. Standard Output Stream Device :

* stdout stands for (Standard Output)
* Screen(Monitor) is standard output device .
* Standard output is data (Often Text) going out from a program.
* The program sends data to output device by using write operation.

Difference Between Std. Input and Output Stream Devices

|  |  |  |
| --- | --- | --- |
| Point | Std i/p Stream Device | Standard o/p Stream Device |
| Stands For | Standard Input | Standard Output |
| Example | Keyboard | Screen/Monitor |
| Data Flow | Data (Often Text) going into a program | data (Often Text) going out from a program |
| Operation | Read Operation | Write Operation |

Some Important Summary:

|  |  |  |
| --- | --- | --- |
| Point | Input Stream | Output Stream |
| Standard Device 1 | Keyboard | Screen |
| Standard Device 2 | Scanner | Printer |
| IO Function | ***Scanf()*** and ***gets()*** | ***Printf()*** and ***puts()*** |
| IO Operation | Read | Write |
| Data | Data goes from stream | data comes into stream |

**6.9 File access using fopen(), fclose() and read/write using fgetc(), fputc()**

Opening a file : To open a file and associate it with a stream, use ***fopen()*** it uses STDIO.H. Its prototype is shown here:

***FILE \*fopen(char \*fname, char \*mode);***

Generally we use the following form

***FILE \*file\_pointer;***

***file\_pointer=fopen("file\_name", "mode");***

IN the prototype the name of the file to open is pointed to by fname. It must be a valid file name, as defined by the operating system. The string pointed to by mode determines how the file may be accessed. ANSI C standard values for mode are shown in Table.

* The type FILE is defined in STDIO.H. It is a structure that holds various kinds of information about the file, such as its size, the current location of the file, and its access modes. It essentially identifies the file. (*A structure is a group of variables accessed under one name.*)
* If the open operation is successful, **fopen()** returns a valid file pointer. The ***fopen()*** function returns a pointer to the structure associated with the file by the open process. This pointer will be used with all other functions that operate on the file. It can't be altered or the object it points to.
* If the fopen( ) function fails, it returns a null pointer. The header STDlO.H defines the macro NULL, which is defined to be a null pointer. It is very important to ensure that a valid file pointer has been returned. To do so, check the value returned by fopen( ) (using condition) to make sure that it is not NULL. for example, the proper way to open a file called my file for text input :

**FILE** \*fp;

**if**( *(fp =* ***fopen****("myfile", "r"))* == **NULL**){

**printf**(“Error opening file.\n");

**exit**(1); /\* or substitute your own error handler \*/ }

what actually happening here(CONFUSED !!!) : Here not only the "file error checking " happening inside the condition of "if" statement but also file is opened (if it exists) simultaneously. It is actually equivalent to

fp=**fopen**("myfile", "r"); /\*file opening \*/

**if**(fp===**NULL**){ /\*file error checking\*/

**printf**("file error\n");

**exit**(1); /\* less effect of **exit**() : file error occurs before the conditional statement\*/}

There is no need for a separate comparison step because the assignment and the comparison can be performed at the same time within the if .

* In this case there is no point of using ***exit(),*** after the file error occurred.
* To prevent error of file opening if the file doesn't exist, the ***fopen()*** should occur inside the "if" statement. The whole program will exit (shuts down) if the file doesn't exist using ***exit(1)*** against the "NULL" condition. And no error occurs during file closing (***fclose()***) or any crash.
* Inside the "if" condition the file is opened simultaneously during the "file error checking". But once the file is opened it never close until fclose appeared to corresponding file pointer.
* The point of using ***exit()*** is that when the proper condition occurs for ***exit(1)*** the program will shuts down immediately by returning ***0*** to the operating system. Her the point of using ***exit()*** is that : ***fclose()*** creates system error if its pointer argument is invalid on a null-pointer. Hence when a null-pointer occurs during file operation the program will shuts down.

|  |  |  |  |
| --- | --- | --- | --- |
| Mode | Meaning | Mode | Meaning |
| ***"r"*** | Open a text file for reading | ***"r+"*** | Open a text file for read/write |
| ***"w"*** | Create a text file for writing | ***"w+"*** | Create a text file for read/write |
| ***"a"*** | Append(means : edit/add) to a text file | ***"a+"*** | Append /create a text file for read/write |
| ***"rb"*** | Open a binary file for reading | ***"r+b"*** or ***"rb+"*** | Open a binary file for read/write. Also can use "rb+" |
| ***"wb"*** | Create a binary file for writing | ***"w+b"*** or ***"wb+"*** | Create a binary file for read/write. Also can use "wb+" |
| ***"ab"*** | Append to a binary file | ***"a+b"*** or ***"ab+"*** | Append/create a binary file for read/write. Also can use "ab+" |

Although most of the file modes are self-explanatory, a few comments are in order :

1. "r": when opening a file for read-only operations, the file does not exist, fopen() will fail and NULL-pointer will return.
2. "a": When opening a file using append mode, if the file does not exist, it will be created. Further, when a file is opened for append all new data written to the file will be written to the end of the file. The original contents will remain unchanged.
3. "w": If, when a file is opened for writing, the file does not exist, it will be created. If it does exist, the contents of the original file will be destroyed and a new file created.
4. "r+": It is similar to "r", "r+" will not create a file if it does not exist. It can both read-write.
5. "a+": It is similar to "a", moreover we can read with this mode. IT can both read-append.
6. "w+": similar to "w". It can both read-write. Seems like "r+" & "w+" are same , but there are few differences.
7. Difference between "r+" & "w+": The difference between modes "r+" and "w+" is that "r+" will not create a file if it does not exist; however, "w+" will. Further, if the file already exists, opening it with "w+" destroys its contents; opening it with "r+" does not.

Note : For general purpose use only "r+" and "a+" for read-write-append but be careful using "w"/"w+"

Opening a file : To close a file and disassociate it with a stream, use ***fclose()*** it uses STDIO.H. Its prototype is shown here:

***int fclose(FILE \*fp);***

The ***fclose()*** function closes the file associated with fp, which must be a valid file pointer previously obtained using ***fopen()***, and disassociates the stream from the file.

* Be carefull with using fclose() : You must never call ***fclose()*** with an invalid argument. Doing so will damage the file system and possibly cause irretrievable data loss. Invalid arguments means : you cannot use fclose() with null or empty file pointer or invalid-error causing file pointer (i.e. can't use null-pointer ). Never use fclose before fopen with corresponding file.
* The ***fclose()*** function returns zero if successful. If an error occurs, EOF is returned.
* Flushing the buffer : In order to improve efficiency, most file system implementations write data to disk one sector at a time. Therefore, data is buffered until a sector's worth of information has been output before the buffer is physically written to disk. When you call fclose(), it automatically writes any information remaining in a partially full buffer to disk. This is often referred to as flushing the buffer.

Reading and Writing from/to a files : Once a file has been opened, depending upon its mode, you may read and/or write bytes (i.e., characters) using these two functions:

***int fgetc(FILE \*fp);***

***int fputc(int ch, FILE \*fp);***

* fgetc() : The ***fgetc()*** function reads the next byte from the file described by fp as an unsigned char and returns it as an integer. (The character is returned in the low-order byte.) If an error occurs, ***fgetc()*** returns EOF (int type ***EOF=-1***). The ***fgetc()*** function also returns EOF (i.e ***-1***) when the end of the file is reached. Although ***fgetc()*** returns an integer value, your program can assign it to a char variable since the low-order byte contains the character read from the file.
* fputc() : The ***fputc()*** function writes the byte contained in the low-order byte of ch to the file associated with fp as an unsigned char. Although ch is defined as an int, you may cal1 it using a char, which is the common procedure. The ***fputc()*** function returns the character written if successful or EOF if an error occurs.

Historical note

The traditional names for ***fgetc()*** and ***fputc()*** are ***getc )*** and ***putc()***. The ANSI C standard still defines these names, and they are essential1y interchangeable with ***fgetc()*** and ***fputc().*** One reason the new names were added was for consistency. Al1 other ANSI file system function names begin with ***"f"***: so ***"f"*** was added to ***getc()*** and ***putc()***.

Example : writing and reading a file.

#include<stdio.h>

#include<stdlib.h>

**int** **main**(void){**char** str[80]="Yo babbay. Fuck the file sys!! Aye?";

**FILE** \*f\_point;

**char** \*p;

**int** i;

/\*open the file for output\*/

**if**((f\_point=**fopen**("myfile", "w"))==**NULL**){**printf**("File-Error\n"); **exit**(1);}

/\*write into the file\*/

p=str;

**while**(\*p){**if**((**fputc**(\*p, f\_point)==**EOF**)){**printf**("Write-Error\n"); **exit**(1);}

p++; }

**fclose**(f\_point);

/\*Open the file for the input\*/

**if**((f\_point=**fopen**("myfile", "r"))==**NULL**){**printf**("Opening-Error"); **exit**(1);}

/\*read from file and output\*/

**for**(;;){i=**fgetc**(f\_point);

**if**(i==**EOF**) **break**;

**putchar**(i);}

**fclose**(f\_point);

**return** 0;}

* However ***while(\*p){if((fputc(\*p, f\_point)==EOF)){printf("Write-Error\n"); exit(1);} p++; }*** can be written as :

**while**(\*p) **if**((**fputc**(\*p++, f\_point)==**EOF**)){**printf**("Write-Error\n"); **exit**(1);}

this is the ability of integrating operations, it makes C most powerful.

* In this version, when reading from the file, the return value of ***fgetc()*** is assigned to an integer variable called ***i***

**for**(;;){i=**fgetc**(f\_point); **if**(i==**EOF**) **break**; **putchar**(i);}

The value of this integer is then checked to see if the end of the file has been reached.

For most compilers, however, you can simply assign the value returned by ***fgetc()*** to a char and still check for EOF, as shown in the following fragment:

**Char** ch;

**for**(;;){ch=**fgetc**(f\_point); **if**(ch==**EOF**) **break**; **putchar**(i);}

The reason this approach works is that when a char is being compared to an int (the EOF , which is -1), the char value is automatically elevated to an equivalent int value.

* There is no need for a separate comparison step because the assignment and the comparison can be performed at the same time within the if (Exactly what we did before for opening and closing a file), as shown here:

**for**(;;) { **if**((ch = **fgetc**(f\_point)) == **EOF**) **break**; **putchar**(ch); }

Don't let the statement ***if((ch = fgetc(f\_point)) == EOF)*** fool you. Here’s what is happening. First, inside the if, the return value of ***fgetc()*** is assigned to ch. As you may recall, the assignment operation in C is an expression. The entire value of ***(ch = fgetc(fp))*** is equal to the return value of ***fgetc( ).*** Therefore, it is this integer value that is tested against EOF.

* those fragments written by a professional C programmer as follows:

**while**((ch = **fgetc**(f\_point)) != **EOF**) **putchar**(ch);

Notice that now, each character is read, assigned to ch, and tested against EOF, all within the expression of the while loop that controls the input process. If you compare this with the original version, you can see how much more efficient this one is. In fact, the ability to integrate such operations is one reason C is so powerful. Later we will explore such assignment statements more fully.

**6.10 End of file [EOF] *feof()* And file error checking *ferror()***

As you know, when ***fgetc()*** returns EOF,

1. either an error has occurred
2. or the end of the file has been reached ,

but how do you know which event has taken place? Further if you are operating on a binary file, all values are valid. This means it is possible that a byte will have the same value (when elevated to an int) as EOF, so how do you know if valid data has been returned or if the end of the file has been reached?

The solution to these problems are the functions ***feof()*** and ***ferror()***, whose prototypes are shown here:

***int feof(FILE \*fp);***

***int ferror(FILE \*fp);***

***feof() :*** The ***feof( )*** function returns nonzero if the file associated with fp has reached the end of the file [literally End-Of-File]. Otherwise it returns zero. This function works for both binary files and text files.

***ferror() :*** The ***ferror()*** function returns nonzero if the file associated with fp has experienced an error. Otherwise, it returns zero.

Using the ***feof()*** function, this code fragment shows how to read to the end of a file:

**FILE** \*fp;

. . .

**while**(!**feof**(fp))ch = **fgetc**(fp);

This code works for any type of file and is better in general than checking for EOF. However, it still does not provide any error checking. Error checking is added here:

**FILE** \*fp;

. . .

. . .

**while**(!**feof**(fp)) { ch = **fgetc**(fp);

**if**(**ferror**(fp)) {**printf**("File Error\n"); **break**;}

}

ferror is inside feof because it checks for file error if feof returns zero. Here it works like, fgetc works until end-of-file is reached and during all time ferror works inside the while. (confused).

Note

* Keep in mind that ***ferror()*** only reports the status of the file system relative to the last file access. Therefore, to provide the fullest error checking, you must call it after each file operation.
* Often the only types of errors that actually get passed back to your program are those caused by mistakes on your part, such as accessing a file in a way inconsistent with the mode used to open it or when you cause an out-of-range condition. Usually these types of errors can be trapped by checking the return type of the other file system functions rather than by calling ***ferror()***. For this reason, you will frequently see examples of C code in which there are relatively few (if any) calls to ***ferror()***.

**6.11 String I/O in a File with *fputs()* & *fgets()*. Text I/O with *fprintf()* & *fscanf()***

We use these two functions ***fputs()*** and ***fgets()*** , which write a string to and read a string from a file, respectively. Their prototypes are

***int fputs(char \*str, FILE \*fp);***

***char \*fgets(char \*str, int num, FILE \*fp);***

***fputs() :*** The ***fputs()*** function writes the string pointed to by str to the file associated with fp. It returns EOF if an error occurs and a non-negative value if successful. The null that terminates str is not written. Also, unlike its related function ***puts()*** it does not automatically append a carriage return, linefeed pair. Example : **fputs**(str\_1, f\_point);

***fgets()*** : The ***fgets()*** function reads characters from the file associated with fp into the string pointed to by str until " (*one less than the string length number*) characters have been read, a newline character is encountered, or the end of the file is reached. In any case, the string is null-terminated. Unlike its related function ***gets()***, the newline character is retained. The function returns str if successful and a null pointer if an error occurs. Example : **fgets**(str\_1, 79, f\_point); /\* 0 to 79 = 80 \*/

***fprintf () & fscanf()*** : The C file system contains two very powerful functions ***fprintf()*** and ***fscanf()*** similar to ***printf()*** and ***scanf()***. These functions operate exactly like ***printf()*** and ***scanf()*** except that they work with files. Their prototypes are:

***int fprintf(FILE \*fp, char \*control-string, … );***

***int fscanf(FILE \*fp, char \* control-string, … );***

Instead of directing their I/O operations to the console, these functions operate on the file specified by fp. Otherwise their operations are the same as their console-based relatives. The advantage to ***fprintf()*** and ***fscanf()*** is that they make it very easy to write a wide variety of data to a file using a text format.

**6.12 READ AND WRITE BINARY DATA**

We discussed earlier that how useful ***fprintf()*** and ***fscanf()*** are , but they have some problems :

* They are not necessarily the most efficient way to read and write numeric data. Because both functions perform conversions on the data. For example, when you output a number using ***fprintf()*** the number is converted from its binary format into ASCII text. Conversely, when you read a number using ***fscanf()*** , it must be converted back into its binary representation. For many applications, this conversion time will not be meaningful; for others, it will be a severe limitation.
* Further, for some types of data, a file created by ***fprintf()*** will also be larger than one that contains a mirror image of the data using its binary format.

For these reasons, the C me system includes two important functions: ***fread()*** and ***fwrite()***. These functions can read and write any type of data, using its binary representation. Their prototypes are (with four parameters)

***size\_t fread(void \*buffer, size\_t size, size\_t num, FILE \*fp);***

***size\_t tw\*rite(void \*buffer, size\_t size, size\_t num, FILE \*fp);***

***fread()*** : Description of the four parameters

1. The ***fread()*** function reads from the file associated with fp,
2. num number of objects,
3. each object size bytes long,
4. into the buffer pointed to by buffer.

It returns the number of objects actually read. If this value is less than num, either the end of the file has been encountered or an error has occurred. You can use ***feof()*** or ***ferror()*** to find out which.

***fwrite()*** : The ***fwrite()*** function is the opposite of ***fread().***Description of the four parameters

1. ***fwrite()*** writes to the file associated with fp,
2. num number of objects,
3. each object size bytes long,
4. from the buffer pointed to by buffer.

It returns the number of objects written. This value will be less than num only if an output error has occurred.

void pointer : A void pointer is a pointer that can point to any type of data without the use of a type cast. This is generally referred to as a generic pointer. In C, void pointers are used for two primary purposes.

1. First, as illustrated by ***fread()*** and ***fwrite()***, they are a way for a function to receive a pointer to any type of data without causing a type mismatch error. [*As stated earlier, fread( ) and fwrite( ) can be used to read or write any type of data*]. Therefore, the functions must be capable of receiving any sort of data pointed to by buffer. void pointers make this possible.
2. A second purpose they serve is to allow a function to return a generic pointer.

type size\_t : size\_t is a type which is defined in the STDIO.H header file. (We'll learn how to define types later ). A variable of this type is defined by the ANSI C standard as being able to hold a value equal to the size of the largest object supported by the compiler. For our purposes, you can think of size\_t as being the same as unsigned or unsigned long. The reason that size\_t is used instead of its equivalent built-in type is to allow C compilers running in different environments to accommodate the needs and confines of those environments.

Example : The following program writes an integer to a file called MYFILE\_BI using its internal, binary representation and then reads it back. (The program assumes that integers are 2 bytes long.)

#include<stdio.h>

#include<stdlib.h>

**int** **main**(void){**FILE** \*f\_point;

**int** i, k;

/\*open the file for output\*/

**if**((f\_point=**fopen**("myfile\_bi", "**wb**"))==**NULL**){**printf**("File-Error\n"); **exit**(1);}

i=100; ***/\* value written through i\*/***

/\*write into the file and ***using "!=1" instead of "==EOF" for error checking*** \*/

**if**((**fwrite**(&i, 2, 1, f\_point)!=1)){**printf**("Write-Error\n"); **exit**(1);}

**fclose**(f\_point);

/\*Open the file for the input\*/

**if**((f\_point=**fopen**("myfile\_bi", "**rb**"))==**NULL**){**printf**("Opening-Error"); **exit**(1);}

/\*read from file and output\*/

**if**((**fread**(&k, 2, 1, f\_point)!=1)){**printf**("Read-Error\n"); **exit**(1);}

**printf**(" i is %d ", k); ***/\* value read through k\*/***

**fclose**(f\_point);

**return** 0;}

Notice how error checking is easily performed in this program by simply comparing the number of items written or read with that requested. But in some situations, however, you will still need to use ***feof()*** or ***ferror()*** to determine if the end of the file has been reached or if an error has occurred.

The ***sizeof()*** keyword & its use : One thing wrong with the preceding example is that an assumption about the size of an integer has been made and this size is hardcoded into the program. Therefore, *the program will not work properly with compilers that use 4-byte integers* [More generally, the size of many types of data changes between systems or is difficult to determine manually] .

For this reason, C includes the keyword sizeof, which is a compile-time operator that returns the size , in bytes, of a data type or variable. It takes the general forms :

***sizeof(type)*** or ***sizeof var\_name***

For example, if floats are four bytes long and f is a float variable, both of the following expressions evaluate to 4:

**sizeof f** or **sizeof(float)**

* When using sizeof with a type, the type must be enclosed between parentheses. No parentheses are needed when using a variable name, although the use of parentheses in this context is not an error.

Example : An improved version of the preceding program is shown here, using sizeof.

#include<stdio.h>

#include<stdlib.h>

**int** **main**(void){**FILE** \*f\_point;

**int** i, k;

/\*open the file for output / append\*/

**if**((f\_point=**fopen**("myfile\_4", "**ab+**"))==**NULL**){**printf**("File-Error\n"); **exit**(1);}

i=400; ***/\* value written through i\*/***

/\*write into the file and ***using "!=1" instead of "==EOF" for error checking*** \*/

**if**((**fwrite**(&i, **sizeof**(**int**), 1, f\_point)!=1)){**printf**("Write-Error\n"); **exit**(1);}

**fclose**(f\_point);

/\*Open the file for the input\*/

**if**((f\_point=**fopen**("myfile\_4", "**rb**"))==**NULL**){**printf**("Opening-Error"); **exit**(1);}

/\*read from file and output\*/

**if**((**fread**(&k, **sizeof** k, 1, f\_point)!=1)){**printf**("Read-Error\n"); **exit**(1);}

**printf**(" i is %d ", k); ***/\* value read through k\*/***

**fclose**(f\_point);

**return** 0;}

Note

* When using ***fread()*** or ***fwrite()*** to input or output binary data, the file must be opened for binary operations. Forgetting this can cause hard-to-find problems.
* By using sizeof, not only do you save yourself the drudgery of computing the size of some object by hand, but you also ensure the portability of your code to new environments.

**6.13 Random access using *fseek()***

Above we discussed about write or read a file sequentially from its beginning to its end using ***fgetc(), fputc(), fputs(), fgets(), fprintf(), fscanf(), fread()*** and ***fwrite()***.

To access a file randomly (i.e. any where of a file ): Using ***fseek()*** we can access any point in a file at any time. It is another of C's file system functions. The prototype of ***fseek()*** is

***int fseek(FILE \*fp, long offset, int origin);***

* The ***fseek()*** function returns zero when successful and nonzero if a failure occurs. In most implementations, you may seek past the end of the file, but you may never seek to a point before the start of the file.
* Here, fp is associated with the file being accessed. The value of offset determines the number of bytes from origin to make the new current position. origin must be one of these macros, shown here with their meanings:

|  |  |
| --- | --- |
| Origin | Meaning |
| ***SEEK\_SET*** | Seek from start of file |
| ***SEEK\_CUR*** | Seek from current location |
| ***SEEK\_END*** | Seek from end of file |

These macros are defined in STDIO.H. For example, if you wanted to set the current location 100 bytes from the start of the file, then origin will be SEEK\_SET and offset will be 100.

Determine the current location using ***ftell()*** : You can determine the current location of a file using ***ftell()***. It is another of C's file system functions. Its prototype is

***long ftell(FILE \*fp);***

It returns the location of the current position of the file associated with fp. If a failure occurs, it returns ***-1***.

Notes

1. In general, use random access only on binary files. Because text files may have character translations performed on them, there may not be a direct correspondence between what is in the file and the byte to which it would appear that we want to seek.
2. The only time you should use ***fseek()*** with a text file is when seeking to a position previously determined by ***ftell(),*** USing ***SEEK\_SET*** as the origin.
3. Even a file that contains only text - can be opened as a binary file, if you like. There is no inherent restriction about random access on files containing text. The restriction applies only to files opened as text files.

**6.14 Some other important File-System functions**

Rename a file : To rename a file use ***rename()***, shown here:

***int rename(char \*oldname, char \*newname);***

Here, oldname points to the original name of the file and newname points to its new name. The function returns zero if successful and nonzero if an error occurs.

Erase a file : To erase a file use ***remove()***, Its prototype is

***int remove(char \*file-name);***

This function will erase the file whose name matches that pointed to by file-name. It returns zero if successful and nonzero if an error occurs.

Position change : To position a file's current location to the start of the file use ***rewind()***. Its prototype is

***void rewind(FILE \*fp);***

It rewinds the file associated with fp. The ***rewind()*** function has no return value, because any file that has been successfully opened can be rewound.

Flush disk buffer : To cause a file's disk buffer to be flushed use ***fflush()***. Its prototype is

***int fflush(FILE \*fp);***

It flushes the buffer of the file associated with fp. The function returns zero if successful, EOF if a failure occurs. If you call ***fflush()*** using a NULL for fp, all existing disk buffers are flushed.

**6.15 THE STANDARD STREAMS**

Standard streams : When a C program begins execution, three streams are automatically opened and available for use. These streams are called

* standard input (stdin),
* standard output (stdout), and
* standard error (stderr).

stdin inputs from the keyboard; stdout and stderr write to the screen. By default, they refer to the console, but in environments that Support redirectable I/O, they can be redirected by the operating system to some other device.

1. These standard streams are ***FILE*** pointers and may be used with any function that requires a variable of type ***FILE*** ***\*.*** For example, you can use ***fprintf()*** to print formatted output to the screen by specifying ***stdout*** as its output stream. The following two statements are functionally the same:

**fprintf**(**stdout**, "%d %c %s", 100, 'c', "this is a string");

**printf**("%d %c %s", 100, 'c', "this is a string");

1. In actuality, C makes little distinction between console I/O and file I/O. As just shown, it is possible to perform console I/O using several of the file-system functions.
2. It is also possible to perform disk file I/O using console I/O functions, such as ***printf()*** . Here's why. All of the functions described in 6.1 to 6.7 referred to as "console I/O functions" are actually special-case file-system functions that automatically operate on stdin and stdout.
3. As far as C is concerned, the console is simply another hardware device. You don't actually need the console functions to access the console. Any file -system function can access it. (*Of course, non-standard I/O functions like* ***getche()*** *are differentiated from the standard file-system functions and do, in fact, operate only on the console*.)

Redirection of std streams : In environments that allow redirection of I/O, stdin and stdout could refer to devices other than the keyboard and screen. Since the console functions operate on stdin and stdout. if these streams are redirected, the "console" functions can be made to operate on other devices. For example, by redirecting the stdout to a disk file. you can use a "console" I/O function to write to a disk file.

Note

* stdin, stdout, and stderr are not variables. They may not be assigned a value using ***fopen()*** , nor should you attempt to close them using ***fclose()*** . These streams are maintained internally by the compiler. You are free to use them, but not to change them.