UNIT 12 FILES

Structure

12.0 INTRODUCTION

The examples we have seen so far in the previous units deal with standard input and output. When data is stored using variables, the data is lost when the program exits unless something is done to save it. This unit discusses methods of working with files, and a data structure to store data. C views file simply as a sequential stream of bytes. Each file ends either with an *end-of-file* marker or at a specified byte number recorded in a system maintained, administrative data structure. C supports two types of files called ***binary*** *files* and ***text*** *files*.

The difference between these two files is in terms of storage. In *text files*, everything is stored in terms of text *i.e.* even if we store an integer 54; it will be stored as a 3-byte string - "54\0". In a text file certain character translations may occur. For example a *newline(\n)* character may be converted to a carriage return, linefeed pair. This is what Turbo C does. Therefore, there may not be one to one relationship between the characters that are read or written and those in the external device. A *binary file* contains data that was written in the same format used to store internally in main memory.

For example, the integer value 1245 will be stored in 2 bytes depending on the machine while it will require 5 bytes in a text file. The fact that a numeric value is in a standard length makes binary files easier to handle. No special string to numeric conversions is necessary.

The disk I/O in C is accomplished through the use of library functions. The ANSI standard, which is followed by TURBO C, defines one complete set of I/O functions. But since originally C was written for the UNIX operating system, UNIX standard defines a second system of routines that handles I/O operations. The first method, defined by both standards, is called a buffered file system. The second is the unbuffered file system.

In this unit, we will first discuss buffered file functions and then the unbuffered file functions in the following sections.

12.1 OBJECTIVES

After going through this unit you will be able to:

- define the concept of file pointer and file storage in C;
- create text and binary files in C;
- read and write from text and binary files;
- deal with large set of Data such as File of Records; and
- perform operations on files such as count number of words in a file, search a word in a file, compare two files etc.

12.2 FILE HANDLING IN C USING FILE POINTERS

As already mentioned in the above section, a sequential stream of bytes ending with an *end-of-file* marker is what is called a ***file***. When the file is opened the stream is associated with the file. By default, three files and their streams are automatically opened when program execution begins - the ***standard input***, ***standard output***, and the ***standard error***. Streams provide communication channels between files and programs.

For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables to write data on the screen. Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information, such as size, current file pointer position, type of file etc., to perform operations on the file. This structure also contains an integer called a *file descriptor* which is an index into the table maintained by the operating system namely, the *open file table.* Each element of this table contains a block called *file control block (FCB)* used by the operating system to administer a particular file.

The standard input, standard output and the standard error are manipulated using file pointers *stdin, stdout* and *stderr.* The set of functions which we are now going to discuss come under the category of buffered file system. This file system is referred to as buffered because, the routines maintain all the disk buffers required for reading / writing automatically.

To access any file, we need to declare a pointer to FILE structure and then associate it with the particular file. This pointer is referred as a *file pointer* and it is declared as follows:

*FILE *fp;*

12.2.1 Open A File Using The Function *fopen()*

Once a file pointer variables has been declared, the next step is to open a file. The *fopen()* function opens a stream for use and links a file with that stream. This function returns a file pointer, described in the previous section. The syntax is as follows:

*FILE *fopen(char *filename,*mode);*

where ***mode*** is a string, containing the desired open status. The filename must be a string of characters that provide a valid file name for the operating system and may include a path specification. The legal mode strings are shown below in the table 12.1:

Table 12.1: Legal values to the *fopen( )* mode parameter

MODE	MEANING
"r" / "rt"	opens a text file for read only access
"w" / "wt"	creates a text file for write only access
"a" / "at"	text file for appending to a file
"r+t"	open a text file for read and write access
"w+t"	creates a text file for read and write access,
"a+t"	opens or creates a text file and read access
"rb"	opens a binary file for read only access
"wb"	create a binary file for write only access
"ab"	binary file for appending to a file
"r+b"	opens a binary or read and write access
"w+b"	creates a binary or read and write access,
"a+b"	open or binary file and read access

The following code fragment explains how to open a file for reading.

Code Fragment 1

```
#include <stdio.h>

main ()
  {

    FILE *fp;
    if ((fp=fopen("file1.dat", "r"))==NULL)
    {
   printf("FILE DOES NOT EXIST\n");
   exit(0);
    }
 }
```

The value returned by the *fopen( )* function is a file pointer. If any error occurs while opening the file, the value of this pointer is *NULL*, a constant declared in <*stdio.h*>. Always check for this possibility as shown in the above example.

12.2.2 Close A File Using The Function Fclose( )

When the processing of the file is finished, the file should be closed using the fclose() function, whose syntax is:

*int fclose(FILE *fptr);*

This function flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, and then closes the stream. The return value is 0 if the file is closed successfully or a constant *EOF*, an end-of file marker, if an error occurred. This constant is also defined in <*stdio.h*>. If the function *fclose()* is not called explicitly, the operating system normally will close the file when the program execution terminates.

The following code fragment explains how to close a file.

Code Fragment 2

```
# include <stdio.h>
main ( )
{
 FILE *fp;
 if ((fp=fopen("file1.dat", "r"))==NULL)
    {
     printf("FILE DOES NOT EXIST\n");
     exit(0);
    }
……..
……..
……..
…….
/* close the file */
fclose(fp);
}
```

Once the file is closed, it cannot be used further. If required it can be opened in same or another mode.

Check Your Progress 1

1. How does fopen( ) function links a file to a stream?

……

……

……..….

2. Differentiate between text files and binary files.

……

……

……..

3. What is EOF and what is its value?

……

……

……..….

12.3 INPUT AND OUTPUT USING FILE POINTERS

After opening the file, the next thing needed is the way to read or write the file. There are several functions and macros defined in <*stdio.h*> header file for reading and writing the file. These functions can be categorized according to the form and type of data read or written on to a file. These functions are classified as:

- Character input/output functions
- String input/output functions
- Formatted input/output functions
- Block input/output functions.

12.3.1 Character Input and Output in Files

ANSI C provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library. They are listed and described below:

- getc()
- putc()

getc( ) is used to read a character from a file and *putc( )* is used to write a character to a file. Their syntax is as follows:

*int putc(int ch, FILE *stream);*
*int getc(FILE *stream);*

The file pointer indicates the file to read from or write to. The character ***ch*** is formally called an integer in *putc( )* function but only the low order byte is used. On success *putc( )* returns a character(in integer form) written or EOF on failure. Similarly getc( ) returns an integer but only the low order byte is used. It returns *EOF* when end-of-file is reached. *getc( )* and *putc( )* are defined in <stdio.h> as macros not functions.

fgetc() and fputc()

Apart from the above two macros, C also defines equivalent functions to read / write characters from / to a file. These are:

*int fgetc(FILE *stream);*
*int fputc(int c, FILE *stream);*

To check the end of file, C includes the function *feof( )* whose prototype is:

*int feof(FILE *fp);*

It returns **1** if end of file has been reached or **0** if not. The following code fragment explains the use of these functions.

Example 12.1

Write a program to copy one file to another.

```
/*Program to copy one file to another */

 #include <stdio.h>
main( )
{
  FILE *fp1;
  FILE *fp2;
  int ch;
  if((fp1=fopen("f1.dat","r")) == NULL)

   {
    printf("Error opening input file\n");
     exit(0);
    }
       if((fp2=fopen("f2.dat","w")) == NULL)
  {
   printf("Error opening output file\n");
          exit(0);
```

```
        }

        while (!feof(fp1))
        {
          ch=getc(fp1);
          putc(ch,fp2);
        }
        fclose(fp1);
        fclose(fp2);
}
```

OUTPUT

If the file ”f1.dat” is not present, then the output would be:
Error opening input file
If the disk is full, then the output would be:
Error opening output file

If there is no error, then “f2.dat” would contain whatever is present in “f1.dat” after the execution of the program, if “f2.dat” was not empty earlier, then its contents would be overwritten.

12.3.2 String Input/Output Functions

If we want to read a whole line in the file then each time we will need to call character input function, instead C provides some string input/output functions with the help of which we can read/write a set of characters at one time. These are defined in the standard library and are discussed below:

- *fgets( )*
- *fputs( )*

These functions are used to read and write strings. Their syntax is:

*int fputs(char *str, FILE *stream);*
*char *fgets(char *str, int num, FILE *stream);*

The integer parameter in *fgets( )* is used to indicate that at most num-1 characters are to be read, terminating at end-of-file or end-of-line. The end-of-line character will be placed in the string ***str*** before the string terminator, if it is read. If end-of-file is encountered as the first character, EOF is returned, otherwise str is returned. The *fputs( )* function returns a non-negative number or EOF if unsuccessful.

Example 12.2

Write a program read a file and count the number of lines in the file, assuming that a line can contain at most 80 characters.

```
/*Program to read a file and count the number of lines in the file */
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
  FILE *fp;
  int cnt=0;
  char str[80];
```

```
/* open a file in read mode */

  if ((fp=fopen("lines.dat","r"))== NULL)
  {    printf("File does not exist\n");
       exit(0);
  }
/* read the file till end of file is encountered */
  while(!(feof(fp)))
  {  fgets(str,80,fp);      /*reads at most 80 characters in str */
     cnt++;                 /* increment the counter after reading a line */
  }
}/* print the number of lines */
printf("The number of lines in the file is :%d\n",cnt);
fclose(fp);
}
```

OUTPUT

Let us assume that the contents of the file "***lines.dat***" are as follows:

This is C programming.
I love C programming.

To be a good programmer one should have a good logic. This is a must.
C is a procedural programming language.

After the execution the output would be:

The number of lines in the file is: 4

12.3.3 Formatted Input/Output Functions

If the file contains data in the form of digits, real numbers, characters and strings, then character input/output functions are not enough as the values would be read in the form of characters. Also if we want to write data in some specific format to a file, then it is not possible with the above described functions. Hence C provides a set of formatted input/output functions. These are defined in standard library and are discussed below:

fscanf() and ***fprintf()***

These functions are used for formatted input and output. These are identical to *scanf()* and *printf()* except that the first argument is a file pointer that specifies the file to be read or written, the second argument is the format string. The syntax for these functions is:

```
int  fscanf(FILE *fp, char *format,. . .);
int fprintf(FILE *fp, char *format,. . .);
```

Both these functions return an integer indicating the number of bytes actually read or written.

Example 12.3

Write a program to read formatted data (account number, name and balance) from a file and print the information of clients with zero balance, in formatted manner on the screen.

/* Program to read formatted data from a file */

```
#include<stdio.h>
main()
{
  int account;
  char name[30];
  double bal;
  FILE *fp;

 if((fp=fopen("bank.dat","r"))== NULL)
        printf("FILE not present \n");
     else
        do{
          fscanf(fp,"%d%s%lf",&account,name,&bal);
           if(!feof(fp))
               {
               if(bal==0)
               printf("%d %s %lf\n",account,name,bal);
                 }
            }while(!feof(fp));
 }
```

OUTPUT

This program opens a file "***bank.dat***" in the read mode if it exists, reads the records and prints the information (account number, name and balance) of the zero balance records.

Let the file be as follows:

```
101     nuj      1200
102     Raman  1500
103     Swathi  0
104     Ajay     1600
105     Udit     0
```

The output would be as follows:

```
103     Swathi  0
105     Udit     0
```

12.3.4 Block Input/Output Functions

Block Input / Output functions read/write a block (specific number of bytes from/to a file. A block can be a record, a set of records or an array. These functions are also defined in standard library and are described below.

- *fread( )*
- *fwrite( )*

These two functions allow reading and writing of blocks of data. Their syntax is:

*int fread(void *buf, int num_bytes, int count, FILE *fp);*

*int fwrite(void *buf, int num_bytes, int count, FILE *fp);*

In case of *fread(),* buf is the pointer to a memory area that receives the data from the file and in *fwrite(),* it is the pointer to the information to be written to the file. *num_bytes* specifies the number of bytes to be read or written. These functions are quite helpful in case of binary files. Generally these functions are used to read or write array of records from or to a file. The use of the above functions is shown in the following program.

Example 12.4

Write a program using *fread( )* and *fwrite()* to create a file of records and then read and print the same file.

```
/* Program to illustrate the fread() and fwrite() functions*/
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<string.h>

void main()
{
  struct stud
         {
          char name[30];
          int age;
          int roll_no;
         }s[30],st;
  int i;
  FILE *fp;

/*opening the file in write mode*/
       if((fp=fopen("sud.dat","w"))== NULL)
         {   printf("Error while creating a file\n");
                    exit(0);    }

/* reading an array of students */
          for(i=0;i<30;i++)
         scanf("%s %d %d",s[i].name,s[i].age,s[i].roll_no);

 /* writing to a file*/
         fwrite(s,sizeof(struct stud),30,fp);
         fclose(fp);

/* opening a file in read mode */
         fp=fopen("stud.dat","r");

/* reading from a file and writing on the screen */
         while(!feof(fp))
         {
           fread(&st,sizeof(struct stud),1,fp);
           fprintf("%s %d %d",st.name,st.age,st.roll_no);
          }
         fclose(fp);       }
```

OUTPUT

This program reads 30 records (name, age and roll_number) from the user, writes one record at a time to a file. The file is closed and then reopened in read mode; the records are again read from the file and written on to the screen.

Check Your Progress 2

1. Give the output of the following code fragment:

```
#include<stdio.h>
#include<process.h>
#include<conio.h>
main()
{
FILE * fp1, * fp2;
       double a,b,c;

fp1=fopen("file1", "w");
fp2=fopen("file2", "w");

fprintf(fp1,"1 5.34 –4E02");
fprintf(fp2,"-2\n1.245\n3.234e02\n");
     fclose(fp1);
     fclose(fp2);

fp1=fopen("file1", "r");
fp2=fopen("file2","r");

 fscanf(fp1,"%lf %lf %lf",&a,&b,&c);
 printf("%10lf %10lf %10lf",a,b,c);
 fscanf(fp2,"%lf %lf %lf",&a,&b,&c);
 printf("%10.1e %10lf %10lf",a,b,c);

      fclose(fp1);
      fclose(fp2);
  }
```

……

……

……

2. What is the advantage of using fread/fwrite functions?

……

……

……

3. _____ and _____ functions are used for formatted input and output from a file.

……

……

……

12.4 SEQUENTIAL Vs RANDOM ACCESS FILES

We have seen in section 12.0 that C supports two type of files – text and binary files, also two types of file systems – buffered and unbuffered file system. We can also differentiate in terms of the type of file access as Sequential access files and random access files. Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence. All the above examples

that we have considered till now in this unit are performing sequential access. Random access files allow reading data from any location in the file. To achieve this purpose, C defines a set of functions to manipulate the position of the file pointer. We will discuss these functions in the following sections.

12.5 POSITIONING THE FILE POINTER

To support random access files, C requires a function with the help of which the file pointer can be positioned at any random location in the file. Such a function defined in the standard library is discussed below:

The function *fseek( )* is used to set the file position. Its prototype is:

*int fseek(FILE *fp, long offset, int pos);*

The first argument is the pointer to a file. The second argument is the number of bytes to move the file pointer, counting from zero. This argument can be positive, negative or zero depending on the desired movement. The third parameter is a flag indicating from where in the file to compute the offset. It can have three values:

SEEK_SET(or value 0) the beginning of the file,
SEEK_CUR(or value 1) the current position and
SEEK_END(or value 2) the end of the file

These three constants are defined in <*stdio.h*>. If successful *fseek( )* returns zero. Another function *rewind()* is used to reset the file position to the beginning of the file. Its prototype is:

*void rewind(FILE *fp);*

A call to rewind is equivalent to the call

fseek(fp,0,SEEK_SET);

Another function *ftell()* is used to tell the position of the file pointer. Its prototype is:

*long ftell(FILE *fp);*

It returns –1 on error and the position of the file pointer if successful.

Example 12.5

Write a program to search a record in an already created file and update it. Use the same file as created in the previous example.

```
/*Program to search a record in an already created file*/

#include<stdio.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
#include<string.h>
void main()
{
        int r,found;
        struct stud
         {
```

```
        char name[30];
        int age;
        int roll_no;
       }st;
     FILE *fp;
    /* open the file in read/write mode */

     if((fp=fopen("f1.dat","r+b"))==NULL)
     {  printf("Error while opening the file \n");
       exit(0);
     }

/* Get the roll_no of the student */
    printf("Enter the roll_no of the record to be updated\n");
    found=0;
    scanf("%d",&r);

 /* check in the file for the existence of the roll_no */
    while((!feof(fp)) && !(found))
    {   fread(&st,sizeof(stud),1,fp);
       if(st.roll_no == r)

 /* if roll_no is found then move one record   backward to update it */
        {   fseek(fp,- sizeof(stud),SEEK_CUR);
           printf("Enter the new name\n");
           scanf("%s",st.name);
           fwrite(fp,sizeof(stud),1,fp);
           found=1;
         }
     }
     if (!found)
        printf("Record not present\n");
     fclose(fp);
    }
```

OUTPUT

Let the input file be as follows:

```
Geeta   18      101
Leena   17      102
Mahesh 23       103
Lokesh  21      104
Amit    19      105
```

Let the roll_no of the record to be updated be 106. Now since this roll_no is not present the output would be:

Record not present

If the roll_no to be searched is 103, then if the new name is Sham, the output would be the file with the contents:

```
Geeta   18      101
Leena   17      102
Sham    23      103
Lokesh  21      104
Amit    19       105
```

12.6 THE UNBUFFERED I/O – THE UNIX LIKE FILE ROUTINES

The buffered I/O system uses buffered input and output, that is, the operating system handles the details of data retrieval and storage, the system stores data temporarily (buffers it) in order to optimize file system access. The buffered I/O functions are handled directly as system calls without buffering by the operating system. That is why they are also known as low level functions. This is referred to as unbuffered I/O system because the programmer must provide and maintain all disk buffers, the routines do not do it automatically.

The low level functions are defined in the header file <*io.h*>.

These functions do not use file pointer of type FILE to access a particular file, but they use directly the file descriptors**,** as explained earlier, of type integer**.** They are also called *handles***.**

Opening and closing of files

The function used to open a file is *open( )*. Its prototype is:

int open(char *filename, int mode, int access);

Here *mode* indicates one of the following macros defined in <*fcntl.h*>.

Mode:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read / Write

The *access* parameter is used in UNIX environment for providing the access to particular users and is just included here for compatibility and can be set to zero. o*pen()* function returns *–1* on failure. It is used as:

Code fragment 2

```
int fd;

if ((fd=open(filename,mode,0)) == -1)
 {    printf("cannot open file\n");
      exit(1);    }
```

If the file does not exist, *open()* the function will not create it. For this, the function *creat()* is used which will create new files and re-write old ones. The prototype is:

*int creat(char *filename, int access);*

It returns a file descriptor; if successful else it returns –1. It is not an error to create an already existing file, the function will just truncate its length to zero. The *access* parameter is used to provide permissions to the users in the UNIX environment.
The function *close()* is used to close a file. The prototype is:

int close(int fd);

It returns zero if successful and –1 if not.

Reading, Writing and Positioning in File

The functions *read()* and *write()* are used to read from and write to a file. Their prototypes are:

*int read(int fd, void *buf, int size);*
*int write(int fd, void *buf, int size);*

The first parameter is the file descriptor returned by *open()*, the second parameter holds the data which must be typecast to the format needed by the program, the third parameter indicates the number of bytes to transferred. The return value tells how many bytes are actually transferred. If this value is –1, then an error must have occurred.

Example 12.6

Write a program to copy one file to another to illustrate the use of the above functions. The program should expect two command line arguments indicating the name of the file to be copied and the name of the file to be created.

```
/* Program to copy one file to another file to illustrate the functions*/
# include<stdio.h>
# include<io.h>
#include<process.h>

typedef char arr[80];
typedef char name[30];

main()
{
arr buf;
name fname, sname;
int fd1,fd2,size;

 /* check for the command line arguments */
if (argc!=3)
 {   printf("Invalid number of arguments\n");
          exit(0);
 }
  if ((fd1=open(argv[1],O_RDONLY))<0)
    {   printf("Error in opening file %s \n",argv[1]);
          exit(0);
        }
if ((fd2=creat(argv[2],0))<0)
       {   printf("Error in creating file %s \n",argv[2]);
          exit(0);}

 open(argv[2],O_WRONLY);
 size=read(fd1,buf,80);  /* read till end of file */

while (size>0)
      {  write(fd2,buf,80);
        size=read(fd1,buf,80);
       }
       close(fd1);
       close(fd2);
}
```

OUTPUT

If the number of arguments given on the command line is not correct then output would be:

Invalid number of arguments

One file is opened in the read mode, and another file is opened in the write mode. The output would be as follows is the file to be read is not present (let the file be *f1.dat*):

Error in opening file *f1.dat*

The output would be as follows if the disk is full and the file cannot be created (let the output file be *f2.dat*):

Error in creating file *f2.dat*

If there is no error contents of *f1.dat* will be copied to *f2.dat*.

lseek()

The function *lseek()* is provided to move to the specific position in a file. Its prototype is:

long lseek(int fd, long offset, int pos);

This function is exactly the same as *fseek()* except that the file descriptor is used instead of the file pointer.

Using the above defined functions, it is possible to write any kind of program dealing with files.

Check Your Progress 3

1. Random access is possible in C files using function _____.

2. Write a proper C statement with proper arguments that would be called to move the file pointer back by 2 bytes.

……

……

3. Indicate the header files needed to use unbuffered I/O.

……

……

……..….

12.7 SUMMARY

In this unit, we have learnt about files and how C handles them. We have discussed the buffered as well as unbuffered file systems. The available functions in the standard library have been discussed. This unit provided you an ample set of programs to start with. We have also tried to differentiate between sequential access as well as random access file. The file pointers assigned to standard input, standard output and standard error are *stdin, stdout,* and *stderr* respectively. The unit clearly explains the different

type of modes oof opening the file. As seen there are several functions available to read/write from the file. The usage of a particular function depends on the application. After reading this unit one must be able to handle large data bases in the form of files.

12.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. *fopen()* function links a file to a stream by returning a pointer to a FILE structure defined in <*stdio.h*>. This structure contains an index called file descriptor to a File Control Block, which is maintained by the operating system for administrative purposes.

2. Text files and binary files differ in terms of storage. In text files everything is stored in terms of text while binary files stores exact memory image of the data i.e. in text files 154 would take 3 bytes of storage while in binary files it will take 2 bytes as required by an integer.

3. *EOF* is an end-of-file marker. It is a macro defined in <*stdio.h*>. Its value is –1.

Check Your progress 2

1. The output would be:

   1.000000 5.340000 –400.000000 -2.0e+00 1.245000 323.400000

2. The advantage of using these functions is that they are used for block read/write, which means we can read or write a large set of data at one time thus increasing the speed.

3. *fscanf()* and *fprintf()* functions are used for formatted input and output from a file.

Check Your progress 3

1. Random access is possible in C files using function *fseek()*.

2. fseek(fp, -2L, SEEK_END);

3. <io.h> and <fcntl.h>

12.9 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. C How to Program, *Deitel & Deitel*, Pearson Education, 2002.
3. Practical C Programming, *Steve Oualline*, Oreilly Publication, 2003.

APPENDIX-A

THE ASCII SET

The ASCII (American Standard Code for Information Interchange) character set defines 128 characters (0 to 127 decimal, 0 to FF hexadecimal, and 0 to 177 octal). This character set is a subset of many other character sets with 256 characters, including the ANSI character set of MS Windows, the Roman-8 character set of HP systems, and the IBM PC Extended Character Set of DOS, and the ISO Latin-1 character set used by Web browsers. They are not the same as the EBCDIC character set used on IBM mainframes. The first 32 values are non-printing **control characters**, such as *Return* and *Line feed*. You generate these characters on the keyboard by holding down the Control key while you strike another key. For example, Bell is value 7, Control plus G, often shown in documents as ^G. Notice that 7 is 64 less than the value of G (71); the Control key subtracts 64 from the value of the keys that it modifies. The table shown below gives the list of the control and printing characters.

The Control Characters

Char	Oct	Dec	Hex	Control-Key	Control Action
NUL	0	0	0	^@	Null character
SOH	1	1	1	^A	Start of heading, = console interrupt
STX	2	2	2	^B	Start of text, maintenance mode on HP console
ETX	3	3	3	^C	End of text
EOT	4	4	4	^D	End of transmission, not the same as ETB
ENQ	5	5	5	^E	Enquiry, goes with ACK; old HP flow control
ACK	6	6	6	^F	Acknowledge, clears ENQ logon hand
BEL	7	7	7	^G	Bell, rings the bell...
BS	10	8	8	^H	Backspace, works on HP terminals/computers
HT	11	9	9	^I	Horizontal tab, move to next tab stop
LF	12	10	a	^J	Line Feed
VT	13	11	b	^K	Vertical tab
FF	14	12	c	^L	Form Feed, page eject
CR	15	13	d	^M	Carriage Return
SO	16	14	e	^N	Shift Out, alternate character set
SI	17	15	f	^O	Shift In, resume defaultn character set
DLE	20	16	10	^P	Data link escape
DC1	21	17	11	^Q	XON, with XOFF to pause listings; ":okay to send".
DC2	22	18	12	^R	Device control 2, block-mode flow control
DC3	23	19	13	^S	XOFF, with XON is TERM=18 flow control
DC4	24	20	14	^T	Device control 4
NAK	25	21	15	^U	Negative acknowledge
SYN	26	22	16	^V	Synchronous idle
ETB	27	23	17	^W	End transmission block, not the same as EOT
CAN	30	24	17	^X	Cancel line, MPE echoes !!!
EM	31	25	19	^Y	End of medium, Control-Y interrupt
SUB	32	26	1a	^Z	Substitute
ESC	33	27	1b	^[	Escape, next character is not echoed
FS	34	28	1c	^\	File separator
GS	35	29	1d	^]	Group separator
RS	36	30	1e	^^	Record separator, block-mode terminator
US	37	31	1f	^_	Unit separator

Printing Characters

Char	Octal	Dec	Hex	Description
SP	40	32	20	Space
!	41	33	21	Exclamation mark
"	42	34	22	Quotation mark (" in HTML)
#	43	35	23	Cross hatch (number sign)
$	44	36	24	Dollar sign
%	45	37	25	Percent sign
&	46	38	26	Ampersand
`	47	39	27	Closing single quote (apostrophe)
(	50	40	28	Opening parentheses
)	51	41	29	Closing parentheses
*	52	42	2a	Asterisk (star, multiply)
+	53	43	2b	Plus
,	54	44	2c	Comma
-	55	45	2d	Hyphen, dash, minus
.	56	46	2e	Period
/	57	47	2f	Slant (forward slash, divide)
0	60	48	30	Zero
1	61	49	31	One
2	62	50	32	Two
3	63	51	33	Three
4	64	52	34	Four
5	65	53	35	Five
6	66	54	36	Six
7	67	55	37	Seven
8	70	56	38	Eight
9	71	57	39	Nine
:	72	58	3a	Colon
;	73	59	3b	Semicolon
<	74	60	3c	Less than sign (< in HTML)
=	75	61	3d	Equals sign
>	76	62	3e	Greater than sign (> in HTML)
?	77	63	3f	Question mark
@	100	64	40	At-sign
A	101	65	41	Uppercase A
B	102	66	42	Uppercase B
C	103	67	43	Uppercase C
D	104	68	44	Uppercase D
E	105	69	45	Uppercase E
F	106	70	46	Uppercase F
G	107	71	47	Uppercase G
H	110	72	48	Uppercase H
I	111	73	49	Uppercase I
J	112	74	4a	Uppercase J
K	113	75	4b	Uppercase K
L	114	76	4c	Uppercase L
M	115	77	4d	Uppercase M
N	116	78	4e	Uppercase N

O	117	79	4f	Uppercase O
P	120	80	50	Uppercase P
Q	121	81	51	Uppercase Q
R	122	82	52	Uppercase R
S	123	83	53	Uppercase S
T	124	84	54	Uppercase T
U	125	85	55	Uppercase U
V	126	86	56	Uppercase V
W	127	87	57	Uppercase W
X	130	88	58	Uppercase X
Y	131	89	59	Uppercase Y
Z	132	90	5a	Uppercase Z
[	133	91	5b	Opening square bracket
\	134	92	5c	Reverse slant (Backslash)
]	135	93	5d	Closing square bracket
^	136	94	5e	Caret (Circumflex)
_	137	95	5f	Underscore
`	140	96	60	Opening single quote
a	141	97	61	Lowercase a
b	142	98	62	Lowercase b
c	143	99	63	Lowercase c
d	144	100	64	Lowercase d
e	145	101	65	Lowercase e
f	146	102	66	Lowercase f
g	147	103	67	Lowercase g
h	150	104	68	Lowercase h
i	151	105	69	Lowercase i
j	152	106	6a	Lowercase j
k	153	107	6b	Lowercase k
l	154	108	6c	Lowercase l
m	155	109	6d	Lowercase m
n	156	110	6e	Lowercase n
o	157	111	6f	Lowercase o
p	160	112	70	Lowercase p
q	161	113	71	Lowercase q
r	162	114	72	Lowercase r
s	163	115	73	Lowercase s
t	164	116	74	Lowercase t
u	165	117	75	Lowercase u
v	166	118	76	Lowercase v
w	167	119	77	Lowercase w
x	170	120	78	Lowercase x
y	171	121	79	Lowercase y
z	172	122	7a	Lowercase z
{	173	123	7b	Opening curly brace
|	174	124	7c	Vertical line
}	175	125	7d	Cloing curly brace
~	176	126	7e	Tilde (approximate)
DEL	177	127	7f	Delete (rubout), cross-hatch box