MODULE 9
C FILE INPUT/OUTPUT

MODULE 19
C++ FILE I/O
create this, delete that, write this, read that, close this, open that

My Training Period: hours

Abilities

Trainee must be able to understand and use:

- The basic of the data hierarchy.
- A sequential access file – Read and Write related functions.
- Characters, lines and blocks disk file reading and writing related functions.
- A Random access files – Read and Write related functions.
- Some File Management Functions.
- Other libraries used for file I/O.

9.1 Introduction

- This Module actually shows you how to use the functions readily available in the C standard library. Always remember this, using the standard library (ISO/IEC C, Single Unix specification or glibc); you must know which functions to call and which header files provide these functions. Then, you must be familiar with the proper prototype of the function call.
- The problems normally exist when dealing with the parameters passed to the functions and the return value of the functions. We will explore some of the very nice and one of the heavily used functions that available in the `stdio.h` header file, for our file processing and management tasks.
- Keep in mind that in C++ we will use member functions in class objects for file processing and some of the advanced file processing examples will be discussed in C++ file I/O Module.
- Storage of data file as you have learned is temporary, all such data is lost when a program terminates. That is why we have to save files on primary or secondary storage such as disks for future usage.
- Besides that we also need to process data from external files that may be, located in secondary storage as well as writing file during software installation and communicating with computer devices such as floppy, hard disk, networking etc.
- And in socket programming (networking) you will also deal a lot with these open, close, read write activities.
- File used for permanent retention of large amounts of data, stored online or offline in secondary storage devices such as hard disk, CD-Rs/DVDs, tape backup or Network Attached Storage (NAS).

9.2 Basic of The Data Hierarchy

- Ultimately, all data items processed by a computer are just combinations of zeroes and ones.
- The smallest data item in computer can assume the value 0 or 1, called a **bit** (binary digit).
- But, human being prefer to work with data in the form of decimal digits (i.e. 0, 1, 2, 3, 4, 5, 6, 7…9), letters (i.e. A – Z and a – z) and special symbols (i.e. $, @, %, &, *, (,), -, +, ? and many others) or in readable format.
- As you know, digits, letters and special symbols are referred to as characters, the keys on your keyboard based on whether the ASCII, EBCDIC, Unicode or other proprietary characters set.
- Every character in a computer's character set is represented as a pattern of 1's and 0's, called **byte** (consists 8 bits-ASCII, EBCDIC), and for Unicode it uses multibyte or wide characters.
- **Characters** are composed of bits, and then **fields** (columns) are composed of characters.
- A **field** is a group of characters that conveys meaning such as a field representing a month of year.
- Data items processed by computer form a data hierarchy in which data items become larger and more complex in structure as we progress from bits, to char (byte) to field and so on.
- A **record** (row or tuple) is composed of several fields.
- For example, in a payroll system, a record for a particular employee might consist of the following fields:

    0. Name.
    0. Address.

0. Security Social Number (SSN)/Passport Number
0. Salary.
0. Year to date earnings.
0. Overtime claims.

- So, a record is a group of related fields.
- For the payroll example, each of the fields belong to the same employee, in reality a company may have many employees, and will have a payroll records for each employee.
- Conceptually, a **file** is a group of related records.
- A company's payroll file normally contains one record for each employee, thus, a payroll file for a company might contain up to 100, 000 records.
- To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key.
- There are many ways of organizing records in a file. Maybe, the most popular type of organization is called a sequential file in which records are typically stored in order by the record **key** field.

Figure 9.1: An illustration of a simple data hierarchy.

- For example, in a payroll file, records are usually placed in order by Social Security Number (SSN). The first employee record in the file contains the lowest SSN number and subsequent records contain increasingly higher SSN numbers.
- Most business may utilize many different files to store data, for example inventory files, payroll files, employee files and many other types of files.
- For larger application, a group of related files may be called database.
- An application designed to create and manage databases is called a database management system (DBMS). This DBMS term used here just for the data structure discussion, in database it may be different. Popular type of DBMS is Relational DataBase Management System (RDBMS).
- A complete discussion of programming related to the databases can be found in data structure books.
- Here we just want to have some basic knowledge about the construct of the data that the computer processes, from bit to characters to fields and so on until we have a very readable data format organized in a structured manner.

9.3 Files And Streams

- In C, a file can refer to a disk file, a terminal, a printer, a tape drive, sockets or other related devices. Hence, a file represents a concrete device with which you want to exchange information.
- Before you perform any communication to a file, you have to open the file. Then you need to close the opened file after finish exchanging or processing information with it.
- The main file processing tasks may involve the opening, reading, writing and closing.
- The data flow of the transmission from your program to a file, or vice versa, is called a **stream**, which is **a series of bytes**. Different with file, a stream is device-independent. All streams have the same behavior including that used in sockets programming such as the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) streams.

- Hence, to perform I/O operations, you can read from or write to any type of files by simply associating a stream to the file.
- There are two formats of streams. The first one is called the **text stream**, which consists of a sequence of characters (e.g. ASCII data). Depending on the compilers, each character line in a text stream may be terminated by a newline character. Text streams are used for textual data, which has a consistent appearance from one system to another.
- The second format of stream is called the **binary stream**, which is still a series of bytes. The content of an `.exe` file would be one example. It is primarily used for non-textual data, which is required to keep the exact contents of the file.
- And for Unicode it is Unicode stream in text or binary modes.
- In C, a memory area, which is temporarily used to store data before it is sent to its destination, is called a **buffer**. By using buffer, the operating system can improve efficiency by reducing the number of accesses to I/O devices.
- By default all I/O streams are buffered. The buffered I/O is also called the high-level I/O and the low-level I/O refers to the unbuffered I/O.
- Keep in mind that in order to grasp the basic concepts, this Module will deal mainly with unformatted text files.

9.4 Directories, Files and streams

9.4.1 Directories (Folders)

- Every OS have different file system for example ext2, ext3 (Linux), FAT, FAT32, NTFS, NTFS 5 (Windows). In general this discussion is biased to Linux file system.
- A file system is organized into a hierarchy of directories. For example:

```
C:\Program Files\Microsoft Visual Studio\VC98\Bin
```

- Or in Linux:

```
/testo1/testo2/testo3
```

- Or by issuing a `tree` command at Windows command prompt:

- A directory is a file that contains information to associate other files with names; these associations are called links (shortcuts) or directory entries. Actually, a directory only contains **pointers to files**, not the files themselves but as users we usually just say "files in a directory".
- The name of a file contained in a directory entry is called a **file name component**.
- In general, a file name consists of a sequence of one or more such components, separated by the slash character (`/`). So, a file name which is just one component, names a file with respect to its directory. A file name with multiple components, names a directory, and then a file in that directory, and so on.
- Some other documents, such as the POSIX standard, use the term **pathname** for what was call a **file name** here and either filename or pathname component should refer to the same meaning in this discussion.

9.4.2 File Name Resolution

- A file name consists of file name components separated by slash (/) characters. On the systems that the GNU C library supports, multiple successive / characters are equivalent to a single / character.
- The process of determining what file a file name refers to is called file name resolution. This is performed by examining the components that make up a file name in left-to-right order, and locating each successive component in the directory, named by the previous component.
- Each of the files that are referenced as directories must actually exist, be directories instead of regular files, and have the appropriate permissions to be accessible by the process; otherwise the file name resolution fails.
- Unlike some other operating systems such as Windows, the Linux system doesn't have any built-in support for file types (or extensions) or file versions as part of its file name prototype.
- Many programs and utilities use conventions for file names. For example, files containing C source code usually have names suffixed with `.c` and executable files have `.exe` extension, but there is nothing in the Linux file system itself that enforces this kind of convention.
- May be you can better differentiate those file types by using the –F option for `ls` directory listing command (`ls –F`).
- If a file name begins with a /, the first component in the file name is located in the root directory of the process (usually all processes on the system have the same root directory). In Windows it is normally a C: drive. Such a file name is called an absolute file name.
- Otherwise, the first component in the file name is located in the current working directory and this kind of file name is called a relative file name. For example, the `Secondir` and `Thirdir` should be relative file name and `Firstdir` is an absolute filename.

```
/Firstdir/Secondir/Thirdir
```

- The file name components `.` ("dot") and `..` ("dot-dot") have special meanings.

```
C:\Firstdir>dir /a
 Volume in drive C has no label.
 Volume Serial Number is E8E3-18E2

 Directory of C:\Firstdir

04/18/2005  03:09p      <DIR>          .
04/18/2005  03:09p      <DIR>          ..
04/18/2005  03:08p                     0 first.txt
04/18/2005  03:09p      <DIR>          Secondir
               1 File(s)              0 bytes
               3 Dir(s)   1,326,395,392 bytes free
```

- Every directory has entries for these file name components. The file name component `.` refers to the directory itself, while the file name component `..` refers to its parent directory (the directory that contains the link for the directory in question). That is why if we want to change to the parent directory of the current working directory we just issue the **`cd ..`** command for Linux and Windows.
- Then in Linux, to run a program named `testrun` in the current working directory we issue the following command:

```
./testrun
```

- As a special case, `..` in the root directory refers to the root directory itself, since it has no parent; thus `/..` is the same as `/`.
- Here are some examples of file names:

File name	Description
`/a`	The file named `a`, in the root directory.
`/a/b`	The file named `b`, in the directory named `a` in the root directory.
`a`	The file named `a`, in the current working directory.
`/a/./b`	This is the same as `/a/b`.
`./a`	The file named `a`, in the current working directory.
`../a`	The file named `a`, in the parent directory of the current working directory.

Table 9.1: File names examples

- A file name that names a directory may optionally end in a `/`. You can specify a file name of `/` to refer to the root directory, but the empty string is not a meaningful file name.
- If you want to refer to the current working directory, use a file name of `.` or `./`. For example to run a program named `testprog` that located in the current working directory we just prefixes the `./` to the program name.

```
./testprog
```

9.4.3 Streams and `FILE` structure

- The type of the C data structure that represents a stream is called `FILE` rather than "stream". Since most of the library functions deal with objects of type `FILE *`, sometimes the term file pointer is also used to mean "stream". This leads to confusion over terminology in many reference materials and books on C.
- The `FILE` type is declared in the `stdio.h` header file.

`FILE` data type
This is the data type used to represent stream objects. A `FILE` object holds all of the internal state information about the connection to the associated file, including such things as the file position indicator and buffering information. Each stream also has error and `end-of-file` status indicators that can be tested with the `ferror` and `feof` functions.

Table 9.2: `FILE` data type

- `FILE` objects are allocated and managed internally by the I/O library functions.

9.4.4 Standard Streams

- When the main function of your program is invoked, it already has three predefined streams open and available for use. These represent the standard input and output channels that have been established for the process. A process here means a running program.
- These streams are declared in the `stdio.h` header file and summarized in the following Table.

Standard stream	Description
`FILE * stdin`	The *standard input* stream variable, which is the normal source of input for the program.
`FILE * stdout`	The *standard output* stream variable, which is used for normal output from the program.
`FILE * stderr`	The *standard error* stream variable, which is used for error messages and diagnostics issued by the program.

Table 9.3: Standard streams

- In the Linux system, you can specify what files or processes correspond to these streams using the **pipe** and **redirection facilities** provided by the shell.
- Most other operating systems provide similar mechanisms, but the details of how to use them can vary.
- In the GNU C library, `stdin`, `stdout`, and `stderr` are normal variables which you can set just like any others. For example, to redirect the standard output to a file, you could do:

```
fclose(stdout);
stdout = fopen ("standard-output-file", "w");
```

- However, in other systems `stdin`, `stdout`, and `stderr` are macros instead of variables that you cannot assign to in the normal way. But you can use for example `freopen()` function to get the effect of closing one and reopening it.

9.5 Links Story

9.5.1 Hard Links

- In POSIX systems, one file can have many names at the same time. All of the names are equally real, and no one of them is preferred to the others. In Windows it is called shortcuts.
- To add a name to a file, use the `link()` function (The new name is also called a hard link to the file). Creating a new link to a file does not copy the contents of the file; it simply makes a new name by which the file can be known, in addition to the file's existing name or names.
- One file can have names in several directories, so the organization of the file system is not a strict hierarchy or tree.
- In most implementations, it is not possible to have hard links to the same file in multiple file systems. `link()` reports an error if you try to make a hard link to the file from another file system when this cannot be done.
- The prototype for the `link()` function is declared in the header file `unistd.h` and is summarized below.

`int link(const char *oldname, const char *newname)`	
The `link` function makes a new link to the existing file named by *oldname*, under the new name *newname*. This function returns a value of `0` if it is successful and `-1` on failure. In addition to the usual file name errors, for both *oldname* and *newname*, the following `errno` error conditions are defined for this function:	
`EACCES`	You are not allowed to write to the directory in which the new link is to be written.
`EEXIST`	There is already a file named *newname*. If you want to replace this link with a new link, you must remove the old link explicitly first.
`EMLINK`	There are already too many links to the file named by *oldname*. (The maximum number of links to a file is `LINK_MAX`).
`ENOENT`	The file named by *oldname* doesn't exist. You can't make a link to a file that doesn't exist.
`ENOSPC`	The directory or file system that would contain the new link is full and cannot be extended.
`EPERM`	In the GNU system and some others, you cannot make links to directories. Many systems allow only privileged users to do so. This error is used to report the problem.
`EROFS`	The directory containing the new link can't be modified because it's on a read-only file system.
`EXDEV`	The directory specified in *newname* is on a different file system than the existing file.
`EIO`	A hardware error occurred while trying to read or write the to filesystem.

Table 9.4: `link()` function

9.5.2 Symbolic Links

- The Linux system for example, supports soft links or symbolic links. This is a kind of "file" that is essentially a pointer to another file name.
- Unlike hard links, symbolic links can be made to directories or across file systems with no restrictions. You can also make a symbolic link to a name which is not the name of any file. (Opening this link will fail until a file by that name is created).
- Likewise, if the symbolic link points to an existing file which is later deleted, the symbolic link continues to point to the same file name even though the name no longer names any file.
- The reason symbolic links work the way they do is that special things happen when you try to open the link. The `open()` function realizes you have specified the name of a link, reads the file name contained in the link, and opens that file name instead.
- The `stat()` function (used for file attributes information) likewise operates on the file that the symbolic link points to, instead of on the link itself.
- By contrast, other operations such as deleting or renaming the file operate on the link itself. The functions `readlink()` and `lstat()` also refrain from following symbolic links, because their purpose is to obtain information about the link.
- `link()`, the function that makes a hard link, does too. It makes a hard link to the symbolic link, which one rarely wants.
- Some systems have for some functions operating on files have a limit on how many symbolic links are allowed when resolving a path name. The limit if exists is published in the `sys/param.h` header file.
- The following lists functions and macros used for links.

`int MAXSYMLINKS`
The macro `MAXSYMLINKS` specifies how many symlinks some function will follow before returning `ELOOP`. Not all functions behave the same and this value is not the same as a returned for `_SC_SYMLOOP` by `sysconf`. In fact, the `sysconf` result can indicate that

there is no fixed limit although MAXSYMLINKS exists and has a finite value.

Table 9.5: MAXSYMLINKS macro

- Prototypes for most of the functions listed in the following section are in unistd.h.

int symlink(*const char *oldname, const char *newname*)	
The symlink function makes a symbolic link to *oldname* named *newname*. The normal return value from symlink is 0. A return value of -1 indicates an error. In addition to the usual file name prototype errors (see File Name Errors), the following errno error conditions are defined for this function:	
EEXIST	There is already an existing file named *newname*.
EROFS	The file *newname* would exist on a read-only file system.
ENOSPC	The directory or file system cannot be extended to make the new link.
EIO	A hardware error occurred while reading or writing data on the disk.

Table 9.6: symlink() function

int readlink(*const char *filename, char *buffer, size_t size*)	
The readlink function gets the value of the symbolic link *filename*. The file name that the link points to is copied into *buffer*. This file name string is *not* null-terminated; readlink normally returns the number of characters copied. The *size* argument specifies the maximum number of characters to copy, usually the allocation size of *buffer*. If the return value equals *size*, you cannot tell whether or not there was room to return the entire name. A value of -1 is returned in case of error. In addition to the usual file name errors, following errno error conditions are defined for this function:	
EINVAL	The named file is not a symbolic link.
EIO	A hardware error occurred while reading or writing data on the disk.

Table 9.7: readlink() function

- In some situations it is desirable to resolve all the symbolic links to get the real name of a file where no prefix, names a symbolic link which is followed and no filename in the path is . or .. .
- This is for example desirable if files have to be compared in which case different names can refer to the same inode. For Linux system we can use canonicalize_file_name() function for this purpose.
- The UNIX standard includes a similar function which differs from canonicalize_file_name() in that the user has to provide the buffer where the result is placed in. It uses realpath() function.
- The advantage of using this function is that it is more widely available. The drawback is that it reports failures for long path on systems which have no limits on the file name length.

9.6 The Basic Of Disk File I/O

9.6.1 Opening And Closing A Disk File

- Before we dive into the details, take note that the program examples presented here just for basic file I/O that applies to DOS and Linux.
- For Windows, you have to study the Win32 programming that provides specifics file I/O and other related functions. Here we do not discuss in details regarding the permission, right and authorization such as using Discretionary Access Control List (DACL) and Security Access Control List (SACL) implemented in Windows OS.
- Furthermore for DOS type OS also, Microsoft uses Microsoft C (C Runtime – CRT). Nevertheless the concepts still apply to any implementation.
- As explained before, in C, a FILE structure is a file control structure defined in the header file stdio.h. A pointer of type FILE is called a file pointer, which references a disk file.
- A file pointer is used by stream to conduct the operation of the I/O functions. For instance, the following declaration defines a file pointer called fpter:

```
FILE  *fpter;
```

- In the `FILE` structure there is a member, called the file position indicator, which points to the position in a file where data will be read from or written to.
- The I/O function `fopen()` gives you the ability to open a file and associate a stream to the opened file. You need to specify the **way to open** a file and the **filename** with the `fopen()` function. The prototype is:

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

- Here, `filename` is a `char` pointer that references a string of a filename. The filename is given to the file that is about to be opened by the `fopen()` function. `mode` points to another string that specifies the way to open the file.
- The `fopen()` function returns a pointer of type `FILE`. If an error occurs during the procedure to open a file, the `fopen()` function returns a null pointer.
- Table 9.8 shows the possible ways to open a file by various strings of modes.
- Note that, you might see people use the mode `rb+` instead of `r+b`. These two strings are equivalent. Similarly, `wb+` is the same as `w+b`, `ab+` is equivalent to `a+b`.
- The following program segment example try to open a file named `test.txt`, located in the same folder as the `main()` program for reading.

```
FILE *fptr;
if((fptr = fopen("test.txt","r")) == NULL)
{
   printf("Cannot open test.txt file.\n");
   exit(1);
}
```

- Here, "`r`" is used to indicate that the text file is about to be opened for reading only. If an error occurs such as the file is non-exist, when the `fopen()` function tries to open the file, the function returns a null pointer.
- Then an error message is printed out by the `printf()` function and the program is aborted by calling the `exit()` function with a nonzero value to handle the exception.

Mode	Description
`r`	Open a file for reading.
`w`	Create a file for writing. If the file already exists, discard the current contents.
`a`	Append, open or create a file for writing at the end of the file.
`r+`	Open a file for update (reading and writing).
`w+`	Create a file for update. If the file already exists, discard the current contents.
`a+`	Append, open or create a file for update, writing is done at the end of the file.
`rb`	Opens an existing binary file for reading.
`wb`	Creates a binary file for writing.
`ab`	Opens an existing binary file for appending.
`r+b`	Opens an existing binary file for reading or writing.
`w+b`	Creates a binary file for reading or writing.
`a+b`	Opens or creates a binary file for appending.

Table 9.8: Possible ways opening a file by various strings of modes in C

- After a disk file is read, written, or appended with some new data, you have to disassociate the file from a specified stream by calling the `fclose()` function.
- The prototype for the `fclose()` function is:

```
int fclose(FILE *stream);
```

- Here, stream is a file pointer that is associated with a stream to the opened file. If `fclose()` closes a file successfully, it returns `0`. Otherwise, the function returns EOF.
- By assuming the previous program segment successfully opened the `test.txt` for reading, then to close the file pointer we should issue the following code:

```
fclose(fptr);
```

- Normally, the `fclose()` function fails only when the disk is removed before the function is called or there is no more space left on the disk.
- The end-of-file (EOF) combination key for different platform is shown in table 9.9. You have to check your system documentation.

Computer system	Key combination
UNIX® systems	<return> <ctrl> d
IBM® PC and compatibles	<ctrl> z
Macintosh® - PowerPC	<ctrl> d
VAX® (VMS)	<ctrl> z

Table 9.9: End-of-file (EOF) key combinations for various computer systems.

- Since all high-level I/O operations are buffered, the `fclose()` function flushes data left in the buffer to ensure that no data will be lost before it disassociates a specified stream with the opened file.
- A file that is opened and associated with a stream has to be closed after the I/O operation. Otherwise, the data saved in the file may be lost or some unpredictable errors might occur during the next time file opening.
- Let try the following program example, which shows you how to open and close a text file and how to check the returned file pointer value as well.
- First of all you have to create file named `tkk1103.txt`. This file must be in the same folder as your running program, or you have to provide the full path string if you put it in other folder.
- By default program will try finding file in the same folder where the program is run.
- For example, if you run your C program in folder:

C:\BC5\Myproject\testing

- Then, make sure you put the `tkk1103.txt` file in the same folder:

C:\BC5\Myproject\testingtkk1103.txt

```
1.  //Opening and closing file example
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.
5.  //SUCCESS = 0, FAIL = 1 using enumeration
6.  enum {SUCCESS, FAIL};
7.
8.  int main (void)
9.  {
10.     FILE   *fptr;
11.     //the filename is tkk1103.txt and located
12.     //in the same folder as this program
13.     char filename[] = "tkk1103.txt";
14.
15.     //set the value reval to 0
16.     int reval = SUCCESS;
17.     //test opening file for reading, if fail...
18.     if((fptr = fopen(filename, "r")) == NULL)
19.     {
20.          printf("Cannot open %s.\n", filename);
21.          reval = FAIL;      //reset reval to 1
22.     }
23.     //if successful do...
24.     else
25.     {
26.          printf("Opening the %s file successfully\n", filename);
27.          //the program will display the address where
28.          //the file pointer points to..
29.          printf("The value of fptr: 0x%p\n", fptr);
30.          printf("\n....file processing should be done here....\n");
31.          printf("\nReady to close the %s file.\n", filename);
32.          //close the file stream...
33.          if(fclose(fptr)==0)
34.          printf("Closing the %s file successfully\n", filename);
35.     }
36.     //for Borland…can remove the following pause and the library,
37.     //stdlib.h for other compilers
38.     system("pause");
39.     return reval;
```

```
40. }
```

40 lines: Output:

- If opening the file is fails, the following will be output:

- Remember, for the `"r"` mode, you have to create and save `tkk1103.txt` file in the same folder where the `.exe` file for this program resides or provide the full path strings in the program.
- This program shows you how to open a text file. `fopen()` function tries to open a text file with the name contained by the string array `filename` for reading. The `filename` (stored in array) is defined and initialized with to `tkk1103.txt`.
- If an error occurs when you try to open the text file, the `fopen()` function returns a null pointer. Next line then prints a warning message, and assigns the value represented by the `enum` name `FAIL` to the `int` variable `reval`. From the declaration of the `enum` data type, we know that the value of `FAIL` is 1.
- However, if the `fopen()` function opens the text file successfully, the following statement:

```
printf("The value of fptr: 0x%p\n", fptr);
```

- Will print the value contained by the file pointer `fptr`.
- At the end line of code tells the user that the program is about to close the file, and then `fclose(fptr);` closes the file by calling the `fclose()` file.
- `return reval;` the return statement returns the value of `reval` that contains 0 if the text file has been opened successfully or 1 otherwise.
- From the output, the value held by the file pointer is `0x0D96:01C2` (memory address) after the text file is open successfully. Different pc will have different address.
- If your `tkk1103.txt` file is not in same folder as your `main()` program, you have to explicitly provide the full path of the file location.
- For example, if your `tkk1103.txt` is located in `C:\Temp` folder, you have to change:

```
char filename[] = "tkk1103.txt";
```

- To

```
char filename[] = "c:\\Temp\\tkk1103.txt";
```

9.6.2 Reading And Writing Disk File

- The previous program example does not do anything with the text file, `tkk1103.txt`, except open and close it. Some text has been saved in `tkk1103.txt`, so how can you read them from the file?
- In C you can perform I/O operations in the following ways:

    0. Read or write one character at a time.
    0. Read or write one line of text (that is, one line of characters) at a time.

0. Read or write one block of characters at a time.

9.6.2.1 One Character At A Time

- Among the C I/O functions, there is a pair of functions, `fgetc()` and `fputc()`, that can be used to read from or write to a disk file one character at a time.
- The prototype for the `fgetc()` function is:

```
int  fgetc(FILE  *stream);
```

- The `stream` is the file pointer that is associated with a stream. The `fgetc()` function fetches the next character from the stream specified by `stream`. The function then returns the value of an `int` that is converted from the character.
- The prototype for the `fputc()` function is:

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

- `c` is an `int` value that represents a character. In fact, the `int` value is converted to an `unsigned char` before being output. `stream` is the file pointer that is associated with a stream. The `fputc()` function returns the character written if the function is successful, otherwise, it returns `EOF`. After a character is written, the `fputc()` function advances the associated file pointer.
- Let explore the program example. Before that, you have to create two text files named, `testone.txt` and `testtwo.txt` then save it in the same folder where the your `main()` program is or provide the full path strings if the files is in another folder. Then for file `testtwo.txt`, write the following texts and save it

Content of file `testtwo.txt`

- Then, if you run the program with no error, modify the content of the `testtwo.txt`, recompile and rerun the program. The displaying texts and the content of `testone.txt` also will change.
- Next check also the content of `testone.txt` file, the content should be same as `testtwo.txt` file and the texts displayed on your screen.

```
1.    //Reading and writing one character at a time
2.    #include <stdio.h>
3.    #include <stdlib.h>
4.
5.    //enumerated data type, SUCCESS = 0, FAIL = 1
6.    enum  {SUCCESS, FAIL};
7.
8.    //prototype function for reading from and writing...
9.    void CharReadWrite(FILE *fin, FILE *fout);
10.
11.   int main()
12.   {
13.      //declare two file pointers...
14.      FILE *fptr1, *fptr2;
15.      //define the two files name...
16.      char filename1[] = "testone.txt";
17.      char filename2[] = "testtwo.txt";
18.      int reval = SUCCESS;
19.
20.      //test the opening filename1 for writing....
21.      //if fails....
22.      if ((fptr1 = fopen(filename1, "w")) == NULL)
23.      {
24.           printf("Problem, cannot open %s.\n", filename1);
25.           reval = FAIL;
26.      }
```

```
27.      //if opening filename1 for writing is successful,
28.      //test for opening for reading filename2, if fails...
29.      else if ((fptr2 = fopen(filename2, "r")) == NULL)
30.      {
31.           printf("Problem, cannot open %s.\n", filename2);
32.
33.           reval = FAIL;
34.      }
35.      //if successful opening for reading from filename2
36.      //and writing to filename1...
37.      else
38.      {
39.           //function call for reading and writing...
40.           CharReadWrite(fptr2, fptr1);
41.           //close both files...
42.           if(fclose(fptr1)==0)
43.            printf("%s closed successfully\n", filename1);
44.           if(fclose(fptr2)==0)
45.            printf("%s closed successfully\n", filename2);
46.      }
47.   //For Borland if compiled using its IDE…
48.   system("pause");
49.   return reval;
50.   }
51.
52.   //read write function definition
53.   void  CharReadWrite(FILE *fin, FILE *fout)
54.   {
55.      int  c;
56.      //if the end of file is reached, do...
57.      while ((c =           ) != EOF)
58.      {
59.           //write to a file...
60.                         ;
61.           //display on the screen...
62.           putchar(c);
63.      }
64.      printf("\n");
65.   }
```

65 lines: Output:

- This program read one character from a file, writes the character to another file, and then display the character on the screen.

9.6.2.2 One Line At A Time

- Besides reading or writing one character at a time, you can also read or write one character line at time. There is a pair of C I/O functions, `fgets()` and `fputs()`, that allows you to do so.
- The prototype for the `fgets()` function is:

```
char *fgets(char *s, int n, FILE *stream);
```

- `s`, references a character array that is used to store characters read from the opened file pointed to by the file pointer `stream`. `n` specifies the maximum number of array elements. If it is successful, the `fgets()` function returns the `char` pointers `s`. If `EOF` is encountered, the `fgets()` function

returns a null pointer and leaves the array untouched. If an error occurs, the function returns a null pointer, and the contents of the array are unknown.

- The `fgets()` function can read up to `n-1` characters, and can append a null character after the last character fetched, until a newline or an `EOF` is encountered.
- If a newline is encountered during the reading, the `fgets()` function includes the newline in the array. This is different from what the `gets()` function does. The `gets()` function just replaces the newline character with a null character.
- The prototype for the `fputs()` function is:

```
int fputs(const char *s, FILE *stream);
```

- `s` points to the array that contains the characters to be written to a file associated with the file pointer `stream`. The `const` modifier indicates that the content of the array pointed to by `s` cannot be changed. If it fails, the `fputs()` function returns a nonzero value, otherwise, it returns zero.
- The character array must include a null character at the end as the terminator to the `fputs()` function. Also, unlike the `puts()` function, the `fputs()` function does not insert a newline character to the string written to a file.
- Let try a program example. First of all, create two text file named `testhree.txt` and `testfour.txt` and put it under folder `C:\`. File `testfour.txt` should contain the following texts:

Content of `testfour.txt` file

```
1.    //Reading and writing one line at a time
2.    #include <stdio.h>
3.    #include <stdlib.h>
4.
5.    enum  {SUCCESS, FAIL, MAX_LEN = 100};
6.
7.    //function prototype for read and writes by line...
8.    void  LineReadWrite(FILE *fin, FILE *fout);
9.
10.   int main(void)
11.   {
12.   FILE *fptr1, *fptr2;
13.   //file testhree.txt is located at the root, c:
14.   //you can put this file at any location provided
15.   //you provide the full path, same for testfour.txt
16.   char filename1[] = "c:\\testhree.txt";
17.   char  filename2[] = "c:\\testfour.txt";
18.   char  reval = SUCCESS;
19.
20.   //test opening testhree.txt file for writing, if fail...
21.   if((fptr1 = fopen(filename1,"w")) == NULL)
22.   {
23.         printf("Problem, cannot open %s for writing.\n", filename1);
24.         reval = FAIL;
25.   }
26.
27.   //test opening testfour.txt file for reading, if fail...
28.   else if((fptr2=fopen(filename2, "r"))==NULL)
29.   {
30.         printf("Problem, cannot open %s for reading.\n", filename2);
31.         reval = FAIL;
32.   }
33.
34.   //if opening fro writing and reading successful, do...
35.   else
36.   {
37.         //function call for read and write, line by line...
38.         LineReadWrite(fptr2, fptr1);
```

```
39.         //close both files stream...
40.         if(fclose(fptr1)==0)
41.          printf("%s successfully closed.\n", filename1);
42.         if(fclose(fptr2)==0)
43.          printf("%s successfully closed.\n", filename2);
44.   }
45.   //For Borland screenshot
46.   system("pause");
47.   return reval;
48.   }
49.
50.   //function definition for line read, write…
51.   void LineReadWrite(FILE *fin, FILE *fout)
52.   {
53.   //local variable...
54.   char buff[MAX_LEN];
55.   while(                          !=NULL)
56.   {
57.         //write to file...
58.                          ;
59.         //write to screen...
60.         printf("%s", buff);
61.   }
62.   }
```

62 lines: Output:

- In this program example, the text files are located in `C:\` drive. The `fgets()` function is called repeatedly in a while loop to read one line of characters at a time from the `testfour.txt` file, until it reaches the end of the text file.
- In line 54, the array name `buff` and the maximum number of the array elements `MAX_LEN` are passed to the `fgets()` function, along with the file pointer `fin` that is associated with the opened `testfour.txt` file.
- Meanwhile, each line read by the `fgets()` function is written to another opened text file called `testhree.txt` that is associated with the file pointer `fout`. This is done by invoking the `fputs()` function in line 58.
- The statement in line 60 prints the contents of each string on the screen so that you see the contents of the `testfour.txt` file. You also can view the `testhree.txt` file content in a text editor to make sure that the contents of the `testfour.txt` file have been copied to the `testhree.txt` file.

9.6.2.3 One Block At A Time

- You can also read or write a block of data at a time. There are two C I/O functions, `fread()` and `fwrite()`, that can be used to perform block I/O operations.
- The prototype for the `fread()` function is:

```
size_t fread(void *ptr, size_t size, size_t n,
FILE *stream);
```

- The `ptr` is a pointer to an array in which the data is stored. `size` indicates the size of each array element. `n` specifies the number of elements to be read. `stream` is a file pointer that is associated with the opened file for reading.

- `size_t` is an integral type defined in the header file `stdio.h`. The `fread()` function returns the number of elements actually read.
- The number of elements read by the `fread()` function should be equal to the value specified by the third argument to the function, unless an error occurs or an `EOF` is encountered.
- The `fread()` function returns the number of elements that are actually read, if an error occurs or an `EOF` is encountered.
- The prototype for the `fwrite()` function is:

```
size_t fwrite(const void *ptr, size_t size,
size_t n, FILE *stream);
```

- `ptr` references the array that contains the data to be written to an opened file pointed to by the file pointer `stream`. `size` indicates the size of each element in the array. `n` specifies the number of elements to be written.
- The `fwrite()` function returns the number of elements actually written.
- If there is no error occurring, the number returned by `fwrite()` should be the same as the third argument in the function. The return value may be less than the specified value if an error occurs.
- That is the programmer's responsibility to ensure that the array is large enough to hold data for either the `fread()` function or the `fwrite()` function.
- In C, a function called `feof()` can be used to determine when the end of a file is encountered. This function is more useful when you are reading a binary file because the values of some bytes may be equal to the value `EOF`.
- If you determine the end of a binary file by checking the value returned by `fread()`, you may end up at the wrong position.
- Using the `feof()` function helps you to avoid mistakes in determining the end of a file. The prototype for the `feof()` function is:

```
int feof(FILE *stream);
```

- Here, `stream` is the file pointer that is associated with an opened file. The `feof()` function returns `0` if the end of the file has not been reached, otherwise, it returns a nonzero integer.
- Let take a look at the program example. Create two files named it `testfive.txt` and `testsix.txt` in `C:\Temp` folder or other folder that you choose provided that you provide the full path strings in the program. Write the following texts into `testsix.txt` file and save it.

Content of `testsix.txt` file

```
1.    //Reading and writing one block at a time
2.    #include <stdio.h>
3.    #include <stdlib.h>
4.
5.    //declare enum data type, you will this
6.    //learn in other module...
7.    enum  {SUCCESS, FAIL, MAX_LEN = 80};
8.
9.    //function prototype for block reading and writing
10.   void  BlockReadWrite(FILE  *fin, FILE  *fout);
11.   //function prototype for error messages...
12.   int   ErrorMsg(char  *str);
13.
14.   int main(void)
15.   {
16.         FILE  *fptr1, *fptr2;
17.         //define the filenames...
18.         //the files location is at c:\Temp
19.         char  filename1[] = "c:\\Temp\\testfive.txt";
20.         char  filename2[] = "c:\\Temp\\testsix.txt";
21.         int  reval = SUCCESS;
```

```
22.
23.         //test opening testfive.txt file for writing, if fail...
24.         if((fptr1 = fopen(filename1, "w")) == NULL)
25.         {
26.              reval = ErrorMsg(filename1);
27.         }
28.
29.         //test opening testsix.txt file for reading, if fail...
30.         else if ((fptr2 = fopen(filename2, "r")) == NULL)
31.         {
32.              reval = ErrorMsg(filename2);
33.         }
34.         //if opening files for writing and reading is successful, do...
35.         else
36.         {
37.              //call function for reading and writing
38.              BlockReadWrite(fptr2, fptr1);
39.              //close both files streams...
40.              if(fclose(fptr1)==0)
41.               printf("%s successfully closed\n", filename1);
42.              if(fclose(fptr2)==0)
43.               printf("%s successfully closed\n", filename2);
44.         }
45.         printf("\n");
46.         //for Borland...
47.         system("pause");
48.         return  reval;
49.   }
50.
51.   //function definition for block read, write
52.   void   BlockReadWrite(FILE  *fin, FILE  *fout)
53.   {
54.         int   num;
55.         char  buff[MAX_LEN + 1];
56.         //while not end of file for input file, do...
57.         while(!feof(fin))
58.         {
59.              //reading...
60.              num =                                          ;
61.              //append a null character
62.              buff[num * sizeof(char)] =  \0 ;
63.              printf("%s", buff);
64.              //writing...
65.                                                  ;
66.         }
67.   }
68.
69.   //function definition for error message
70.   int  ErrorMsg(char  *str)
71.   {
72.         //display the error message...
73.         printf("Problem, cannot open %s.\n", str);
74.         return    FAIL;
75.   }
```

75 lines: Output:

- Note the use of `fread()` and `fwrite()` functions in the program. This program shows you how to invoke the `fread()` and `fwrite()` to perform block I/O operations.

- The `testsix.txt` file is read by the `fread()` function, and the `fwrite()` function used to write the contents read from `testsix.txt` to another file called `testfive.txt`.

9.7 Random Access To Disk Files

- Before this you have learned how to read or write data sequentially. In many cases, however, you may need to access particular data somewhere in the middle of a disk file.
- Random access is another way to read and write data to disk file. Specific file elements can be accessed in random order.
- There are two C I/O functions, `fseek()` and `ftell()`, that are designed to deal with random access.
- You can use the `fseek()` function to move the file position indicator to the spot you want to access in a file. The prototype for the `fseek()` function is:

```
int fseek(FILE *stream, long offset, int whence);
```

- `stream` is the file pointer with an opened file. `offset` indicates the number of bytes from a fixed position, specified by `whence`, that can have one of the following integral values represented by `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.
- If it is successful, the `fseek()` function return `0`, otherwise the function returns a nonzero value.
- `whence` provides the offset bytes from the file location. `whence` must be one of the values 0, 1, or 2 which represent three symbolic constants (defined in `stdio.h`) as follows:

Constants	whence	File location
SEEK_SET	0	File beginning
SEEK_CUR	1	Current file pointer position
SEEK_END	2	End of file

Table 9.10: offset bytes

- If `SEEK_SET` is chosen as the third argument to the `fseek()` function, the offset is counted from the beginning of the file and the value of the offset is greater than or equal to zero.
- If however, `SEEK_END` is picked up, then the offset starts from the end of the file, the value of the offset should be negative.
- When `SEEK_CUR` is passed to the `fseek()` function, the offset is calculated from the current value of the file position indicator.
- You can obtain the value of the current position indicator by calling the `ftell()` function. The prototype for the `ftell()` function is,

```
long ftell(FILE *stream);
```

- `stream` is the file pointer associated with an opened file. The `ftell()` function returns the current value of the file position indicator.
- The value returned by the `ftell()` function represents the number of bytes from the beginning of the file to the current position pointed to by the file position indictor.
- If the `ftell()` function fails, it returns `–1L` (that is, a long value of minus 1). Let explore the program example. Create and make sure text file named `tesseven.txt` is located in the `C:\Temp` folder before you can execute the program. The contents of the `tesseven.txt` is,

The content of `tesseven.txt` file

```
1.    //Random access to a file
2.    #include <stdio.h>
3.    #include <stdlib.h>
4.
5.    enum  {SUCCESS, FAIL, MAX_LEN = 120};
6.
7.    //function prototypes, seek the file position indicator
8.    void  PtrSeek(FILE  *fptr);
```

```
9.    //function prototype, tell the file position indicator…
10.   long  PtrTell(FILE  *fptr);
11.   //function prototype read and writes…
12.   void  DataRead(FILE  *fptr);
13.   int   ErrorMsg(char  *str);
14.
15.   int main(void)
16.   {
17.         FILE  *fptr;
18.         char filename[] = "c:\\Temp\\tesseven.txt";
19.         int  reval  = SUCCESS;
20.
21.         //if there is some error opening file for reading…
22.         if((fptr = fopen(filename, "r")) == NULL)
23.         {
24.              reval = ErrorMsg(filename);
25.         }
26.         //if opening is successful…
27.         else
28.         {
29.              //PtrSeek() function call…
30.              PtrSeek(fptr);
31.              //close the file stream…
32.              if(fclose(fptr)==0)
33.               printf("%s successfully closed.\n", filename);
34.         }
35.          //for Borland...
36.         system("pause");
37.         return  reval;
38.   }
39.
40.   //PtrSeek() function definition
41.   void  PtrSeek(FILE  *fptr)
42.   {
43.         long   offset1, offset2, offset3, offset4;
44.
45.         offset1 = PtrTell(fptr);
46.         DataRead(fptr);
47.         offset2 = PtrTell(fptr);
48.         DataRead(fptr);
49.         offset3 = PtrTell(fptr);
50.         DataRead(fptr);
51.         offset4 = PtrTell(fptr);
52.         DataRead(fptr);
53.
54.         printf("\nReread the tesseven.txt, in random order:\n");
55.         //reread the 2nd line of the tesseven.txt
56.         fseek(fptr, offset2, SEEK_SET);
57.         DataRead(fptr);
58.         //reread the 1st line of the tesseven.txt
59.         fseek(fptr, offset1, SEEK_SET);
60.         DataRead(fptr);
61.         //reread the 4th line of the tesseven.txt
62.         fseek(fptr, offset4, SEEK_SET);
63.         DataRead(fptr);
64.         //reread the 3rd line of the tesseven.txt
65.         fseek(fptr, offset3, SEEK_SET);
66.         DataRead(fptr);
67.   }
68.
69.   //PtrTell() function definition
70.   long   PtrTell(FILE  *fptr)
71.   {
72.         long  reval;
73.         //tell the fptr position…
74.         reval = ftell(fptr);
75.         printf("The fptr is at %ld\n", reval);
76.         return  reval;
77.   }
78.
79.   //DataRead() function definition
80.   void  DataRead(FILE  *fptr)
81.   {
82.         char  buff[MAX_LEN];
83.         //reading line of text at the fptr position…
84.         fgets(buff, MAX_LEN, fptr);
85.         //and display the text…
86.         printf("-->%s\n", buff);
87.   }
88.
```

```
89.   //Error message function definition
90.   int  ErrorMsg(char *str)
91.   {
92.         //display this error message…
93.         printf("Problem, cannot open %s.\n", str);
94.         return FAIL;
95.   }
```

95 lines: Output:

- We try to open the `tesseven.txt` file for reading by calling the `fopen()` function. If successful, we invoke the `PtrSeek()` function with the `fptr` file pointer as the argument in line 30.

```
PtrSeek(fptr);
```

- The definition of our first function `PtrSeek()` is shown in lines 41-67. The statement in line 45 obtains the original value of the `fptr` file pointer by calling another function, `PtrTell()`, which is defined in lines 70–77.
- The `PtrTell()` function can find and print out the value of the file position indicator with the help of the `ftell()` function.
- The third function, `DataRead()` is called to read one line of characters from the opened file and print out the line of characters on the screen. Line 47 gets the new value of the `fptr` file position indicator right after the reading and assigns the value to another `long` variable, `offset2`.
- Then the `DataRead()` function in line 48 reads the second line of characters from the opened file. Line 49 obtains the value of the file position indicator that points to the first byte of the third line and assigns the value to the third `long` variable `offset3` and so on for the fourth line of text.
- Line 50 calls the `DataRead()` function to read the third line and print it out on the screen.
- From the first portion of the output, you can see the four different values of the file position indicator at four different positions, and the four lines of texts. The four values of the file position indicator are saved by `offset1`, `offset2`, `offset3` and `offset4` respectively.
- Then, we read the lines of text randomly, one line at a time. Firstly read the second line, then the first line, fourth and finally the third one.
- C function, called `rewind()`, can be used to rewind the file position indicator. The prototype for the `rewind()` function is:

```
void rewind(FILE *stream);
```

- Here, `stream` is the file pointer associated with an opened file. No value is returned by `rewind()` function. In fact the following statement of `rewind()` function:

```
rewind(fptr);
```

- Is equivalent to this:

```
(void) fseek(fptr, 0L, SEEK_SET);
```

- The `void` data type is cast to the `fseek()` function because the `rewind()` function does not return a value. Study the following program example.
- This program also contains example of reading and writing binary data. We create and open the `teseight.bin` file for writing.

```
1.   //Reading, writing, rewind and binary data
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   enum   {SUCCESS, FAIL, MAX_NUM = 5};
6.
7.   //functions prototype...
8.   void  DataWrite(FILE  *fout);
9.   void  DataRead(FILE  *fin);
10.  int   ErrorMsg(char  *str);
11.
12.  int main(void)
13.  {
14.       FILE  *fptr;
15.       //binary type files...
16.       char  filename[] = "c:\\Temp\\teseight.bin";
17.       int  reval = SUCCESS;
18.
19.       //test for creating, opening binary file for writing...
20.       if((fptr = fopen(filename, "wb+")) == NULL)
21.       {
22.            reval   =  ErrorMsg(filename);
23.       }
24.       else
25.       {
26.            //Write data into file teseight.bin
27.            DataWrite(fptr);
28.            //reset the file position indicator...
29.            rewind(fptr);
30.            //read data...
31.            DataRead(fptr);
32.            //close the file stream...
33.            if(fclose(fptr)==0)
34.             printf("%s successfully closed\n", filename);
35.       }
36.       //for Borland
37.       system("pause");
38.       return  reval;
39.  }
40.
41.  //DataWrite() function definition
42.  void  DataWrite(FILE  *fout)
43.  {
44.       int   i;
45.       double   buff[MAX_NUM] = { 145.23, 589.69, 122.12, 253.21, 987.234};
46.
47.       printf("The size of buff: %d-byte\n", sizeof(buff));
48.       for(i=0; i<MAX_NUM; i++)
49.       {
50.            printf("%5.2f\n", buff[i]);
51.                                                    ;
52.       }
53.  }
54.
55.  //DataRead() function definition
56.  void  DataRead(FILE  *fin)
57.  {
58.       int   i;
59.       double  x;
60.
61.       printf("\nReread from the binary file:\n");
62.       for(i=0; i<MAX_NUM; i++)
63.       {
64.                                                    ;
65.            printf("%5.2f\n", x);
66.       }
```

```
67.   }
68.
69.   //ErrorMsg() function definition
70.   int  ErrorMsg(char  *str)
71.   {
72.        printf("Cannot open %s.\n", str);
73.        return  FAIL;
74.   }
```

74 lines
Output:

- This program writes five values of the `double` data type into a binary file named `teseight.bin` and then rewind the file position indicator and re read the five double values from the binary file.
- The two functions, `DataWrite()` and `DataRead()`, that perform the writing and reading, declared in lines 8 and 9. The `enum` names, `SUCCESS`, `FAIL`, and `MAX_NUM`, are defined in line 5 with values 0, 1, and 5 respectively.
- The statement in line 20, tries to create and open a binary file called `teseight.bin` for both reading and writing.
- If the `fopen()` function is successful, the `DataWrite()` function is called in line 27 to write four `double` data items, into the opened binary file, according to the definition of the `DataWrite()` function.
- The `fwrite()` function in line 51 does the writing. Right after the execution of the `DataWrite()` function, the file position indicator is reset to the beginning of the binary file by calling the `rewind()` function in line 29 because we want to re read all five double data items written to the file.
- The `fread()` function is used to perform the reading operation. The output from running the program shows the five `double` data items before the writing and after the reading as well.
- As you learned, two C library functions `scanf()` and `printf()` can be used to read or write formatted data through the standard I/O (that is, `stdin` and `stdout`). For C disk file I/O functions, there are two equivalent functions; `fscanf()` and `fprintf()` functions allow the programmer to specify I/O streams.
- The prototype for the `fscan()` function is:

```
int fscanf(FILE *stream, const char *format,…);
```

- `stream` is the file pointer associated with an opened file. `format`, which usage is the same as in the `scanf()` function, is a char pointer pointing to a string that contains the format specifiers. If successful, the `fscan()` function returns the number of data items read. Otherwise, the function returns `EOF`.
- The prototype for the `fprintf()` function is:

```
int fprintf(FILE *stream, const char *format, …);
```

- Here, `stream` is the file pointer associated with an opened file. `format`, whose usage is the same as in the `printf()` function, is a `char` pointer pointing to a string that contains the format specifiers.
- If successful, the `fprintf()` function returns the number of formatted expressions. Otherwise, the function returns a negative value.
- Let try a program example. Firstly create `testcal.txt` file with the following data and save it.

- Then create another text file named `testavg.txt` for writing the average value computed from data read from `testcal.txt` file. Then compile and run the following program.

```
/*C Program to calculate the average of a list of numbers.*/
/*calculate the total from one file, output the average*/
/*into another file*/
#include <stdio.h>
/*for exit()*/
#include <stdlib.h>

int main(void)
{
int value, total = 0, count = 0;

/*fileptrIn and fileptrOut are variables of type (FILE *)*/
FILE * fileptrIn, * fileptrOut;
char filenameIn[100], filenameOut[100];

printf("Please enter an input filename (use path if needed):\n");
scanf("%s", filenameIn);
printf("Please enter an output filename (use path if needed):\n");
scanf("%s", filenameOut);

/*open files for reading, "r" and writing, "w"*/
if((fileptrIn = fopen(filenameIn, "r")) == NULL)
{
    printf("Error opening %s for reading.\n", filenameIn);
    exit (1);
}
else
printf("Opening %s for reading is OK.\n", filenameIn);

if((fileptrOut = fopen(filenameOut, "w")) == NULL)
{
    printf("Error opening %s for writing.\n", filenameOut);
    exit (1);
}
else
printf("Opening %s for writing is OK.\n", filenameOut);

/*fscanf*/
printf("\nCalculate the total...\n");
while(EOF != fscanf(fileptrIn, "%i", &value))
{
    total += value;
    ++count;
}/*end of while loop*/

/*Write the average value to the file.*/
/*fprintf*/
printf("Calculate the average...\n\n");
fprintf(fileptrOut, "Average of %i numbers = %f \n", count, total/(double)count);
printf("Average of %i numbers = %f \n\n", count, total/(double)count);
printf("Check also your %s file content\n", filenameOut);

if(fclose(fileptrIn) == 0)
printf("%s closed successfully\n", filenameIn);
if(fclose(fileptrOut) == 0)
printf("%s closed successfully\n", filenameOut);
return 0;
}
```

Output:

9.8 Redirecting The Standard Streams With `freopen()`

- We will discuss how to redirect the standard streams, such as `stdin` and `stdout`, to disk files. We can use `freopen()` function, which can associate a standard stream with a disk file.
- The prototype for the `freopen()` function is:

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

- `filename` is a `char` pointer referencing the name of a file that you want to associate with the standard stream represented by `stream`.
- `mode` is another `char` pointer pointing to a string that defines the way to open a file. The values that `mode` can have in `freopen()` are the same as the `mode` values in the `fopen()` function.
- The `freopen()` function returns a null pointer if an error occurs. Otherwise, the function returns the standard stream that has been associated with a disk file identified by `filename`.
- Let try a program example.

```
1.   //Redirecting a standard stream
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   enum  {SUCCESS, FAIL, STR_NUM = 6};
6.
7.   void  StrPrint(char  **str);
8.   int   ErrorMsg(char  *str);
9.
10.  int main(void)
11.  {
12.       //declare and define a pointer to string...
13.       char   *str[STR_NUM] = {
14.            "Redirecting a standard stream to the text file.",
15.            "These 5 lines of text will be redirected",
16.            "so many things you can do if you understand the",
17.            "concept, fundamental idea - try this one!",
18.            "--------------DONE-------------------------"};
19.
20.       char filename[] = "c:\\Temp\\testnine.txt";
21.       int  reval = SUCCESS;
22.
23.       StrPrint(str);
24.       //create file if not exist and open for writing...
25.       //if exist, discard the previous content...
26.       if(       (filename, "w", stdout) == NULL)
27.       {
28.            reval = ErrorMsg(filename);
29.       }
30.       else
31.       {
32.            //call StrPrint() function...
33.            StrPrint(str);
34.            //close the standard output...
35.            fclose(stdout);
36.        }
37.       return reval;
38.  }
```

```
39.
40.  //StrPrint() function definition
41.  void  StrPrint(char  **str)
42.  {
43.       int  i;
44.       for(i=0; i<STR_NUM; i++)
45.            //to standard output-screen/console...
46.            printf("%s\n", str[i]);
47.       system("pause");
48.  }
49.
50.  //ErrorMsg() function definition
51.  int  ErrorMsg(char  *str)
52.  {
53.       printf("Problem, cannot open %s.\n", str);
54.       return  FAIL;
55.  }
```

55 lines: Output:

- Notice that the last line in the output is `NULL`, why? Because `NULL` is appended at the end of the string. We enumerate `STR_NUM = 6`, but there are only 5 lines of text, if you don't want to see the `NULL`, change `STR_NUM = 5`.
- The purpose of this program is to save a paragraph, consist of five lines of text, into a text file, testnine.txt. We call the `printf()` function instead of the `fprintf()` function or other disk I/O functions after we redirect the default stream, stdout, of the `printf()` function to point to the text file.
- The function that actually does the writing is called `StrPrint()`, which invoke the C function `printf()` to send out formatted character strings to the output stream.
- In `main()` function, we call the `StrPrint()` function in line 33 before we redirect stdout to the `testnine.txt` file. The paragraph is printed on the screen because the `printf()` function automatically sends out the paragraph to `stdout` that directs to the screen by default.
- Then in line 26, we redirect `stdout` to the `testnine.txt` text file by calling the `freopen()` function. The "`w`" is used as the mode that indicates to open the text file for writing.
- If `freopen()` is successful, we then call the `StrPrint()` function in line 33. However, this time, the `StrPrint()` function writes the paragraph into the opened text file, `testnine.txt`. The reason is that `stdout` is now associated with the text file, not the screen.
- There is a set of low-level I/O functions, such as `open()`, `create()`, `close()`, `read()`, `write()`, `lseek()` and `tell()` that you may still see them in some platform-dependent `C` programs.

9.9 File Management Functions

- It refers to dealing with existing files, not reading or writing to them, but renaming, deleting and copying them. Normally the file management functions are provided in the standard library function. Again, do not reinvent the wheels :o).

9.9.1 Deleting A File

- We use function `remove()` to delete a file. Its prototype is in `stdio.h` file and the prototype is as follows:

```
int remove(const char *filename);
```

- The variable `filename` is a pointer to the name of the file to be deleted. The specified file must not be opened. If the file exists, it is deleted (just as if the `del` in DOS and `rm` command in UNIX), and `remove()` return 0.
- If the file doesn't exist, if it's read only, if you don't have sufficient access rights or permission, or if some other error occurs, `remove()` return `–1`. Be careful if you remove a file, it is gone forever.
- Let try a program example.

```
1.   //Demonstrate the remove() function
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   void main()
6.   {
7.        //declare an array to store file name...
8.        char  filename[80];
9.
10.       printf("Enter the filename to be deleted:  ");
11.       gets(filename);
12.
13.       //check any error...
14.       if(remove(filename) == 0)
15.       printf("File %s  has been deleted.\n", filename);
16.       else
17.       fprintf(stderr, "Error deleting file %s.\n", filename);
18.       system("pause");
19.  }
```

19 lines: Output:

- This program prompts the user on line 10 for the file name to be deleted. Line 14 then calls `remove()` to delete the entered file. If the return value is 0, the file was removed, and a message is displayed stating this fact. If the return value is not zero, an error occurred, and the file was not removed.

9.9.2 Renaming A File

- The `rename()` function changes the name of an existing disk file. The function prototype, in `stdio.`h, is as follows:

```
int rename(const char *oldname, const char *newname);
```

- Both names must refer to the same disk drive; you can't rename a file to a different disk drive means if the old name is in drive `C:\test.txt`, you can't rename it to `D:\testnew.txt`.
- The function `rename()` returns 0 on success, or –1 if an error occurs. Errors can be caused by the following conditions (among others):

    0. The file `oldname` does not exist.
    0. A file with the name `newname` already exists.
    0. You try to rename to another disk.

- Let take a look at a program example.

```
1.   //Using rename() to change a filename
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   void main()
6.   {
7.        char  oldname[80], newname[80];
8.
```

```
9.        printf("Enter current filename: ");
10.       gets(oldname);
11.       printf("Enter new name for file: ");
12.       gets(newname);
13.
14.       if(rename(oldname, newname) == 0)
15.       {
16.            printf("%s has been rename %s.\n", oldname, newname);
17.       }
18.       else
19.       {
20.       fprintf(stderr, "An error has occurred renaming %s.\n", oldname);
21.       }
22.       system("pause");
23.  }
```

23 lines: Output:

- This program example, with only 23 lines of code, replaces an operating system command `rename`, and it's a much friendlier function. Line 9 prompts for the name of the file to be renamed. Line 11 prompts for the new filename.
- The `if` statement checks to ensure that the renaming of the file was carried out correctly. If so, line 16 prints an affirmative message, otherwise, line 20 prints a message stating that there was an error.

9.7.2 Copying A File

- Copying a file performs an exact duplicate with a different name (or with the same name but in a different drive or directory). There are no library functions; you have to write your own.
- The steps:

    0. Open the source file for reading in binary mode, using binary mode ensures that the function can copy all sorts of content, not just texts.
    0. Open the destination file for writing in binary mode.
    0. Read a character from the source file. When a file is first opened, the pointer is at the start of the file, so there is no need to position the file pointer explicitly.
    0. If the function `feof()` indicates that you're reached the end of the source file, you're done and can close both files and return to the calling program.
    0. If you haven't reached end-of-file, write the character to the destination file, and then loop back to step 3.

- Let try a program example.

```
1.   //Copying a file
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   int  file_copy(char   *oldname, char   *newname);
6.
7.   void  main()
8.   {
9.        char  source[80], destination[80];
10.
11.       //get the source and destination names
12.       printf("\nEnter source file: ");
13.       gets(source);
14.       printf("\nEnter destination file: ");
15.       gets(destination);
16.
17.       if(file_copy(source, destination) == 0)
18.            puts("Copy operation successful");
19.       else
20.            fprintf(stderr, "Error during copy operation");
```

```
21.       system("pause");
22.  }
23.
24.  int  file_copy(char *oldname, char  *newname)
25.  {
26.       FILE  *fold, *fnew;
27.       int  c;
28.
29.       //Open the source file for reading in binary mode
30.       if((fold = fopen(oldname, "rb")) == NULL)
31.       return  -1;
32.       //Open the destination file for writing in binary mode
33.       if((fnew = fopen(newname, "wb" )) == NULL)
34.       {
35.            fclose(fold);
36.            return  -1;
37.       }
38.
39.  //Read one byte at a time from the source, if end of file
40.  //has not been reached, write the byte to the destination
41.       while(1)
42.       {
43.            c  =  fgetc(fold);
44.
45.            if(!feof(fold))
46.                 fputc(c, fnew);
47.            else
48.                 break;
49.       }
50.       fclose(fnew);
51.       fclose(fold);
52.       return  0;
53.  }
```

53 lines: Output:

- The `file_copy()` function let you copy anything from a small text file to a huge program file. But for this program, if the destination file already exists, the function overwrites it without asking.
- Lines 24 through 37 create a copy function. Line 30 open the source file, pointed by `fold` pointer, in binary read mode.
- Line 33 open the destination file, pointed by `fnew` pointer, in binary write mode. Line 35 closes the source file if there is an error opening the destination file. The while loop does the actual copying of the file. Line 43 gets a character from the source file, pointed by `fold` pointer assign to the variable `c`.
- Line 45 tests to see whether the end-of-line marker was read. If the end of the file has been reached, a `break` statement is executed in order to get out of the `while` loop in line 48.
- If the end of the file has not been reached, the character is written to the destination file, pointed by `fnew` pointer in line 46.
- For C++ file I/O it is discussed in Module 19.
- The following is a previous C program example, read and write files under the current working directory using **`gcc`**. Create 2 files named `testhree.txt` and `testfour.txt` under the current working directory and save some texts in the `testfour.txt`.

```
/***************readline.c************
/*Reading and writing one line at a time*/
#include <stdio.h>
#include <stdlib.h>

enum  {SUCCESS, FAIL, MAX_LEN = 100};

/*function prototype for read and writes by line...*/
void LineReadWrite(FILE *fin, FILE *fout);
```

```
int main(void)
{
FILE *fptr1, *fptr2;
/*file testhree.txt is located at current directory.
  you can put this file at any location provided
  you provide the full path, same for testfour.txt*/

char filename1[] = "testhree.txt";
char filename2[] = "testfour.txt";
char reval = SUCCESS;

/*test opening testhree.txt file for writing, if fail...*/
if((fptr1 = fopen(filename1,"w")) == NULL)
{
        printf("Problem, cannot open %s for writing.\n", filename1);
        reval = FAIL;
}

/*test opening testfour.txt file for reading, if fail...*/
else if((fptr2=fopen(filename2, "r"))==NULL)
{
        printf("Problem, cannot open %s for reading.\n", filename2);
        reval = FAIL;
}

/*if opening fro writing and reading successful, do...*/
else
{
/*function call for read and write, line by line...*/
LineReadWrite(fptr2, fptr1);
/*close both files stream...*/
if(fclose(fptr1)==0)
printf("%s successfully closed.\n", filename1);
if(fclose(fptr2)==0)
  printf("%s successfully closed.\n", filename2);
}
return reval;
}

/*function definition for line read, write.*/
void LineReadWrite(FILE *fin, FILE *fout)
{
        /*local variable...*/
        char buff[MAX_LEN];
        while(fgets(buff, MAX_LEN, fin) !=NULL)
        {
                /*write to file...*/
                fputs(buff, fout);
                /*write to screen...*/
                printf("%s", buff);
        }
}

[bodo@bakawali ~]$ gcc readline.c -o readline
[bodo@bakawali ~]$ ./readline

------------------LINUX LOR!------------------------
------------FEDORA 3, gcc x.x.x--------------------
OPENING, READING, WRITING one line of characters
----------------------------------------------------
This is file testfour.txt.  This file's content will
be read line by line of characters till no more line
of character found.  Then, it will be output to the
screen and also will be copied to file testhree.txt.
Check the content of testhree.txt file...
----------------------------------------------------
------------------HAVE A NICE DAY-------------------

testhree.txt successfully closed.
testfour.txt successfully closed.
```

- Another program example for non-current directory files location. Our program under `/home/bodo/` directory but we try to create teseight.bin under `/testo1/testo2/` directory. You must have root privilege to create files in this case.

```
////////////rwbinary.c//////////
/////FEDORA 3, gcc x.x.x/////
```

```
//Reading, writing, rewind and binary data
#include <stdio.h>

enum   {SUCCESS, FAIL, MAX_NUM = 5};

//functions prototype...
void DataWrite(FILE  *fout);
void DataRead(FILE  *fin);
int   ErrorMsg(char  *str);

int main(void)
{
        FILE *fptr;
        //binary type files...
        char filename[] = "/testo1/testo2/teseight.bin";
        int reval = SUCCESS;

        //test for creating, opening binary file for writing...
        if((fptr = fopen(filename, "wb+")) == NULL)
        {
                reval   =  ErrorMsg(filename);
        }
        else
        {
                //Write data into file teseight.bin
                DataWrite(fptr);
                //reset the file position indicator...
                rewind(fptr);
                //read data...
                DataRead(fptr);
                //close the file stream...
                if(fclose(fptr) == 0)
           printf("%s successfully closed\n", filename);
        }
        return reval;
        }

//DataWrite() function definition
void DataWrite(FILE  *fout)
{
        int   i;
        double   buff[MAX_NUM] = {145.23, 589.69, 122.12, 253.21, 987.234};

        printf("The size of buff: %d-byte\n", sizeof(buff));
        for(i=0; i<MAX_NUM; i++)
        {
                printf("%5.2f\n", buff[i]);
                fwrite(&buff[i], sizeof(double), 1, fout);
        }
}

//DataRead() function definition
void DataRead(FILE  *fin)
{
        int   i;
        double x;

        printf("\nReread from the binary file:\n");
        for(i=0; i<MAX_NUM; i++)
        {
                fread(&x, sizeof(double), (size_t)1, fin);
                printf("%5.2f\n", x);
        }
}

//ErrorMsg() function definition
int ErrorMsg(char *str)
{
        printf("Cannot open %s.\n", str);
        return FAIL;
}

[root@bakawali bodo]# gcc rwbinary.c -o rwbinary
[root@bakawali bodo]# ./rwbinary

The size of buff: 40-byte
145.23
589.69
122.12
253.21
```

```
987.23

Reread from the binary file:
145.23
589.69
122.12
253.21
987.23
/testo1/testo2/teseight.bin successfully closed
```

Further readings

- The following sections compiled from GNU `glibc` library documentation, provide a summary and other collections that you may interested☺ related to file I/O. Sockets will be discussed in another Module. It looks that the file attributes also not discussed here.

A. Simple Output by Characters or Lines

- The following Table describes functions for performing character and line-oriented output.
- These narrow streams functions are declared in the header file `stdio.h` and the wide stream functions in `wchar.h`.

`int fputc(int c, FILE *stream)`
The `fputc()` function converts the character *c* to type `unsigned char`, and writes it to the stream *stream*. `EOF` is returned if a write error occurs; otherwise the character *c* is returned.
`wint_t fputwc(wchar_t wc, FILE *stream)`
The `fputwc()` function writes the wide character *wc* to the stream *stream*. `WEOF` is returned if a write error occurs; otherwise the character *wc* is returned.
`int fputc_unlocked(int c, FILE *stream)`
The `fputc_unlocked()` function is equivalent to the `fputc` function except that it does not implicitly lock the stream.
`int putc(int c, FILE *stream)`
This is just like `fputc()`, except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. `putc` is usually the best function to use for writing a single character.
`wint_t putwc(wchar_t wc, FILE *stream)`
This is just like `fputwc()`, except that it can be implement as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. `putwc()` is usually the best function to use for writing a single wide character.
`int putc_unlocked(int c, FILE *stream)`
The `putc_unlocked()` function is equivalent to the `putc` function except that it does not implicitly lock the stream.
`int putchar(int c)`
The `putchar()` function is equivalent to `putc` with `stdout` as the value of the *stream* argument.
`wint_t putwchar(wchar_t wc)`
The `putwchar()` function is equivalent to `putwc` with `stdout` as the value of the *stream* argument.
`int putchar_unlocked(int c)`
The `putchar_unlocked()` function is equivalent to the `putchar` function except that it does not implicitly lock the stream.
`int fputs(const char *s, FILE *stream)`

The function `fputs()` writes the string *s* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string.
This function returns `EOF` if a write error occurs, and otherwise a non-negative value.
For example:
 	
`fputs ("Are ", stdout);`
`fputs ("you ", stdout);`
`fputs ("hungry?\n", stdout);`
 	
Outputs the text `Are you hungry?` followed by a newline.

`int fputws(const wchar_t *ws, FILE *stream)`
The function `fputws()` writes the wide character string *ws* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string. This function returns `WEOF` if a write error occurs, and otherwise a non-negative value.
`int puts(const char *s)`
The `puts()` function writes the string *s* to the stream `stdout` followed by a newline. The terminating null

character of the string is not written. (Note that `fputs` does *not* write a newline as this function does.) `puts` is the most convenient function for printing simple messages. For example: `puts("This is a message.");` Outputs the text `This is a message.` followed by a newline.

Table 9.11: Output by characters or lines functions

B. Character Input

- This section describes functions for performing character-oriented input. These narrow streams functions are declared in the header file `stdio.h` and the wide character functions are declared in `wchar.h`.
- These functions return an `int` or `wint_t` value (for narrow and wide stream functions respectively) that is either a character of input, or the special value `EOF`/`WEOF` (usually -1). For the narrow stream functions it is important to store the result of these functions in a variable of type `int` instead of `char`, even when you plan to use it only as a character.
- Storing `EOF` in a `char` variable truncates its value to the size of a character, so that it is no longer distinguishable from the valid character `(char) -1`.
- So always use an `int` for the result of `getc` and friends, and check for `EOF` after the call; once you've verified that the result is not `EOF`, you can be sure that it will fit in a `char` variable without loss of information.

`int fgetc(FILE *stream)`
This function reads the next character as an `unsigned char` from the stream *stream* and returns its value, converted to an `int`. If an end-of-file condition or read error occurs, `EOF` is returned instead.
`wint_t fgetwc(FILE *stream)`
This function reads the next wide character from the stream *stream* and returns its value. If an end-of-file condition or read error occurs, `WEOF` is returned instead.
`int fgetc_unlocked(FILE *stream)`
The `fgetc_unlocked()` function is equivalent to the `fgetc()` function except that it does not implicitly lock the stream.
`int getc(FILE *stream)`
This is just like `fgetc()`, except that it is permissible (and typical) for it to be implemented as a macro that evaluates the *stream* argument more than once. `getc` is often highly optimized, so it is usually the best function to use to read a single character.
`wint_t getwc(FILE *stream)`
This is just like `fgetwc()`, except that it is permissible for it to be implemented as a macro that evaluates the *stream* argument more than once. `getwc()` can be highly optimized, so it is usually the best function to use to read a single wide character.
`int getc_unlocked(FILE *stream)`
The `getc_unlocked()` function is equivalent to the `getc` function except that it does not implicitly lock the stream.
`int getchar(void)`
The `getchar()` function is equivalent to `getc()` with `stdin` as the value of the *stream* argument.
`wint_t getwchar(void)`
The `getwchar()` function is equivalent to `getwc()` with `stdin` as the value of the *stream* argument.
`int getchar_unlocked(void)`
The `getchar_unlocked()` function is equivalent to the `getchar()` function except that it does not implicitly lock the stream.

Table 9.12: Character oriented input functions

- An example of a function that does input using fgetc, it would normally work just as well using `getc()` instead, or using `getchar()` instead of `fgetc(stdin)`. The code would also work for the wide character stream functions as well.

C. Line-Oriented Input

- Since many programs interpret input on the basis of lines, it is convenient to have functions to read a line of text from a stream. Standard C functions for these tasks aren't very safe: null characters and even (for `gets()`) long lines can confuse them.

- This vulnerability creates exploits through buffer overflows. That is why you see warning everywhere; you may check your implementation documentation for safer version of those functions. All these functions are declared in `stdio.h`.

`char * fgets(char *s, int count, FILE *stream)`
The `fgets()` function reads characters from the stream *stream* up to and including a newline character and stores them in the string *s*, adding a null character to mark the end of the string. You must supply *count* characters worth of space in *s*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null character at the end of the string. If the system is already at end of file when you call `fgets`, then the contents of the array *s* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *s*. **:** If the input data has a null character, you can't tell. So don't use `fgets` unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message. We recommend using `getline` instead of `fgets`.
`wchar_t * fgetws(wchar_t *ws, int count, FILE *stream)`
The `fgetws()` function reads wide characters from the stream *stream* up to and including a newline character and stores them in the string *ws*, adding a null wide character to mark the end of the string. You must supply *count* wide characters worth of space in *ws*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null wide character at the end of the string. If the system is already at end of file when you call `fgetws`, then the contents of the array *ws* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *ws*. **:** If the input data has a null wide character (which are null bytes in the input stream), you can't tell. So don't use `fgetws` unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message.
`char * gets(char *s)`
The function `gets()` reads characters from the stream `stdin` up to the next newline character, and stores them in the string *s*. The newline character is discarded (note that this differs from the behavior of `fgets`, which copies the newline character into the string). If `gets` encounters a read error or end-of-file, it returns a null pointer; otherwise it returns *s*. **:** The `gets` function is very dangerous because it provides no protection against overflowing the string *s*. The GNU library includes it for compatibility only. You should always use `fgets` or `getline` instead. To remind you of this, the linker (if using GNU `ld`) will issue a warning whenever you use `gets`.

Table 9.13: Line oriented input functions

D. Block Input/Output

- This section describes how to do the input and output operations on blocks of data. You can use these functions to read and write binary data, as well as to read and write text in fixed size blocks instead of by characters or lines.
- Binary files are typically used to read and write blocks of data in the same format as is used to represent the data in a running program.
- In other words, arbitrary blocks of memory, not just character or string objects, can be written to a binary file, and meaningfully read in again by the same program.
- Storing data in binary form is often considerably more efficient than using the formatted I/O functions.
- Also, for floating-point numbers, the binary form avoids possible loss of precision in the conversion process. On the other hand, binary files can't be examined or modified easily using many standard file utilities (such as text editors), and are not portable between different implementations of the language, or different kinds of computers.
- These functions are declared in `stdio.h`.

`size_t fread(void *data, size_t size, size_t count, FILE *stream)`
This function reads up to *count* objects of size *size* into the array *data*, from the stream *stream*. It returns the number of objects actually read which might be less than *count* if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either *size* or *count* is zero. If `fread` encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.
`size_t fwrite(const void *data, size_t size, size_t count, FILE *stream)`
This function writes up to *count* objects of size *size* from the array *data*, to the stream *stream*. The return value is normally *count*, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

Table 9.14: Block oriented I/O functions

E. Some File System Interfaces

E.1 Deleting Files

- You can delete a file with `unlink()` or `remove()`.
- Deletion actually deletes a file name. If this is the file's only name, then the file is deleted as well. If the file has other remaining names, it remains accessible under those names.

`int rmdir(const char *filename)`
The `rmdir()` function deletes a directory. The directory must be empty before it can be removed; in other words, it can only contain entries for `.` and `..`. In most other respects, `rmdir()` behaves like `unlink()`.
`int remove(const char *filename)`
This is the ISO C function to remove a file. It works like `unlink()` for files and like `rmdir()` for directories. `remove()` is declared in `stdio.h`.

Table 9.15: Remove directory and file functions

- The `rename()` function is used to change a file's name.

`int rename(const char *oldname, const char *newname)`
The `rename()` function renames the file *oldname* to *newname*. The file formerly accessible under the name *oldname* is afterwards accessible as *newname* instead. (If the file had any other names aside from *oldname*, it continues to have those names.) The directory containing the name *newname* must be on the same file system as the directory containing the name *oldname*. One special case for `rename` is when *oldname* and *newname* are two names for the same file. The consistent way to handle this case is to delete *oldname*. However, in this case POSIX requires that `rename` do nothing and report success--which is inconsistent. We don't know what your operating system will do. If *oldname* is not a directory, then any existing file named *newname* is removed during the renaming operation. However, if *newname* is the name of a directory, `rename` fails in this case. If *oldname* is a directory, then either *newname* must not exist or it must name a directory that is empty. In the latter case, the existing directory named *newname* is deleted first. The name *newname* must not specify a subdirectory of the directory `oldname` which is being renamed. One useful feature of `rename` is that the meaning of *newname* changes "atomically" from any previously existing file by that name to its new meaning (i.e. the file that was called *oldname*). There is no instant at which *newname* is non-existent "in between" the old meaning and the new meaning. If there is a system crash during the operation, it is possible for both names to still exist; but *newname* will always be intact if it exists at all.

Table 9.16: Rename function

E.2 Creating Directories

- Directories are created with the `mkdir` function. There is also a shell command `mkdir` which does the same thing.

`int mkdir(const char *filename, mode_t mode)`
The `mkdir()` function creates a new, empty directory with name *filename*. The argument *mode* specifies the file permissions for the new directory file.

Table 9.17: Create directory function

F. Pipes and FIFOs

- A `pipe` is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.
- A `FIFO special file` is similar to a pipe, but instead of being an anonymous, temporary connection, a `FIFO` has a name or names like any other file. Processes open the `FIFO` by name in order to communicate through it.

- A pipe or `FIFO` has to be open at both ends simultaneously. If you read from a pipe or `FIFO` file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file.
- Writing to a pipe or `FIFO` that doesn't have a reading process is treated as an error condition; it generates a `SIGPIPE` signal, and fails with error code `EPIPE` if the signal is handled or blocked.
- Neither pipes nor `FIFO` special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

---------o0o---------

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.
2. Wide character/Unicode is discussed HERE and the implementation using Microsoft C is discussed HERE.
3. Implementation specific information for Microsoft can be found HERE (CRT) and HERE (Win32).