PERL - FILE I/O

The basics of handling files are simple: you associate a **filehandle** with an external entity *usuallyafile* and then use a variety of operators and functions within Perl to read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR,** which represent standard input, standard output and standard error devices respectively.

Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

```
open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE
```

Here FILEHANDLE is the file handle returned by the **open** function and EXPR is the expression having file name and mode of opening the file.

Open Function

Following is the syntax to open **file.txt** in read-only mode. Here less than < sign indicates that file has to be opend in read-only mode.

```
open(DATA, "<file.txt");
```

Here DATA is the file handle which will be used to read the file. Here is the example which will open a file and will print its content over the screen.

```
#!/usr/bin/perl

open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";

while(<DATA>){
   print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend in the writing mode.

```
open(DATA, ">file.txt") or die "Couldn't open file file.txt, $!";
```

This example actually truncates *empties* the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it −

```
open(DATA, "+<file.txt"); or die "Couldn't open file file.txt, $!";
```

To truncate the file first −

```
open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode writing point will be set to the end of the file.

```
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it −

```
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table which gives the possible values of different modes.

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> or a+	Reads, Writes, Appends, and Creates

Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the system **open** function, using the parameters supplied to it as the parameters for the system function −

For example, to open a file for updating, emulating the **+<filename** format from open −

```
sysopen(DATA, "file.txt", O_RDWR);
```

Or to truncate the file before updating −

```
sysopen(DATA, "file.txt", O_RDWR|O_TRUNC );
```

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only mode and O_RDONLY - to open file in read only mode.

The **PERMS** argument specifies the file permissions for the file specified if it has to be created. By default it takes **0x666**.

Following is the table, which gives the possible values of MODE.

Entities	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only
O_CREAT	Create the file
O_APPEND	Append the file

O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
O_NONBLOCK	Non-Blocking usability

Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the **close** function. This flushes the filehandle's buffers and closes the system's file descriptor.

```
close FILEHANDLE
close
```

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

```
close(DATA) || die "Couldn't close file properly";
```

Reading and Writing Files

Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The <FILEHANDL> Operator

The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example −

```
#!/usr/bin/perl

print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array −

```
#!/usr/bin/perl

open(DATA,"<import.txt") or die "Can't open data";
@lines = <DATA>;
close(DATA);
```

getc Function

The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified −

```
getc FILEHANDLE
getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

read Function

The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

print Function

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

```
print FILEHANDLE LIST
print LIST
print
```

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle *STDOUTbydefault*. For example −

```
print "Hello World!\n";
```

Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generate another copy file file2.txt.

```
#!/usr/bin/perl

# Open file to read
open(DATA1, "<file1.txt");

# Open new file to write
open(DATA2, ">file2.txt");

# Copy data from one file to another.
while(<DATA1>)
{
   print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
#!/usr/bin/perl

rename ("/usr/test/file1.txt", "/usr/test/file2.txt" );
```

This function **renames** the takes two arguments and it just rename existing file.

Deleting an Existing File

Here is an example, which shows how to delete a file file1.txt using the **unlink** function.

```
#!/usr/bin/perl

unlink ("/usr/test/file1.txt");
```

Positioning inside a File

You can use to **tell** function to know the current position of a file and **seek** function to point a particular position inside the file.

tell Function

The first requirement is to find your position within a file, which you do using the tell function −

```
tell FILEHANDLE
tell
```

This returns the position of the file pointer, in bytes, within FILEHANDLE if specified, or the current default selected filehandle if none is specified.

seek Function

The seek function positions the file pointer to the specified number of bytes within a file −

```
seek FILEHANDLE, POSITION, WHENCE
```

The function uses the fseek system function, and you have the same ability to position relative to three different points: the start, the end, and the current position. You do this by specifying a value for WHENCE.

Zero sets the positioning relative to the start of the file. For example, the line sets the file pointer to the 256th byte in the file.

```
seek DATA, 256, 0;
```

File Information

You can test certain features very quickly within Perl using a series of test operators known collectively as -X tests. For example, to perform a quick test of the various permissions on a file, you might use a script like this −

```
#/usr/bin/perl

my $file = "/usr/test/file1.txt";
my (@description, $size);
if (-e $file)
{
   push @description, 'binary' if (-B _);
   push @description, 'a socket' if (-S _);
   push @description, 'a text file' if (-T _);
   push @description, 'a block special file' if (-b _);
   push @description, 'a character special file' if (-c _);
   push @description, 'a directory' if (-d _);
   push @description, 'executable' if (-x _);
   push @description, (($size = -s _)) ? "$size bytes" : 'empty';
   print "$file is ", join(', ',@description),"\n";
}
```

Here is the list of features, which you can check for a file or directory −

Operator	Definition
-A	Script start time minus file last access time, in days.
-B	Is it a binary file?
-C	Script start time minus file last inode change time, in days.
-M	Script start time minus file modification time, in days.
-O	Is the file owned by the real user ID?
-R	Is the file readable by the real user ID or real group?
-S	Is the file a socket?

-T	Is it a text file?
-W	Is the file writable by the real user ID or real group?
-X	Is the file executable by the real user ID or real group?
-b	Is it a block special file?
-c	Is it a character special file?
-d	Is the file a directory?
-e	Does the file exist?
-f	Is it a plain file?
-g	Does the file have the setgid bit set?
-k	Does the file have the sticky bit set?
-l	Is the file a symbolic link?
-o	Is the file owned by the effective user ID?
-p	Is the file a named pipe?
-r	Is the file readable by the effective user or group ID?
-s	Returns the size of the file, zero size = empty file.
-t	Is the filehandle opened by a TTY *terminal*?
-u	Does the file have the setuid bit set?
-w	Is the file writable by the effective user or group ID?
-x	Is the file executable by the effective user or group ID?
-z	Is the file size zero?