How to do it...

1. Redirecting or saving output text to a file can be done as follows:

   ```
   $ echo "This is a sample text 1" > temp.txt
   ```

   This would store the echoed text in `temp.txt` by truncating the file, the contents will be emptied before writing.

2. To append text to a file, consider the following example:

   ```
   $ echo "This is sample text 2" >> temp.txt
   ```

3. You can view the contents of the file as follows:

   ```
   $ cat temp.txt
   This is sample text 1
   This is sample text 2
   ```

4. Let us see what a standard error is and how you can redirect it. `stderr` messages are printed when commands output an error message. Consider the following example:

   ```
   $ ls +
   ls: cannot access +: No such file or directory
   ```

   Here + is an invalid argument and hence an error is returned.

   Successful and unsuccessful commands

   When a command returns after an error, it returns a nonzero exit status. The command returns zero when it terminates after successful completion. The return status can be read from special variable `$?` (run `echo $?` immediately after the command execution statement to print the exit status).

   The following command prints the `stderr` text to the screen rather than to a file (and because there is no `stdout` output, `out.txt` will be empty):

   ```
   $ ls + > out.txt
   ls: cannot access +: No such file or directory
   ```

   In the following command, we redirect `stderr` to `out.txt`:

   ```
   $ ls + 2> out.txt # works
   ```

   You can redirect `stderr` exclusively to a file and `stdout` to another file as follows:

   ```
   $ cmd 2>stderr.txt 1>stdout.txt
   ```

It is also possible to redirect `stderr` and `stdout` to a single file by converting `stderr` to `stdout` using this preferred method:

```
$ cmd 2>&1 output.txt
```

Or the alternate approach:

```
$ cmd &> output.txt
```

5. Sometimes, the output may contain unnecessary information (such as debug messages). If you don't want the output terminal burdened with the `stderr` details then you should redirect the `stderr` output to `/dev/null`, which removes it completely. For example, consider that we have three files `a1`, `a2`, and `a3`. However, `a1` does not have the read-write-execute permission for the user. When you need to print the contents of files starting with `a`, we use the `cat` command. Set up the test files as follows:

   ```
   $ echo a1 > a1
   $ cp a1 a2 ; cp a2 a3;
   $ chmod 000 a1  #Deny all permissions
   ```

   While displaying contents of the files using wildcards (`a*`), it will show an error message for file `a1` as it does not have the proper read permission:

   ```
   $ cat a*
   cat: a1: Permission denied
   a1
   a1
   ```

   Here, `cat: a1: Permission denied` belongs to the `stderr` data. We can redirect the `stderr` data into a file, whereas `stdout` remains printed in the terminal. Consider the following code:

   ```
   $ cat a* 2> err.txt #stderr is redirected to err.txt
   a1
   a1

   $ cat err.txt
   cat: a1: Permission denied
   ```

   Take a look at the following code:

   ```
   $ cmd 2>/dev/null
   ```

   When redirection is performed for `stderr` or `stdout`, the redirected text flows into a file. As the text has already been redirected and has gone into the file, no text remains to flow to the next command through pipe (`|`), and it appears to the next set of command sequences through `stdin`.