When working with files, you often need to write information to them. Scripts can make this task much easier. But before we learn how to write data to files, let's get to know what file descriptors are. In Unix systems, each input and output (IO) resource has descriptors. These descriptors show where the IO operations will happen. Now, let's take a closer look at what a descriptor is and how to use it.

**What is a descriptor**

A **descriptor**is a non-negative number assigned to a file or other IO resource. For the rest of this topic, we'll refer to all these resources simply as 'files.' Descriptors allocate in the order that files open, and each subsequent file gets the next available descriptor. Through these descriptors, we can access various IO streams like standard input, standard output, and standard error:

* **Standard Input (stdin) - Descriptor 0:** is a channel where a program receives data for processing. In simple terms, stdin is like a "mailbox" where the program collects the incoming data it needs to work on. For example, when you type commands into a terminal, you're sending data through stdin to be processed by the system.
* **Standard Output (stdout) - Descriptor 1**: This is where the program sends data after processing it.
* **Standard Error (stderr) - Descriptor 2**: This stream is used for error messages and diagnostics.

**File redirections**

Redirection is a popular feature in Unix/Linux systems. For example, you can write new information to a file using the terminal. Two symbols, >> and >, serve this purpose:

* >> appends new content to the file without removing the old information.
* # adding new information to a diary

echo "Also my favorite things are ..." >> diary.txt

* > overwrites the file, removing any old content.
* # rewriting the file content

echo "My new everlasting love is ..." > secret.txt

Great, now you know how to write information to files. But what if you need to redirect the output of the descriptors?

**Redirections of descriptors**

As mentioned earlier, using scripts can sometimes simplify writing information to files. Let's consider an example.

Let's say you need to write two different messages to two different files simultaneously. You can create a program.sh script for this task. The script contains two echo commands: one for a normal message and another for an error message.

echo "Just a normal message"

echo "ERROR!" >&2

In this example, >&2 directs the "ERROR!" message to Standard Error (stderr), represented by descriptor 2.

To execute this, you can run:

$ bash program.sh 1> /tmp/log.txt 2> /dev/null

Here, 1> /tmp/log.txt redirects the normal message to log.txt, while 2> /dev/null routes the error message to /dev/null, a special file that discards any data sent to it.

Besides redirecting standard output, you can also redirect error messages. If you want to record all errors in one place, run program.sh 2> errors.txt.

Both program.sh 1> file and program.sh > file redirect stdout, doing the same thing.

This covers the basics of how to redirect output using descriptors, but there's more to learn. There is one more way to redirect IO streams, and it is called a pipeline with the corresponding operator |.

**Pipe operator**

The purpose of the pipe (|) is to redirect the output of one command as input to another command. You can string together multiple commands like this: command1 | command2 | command3 .... In this arrangement, all data displayed by the first command becomes the input for the second command, as if you were typing it from the keyboard. The first command processes its input and passes its output to the second command, and so on. It's important to note that these commands run concurrently. That means the output from the first command moves immediately to the input of the second command without waiting for the first to finish.

For example, let's look at a simple pipeline that uses echo and grep. The echo command prints out "Hmm...\nBrr...\nMmm...", and the grep command looks for all occurrences of the letter "m" in these lines:

$ echo -e "Hmm...\nBrr...\nMmm..." | grep "m"

The output will be:

Hmm...

Mmm...

We will tell you more about the grep command later. Using the pipeline, you can implement more complex chains of commands, but the main thing is that now we know what it's generally for.

**Conclusion**

In conclusion:

* Descriptors allow us to access IO streams;
* The > and >> operators can be used for file redirection;
* > overwrites file content, while >> appends to it;
* Descriptors can be redirected using these same operators;
* A pipeline (|) lets us send the standard output (stdout) of one command to the standard input (stdin) of another.

## Change file content

 Report a typo

Write a program that will change the content of the /tmp/file.txt to I am the bash master! to make us know you've mastered the topics well!

**Sample Input 1:**

Hello, world!

**Sample Output 1:**

I am the bash master!

#!/usr/bin/env bash

solve() {

# add your solution here

echo "I am the bash master!" > /tmp/file.txt

}

## Defining descriptors

What is a descriptor?

a number that indicates a IO resource, e.g. a file

**Redirect the output**

Write a script that will redirect the stdout of the run command to the /tmp/logs.txt file.

Sample Input 1:

Ready

Sample Output 1:

Ready

#!/usr/bin/env bash

solve() {

run > /tmp/logs.txt # add your redirections here

}

**Busy descriptors**

Which files are used for the most basic actions and always have the same file descriptors from 0 to 2?

The correct answers are:

✅ **stdin**  
✅ **stdout**  
✅ **stderr**

These are the three basic files (IO streams) used by every process and always have the same file descriptors:

* **0** → stdin (standard input)
* **1** → stdout (standard output)
* **2** → stderr (standard error)

**Deciphering complex command redirection**

In a Unix-like command-line environment, consider the following command:

command 2>&1 >file1 | anothercommand >file2

What is the function of this command?

Redirects standard output (stdout) from 'command' to 'file1' and standard error (stderr) to stdout, then pipes the output to 'anothercommand' redirecting its standard output to 'file2'

### Why?

Let’s walk through it briefly again:

* 2>&1 → redirects **stderr to current stdout (the terminal)**
* >file1 → redirects **stdout to file1**, **after** stderr was redirected
* So:
  + **stderr** goes to the **pipe** (because it's still going to the terminal, which the pipe reads from)
  + **stdout** goes to file1
* Pipe | sends stderr into anothercommand
* >file2 sends **stdout of anothercommand** to file2

**Add new line in a file**

Veronica maintains a to-do list in the file my\_chores.txt. It is already written there that she needs to finish writing her report and walk her dog. But she remembers that she also needs to go to the store and buy food. She decides to add this task to the file. What command should Veronica use?

echo "Buy some food" >> my\_chores.txt

**Redirect errors**

 Report a typo

Write a script that will redirect the stderr of the run command to the /tmp/errors.txt file.

Sample Input 1:

[warn] someone tried to access your computer

Sample Output 1:

[warn] someone tried to access your computer

#!/usr/bin/env bash

solve() {

run 2>/tmp/errors.txt # add your redirections here

}

**Decoding Unix File Redirection**

 Report a typo

In UNIX command line operations, consider the code snippet below. What are the concepts illustrated with the use of the '>', '<', and '2>' symbols? What file descriptors do they represent respectively?

command1 > output.txt 2> error.txt < input.txt

< is for redirection of the standard input (file descriptor 0), > is for redirection of the standard output (file descriptor 1), 2> is for redirection of error messages (file descriptor 2)

**Add new content**

 Report a typo

The programming teacher, Mr. Smith, has a list of his students. He keeps this list in /tmp/students.txt file. Jerry has just joined Mr. Smith's class. Write a program that will add Jerry to the list of other students.

Sample Input 1:

Dina

Bob

Katy

Sample Output 1:

Dina

Bob

Katy

Jerry

#!/usr/bin/env bash

solve() {

# add your solution here

echo -e "\nJerry" >> /tmp/students.txt

}