# Unix, Standard I/O and command line arguments

For any programming assignments I give you, I expect a program that reads and writes to standard input and output, taking any extra parameters from the command line. This handout explains how to do that. I have also appended a small appendix of useful Unix commands.

I recommend that you go to a Unix terminal and type in and run all of the examples in the gray boxes. You can do this from the Terminal application on a MacIntosh, or from a terminal window in GNU/Linux or any Unix-like operating system, from Cygwin in Windows maybe, or by connecting to bingsuns using SSH.

# Overview

Most programming languages feature "standard," or default, input and output channels where I/O goes unless otherwise specified. In C, the functions `scanf()`, `gets()` and `getchar()` read from standard input, while `printf()`, `puts()` and `putchar()` write to standard output. In Tcl, the `gets` and `puts` commands read and write standard input and output. In awk and perl, you just sort of use the force and your program receives a line of input from somewhere. Let's stick with C for the time being.[1]

When a program runs from a terminal, standard input is usually the user's keyboard input, while standard output is displayed as text on-screen. Unix and Unix-like operating systems allow you to intercept the standard I/O channels of a program to redirect them into files or other programs. This gives you the ability to chain many simple programs to do something genuinely useful.

# Redirection

In a terminal shell, the redirection operators > and < use a file for standard input or output in place of the keyboard and screen. To see this, try typing this `tr` command, which flops the case of anything you type in:

```
bash-3.2$ tr A-Za-z a-ZA-Z

Hello World!
hELLO wORLD!
[Ctrl-D to quit]
```

[1] In both C and Tcl, these channels are actually named stdin and stdout. So instead of writing `printf("Hello")` you can equivalently write `fprintf(stdout,"Hello")`.

Now, create a text file called foo.txt in your home directory and redirect it into `tr`:

```
bash-3.2$ cat foo.txt
I am the mother of all things, and all things should wear a
sweater.
bash-3.2$ tr A-Za-z a-ZA-Z < foo.txt
i AM THE MOTHER OF ALL THINGS, AND ALL THINGS SHOULD WEAR A
SWEATER.
bash-3.2$ tr A-Za-z a-ZA-Z < foo.txt > output.txt
bash-3.2$ cat output.txt
i AM THE MOTHER OF ALL THINGS, AND ALL THINGS SHOULD WEAR A
SWEATER.
```

First, we dump the contents of foo.txt into standard input; then, we pour the standard output into a file that we can read later (using the `cat` command.)

# Piping

The vertical pipe operator | (shift-backslash on most keyboards) tells your terminal to take the standard output of one program and patch it into the standard input of a second program. Let's use `tr` to replace every space in a file with a newline character[2]:

```
bash-3.2$ tr "[:space:]" "\n" < foo.txt
I
am
the
mother
of [... etc etc]
bash-3.2$ tr "[:space:]" "\n" < foo.txt | sort
I
a
all
all [... etc etc]
bash-3.2$ tr "[:space:]" "\n" < foo.txt | sort | tail -1
wear
```

[2] In some Unix shell environments, you have to surround everything in quotes to prevent confusion. On MacOS X, no quotes are needed around the [:space:] argument, but they are necessary on bingsuns.

The final `tr | sort | tail` command outputs the one word in the file that appears last in the dictionary. Make special note of what we didn't do in this example: *we didn't write a computer program to do this*. Many data processing tasks can be achieved simply by piping an input file through a series of rudimentary Unix commands. For another example, try this command chain to get a frequency count of words in a file:

```
bash-3.2$ tr -s "[:space:]" "\n" < foo.txt | sort | uniq -c
```

# Command line arguments

Notice that some of our commands have arguments to them, like the `-c` in `uniq -c`. These provide input parameters to your program separate from the input and output streams. This will be particularly useful to us, because ciphers require two inputs: the stuff to encrypt (the plaintext), and a key. It will make sense to provide the key as a command line argument, and process the plaintext as standard input.

To handle command line arguments in C, declare your main function to have two arguments, `int argc` and `char ** argv`. The variable `argv` is an array of character strings holding the command line, with `argv[0]` holding the name of the command. The value `argc` tells you how many arguments are waiting for you (to be more precise, it tells you the array length of `argv`). Type in and run this program:

```
#include<stdio.h>

int main( int argc, char ** argv) {
     int i;
     for( i=0; i<argc; i++ )
          printf( "Argument %d is %s\n", i, argv[i] );
     return 0;
}
```

```
bash-3.2$ gcc -o foo foo.c
bash-3.2$ ./foo -r -x 333
Argument 0 is ./a.out
Argument 1 is -r
Argument 2 is -x
Argument 3 is 333
```

Note that argument 3 is not a number, but he string “333”. If you need to get a number into your program, use the `atoi()` function (“array to integer”) to convert the string.

In Tcl, there is also an array called `argv` that can be used in the same way.

```
#!/usr/bin/tclsh
foreach a $argv {puts "Argument $a"}
```

```
bash-3.2$  chmod u+x args.tcl
bash-3.2$  ./args.tcl one two
Argument one
Argument two
```

## Our two channels are stdin, stdout and stderr---three, three channels

There is also a third standard I/O channel called `stderr`, or standard error, to which you can print error messages. In C, use fprintf(stderr, “stuff”) to print to standard error.

Like standard output, `stderr` dumps to the screen, but it is a distinct channel. This is nice because you can write output to `stdout` and error messages to `stderr`, and if a user redirects your program’s output to a file, only the output will go to the file while the error messages will spill onto the screen. This prevents a program’s useful output from becoming contaminated with wacky status messages.

## Please do it this way

Unless it is absolutely necessary, write your programs to get input data from `stdin` and write output to `stdout`, and to get all configuration info, such as keys, from command line arguments. This maximizes your program’s utility, allowing us to feed it input from other commands, and to process its output using other commands.

Bear in mind, however, that it is a security risk to type in key data on the command line. The commands you type in the terminal are saved to a history file, and whoever reads that file can get your key. Also, if multiple people are logged in to a single computer, one can often see what commands other people are executing (try typing `ps -fa` on bingsuns, for example.)

# Command summary

Here is a short list of useful Unix commands, in no particular order:

| Command | Description |
|---|---|
| cat *file1 file2 file3...* | Con**cat**enates a bunch of files and outputs them. The command *cat file* will print out the file to `stdout`. |
| ls<br><br>cd *directory*<br><br>pwd | Lists the files in your directory. The command ls -l will also dump out file sizes, permissions, and creation dates.<br><br>Changes to a directory. Typing cd without an argument will change you to your home directory.<br><br>Print Working Directory: tells you where you are. |
| chmod *pattern file* | Changes permissions. This command is useful to make a computer program executable (chmod u+x program). The pattern consists of some combination of u, g and o (user, group and other), then a plus or minus to add or revoke permissions, and then r, w and x (read write and execute.) So `chmod ugo+r *` will let everyone in the world read all of your stuff---never do this.<br><br>Note that if a program is executable, you execute it by typing its name as a command---however, you may have to provide a path name, such as `./foo` instead of `foo` |
| sort *file* | Sorts a file line-by-line. You can also call sort without any filename and pipe in standard input instead. The sort command sorts lexicographically, but sort -n will sort the output numerically instead. |
| tr *pattern pattern* | The tr command translates characters in the first pattern to characters in the second pattern. You're better off looking at the tr manual page (type man tr) to see examples of this in use. |
| grep *pattern* | The grep command scans input for all lines that match a given pattern. The patterns are expressed in a hilariously complex language that deserve their own lecture or sequence of lectures. Again, check out the manual page (type man grep) to see some examples. |

| Command | Description |
|---|---|
| man command | Displays the manual page for this command.<br>Manual pages are mildly informative, but not useful as tutorials. If you want to know how to use a Unix command you are often better off Googling for examples (or in the case of some commands, like `awk`, buying the book.) |
| whoami | Tells you who you are. |
| cp *file1 file2*<br><br>mv *file1 file2*<br><br>rm *file* | Copies *file1* to *file2*.<br><br>Moves (renames) *file1* to *file2.*<br><br>Erases *file.* Be careful with this command. Unix makes it easy to erase everything in a directory (`rm *`) or to erase everything in all subdirectories (`rm -r *`) or to erase everything on the entire system that you have permission to erase (`rm -r /`). |
| mkdir *directory*<br>rmdir *directory* | Creates and deletes a directory (folder.) |
| pico *textfile* | Pico is a user-friendly text editor. It has an interface similar to most PC text editors, and it always shows you a list of commands and their hotkeys at the bottom of the screen: most useful are the keys to save (Ctrl-O) and exit (Ctrl-X). |
| vi *textfile* | Vi is a non-friendly text editor. It shows you nothing, and the commands are obscure and modal. To start typing, first enter an editing mode by hitting the a key or the i key or the A key or the I key or the S key or the s key. Do not use the arrow keys on your keyboard at this time, or something horrible will happen. To quit vi, type the imperative quit sequence `:q` (or `:q!` to override warnings.) To save and quit, type :wq, or type an uppercase `Z` twice (ZZ is an abbreviation for “save and quit.”) You can only type these commands haver hitting ESC to break out of editing mode.<br><br>Vi is the “visual” version of a program called ed. Ed is basically vi except you can’t see anything. |

| Command | Description |
|---|---|
| more *file* | Displays a long file one screen at a time, allowing you to page through it by typing the Space bar. Hitting <enter> will scroll through the file one line at a time. There is also a more sophisticated version of more, called less. |
| head -*Number file*<br>tail -*Number file* | Head displays the first *Number* lines from the file, while tail displays the last. If no number argument is provided, these commands default to 10. |
| tclsh | Enters the Tcl shell, from which you can run Tcl scripts. For more information about Tcl, consult the handout on Tcl programming |
| gcc file.c<br>gcc -o file file.c | Compiles a C program. If you do not provide an output file name with the -o output, the executable is usually named a.out.<br><br>Gcc automatically creates executable files, so chmod is not needed to prepare them for execution. |