

**Due Date:** see website

In this class, you are required to have familiarity with Unix commands and Unix programming environments. The first part of this exercise is related to making sure you are comfortable in our Unix environment; the remaining parts ask for familiarity with how to invoke programs in a Unix environment and how to make use of the standard input and output streams, as well as understanding the difference between byte and character streams.

## 1 Using Linux

It is crucial that everybody become productive using a Unix command line, even if the computer you are using daily is a Windows or OSX machine. Working on the command line requires working knowledge of a shell such as bash or zsh, but it also requires an understanding of how the shell interacts with system commands and programs from the user's perspective.

Please do the following, then answer the questions below.

- Make sure your personal machine has an ssh client installed. Set your machine up for public key authentication when logging on to `rlogin.cs.vt.edu`. Use `ssh-keygen` to create a key.

There is also a web interface provided by the department that allows you to create a key pair at <https://admin.cs.vt.edu/keys.pl>. However, in this case, you do not maintain continuous possession of the private key from its inception.

At the end of this step, you should be able to ssh into `rlogin` without having to type a password.

- Make sure you know how to use the command line editing facilities of your shell. For bash users, which most of you are by default, examine the effect of the following keys when editing: `^d`, `TAB`, `^a`, `^e`, `^r`, `^k`, and `^w`.

Examine the effect of the following keys when you invoke a program: `^c`, `^s`, `^q` (`x` stands for `Ctrl-x`.)

- Customize your shell and create a custom prompt and any aliases you may need. A custom prompt typically includes the name of the machine you're on and at least part of the pathname of the shell's current directory as when setting `PS1` to `[\u@\h \W]\$`
- Make sure you know how to use **at least one** command line editor, such as `vim`, `nano`, `pico`, or `emacs`.
- Many students set up a remote environment that allows them to use an IDE on their computer. Notably, Microsoft's Visual Studio Code provides an extension that presents a remote environment within the IDE that is well integrated. Although

not mandatory, we highly recommend that you do this as well. The TAs will post instructions on how to do that.

Answer the following questions:

1. What is the name of the file containing your private key and on which computer is it stored?
2. What is the name of the file containing your public key and on which computer is it stored? Provide the full path name.
3. What is the output of running `echo ~cs3214/bin` on an rlogin cluster machine?
4. Make sure that `~cs3214/bin` is part of your `PATH` variable when you log on. To test it, log off and log on again, then type `echo $PATH`. Copy and paste the output you get here.
5. How many machines are part of the rlogin cluster (Hint: visit <http://rlogin.cs.vt.edu/>) this semester? Include only those whose names are derived from trees, e.g. "birch."
6. Make sure your bash prompt includes your username, the name of the current machine, and a suffix of the current directory. Copy the value of your `$PS1` variable here.
7. Some filenames in your home directory start with a dot. How can you list those files or directories?
8. Why do many people define an alias for `rm` using `alias rm='rm -i'`?
9. The `diff` Unix command compares two files line by line. It is typically used to create "patch files" which capture a change made to one of more related files. When using `diff` to create a patch that describes such a change, in which order are the files to be compared usually specified?
10. Which Unix group or group(s) are you currently a member of on our cluster?
11. What Unix permissions does a file or directory need to have to make sure that no one besides yourself can access it?

## 2 Understanding Command Line Arguments and Standard I/O in Unix

In the past, we observed that some students coming into CS 3214 did not understand how programs access their command line arguments and how they make use of the standard input/output facilities, which present one of the basic abstractions provided by an operating system.

**Application Side Note.** Deep knowledge of Unix is an absolute prerequisite for anyone wanting to learn or work with containers. As an example, consider this excerpt [link] from a script used to set up the container in which this semester’s Discourse server runs:

```
run_image='cat $config_file | $docker_path run $user_args \  
    --rm -i -a stdin -a stdout $image ruby -e \  
"require 'yaml'; puts YAML.load(STDIN.readlines.join) ['run_image']"'`
```

This command sets a variable `run_image` to the standard output that results from running the pipeline that is enclosed in backquotes. This pipeline consists of 2 commands: the command `cat`, which is given 1 argument (taken from the value of `$config_file`) and whose standard output is “piped” into the command given by the `$docker_path` variable (probably `docker`), which is invoked with 12 arguments, the last one being a Ruby program that will be run inside the container, but which can access as its standard input (STDIN) the data written to `cat`’s standard output. Being able to understand what commands like this one do is a motivation for this exercise (and hopefully, the following exercise and project will provide an even deeper understanding).

To practice this knowledge, write a C program that concatenates a combination of given files and/or its standard input stream to its standard output stream. The exact specification is as follows.

Your program should be simply called `concatenate.c`.

When invoked without arguments, it should copy the content of its standard input stream to its standard output stream. “Standard input” and “standard output” are standard streams that are set up by a control program that starts your program (often, the control program is a shell).

When invoked with arguments, it should process the arguments in order. Each argument should be treated as the name of a file. These files should be opened and their content should be written to the standard output stream, in the order in which they are listed on the command line. If the name given is `-` (a single hyphen), the program should read and output the content of its standard input stream instead.

If any of the files whose names are given on the command line do not exist, the program’s behavior is undefined.

Your C program may make use of C’s stdio library (e.g., the family of functions including `fgetc`, etc.), or it may use system calls such as `read()` or `write()` directly. You should buffer the data, but you may not assume that it is possible to buffer the entire file content in memory.

**Implementation Requirement:** to make sure you understand the uniformity provided by the POSIX C API, we require that your program use the same function to copy the data contained in files and for the data it reads from its standard input stream. Your

program's `main()` function will then call this single function multiple times, as needed. In other words, do not make use of facilities such as `getchar()` that implicitly refer to the standard input stream.

You may use the script `test-concat.sh` to test your code.

### 3 Understanding how to access the Standard Input and Output Streams in your Preferred Language

Standard input and output are concepts that are not specific to the use of C. Choose a language of your choice that is not C (e.g. C++, Go, Ruby, Java, Python 2, Python 3, JavaScript, etc. etc.) and implement the above concatenate program in your language.<sup>1</sup>

If your language cannot be compiled into an executable, and also cannot be executed directly by an interpreter using the Shebang/Hash-bang convention, you need to create a wrapper script for testing it. This wrapper script may be required for Java, it should invoke your program, passing any command line arguments to it.

You may use `test-concat.sh` to test, by passing the name of your script or executable as an argument.

Hint: most higher-level language allow compact implementations of these tasks. For instance, my Python 2 implementation is 13 lines long.

**Implementation Requirement:** the implementation requirement is the same. Do not special case standard input/output, use a single function.

**Efficiency.** You should use buffered forms of input and output in order to reduce the number of system calls your program makes. For instance, in C, the `stdio` library provides such buffering by default if you use `fgetc()` or `fread()`, whereas if you use the lower-level `read()` call directly you will need to make sure that you do buffering yourselves (in other words, read multiple bytes at once rather than a single byte in each call). The autograder may run your program under a suitable timeout.

**Use of Byte Streams.** For both parts 3 and 2, your program must not attempt to interpret the content of the streams it reads and writes in any way. In other words, it should output the bytes (octets) that appear in the input as they appear, without making assumptions or processing them in any way. This includes the possible occurrence of the byte value `0x00`, which may occur any number of times in the input and must be copied into the output.

---

<sup>1</sup>If you choose C++ or Rust, you must use C++'s or Rust's standard library, not C's.

Similarly, the byte value 0x0A (aka LF, or LINEFEED character) may occur any number of times. Your program should not assign special significance to either of them, so do not assume (a) that data read can be represented as zero-terminated C-style strings, and (b) do not assume that the input can be broken into lines efficiently. (The worst case input would be a sequence consisting of only LF characters.)

Many programs process characters, which has contributed to the fact that the I/O libraries of some higher-level languages default to the assumption that programmers will want to input and/or output character streams in some valid encoding when accessing file streams. The most commonly used character set today is the Unicode character set, and the encoding that is most commonly used is UTF-8. For instance, the unicode character U+263A is encoded as a 3-byte sequence 0xE2 0x98 0xBA in UTF-8. While any sequence of Unicode characters can be encoded into a sequence of bytes, the opposite is not true: not every sequence of bytes represents a valid encoding of some characters.<sup>2</sup>

For the first 2 implementations, do not assume that the input represents characters in any valid encoding. Specifically, the input data may not represent a valid UTF-8 encoding, and therefore, attempts to interpret it as UTF-8 data and decode it will fail for some tests, resulting in exceptions and/or data corruption.

## 4 Understanding Character-Based I/O

In this part of the exercise, you will implement a simple utility that interprets its standard input stream as a stream of UTF-8 encoded Unicode characters which it then counts. If the standard input contains a valid encoding of Unicode characters, the utility should output the number of such characters; else it should report an error and abort<sup>3</sup>. This is similar to the Unix tool 'wc -m' which counts the number of Unicode characters in the input stream, except that 'wc -m' ignores if the input stream is not in a valid encoding.

Write a program `unicodetcount.c` using only functions that are part of the C standard library. You may use the `fgetwc` (easiest) or the `mbrtowc` functions, or identify the length of each encoded Unicode character manually. Your program must use buffering as well. Remember to use the `setlocale(3)` function to set the character type locale (`LC_CTYPE`) to `"en_US.utf8"`.

If the standard input stream does not consist of correctly encoded Unicode characters in the UTF-8 transfer encoding, output:

```
Invalid or incomplete multibyte or wide character
```

<sup>2</sup>For those wanting to learn more about the rationale behind UTF-8, I recommend The history of UTF-8 as told by Rob Pike which describes how Ken Thompson invented UTF-8 in one evening and how they together built the first system-wide implementation in less than a week.

<sup>3</sup>aborting is accomplished via the `abort()` function, which sends the `SIGABRT` signal to the process, which then typically leads to its termination

else output the number of Unicode code points (each representing a Unicode character) found in the input stream.

Finally, write the same program in a high-level language of your choice.

Unlike for the concatenate program, you only need to process the program's standard input stream and you do not need to handle the case where names of files are passed as command line arguments.

You may use the script `test-unicodetest.sh` to test your code. Independent of the size of the input stream, your program must not use more than 50MB of virtual memory.

**What to submit:**

Submit a tar file with your answers, containing the files:

- `answers.txt`, a UTF-8 encoded text file with your answers,
- a C file `concatenate.c` containing your implementation for part 2,
- a C file `unicodetest.c` containing your C implementation for part 4,
- a file `concatenate.?` with a suitable suffix containing your implementation for part 3 in another language,
- a file `unicodetest.?` with a suitable suffix containing your implementation for part 4 in another language.

Do not submit compiled executables. All 4 required programs are short programs.