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 a review to make sure you are comfortable in our Unix environment. The second part relates to the use of basic command line and standard I/O facilities from an application's developer perspective. The third part focuses on the difference between byte and character streams, which is an important and often confusing topic of high practical relevance.

1 Using Linux

It is crucial that everybody become productive using a Unix command line, even if the computer you are using daily is not a Unix machine. Working on the command line requires working knowledge of a shell such as bash, zsh, or fish, but it also requires an understanding of the most common system commands and how the shell interacts with these commands and with user programs.

- **Remote Terminal Access.** Make sure your personal machine has an ssh client program installed. Set your machine up for public key authentication when logging on to rlogin.cs.vt.edu. The exact way to do this will depend on your personal computing environment; on Unix, you would use ssh-keygen to create a key, e.g., `ssh-keygen -t ed25519`.

  There is also an web interface provided by the department that allows you to create a key pair at `https://admin.cs.vt.edu/my-ssh-keys/`. In this case, Techstaff will create a private key for you, and thus you will not be able to maintain continuous possession of the private key from its inception. Remember that your private key represents you, and anyone who can access it can impersonate you as far as accounts go where you have placed the matching public key. This means you should use this key pair only for your SLO/CS account and nowhere else.

  At the end of this step, you should be able to ssh into rlogin without having to type a password from your personal computing device.

- **Command-line Editing.** 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: ˆd, TAB, ˆa, ˆe, ˆr, ˆk, and ˆw. Then memorize these keystrokes, making them part of your finger memory.

  Examine the effect of the following keys when you invoke a program: ˆc, ˆs, ˆq (ˆx stands for Ctrl-x.)

- **Shell Customization.** 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]\$`

- **Terminal Editors.** Make sure you know how to use **at least one** command line editor, such as vim, nano, pico, or emacs. We recommend vim, an editor that according to instructors at MIT "can match the speed at which you think."
- **Visual Studio Code.** 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 provides 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 share instructions on how to do that.

We will skip a quiz on this topic this semester, but we do ask that everyone who is not comfortable with their setup that allows them to use rlogin remotely work with teaching staff.

For this semester, we ask that you add `~cs3214/bin` to your `PATH` variable via your `~/.bash_profile` file. To check that you've done this correctly, type:

```
$ am I set up for CS3214?
```

It should say that you are. Note that properly testing any changes made to your shell initialization files require that you completely log out and log back in. If using vscode, this may require killing the vscode remote server via a command in your UI. For more background, read the section on bash setup in the FAQ.

Submit a screenshot that shows your customized prompt and the output of this command. To be considered set up correctly, your shell must both find the `am` command and your environment must pass the checks this script does, and both must be shown in the screenshot. Make also sure your customized prompt is visible in the screenshot and meets our requirements.

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. For instance, some students came with the mistaken impression that "standard input" and "standard output" always represent input or output from/to some kind of "console."

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] of 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 contain the data produced by the standard output stream 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 called `concatenate.c`.

When compiled and 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, with one exception noted below. 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. The argument `-` (a single hyphen) constitutes an exception and is to be treated differently. If it is given then the program should read and output the content of its standard input stream instead in this place. You may assume that at most one `-` is provided as part of your program's arguments.

If any of the files whose names are given on the command line do not exist, the program's behavior is undefined. Being "undefined" is a fancy way of saying that your program does not need to implement any checks or remedies for user errors such as specifying non-existing files for the purposes of this small exercise.

Just because programs can interact with their standard input stream in a way that abstracts away the concrete nature of the input stream does not mean that it is impossible to figure out what kind of stream it is. In addition to copying the contents of its standard input stream and/or provided files, your program should also output the type of each stream that it is processing, in order. This information should be sent to the standard error stream. When processing a file listed on the command line, it should output the provided name, followed by one of the following

- `is a regular file` if the stream refers to a regular file
- `is a pipe` if the stream refers to a pipe
- `is a character device` if the stream refers to a character device
- `is something else` otherwise.

Examples are shown below:

```
$ ./concatenate
standard input is a character device
abc   <- I typed this
abc   <- your program would output this
^D  <- I typed this, it won't appear on the terminal
$ ./concatenate concatenate.c | wc
concatenate.c is a regular file
     37     118     957
$ ./concatenate < concatenate.c | wc
standard input is a regular file
     37     118     957
$ cat concatenate.c  | ./concatenate | wc
standard input is a pipe
     37     118     957
```

As you can see, my C implementation is only 37 lines.

Your C program may make use of C's stdio library (e.g., the family of functions including `fgetc`, etc.)[1], or it may use system calls such as `read()` or `write()` directly. You should buffer data to avoid frequent system calls, but you may not assume that it is possible to buffer the entire file content in memory all at once. See the paragraph on "Efficiency" below.

Implementation Requirement: to make sure you understand the uniformity provided by the POSIX C API, we require that your program define a function, and then use this function to copy the data contained in files as well as 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 special case standard input/output by providing a separate code path for standard input/output that makes use of facilities such as `getchar()` that implicitly refer to the standard input stream. Your code should be DRY. This same function should make use of the `fstat(2)` system call (man page) to figure out what kind of file it is. Hint: use the `st_mode` field.

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

[1]cppreference.com has a full list at https://en.cppreference.com/w/c/header. Note that some particularly unsafe string functions are banned in CS3214.

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

Standard input and output are not concepts that are specific to the use of C. Choose a language of your choice that is not C (e.g. C++, Go, Ruby, Rust, Java, Python 3, JavaScript, etc.) and implement the above concatenate program in this language.[2] You may use all functions that are part of the language's standard library, but not functionality that requires the installation of extra libraries.[3]

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 will need to create a wrapper script so you can test it.[4] This wrapper script is required for Java, it should invoke your program, passing any command line arguments it receives to it.

As described in the Bash Hacker's Wiki you can use the `"$@"` shorthand to refer to the script's arguments, which are passed onto the Java program:

```
#!/bin/sh
# save this file as wrap-java.sh

java -Xmx120m Concatenate "$@"
```

Side Note. Some of you may never have invoked a Java program on the command line. It is done by compiling the Java code using `javac Concatenate.java` followed by `java Concatenate ...` to start the compiled program, where ... stands for the arguments being passed to it. Recent JDK versions permit the combination of these steps by supporting the invocation `java Concatenate.java ...`, essentially accommodating what used to be the frequent beginner's mistake of asking the JVM to run not-yet-compiled source code.

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

Java is an exception here: although the JVM is an ordinary Unix process, it makes certain assumptions about how much memory is available to it, which means it will not run well when this memory is limited from the outside. For Java implementations, you should run the test with:

```
SKIP_MEMORY_LIMIT=yes ./test-concat.sh ./wrap-java.sh
```

[2]If you choose languages that embed C, such as C++ or Rust, you must use those parts of C++'s or Rust's standard library that do not overlap with C's. So you can't use `::fread` in C++ or the `libc` crate in Rust.

[3]Depending on the language, what constitutes part of a language and what is "external" can be somewhat fuzzy: for the purposes of this exercise, the deciding criterion will be the ability to access this functionality without requiring additional installation steps. For example, in Java, you may use all of `java.` but not Apache Commons or Guava. In Javascript, you may use functionality that is provided by node.js, but not functionality that requires the installation of npm packages. Similar calls can be made for other languages.

[4]Don't submit the wrapper script though, our grader will identify the language and create its own script when necessary.

and **make sure** that the memory your program uses is instead limited in `wrap-java.sh` via the `-Xmx` flag.

What does `SKIP_MEMORY_LIMIT=yes` do? Whenever you prefix a command you type in bash with `ENVVAR=value`, bash will set an environment variable `ENVVAR` and give it the specific value before starting the program that follows. This variable is temporary in that it is in effect only during the execution of the command the user is running. This is a common way to influence the execution of programs without requiring other options such as configuration files or command line parameters. The programs so controlled will use the `getenv()` function to retrieve the value of these variables. If the program being run is a shell script, as in the case of `test-concat.sh`, they can access it directly.
You are encouraged to read `test-concat.sh` as it provides more examples of how to run programs on the command line. It also shows the different ways in which a shell can control a program's standard input streams.

Hint: most higher-level languages allow compact implementations of these tasks. For instance, a Python 3 implementation is 22 lines long.

Implementation Requirement: the implementation requirement is the same. Do not special case standard input/output, use a single function. Unlike for C, this single function can be one that you write, but some languages already provide a suitable function in their standard library that you may be able to look up.

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 yourself (in other words, read multiple bytes at once rather than a single byte in each call). The autograder will run your program under a suitable timeout that is designed to eliminate submissions that lack buffering.

Use of Byte Streams. For both parts 2 and 3, 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.

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 will be a sequence that doesn't contain any LF characters at all.)

Avoid Character-based Input Routines. Many real-world programs process input that is thought to represent 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. Note that character streams are abstractions built on top of byte streams - at the process/OS boundary all I/O is byte-based (this is true for at least the vast majority of contemporary environments).

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, nearly all web content uses this character set and encoding. In the UTF-8 encoding, the unicode character U+263A is encoded as a 3-byte sequence `0xE2 0x98 0xBA`. 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. [5]

For the two implementations of concatenate you're being asked to implement, **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. This means that you must be careful to avoid the default implementation in those languages that default to imposing a character stream abstraction, which include Python 3 and Java. Instead, you will need to examine their API and find the corresponding constructs that give you access to byte-based streams, which are sometimes referred to as "binary" forms of input or output.

4 A UTF-8 to UTF-32 converter

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, decodes them, and then encodes these characters as UTF-32 and outputs the encoded characters to its standard output.

If the standard input contains a valid encoding of Unicode characters, the utility should decode and encode them in UTF-32; else it should report an error and abort[6]. Your program should **not** prepend a byte-order mark (BOM) and it should output UTF-32 using little endian byteorder. For the purposes of this exercise, you may assume that your program is run on a machine that uses little endian byte order. This function is similar to the Unix tool `iconv -f utf8 -t utf32le` that performs the same conversion.

Write a program `utf8_to_utf32.c` using only functions that are part of the C standard library. You may use the `fgetwc` (easiest) or the `mbrtowc` functions, or write

[5]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.

[6]aborting is accomplished via the `abort()` function, which sends the `SIGABRT` signal to the process, which then typically leads to its termination

your own decoder that identifies the length of each encoded Unicode character manually. Your program must use buffering for its system calls as well. Remember to use the `setlocale(3)` function to set the character type locale (`LC_CTYPE`) to `"en_US.utf8"`.

In addition, since the size of the output can be up to four times the size of the input, you may need to implement an additional layer of buffering to reduce the number of times you call your output function. For instance, in C, you can avoid frequent calls to `fwrite()` by first buffering larger chunks. This will reduce the function call overhead in addition to the buffering already done by the standard library that avoids frequent system calls.

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

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

Your program may have already output converted characters to its standard output when it identifies this situation. Our tests will ignore these.

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

Unlike for the concatenate program, you need to process only 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-utf8-to-32.sh` to test your code. Independent of the size of the input stream, your program must not use more than 120MB of virtual memory – this is how we will enforce that your program does not attempt to buffer the entire content of its standard input stream in memory.

For Java users: For Java, we will again use `SKIP_MEMORY_LIMIT=yes` and instead limit heap memory with an `-Xmx120m` switch. Furthermore, note that the Java Unicode API assumes an internal representation of Unicode strings as 16-bit values. Thus, certain Unicode codepoints require 2 Java `chars` to represent them. Java calls these "surrogate pairs." Make sure to pay attention to the case where a surrogate pair occurs at the boundary of a buffer. For instance, if a surrogate pair spans offset 127 and 128 (counting from 0), and you've read 128 characters into your buffer, you will not be able to process this surrogate pair as you have read only the high surrogate character of this pair so far. You also can't avoid this situation by reading all of standard input into a single buffer upfront as that would violate the memory limit requirement. A possible strategy is to check for the case where the last (16-bit) character read is a high surrogate and if so, store it and logically prepend it to the buffer you use when processing the next chunk of input. Alternatively, use a more recently designed language - Python 3, for instance, does not expose its internal character representation in its Unicode support API and thus does not suffer from this issue.

What to submit:

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

- a png file `readyprompt.png` with the screenshot that shows you're ready for CS3214.
- a C file `concatenate.c` containing your implementation for part 2,
- a C file `utf8_to_utf32.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 `utf8_to_utf32.?` 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.

Hint: when preparing your submission, avoid the following mistake. To produce a tar file to submit, run

```
tar cvf ex0submission.tar concatenate.c utf32_to_utf8.c ...
```

where in place of the dots you put the names of the files containing your high-level language implementations. This will create a file `ex0submission.tar`[7] as an archive containing `concatenate.c`, `utf32_to_utf8.c`, and so on.

Don't do

```
tar cvf concatenate.c utf32_to_utf8.c ...
```

Because that would create an archive `concatenate.c` containing `utf32_to_utf8.c` and so on... in the process, and would, **without warning**, clobber the existing `concatenate.c` file you've just spent time creating.

[7]the name you choose doesn't actually matter to our submission system