COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 11 - Pipes, Filters and Regular Expressions

NCC Moore

McMaster University

Summer 2021

Stream Redirection

Glob Patterns

Searching For Files

Regular Expressions!

Advanced Searching

Various Applications

Documentation

Errata

Pipes of Different Types

In Unix, we write programs to handle text streams because of the *universality* of the interface.

- We think about `stdin`, `stdout` and `stderr` as being *streams of data*.
- How does one redirect a stream? Using a **pipe** of course!

Syntax	Description
`x | y`	x's `stdout` becomes y's `stdin`
`x > y`	x's `stdout` is written to file y
`x < y`	file y is redirected to x's `stdin`
`x 2> y`	x's `stderr` is written to file y
`x &> y`	x's `stdout` and `stderr` are written to file y

Check out These Pipes!

We've used redirection to and from files a number of times in lab already, so let's take a look at |.

- Redirect long output so it can be scrolled through:

```
$ make all | less
```

- Retrieve the third line of a file

```
$ cat file | head -3 | tail -1
```

- Sorted list of all unique file extensions in the current directory

```
$ ls | rev | cut -d'.' -f1 | rev | sort | uniq -c
```

Piping and Loops

You can even combine piping with loops in order to loop over the output of different commands, kind of like a Python for loop!

```
ls | while read item
do
  echo "$item exists in this directory!"
done
```

Redirecting to Arguments with xargs

When we pipe stdout to a command, the entire output is directed to stdin, regardless of separators (spaces, newlines, etc.).

What if the command we want expects its input by argument, rather than by stdin?

```
$ ls | rm
  # tries to delete the entire output of ls
```

The xargs command will repeat other commands, feeding them input gathered from stdin.

```
$ ls | xargs rm
```

- In the above case, the output of ls, which is separated by whitespace, is broken up and fed to rm individually.
- This command therefore succeeds!

Putting Arguments In! Their! Place!

Let's say we want to copy all the files in a directory with the ".txt" extension into a directory named `tmp`.

```
$ find . -name *.txt | xargs cp /tmp
cp: -r not specified; omitting directory '/tmp'
```

- By default, xargs pipes in arguments at the *end* of the list of arguments of the command its encapsulating.
- In this case, cp is copying *from* the place we want it to copy *to*!

```
$ find . -name *.txt | xargs -I x cp x /tmp
```

- The `-I` flag lets us select where (and how many times) the argument will be inserted into the target command.

It's `globberin'` Time!

Glob patterns give us a way to represent filepaths that match a pattern. We use special characters to represent multiple characters in various ways.

- `?` → Single character wildcard. Character is required to exist.

```
$ rm Lab??/org.txt
```

- `*` → zero or more continuous `?` wildcards. Effectively, replaces any number of characters, including no characters at all!

```
$ rm *.c
  # delete all .c files
```

- {} → choose between multiple specific strings (comma separated).

```
$ rm *.{c,o,h}
  # delete all .c, .o and .h files
```

Great Green `globs` of Greasy Grimey Gopher Guts!

Brace expansion also supports sequences using .. syntax.

```
$ echo {a..e}
a b c d e
$ echo {w..C}
W X Y Z [   ] ^ _ ‘ a b c
$ echo {10..−10}
10 9 8 7 6 5 4 3 2 1 0 −1 −2 −3 −4 −5 −6 −7 −8 −9 −10
```

It's generally a terrible idea to use glob characters literally in file and directory names, but if you *really have to...*

- \→ Escape a special character.

```
$ touch \*.c
$ ls
'*.rm'
```

Cops and globbers

Glob patterns will expand to to a list of delimiter separated path names.

```
$ cp *.txt /tmp
  # Copies all files with a .txt extension to /tmp
$ cp file.txt ./*
  # Doesn't copy file.txt into all directories in the
    current director.
```

the second command above expands to:

```
$ cp file.tx ./dir1 ./dir2 ./dir3 ... ./dirX
```

This copies everything into `./dirX`!

What a `find`!

The `find` command allows us to locate files in our file system using glob patterns.

```
$ find <starting directory> [-flags] -name <pattern>
```

- Unlike `cp` and `rm`, `find` automatically recurses through directories.

```
$ find /bin -name ls
/bin/ls
```

- To limit how deep `find` goes to find matching files, use the `-maxdepth` flag.

```
$ find ~/ -maxdepth 5 -name *.c
# finds all .c files in the first five directory layers
    after $HOME
```

finders Keepers!

- The -f flag tells find to target only files.
- the -d flag tells find to target only directories.
- You can even use flags to invoke boolean operations, and perform multiple tests at once!

```
$ find . -name *.c -or -name *.h
# finds all .c or .h files, starting in the current
    directory
$ find . -f -not -name *.py
# finds all files which are not python source files
   , starting in the current directory
$ find -d -name Lab** -name *.tex
# finds all directories, starting in the current
   directory, matching both glob patterns.
```

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Regular Expressions

Glob patterns are wonderful for managing the file system, but lack the expressive power to be used on larger targets, such as files themselves.

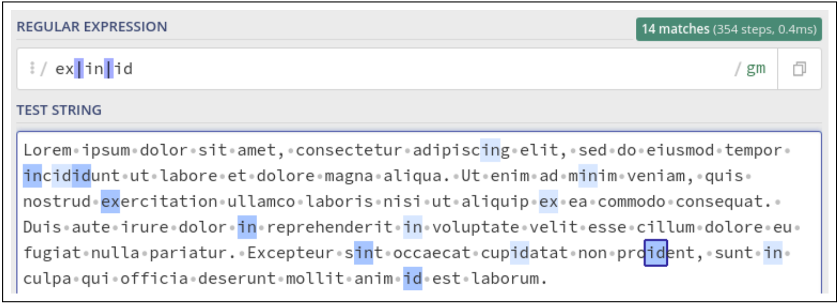
- Enter the **Regular Expression** (or **regex**)!
- Based on a model of computaton called **Finite State Automata**, which is beyond the scope of this course.
- Similarly to glob patterns, regular expressions allow us to write character patterns, which may then be used to test or search large groups of characters (i.e., files).
- An excellent online tool for testing and debugging large and small regex is `https://regex101.com`

Regex Syntax 1: Alternation

The vertical bar separates alternatives:

`a|b`

$$\{a, b\}$$



REGULAR EXPRESSION

14 matches (354 steps, 0.4ms)

```
/ ex|in|id / gm
```

TEST STRING

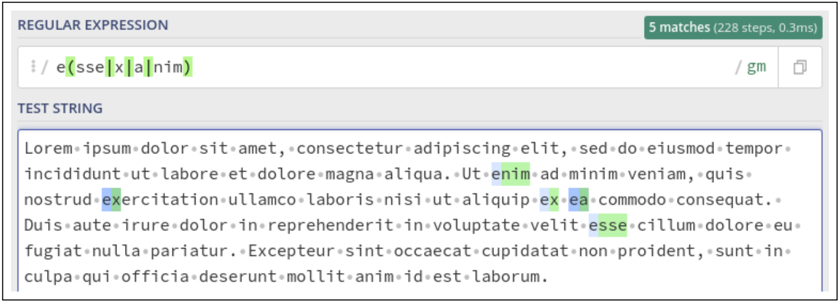
```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor 
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis 
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. 
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu 
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in 
culpa qui officia deserunt mollit anim id est laborum.
```

Regex Syntax 2: Grouping

Round braces determine how a regex operator is bound:

`Tom(ay|ah)to`

$$\{\mathit{Tomayto}, \mathit{Tomahto}\}$$



REGULAR EXPRESSION

5 matches (228 steps, 0.3ms)

```
/ e(sse|x|a|nim) / gm
```

TEST STRING

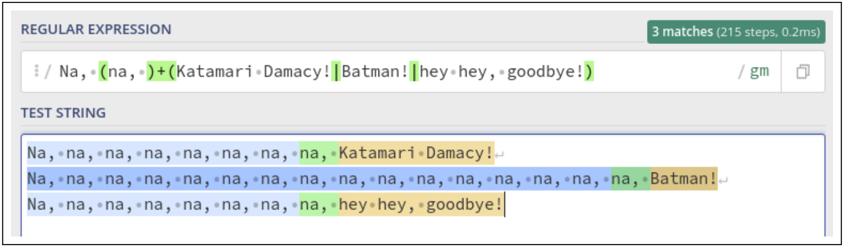
```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.
```

Regex Syntax 3: Quantification 1

A postfix plus specifies *one or more* occurances of the character(s).

`ab+c`

$$\{abc, abbc, abbbc, \ldots\}$$



REGULAR EXPRESSION

3 matches (215 steps, 0.2ms)

```
/ Na, (na, )+(Katamari Damacy!|Batman!|hey hey, goodbye!) / gm
```

TEST STRING

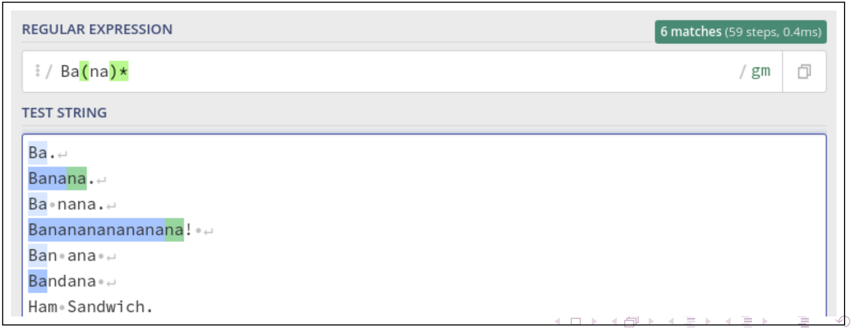
```
Na, na, na, na, na, na, na, na, Katamari Damacy!
Na, na, na, na, na, na, na, na, na, na, na, na, na, na, na, na, Batman!
Na, na, na, na, na, na, na, na, hey hey, goodbye!
```

Regex Syntax 4 : Quantification 2

A postfix asterisk $*$ specifies *zero or more* occurances.

`xy*z`

$$\{xz, xyz, xyyz, xyyyz, \ldots\}$$



REGULAR EXPRESSION

6 matches (59 steps, 0.4ms)

```
/ Ba(na)* / gm
```

TEST STRING

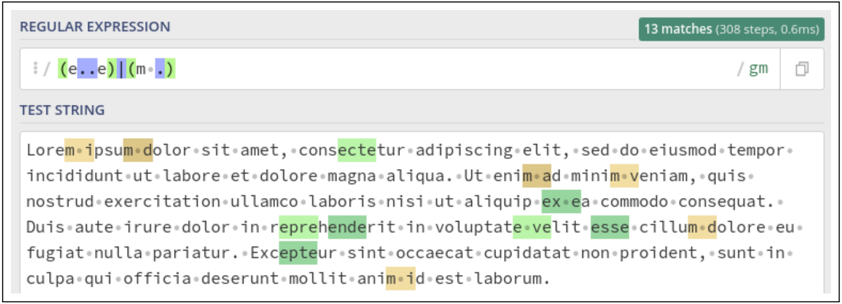
```
Ba.
Banana.
Ba nana.
Bananananananana! 
Ban ana 
Bandana 
Ham Sandwich.
```

Regex Syntax 5: Wildcards

The wildcard character `.` matches any character.

`a.b`

aac, *abc*, *acc*, *adc*, *aec*, ...



REGULAR EXPRESSION

13 matches (308 steps, 0.6ms)

```
/ (e..e)|(m .) / gm
```

TEST STRING

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.
```

The greps of Wrath

While `find` searches the *names* of files, `grep` searches the *contents* of files.

```
$ grep <options> <pattern> <file(s)>
```

- As with many commands, we can specify multiple files to be searched using glob patterns, and we can search directories recursively using the `-r` flag.
- If no file is specified, `grep` searches your working directory.
- The `-E` flag allows us to use **extended regular expressions**, which has some additional operators.

Grep Extended Regex Syntax 1

- | → works as expected.

```
$ grep -E 'It was the (best|worst) of times.' <file>
# Matches either 'It was the best of times' or 'It was
    the worst of times'
```

- [] → You can also use square braces to alternate many characters:

```
$ grep -E '[abcdefghijklmnopqrstuvwxyz]' <file>
# Matches any lowercase letter
```

- Notice how our regex is delimited by single quote characters!
- . is still the single character wildcard.

```
$ grep -E 'Super .ario' <file>
# Matches 'Super Aario', 'Super Bario', 'Super Cario',
    etc.
```

Grep Extended Regex Syntax 2

- ? → postfix operator indicating an item is optional.

```
$ grep -E 'a?b?c' <file>
# Matches 'c', 'ac', 'bc', and 'abc'
```

- ∗ → postfix operator indicating zero or more of an item

```
$ grep -E 'too*' <file>
# Matches 'to', 'too', 'tooo', etc.
```

- + → postfix operator indicating one or more of an item.

```
$ grep -E 'Ba(na)+' <file>
# Matches 'Bana', 'Banana', 'Bananana', etc.
```

- As shown in the above example, round braces are still used for grouping.

Grep Extended Regex Syntax 3

Although we have [and] to alternate larger groups of single characters, some common ones have been collected for us.

Regex Code	Description
`[[:alnum:]]`	Alphanumerics
`[[:alpha:]]`	Alphabetics
`[[:blank:]]`	Spaces and tabs
`[[:space:]]`	All whitespace
`[[:digit:]]`	Numerics
`[[:lower:]]`	Lower-case alphabetics
`[[:upper:]]`	Upper-case alphabetics

```
$ grep -E '[[:lower:]]([[:upper:]][[:lower:]]+)*[[:
    blank:]]' <file>
# matchesAnyThingWrittenInCamelCase
```

Grep Extended Regex Syntax 4

- Piping grep commands together find the *union*.

```
$ grep 'pattern1' file | grep 'pattern2'
# Only matches lines containing both patterns
```

- [^] inverts the selection.

```
$ grep —E '[^(ordinary)]' <file>
# Matches anything but 'ordinary'
```

- ^ at the beginning of a pattern requires the pattern to start at the beginning of the line.
- $ at the end of a pattern requires the pattern to end at the end of the line.

```
grep —E '^So anyways...$' <file>
# Matches 'So anyways...', but only if that's the
   entire line in the file.
```

That’s What She sed!

Any good programmer knows that find-and-replace is the most valuable tool any text editor can have. sed (Stream EDitor) lets us perform find-and-replace operations with all the power of Bash and regular expressions!

```
$ sed -i <flags> <pattern> <input>
# <pattern> := 's/<regex>/<string>/g'
```

- The regular expression tells sed where to perform the substitutions.
- The string is the replacement text.

For example, the following operation:

```
$ sed -i 's/(oak|spruce|larch|ash|maple)/tree/g' file.
    txt
```

Replaces any of the above specified types of trees with the string “tree”

Under sedation

Of course, we can combine this with the power of `find` to be able to perform crazy operations like:

- Perform find and replace operations over every file in the filesystem (that we have permissions for)
- Perform find and replace over all files in a directory and subdirectories of a particular file type.
- Perform find and replace on a file we don't know the exact location of.

```
$ find ~/ --name *.c | sed -i 's/<stdio.h>/"stdio.h
    "/g'
# replaces the braces on stdio.h with quotes in all
    .c files in the user's directory.
```

Problems in Space

In practice, searching commands can take a long time to execute, since they are often sifting through gigabytes of data (i.e., large portions of your filesystem)!

- If we have to perform a `grep` search with a large search area, but we know something about the files we need to search (like their all being .c files), we can pipe the result of `find` into `grep` to *substantially* increase the speed of the search.

```
$ find . -name *.tex | xargs grep -rai 'actually'
```

- One problem we'll run into however, is that `xargs` considers *both* newlines and space characters to be argument separators. This can be a real problem if our directory names contain spaces!

SPAAAAAAAAAAACE!!!!

Fortunately, a number of commands (including find) allow us to set a special delimiter, which `xargs` can be configured to look for.

- Apply `-print0` to `find`
- Apply `-0` to `xargs`
- Profit!

```
$ find . -name *.tex -print0 | xargs -0 grep -rai '
    actually'
./2MP3 Slides/Topic 11/Topic 11 - Other Topics in C++.
    tex:\item The four triangles that compose a
    tetrahedron require some constraints in order that
    they might actually form a tetrahedron.
...
```

Documentation!

”The greatest obstacle to discovery is not ignorance – it is the illusion of knowledge.” – Daniel Boorstin.

Documentation!

Some languages (like Haskell) are somewhat self-documenting. C is not one of those languages.

- Code can be documented either:
  - *In the source code* → useful for programmers.
  - *In an external document* → useful for all humans.
- A (recent?) trend in code documentation is **literate programming**.
  - That is, the source code is annotated in such a way that some documentation can be generated from it automatically.

Documentation Do's and Don'ts

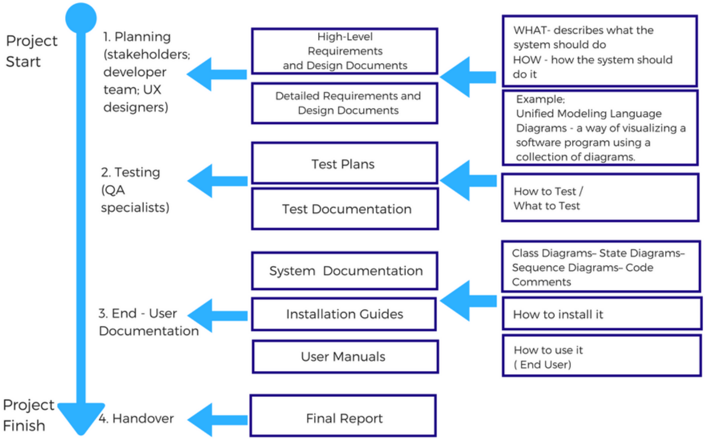
Do

- Write for your audience.
  - i.e., other developers.
- Use clear variable and function names (self-documentation)
- Comment:
  - The top of files
  - Functions
  - structs, typedefs
  - Control structures
- Explain *how* and *why*

Don't

- Explain what each line of code does.
- Explain how the language works.
- Leave sarcastic comments.
- Be emotional in any way.
- Comment each line.
- Write anything you wouldn't want anyone else to see (including your boss).

Types of Documentation



Project Start

1. Planning (stakeholders; developer team; UX designers)

High-Level Requirements and Design Documents

Detailed Requirements and Design Documents

WHAT- describes what the system should do
HOW - how the system should do it

Example:
Unified Modeling Language Diagrams - a way of visualizing a software program using a collection of diagrams.

2. Testing (QA specialists)

Test Plans

Test Documentation

How to Test / What to Test

System Documentation

Class Diagrams- State Diagrams- Sequence Diagrams- Code Comments

3. End - User Documentation

Installation Guides

How to install it

User Manuals

How to use it (End User)

Project Finish

4. Handover

Final Report

Image forcibly collectivized from this source (link)

Doxygen

Wouldn’t it be convenient to be able to generate documentation directly from your source code?

- Enter Doxygen, a popular tool for documentation generation.
- Available at `https://www.doxygen.nl/index.html`
- Languages supported:
  - C
  - PHP
  - Fortran
  - C++
  - Java
  - VHDL
  - Objective-C
  - Python(!)
  - D
  - C#
  - IDL
- It can generate:
  - LaTeX
  - HTML
  - man pages
  - others

Doxygen Tank

To use Doxygen, you must first invoke:

```
$ doxygen -g
```

This generates “Doxyfile”, a configuration file that has a *large* number of configurable options.

- Even more than Dwarf Fortress!

It’s important that the `PROJECT_NAME` field be set, and if your source code is strewn among several directories that will also need to be configured.

- Doxygen checks your working directory for source code files by default.

To generate the document:

```
$ doxygen Doxyfile
```

Get her some Doxygen... Stat!

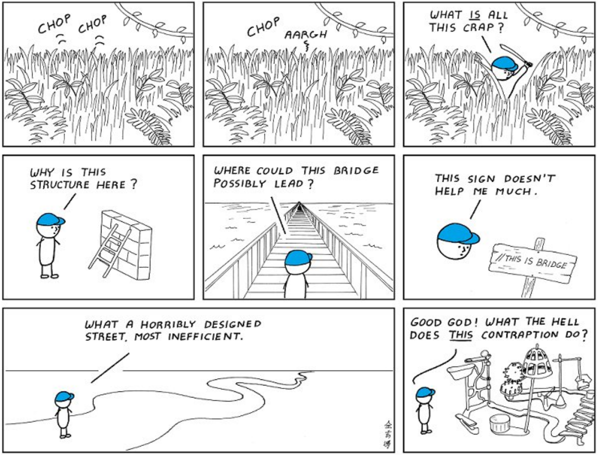
To begin a doxygen comment in C, you have to **annotate** your source code.

```
/** <= Two asterisks start a ”Doxygen comment”.  This
 * tags everything inside the comment for inclusion in
 * the generated documentation.
 */
```

Position these at the top of functions, structs and typedefs to have Doxygen document said construct with your comment.

- Doxygen also looks for **commands** to produce more informative documentation.
- Commands start with the `@` character.
- `@param` documents function parameters.
- There are a bunch of these we'll be exploring in Lab 9.

The Last Slide Comic


CHOP CHOP
CHOP AARGH
WHAT IS ALL THIS CRAP?
WHY IS THIS STRUCTURE HERE?
WHERE COULD THIS BRIDGE POSSIBLY LEAD?
THIS SIGN DOESN'T HELP ME MUCH.
//THIS IS BRIDGE
WHAT A HORRIBLY DESIGNED STREET. MOST INEFFICIENT.
GOOD GOD! WHAT THE HELL DOES THIS CONTRAPTION DO?

I hate reading
other people's code.

Credits

A lot of the contents of these slides were liberally borrowed (with permission) from slides from the Winter 2020 offering of 1XA3 (by Curtis D'Alves), and the Winter 2021 offering of 1XC3 (by Dr. Kevin Browne).