

COMP30023 Computer Systems

Assignment 1 tips and suggestions

The following may be helpful when planning and writing your web crawler.

This is not part of the project specification. If any suggestion here contradicts the specification, then follow the specification.

1. Scalability + Modularity

It's a good idea to make sure you're building your project in a scalable way. Some suggestions to help you achieve this:

- Modularize code into several .c and corresponding .h files (e.g., one file for fetching files, one for regular expressions, etc.) However, having too many files may increase the build complexity unnecessarily. One option is to start with one file and "refactor" into multiple files as your project grows. This is easier if your first file groups functions together in the way you are likely to split later.
- Use a makefile. A good example to learn from / use is here: <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
- Ensure that you test functions before assuming they work, especially considering the edge cases that C can present (NULL pointers, unterminated strings etc.). You might consider a separate file (with a different 'main' function) that executes tests, and compile that in a separate rule in the makefile.

2. URLs

There are several forms that a URL enclosed in an <a> tag can come in. In a reference from <http://www.mysite.com/examples/test2.html>, the following are all equivalent:

Type	Example
Absolute (fully specified)	"http://www.mysite.com/examples/test.html"
Absolute (implied protocol)	"//www.mysite.com/examples/test.html"
Absolute (implied protocol and hostname)	"/examples/test.html"
Relative	"./test.html"

3. Regular Expressions

Regular expressions are a powerful tool to recognize and capture components of unstructured (or structured) data. For this assignment, you can either use the standard C regex functions, or the more powerful PCRE (Perl-Compatible regular expressions) library. The regular expression syntax used by PCRE is very widely used today, including in languages like python and JavaScript.

To install this on a Debian/Ubuntu environment (including Windows Subsystem for Linux), use apt-get, and then compile your c programs with an "-lpcr" flag.

a) Never used regular expressions before?

If you've never used regular expressions before, you'll need to start by doing some research on what they are. (See the regex validation tool in below sections to test some regular expressions while you

learn). Regular expressions are very widely used, and having at least a basic understanding of what they are and how they are used will be useful in your future.

You can start off by learning in a simple language (e.g., python or JavaScript, which have pcre-compatible syntax in their regular expressions), and then you'll understand the gist of pcre.

b) Cheat Sheet

A good resource to have on hand while writing regular expressions is this 'cheat-sheet' (specifically for pcre syntax):

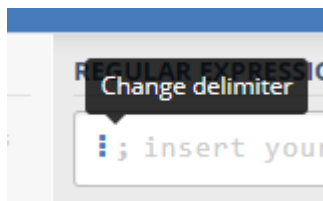
<https://www.debuggex.com/cheatsheet/regex/pcre>

c) Regex Validation

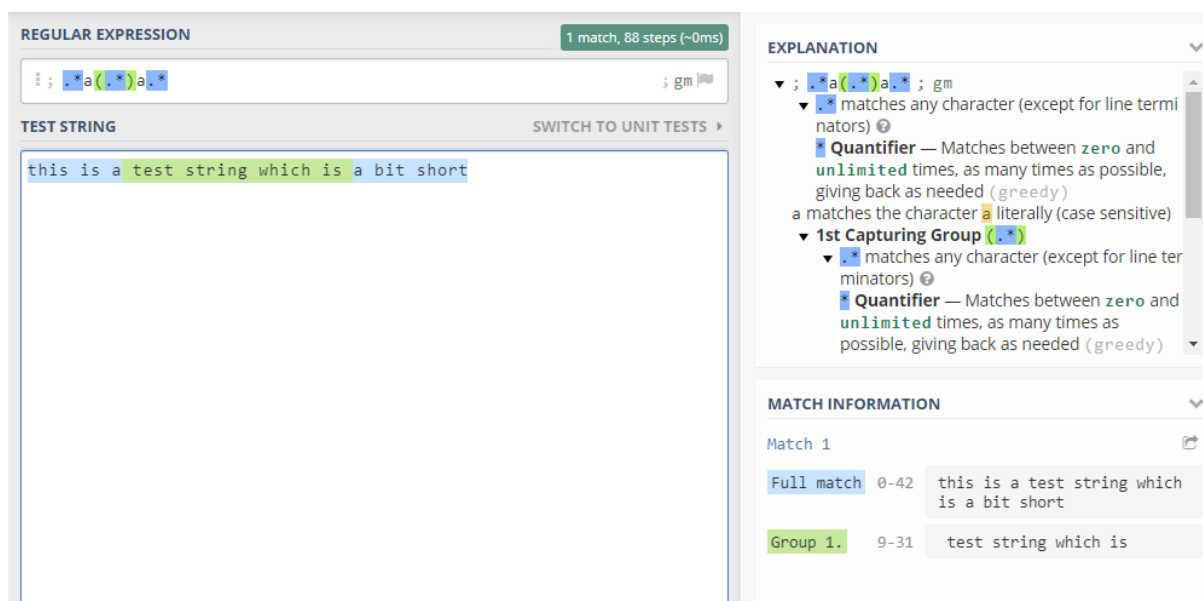
When you're designing regular expressions and you want to test that they are correct, there are several online regex validation tools that can help.

One such example is <https://regex101.com/>

Firstly, click to change delimiter (set it to something we won't be searching for, like ';')



Here's an example of what the interface looks like with a regex `".*a(.*)a.*"` executing on a string `"this is a test string which is a bit short"`. Notice that it shows the 'capture group' in green, and the full match in blue.

A screenshot of the regex101.com interface. The 'REGULAR EXPRESSION' section shows the regex `".*a(.*)a.*"` with a 'Change delimiter' button. The 'TEST STRING' section shows the text `"this is a test string which is a bit short"`. The 'EXPLANATION' section shows the breakdown of the regex:

- `.*` matches any character (except for line terminators)
- `a` matches the character `a` literally (case sensitive)
- `(.*)` 1st Capturing Group
 - `.*` matches any character (except for line terminators)
 - `Quantifier` — Matches between `zero` and `unlimited` times, as many times as possible, giving back as needed (`greedy`)
- `a` matches the character `a` literally (case sensitive)
- `.*` matches any character (except for line terminators)
- `Quantifier` — Matches between `zero` and `unlimited` times, as many times as possible, giving back as needed (`greedy`)

The 'MATCH INFORMATION' section shows the match results:

Match	Full match	Group 1
Match 1	0-42 this is a test string which is a bit short	9-31 test string which is

d) Regular Expressions with PCRE

The linux manual for pcre ("man pcre") is also available online or as a download. The pcre-demo/pcresample program that it comes with is a good starting point to see how the process of compiling and executing a regular expression works.

e) Global regular expressions with PCRE

Global regular expressions in PCRE aren't as easy in C as in other languages (which typically return a list of matches when a "g" flag is included in the regular expression).

To get an idea on how this might be done, consider looking at the PCRE documentation, specifically the pcre-demo/pcresample program mentioned above.

f) Escaping the escape characters

Escape characters are used in before characters that have special meaning to interpret them literally, or to make letters mean something special (e.g., "\." matches '.' literally instead of its usual special meaning of 'any char', and "\d" matches a digit instead of literally matching 'd').

This gets tricky when you take it up one level, because like the pcre compiler, the C compiler (gcc) is *also* using that escape characters to escape things (e.g., '\n'). As such, we need to double escape the escape characters when typing into our code. For example, say we want to match a string which has a single digit in it, like the string "1". The regular expression string we'd write is "\\d", here's why:

Parser	Conversion	Interpretation
C compiler	"\\d" => "\d"	The first \ tells the compiler to interpret the second \ literally (first \ is consumed in the parsing of the string)
PCRE compiler	"\d" => "[0123456789]"	The \ tells pcre to interpret the d as a special character, i.e., to match any digit

4. The Manual

As you've seen in labs, the manual ('man') is an important Unix tool which you should use to get an understanding on how to use Unix programs, but it also has information on C functions.

Try the following on your Unix terminal:

- "man sleep"
- "man 3 sleep"

The integer argument can be used to specify which part of the manual to look in. System calls are in the (2) section, library functions (C functions, since Unix is written in C) is in the (3) section.

The sections you are likely to use are:

- 1 Executable programs or shell commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files (usually found in /dev)
- 5 File formats and conventions e.g., /etc/passwd

(For more information, there's a manual on the manual! Type 'man man')

The first step to understanding a problem which you have never dealt with before is to visit your favourite search engine. E.g. if you were doing a project on concurrency, you might search for 'How to do concurrency in C'. Once you have learned about the broad-stroke concepts, you can start using the man pages to help you really flesh out your understanding of a particular program or function.