COMP30023 Computer Systems
Assignment 1 tips and suggestions

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

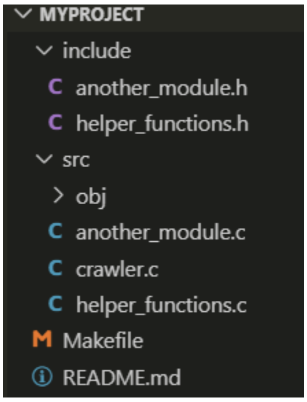
This is not part of the project specification. If any suggestion here contradicts the specification, then follow the specification.

1. Scalability + Modularisation

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

1. Modularize code into several .c and corresponding .h files (e.g., one file for fetching files, one for regular expressions, etc.). Keep in mind that 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. A suggestion might be:
   - Every .c file (aside from perhaps the file with 'main') has a corresponding .h file
   - .h files declare all the public methods and constants to be accessed by other modules, and the corresponding .c file provides the implementations of these functions and also private helper functions
   - .h files should always have "definition guards" (briefly read up on how to use these "#ifndef" guards before you get started)
2. Use a makefile (this is actually a requirement! Your project needs to build when 'make && ./crawler is run). A good example to learn from / use is here: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
3. 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 your makefile.

An *example* project structure might be:



```
MYPROJECT
  include
    another_module.h
    helper_functions.h
  src
    obj
    another_module.c
    crawler.c
    helper_functions.c
  Makefile
  README.md
```

This is a common project structure, and is a *similar* structure to that used in the final makefile provided in the reference above *(the difference is the makefile has been moved to the root of the* *project rather than being located in /src, as per the spec 'make' should be run from the root of the project. This will require you to modify the Makefile accordingly!).* In this structure, your source .c files are in a directory MyProject/src, headers are in a directory MyProject/include and object files should be placed by the compiler in the MyProject/src/obj folder.

If you were going to use a library (e.g., PCRE below), you might add a folder MyProject/lib folder to this structure.

Good programming practice is always to encapsulate your code and only provide access to the methods / constants that you wish to be 'public'. Make sure you're using ".h" (header) files for this purpose!

2. Debugging + Testing with C

Memory Sanitizers + Memory Checkers

C requires manual memory management (there is *no* garbage collection!) and so programs written in C are prone to memory leaks and segfaults.

Memory sanitisers can be useful tools in several ways:

1) If your program is crashing with a segfault or exhibiting strange behavior which you believe might be memory related, you can use a memory sanitiser to hunt down the bug
2) Even if your program appears to be working correctly, more likely than not your program is leaking memory (it is pretty likely that you forgot a free() at least one time somewhere).

There are several options you can choose from, including the Clang MemorySanitizer (requires compiling with Clang instead of gcc), or Valgrind. Another option is to use gdb (see below section) with the 'backtrace full' command which can help you trace memory related issues.

For MemorySanitizer, you'll need to install Clang from your package manager (apt-get, or possibly homebrew on mac) and then use the instructions on their site: https://clang.llvm.org/docs/MemorySanitizer.html

Valgrind can also be installed from your package manager. You can use it in a basic form to check memory errors by running "valgrind ./mycompiledprogramwith-gflag" , where your compiled program must have been compiled with the '-g' flag in gcc.. Some extra functionalites you should certainly investigate are related to the "leak-check" and the "track-origins" flags (man valgrind).

Debuggers for C

Debugging in C is difficult, and using an fprintf statement is often very useful for fixing simple issues while developing. If a serious bug is presenting itself, it might be time to graduate to a full debugger.

If you've never used a debugger in any language before, you might start by researching what a 'breakpoint' is , and what the following mean: 'step over', 'step into', 'step out of', 'continue' (these are all options you have when the execution of your code is suspended).

After you understand what debugging *is*, you're ready to use the GNU debugging tool (gdb).

I (Colton) recommend this video, which not only has information on what a debugger is, but gives in depth information on how to use gdb. https://www.youtube.com/watch?v=bWH-nL7v5F4 (yes, it is a 20-minute video, but could save you hours of debugging with fprintf!)

If you're interested in getting a bit fancier, you can setup visual debugging with vscode using this resource: https://code.visualstudio.com/docs/cpp/cpp-debug

3. Testing: A python3 webserver in one line

If you're interested in doing some testing on your own machine, you can set up a webserver on your own computer (after installing python3) in just one line:

"sudo python3 -m http.server 80"

This will start a web server on your localhost hosted on port 80 (type 127.0.0.1 in your browser to see it). Start the server in the location where the files you want to serve are located.

4. Types of URLs

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

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"

An important note is that a URL without a 'resource' is still valid: e.g. http://www.google.com is a valid URL despite not specifying a path to a resource (compare with http://www.google.com/, note the training '/'). The http protocol, however, specifically notes that 'if the absolute path in the URL is empty, it MUST be given as "/" (the server root) in the http 'Host' header' (https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html). As such, the two urls http://www.google.com and http://www.google.com/ will result in identical HTTP requests.

5. 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 (which uses POSIX syntax), or the PCRE (Perl-Compatible regular expressions) library.

Note: You'll learn a lot more about the theoretical side of Regular Expressions + Regular Languages next semester in COMP30026 Models of Computation.

The regular expression syntax used by PCRE is very widely used today, including in languages like python and JavaScript, so learning it will potentially be more useful. The downside is that you will have to bundle the PCRE files into your application package, and build them from the source when your program is compiled (since PCRE is not installed by default, and you don't have root access to install it on your VM). *See below for some hints on including + compiling from source for PCRE.*

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

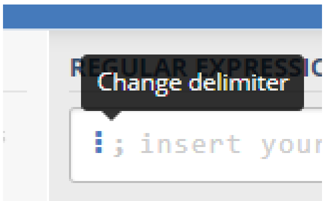
If you're planning on use PCRE, you may start off by learning in a simple language (e.g., python or javascript, which have pcre-like syntax in their regular expressions), and then you'll understand the gist of pcre.

b) Regex Validation

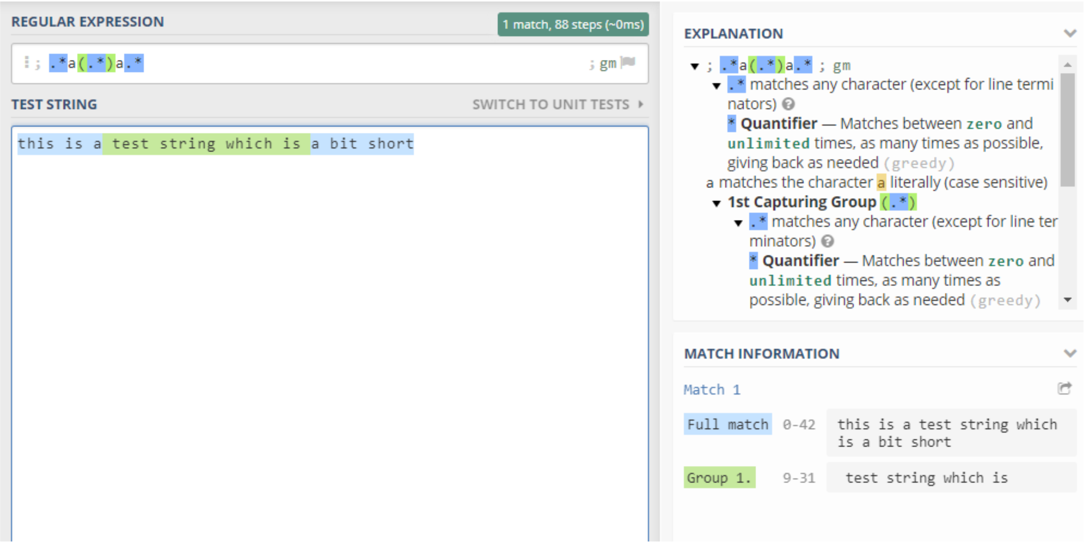
When you're designing regular expressions and you want to test that they are correct, there are several online regex validation tools that can help.

One such example is https://regex101.com/. *(This tool has support for PCRE-like syntax. If you're using the built-in C regex library, you'll need to find a tool that supports that syntax, but this tool might still be useful for testing the logic of your regular expressions, or for learning what regular expressions are)*

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:



```
this is a test string which is a bit short
```

; .*a(.*)a.* ; gm
- .* 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)
- **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)

Match 1

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

c) 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, compared with the usual special meaning of '.' being 'any char'. "\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 character '\' 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, ie to match any digit

d) Installing PCRE

⚠The VMs that will be used to run your assignment do **NOT** have PCRE installed on them, and you don’t have the ability to do so (no sudo access). ⚠

As such, to be able to use PCRE in your assignment, you’ll need to include the source files and compile these as part of your build process. It’s not as scary a process as it sounds, and it’ll teach you a bit about the usual way in which packages are compiled for Linux distribution.

The broad stroke steps are to:

a) Download the source files for PCRE as a package
b) Place the unzipped source files somewhere in your project structure, perhaps in a ‘/lib’ folder, so that when copying over to the server, PCRE’s source code is included as part of your project
c) Do some research to learn about the so-called “./configure, make, make install” default process for building unix applications from source. Many applications use this process to compile and install, however always check the README (available from https://www.pcre.org/readme.txt)
d) Create a step in your Makefile that compiles the PCRE source code and builds a static library file (.a file).
    - A good way to perform the ‘configure’, ‘make’, ‘make install’ process might be to write a small shell script (.sh file) which just runs the commands necessary (remember the ./configure, make, makeinstall process)
    - A hint: look back at the README for PCRE and figure out how to use the ‘prefix’ flag for make install to control where the library files are created: by default install tries to install them for the whole system, but this requires sudo privileges which you don’t have on the VMs.
    - Another hint: the ‘prefix’ flag requires an absolute path not a relative one, try using cd to manoeuvre to the location using relative paths, then use $(pwd) to get the absolute path the shell is currently at. You could also use another tool to resolve relative paths, but be sure to test compatibility on the VMs
e) Connect the static library file to your program when compiling in your Makefile

e) PCRE Usage

The Linux manual for PCRE (“man pcre”) is also available online or as a download. I (Colton) would specifically recommend the pcre-demo/pcresample program that it comes with it as a good starting point to see how to the process of compiling and executing a regular expression works.

f) Global Regular Expressions with PCRE (or Built-in Library)

Global regular expressions in PCRE (or the Built-in Library) 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 previously. This might also help you if you're using the built-in library, to get a broad idea of how to perform such a search.

6. Getting Help

The Manual: Detailed Information and Behaviours

As you've seen in labs, the manual ('man') is an important Unix tool which you should use to get detailed information on the use of 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) are 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

… (there are a few other sections: for more information, there's a manual on the manual! Type 'man man')

The Search Engine + Lecture Slides: Understanding the Problem and the Tools Available

The manual is a useful reference for the detailed implementation of functions and programs, but it's not as much help for figuring out *which* functions to use.

The first step to solving a problem is to understand the problem. Your favourite search engine can help you with this, as can the lecture slides. Once you're confident that you understand what needs to be done to solve the problem, then it's time to start thinking about how to implement that solution, in this case this would involve reading about how to complete this process using C and Unix standard library functions.

As a simple example: Imagine yourself having zero computing/coding experience, and you want to learn how to print out the numbers 1 to 10 on the command line.

a) The first step would involve researching the problem, and eventually figuring out what "loops" and "print statements" are.
b) From there, you might learn about how to implement these constructs in python, and read some example programs that do a similar thing.
c) You now have a solid base on which to begin attempting to implement that program.