H

hw2-doc

Name	Last update
DebuggingRef.md (/cse320/hw2-doc/blob/master/DebuggingRef.md)	about 16 hours ago
README.md (/cse320/hw2-doc/blob/master/README.md)	about 16 hours ago

README.md (/cse320/hw2-doc/blob/master/README.md)

Homework 2 Debugging and Fixing - CSE 320 - Spring 2018

Professor Eugene Stark

Due Date: Friday 02/23/2018 @ 11:59pm

Introduction

In this assignment you are tasked with updating an old piece of software, making sure it compiles, and works properly in your VM environment.

Maintaining old code is a chore and an often hated part of software engineering. It is definitely one of the aspects which are seldom discussed or thought about by aspiring computer science students. However, it is prevalent throughout industry and a worthwhile skill to learn. Of course, this homework will not give you a remotely realistic experience in maintaining legacy code or code left behind by previous engineers but it still provides a small taste of what the experience may be like. You are to take on the role of an engineer whose supervisor has asked you to correct all the errors in the program, plus add additional functionality.

By completing this homework you should become more familiar with the C programming language and develop an understanding of:

- How to use tools such as `gdb` and `valgrind` for debugging C code.
- Modifying existing C code.
- C memory management and pointers.
- Working with files and the C standard I/O library.

The Existing Program

Your goal will be to debug and extend the `snarf` program. This program is a simple HTTP client, which can contact an HTTP server over the Internet and retrieve a document. This particular program was written over twenty years ago, in the very early days of the Web. A version of this program was known to function at one point, though it is not clear whether the version you will receive is the actual functioning code or perhaps an incompletely debugged predecessor. One thing for sure is that the simple way in which this program uses the HTTP protocol is still compatible with modern HTTP servers, so you should not have to worry about debugging the use of HTTP itself. However, as often happens with code that has not been updated for a long time, this program has "rotted" a bit, in the sense that it uses a library function `fgetln()`, which was part of the Berkeley Unix (BSD) standard I/O library but which is not found by default on Linux systems. Although it is possible to find versions of the BSD standard I/O library that will run on Linux, the `fgetln()` function has been deprecated and should no longer be used. To get the program to function, you will need to replace the uses of `fgetln()` by the `getline()` function, which is supported by Linux and is also a POSIX standard.

Getting Started - Obtain the Base Code

Fetch base code for `hw2` as you did for the previous assignments. You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw2 (https://gitlab02.cs.stonybrook.edu/cse320/hw2).

Once again, to avoid a merge conflict with respect to the file `.gitlab-ci.yml`, use the following command to merge the commits:

```
    git merge -m "Merging HW2_CODE" HW2_CODE/master --strategy-option theirs
```

Here is the structure of the base code:

```
.
├── .gitlab-ci.yml
└── hw2
    ├── include
    │   ├── debug.h
    │   ├── http.h
    │   ├── ipaddr.h
    │   ├── snarf.h
    │   └── url.h
    ├── Makefile
    ├── src
    │   ├── args.c
    │   ├── http.c
    │   ├── snarf.c
    │   └── url.c
    └── tests
        └── snarf_tests.c
```

Before you begin work on this assignment, you should read the rest of this document. In addition, we additionally advise you to read the Debugging Document (/cse320/hw2-doc/blob/master/DebuggingRef.md).

HTTP -- In Brief

HTTP (HyperText Transfer Protocol (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)) is the network protocol that is used by Web browsers to contact Web servers and retrieve documents (Web pages and associated data) over the Internet. It was initially developed by Tim Berners-Lee, the "father of the World-Wide Web". You don't really have to know very much technical information about networking or HTTP to do this assignment, but the following discussion should help you to get oriented.

A client application wishing to retrieve a document from a Web server typically starts out with a URL (Uniform Resource Locator (https://en.wikipedia.org/wiki/URL)). Although there is a precise specification (https://tools.ietf.org/html/rfc1738) for the syntax of URLs, to which a standards-compliant client must adhere, for our purposes we will be somewhat informal and simply regard a URL as having the following general form:

```
    method://hostname:port/path-to-resource
```

Here, `method` is a keyword describing the method or protocol to be used to access the resource. For us, `method` will always be `http`. The `hostname` is either an Internet hostname or an IP address in "dotted quad" form (*e.g.* `192.168.1.3`). If a hostname is provided, it is mapped to an IP address using DNS (Domain Name System (https://en.wikipedia.org/wiki/Domain_Name_System)). The IP address obtained from the URL is used to open a connection to the server using TCP (Transmission Control Protocol (https://en.wikipedia.org/wiki/Transmission_Control_Protocol)). The `port` is a number that identifies which one of possibly several services running on the server is to be contacted. Standard Internet protocols such as HTTP have standardized, "well-known" port numbers. For HTTP the well-known port number is 80, though an HTTP server may also accept connections on nonstandard ports.

> 🤓 The URL syntax is fairly flexible: many parts are optional in various circumstances. For example; the `:port` can be omitted, the `/path_to_resource` can be omitted, or the `path_to_resource` can be empty. A program accepting a URL as input from a user should be prepared to handle an arbitrary string. It must check that the provided string can be interpreted as a URL, with necessary fields present, and it must perform this validation without crashing, regardless of the string the user entered.

Once a connection has been established with the server, the client sends information describing the request. We are only interested in the very simplest form of HTTP request: `GET`. To issue a `GET` request to a server it is sufficient for the client to send two lines of text of the following form over the connection.

```
GET method://hostname:port/path-to-resource HTTP/1.0
Host: server
```

In the above, `GET` is a fixed keyword identifying the type of request, `method`, `hostname`, `port`, and `path-to-resource` are as previously described, and `HTTP/1.0` is fixed information indicating that the HTTP protocol, version 1.0, is to be used.

Following the `GET` request line can be one or more *request headers*, each of which is a `keyword: value` pair that occurs on a separate line. The header keywords are not case-sensitive. For HTTP 1.0 there is one required request header, which has keyword `Host` and whose value identifies the server to which the request is directed. Usually, `server` would be the same as `hostname`. The request line and each of the lines with the request headers are terminated by a two-byte CR-LF sequence (`"\r\n"` as a C string). Following the last request header is a blank line, consisting of just the line-termination sequence (again, `"\r\n"`).

When the server sees the blank line that terminates the list of request headers, it sends its response. The response begins with single response line, of the form:

```
HTTP/1.0 code
```

This line uses the same line termination sequence as the request lines. The `HTTP/1.0` echoes the protocol version indicated by the client, and the `code` is a numeric *status code* that indicates the disposition of the request. A list of status codes can be found here (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes), but we are only interested in a few of them and these are listed in comments in the base code you have been given.

Following the response line is a sequence of response headers, similar to the request headers, with a blank line indicating the end of the response headers. Following this blank line is a sequence of bytes comprising the document payload. If the response headers include a header of the form: `Content-length: NNN`, then the number `NNN` indicates the number of bytes of payload that follows the headers. The client can use this to determine how much data to read from the network connection. Once the payload has been consumed, the client closes the network connection and the transaction is complete.

Part 1: Fixing and Debugging

The command line arguments for this program are described in the `USAGE` macro in `include/snarf.h`.

The `USAGE` statement defines the corrected expected operation of the program.

> 😱 You MUST use the `getopt` function to process the command line arguments passed to the program. Your program should be able to handle cases where the flags are passed IN ANY order. This does not apply to positional arguments

You can modify anything you want in the assignment.

Complete the following steps:

1. Fix any compilation issues

2. Ensure runtime correctness by verifying output with the provided Criterion unit tests and your own created unit tests.

3. Use `valgrind` to identify any memory leaks or other memory access errors. Fix any errors you find.

Run `valgrind` using the following command:

```
valgrind --leak-check=full --show-leak-kinds=all [SNARF PROGRAM AND ARGS]
```

> 😱 You are **NOT** allowed to share or post on PIAZZA solutions to the bugs in this program, as this defeats the point of the assignment. You may provide small hints in the right direction, but nothing more.

Part 2: Adding Features

Add the following additional features to complete the program's functionality:

- **Exit status:** If the program is completely successful (HTTP status code 200 and no error of any kind), then the exit status should be 0. If the HTTP status code is anything other than 200, the exit status should be the HTTP status code. If any other kind of error occurs (*e.g.* invalid arguments, error saving document to a file), the exit status should be -1.
- Implement the `-h` option. If this option is present, then the USAGE macro should be used to print a help message and the program should exit with zero exit status.
- Implement the `-q keyword` option. If this option is present, then the HTTP response headers will be searched for header lines having the specified keywords. Any matching header lines will be output *to* `stderr`. This option may be given more than once, with different keywords, to search for several different keywords in the same run.
- Implement the `-o file` options. If this option is present, then the retrieved document payload is saved as the specified file, rather than being printed on `stdout`.

Unit Testing

For this assignment, you have been provided with a basic set of Criterion unit tests to help you debug the program. We encourage you to write your own as well as it can help to quickly test inputs to and outputs from functions in isolation.

In the `tests/snarf_tests.c` file, there are six unit test examples. You can run these with the `bin/snarf_tests` command. Each test is for a separate part of the program. For example, the first test case is for the `parse_url` function. Each test has one or more assertions to make sure that your code functions properly. If there was a problem before an assertion, such as a SEGFAULT, the unit test will print the error to the screen and continue to run the rest of the tests.

To obtain more information about each test run, you can use the verbose print option: `bin/snarf_tests --verbose=1`.

You may write more of your own tests if you wish. Criterion documentation for writing your own tests can be found here (http://criterion.readthedocs.io/en/master/).

Besides running Criterion unit tests, you should also test the final program that you submit with `valgrind`, to verify that no memory errors are found.

Hand-in Instructions

Ensure that all files you expect to be on your remote repository are pushed prior to submission.

This homework's tag is: `hw2`

```
$ git submit hw2
```

> 🤓 When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.