CS142 **Project 1** Spring 2025

Given: Wed, Feb 19
Due: Fri, Mar 28, 10 p.m.

Overview It's 1989. The Internet is almost 20-years-old and it's growing. More and more people are using it to communicate with each other by using something called *electronic mail* (email) and to access remote documents by using the *file transfer protocol* (FTP). Some people are building services that index the growing number of documents available on the Internet to make them easier to find.

But then you have an idea. Right now, when you're reading a document that mentions another document and you decide you want to see that second document, you need to first figure out where that second document is on the Internet and then use FTP to access it. What if you could instead just select the second document from within the first one and be brought directly to that second document?

In other words, what you're proposing is to *link* all these documents together so that you can easily move from one document to another. In a sense, you're proposing to create a *web* of documents, and you want that web to be *world-wide*...Congratulations, you just invented the *World Wide Web*! :-)[1]

This project is to create a prototype for a Web browser. You'll do this by

[1]The actual inventor of the Web is Tim Berners-Lee. That invention earned him a Turing Award, which is widely viewed as the Noble Prize of computing. That award is given out annually by the Association for Computing Machinery (ACM), one of the main associations for computing professionals in the world. You can read more about Berners-Lee's work in the award citation: `https://amturing.acm.org/award_winners/berners-lee_8087960.cfm`.

transforming the file viewer we created in class into a browser for a *web of files* that are all located on a single computer. (This is in contrast to the real World Wide Web whose *pages* are located all over the world.)

Details The files in this web of files are plain text files that may contain *anchors* that provide links to other files. More precisely, an anchor is a string of the form

<a *filename text*>

where *filename* is the name of another file and *text* is a piece of text. For example, here's a line of text that contains an anchor:

```
More <a info.txt information> is available.
```

To keep things simple, assume that every anchor is separated from adjacent text by white space and that the strings *filename* and *text* cannot contain white space.

Your browser should work exactly like the file viewer we created in class, except for the following:

1. When an anchor is displayed, it should appear as

   <*text*>[*n*]

   where *n* is a number that is unique to this anchor. In each file, anchors should be numbered in order, starting at 1. For example, if the line

   ```
   More <a info.txt information> is available.
   ```

   contains the 5th anchor of the file, then it should be displayed as follows:

   ```
   More <information>[5] is available.
   ```

2. The browser should have a new command called *go*. This command asks the user for a number and then opens the file associated with the corresponding anchor:

```
command: g
link number: 5
```

   In the above example, this should cause the file `info.txt` to be displayed.

3. The browser should have another new command called *back* that reopens the previous file. This requires the browser to keep a record of the files that have been visited. For example, if the user visits files A, B and then C, the *back* command would return to B. But note that using the *back* command again would go to A, not back to C. In other words, files should get added to the "history" only when they are visited by following a link (with the *go* command) or by using the *open* command, not by using the *back* command.

4. The browser should format the contents of each file as follows. Each file is assumed to consist of a sequence of "words" separated by white space. Note that, in this context, a word may include punctuation. For example, the following line contains six words:

```
Each component performs one well-defined task.
```

   When a file is displayed, all the words should be separated by a single blank space and arranged so that the lines are as long as possible but no longer than a maximum length specified by the user. If ever a word is longer than that length, it should be displayed on a line by itself. The program should

ask the user for the maximum line length right after it asks for the window height. The maximum line length should be a number of characters, not words.

5. Besides anchors, files can contain two other *elements* that affect how the text is displayed. The *line-break* element, written `<br>`, causes a new line to begin immediately, even if the maximum line length hasn't been reached. The *new-paragraph* element, written `<p>`, causes a new paragraph to be started. When displayed, consecutive paragraphs should be separated by a single blank line. Blank lines in the input file should be ignored. To keep things simple, assume that the `<br>` and `<p>` elements are separated from surrounding text by white space.

Figure 1 shows a sample file and Figure 2 shows how that file should be displayed if the maximum line length is set to 40.

Teamwork The assignment policy available on the course website also applies to this project with one exception: you are allowed to do the project as a team of up to three students. In fact, you are encouraged to do the project as a team because most software is built this way and it's a good idea for you to start getting used to it. It's also less work and it can be more fun.

If you choose to do the project as a team, you should design the browser as a team, but a lot of the implementation work can be divided. For example, one person can write the formatting code (Items 4 and 5 above) while the other writes the code that deals with anchors and the *go* and *back* commands (Items 1 to 3).

If you work as a team of three, since this is not a very large program, it may be difficult to divide the implementation work in three. One option is to have two people work together on the same part of the program. You could do that

```
A <a modularity.txt modular> program consists
of software components that satisfy the following
two properties:
<p> 1. Cohesion: each component performs one
well-defined task.
<p> 2. Independence: each component is as <a
independence.txt independent> as possible from the
others.


<p> Modularity is usually achieved through some
combination of the following:
<p>
procedural abstraction <br>
data abstraction <br>
object-oriented programming
```

Figure 1: Sample file with anchors

```
A <modular>[1] program consists of
software components that satisfy the
following two properties:

1. Cohesion: each component performs one
well-defined task.

2. Independence: each component is as
<independent>[2] as possible from the
others.

Modularity is usually achieved through
some combination of the following:

procedural abstraction
data abstraction
object-oriented programming
```

Figure 2: Sample file of Figure 1 as displayed by the browser with a maximum line length of 40

even if you work as a team of two. But make sure that every team member gets a chance to write some of the code.

Note that even if you divide the implementation work with your teammates, you should expect to have to help each other out throughout the project.

Advice Whether you work as a team or on your own, I strongly suggest that you build the browser very gradually. For example, you could start by displaying anchors properly (Item 1 above). Then you could implement the *go* command. And then the *back* command. Even the formatting can be done gradually. For example, at first, you could ignore the `p` and `br` elements. It's important not to try to do everything at once.

The file viewer reads files one line at a time. In the file browser, you might want to read files one word at a time. This can be done by using the input operator (`>>`) instead of `getline`.

Submission Submit your project on Moodle by the deadline (even if it's not complete). If you work as a team, submit only one copy of your project. Include the following in your submission:

- *A design document*. This should be complete, precise and easy to understand. You can use the design document of the file viewer as a starting point. Make sure your program is highly modular.
- *All your code*. Make sure you write standard C++ code so we can compile it without problems. Your code should be easy to understand. Pay particular attention to indentation and choice of names for variables, classes and functions.
- *A status report*. It should state whether your program works. In case it doesn't work perfectly or isn't complete, explain exactly what doesn't work

or what remains to be done. If you work as a team, list the names of the team members at the top of the report.