```
Some people, when confronted with a problem, think “I know,
I'll use regular expressions.”  Now they have two problems.
-- Jamie Zawinski
```

Regular expressions are introduced in CSCI 274 but unless you have taken a formal languages or compilers it is likely you have not used them very often. Regular expressions can be incredibly powerful and are the perfect tool for solving a large class of problems. Unfortunately, the dense syntax used makes them look far more complex than they really are and causes some to shy away from using them.

The web server project is the perfect example of a problem ideally suited to regular expressions. We find ourselves with a buffer containing the request method sent by the web browser. And we must figure out if it contains a valid request or not.

In other words, we have something that looks like this:

```
GET   /file1.html   HTTP/1.1<CR><LF>   Host:isengard.mines.edu:6246<CR><LF>   Connection:keep-
alive<CR><LF>  Upgrade-Insecure-Requests:1<CR><LF>  User-Agent:Mozilla/5.0  (Macintosh;  Intel
Mac    OS    X    10_15_7)    AppleWebKit/537.36    (KHTML,    like    Gecko)   Chrome/93.0.4577.82
Safari/537.36<CR><LF>Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/avif
,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9<CR><LF>       Accept-
Encoding:gzip,deflate<CR><LF>Accept-Language:en-US,en;q=0.9<CR><LF>Cookie:_gcl_au=1.1.113
0642752.1631067074;_ga=GA1.2.238980797.1631067074;nmstat=233fd061-79bb-5894-97a59462c2db5cd3
;_fbp=fb.1.1631067073791.275400384; _gid=GA1.2.1542338347.1632005173<CR><LF><CR><LF>".
```

Given this input we need to figure out if it contains any of the possible valid GET requests. Since there can be an arbitrary amount of space on the line, at least 20 different valid filenames and 4 different valid versions of HTTP, hard coding a bunch of comparison operators would be messy and error prone. Regular expressions are a much better solution.

I. RECONGIZING VALID GET REQUESTS

The first step is to formalize our understanding of a valid GET request. Fortunately, the authors of the HTTP standard have done that for us. RFC 7230 specifies that a valid HTTP message must have the format:

```
HTTP-message   = start-line
                  *( header-field CRLF )
                  CRLF
                  [ message-body ]
```

Note the * before means “zero or more” header fields. For the assignment I've told you that you may ignore header lines and any message-body, so we can simplify this to

```
HTTP-message   = start-line
                  *(anything)
                 CRLFCRLF
                 *(anything)
```

In other words, all we really need to worry about is the start line. In the specification the start line can be either a request line or a status line (the | means "or").

```
    start-line = request-line | status-line
```

Since we are only going to process requests, we can focus on the request line.

```
    request-line   = method SP request-target SP HTTP-version CRLF
```

For the assignment we only have to accept the GET method, so our request line becomes:

```
    request-line   = "GET" SP request-target SP HTTP-version CRLF
```

The HTTP-version is defined as

```
    HTTP-version  = "HTTP/" DIGIT "." DIGIT
```

The request-target must be one of the allowed filenames (fileX.html or imageX.jpg) where X is a single digit.

```
  Request-target = "file" DIGIT ".html" | "image" DIGIT ".jpg"
```

II. WRITING REGULAR EXPRESSION LANGUAGE

There are many flavors of the regular expression language, I will use the version called ECMAScript which is the default for the C++ standard library. We only need five pattern matching characters:

```
  \s    => means a whitespace (space,tab,LF,CR etc.)
  \S    => means anything but whitespace.
  \d    => means a digit (0-9.
  \r    => means a carriage return.
  \n    => means a line feed.
  \.    => means a period.
```

We also need a quantifier (symbols that indicate something repeats).

```
  +     => means one or more times.
  |     => means match one or the other
```

With just those symbols, we can write an expression that will match any request that is valid for this assignment.

$$\underbrace{\texttt{GET}}_{\text{method}}\texttt{\textbackslash s+/}\underbrace{\texttt{(file\textbackslash d\textbackslash .html|image\textbackslash d\textbackslash .jpg)}}_{\text{Request Target}}\texttt{\textbackslash s+}\underbrace{\texttt{HTTP/\textbackslash d\textbackslash .\textbackslash d}}_{\text{HTTP-Version}}\texttt{\textbackslash r\textbackslash n}$$

II. CODING THE EXPRESSION IN C++

The C++ regular expression library has you create the regular expression you want to match, and then allows you to search a string for that expression (https://www.cplusplus.com/reference/regex/).

You will want to look at two object types and one method. The object types we need to use are and expression and a search-match (smatch)

II.A std::regex – define the regular expression

The expression type (std::regex) will contain the regular expression we are going to search for. For example, if we want to search a for the substring that looks like a filename at the end of a line. That is anything that is a word, a dot, and then three characters then a carriage return and a newline.. To do this we can use the regular expression `\sW+.\W\W]W\r\n`. In C++ we will create the regular expression object.

```
#include <regex>

std::regex e ("\\s\\W+\\.\\W\\W\\W\\r\\n");
```

Notice the extra backslashes in the string. Because the C++ compiler has its own set of escape characters it will try and interpret a single backslash as a C++ escape and will remove what it thinks are illegal or unknow escape characters. To have the compiler create a single backslash ("\") we have to use the C++ notation for a backslash, which is two backslashes ("\\")

II.B std::smatch – capture a substring

One exceptionally useful feature of the C++ regular expression library is called *capturing*. Capturing allows us to set a variable with part of a match so we can see what it contained. The C++ syntax for capturing is to put the part of the regular expression we want to capture in parentheses. If we want to know what the filename that was in our string looked like we would write[1]:

```
#include <regex>

std::regex e ("\\s(\\w+\\.\\w\\w\\w)\\r\\n");
```

The library puts anything we capture in the an array like structure called a smatch:

```
#include <regex>

std::smatch sm;
```

II.C std::search – search the string for a matching regular expression.

[1] The \w escape character is any word character. That is anything in the range a-zA-Z0-9

The regular expression library includes two types of methods that perform the search. A “match” and a “search”. The match requires that the whole string be matched for the regular expression. If we wanted to use match we would have to write a regular expression that matches everything the browser might send (which is more work that we want/need to do).

A search looks through the input to see if any part of the input matches the regular expression. We will use search. The search method takes three parameters: the input, the capture structure, and the regular expression. It will return true if the expression matches somewhere in the input, false if there is no match.

```
#include <regex>

std::string inputString ("Search in this string for foo.bar\r\n");
std::regex e ("\\s(\\w+\\.\\w\\w\\w)\\r\\n");
std::smatch sm;


if (std::regex_search (inputString, sm, e)) {
    std::cout << "string object matched" << std::endl;
    std::cout << "sm[0] = " << sm[0] << std::endl;
    std::cout<< "sm[1] = " << sm[1] << std::endl;
}
else{
    std::cout << "Input string did not match" << std::endl;
}
```