# IT441

## Network Services Administration

# Handling Text:

# *Regular Expressions*

DRAFT

# Searching for Text in a File

- Make note of the following directory:

  **`/home/ckelly/course_files/it441_files`**

- Given the file **`gettysburg.txt`** in my **`it441_files`** directory) you will write code to find if a given word is used in the file.
- You will use the **`split`** function as such:
  - **`my $word (split)`** (*will make more sense in context*)
  - This will *break up* the line into words separated by *whitespace* characters (space, tab, newline, etc.)
- If the word is found, print out a message indicating such.
- **`readAndPrint1.pl`** in **`it441_files`** shows it as a script.

# Using sed

- Get the file `now1.txt` from `it441_files`
- How would you use `sed` to change the word `Now` to `Then`?
- How would you used `sed` to change the phrase `Now is` to the phrase `Then was` ?
- Congratulations, you have just used your first ***regular expression***!

# Regular Expressions

- Often called "*regex*", for short
- One of the most useful features of Perl
- Available in most programming languages
- You used it in ***bash***
- What does this do?  (*Contrast to transliteration*)

  `s/abc/ABC/`
- Welcome to the world of regular expressions

# Searching for Text in a File

- Given the text file `gettysburg.txt` in `it441_files`, you'll write code to find if a given word is used in the file.
- Use a regular expression such as

  `/and/`
- If the word is found, print out a message indicating such.
- `readAndPrint.pl` in `it441_files` shows the solution in the form of a script.

# Searching for Text in a File

- What is different between the two methods of searching?

- What if I want to ignore the case of the letters?
  - How would you modify the **`readAndPrint1.pl`** program to check without regard to capitalization?
  - It is not very easy is it!
- With regular expressions it is much simpler.

  **`/$word/i`**

# Anchors

- What if I wanted to find a pattern only at the start of a line or at the end of a line?
- In a regex, we can use *anchors*.
  - To indicate the pattern must be at the ***start*** of a line, we use the anchor `^`

    e.g., `/^The/`
  - To indicate the pattern must be at the ***end*** of the line, we use the anchor `$`

    e.g., `/end$/`

# Metacharacters

- We can make our regex more general by using ***metacharacters***.
  - See the table on page **167** in the text
- Four of the more common metacharacters are:
  - `.` Matches **any character** (except newline)
  - `?` Preceding character or group may be present **0 or 1** time
  - `+` Preceding character or group is present **1 or more** times
  - `*` Preceding character or group may be present **0 or more** times

# Metacharacters

- What does the metacharacter `.` do?
  - It matches any single character!
  - So give some examples of what `/Bet.y/` would match?
- What does the metacharacter `*` do?
  - It matches if the preceding character or group may be present `0` or more times
  - So give some examples of what `/Bet*y/` would match

# Metacharacters

- What will happen if we combine these two metacharacters?
  - What will the regex `/fred.*barney/` match?
  - How about the regex `/.*/` ?
- How about if we wanted to find the string **3.14159** ?
  - What would this regex match?
    - `/3.14159/` (remember the `.` is a metacharacter)
  - So we need to "*escape*" it like we do in a double quoted string
    - `/3\.14159/` will work
- If we want to use any metacharacter to represent itself we need to "escape" it, using the backslash: `\`

# Metacharacters

- What does the metacharacter `+` do?
  - It matches if the preceding character or group is present 1 or more times
- What string will the following regex match?

  `/cat/`
- What about?

  `/cat+/`
- Or?

  `/(cat)+/`

# Groups and Memories

- What did the string `/(cat)+/` match?
- Using parenthesis causes the string to be ***grouped***
- The `+` sign will apply to the *group* `cat`, matching the following...
  - **cat**
  - **catcat**
  - **catcatcat**
  - **catcatcatcat**
  - *And so forth...*

# Groups and Memories

- Grouping also allows us to store matches in variables
- You do this by using the string and regex with the binding operator =~ ...for example:

```
"foobar" =~ /(oba)/ ;
```
**OR**
```
$str =~ /(oba)/ ;
```

- It will store the matched group in the variable **$1** , where you can use it like any other variable:

```
print "Matched: $1"
```
→
```
"Matched: oba"
```

- You can also store matches from multiple groups, which will be stored in **$1** , **$2** , **$3** , etc.

# Groups and Data Capture

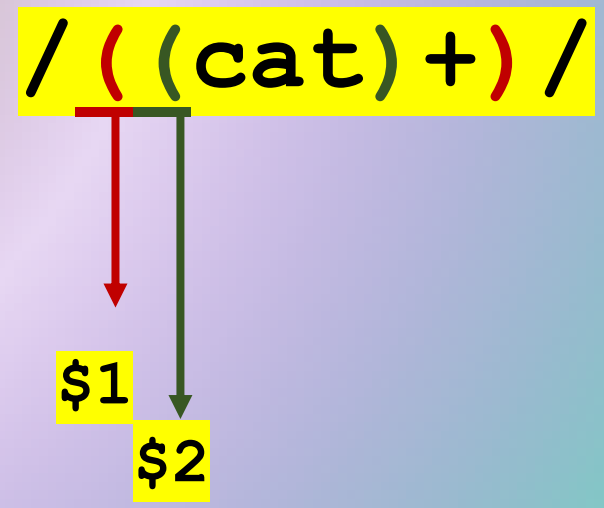
- For example, consider the following.

```
$info = "I have a catcatcatcat, yes I do!" ;
$info =~ /((cat)+)/ ;
```

- We would then have info stored in **$1** and **$2** :

```
print "1st: $1\n2nd: $2\n";
1st: catcatcatcat
2nd: cat
```

- This is because Perl matches *groups* to *numeric variables* according opening parentheses, left to right



# Groups and Data Capture

- You can also put multiple matches into an array!

```
$info = "My friends are Betty, Betsy, and Betey." ;
@names = ( $info =~ /(Bet.y)/g );
```

- We would then have the matches stored in `@names` :

```
foreach $name (@names) { print "$name\n"; }
Betty
Betsy
Betey
```

- The `g` is one of many options you can add to a regex
- For example, you can also use the `i` option for a case-insensitive match

# Alternatives

- What if we want a *choice* in our regex?
  - We can use the ***logical OR:*** `|`
  - We want to find out if either the name `fred` or `barney` is in the text. We would write the regex:

    `/(fred)|(barney)/`

# Character Classes

- We can include *groups of characters* in a regex
  - `[0-9]` will match any ***number***
  - `[a-z]` will match any ***lower*** case letter
  - `[A-Z]` will match any ***upper*** case letter
- How about `[a-zA-Z]`?
- There are shortcuts for these classes
  - `\d` represents `[0-9]` a ***digit***
  - `\D` represents `[^0-9]` a ***non-digit***

  There are others as we will see on page 167

# The Binding Operator

- So far, we have been matching against the default **$_**
- What if we want to match against *another* variable?
- We need to use the ***binding operator*** **=~**
  - `if ($someOther =~ /fred/) { action}`
  - This will perform the action if the match is true

# Two Functions

- Remember, earlier, we used the `split()` function to break the string up into substrings separated by whitespace.
- We can use any arbitrary character to split the string.
- For example, `my @field = split (/,/, $str_gettys)` will
  - Split the `gettysburg.txt` file (whose contents are in `$str_gettys`) into phrases separated by commas and
  - Store each phrase in a separate entry in the array `@field`
- You will write code to try this.

# Two Functions (cont.)

- There is also a `join()` function, to which you supply:
  - a string (the "*joiner*")
  - a *sequence*, such as an array
- For example, `join ("#", @fields)` will cause
  - the elements in the array `@fields`
  - to be combined into one string
  - with each field separated with the character `#`