I/O and Text Processing

Data into and out of programs

Extending I/O

- You have seen that input to your program can come from the keyboard and that in Perl, a statement such as

  ```
  $var = <STDIN>;
  ```

  reads a line of text typed at the keyboard into the variable named **`$var`**.
- You have also seen that you can send output to the console with the **`print`** statement.
- In this lesson, a program's ability to perform I/O is extended to include I/O to and from arbitrary files and devices.

I/O connections

- So far you have been able to get input from the keyboard and send input to the console from within Perl using **`<STDIN>`** and **`<>`** to read lines and the **`print`** statement to send output.
- You have also seen how shell I/O redirection operators can be used to replace standard input by data from files, to send output data to files instead of the console, and to pipe data from a command into your program, as in

```
$ nextdemo_02.pl < datafile
$ cat datafile | nextdemo_02.pl  > outfile
```

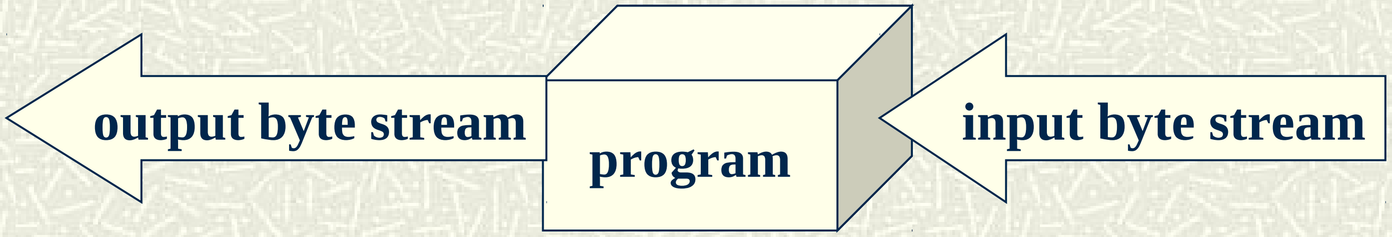
- How does UNIX make this possible?

Device independent I/O

- UNIX makes it possible for you to write programs that do not need to be aware of whether data comes from the keyboard, a different input device, a file, or even a program running on the computer. The same concept holds for output.
- It achieves this by creating a level of abstraction that separates your program from the source of input and destination of output.
- The abstraction is called a ***byte stream***, or ***data stream***.

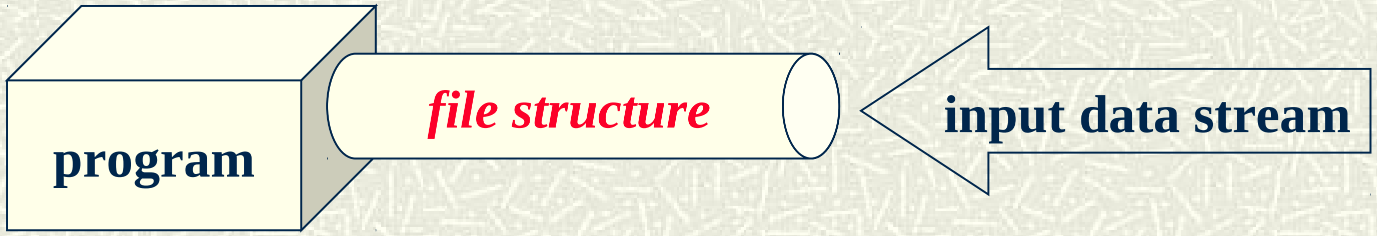
Byte streams

- A ***byte stream*** is an abstraction of a flow of data into or out of your program. The data can be a stream of characters typed on the keyboard, flowing into your program, or the characters sent to the terminal by your program.
- Data can come from or go to devices, such as keyboards or terminals, files, other programs, and network connections.



File structures

- To allow a data stream to be transmitted to or from a program, the UNIX kernel maintains a ***file structure*** to control the flow of that data stream. This structure provides, among other things, temporary storage for the data, status information, and pointers to the next place to read or write. Data passes through this ***file structure*** en route to or from your program.



I/O under kernel control

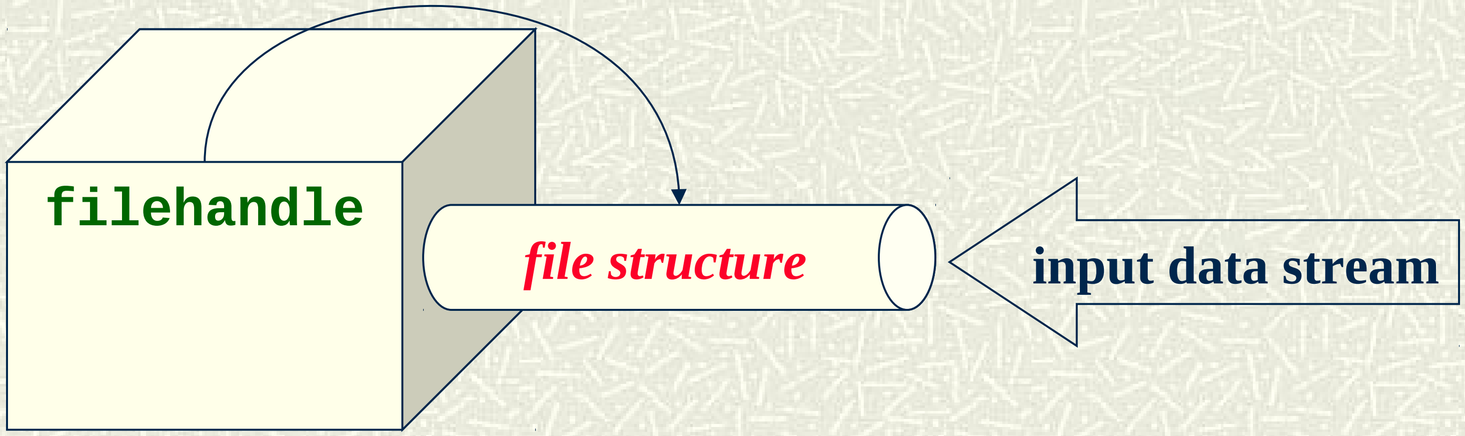
- In UNIX, user programs are not allowed to read from or write to devices or files; ***all input and output operations are performed by the operating system***.
- When your program needs I/O, the program issues a request to the UNIX kernel to perform that I/O. The kernel itself performs the operation and notifies the program when it is finished, and whether it was successful.
- Not all I/O is successful; errors do happen. The kernel detects errors and responds to device failures and other I/O problems. It also provides a means for programs to obtain the status of I/O jobs.

File descriptors

- In order to allow a program to perform I/O, UNIX creates the byte stream's file structure that was mentioned earlier, and creates a unique identification number for that structure. This identification number is called a ***file descriptor***.
- ***The file descriptor is needed in order to perform any I/O operation on the stream, whether it is reading, writing, or both.***

File handles

- Perl programs do not use file descriptors. They use ***file handles***. A ***file handle*** is a name in a program associated with a file descriptor. A file handle is what the program uses to access the byte stream that the file descriptor references.



Three standard file handles

- When a Perl program starts, it is given three, predefined file handles: **`STDIN`**, **`STDOUT`**, and **`STDERR`**.
  - **`STDIN`** is the standard input stream, ***usually associated with the keyboard*** unless the program was opened with an I/O redirection operator such as **`<`**.
  - **`STDOUT`** is the standard output stream, ***usually associated with the console***, unless I/O redirection was used.
  - **`STDERR`** is the standard error stream, ***usually associated with the console***. It is the output stream in which error messages are written.

Using the standard file handles

- You have already seen that when **`STDIN`** is enclosed in angle brackets, it becomes an input operator.
- Truth be told, **`<>`** is an operator called the ***angle operator***, or ***input operator***. If a file handle is placed inside it, it acts as a source of input data.
- In contrast, to print to **`STDOUT`**, you put the file handle directly after the word **`print`** in a **`print`** statement, e.g.,

```
print STDOUT "Welcome.\n";
```

- Of course if you omit **`STDOUT`**, output goes to the console anyway. ***Notice that there are no commas before or after*** **`STDOUT`**.

Writing to standard error

- To print to **STDERR**, you put the file handle **STDERR** directly after the word **print** in a **print** statement.

```
print STDERR "Error 52 occurred. Call tech
  support at 1-800-NOWHERE.\n";
```

- Standard error is a separate stream in UNIX. It is useful to be able to separate error messages from ordinary output, because certain users do not want to see the errors and others should not see the messages. You can redirect standard error in **bash** using the syntax **2>** ***file***. For example:

```
$ prog 2> errormessages
```

Creating a file handle

- To open a file and associate a file handle to it, use the Perl **`open()`** function. The function call

  ```
  open(FILEHANDLE, 'filename');
  ```

  attempts to open the file named **`filename`** in the current working directory for reading.
- If it is successful, it attaches the file handle named **`FILEHANDLE`** to this file and returns the number **`1`**. The program can then use **`FILEHANDLE`** to read from **`filename`**.

Responding to failed **`open()`** calls

- The **`open()`** function can fail for many reasons. The file may not exist; your program may not have permission to read it; the file may be opened for writing by another process and cannot be opened by any process at the moment.
- If the **`open()`** function fails, it returns **`undef`**. Your program should not attempt to read from the file if **`open()`** failed. It must check if **`open()`** failed, print an error message, and exit gracefully.

The `die()` function

- The `die()` function in Perl has the form

  ```
  die LIST;
  ```

- `die()` will print ***`LIST`*** to `STDERR` and exit the program (returning the error number or error string of the last error in a system call in a special variable named `$!` in Perl.)
- The ***`LIST`*** is usually just a string, as in the previous example. If the last element of the list has a newline as its last character, just the ***`LIST`*** is printed. If no newline is there, `die()` prints the line number and file number at which it exited.

Using the **die()** function

- Because **open()** returns **undef**, which is FALSE, if it fails, and **1**, which is TRUE, if it succeeds, in the expression:

```
open(FILE, 'filename')  or
          die 'failed to open filename';
```

  the **die()** function will be executed only if **open()** fails, because **or** is a lazy operator (as described in Lesson 15).
- Your program can print the error number from the failed **open()** as follows:

```
open(FILE, 'filename')  or die "failed to open",
     "filename: error number $!", "stopped";
```

Example

- The following opens the file named strings and searches for the string **`ctagcatgccag`** in it, line by line, printing out the lines that contain it.

```
open(DNA, "strings") or
        die "Could not open strings;
 stopped";
while ($line = <DNA> ) {
    if ( $line !~ /ctagcatgccag/ ) {
        next;
    }
    print $line;
}
```

Opening files for writing and more

- The **`open()`** function can open files for reading, for writing only, for reading and writing, and for appending. The syntax is:

```
open(FILE, "filename");  # opens for reading
open(FILE, ">filename"); # for writing only
open(FILE, "+<filename");# for reading and writing
open(FILE, ">>filename");# for appending
```

- ***Opening a file for writing erases any contents the file had before.*** Opening a file for reading and writing means that the program can read from the file and also write to it. Appending means adding to the end of the file.

The `close` Function

- ***A program that opens a file and associates a file handle to it, should close the file when it is finished with it.*** There are a few reasons for closing a file:
  - If the file is opened for writing, changes to the file will not take effect until it is closed.
  - If the file is opened for exclusive reading, no other processes will be able to access the file until your program finishes with it. To use files efficiently, programs should close them as soon as they are finished with them.
- To close the file associated with **`FILEHANDLE`**, use

```
close FILEHANDLE;
```

Reading using the input operator

- The input operator **`<>`** returns a single line from the input stream when it is used in a scalar context, as in

```
    while ( $line = <FILE>) { … }
```

- When it is used in an array context, as in

```
    @lines = <FILE>;
```

- the entire file is copied into the array variable, **`@lines`**, one line of the file per array entry. Thus, the first line is in **`$lines[0]`**, the second in **`$lines[1]`**, and so on.

Example of list context input

- This program reads all lines from file **`input`** at once, stores them in array variable **`@lines`**, and prints them out one by one, prefixing each with a line number.

```
open(FILE, "input") || die "Could not open\n";
my @lines = <FILE>;
my $count = 1;
foreach my $line ( @lines ) {
    print "$count:\t",$line;
    $count++;
}
```

Perl's predefined variables

- Perl has many predefined variables. Some of these have "***punctuation names***" that many people find hard to remember or just too strange to accept. For example, **`$_`** is a predefined variable.
- To appeal to those people who are unhappy with punctuation names, Perl has a module named **`English`** that can be included in a program with a **`use`** pragma:

```
use English;
```

- By including this module, a programmer has the additional choice of using English versions of these names, for instance **`$ARG`** instead of **`$_`**.

Some useful predefined variables

- In the rest of these slides, when I introduce a predefined variable, I will include its English name in parentheses.
- You already saw one predefined variable, `$!` (in English, `$ERRNO`), which is the value of the last error caused by a system call. It is a number in numeric context and a string in string context.
- You can use the value of `$!` only immediately after the system call, because its value may change soon after.

The default input/output variable **`$_`**

- **`$_`** (in English, **`$ARG`**) is the single, most important predefined variable. It is the default input and pattern matching variable. Also, for many functions, such as **`print`**, Perl will use this variable when you do not give the function an argument.
- The **`$_`** variable is automatically used in an input operation inside the condition of a while loop when no variable is supplied, as in

```
    while (<STDIN>) { … }
```

This is equivalent to

```
    while ($_ = <STDIN>) { … }
```

The **$_** variable for input and output

- Example:

```
        while (<FILE>) {
                print;
         }
```

  is equivalent to

```
        while (defined($_ = <FILE>)) {
                print $_;
         }
```

  because the input is placed in **$_** and **print** prints **$_** if no variable is supplied. **defined()** returns **undef** if there is no input.

$_ for pattern matching

- The **$_** variable is also used when you do not specify what to match against a pattern. For example, the following two statements are equivalent.

```
/^pattern/
$_ =~ /^pattern/
```

- because in the absence of a variable, the pattern is matched against **$_**. They both mean, "check if the string stored in **$_** is matched by the pattern **/^pattern/**" and have the value true if there is a match and false if there is not.

$_ in loops

- The following are equivalent:

```
 foreach ( @mylist ) {
     print;
 }
```

and

```
 foreach $_ ( @mylist ) {
     print $_;
 }
```

because `$_` is used if no iterator variable is used in the `foreach` loop.

$_ used in other functions

- If you write

  ```
  chomp;
  ```

  it is short for

  ```
  chomp($_);
  ```

- It is also used in many functions that expect a single argument, such as **`int()`** and others you have not seen yet, like **`abs()`**, **`sin()`**, **`cos()`** and other numeric functions.

The input record separator `$/`

- `$/` (`$INPUT_RECORD_SEPARATOR`) defines what Perl thinks of as a line in an input file. When you use a statement such as

  ```
  $var = <FILE>;
  ```

  to read a line from the file associated with `FILE`, Perl uses this variable to determine what you mean by a "line". The default is all characters up to and including the next newline character. These chunks of characters are called ***records***.
- You can change this by assigning a different string or character to `$/`.

Uses of `$/`

- If `$/` is set to the empty string, Perl will treat all sequences of characters up to the next empty line as a record. This is a way of storing paragraphs separated by blank lines into separate variables or cells of an array.
- If `$/` is set to `"\n\n"` Perl will treat two consecutive newline characters as the record separator, in case records are separated by double blank lines.
- If `$/` is a reference to an integer ***N*** or to a scalar containing an integer ***N***, Perl will treat each sequence of *N* characters as a line.

Example

- This reads the chunk size from the command line and then reads chunks of the given size.

```
my $size = shift( @ARGV );
$/ = \$size;
while ( <> ) {
    $count++;
    $chunk =  $_;
    print "$count\t$chunk\n";
}
```

Output field separator `$,`

- `$,` (`$OUTPUT_FIELD_SEPARATOR`) is what Perl uses to separate the arguments in the list given to the `print()` function:

```
print  5, 6, 7;
```

- will print **`567`** by default, i.e., `$,` is the empty string. You can make it any string, such as a blank, a comma, or whatever suits the application.

```
$, = ', ';
print 5, 6, 7;  # prints 5, 6, 7
```

The **@ARGV** array

- The **@ARGV** array is a special, predefined array in Perl. When a program is run from the command line, all of the arguments on the command line are stored in the **@ARGV** array. For example, if you enter

```
$ myprog file1 file2 file3
```

  then **@ARGV** contains the list `('file1','file2','file3')` when the program starts up. The first argument after the command name itself (`myprog`) is stored in `$ARGV[0]`.

Example of **@ARGV**

- This is a program that will display its command line arguments, one per line:

```
my ($count, $arg ) = (0);
foreach $arg ( @ARGV ) {
     $count++;
     print "Argument $count is $arg \n";
}
```

More about command line parsing

- The angle operator **`<>`** has special meaning when it is empty. When it is used in the condition of a **`while`** loop, each word contained in the **`@ARGV`** array is treated like the name of a file and opened for reading.
- Since the command line arguments are stored in **`@ARGV`**, **`<>`** in the **`while`** condition reads each file's contents in the order they are listed on the command line.
- The name of the current file is stored in the special variable **`$ARGV`** and the file handle to it is in the special file handle **`ARGV`**.

Using **$ARGV**

- This is a program that will display each command line argument, open it, and display its contents:

```
my $oldfilename = "";
while  ( <> ) {
    if ( $oldfilename ne  $ARGV ) {
        print "\n$ARGV:\n";
        $oldfilename = $ARGV;
    }
    print;
}
```

- It does no error checking; if a command line argument is not a readable file, the program will fail.

More About <>

- Another useful thing to remember about the **<>** operator is that it first tries to read the contents of the **@ARGV** array and use them as filenames. If a program assigns a list of filenames to **@ARGV**, then **<>** will use that list instead of the files on the command line.
- To demonstrate, this program always displays itself, ignoring the command line arguments:

```
@ARGV = ($0);  # $0 is the program pathname
while ( <> ) {
    print;
}
```

Regular expressions in Perl

- Pattern matching in Perl is pretty much an extension of the regular expression capabilities found in the UNIX shell.
- Fortunately, almost all of the syntax of regular expressions used in the various shell tools (as defined by the **`regex`** (7) man page) carries over into Perl.
- For example, the bracket operators **`[]`**, the quantifiers **`*`**, **`+`**, and **`?`**, the anchors **`^`** and **`$`**, the escape character **`\`**, and the wildcard **`.`** are all in Perl's regular expression language. The use of parentheses to remember matched substrings (***backreferences***) is also in Perl.

Help with Perl regular expressions

- The easiest way to get started with Perl's regular expressions is to read the man pages. The following Perl man pages provide help with regular expressions:

`perlrequick`	Basics of regular expressions
`perlretut`	Tutorial on regular expressions
`perlreref`	Quick reference for regular expressions
`perlre`	Reference page for regular expression syntax
`perlop`	Sections on regex operators such as **`m//`**

What is different in Perl

- Perl has many more capabilities than are present in the `regex` language of the shell. I will not cover all of these capabilities in this set of slides.
  - Perl has an alternation operator `|` that is like logical-or and is the same as the one in the extended regular expressions.
  - It lets you choose between ***greedy*** and ***non-greedy*** quantifiers.
  - It has many more character classes than the shell.
  - It has different ***word boundary anchors***
  - It has different rules for backreferences.

Pattern matching revisited

- Pattern matching is the act of checking whether a string is matched by a pattern. Remember that a regular expression is a compact notation that defines a possibly infinite set of strings.
- A ***regular expression matches a string*** (or the string is matched by the regular expression) if the string is in the set defined by the regular expression. Thus,

  "**`abracadabra`**" is matched by **`abra…abra`** and by **`(abra)…\1`**
- A pattern match is therefore a test: it is an expression that is true if the pattern matches the string and false if it does not.

The pattern matching operator

- The pattern matching operator in Perl is used where conditions are expected, such as in if statements and while statements.
- The pattern matching operator is

  `m/`***pattern***`/`

  where ***pattern*** is a regular expression. It is true if **`$_`** is matched by ***pattern*** and false if it is not. To check whether a different variable is matched by the pattern, you use the ***binding operator*** **`=~`** or its negation, **`!~`**, as in

```
if ( $var =~ m/pattern/ ) { … }
```

Other forms of pattern match

- The `m` can be omitted from the match operator `m//`. Instead, one can use

```
    if ( $var =~ /pattern/ ) { … }
```

- Or, you can keep the `m` and choose other paired delimiters such as `m(`*pattern*`)` or `m<`*pattern*`>`.
- As a third alternative, you can keep the `m` and use any single non-alphanumeric character repeated, as in `m|`*pattern*`|` or `m#`*pattern*`#`.
- I usually use the `/`*pattern*`/` form since it is the least typing. :-)

Example

- Here is a simple Perl program that combines many of the ideas introduced in this chapter.

```
my $pattern = shift(@ARGV);
while (<>) {
    if (/$pattern/) { # if $pattern matches line
        print;        # print line on output
    }
}
```

- The shift removes the first argument from the command line and stores it in `$pattern`. The loop then reads lines from each file left on the command line, checking if they match the pattern. This is a simple version of **`grep`**!

Character classes in Perl

- Perl has several constructs besides **`[]`** for defining character classes :

`\w`	word char, same as **`[a-zA-Z0-9_]`**
`\W`	non-word char
`\d`	digit, same as **`[0-9]`**
`\D`	non-digit
`\s`	white space, same as **`[\r\f\n\t ]`**
`\S`	non-white space char
`\b`	word boundary
`\B`	not a word boundary
`.`	any char except newline

More on boundaries

- When the beginning or end of line anchors `^` and `$` are used in a pattern, Perl interprets them as anchors to the beginning and end of a string. Thus,

  ```
      $var =~  /^\d+$/
  ```

  anchors `\d+` to the beginning and end of `$var`, so the expression is true if and only if `$var` consists exactly of one or more digits.
- The anchors `/A` and `/Z` are equivalent to `^` and `$`.
- Examples of word anchors follow in the next slide.

Word boundary examples

- Word and non-word boundaries examples:

`/fred\b/;`	matches **`fred`** and **`alfred`** but not **`freddy`**
`/\bfred/;`	matches **`fred`** and **`freddy`** but not **`alfred`**
`/\bfred\b/;`	matches only **`fred`** - neither **`freddy`** **`nor`** **`alfred`**
`/\bfred\B/;`	matches **`freddy`** but not **`fred`** or **`alfred`**

Quantifiers

- Recall that Perl has regular expression ***modifiers*** that alter the meaning of the regular expression to their left. They are called ***quantifiers*** because they change the quantity required in the string.

`*`	match ***0*** or more occurrences
`+`	match ***1*** or more occurrences
`?`	match ***0*** or ***1*** occurrence
`{n}`	match exactly ***n*** occurrences
`{n,}`	match at least ***n*** occurrences
`{n,m}`	match at least ***n*** but not more than ***m*** occurrences

Quantifier examples

- In the following table, `''` denotes the empty string.

Expression	Matched by:
`(ab)*`	`''`, **ab**, **abab**, **ababab**, **abababab**, …
`(ab)+`	**ab**, **abab**, **ababab**, **abababab**, …
`(ab)?`	`''` and **ab**
`[abc]{2}`	**aa**, **ab**, **ac**, **ba**, **bb**, **bc**, **ca**, **cb**, and **cc**
`b{3,}`	**bbb**, **bbbb**, **bbbbb**, **bbbbbb**, …
`b{2,3}`	**bb** and **bbb**

Alternation

- The alternation metacharacter **`|`** is like an "or" -- the pattern **`a | b`** matches either **`a`** or **`b`**.
- At first this does not seem like it adds much to the language, since you could write **`[ab]`** and it would mean the same thing, if **`a`** and **`b`** were single characters each. But the arguments to **`|`** can be arbitrary patterns or strings, as in

  ```
  /Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec/
  ```

  which matches any three letter month name.

Captured matches

- The construct (***re***) creates a ***capture buffer*** in Perl. Perl saves the sub-string that matched the ***re*** in this buffer. The program can refer to this buffer using the notation **`\1`**,**`\2`**,**`\3`**, and so on. The first captured sub-string is in **`\1`**, the second in **`\2`**, and so on.
- Thus,

```
/([acgt])([acgt])([acgt])[acgt]*\3\2\1/
```

  matches any string that starts with three nucleotides and ends with the same nucleotides in reverse order, such as **`acgtttagca`**.

More about captured matches

- Inside a pattern **`\1`**, **`\2`**, **`\3`**, ... refer to the first, second, thirds matched sub-strings
- Outside of the pattern, the variables **`$1`**, **`$2`**, **`$3`**, refer to the same buffers as **`\1`**, **`\2`**, and so on, so that these captured strings can be accessed in other Perl statements. E.g.,

```
   /href[ ]*="([^"]*)"/ and print "$1\n";
```

- This matches lines containing a webpage link of the form

```
   href = "url"
```

  captures the url in **`$1`** and prints **`$1`**.

Capture buffer examples

- A few interesting things are going on in the code below.

```
 $month =    "Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|
Oct|Nov|Dec";
  while (<STDIN>) {
      print $1,"\n" if  /\b($month)\b/;
  }
```

- First, you can store a pattern in a variable and use the variable name in the capturing brackets. Second, the backreference, **`$1`**, textually precedes the pattern, but it still refers to the match because the **`print`** statement is evaluated after the pattern. This prints the month name if it is found on the line.

Position of Matches

- There are three useful special variables: after a matching operation, Perl sets **`` $` ``** to the part of the string ***before*** the match, sets **`$&`** to the part of the string that ***matched***, and sets **`$'`** to the part of the string ***after*** the match:

```
$x = "the lion ate the lamb";
$x =~ / ate /;
# $` = 'the lion', $& = ' ate ' # $' = 'the lamb'
$x =~ /the/;
# $` = '', $& = 'the', $' = ' ate the lamb'
```

- In the second match, **`` $` ``** equals **`''`** because the regexp matched at the first character position in the string and stopped; it never saw the second **`'the'`**.

Substitution

- The substitution operator has the form

  `s/`*target-pattern***`/`***replacement-text***`/`**

- This searches a string for the target pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (really, the empty string, which is false in a logical context).
- To perform a substitution in a string in a variable, use

  `$var =~ s/`*pattern***`/`***replacement***`/;`**

- If no variable is specified, it uses **`$_`** by default.

Examples

- This is a simple example, for starters.

```
my $string = 'Barney said, "Yabadabadoo."';
$string =~ s/Y.*a/Fred will /;
print "$string\n";
# prints: Barney said, "Fred will doo."
```

- The above example also illustrates that Perl chooses the longest matching string when matching. It could have matched **`Ya`**, **`Yaba`**, **`Yabada`**, or **`Yabadaba`**. It chose the longest.

Another example

⌗ The ability to capture substrings makes substitution powerful. In this example, a substitution occurs only if the string contains other text matching a pattern, but that text is preserved.

```
while (<>) {
     s/(.*:.*:.*:300:.*)csh/$1bash/;
     print;
}
```

⌗ If we pass the contents of the passwd file to this script, it will replace the word **csh** by **bash** in each line in which the group field is **300**. It does not change the original file -- it only prints a changed copy on the console.

Global Replacement

- The substitution operator takes optional modifiers, the most useful of which is the 'g' (global) modifier. The syntax is

  `s/`*pattern*`/`*replacement-text*`/g`

- With the **`/g`** modifier, every occurrence of the pattern in the string is replaced by the replacement text. Without it, only the first occurrence is replaced. For example:

```
my $string = 'Barney said, "Yabadabadoo."';
$string =~ s/aba/aka/g;
print "$string\n";
# prints: Barney said, "Yakadakadoo."
```

The `join()` function

- Splitting was introduced in an earlier lesson without mentioning its mate, **`join()`**, which is like the inverse of **`split()`** -- it glues the components of an array into a string.

  `join(` ***gluestring*** **`,`** ***list*** **`)`**

  joins the separate strings of ***list*** into a single string with fields connected with the ***gluestring***, and returns that new string.

- For example:

```
$string = join('-', 0,1,2,3,4,5,6,7,8,9);
print $string;
```

  prints

```
0-1-2-3-4-5-6-7-8-9
```

Another example

- This is another example of joining things:

```
my $rhyme = 'Humpty Dumpty sat on a wall';
my @Humpty = split($rhyme);
# @Humpty is a list of words now
my $Dumpty = join(':',@Humpty);
# $Dumpty is @Humpty together again
print $Dumpty;
# prints Humpty:Dumpty:sat:on:a:wall
```

The `substr()` Function

- The **`substr()`** function is very useful.

```
    substr( $mystr,$offset, $length)
```

  returns the substring of **`$mystr`** starting at offset **`$offset`**, of length **`$length`**. If length is omitted, it returns everything to the end of the string. If length is negative, it leaves that many characters off the end of the string:

```
    $str = “The black cat climbed the tree";
    substr($str,4,5)  # “black”
    substr($str,4) # “black cat climbed the tree”
    substr($str,4,-9) # “black cat climbed”
```

The **DATA** Filehandle

- In addition to **STDIN**, **STDOUT**, and **STDERR**, Perl predefines the **DATA** file handle. A program does not have to open this handle; it is open automatically, and used in conjunction with the literal **__DATA__** (two underscores on each side, not one.) The program can use **__END__** instead of **__DATA__**; there is no difference.
- The token **__DATA__** is placed at the end of the file containing the program, and the data that the **DATA** filehandle reads are all the lines after the **__DATA__** token and before the real end of the file. ***The program must close the* DATA *handle when it is finished reading***.

For What Purpose?

- Sometimes programs can only run when they are accompanied by datasets such as lookup tables. If these tables are in separate files, they can get separated from the program when users install them. Putting the data in the program file itself prevents this.
- When you are working on a program and need to test it, instead of having to open and edit test files, add **`open()`** statements, and go back and forth between program and test file, you can read from the **`DATA`** filehandle and just put the tests after the **`__DATA__`** token. This makes testing easier.

Example

- This simple example just prints out the data following the **__DATA__** token. It is like an executable text file that displays itself.

```
while (<DATA>) {
    print;
}
close DATA;
# the data is here
__DATA__
LET us go then, you and I,
When the evening is spread out against the sky
Like a patient etherised upon a table;
```

DATA handle example

- This stores constant type data in a program instead of an external file:

```
while (<DATA>) {
    /([^:]*):.*:(.*)/;
    $category{$1}= $2;
}
close DATA;

# the data is here
__DATA__
Ser:Serine:S:polar
Phe:Phenylalanine:F:non-polar
His:Histidine:H:basic
Asp:Aspartate:D:acidic
...
```

Putting things together

- This chapter added several new capabilities to your Perl toolkit. The logical next step is to integrate these new tools into a few short programs.
- The demo directory for this chapter has a few programs that integrate these new tools and concepts. You should study them and tinker with them.

Perl Documentation

- There is a wealth of documentation and help for using Perl on a UNIX system. The default Perl installation comes with a collection of over one hundred man pages on various topics.
- The root page is simply named **`perl`**. Typing **`man perl`** will display a table of contents of the other man pages. If you want to read about predefined variables, read **`perlvar`**. The page about Perl's built-in functions is **`perlfunc`**. The page with its operators is **`perlop`**, and so on.
- Perl also has 10 **`perlfaq`** man pages (frequently asked questions), named **`perlfaq`**, **`perlfaq1`**, …, **`perlfaq9`**.

Perl Tutorials

- Among the man pages are a few tutorials. They are:

perlreftut	Perl references short introduction
perldsc	Perl data structures intro
perllol	Perl data structures: arrays of arrays
perlrequick	Perl regular expressions quick start
perlretut	Perl regular expressions tutorial
perlboot	Perl OO tutorial for beginners
perltoot	Perl OO tutorial, part 1
perltooc	Perl OO tutorial, part 2
perlbot	Perl OO tricks and examples
perltrap	Perl traps for the unwary
perldebtut	Perl debugging tutorial

Summary

- A byte stream is a conceptualization of any I/O, whether from files, devices, or other programs.
- Filehandles are names that are associated with byte streams, and are used in I/O statements to connect the program to those streams.
- Perl has 4 predefined filehandles: `STDIN`, `STDOUT`, `STDERR`, and `DATA`.
- `open()` binds data streams to file handles; `close()` breaks the binding.
- `<>` is the input operator in Perl.
- `die()` allows a program to exit and send output to `STDERR`.
- Perl has several predefined variables such as `$_`, `@ARGV`, `$,` and `$\`.

Summary continued

- `@ARGV` is the array that stores the command line arguments.
- The `m//` operator is the pattern match operator in Perl.
- The `s///` operator is the substitution operator.
- Perl has a rich set of regular expression characters and metacharacters that includes all of the characters from the shell's regex language.
- Perl also has alternation and a more powerful set of metacharacters than the shell.
- Perl also has `split()` and `join()` functions for working with text data and lists.