Performing Substitutions

- The s/.../.../ form can be used to make substitutions in the specified string. Note that if paired delimiters are used, then you have to use two pairs of the delimiters. 'g' after the last delimiter indicates to replace more than just the first occurrence. The substitution can be bound to a string using "=~". Otherwise it makes the substitutions in $_. The operation returns the number of replacements performed, which can be more than one with the 'g' option.

```
# "search" is replaced with "replace" for all
# occurrences in $s and the number of replacements
# is assigned to $var
$var = $s =~ s/search/replace/g;
```

Substitution Examples

```
s/\bfigure (\d+)/Figure $1/
                        # capitalize references to
                        # figures
s{//(.*)}{/\*$1\*/}     # use old style C comments
s!\bif(!if (!           # put a blank between "if"
                        # and "("
s(!)(.)                 # tone down that message
s[!][.]g                # replace all occurrences
                        # of '!' with '.'
```

Case Shifting

- In the replacement string, you can force what follows a given point in the replacement string to be upper or lower case by using the \U or \L indicators, respectively.

```
# change acm or ieee to uppercase within $text
$text =~ s/(acm|ieee)/\U$1/;

# change course prefix to lowercase in $text and
# assign to $num the number of replacements made
$num = $text =~ s/\b(COP|CDA)\d+/\L$&/g;
```

Performing Translations

- In Perl you can also convert one set of characters to another using the tr/.../.../ form. However, rather than specifying a pattern, you specify two strings in a manner similar to the *tr* Unix utility. Any character found that is in the first string is replaced with the corresponding character in the second string. It returns the number of characters replaced or deleted. If the replacement string is empty, then the search string is used by default and there is no effect on the string being searched. If there are fewer replacement characters, then the final one is replicated. The *d* modifier deletes characters not given a replacement. The *s* modifier squashes duplicate replaced characters.

Translation Examples

```
# convert letters in $text to lowercase
$text =~ tr/A-Z/a-z/;

# count the digits in $_ and assign to $cnt
$cnt = tr/0-9//;

# get rid of redundant blanks in $_
tr/ //s;

# delete *'s in $text
$text =~ tr/\*//d;

# replace [ and { with ( in $text
$text =~ tr/[{/(/;
```

Split Operator

- The *split* operator breaks up a string according to a specified separator pattern and generates a list of the substrings. Leading empty fields become null strings, but trailing empty fields are discarded.
- General form.

  split /<separator pattern>/, <string>
- Example:

```
$line = “This sentence contains five words.”;
@fields = split / /,$line; # @fields = (“This”, “sentence”,
                           # “contains”, “five”, “words.”);
```

Join Function

- The *join* function has the opposite effect of the split operator. It takes a list of strings and concatenates them together into a single string. The first argument is a glue string and the remaining arguments are a list and it returns a string containing the remaining arguments separated by the glue string.

```
@fields = (“This”, “sentence”, “contains”, “5”, “words.”);

# The statement below has the following effect:
# $line = “This sentence contains 5 words.”;
$line = join “ ”, @fields;
```

Filehandles

- A filehandle is an I/O connection between your process and some device or file.
- Perl has three predefined filehandles.

  STDIN – standard input

  STDOUT – standard output

  STDERR – standard error

Opening Filehandles

- You can open your own filehandle. Unlike other variables, filehandles are not declared with the *my* operator. The convention is to use all uppercase letters when referring to a filehandle.
- The *open* operator takes two arguments, a filehandle name and a connection (e.g. filename). The connection can start with "<", ">", or ">>" to indicate read, write, and append access.

```
open IN, "in.dat";      # open "in.dat" for input
open IN2, "<$file";     # open filename in $file for input
open OUT, ">out.dat";   # open "out.dat" for output
open LOG, ">>log.txt";  # open "log.txt" to append output
```

Closing Filehandles

- The *close* operator closes a filehandle. This causes any remaining output data associated with this filehandle to be flushed to the file. Perl automatically closes a filehandle if you reopen it or if you exit the program.

```
close IN;     # closes the IN filehandle
close OUT;    # closes the OUT filehandle
close LOG;    # closes the LOG filehandle
```

Exiting the Process

- You can exit a process by using the exit function. It takes an argument that indicates the exit status.

```
exit 0;    # everything is fine
exit 1;    # something went wrong
```

- You can also exit a process by using the die function, which in addition prints a message to STDERR. Perl also automatically appends the name of the program and the current line to the message.

```
die "Something went wrong.";
```

Checking the Status of Open

- You can check the status of opening a file by examining the result of the open operation. It returns true for a successful open and false for failure.

```
if (!open OUT, ">out.dat") {
    die "Could not open out.dat.";
}
```

Using Filehandles

- After opening a filehandle, you can use it to read or write depending on how you opened it. Note that in a print or printf statement, the filehandle name is not followed by a comma.

```
open IN, “<in.dat”;
open OUT, “>out.dat”;
$i = 1;
while ($line = <IN>) {
    printf OUT “%d: $line”, $i;
}
```

Reopening a Standard Filename

- You can reopen a standard filename. This feature allows you to not only perform input or output in a normal fashion, but to also redirect the I/O from/to a file within the Perl program.

```
# redirect standard output to “out.txt”
open STDOUT, “>out.txt”;
printf “Hello world!\n”;

# redirect standard error to append to
“log.txt”
open STDERR, “>>log.txt”;
```

Reopening a Standard Filename

- You can reopen a standard filename. This feature allows you to not only perform input or output in a normal fashion, but to also redirect the I/O from/to a file within the Perl program.

```
# redirect standard output to “out.txt”
open STDOUT, “>out.txt”;
printf “Hello world!\n”;

# redirect standard error to append to “log.txt”
open STDERR, “>>log.txt”;
```

Checking the Status of a File

- You can check the status of a file by performing a file test. Each file test returns a boolean value that can be referenced in control structures. These file tests are similar to those available in the shell.
- General form.

  –option filename

- Some common options are:

-r (file is readable), -w (file is writeable), -x (file is executable), -e (file exists), -f (is a plain file), -d (is a directory)

Example of Checking the Status of a File

```
# open the file
if (! open IN, $filename) {

   # print reason why file could not be opened
   if (! -e $filename) {
      die “$filename does not exist.”;
   }
   if (! -r $filename) {
      die “$filename is not readable.”;
   }
   if (-d $filename) {
      die “$filename should not be a directory.”;
   }
   die “$filename could not be opened.”;
}
```

Defining Subroutines in Perl

- Perl also supports subroutines (i.e. functions). Declarations of subroutines can be placed anywhere.
- General form.

```
sub <name> {
     <one_or_more_statements>
}
```

- Example

```
sub read_fields {
    $line = <>;
    @fields = split / /,  $line;
}
```

Invoking Subroutines

- You invoke a subroutine by preceding the name with an '&' character.

```
&read_fields;
printf OUT “The number of fields is %d.\n”,
       $#fields+1;
```

Return Values

- A Perl subroutine can return a value.
  - The result of the last calculation performed is the return value.
  - A value can be explicitly returned with the return statement.

```
sub maxsize {
    1000;
}

sub nextval {
    $val++;
    return $val;
}
```

Subroutine Arguments

- To pass arguments to a subroutine, simply put a list expression in parentheses after the name of the subroutine when you invoke it.
- The arguments are received in the array @_. You can determine the number of arguments by examining $#_. You can access individual arguments using $_[0], $_[1], etc. So this feature allows routines to be easily written to handle a variable number of arguments.

Example Use of Arguments

```
sub max {
   my $maxnum = shift @_;  # shift the first arg off
   foreach $val (@_) {     # for each remaining arg
      if ($val > $maxnum) {
         $maxnum = $val;
      }
   }
   return $maxnum;
}

$num = &max($a, $b, $c);   # can pass one or more args
```

Variables within Subroutines

- Variables declared with the my operator are only visible within that block. Thus, variables declared within a subroutine are only visible within that subroutine. This means that the declaration of a variable within a block will prevent access to a variable with the same name outside the block.

Sort Subroutines

- The sort operator can accept the name of a subroutine to determine the order between a pair of elements. Rather than receiving arguments in @_, the sort subroutine instead receives arguments in $a and $b. It returns a -1 if $a should appear before $b, a 1 if $b should appear before $a, and 0 if the order does not matter.

Sorting Operators

- The <=> operator returns -1, 0, or 1 depending on the numerical relationship between the two operands.

```
sub by_number {
    return $a <=> $b;
}
```

- The cmp operator returns -1, 0, or 1 depending on the string comparison relationship between the two operands. By default the sort operator does this type of comparison.

```
sub by_ASCII {
    return $a cmp $b;
}
```

Using Sorting Functions

```
@vals = sort by_number @vals;      # sort @vals numerically
@s = sort by_ASCII @fields;        # sort @fields in ASCII order
@vals = sort {$a <=> $b} @vals;    # numerical sort specified
                                   # inline
@s = sort {$a cmp $b} @fields;     # ASCII sort specified inline
@k = sort {$n{$a} cmp $n{$b}} keys %n;
                                   # returns list of keys based
                                   # on ASCII sort of hash values
```

Picking Items from a List with Grep

- The *grep* operator extracts items from a list. The first argument is a function that returns true or false. The remaining arguments are the list of items. The *grep* operator returns a list. The function uses $_ to access each item in the list. This is really a shortcut to avoid using a foreach statement.

```
foreach @vals {          # extract the odd values
   if ($_ % 2) {
      push @oddvals, $_;
   }
}

# extract the odd values in a single line of code
@oddvals = grep { $_ % 2 } @vals;
```

Transforming Items with a Map

- The *map* operator is similar to grep, except that the value returned from the function is always added to the resulting list returned by *map*.

```
# take the absolute values of each element
foreach (@vals) {
      if ($_ < 0)
          push @absvals, -$_;
      else
          push @absvals, $_;
}

# take the absolute values in a single line of code
@absvals = map { $_ < 0 ? -$_ : $_ } @vals;
```

Directory Operations

- Perl provides directory operations that are portable across different operating systems. The general form and an example of each given below.

```
chdir dirname;          # cd to a new directory
chdir “asg1”;           # cd to subdirectory “asg1”

glob filename_pattern; # return list of filenames
glob “*.c”             # return list of *.c filenames
```

Manipulating Files and Directories

- Perl functions exist to change files and directories. The general form and an example of each are given below.

```
unlink filenames;              # remove list of files
unlink in.dat, out.dat;        # remove two files

rename oldfile, newfile;       # rename a file
rename “tmp.out”, “data.out”; # renamed an output file

mkdir dirname, permissions;    # make a new directory
mkdir “asg1”, 0700;            # mkdir asg1 where only
                               # the user can access it

rmdir dirnames;                # remove list of directories
rmdir “asg1”;                  # remove asg1 directory
```

Manipulating Files and Directories (cont.)

```
chmod perms, filenames; # change permissions
chmod 0755, “asg1”      # change permissions on asg1
```

Invoking Processes

- Can use the system command to create a child process.

```
system “date”;     # invokes the Unix date command
```

- Can use backquotes to capture output.

```
$time = `date`;    # capture Unix date command output
```