cs3157: unix lecture (mon-21-mar-2005)

- today:
  - C programs with multiple files
  - regular expressions
  - filters (cut, grep, sed, sort, wc)
  - sh scripts

C programs with multiple files.

- sometimes it is more convenient to write a program in more than one source file
- this is especially where header files (`.h`) come in handy
- I divided the `struct4.c` example from last class into 3 files:
  - `person.h`: which contains the constant (`#define`) and data type (`typedef`) definitions and function prototypes for functions that operate on the data structures defined in this header file
  - `person.c`: which contains the function definitions for the functions that operate on the data structures defined in `person.h`;
  - `pmain.c`: which contains the main program
- note that `person.c` and `person.h` go together!
- the program is built using the compiler (`gcc -c`) and linker (`gcc`) separately:

```
gcc -c person.c -o person.o
gcc -c pmain.c -o pmain.o
gcc person.o pmain.o -o pmain
```


regular expressions

- describe patterns or sequences of characters
- aka *pattern matching*
- *expression* ⇐ it gets evaluated
- basic operator is *concatenation*
- example:

  ABC

  matches A followed by B followed by C
- case sensitive!
  i.e., does not match a followed by b followed by c
- programs that use regular expressions *evaluate* them first, then seek matches
  - examples: perl, grep, sed, awk

regular expressions – pattern matching

- example: match the pattern
  `your`
  with the input line:
  Do you like my hat?
- 1. compare y with *D*
- 2. compare y with *o*
- 3. compare y with □ (space)
- 4. compare y with *y* — aha!
- 5. compare o with *o* — aha!
- 6. compare u with *u* — aha!
- 7. compare r with □ (space) — bummer :-(

regular expressions – basic format

- *character sets*: matches one or more characters in a single position
- *modifiers*: specifies how many times the previous character set is repeated
- *anchors*: specifies the position of a pattern in relation to a line of text
- use backslash (\) to match special characters

regular expressions – meta-characters

- *meta-characters* provide the real power to regular expressions
  "i never meta-character i didn't like..."
  - dot (`.`) matches any single character
  - asterisk (`*`) matches zero or more occurrences of the *preceding* regular expression (NOT the same as * wildcard in shells!!)
  - plus (+) matches any single character one or more times

regular expressions – anchors

- *anchors*: specifies the position of a pattern in relation to a line of text
- caret (`^`) means *beginning of the line* (but only if used as the first character in the regular expression)
- dollar sign (`$`) means *end of the line* (but only if used as the last character in the regular expression)

regular expressions – ranges

- use square brackets (`[]`) to specify a range of characters, e.g.:
  - `[abc]`
  - `^[abc]`
  - `^[abc]$`
- use hyphen (`-`) inside the square brackets to include all characters between starting and ending characters, e.g.:
  - `[a-c]`
  - `[0-9]`
  - `^[a-z]`
  - `^[a-z]$`
- use carat (`^`) *inside* the square brackets to match characters EXCEPT those specified, e.g.:
  - `[^abc]`
  - `[^0-9]`

meta-character examples

what matches the following?

```
AE
A.E
A*E
AE*
A.*E
A*.E
A.E*
A*E.
```


input:

```
AE
ABE
ABBE
A
E
B
```

solution:

```
AE   matches: AE             and not: ABE ABBE A E B
A.E  matches: ABE            and not: AE ABBE A E B
A*E  matches: AE ABE ABBE E  and not: A B
AE*  matches: AE ABE ABBE A  and not:
A.*E matches: AE ABE ABBE    and not: A E B
A*.E matches: AE ABE ABBE    and not: A E B
A.E* matches: AE ABE ABBE    and not: A E B
A*E. matches:                and not: AE ABE ABBE A E B
```


anchor examples

what matches the following?

```
^A
A$
^A*
A*$
A.$
```


input:

```
AE
ABE
ABBE
A
E
B
```

solution:

```
^A  matches: AE ABE ABBE A      and not: E B
A$  matches: A                  and not: AE ABE ABBE E B
^A* matches: AE ABE ABBE A E B  and not:
A*$ matches: AE ABE ABBE A E B  and not:
A.$ matches: AE                 and not: ABE ABBE A E B
```

examples – you try it

- what is the regular expression to match all lines that start with a vowel?
- what is the regular expression to match all lines that end with a digit?
- what is the regular expression to match all lines that do not contain an underscore (_)?

fi lters.

- wc – counts characters, words and lines in input
- grep – matches regular expression patterns in input
- cut – extracts portions of each line from input
- sort – sorts lines of input
- sed – stream edits input

wc

- unix command: counts the number of characters/words/lines in its input
- input can be a file or a piped command (see below)
- example:
  filename = "hello.dat"

```
hello
world
```

usage:

```
unix-prompt$ wc hello.dat
       2       2      12 hello.dat
unix-prompt$ wc -l hello.dat
       2 hello.dat
unix-prompt$ wc -c hello.dat
      12 hello.dat
unix-prompt$ wc -w hello.dat
       2 hello.dat
```


grep (1)

- Global Regular Expression Parser
- one of the most useful tools in unix
- three standard versions:
  - plain old *grep*
  - extended grep: *egrep*
  - fast grep: *fgrep*
- used to search through files for ... regular expressions!
- prints only lines that match given pattern
- a kind of *filter*
- BUT it's *line oriented*

grep (2)

- input can be one or more files or can be *piped* into grep
- examples:
```
grep "^[aeiou]" myfile
ls -1 | grep t
```
- useful options:
  - `-i` —- ignore case
  - `-w` —- match pattern as a word
  - `-l` —- return only the filename if there's a match
  - `-v` —- reverse the normal action (i.e., return what doesn't match)

grep (3)

- examples:
```
grep -i "^[aeiou]" myfile
grep -v "^[aeiou]" myfile
grep -iv "^[aeiou]" myfile
```
- how do you list all lines containing a digit?
- how do you list all lines containing a `5`?
- how do you list all lines containing a `0`?
- how do you list all lines containing `50`?
- how do you list all lines containing a `5` and an `0`?

cut

- unix command: extracts portions of each line from input
- input can be a file or a piped command (see below)
- syntax: `cut <-c|f> <-d>`
  note that `c` and +f+ start with 1; default delimiter is TAB
- example:
  filename = "snowy"

```
There was movement at the station, for the word had passed around
That the colt from old Regret had got away
And had joined the wild bush horses -- he was worth a thousand pour
So all the cracks had gathered to the fray.
All the tried and noted riders from the station near and far
Had mustered at the homestead overnight,
For the bushmen love hard riding where the wild bush horses are,
And the stock-horse snuffs the battle with delight.
```

  usage:


```
unix-prompt$ cut -c1 snowy
T
T
A
...
unix-prompt$ cut -f1 -d' ' snowy
There
That
And
...
```

sort

- unix command: sorts lines of input
- input can be a file or a piped command (see below)
- three modes: sort, check (`sort -c`), merge (`sort -m`)
- syntax: `sort <-t> <-n> <-r> <-o>` POS1 -POS2+
  note that `POS` starts with 0; default delimiter is whitespace
- example:

```
unix-prompt$ sort snowy
All the tried and noted riders from the station near and far
And had joined the wild bush horses -- he was worth a thousand pour
And the stock-horse snuffs the battle with delight.
...
unix-prompt$ sort +2 -3 snowy
Had mustered at the homestead overnight,
For the bushmen love hard riding where the wild bush horses are,
That the colt from old Regret had got away
```


sed (1)

- *stream editor*
- does not change the file it "edits"
- commands are implicitly global
- input can be a file or can be *piped* into sed
- example: substitute all `A` for `B`:

```
sed 's/A/B/' myfile
cat myfile | sed 's/A/B/'
```

- use the `-e` option to specify more than one command at a time:

```
sed -e 's/A/B/' -e 's/C/D/' myfile
```

- pipe output to a file in order to save it:

```
sed -e 's/A/B/' -e 's/C/D/' myfile >mynewfile
```


sed (2)

- sed can specify an *address* of the line(s) to affect
- if no address is specified, then all lines are affected
- if there is one address, then any line matching the address is affected
- if there are two (comma separated) addresses, then all lines between the two addresses are affected
- if an exclamation mark (`!`) follows the address, then all lines that DON'T match the address are affected
- addresses are used in conjunction with commands
- examples (using the delete (`d`) command):

```
sed '$d' myfile
sed '/^$/d' myfile
sed '1,/under/d' myfile
sed '/over/,/under/d' myfile
```


sed (3)

- order of commands is important
- input is *line* oriented
- all editing commands are applied to each line, one at a time
- then next line is read and editing commands are applied to that linei
- etc
- for example:

```
sed -e 's/pig/cow/' -e 's/cow/horse' myfile
```

what does this do?
is this right???

sed (4)

- delimiter is slash (/)
- backslash (escape) it if it appears in the command, e.g.:
```
sed 's/\/usr\/bin\//\/usr\/etc/' myfile
```


sed (5)

- meta-character ampersand (&) represents the extent of the pattern matched
- example:
```
sed 's/[0-9]/#&/' myfile
```
what does this do?
- you can also save portions of the matched pattern:
```
sed 's/\([0-9]\)/#\1/' myfile
sed 's/\([0-9]\)\([0-9]\)/#\1-\2/' myfile
```


sed (6)

- transformation command: y
- example:
```
sed 'y/ABC/abc' myfile
```


sed (7)

- print command: p
- example:
```
sed '/begin/,/end/p' myfile
sed -n '/begin/,/end/p' myfile
```

sed (8)

- examples: what do the following sed commands do?
```
sed 's/xx/yy' myfile
sed '/BSD/d' myfile
sed '/^BEGIN/,/^END/p@' myfile
```
- how do you change the content of all your html files to lowercase?
- how do you change all the html commands to lowercase?

sh (1)

- `sh` is the "Bourne shell", the first scripting language
- it is a program that interprets your command lines and runs other programs
- it can invoke Unix commands and also has its own set of commands
- example:

```
while ( 1 ) {
  print prompt and wait for user to enter input;
  read input from terminal;
  parse into words;
  substitute variables;
  execute commands (execv or builtin);
}
```


sh (2)

- shell commands can be read:
  - from a terminal ⇒ *interactive*
  - from a file ⇒ *shell script*
- search path
  - the place where the shell looks for the commands it runs
  - should include standard directories:
    - `/bin`
    - `/usr/bin`
  - it should also include your current working directory ()

sh (3)

- are you runnning the Bourne shell?
  - type:
    `unix-prompt# echo $SHELL`
  - if the answer is `/bin/sh`, then you are
  - if the answer is `/bin/bash`, then that's close enough
  - otherwise, you can start the Bourne shell by typing `sh` at the UNIX prompt
  - enter `Ctrl-D` or `exit` to exit the Bourne shell and go back to whatever shell you were running before...

sh (4)

- capable of both synchronous and asynchronous execution
  - synchronous: wait for completion
  - asychronous: in parallel with shell (runs in the background)
- allows control of `stdin, stdout, stderr`
- enables environment setting for processes (using inheritance between processes)
- sets default directory

sh (5)

- creating your own shell scripts
- naming:
  - DON'T ever name your script (or any executable file) "test"
  - since that's a `sh` command
- executing
  - the notation `#!` inside your file tells UNIX which shell should execute the commands in your file
- example — create a file called "myscript.sh"

```
#!/bin/sh
echo hello world
```

- make the script executable: `unix-prompt# chmod +x myscript.sh`
- execute the script: `unix-prompt# ./myscript.sh` or just `unix-prompt# myscript.sh`

(note that `unix-prompt#` *means the unix prompt, like* `unix$` *or* `bash#`*)*

sh (6) —quoting

- quote (`'`)
  `'something'`: preserve everything literally and don't evaluate anything that is inside the quotes
- double quote (`"`)
  `"something"`: preserve most things literally, but also allow `$` variable expansion (but not `'` evaluation)
- backquote (`` ` ``)
  `` `something` ``: try to execute `something` as a command

sh (7) —quoting example

- filename=t.sh

```
#!/bin/sh
hello="hi"
echo 0=$hello
echo 1='$hello'
echo 2="$hello"
echo 3=`$hello`
echo 4="'$hello`"
echo 5="'$hello'"
```

- filename=hi

```
#!/bin/sh
echo "how did you get in here?"
```

- output=

```
unix$ t.sh
0=hi
1=$hello
2=hi
3=how did you get in here?
4=how did you get in here?
5='hi'
```

sh (8) —comments

- single line comments only (no multi-line comments)
- line begins with # character

sh (9) —simple commands

- sequence of words
- first word defines command
- can be combined with `&&`, `||`, `;`
  - to execute commands sequentially:
    `cmd1; cmd2;`
  - to execute a command in the background :
    `cmd1&`
  - to execute two commands asynchronously:
    `cmd1&`
    `cmd2&`
  - to execute `cmd2` if `cmd1` has zero exit status:
    `cmd1 && cmd2`
  - to execute `cmd2` only if `cmd1` has non-zero exit status:
    `cmd1 || cmd2`
- set exit status using `exit` command (e.g., `exit 0` or `exit 1`)

sh (10) —pipes

- sequence of commands
- connected with `|`
- each command reads previous command's output and takes it as input
- example:

```
unix-prompt# echo "hello world" | wc -w
2
```


sh (11) —shell variables

- variables are placeholders for values
- shell does variable substitution
- `$var` or `${var}` is the value of the variable
- assignment:
  - `var=value` (with no spaces before or after!)
  - `let "var = value"`
  - `export var=value`
- BUT values go away when shell is done executing
- uninitialized variables have no value
- variables are untyped, interpreted based on context
- standard shell variables:
  - `${N}` = shell Nth parameter
  - `$$` = process ID
  - `$?` = exit status

sh (12) —shell variables example

- filename=u.sh

```
#!/bin/sh
echo 0=$0
echo 1=$1
echo 2=$2
echo 3=$$
echo 4=$?
```

- output

```
unix$ u.sh
0=.//u.sh
1=
2=
3=21093
4=0

unix$ u.sh abc 23
0=.//u.sh
1=abc
2=23
3=21094
4=0
```


sh (13) —environment variables

- shell variables are generally not visible to programs
- environment variables are a list of name/value pairs passed to sub-processes
- all environment variables are also shell variables,
  but not vice versa
- show with `env` or `echo $var`
- standard environment variables include:
  - HOME = home directory
  - PATH = list of directories to search
  - TERM = type of terminal (vt100, ...)
  - TZ = timezone (e.g., US/Eastern)
- example:

```
unix-prompt# echo $TERM
vt100
```


sh (14) —looping constructs

- similar to C/Java constructs, but with commands
- `until test-commands; do consequent-commands; done`
- `while test-commands; do consequent-commands; done`
- `for name [in words ...]; do commands; done`
- also on separate lines
- `break` and `continue` control loop

sh (15) —loop examples

- while

```
i=0
while [ $i -lt 10 ]; do
  echo "i=$i"
  ((i=$i+1))   # same as let "i=$i+1"
done
```

- for

```
for counter in ‘ls *.c‘; do
  echo $counter
done
```

sh (16) —if

- syntax

```
if test-commands; then
        consequent-commands;
        [elif more-test-commands; then
                more-consequents;]
        [else alternate-consequents;]
fi
```

- colon (`:`) is a null command
- example

```
#!/bin/sh
if expr $TERM = "xterm"; then
  echo "hello xterm";
else
  echo "something else";
fi
```


sh (17) —case

- example:

```
case test-var in
value1) consequent-commands;;
value2) consequent-commands;;
*) default-commands;
esac
```

- pattern matching:
  - `?)` matches a string with exactly one character
  - `?*)` matches a string with one or more characters
  - `[yY]|[yY][eE][sS])` matches `y, Y, yes, YES, yES`...
  - `/*/*[0-9])` matches filename with wildcards like `/xxx/yyy/zzz3`
  - notice two semi-colons at the end of each clause
  - stops after first match with a value
  - you don't need double quotes to match string values!

sh (18) —case example

```
#!/bin/sh
case "$TERM" in
xterm) echo "hello xterm";;
vt100) echo "hello vt100";;
*) echo "something else";;
esac
```


sh (19) —expansion

- biggest difference from traditional programming languages
- shell substitutes and executes
- order:
  - brace expansion
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - word splitting
  - filename expansion

sh (20) —brace expansion

- expand comma-separated list of strings into separate words:

```
unix-prompt# echo a{d,c,b}e
ade ace abe
```

- useful for generating list of filenames:

```
unix-prompt# mkdir hw{1,2,3}
unix-prompt# ls
hw1  hw2  hw3
```


sh (21) —tilde expansion

- `~` expands to `$HOME`
- examples:
  `~cs3157` ⇒ `/home/cs3157`
  `~/html` ⇒ `/home/sklar/html`

sh (22) —command substitution

- replace `$(command)` or `‘command‘` by stdout of executing command
- can be used to execute content of variables:

```
unix$ x=ls
unix$ $x
myfile.c
a.out
unix$ echo $x
ls
unix$ echo ‘ls‘
myfile.c
a.out
unix$ echo ‘x‘
sh: x: command not found
unix$ echo ‘$x‘
myfile.c
a.out
unix$ echo $(ls)
myfile.c
a.out
unix$ echo $(x)
sh: x: command not found
unix$ echo $($x)
myfile.c
a.out
```


sh (23) —fi lename expansion

- any word containing `*?([` is considered a *pattern*
- `*` matches any string
- `?` matches any single character
- `[...]` matches any of the enclosed characters

```
unix$ ls
myfile.c
a.out
a.b
unix$ ls a*
a.out
a.b
unix$ ls a?
ls: No match.
unix$ ls a.*
a.out
a.b
unix$ ls a.?
a.b
unix$ ls a.???
a.out
unix$ ls [am].b
a.b
```

sh (24) —redirections

- `stdin`, `stdout` and `stderr` may be redirected
- `<` redirects `stdin` (0) to come from a file
- `>` redirects `stdout` (1) to go to file
- `>>` appends `stdout` to the end of a file
- `&>` redirects `stderr` (2)
- `>&` redirects `stdout` and `stderr`, e.g.: `2>&1` sends `stderr` to the same place that `stdout` is going
- `<<` gets input from a *here document*, i.e., the input is what you type, rather than reading from a file

built-in commands (1)

- `alias, unalias` — create or remove a pseudonym or shorthand for a command or series of commands
- `jobs, fg, bg, stop, notify` — control process execution
- `command` — execute a simple command
- `cd, chdir, pushd, popd, dirs` — change working directory
- `echo` — display a line of text
- `history, fc` — process command history list
- `set, unset, setenv, unsetenv, export` — shell built-in functions to determine the characteristics for environmental variables of the current shell and its descendents
- `getopts` — parse utility options
- `hash, rehash, unhash, hashstat` — evaluate the internal hash table of the contents of directories
- `kill` — send a signal to a process

built-in commands (2)

- `pwd` — print name of current/working directory
- `shift` — shell built-in function to traverse either a shell's argument list or a list of field-separated words
- `readonly` — shell built-in function to protect the value of the given variable from reassignment
- `source` — execute a file as a shell script
- `suspend` — shell built-in function to halt the current shell
- `test` — check file types and compare values
- `times` — shell built-in function to report time usages of the current shell
- `trap, onintr` — shell built-in functions to respond to (hardware) signals
- `type` — write a description of command type
- `typeset, whence` — shell built-in functions to set/get attributes and values for shell variables and functions

built-in commands (3)

- `limit, ulimit, unlimit` — set or get limitations on the system resources available to the current shell and its descendents
- `umask` — get or set the file mode creation mask