

CS246

Unix: more shell scripting

C:queues

April 8

# apropos

- get a listing of all commands that have the word in a short version of their man page
- UNIX> apropos file
  - not working on my mac
- UNIX> apropos file | wc returns 1198 possible items
- Contrast with “which”

```
gtowell@benz:~$ apropos gzip
```

```
gzip (1)      - compress or expand files
lz (1)       - gunzips and shows a listing of a gzip'd tar'd archive
tgz (1)      - makes a gzip'd tar archive
uz (1)       - gunzips and extracts a gzip'd tar'd archive
zforce (1)   - force a '.gz' extension on all gzip files
```

# Shell script loops

- while loop looks a lot like if except
  - if then if
  - while do done

```
1.#!/bin/bash
2.# Basic while loop

4.counter=1
5.while [ $counter -le 10 ]
6.do
7.echo $counter
8.((counter++))
9.done

11.echo All done
12.
```

# loops in shell scripts

- `$(( ))` in shell script mean – roughly “I am doing math inside here”
- hence “`counter=$((counter+1))`” does what you think
  - unlike `[]` not space sensitive
- `${}` notation for vars is never wrong to use
  - `counter=$(( ${counter}+1 ))`
- Sometimes there are shortcuts
  - `((counter++))`
    - note no leading `$`
    - putting `$` would indicate that the math returned a value
  - I tend away from shortcuts because I never remember them

```
#!/bin/bash
counter=1
while [ $counter -le 10 ]
do
echo $counter
counter=$((counter+1))
((counter++)) # shortcut form
done
echo All done
```

# shell loops

- Several forms of for loops exist.
  - This one takes string, splits it and works with each part
- The string to be split can come from anywhere
  - for instance, from executing ls

```
#!/bin/bash
# Basic for loop
names='Stan Kyle Cartman'
# names=`ls`
for name in $names
do
echo $name
done
echo All done
```

# more shell loops

- note (( )) to invoke math-like operations,
  - again, the shortcut form

```
#!/bin/bash
# Basic range in for loop
for value in {1..5}
do
echo $value
done
echo All done
```

```
#!/bin/bash
# Basic C-like for loop
for (( i=1; i<=5; i++ ))
do
echo $i
done
echo All done
```

# Space used by files with an extension

- usage: UNIX> l5.sh sh
- This gives the number of bytes used by all files with .sh extension in the current directory
- The FILES var is actually unnecessary
  - kind of wasteful also
  - change  
for FILE in \$FILES  
to  
for FILE in \*.\$1

```
file l5.sh

FILES=`ls *.${1}`
TOT=0
for FILE in $FILES
do
    DET=`ls -l $FILE`
    CNT=0
    for DETP in $DET
    do
        #echo "$CNT $DETP"
        if [ $CNT -eq 4 ]
        then
            TOT=$(( $TOT + $DETP ))
            #echo $DETP
        fi
        ((CNT++))
    done
done
echo $TOT
```

# the publish problem

## Solution

- make a link to the file using `ln -s` where the link has a browser-known extension
  - e.g. “.txt”
  - Allows updating without having to recopy a lot
- Create a shell script to do this in bulk.

bash has some built in text manipulation. This uses it  
`${VAR/replace/replacement}`

Suppose you did not know of bash text manipulation

```
aa=`echo $f | tr -d .`
```

or

```
aa=`echo $f | tr . d`  
ln -s $f ${a}.txt
```

```
#!/bin/bash  
param="java"  
if [ "$1" != "" ]; then  
    param=$1  
fi  
for f in *.$param;  
do  
    echo ${f/.//}  
    ln -s $f ${f/.//}.txt  
done
```



# LAB

- write a shell script that lists all files in the current directory and all direct subdirectories.
  - This should not be a recursive listing of all subdirectories. Just go one down

# Queues

- Use the DLL struct
  - needs more
    - removeTail
- Revisit DLLItem constructor/destructor and eliminate the copy into new memory. Just take the thing supplied
- Otherwise need to take care to free the returned thing!!

```
char* removeTail(DLL *dll) {  
    if (dll->count<=0)  
        return NULL;  
    dll->count--;  
    DLLItem *itm = dll->tail;  
    DLLItem *tprev = itm->prev;  
    if (tprev==NULL) {  
        dll->head = NULL;  
        dll->tail = NULL;  
        return;  
    }  
    dll->tail = tprev;  
    tprev->next = NULL;  
    char *rtn = malloc((strlen(itm->payload) + 1) * sizeof(char));  
    strcpy(rtn, itm->payload);  
    freeDLLItem(itm);  
}
```

required because  
the info would be lost  
otherwise, but!!!!

# Q Basics

- Constructor, destructor, and struct are pretty minimal

```
typedef struct {  
    DLL *internal;  
} Queue;
```

```
Queue* makeQueue() {  
    Queue *rtn = malloc(1 * sizeof(Queue));  
    rtn->internal = makeDLL();  
    return rtn;  
}
```

```
void freeQueue(Queue* q) {  
    freeDLL(q->internal);  
    free(q);  
}
```

# Q more

- rest is pretty basic also
- Essentially all work done by DLL!

```
void add2Queue(Queue* q, char* item) {  
    addDLLHead(q->internal, item);  
}
```

```
char* pullFromQueue(Queue* q) {  
    return removeTail(q->internal);  
}
```

# Splitting & Making

- I made a single dll.c and dll.h
  - IMHO DLLItem is more a private inner class and so it does not get its own file(s)
    - but this is a style choice
- Also a .c and .h for queue

```
CFLAGS = -g -O2
```

```
dll: dll.c dll.h  
    gcc $(CFLAGS) -o dll dll.c
```

```
queue: dll.o queue.c  
    gcc $(CFLAGS) -o queue dll.o queue.c
```

```
dll.o: dll.c dll.h  
    gcc $(CFLAGS) -c -D DOT0=1 dll.c
```

```
clean:  
    rm *.o dll queue
```

# Splitting Issues 1

- C does not allow typedefs to be declared multiple times
  - explicitly forbidden in C11
- Consider three files at right
  - when compile get error
  - a.h:3:3: note: previous declaration of 'A' was here  
3 | } A;
- Solution 1 ... instruct users to avoid the issue ...
  - do not actually need the a.h include since it comes in with b.h
    - lousy solution since it relies on everyone knowing and REMEMBERING
  - Also, a.h may come in via some other .h file so even if you assume people remember a conflict can occur

```
file a.h

typedef struct {
    int v1;
} A;
```

```
file b.h

#include "a.h"
typedef struct {
    A oneA;
} B;
```

```
file b.c

#include "a.h"
#include "b.h"
B b;
A a;
int main(int argc, char const *argv[]) {
    return 0;
}
```

# Splitting issues 1 continued

- Better solution is to ensure that definitions can only occur once!!
  - use `#ifndef ... #endif`

```
file a.h

#ifndef TYPE_A_DEFINED
typedef struct {
    int v1;
} A;
#define TYPE_A_DEFINED 1
#endif
```

```
file b.h

#include "a.h"

#ifndef TYPE_B_DEFINED
typedef struct {
    A oneA;
} B;
#define TYPE_B_DEFINED 1
#endif
```

# Splitting Issues 2

- Problem
  - this all looks correct ... and it is
  - but gcc fails

```
[gtowell@powerpuff L14]$ gcc s2.c
/bin/ld: /tmp/cc8w1v2M.o: in function `main':
s2.c:(.text+0x15): undefined reference to `doS1'
collect2: error: ld returned 1 exit status
```

- Why?
  - Linking issues!!!

```
file s1.h
int doS1();
```

```
file s1.c
#include "s1.h"
int doS1() {
    return 42;
}
```

```
file s2.c
#include "s1.h"
#include <stdio.h>
int main(int argc, char const *argv[])
{
    printf("%d\n", doS1());
    return 0;
}
```



# Splitting Issues 2

## Continued

- Recall that gcc actually takes 3 steps
  - 1. Preprocess
  - 2. Compile
  - 3. Link
- In this case preprocess and compile both work fine.
- Those steps take the function prototypes as promises that the definition will appear later
  - later **MUST** be in the link.
    - in this case there is no body supplied for `doS1()`



Use a makefile!!!

```
[gtowell@powerpuff L14]$ gcc -c s1.c  
[gtowell@powerpuff L14]$ gcc -o s2 s2.c s1.o
```

# Splitting issue 3

- I want each of my .c files to have a main function for testing purposes. But if I do that, the compiler (actually the linker) complains

```
[gtowell@powerpuff L14]$ gcc m1.c
[gtowell@powerpuff L14]$ ./a.out
M1 42
[gtowell@powerpuff L14]$ gcc -c m1.c
[gtowell@powerpuff L14]$ gcc m2.c m1.o
/bin/ld: m1.o: in function `main':
m1.c:(.text+0xb): multiple definition of `main'; /tmp/
cctzlOl8.o:m2.c:(.text+0xb): first defined here
collect2: error: ld returned 1 exit status
```

```
file m1.h

int dom1();
```

```
file m2.h

int doS1();
```

```
file m1.c

int dom1() {
    return 42;
}

int main(int argc,
char const
*argv[])
{
    printf("M1
%d\n", dom1());
    return 0;
}
```

```
file m2.c

#include "m1.h"

int dom2() {
    return 84;
}

int main(int argc,
char const *argv[])
{
    printf("M2  %d
%d", dom1(),
dom2());
    return 0;
}
```

# Splitting issue 3 continued

- The problem is that there are two implementations of main and they conflict.
- Once solution would be to put the main functions into their own file and compile/link appropriately
- This gets cumbersome (lots of files) and it loses the clear linkage between the main and the functions being tested.
- Better is to wrap main `#ifndef`
- then when compiling with `-c` flag a `-D` to define `M1C_MAIN`

```
[gtowell@powerpuff L14]$ gcc -c -DM1C_MAIN m1.c  
[gtowell@powerpuff L14]$ gcc m2.c m1.o  
[gtowell@powerpuff L14]$ a.out
```

```
file m1.c  
  
int dom1() {  
    return 42;  
}  
  
#ifndef M1C_MAIN  
int main(int argc,  
char const  
*argv[])  
{  
    printf("M1  
%d\n", dom1());  
    return 0;  
}  
#endif
```

# Review

- Pointers, pointers and more pointers
- recursion and tail recursion
  - why tail recursion matters
- typedefs
- structs
  - why not pass by value
  - constructors and destructors
- malloc and free
- UNIX
  - putting it all together with scripts

STOP HERE

# Garbage Collection in Java

- Why doesn't java need free?

# GDB

- “Gnu DeBugger”
- Allows you to inspect program while running
  - breakpoints
  - conditional breakpoints
- Another way to attack segmentation faults
  - arguably better
- Debuggers arguably give a lot more flexibility than print statements

# A Program that breaks

- gdb loves line numbers
  - cat -n xxx.c
- Program has three issues

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void smashing() {
5      int aa[10];
6      for (int i = 0; i < 20; i++) {
7          aa[i] = i;
8      }
9  }
10 int main(int argc, char const *argv[])
11 {
12     int strt = atoi(argv[1]);
13     int aa[strt];
14     smashing();
15     for (int i = 0; i < 1000; i++)
16     {
17         printf("%d %d\n", i, aa[i]);
18     }
19     return 0;
20 }
```



# gdb usage

- gcc
  - compile with -g flag
    - like valgrind
- UNIX> gdb executable
  - Equivalently
    - UNIX> gdb
    - (gdb) file executable
  - like valgrind
- Does not start the program

```
[gtowell@powerpuff L14]$ gcc -g broken.c
```

```
[gtowell@powerpuff L14]$ gdb a.out
```

```
GNU gdb (GDB) 9.1
```

```
Copyright (C) 2020 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
```

```
This is free software: you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law.
```

```
Type "show copying" and "show warranty" for details.
```

```
This GDB was configured as "x86_64-pc-linux-gnu".
```

```
Type "show configuration" for configuration details.
```

```
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>.
```

```
Find the GDB manual and other documentation resources online at:
```

```
<http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".
```

```
Type "apropos word" to search for commands related to "word"...
```

```
--Type <RET> for more, q to quit, c to continue without paging--q
```

```
Quit
```

```
(gdb)
```

# gdb help

- gdb is interactive and runs its own shell-like thing
  - tab completion for commands
  - file name completion
  - help
- (gdb) help [command]

# gdb basic usage

- quit
  - exit gdb
- run
  - runs the program without args
- run arg1 arg2 ...
  - exactly like UNIX> executable arg1 arg2 ...
-

# gdb breakpoints

- places where the program execution will stop
  - you can set as many as you want
- by line number
  - (gdb) break filename:linenumber
  - if only a single file can omit filename
  - gdb broken.c:12
- by function:
  - (gdb) break smashing
    - no filename since function names are unique in C

# gdb doing things at a pause

- (gdb) continue
  - resume program execution
- (gdb) step
  - advance one line in program
  - will go into called functions
- (gdb) next
  - does not go into called functions
    - other debuggers call this “step over”
- (gdb)<ENTER> repeat last command

# gdb — inspecting when program paused

- to look at the value of a variable when program is paused
  - (gdb) print varName
- (gdb) watch varName
  - program pauses whenever named var changes!
-

# Conditional breakpoints

- (gdb) break 12 if i>10
-