CS246
Unix: History
C: reading files, Pointers, Makefiles

March 8
Thursday’s Lab

- Lines containing z: “z”
- 2 instances of z: “z.*z”
- 2 non-consecutive instances of z: “z.*.z”, or “z..*z”
- At least 2 uppercase vowels: “[AEIOU]*[AEIOU]”
- 2 non-I uppercase vowels separated by 10 or more characters: “[AEOU]………..*[AEOU]”
  - some people found numeric quantifiers and wrote
    - [AEOU].{10,}[AEOU]
    - [AEOU].{10}.*[AEOU]
- fgrep, grep and egrep
  - fgrep – basically no regular expression O(M+N)
  - grep O(MN)
  - egrep – extended regular expression syntax
Unix: History

• Shells remember what you have done
  • up arrow to get previous command(s)
• Lines can be edited
  • ctrl-a beginning of line
  • ctrl-e end of line
  • backspace delete prev char
  • ctrl-d delete next char

• History goes back a ways
  • shell dependent but often 500 or more
Unix: History

- UNIX> history
  - command to show you all of the previous commands remembers
- List is long
  - how long??
  - history | wc
- really boring to search with up arrow!
  - Use grep!!!
  - history | grep grep
  - shows all of my usages of grep in the history

492  gcc mystrcpy.c
493  a.out
494  gcc mystrcpy.c
495  a.out
496  exit
497  ~/public/206/a4/dickens.txt | wc
498  grep z..*z ~/public/206/a4/dickens.txt | wc
499  exit
500  grep z..*z ~/Public/206/a4/dickens.txt | wc
501  grep z.+z ~/Public/206/a4/dickens.txt | wc
502  grep "[^z]*z[^z]*z[^z]*" ~/Public/206/a4/dickens.txt
503  grep "[^z]*z[^z]*z[^z]*" ~/Public/206/a4/dickens.txt
UNIX: history

• If just want to repeat a command
  • !123
    • execute the command with number 123 in the history list
  •
head, tail, and less

• “cat” is OK. It shows the file but it is inconvenient especially on big files

• less == cat with pagination
  • spacebar == forward a page
  • return == forward a line
  • b == backward a page
  • /xxx search for xxx

• head
  • show the first 10 lines of file
  • head -N == show the first N lines of file

• less
  • show the last 10 lines of a file
  • less -N
Reading Files

- `fopen` to read a file
  - "r" means open for reading
  - Style — I name all file vars "f***" and try to avoid f* for anything else
- Every call to `fopen` should be followed by check to make sure it worked
- `fprintf "file printf"
  - first param is the file to print to
- Read just like reading from stdin
  - stdin is a FILE*
- Everything opened must be closed

```c
file: OpenRead.c

int main(int argc, char const *argv[]) {
    FILE *fInput = fopen("OpenClose.c", "r");
    if (NULL == fInput) {
        fprintf(stderr, "Failed to open file for reading...
        return 1;
    }
    char line[LINE_LEN];
    while (NULL != fgets(line, LINE_LEN, fInput)) {
        fprintf(stdout, "%s", line); 
    }
    fclose(fInput);
    return 0;
}
```
Reading and Writing

- `fopen`
  - "r" — read
  - "w" — write
  - "a" — append
- You can open a lot of `FILE*`
  - there is a bound
- Again, looks almost identical to writing to `stdout`
- This copier works only on text files
- `fscanf` and the buffer overflow attack
  - so avoid use except, maybe, for keyboard input
    - problem, you really do not know what `stdin` is reading from

```c
#define LINE_LEN 256

int main(int argc, char const *argv[]) {
    if (argc < 3) {
        printf("Usage: xxx existing_file_name name_of_copy
        return 0;
    }
    FILE *fInput = fopen(argv[1], "r");
    if (NULL == fInput){
        fprintf(stderr, "Failed to open %s for reading ... terminating
        return 1;
    }
    FILE *fOutput = fopen(argv[2], "w");
    if (NULL == fOutput){
        fprintf(stderr, "Failed to open %s for output ... terminating
        return 1;
    }
    char line[LINE_LEN];
    while (NULL != fgets(line, LINE_LEN, fInput)) {
        fprintf(fOutput, "%s", line);
    }
    fclose(fInput);
    fclose(fOutput);
    return 0;
}
```
Returning multiple values from a function

• C functions only return 1 value
• But can use Pbr to get round this limitation
  • see also scanf

```c
#include "RetThree.c"

int mreturn(int *i1, double *d1, float *f1);

int main(int argc, char const *argv[]) {
    int ival = 9;
    double dval = 12.0;
    float fval = 12.9f;
    printf("%7d %7.2f %7.2f\n", ival, dval, fval);
    mreturn(&ival, &dval, &fval);
    printf("%7d %7.2f %7.2f\n", ival, dval, fval);
    return 0;
}

int mreturn(int *i1, double *d1, float *f1) {
    *i1 = *i1 - 5;
    *d1 = *i1 / *d1;
    *f1 = *d1 * *f1;
}
```
Arrays, the C way

- recall that for an 2 dimensional array the location calculation is
  
  \[ \text{LOC} = \text{start} + \text{index2} \times \text{RowLength} \times \text{sizeof(storedThing)} + \text{index1} \times \text{sizeOf(storedThing)} \]

  - Thus every lookup in a 2d-array requires 2 adds and 3 multiplies
  - 3-D: 6 multiplies and 3 adds
  - 4-D: 10 multiplies and 4 adds
  - etc
  - (a smart compiler can reduce this in many circumstances)
C style Array access

- Use pointers!
- to advance through array, just increment the pointer
  - `++` moves the pointer forward by `sizeof(type)`
  - `+= N` move forward by `N*sizeof(type)`
- set through pointers also
  - not shown here

file: Point1.c

```c
int main(int argc, char const *argv[]) {
    int arr[10];
    for (int i = 0; i < 10; i++)
        arr[i] = i+100;

    int *arrp = arr;
    for (int i = 0; i < 10; i++) {
        printf("%d %d %12d\n", i, *arrp, arrp);
        arrp++;
    }
    return 0;
}
```
Pointer array access in 2D

• need to know where you are
  • ROW-MAJOR
• in a 2d array, to get the starting point need the starting point of a 1d array
  • int arr[2][5];
  • int *arrp = arr[0];
• while loop is more efficient form for pointer move over array
  • note *earr calculation

file: Point2.c

int main(int argc, char const *argv[]) {
  int arr[2][5];
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < 5; j++)
      arr[i][j] = i*100 + j;

  int *arrp = arr[0];
  for (int i = 0; i < 10; i++) {
    printf("%5d %5d %12d\n", i, *arrp, arrp);
    arrp++;
  }

  int *parr = arr[0];
  int *earr = parr + (2 * 5);
  while (parr < earr) {
    printf("%c %5d %12d\n", ' ', *parr, parr);
    parr++;
  }
}
Speed of Pointers vs array access

• For common array operations a modern compiler can optimize array access so much that using pointers is slower!
  • Once upon a time this was always a big win
• Now you have to work harder for the win.
  • and the win is often small
  • But it can be big
• Lesson:
  • if you are doing things with arrays that use conventional indices, then use array notation
  • But think about being tricky with pointers if you really need the speed

Point3speed1.c
  array indices are faster by about 15%

Point3speed2.c
  pointers are faster by about 5%

Point3speed3.c
  pointers are faster by about 20%
Splitting c across files and Makefiles

• Recall the problem of splitting files and building
• Consider Point3speed3.c
  • break it up into 2 .c file and a .h
    • splitM.c
      • only main and the global array
    • splitF.c
      • the other functions
  • split.h
    • the defines
    • function signatures for splitF
      • only need those used in main
    • the global array from splitM

file: split.h
#define D1 100
#define D2 100
#define D3 100
#define COLUMN "%10.6f"
extern int arr[D1][D2][D3];
void t1();
void t2();
Compiling and makefiles

• Then to compile:
  gcc -c splitF.c
  gcc -c splitM.c
  gcc -o split splitM.o splitF.o

• When there are only two files remembering all the steps is not hard. When there are 200 (or more) it gets really hard
  • Java: in the first pass through, the java compiler figures out what is dependent on which and what has changed
    • In second pass (re)compile as necessary
• Makefiles
  • a manual setup for what Java does
    • (Many IDE’s will generate makefiles)
Makefiles

• usually in a file named “makefile”
• invoked by Unix command “make”
  • make -f “file name other then makefile”

• A simple makefile consists of “rules” which are followed by “actions”

• A rule looks like
  • name: [dependency]*
    • that is a name followed by a list of 0 or more dependencies
  • name may either be a useful identifier or the name of a file
  • a dependency is either a file name or a rule name

• Actions
  • actions must be **indented with a tab**
  • are one or more unix actions
  • must be separated from the next rule by a blank line
Makefile rules and dependencies

- Rules determine if they need to be invoked
  - if the dependency is a name that is not the name of a file
    - the rule will be invoked
  - if the name is that of a file:
    - if dependency is a file
      - the dependency file has changed more recently than the named file
    - if the dependency is another name
      - that rule determines that it must be invoked
  - For example, to determine if the rule “splitF.o” should be invoked, compare the modification dates of splitF.c split.h to the file splitF.o
    - if either is newer, then this rule is invoked

splitF.o: splitF.c split.h
        gcc -c splitF.c
Makefile for split

• makefile may also define constants for use in the makefile
  • for instance first two lines at right
• full command to invoke
  • make -f makefile split
• default is to use makefile or Makefile
  • normally -f is unnecessary
• default is to use first rule
  • so just “make” in this case

• rule submit:
  • no dependencies so just do it
    • the “cd ..” is not permanent; its effect does not extend beyond the line it is on.

file: makefile
var = $(notdir $(CURDIR))
c = gcc

split: splitM.o splitF.o
$(c) -o split splitM.o splitF.o

splitF.o: splitF.c split.h
$(c) -c splitF.c

splitM.o: splitM.c split.h
$(c) -c splitM.c

submit:
    cd ..; /home/gtowell/bin/submit -c 246 -p 20 -d $(var)
Lab

• Write a Makefile that has 2 rules
  • Rule 1. a compile rule that compiles at least one of the c programs you wrote for homework 2
  • Rule 2. a “clean” rule which deletes a.out and any other executables in the directory
    • you can, and should, just hard code in the names of the other executables to be deleted
  • The compile rule should be the default

• If your make file is more than 6 lines long, you are probably doing something wrong.