CMPUT201W20B2 Week 13

Abram Hindle

April 7, 2020

Contents

1 Week13

`https://github.com/abramhindle/CMPUT201W20B2-public/tree/master/week12`

1.1 Copyright Statement

1.1.1 License

1.1.2 Hazel Code is licensed under AGPL3.0+

Hazel's code is also found here `https://github.com/hazelybell/examples/tree/C-2020-01`

Hazel code is licensed: The example code is licensed under the AGPL3+ license, unless otherwise noted.

1.2 Alternative version

Checkout the .txt, the .pdf, and the .html version

1.3 Init ORG-MODE

```
;; I need this for org-mode to work well
;; If we have a new org-mode use ob-shell
;; otherwise use ob-sh --- but not both!
(if (require 'ob-shell nil 'noerror)
  (progn
    (org-babel-do-load-languages 'org-babel-load-languages '((shell . t))))
  (progn
    (require 'ob-sh)
    (org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-src-fontify-natively t)
(setq org-confirm-babel-evaluate nil) ;; danger!
(custom-set-faces
```

```
 '(org-block ((t (:inherit shadow :foreground "black"))))
 '(org-code ((t (:inherit shadow :foreground "black")))))
```

1.3.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

1.3.2 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

1.4 Remember how to compile?

gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o programname programname.c

1.5 IO

stdio.h in C contains numerous IO routines.

You use it primarily for printf and scanf.

1.5.1 Streams

Programs that run in the UNIX terminal have 3 main streams:

- stdin
  - standard in or standard input to read
  - shell: ‘<‘ ‘|‘
  - C: ‘gets, getchar, scanf, fgets(stdin,...), read(stdin,...) , ...‘
- stdout
  - standard out or standard output to write out to the terminals

  - shell: ‘>‘ ‘|‘
  - C: ‘puts, printf, fputs(stdout,...), fputc(stdout,...), ...‘

- stderr:
  - standard err or standard error to write out to terminals but not modify the main output
  - ‘2>‘ or ‘2>&1 |‘ or ‘|&‘ (bash only)
  - C: ‘fputs(stderr,...), fputchar(stderr,...), fprintf(stderr,...), ...‘

They are called streams because you serially output information to them. And multiple sources can write to the stream. It's like talking or a stream of consciousness. 1 byte after another.

1. shell

   Typically a terminal will mix stdout and stderr.

   You can type in input to standard input.

   You can "pipe" input to standard input: |

   You can redirect file input to standard input: <

   Example: using a pipe to pipe the string ‘ALL CAPS‘ through the tr program to lower case it.

   ```
   echo ALL CAPS | tr '[:upper:]' '[:lower:]'

   all caps
   ```

   tr is a translation program it takes characters from 1 argument and turns them into another.

   ```
   echo ALL CAPS | tr 'ALC' 'ODP'

   ODD POPS
   ```

   We can make files by redirecting stdout to a file

   ```
   echo ALL CAPS > allcaps.txt
   cat allcaps.txt | tr 'AL' 'OP'
   echo From Ontario, catch those # just echo to stdout
   tr 'AC' 'IR' < allcaps.txt
   ```

```
OPP COPS
From Ontario, catch those
ILL RIPS
```

We can filter arbitrary commands:

```
ls | tr '[:lower:]' '[:upper:]'
```

```
20.TXT
30.TXT
ALLCAPS.TXT
ARGV
ARGV.C
ARGV-NEW
ARGVRAND
ARGVRAND.C
AUTO
BINARY.BIN
BINARYREAD.C
BINARYWRITE.C
COOLBEARS.TXT
FFLUSHRANDR
FFLUSHRANDR.C
FFLUSHREADER
FFLUSHREADER.C
FFLUSH.SH
FFLUSH.TXT
FGETS.TXT
FPRINTF.TXT
K_T
MMAPREAD.C
PERROR
PERROR.C
PRESENTATION.HTML
PRESENTATION.HTML~
PRESENTATION.ORG
PRESENTATION.ORG~
PRESENTATION.PDF
PRESENTATION.TEX
```

```
PRESENTATION.TEX~
PRESENTATION.TXT
PRESENTATION.TXT~
STDIN-EXAMPLE
STDIN-EXAMPLE.C
STDOUT-EXAMPLE
STDOUT-EXAMPLE.C
```

We can chain pipes:

```
echo translate AC IR LL LK
tr 'AC' 'IR' < allcaps.txt | sed -e 's/LL/LK/'
echo translate AC IR LL LK ^S
tr 'AC' 'IR' < allcaps.txt | sed -e 's/LL/LK/' | \
   sed -e 's/^/S/'
# we can chain commands together
echo translate AC IR LL LK ^S K K T
tr 'AC' 'IR' < allcaps.txt | sed -e 's/LL/LK/' | \
   sed -e 's/^/S/' | \
   sed -e 's/K /K T/'
ls | grep .org | sort
```

```
translate AC IR LL LK
ILK RIPS
translate AC IR LL LK ^S
SILK RIPS
translate AC IR LL LK ^S K K T
SILK TRIPS
presentation.org
presentation.org~
```

sed is a useful regular expression program for manipulating text.

(a) stderr & shell

```
ls -l missing
exit 0
```

Where is it?

```
ls -l missing 2>&1
exit 0
```

```
ls: cannot access 'missing': No such file or directory
```

Once we redirect stderr to stdout we can pipe it and manipulate it!

```
ls -l missing 2>&1 | tr '[:lower:]' '[:upper:]'
```

```
LS: CANNOT ACCESS 'MISSING': NO SUCH FILE OR DIRECTORY
```

Or perhaps we don't want to see the error

```
ls -l missing 2> /dev/null
exit 0
```

Maybe we just want stderr

```
ls -l *.org missing 2>&1 > /dev/null
exit 0
```

```
ls: cannot access 'missing': No such file or directory
```

Maybe we just want BOTH

```
ls -l *.org missing 2>&1
exit 0
```

```
ls: cannot access 'missing': No such file or directory
-rw-r--r-- 1 hindle1 hindle1 27925 Apr  7 11:17 presentation.org
```

2. C

   (a) output

```
#include <stdio.h>
int main() {
    printf("OK this is to stdout!\n");
    fprintf(stdout,"OK this is to stdout as well!\n");
    fprintf(stderr,"OK this is to stderr!\n");
    return 0;
}
```

```
OK this is to stdout!
OK this is to stdout as well!
```

Hmmm org-mode ignores stderr

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o stdout-example stdout-
./stdout-example 2>&1
```

```
OK this is to stderr!
OK this is to stdout!
OK this is to stdout as well!
```

Ah now it appears

(b) input

```
#include <stdio.h>
int main() {
    int input;
    if (scanf("%d", &input)!=1) abort();
    fprintf(stdout, "From stdin %d\n", input);
    fprintf(stderr, "ERR: From stdin %d\n", input);
    return 0;
}
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o stdin-example stdin-ex
echo 10 | ./stdin-example 2>&1
echo 20 > 20.txt
echo 30 > 30.txt
./stdin-example 2>&1 < 20.txt
./stdin-example < 30.txt 2>&1
```

```
ERR: From stdin 10
From stdin 10
ERR: From stdin 20
From stdin 20
ERR: From stdin 30
From stdin 30
```

You've mostly seen this before except the 'fprintf(stderr,...)' part.

1.5.2 Files

Files can be addressed as streams as well. But we have to open and close them. So we can treat files exactly like stdin and stdout but with a few changes.

1. We need a file handle (like stdin, stdout, or stderr). This handle is for the OS to know which file the process is talking about.
2. We need to decide if we are reading write or both and we need to open a file to produce a file handle. Or use an existing one.
   - fopen
3. We need to write to it using write and f* operations.
   - fprintf
   - fputs
4. We need to read from it using read and f* operations.
   - fgets
   - fgetc
5. We need to close the file after we're done. fclose.

1. open and close

   To open a file we use fopen. To close it we fclose. Don't use open and close because that's not portable. That's for the OS.

   ```
   FILE *fopen(const char *pathname, const char *mode);
   int fclose(FILE *stream);
   ```

   The mode is a string

   - "r" - read
   - "w" - write (erase file)
   - "a" - append (add to end of file)
   - "r+" - read and write
   - "w+" - write and read (erase file)
   - "a+" - append and read

   ```
   FILE * f_cb = fopen("coolbears.txt", "w"); // open coolbears.txt for writing
   int fclose(f_cb); // close coolbears.txt
   ```

If you don't close a file you can lose bytes you wrote to it because they didn't get flushed to disk. This is important because people might kill your program your you might reboot or shutdown the computer. If you want to ensure data is written try to engage in flush. Sometimes no data will appear until you flush or close the file. Keep those pipes clean.

(a) fopen

```
#include <stdio.h>
#define SIZE 1024
int main() {
    char buffer[SIZE] = {'\0'};
    // open coolbears.txt for writing
    FILE * f_cb  = fopen("coolbears.txt", "w");
    if (f_cb == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fputs("Polar bears", f_cb);
    fclose(f_cb);
    FILE * f_cbr = fopen("coolbears.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fgets(buffer, SIZE, f_cbr);
    printf("%s\n",buffer);
    fclose(f_cbr);
}
```

```
Polar bears
```

(b) perror

perror produces nice errors.

perror("An error string"); will report the immediate fopen error if there is one.

Copy this code or put it in macro.

```
FILE * file = fopen("filename", "w"); // open coolbears.txt for writing
if (file == NULL) {
```

```
    perror("filename could not be opened");
    abort();
}
```

```
#include <stdio.h>
int main() {
    // open a file I can't open
    FILE * f_cb  = fopen("/proc/whatever", "w");
    if (f_cb == NULL) {
        perror("Couldn't open /proc/whatever");
        abort();
    }
    printf("We shouldn't be here!\n");
}
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o perror ./perror.c && \
./perror 2>&1
exit 0
```

```
Couldn't open /proc/whatever: No such file or directory
Aborted (core dumped)
```

(c) closing

OK but what if we don't close it?

```
#include <stdio.h>
#define SIZE 1024
int main() {
    char buffer[SIZE] = {'\0'};
    // open coolbears.txt for writing
    FILE * f_cb  = fopen("coolbears.txt", "w");
    if (f_cb == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fputs("Polar bears", f_cb);
    FILE * f_cbr = fopen("coolbears.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
```

```
    fgets(buffer, SIZE, f_cbr);
    printf("This is the buffer before close: %s\n",buffer);
    fclose(f_cbr);
    fclose(f_cb);
    f_cbr = fopen("coolbears.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fgets(buffer, SIZE, f_cbr);
    printf("This is the buffer after close: %s\n",buffer);
    fclose(f_cbr);
    printf("Close your buffers!");
}
```

```
This is the buffer before close:
This is the buffer after close: Polar bears
Close your buffers!
```

```
cat coolbears.txt
```

```
Polar bears
```

i. fflushing and fclose

Now let's see what flush does for us!

```
#include <stdio.h>
#define SIZE 1024
int main() {
    char buffer[SIZE] = {'\0'};
    // open coolbears.txt for writing
    FILE * f_cb  = fopen("coolbears.txt", "w");
    if (f_cb == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fputs("Polar bears", f_cb);
    fflush(f_cb); // WE'RE FLUSHING!
    FILE * f_cbr = fopen("coolbears.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
```

```
    }
    fgets(buffer, SIZE, f_cbr);
    printf("This is the buffer before close but after flush: %s\n",buffer)
    fclose(f_cbr);
    fclose(f_cb);
    f_cbr = fopen("coolbears.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open coolbears.txt");
        abort();
    }
    fgets(buffer, SIZE, f_cbr);
    printf("This is the buffer after close: %s\n",buffer);
    fclose(f_cbr);
    printf("Close your buffers! Keep your pipes clean");
}
```

```
This is the buffer before close but after flush: Polar bears
This is the buffer after close: Polar bears
Close your buffers! Keep your pipes clean
```

2. writing

   (a) fprintf

   fprintf is printf for files. It takes a FILE * as the first argument and then it looks like printf after that.

   fputs is available too and does the same thing except no laying out of strings.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1024
int main() {
    srand(time(NULL));
    char buffer[SIZE] = {'\0'};
    // open coolbears.txt for writing
    FILE * f_cb  = fopen("fprintf.txt", "w");
    if (f_cb == NULL) {
        perror("Couldn't open fprintf.txt");
        abort();
    }
    // It's just like printf!
```

```
    fprintf(f_cb,"A random number %d\n", rand());
    fclose(f_cb);
    FILE * f_cbr = fopen("fprintf.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open fprintf.txt");
        abort();
    }
    fgets(buffer, SIZE, f_cbr);
    printf("%s\n",buffer);
    fclose(f_cbr);
}

A random number 1047100006
```

3. reading

   For reading text from a file you options like fgets, fgetc, and fscanf.

   (a) fscanf

   fscanf looks and feels like scanf except it outputs to FILE * streams. The first argument is a FILE *.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1024
#define CHECK(x) ((x)==1)?1:(abort(),0);
int main() {
    srand(time(NULL));
    char buffer[SIZE] = {'\0'};
    // open coolbears.txt for writing
    FILE * f_cb  = fopen("fprintf.txt", "w");
    if (f_cb == NULL) {
        perror("Couldn't open fprintf.txt");
        abort();
    }
    // It's just like printf!
    fprintf(f_cb,"A random number %d\n", rand());
    fclose(f_cb);
    FILE * f_cbr = fopen("fprintf.txt", "r");
    if (f_cbr == NULL) {
        perror("Couldn't open fprintf.txt");
```

```
        abort();
    }

    for (int i = 0 ; i < 3; i++) {
        CHECK(fscanf(f_cbr, "%s",buffer));
        printf("%s\n",buffer);
    }
    int input=0;
    CHECK(fscanf(f_cbr, "%d",&input));
    printf("%d\n", input);
    fclose(f_cbr);
    return 0;
}
```

```
A
random
number
1047100006
```

(b) fgets

fgets gets a little complicated because you have to test for EOF. You can check for an null response and use the feof function, but probably you have to do both. If you find you're repeating the last line of a file it is because you are reading nothing and you're reusing the buffer you just used.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 1024
#define N 8
int main() {
    char buffer[SIZE] = {'\0'};
    srand(time(NULL));
    // open coolbears.txt for writing
    FILE * filew  = fopen("fgets.txt", "w");
    if (filew == NULL) {
        perror("Couldn't open fgets.txt");
        abort();
    }
```

```
    // It's just like printf!
    const int totalLines = 1 + (rand() % N);
    for (int i = 0 ; i < totalLines; i++) {
        fprintf(filew,"A random number %d\n", rand());
    }
    fclose(filew);
    FILE * filer = fopen("fgets.txt", "r");
    if (filer == NULL) {
        perror("Couldn't open fgets.txt");
        abort();
    }

    while(!feof(filer)) {
        if (fgets(buffer, SIZE, filer)) {
           printf("fgets.txt: %s", buffer);
        }
    }
    fclose(filer);
    return 0;
}
```

```
fgets.txt: A random number 1872523400
fgets.txt: A random number 248514355
```

4. flushing

   If you want to ensure something gets to a file or a term you should flush. Typically I/O is BUFFERED. Buffered means it is flushed once a certain threshold is met, typically size but sometimes time (depending on the system). Buffered will increase latency to print something but will often improve overall bandwidth to disk.

   ```
   fflush(FILE * stream); // will flush your stream
   ```

   Flush when you need to.

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <time.h>
   #include <unistd.h>
   ```

```
#define SIZE 5
int main() {
    srand(time(NULL));
    FILE * file = fopen("fflush.txt", "w");
    if (file == NULL) {
        perror("Couldn't open fflush.txt");
        abort();
    }
    for (int i = 0 ; i < SIZE; i++) {
        fprintf(file, "%d\n", rand());
        fflush(file); // WE'RE FLUSHING!
        sleep(1);
    }
    fclose(file);
}
```

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 20
#define BUFF 1024
int main() {
    char buffer[BUFF];
    FILE * file = fopen("fflush.txt", "r");
    if (file == NULL) {
        perror("Couldn't open fflush.txt");
        abort();
    }
    while(!feof(file)) {
        if (!fgets(buffer,BUFF,file)) {
            break;
        }
        printf("%s", buffer);
        sleep(1);
    }
    fclose(file);
}
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o fflushrandr fflushrandr.c
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o fflushreader fflushreader.
```

```
echo This will take 7 seconds && \
( ./fflushrandr & sleep 2; ./fflushreader)

This will take 7 seconds
738397049
1872523400
248514355
1021488994
530744055
```

5. Binary Files

   From stdio.h:

   ```
   size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

   size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                 FILE *stream);
   ```

   fread and fwrite will write memory to a stream and back again. Any pointer can be used, the bytes in memory will be serialized in and out. It will not be compiler and architecture portable so carefully craft your structs before you write them out. Use explicit padding. For 64-bit and 32-bit compatibility pad to modulus 8 bytes.

   (a) Writing Binary Files

   fwrite is our buddy. It will help us write arbitrary sections of memory to a file.

   ```
   #include <assert.h>
   #include <stdio.h>
   #include <stdlib.h>
   #include <time.h>
   #define SIZE 5
   struct demo {
       int i;
       float f;
       double d;
       char c;
   };
   int main() {
   ```

```
    srand(time(NULL));
    FILE * file = fopen("binary.bin", "w");
    if (file == NULL) {
        perror("Couldn't open binary.bin");
        abort();
    }

    int realSize = 1 + (rand() % SIZE);
    assert(1==fwrite(&realSize, sizeof(int), 1, file));
    for (int i = 0 ; i < realSize; i++) {
        struct demo randd;
        randd.i = rand();
        randd.f = rand() / 2.0F;
        randd.d = 1.0 / (rand()+1.0);
        randd.c = 'X';
        printf("Writing %d\n", randd.i);
        printf("\tWriting %g\n", randd.f);
        printf("\tWriting %g\n", randd.d);
        printf("\tWriting %c\n", randd.c);
        assert(1==fwrite(&randd, sizeof(randd), 1, file));
    }
    fclose(file);
}
```

```
Writing 1646883231
Writing 7.92256e+08
Writing 2.11212e-09
Writing X
Writing 1695274898
Writing 1.04839e+08
Writing 1.32348e-08
Writing X
Writing 199259769
Writing 1.24009e+08
Writing 1.70407e-08
Writing X
Writing 1618074551
Writing 1.02567e+09
Writing 1.86372e-09
Writing X
```

```
Writing 827260517
Writing 2.30443e+08
Writing 1.78874e-09
Writing X
```

So those structs are written to binary.bin

When you write out structs, watch for padding. Look for the letter X. Count the number of bytes after the last X.

```
hexdump -C binary.bin
```

```
00000000  05 00 00 00 9f 71 29 62  7d e3 3c 4e 5c f2 aa 69  |.....q)b}.<N\..i
00000010  98 24 22 3e 58 4c 4e d8  9b 55 00 00 92 d7 0b 65  |.$">XLN..U.....e
00000020  b5 f6 c7 4c 78 97 33 bc  e5 6b 4c 3e 58 4c 4e d8  |...Lx.3..kL>XLN.
00000030  9b 55 00 00 79 76 e0 0b  15 87 ec 4c 60 15 89 d4  |.U..yv.....L`...
00000040  1d 4c 52 3e 58 4c 4e d8  9b 55 00 00 b7 db 71 60  |.LR>XLN..U....q`
00000050  c9 89 74 4e 9e 50 2b d3  5a 02 20 3e 58 4c 4e d8  |..tN.P+.Z. >XLN.
00000060  9b 55 00 00 65 fe 4e 31  94 c4 5b 4d 36 e2 8e 5a  |.U..e.N1..[M6..Z
00000070  f8 ba 1e 3e 58 4c 4e d8  9b 55 00 00              |...>XLN..U..|
0000007c
```

(b) Reading binary

Reading binary requires that you know what types you are reading. Be warned that if you mix different types you need to read them in proper order.

```
#include <assert.h>
#include <time.h>
#include <stdio.h>
#define BUFF 1024
struct demo {
    int i;
    float f;
    double d;
    char c;
};
int main() {
    char buffer[BUFF];
    FILE * file = fopen("binary.bin", "r");
    if (file == NULL) {
        perror("Couldn't open binary.bin");
```

```
        abort();
    }
    int size=0;
    assert(1==fread(&size, sizeof(size), 1, file));
    // we'll just ignore the size and just keep reading until we have
    // to stop.
    while(!feof(file)) {
        struct demo readDemo;
        if (1!=fread(&readDemo, sizeof(readDemo), 1, file)) {
           break;
        }
        printf("Reading %d\n", readDemo.i);
        printf("\tReading %f\n", readDemo.f);
        printf("\tReading %g\n", readDemo.d);
        printf("\tReading %c\n", readDemo.c);
    }
    fclose(file);
}
```

```
Reading 1646883231
Reading 792256320.000000
Reading 2.11212e-09
Reading X
Reading 1695274898
Reading 104838568.000000
Reading 1.32348e-08
Reading X
Reading 199259769
Reading 124008616.000000
Reading 1.70407e-08
Reading X
Reading 1618074551
Reading 1025667648.000000
Reading 1.86372e-09
Reading X
Reading 827260517
Reading 230443328.000000
Reading 1.78874e-09
Reading X
```

1.5.3 Command line arguments

How do I make program like?

```
./argv some commandline args -l
```

To get arguments from the commandline you can add the parameters:

- 'int argc' – number of commandline arguments
- 'char ** argv' – array of strings of commandline arguments

```
#include <stdio.h>

int main(int argc, char ** argv) {
    for (int i = 0 ; i < argc; i++) {
        printf("arg %d: %s\t", i, argv[i]);
    }
    puts("");
}
```

```
arg 0: /tmp/babel-7888jxN/C-bin-7888CpK
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o argv argv.c && \
./argv &&  \
./argv 1  && \
./argv 1 2 && \
./argv 1 2 3 && \
cp -f argv argv-new && \
./argv-new 1 2 3
```

```
arg 0: ./argv
arg 0: ./argv arg 1: 1
arg 0: ./argv arg 1: 1 arg 2: 2
arg 0: ./argv arg 1: 1 arg 2: 2 arg 3: 3
arg 0: ./argv-new arg 1: 1 arg 2: 2 arg 3: 3
```

1. atoi for integer arguments

   So atoi is your friend :)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv) {
    if (argc != 2) { exit(1); }
    int n = atoi(argv[1]);
    for (int i = 0 ; i < n; i++) {
        printf("%d\t", rand());
    }
    printf("\n");
}
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
     -o argvrand argvrand.c && \
(./argvrand || echo not enough args: $?) && \
./argvrand 1 && \
./argvrand 2 && \
./argvrand 3 && \
./argvrand 0
```

```
not enough args: 1
1804289383
1804289383 846930886
1804289383 846930886 1681692777
```

1.5.4 mmap()

mmap is neat, it maps memory to and from a file or even another process. We do this with shared libraries too. So you can read and write to a file just by writing to memory. The OS deals with it very effeciently you just have to be very size aware. mmapping files is not good for streams and stream processing, it gets complicated. It is good for fixed sized structs.

```
#include <assert.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define SIZE 2
struct demo {
    int i;
    float f;
    double d;
    char c;
    // char cc[7]; // you can make padding explicit
};
int main() {
    srand(time(NULL));
    FILE * file = fopen("binary.bin", "r+");
    if (file == NULL) {
        perror("Couldn't open binary.bin");
        abort();
    }
    int fd = fileno(file);
    int rsize = 0;
    assert(1==fread(&rsize, sizeof(rsize), 1, file));
    const size_t size = sizeof(int) + sizeof(struct demo) * rsize;
    printf("N %d struct demos are in binary.bin\n", rsize);
    const size_t new_size = size + sizeof(struct demo);
    // if you want to increase a file's size use ftruncate
    // before you do this
    ftruncate(fd, new_size);
    printf("mmapping %u bytes of memory from the file\n", new_size);
    int * mapped = mmap(0,
        new_size,
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        fd,
        0);
    if (mapped == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    // read 4 bytes from the head
    assert(rsize == (int)*mapped);
    // really abusive but we're 1 int after the start eh
    struct demo * demos = (struct demo *)(mapped+1);
    // OK now look for the read ?
```

```
    for (int i = 0 ; i < rsize; i++) {
        struct demo randd = demos[i];
        printf("Reading %d\n", randd.i);
        printf("\tReading %g\n", randd.f);
        printf("\tReading %g\n", randd.d);
        printf("\tReading %c\n", randd.c);
    }
    // demo we can write a -1
    // run the program twice and your first integer is -1
    demos[0].i = -1;
    // now let's extend the file by 1 record!
    demos[rsize] = demos[rsize-1];
    mapped[0] = rsize+1;
    munmap(demos, size);
    fclose(file);
}
```

```
N 5 struct demos are in binary.bin
mmapping 148 bytes of memory from the file
Reading 1646883231
Reading 7.92256e+08
Reading 2.11212e-09
Reading X
Reading 1695274898
Reading 1.04839e+08
Reading 1.32348e-08
Reading X
Reading 199259769
Reading 1.24009e+08
Reading 1.70407e-08
Reading X
Reading 1618074551
Reading 1.02567e+09
Reading 1.86372e-09
Reading X
Reading 827260517
Reading 2.30443e+08
Reading 1.78874e-09
Reading X
```

```
ls -l binary.bin
```

```
-rw-rw-r-- 1 hindle1 hindle1 148 Apr  7 11:18 binary.bin
```

If you want to see some bad code that's small and uses mmap checkout:
`https://github.com/abramhindle/a-simple-pseudo-bayesian-spam-filter/blob/master/filter.c`

1.6 References

KN King, C Programming, Chapter 28, 2nd Edition

Hazel Cambell's thorough notes on Stream I/O: `https://docs.google.com/document/d/1b48EzfP03JYEFt42wCajU5kv76oVbTxEXa2JO0q17ag/edit`