Assignment 4 – XD Report Template

Mira Saini

CSE 13S – Spring 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

This program takes in a text file or user input through stdin and prints out the index line byte value in hex, the byte value of each char in hex, and the ASCII representation of each char for every 16 byte line. To run the program, type in the command

```
./xd <file_name>
```

If no filename is entered, then the program will allow text input through stdin. If too many arguments are given after ./xd, then the program will exit with 1. If a filename is invalid, then the program exits with 1. If the filename is valid, the program runs through 16 byte lines in the file/text and first prints the line number in hex, the hex values of each char, and then the ASCII representation of each char, until the EOF is reached. If the EOF is reached without completing the next 16 byte line, when the input is taken from a file, the program will simply print the hex values until EOF and then a space for each missing byte value until the end of the 16 byte line. For stdin, only when 16 bytes are read into the buffer, will the program actually print the hex dump. If the input is not a multiple of 16, then the stream of input takes the next input the user gives and adds it to the original buffer, containing the last few bytes from the prior input. Then the ASCII representation will be printed as well. If a character is not printable, then a "." is printed in its place. Overall, this program should mimick xxd, with an additional feature of taking input through stdin.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- What is a buffer? Why use one?

  A buffer is memory storage that can store data that will be later placed in a different location. A buffer stores data in an array until all of the data is ready to be pushed somewhere else. We can use a buffer to do some task after a certain number of bytes is loaded into the buffer. Waiting until a certain amount of data has been read can reduce the number of transfers when we move that data someplace else.

- What is the return value of `read()`? What are the inputs?

  read() will return the number of bytes read from the buffer that is provided as once of the parameters.

The read() function has the parameters: file descriptor, a buffer, and the size of the buffer in bytes. The file descriptor is assigned to open(filename, O_RDONLY) which open a file to only read. If the file descriptor equals -1, then an error has occured. If the file descriptor doesn't return anything, then the file has been opened successfully. The file descriptor can also be set to 0 (special case) if we want the program to read from stdin. The read function will take the file, the buffer, and the size of the buffer in bytes and return the number of read bytes from the files into the buffer.

- What is a file no. ? What are the file numbers of `stdin`, `stdout`, and `stderr`?

  A file no. is a file descriptor which do different things depending on the value. The file descriptor is set to open(filename, O_RDONLY) in this assignment and will return -1 if there is an error opening the file or nothing (non negative) if the file was successfully open. There are special file descriptor values that can be assgined to the file descriptor to either read from stdin, from stdout or from stderr. File descriptor numbers:

  For stdin, fd = 0
  For stdout, fd = 1
  For stderr, fd = 2

- What are the cases in which `read(0,16)` will return 16? When will it *not* return 16?

  read(fd, buffer, buffer_byte_size) read(0,16) will return 16 if there are 16 bytes that can be read from stdin because the file descriptor/no. is 0. If there aren't 16 bytes and let's say there are only 12 bytes, then read will return 12. If fewer than 16 characters are entered, then read() will return that number of characters.

- Give at least 2 (very different) cases in which a file can not be read all at once

A file cannot be read all at once if the file size is larger than its buffer size. If the buffer can hold 16 bytes at a time and the file contains 20 bytes, the file must load the buffer twice to read the entire file.

Another reason could be that the place where the file is being read to is too small in size compared to the file itself. If the file has 40 bytes and the buffer can hold 40 bytes, but reads it into a string that can only hold 20 characters, then the string is too small to hold all of the file's contents.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays.

I could create a few different files that contain different inputs that I want to test in my xd file. Assuming that I have a delayinput.sh file in my directory, I could create a script that tests specific inputs that I can place in a different file.

```
#!/bin/bash

bash delayinput.sh my_file.txt | ./xd > actual.txt
```

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

```
xxd my_file.txt > expected.txt
cat actual.txt
cat expected.txt
if diff actual.txt expected.txt; then
  echo "passed"
else
  echo "failed"
fi
```

The file my_file.txt would contain the input that I would want to test, and then output of my xd.c progam would be placed in actual.txt. I could then have a expected file called expected.txt where the xxd progam output for the same input in my_file.txt would be placed. Then I could compare the input of these two files and if they are the same in this case, then the test would pass, otherwise the test would fail. I could also print the contents of my actual and expected files using "cat." If the tests failed, printing the contents would be helpful to compare their differences. I could put any contents in the my_file.txt to test the output and compare in the real xxd program and my xd program. Additionally, I could create a file for stdin input by manually entering the expected solution since xxd doesn't support stdin input. Then I could compare this file to my actual.txt file, containing the output from my xd file.

UPDATE: Since I have a runner.sh file, I can instead make it a bit easier on myself. I can mkdir a directory called tests where I put all my tests scripts. I should keep runner.sh in the same directory that houses my tests directory (not inside the test directory though). I can write various test scripts that will either return a 0 if no problems are found and a 1 if there are differences between the actual and expected outputs. Then I can run "bash runner.sh" to test my programs.

Some of the tests that I would include would check to make sure that printable chars are being printed while non-printable chars are replaced with a "." I would check that the spacing and newlines are correct in my output. I could also check to make sure that the program waits for 16 bytes to be loaded into the buffer before printing its contents for stdin inputs. I could also test to make sure that the program throttles correctly and I could test inputs with no chars, with less than 16 bytes, with more than 16 bytes, using stdin, etc. I could make additional tests to check that the functionality of my bad_xd file is the same as my xd file. Additionally, I could check that bad_xd was under 1000 chars after being clang-formatted.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To use this program, type the command

```
./xd <file_name>
```

The "file_name" should be the name of the file that you are trying to do a hex dump on, meaning you want to print out the byte value of the line in hex, the hex values of each char, and the ASCII representation of each char for every 16 byte line. If you want to enter your own text so that the program performs the same operations, type

```
./xd
```

press enter and then type the chars that you want the program to run through. The stdin stream is left open so if your input is not a multiple of 16, then your last input taken into the buffer will remain and additional input will be added on top to reach 16 bytes before the next line is printed. To exit out of the program, you may simply do Ctrk-Z or Ctrl-C.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{ lstlisting}`
which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[2]
Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

Since this program will most likely be fairly concise, I do not plan on adding functions in addition to main.

Without writing functions, I would organize my code like so:

First include all of my necessary header files and define my buffer-size variable to be 16 and then start writing main(int argc, char **argv). In main, I first check to make sure there are an appropriate amount of arguments in the command line. If there are more than 2, then exit(1). Then I declare a pointer to a file which would initialize to the second argument in the command line if there are exactly two args in the command line, otherwise it initializes to one because only one command was listed (the command to start the program ./xd). Then I declare my file descriptor and initialize it to 0 if my file pointer was initialized to one (no file given so take input from stdin), otherwise I initialize it to open(file, O_RDONLY) to actually read the file. Then I check to make sure that open didn't return -1 (invalid file).

Now I declare my buffer, a var to print out the line index in hex, an index to keep track of the next char to be processed, and my read var that holds the value returned by read() which is the number of bytes read in from the file. I use a while loop with condition 1 that will only terminate if the buffer_index reaches 0. Inside that loop, I have another while loop conditioned on the buffer index being less than the buffer size. Inside, I intialize my read variable using the read function and then check to make sure that my read var is still reading more than 0 bytes. If it is, then I increment my buffer index by the amount of bytes I just read in. This loop will keep repeating until the buffer index is equal to 16 bytes. Once it exits, the buffer_size var contains 16 bytes and the starting hex line index value is printed. I then use the first for loop to make 'i' increment until it reaches the value of the buffer_size (16 bytes). Inside the loop, I use an if-else statement which will print the unsigned char value of the buffer indexed at 'i' if the value of 'i' is less than the buffer_index value (basically does this for all 16 bytes in the line). Otherwise, it will just print three spaces. Then I do another check using an and mask which will print a space if the mask of i and 1 is 1. This creates the spaces between the bytes. Then I go on to use another for loop to basically do the same thing but print the ASCII representation instead. I loop through the range of the buffer index starting at 0 and then assign a var to the index of the buffer so then I can print that char if it's a printable char, otherwise I print a period. Then I use third for loop to shift the rest of the chars in the end of the non-multiple of 16 byte buffer to the beginning of the buffer. I loop through the range of the buffer-size minus the current buffer index and reassign the indexes of the buffer to the initial index + the buffer index value which moves everything to the beginning. After the loop, I reassign the buffer_index value to the buffer_size minus itself

[2]This is my footnote.

because that's how many spots are left in the buffer. Lastly, I increment the var that prints out the hex value index line by the buffer_size (16 bytes). Once the outside while loop breaks, the file descriptor closes.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

```
#include <stdio.h>
//include other header files here
//define buffer_size 16

int main(int argc, char **argv){
    int start = 0x00000000; //line index in hex to print
    if argc > 2{exit(1)}

    //declare n initalize file pointer
    if argc == 2{
        file == argv[1]
    }
    else{file == NULL}


    //declare buffer, read_in var, buffer_index var, start var to hold line index hex value

    while(1){
        while buffer_index < buffer_size{
            read_in = read(fd, buffer + buff_index, buff_size - buff_index)
            if read_in ==0 {break}
            //increment buff_index by itself + read_in
        }
    //print the start var

    //for loop to print the middle: chars in hex
    for i < range(buffer_size){
        if i < buff_index{
            //print the indexed char in hex
        }
        else{//print three spaces}
       //if statement to compare and mask of i & 1 ==1
       if i & 1 ==1{//print two spaces}
    }

    //print space

    //another for loop to print the ASCII reps of chars
    for i in range(buff_index){
        //assign char c to the indexed value of buffer
        if isprint(c){//print the char}
        else{print "."}
    }

    //last for loop to shift the remaining chars to beginning of buffer
    for i in range(buffer_size - buff_index){
        buffer[i] = buffer[buff_index + i]
    }

    buff_index = buff_size - buff_index
    start += buff_size
    if read_in == 0{break}
    }
    close(fd)
    return 0}
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- The inputs of every function (even if it's not a parameter)
- The outputs of every function (even if it's not the return value)
- The purpose of each function, a brief description about a sentence long.
- For more complicated functions, include pseudocode that describes how the function works
- For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

My only function that I have is my main function which is explained in the previous sections. I don't have any other functions.

Optimizations

This section is optional, but is required if you do the extra credit. It due `only` on your final design. You do not need it on your initial.

In what way did you make your code shorter. List everything you did!

I managed to decrease my char count from 1951 chars to 953 chars in my bad_xd file. The first thing I did to reduce my char count was change all of my variable names to single letter names. I changed argc to c and argv to v. I also changed my file pointer variable from file to just f. I changed BUFFER_SIZE to B, buffer to b, buffer_index to i, start to s, read_in to r, and fd to d. I also combined the assigning/initialization of some variables with the same type into a single line like int x, s, and d. I got rid of the 'const' in front of my file pointer variable because it was unneeded. After the declaration of my vars and my buffer, I did an if statement to check that if 'c¿2' I would exit(1) which is almost the same as in my original, except that the var name changed from argc to 'c' and I didn't use curly braces to contain the body of my if statement since it was only one line of code. After that I decided to use the ternary operation to change all of my if-else 5 line pieces of code into one line of code. I reduced the initialization of my file pointer and combined it with an if-else statement to determine what my file pointer 'f' should be assigned to (if c==2, then v[1], otherwise just 0). I also reduced the initialization and assigning of my file descriptor 'd' by doing the same thing (if 'f' != 0, then open(f, O_RDONLY), otherwise assign 'd' to 0. After this, I did another if statement check so that if my file descriptor 'd' returned -1 (meaning invalid), main would exit(1). I also didn't use curly braces for this if statement.

Moving on to my multiple loops, I decided to change my while(1) loop that had a break statement into a do-while loop. The break condition of the while would be that my read_in 'r' var equalled 0, meaning the read function wasn't reading anymore bytes. Inside of my do-while loop, I had a regular while loop that checked two conditions: buffer_index 'x' was ¡ BUFFER_SIZE 'B' and that read of my file did not return a 0. The while loop body was only one line long, so I didn't include braces here. I then check to make sure that my buffer_index 'x' didn't equal 0 (again without braces). Then I did a regular print statement like in xd to print out the hex value of the line index.

My next three for loops were drastically reduced in my bad_xd file. I kept the conditions of the loops the same, but I combined the code inside the bodies. For the first for loop, I implemented the ternary operator to combine my if-else 5 line statement and then used it again to combine my if statement that did the and masking. The next for loop was only one line, so I cut out the braces and used the ternary operator again to combine my if-else statement that checked if the char was printable. My last for loop was also only one line (no braces), this is where I shifted my buffer values to the front. When I incremented/decremented my buffer_index 'x' and my start 's' vars respectively, I used the combined operator '-=' and '+='

Outside of the do-while loop, I simply closed the file descriptor 'd' normally and returned 0.

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.