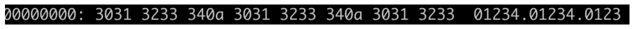
Assignment 7 – XXD

Johann Lee

CSE 13S – Fall 2023

Purpose and How to use.

The purpose of this project is to read a series of characters and translate them into hex characters. This is done in buffers of 16 bytes with groups of 8, 4 character groupings. Each grouping represents 2 characters from the original input. Compiling the program uses ”make” which will create the ”xd” binary. Run this binary using either ”./xd” or ”./xd ’filename.txt’”. The former takes in input from standard input and the latter reads from a text file. The output should look something like below.



```
00000000: 3031 3233 340a 3031 3233 340a 3031 3233  01234.01234.0123
```

Figure 1: Here is an example output. The first part represents how many bytes into the input the code has read. With next 8 groups are as mentioned before the input characters. Next is the line of characters that were translated for that specific group of 16 bytes.

File read will go through the entire file and exit by itself. Stdin on the other hand will require the user to exit. This is done by using ”ctrl d”. The program will print whatever is left in the buffer and close itself.

Program Design

My buffered reader works by using the read function within a while loop which is constantly updated based on past read data. This will be further elaborated in the pseudo-code. Within this while loop will be a check for if the buffer is full and if so a print loop will occur. This for the most part is how my buffered reader works.

```
while reader = read(file, buffer + past read bytes, buffer size - past read bytes)
    update readbytes =+ reader

    //check full buffer
    if readbytes == buffersize
        print buffer and index

        change any ASCII not between
        27 and 127 to ’.’
        print original input with
        changed ASCII

        reset readbytes
        increase index
```

My main function does not elaborate that much more on this buffered reader.

```
main(argc, argv)
    int fd
```

```
        char buffer[buffersize]
        ssize reader
        size readbytes
        int index

        if argc == 2
            open from argv[1] in read mode

        buffer reader //shown above in verbatim

        if readBytes > 0
            print buffer for readbytes
            once readBytes is full change to spaces
            to account for formatting

            change nonASCII characters to '.'
            putchar for readBytes

        close file
        return 0
```

Some errors that I currently expect and am in the process of fixing are lining up my original input on each new index when printing the leftover buffer as the length is not the same. Another issue is how my output currently treats newline as '0A' while the binary prints '0a'.

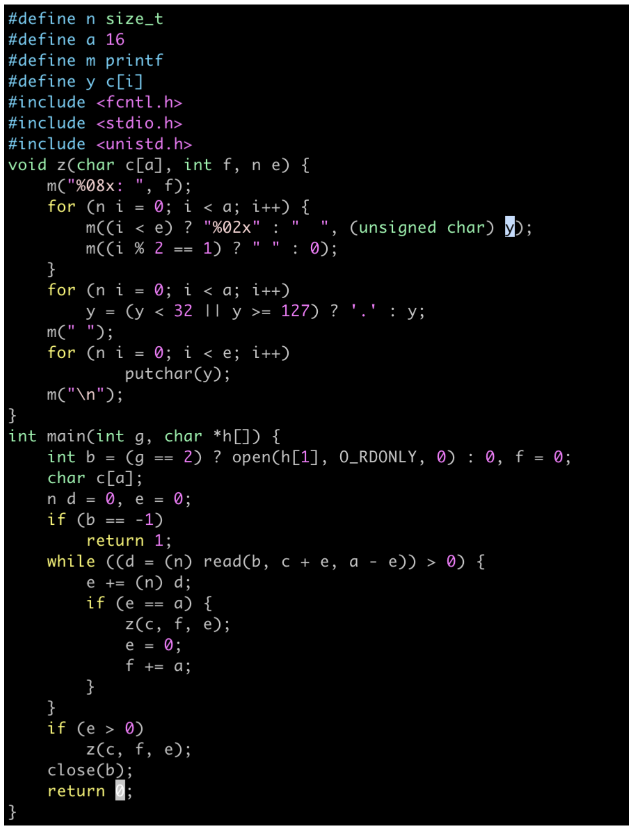
Result

I will be attempting extra credit and this will be done with a variety of techniques. First of these are changing variable names to shortened version.

```
    BUFFER_SIZE = a
    int fd = 0; = b
    char buffer[BUFFER_SIZE]; = c
    ssize_t reader = 0; = d
    size_t readBytes = 0; = e
    int index = 0; = f

    printf("%08x: ", index);
        for (size_t i = 0; i < BUFFER_SIZE; i++) {
            if (i < readBytes) {
                printf("%02x", (unsigned char) buffer[i]);
            } else
                printf("  ");
            if (i % 2 == 1) {
                printf(" ");
            }
        }
    for (size_t i = 0; i < a; i++) {
        if (c[i] < 32 || c[i] >= 127) {
            c[i] = '.';
        }
    }
    Changed this into a function void z(char c[a], int f, n e)
```

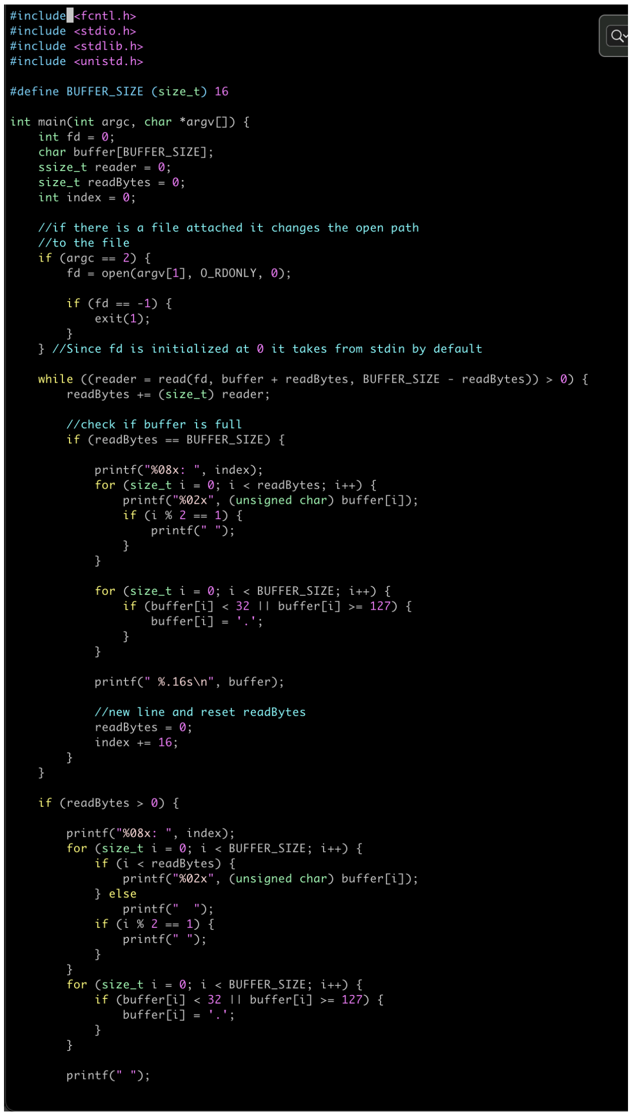
```
This was then used twice to replace
Furthermore, the print statements
used for printing the buffer
were replaced with putchar within
function z

Define was used 4 times
a 16
m printf
n size_t
y c[i]
These were each used to replace their
respective phrase. This was justified based
on how many times the specific phrase was used.

Next are the ternary operator changes
    When printing hex characters the
    prints statements were edited

    m((i < e) ? "%02x" : "  ", (unsigned char) y);
    m((i % 2 == 1) ? " " : 0);

    These ternary operations are within printf which are
    defined as 'm' as seen above. The first one checks
    that counter i < readBytes. If true it prints the
    according hexcharacter based on buffer[i]. Else it
    prints a double space which would replace the position
    of the hexcharacter. This was done to line up the statements.
    The second print statement checks to see if a space should be
    printed every other print statement from before. If it is false
    it does nothing.

    y = (y < 32 || y >= 127) ? '.' : y;

    This operation is identical to before when swapping any invalid
    ASCII character at buffer[i]. It if it true it sets buffer[i] to
    '.' else it just set its to itself

m(" ");
for (n i = 0; i < e; i++) {
    (y < 32 || y >= 127) ? putchar('.') : putchar(y);
}
m("\n");
This was the swapped printf to putchar method. Previously I had a printf
for a full buffer and a putchar for leftover buffer. However, I realized
it would be shorter to include this putchar loop into function z(). This
combines both my ASCII change line and the print line together into a putchar
ternary. It places a '.' if the character is not within bound or it puts
the ASCII character it if is within bounds.

Within main() are the standard methods of shortening such
as declaring variables on one line. I changed exit(1) to return 1
as it serves the same purpose and I could remove the stdlib.h file from
my included headers.
```

```
Another large change was changing the initializing of int b

int b = (g == 2) ? open(h[1], O_RDONLY, 0) : 0, f = 0;

by attaching the creation of b to a ternary operator I save 2 extra lines
of the if statement. Here it sets b to either open() if there are
2 argc values or to 0 if this is not true. 0 by default reading off
of the stdin.

Aside from these changes the main function
looks identical to the original main function if
the atrocious variable names are excluded along with the inclusion
of void z() and the removal of extra {} on single line conditionals.
```



```c
#define n size_t
#define a 16
#define m printf
#define y c[i]
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
void z(char c[a], int f, n e) {
    m("%08x: ", f);
    for (n i = 0; i < a; i++) {
        m((i < e) ? "%02x" : "  ", (unsigned char) y);
        m((i % 2 == 1) ? " " : 0);
    }
    for (n i = 0; i < a; i++)
        y = (y < 32 || y >= 127) ? '.' : y;
    m(" ");
    for (n i = 0; i < e; i++)
            putchar(y);
    m("\n");
}
int main(int g, char *h[]) {
    int b = (g == 2) ? open(h[1], O_RDONLY, 0) : 0, f = 0;
    char c[a];
    n d = 0, e = 0;
    if (b == -1)
        return 1;
    while ((d = (n) read(b, c + e, a - e)) > 0) {
        e += (n) d;
        if (e == a) {
            z(c, f, e);
            e = 0;
            f += a;
        }
    }
    if (e > 0)
        z(c, f, e);
    close(b);
    return 0;
}
```

Figure 2: Here is an image of bad_xd.c



```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE (size_t) 16

int main(int argc, char *argv[]) {
    int fd = 0;
    char buffer[BUFFER_SIZE];
    ssize_t reader = 0;
    size_t readBytes = 0;
    int index = 0;

    //if there is a file attached it changes the open path
    //to the file
    if (argc == 2) {
        fd = open(argv[1], O_RDONLY, 0);

        if (fd == -1) {
            exit(1);
        }
    } //Since fd is initialized at 0 it takes from stdin by default

    while ((reader = read(fd, buffer + readBytes, BUFFER_SIZE - readBytes)) > 0) {
        readBytes += (size_t) reader;

        //check if buffer is full
        if (readBytes == BUFFER_SIZE) {

            printf("%08x: ", index);
            for (size_t i = 0; i < readBytes; i++) {
                printf("%02x", (unsigned char) buffer[i]);
                if (i % 2 == 1) {
                    printf(" ");
                }
            }

            for (size_t i = 0; i < BUFFER_SIZE; i++) {
                if (buffer[i] < 32 || buffer[i] >= 127) {
                    buffer[i] = '.';
                }
            }

            printf(" %.16s\n", buffer);

            //new line and reset readBytes
            readBytes = 0;
            index += 16;
        }
    }

    if (readBytes > 0) {

        printf("%08x: ", index);
        for (size_t i = 0; i < BUFFER_SIZE; i++) {
            if (i < readBytes) {
                printf("%02x", (unsigned char) buffer[i]);
            } else
                printf("  ");
            if (i % 2 == 1) {
                printf(" ");
            }
        }
        for (size_t i = 0; i < BUFFER_SIZE; i++) {
            if (buffer[i] < 32 || buffer[i] >= 127) {
                buffer[i] = '.';
            }
        }

        printf(" ");
```

Figure 3: This is xd.c for reference at the same text size. It can be seen that bad_xd massively reduced the length of the program.