Assignment 4 – XXD
Dalal Arafeh
CSE 13S – Winter 24

Purpose

This program mimics the functionality of the xxd command, focusing on the conversion and display of binary data in both hexadecimal and ASCII formats. This program takes input from a file or directly from the user and reads it into a 16-byte buffer. It continues reading in a loop until it has processed all the data. For each 16-byte chunk, it displays the starting position in hexadecimal, followed by the hexadecimal value of each byte. Additionally, it saves these bytes in another buffer to show their ASCII characters on the screen. Once all the data is processed, the program outputs everything to the terminal.

Questions

- **What is a buffer (talk about the data type and the purpose)? Why use one?**
  An array of characters used to temporarily store data. In this assignment, we are using a buffer to read and process the input in chunks rather than one byte at a time. This is useful for this assignment because buffering accumulates a set of bytes before converting them to hexadecimal (aligns the output format).

- **What is the return value of read()? What are the inputs?**
  - The read() returns
    - An int of amount of characters successfully read (positive integer)
    - An error (-1)
    - End of File (0)
  - Inputs
    - int: file descriptor
    - void*: pointer to the buffer where data should be stored
    - size_t: maximum number of bytes to read

- What is a file no. ? What are the file numbers of stdin, stdout, and stderr?
  - A file no. is an integer that references a files and I/O operations
  - 0: stdin, 1: stdout, 2: stderr

- **What are the cases in which read(0,buffer,16) will return 16? When will it not return 16?**
  - read(0,buffer,16) will return 16 when there are at least 16 bytes of data available to be read from stdin. It will not return 16 if there are fewer than 16 bytes before reaching EOF or reading from a source that doesn't have enough data immediately available

- **Give at least 2 (very different) cases in which a file can not be read all at once**
  1) If the user is typing slowly or taking breaks between inputs. It will read the input character by character as it becomes available

2) If a program is reading a file from a remote server over a slow or congested network connection, there may be delays between transmitting each line of the file. It may need to wait for each line to be transmitted before proceeding

- **What is the range of char? A byte? The ASCII table? Which range should your program accept (if any of those)**
  - char:
    - Signed: -128 to 127
    - Unsigned: 0 to 255
  - Byte: all 0s to all 1s
  - ASCII: 0 to 127
    The program should treat data as bytes by converting the occurrences of 0s and 1s to hexadecimal regardless of the content of the file, since it's displaying binary data in hexadecimal form.

- **What is the decimal equivalent of the 8 bit integer 0b1001 0110?**
  - Unsigned: 150
  - Signed: -106

- Convert the 8 bit integer above to a 32 bit integer.
  0b00000000 00000000 00000000 10010110
  00000000 00000000 00000000 10010110

- **What does the %X format specifier mean? What type of data does it expect?**
  - %X: printing an unsigned integer in hexadecimal(X) with uppercase letters. It expects an unsigned int.

Testing

- **List what you will do to test your code. Make sure this is comprehensive. 4 Be sure to test inputs with delays and a wide range of files/characters**
  - Test with files exactly 16 bytes, more than 16 bytes, and less than 16 bytes
  - Test program can handle reading non-existing file by returning a non-zero error code

How to Use the Program

- Program reads input either from a file, user specifies the name as a command-line argument, or from stdin if no file is specified. The program then prints the hexadecimal representation of the input data alongside its ASCII equivalent.
  - Compiling the program using the Makefile, using make and clang commands.
  - Program tries to open a file if a name is provided or read from stdin if no file is specified
  - Each line starts with an offset, showing the position in the file in hexadecimal, followed by the hexadecimal representation of up to 16 bytes of data from the file

- Prints the ASCII representations of those bytes
- Non-printable characters are printed as a dot
- If the last line of input doesn't have 16 bytes, it prints extra spaces to align the ASCII representation correctly
- The program checks if the file opening was successful. If not, it returns 1 and exits, ensuring it does not attempt to read from an invalid file descriptor

Program Design

- Input buffer: buffer of size 16 bytes to open and read the input
- Line buffer: buffer for storing a line's worth of characters for ASCII display
- Track the number of bytes read in each input operation
- Attempt to open the specified file for reading; if unsuccessful, exit the program with an error code
- Exit the loop if no bytes were read (EOF or a read error)
- Print the hexadecimal representation of the byte
- After every second byte, print a space
- Store the byte in the line buffer for subsequent ASCII representation
- If reached max buffer size, end of line processing
- Print a space to separate hexadecimal and ASCII representations.
- Iterate over the line buffer, printing each byte's ASCII character if it's printable. Otherwise, print a dot
- Print a newline to complete the line's output
- Reflect the position of the next line in the data stream

Pseudocode

```
//headers for file operations and standard I/O

//buffer for reading input
unsigned char buffer[buffer size]

//buffer storing characters for ASCII display
unsigned char line buffer[buffer size]

//number of bytes read
ssize_t bytes read = 0
//processed bytes count
int bytes count = 0
//current offset
int current offset = 0

//print a single byte in hexadecimal format
        printf %02x
```

```
//print the ASCII representation of bytes stored in line buffer
        //iterate over each byte in the line buffer up to 'count' bytes
        for (ssize_t i = 0; i < count; ++i)
                If printable print character
                Otherwise print dot

//handle the end of a line's output
        //print two spaces before the ASCII representation
        //print ASCII characters


//process each byte read into the buffer
        //check if it's the start of a new line and if it is, print offset
        byte_count == 0
                //print offset
                printf %08x
                //increment offset for the next line
                offset += buffer size;

        //print the current byte in hexadecimal format
        //print a space for every two bytes

        //store the byte in line buffer for ASCII representation
        //increment byte count and loop after 16 bytes

//main function
int main(int argc, char **argv)
        //open the specified file if given, otherwise use standard input
        //exit with an error code if the file couldn't be opened

        while (1)
                //read up to max buffer size bytes from the input into the buffer
                //break to leave loop if no bytes were read (EOF)

                //process each byte read
                //for loop
                        //call process byte function
                        //if a line of bytes has been fully processed, handle the line end

        //handle any remaining bytes if the last line is incomplete
                //print padding spaces for incomplete lines
                //handle the end of the final line
```

//exit the program

Function Descriptions

- Main function
    - Determine the input source based on command-line arguments
    - Open the input source for reading
    - Enter a loop to read and process data in chunks until EOF
    - For each chunk of data, process and print its representation
    - Handle any remaining bytes and incomplete lines at the end of the input stream
    - Exit the program
    - Input: reads either from a file, user specifies the name as a command-line argument, or from stdin if no file is specified
    - Output: Hex and ASCII representation of input to stdout and an int exit code indicating success or failure
- Print hex byte
    - Print a single byte in hexadecimal format (binary data for us to understand)
    - Input: single unsigned char byte to be printed to hex
    - Output: outputs the hex representation of the byte to stdout
- Print ASCII character
    - Print the ASCII representation of the bytes stored in the line buffer and converting non-printable characters to dots
    - Input: single int specifying how many characters should be printed from the line buffer
    - Output: outputs characters to stdout, either as readable ASCII characters or as dots
- Handle line end
    - Manage the transition between lines in the output, ensure proper spacing and of the ASCII representation
    - Input: Determine how many characters to print
    - Output: spaces for alignment and the ASCII representation of bytes in line buffer to stdout
- Process byte
    - Check if starting a new line; if so, print the current offset
    - Print the byte in hex format
    - Every two bytes, add a space
    - Store the byte in line buffer
    - Increment byte count and reset if a line is completed.
    - Input: takes a single byte from the buffer
    - Output: hex representation of byte to stdout and stores the byte in line buffer for ASCII printing

If a user enters more than two arguments, the problem handles the error and terminates. If there is an error opening and reading the file, the program should also exit.

Optimization

- In what way did you make your code shorter? List everything you did!
    - I could replace some if-else statements with ternary operators
    - Make the names of my functions and variables short and concise
    - I could make implement most of my function in the main and limit calling functions
    - Error checking could be in fewer lines