

Problem Set 3

Due before midnight on Tuesday, February 5, 2008.

1 Assignment Goals

1. Learn how to read hex numbers and character strings from input streams.
2. Learn how to process streams where new line characters are semantically meaningful.
3. Learn how to detect end of file.
4. Learn how to use `sscanf()` to read from a string.
5. Learn how to write raw bytes to an output stream.
6. Learn how to use `char` arrays and strings.
7. Learn about the `od` command.
8. Learn about hex numbers for describing binary data.
9. Get practice manipulating different data representations.
10. [Extra credit] Learn how to use `union` to deal with byte order issues in C, and learn how to synchronize input and output where current output depends on future input.

2 The `od` Command

A file is a sequence of bytes. What those bytes represent and how they are to be interpreted is application dependent.

The command `od` is like a microscope for examining a file and looking directly at the bytes that comprise it. It is a useful tool for finding out what is “really” there as opposed to what the application that created the file is willing to let you see. Files often contain hidden data, which a tool like `od` can expose.

The name `od` is for “octal dump” and derives from the day when characters were 6 bits long and conventionally represented in octal (base 8). However, the `od` command now has many options and can dump a file in hex (base 16), which is convenient for representing 8-bit bytes. (See handout 5 on Bits and Bytes.)

The command `od -v -Ad -tx1 file` reads the specified file a byte at a time and prints out each byte as a hex number (without the leading “0x”). For example, if the file “hello.txt” contains the single line “hello!” with its terminating new line character, then `od -v -Ad -tx1 hello.txt` gives

```
0000000 68 65 6c 6c 6f 21 0a
0000007
```

The first number on each line is a decimal *byte position*. The remaining numbers are *byte values* in hex, each representing a single byte of the file. The byte position is the position in the file corresponding to the first byte value on the line. Bytes are numbered starting with zero, so the first line of output always has byte position 0. The last line contains a byte position but no byte values. The byte position is the position of the next byte of the file if there were one. Equivalently it's the length of the file.

In the above example, the first byte, 0x68, is the code for the character 'h'. The last byte, 0x0a, is the code for the new line character. The file itself contains 7 characters in positions 0 through 6, so the first "unused" position in the file is 7, which is the same as the file's length.

The `-Ad` switch to `od` causes the byte position to be represented in decimal. The `-v` switch disables line suppression (which makes the output simpler, though less concise). The `od` command has many options. Type `man od` or `info od` for detailed information on what they do.

If we change `-tx1` to `-tx2` in the above example, then the file is read two bytes at a time. Each pair of bytes is interpreted as a short unsigned int and printed out as a 4-digit hex number. Thus, `od -v -Ad -tx2 hello.txt` gives

```
0000000 6568 6c6c 216f 000a
0000007
```

Notice that an extra zero byte was placed at the end of the file. However, the file length in the last line is 7, not 8, so we know that the apparent 8th byte of the file should be ignored. Notice also the different order that the bytes appear. This is because this example was produced on a little-endian machine (like the Zoo machines). The first byte 0x68 becomes the lower-order byte of the 2-byte integer and the second byte 0x65 becomes the high-order byte. Thus, we have

$$\boxed{0x65} \boxed{0x68} = \boxed{01100101} \boxed{01101000} = 0x6568$$

3 Assignment

3.1 Required part

Your job is to write a program `unodx1` to perform the inverse of `od -v -Ad -tx1`. Suppose file `foo.dump1` is the output of `od`, produced from `foo` by

```
od -v -Ad -tx1 foo > foo.dump1
```

Then

```
unodx1 < foo.dump1 > foo_COPY
```

should produce a new file `foo_COPY` that is identical to the original file `foo`. Thus, `unodx1` undoes what `od -tx1 ...` does. You can use the `cmp` (compare) command to test if two files are identical or not. Your program should read from standard input and write to standard output. The shell input and output redirection characters, '`<`' and '`>`' respectively, can be used to redirect standard input and standard output to files as illustrated above.

Your program should check its input file for validity and reject invalid files with an appropriate error comment. Defining what constitutes an invalid file is not so easy. We say a file is *strictly correct* if it could be the output of `od -v -Ad -tx1` on the Zoo machines. A file is *valid* if it is strictly correct or is a minor variant of a strictly correct file, as defined below. All other files are *invalid*.

A strictly correct file has three kinds of lines. A *full line* contains a decimal byte position followed by 16 2-digit hex numbers. A *partial line* contains a decimal byte position followed by at least one but fewer than 16 2-digit hex numbers. A *terminal line* contains only a decimal number which is the total file length. A strictly correct file consists of zero or more full lines, followed by zero or one partial line, followed by a terminal line. The byte position of the first line is 0, and for every full or partial line, the byte position of the following line is equal to the byte position of the current line plus the number of hex numbers on the current line.

A valid file is one that is similar in form to a strictly correct file except for the following:

1. Additional white space may be present before or after any of the numbers or between lines.
2. The decimal numbers can be written in any form that `scanf()` accepts with the `%d` format specifier, as long as they have the correct value. (For example, a byte position could be written as `+004`, even though `od` never outputs `+`.)
3. The hex numbers can be anything that `scanf()` accepts with the `%hhx` format specifier. No restrictions are placed on their values.
4. The last line of the file might lack a terminating new line character.

Your program should accept all strictly correct files and work correctly on them. It should reject all invalid files. Valid files that are not strictly correct may be accepted by your program or rejected with an error comment as you see fit, but they should never cause your program to go into an infinite loop. If your program accepts a valid but not strictly correct file, it should produce “reasonable” output. (For example, if a hex byte code is more than 8 bits long, then its reasonable to output only the 8 low-order bits.)

You may use the function `fatal()` defined in `util.h` to report errors. The arguments to `fatal()` are the same as for `printf()`: a format string followed by data values corresponding to any data-conversion specifications in the format. `fatal()` prints its error comment to `stderr` followed by a new line and then terminates the program.

To use `fatal()` in your program, you will need to `#include util.h` in your source file. You will also need to modify your `Makefile` to compile `util.c` and to link in `util.o` with your object code. The files `util.c` and `util.h` are located on the Zoo in `/c/cs223/assignments/ps3`.

3.2 Extra credit

For extra credit, write another program `unodx2` to perform the inverse of `od -v -Ad -tx2`. Do not try this until you have `unodx1` finished. It’s not completely straightforward and requires use of `union` or pointer casts to handle the byte order problem, and it requires that you implement some sort of read-ahead scheme for the input or buffering of the output to prevent the last byte of each line from being written out until you know for sure that it belongs in the file and isn’t just a filler byte.

4 Program Details

You can use `scanf()` with format specifier `%hhx` to read hex number into an `unsigned char`, and `%hx` to read a hex number into a `short unsigned int`.

To do line-at-a-time processing, you can read a line into a buffer using `scanf()` with format specifier `%num[^\n]`, where `num` is a decimal number that is one less than the size of the buffer.

Thus, if the line buffer is size 100, then the format would be `%99[^\n]`. This reads until a new line character is encountered or until a total of 99 characters have been read. It then puts a null character `'\0'` into the buffer.

Once the line is in a buffer, you can read it and convert numbers from it by using `sscanf()`. `sscanf()` takes a string as its first argument but otherwise works just like `scanf()`. When using multiple calls on `sscanf()` to process a single input line, you need to know how much of the string each call has read. This can be obtained by putting a `%n` format specifier at the end of the format. `%n` doesn't read any characters but instead stores the number of characters read so far through the last argument, which must be a pointer to `int`.

5 Testing

I will supply a files `mystery.dump1` and `mystery.dump2` which, when decoded by `unodx1` and `unodx2`, respectively, should give intelligible output. You should prepare a set of test files for each program to exercise the various special cases that your program might encounter, for example, a file that has no partial line, a file with a partial line, a file of length 0, a file of odd length, a file of even length, and so forth. You should also include in your test suite invalid files to trigger each of your error messages and files that contain as many different kinds of errors as you can think of, e.g., truncated files, files with garbage on the lines, files with too many numbers on a line, files with improper byte positions, and so forth. You should use the suffix `.dump1` on the names of test data files intended as input to `unodx1`. If you do the extra credit part, you should similarly use `.dump2` for files intended for `unodx2`.

6 Submission

You should name your source code file `unodx1.c`. You should provide a `Makefile` that will build `unodx1` in response to either `make` or `make all` and that will delete the `.o` and executable files in response to `make clean`. You should submit your source files, `Makefile`, test input files, and the output produced by your program on your test data.

If you do the extra credit part, your `Makefile` should build both `unodx1` and `unodx2` in response to `make` or `make all`. Your submission should then of course also contain the necessary source code, test files, and output for the extra credit part.