Convert string to hexadecimal on command line

I'm trying to convert "Hello" to `48 65 6c 6c 6f` in hexadecimal as efficiently as possible using the command line.

I've tried looking at `printf` and google, but I can't get anywhere.

Any help greatly appreciated.

Many thanks in advance,

linux string command-line hex printf

edited Feb 10 '14 at 10:30 TMS 30.9k 33 154 282

asked Jul 22 '11 at 14:50 Eamorr 5,005 29 97 182

What operating system? There are lots of "command lines"... – Nemo Jul 22 '11 at 14:53

Have you looked in here: commandlinefu.com/commands/view/6066/… and here: stackoverflow.com/questions/2103698/…? – Mrchief Jul 22 '11 at 14:54

@Nemo I'm on Ubuntu Linux... – Eamorr Jul 22 '11 at 15:05

2 Answers

```
echo -n "Hello" | od -A n -t x1
```

Explanation:

- The `echo` program will provide the string to the next command.
- The `-n` flag tells echo to not generate a new line at the end of the "Hello".
- The `od` program is the "octal dump" program. (We will be provide a flag to tell it to dump it in hexadecimal instead of octal.)
- The `-A n` flag is short for `--address-radix=n`, with n being short for "none". Without this part, the command would output an ugly numerical address prefix on the left side. This is useful for large dumps, but for a short string it is unnecessary.
- The `-t x1` flag is short for `--format=x1`, with the x being short for "hexidecimal" and the 1 meaning 1 byte.

edited May 27 '16 at 16:38 James 318 3 13

answered Jul 22 '11 at 14:57 TMS 30.9k 33 154 282

Hey, that works really well. Do you, by any chance, know of a way to get rid of the spaces??? – Eamorr Jul 22 '11 at 15:06

2 echo -n "Hello" | od -A n -t x1 | sed 's/^ *//' – TMS Jul 22 '11 at 15:15

I believe you meant echo -n "Hello" | od -A n -t x1 | sed 's/ *//g' – ndvo Feb 14 at 10:57

Evidentally, @TMS can't RTFM. (see his comments to his previous answer)

If you want to do this and remove the spaces you need:

```
echo -n "Hello" | od -A n -t x1 | sed 's/ *//g'
```

The first two commands in the pipeline are well explained by @TMS in his answer, as edited by @James. The last command differs from @TMS comment in that it is both correct and has been tested. The explanation is:

- `sed` is a **s**tream **ed**itor.
- `s` is the **s**ubstitute command.
- `/` opens a regular expression - any character may be used. `/` is conventional, but inconvenient for processing, say, XML or path names.
- `/` or the alternate character you chose, closes the regular expression and opens the substitution string.
- In `/ */` the `*` matches any sequence of the previous character (in this case, a space).

- `/` or the alternate character you chose, closes the substitution string. In this case, the substitution string `//` is empty, i.e. the match is deleted.
- `g` is the option to do this substitution **g**lobally on each line instead of just once for each line.
- The quotes keep the command parser from getting confused - the whole sequence is passed to `sed` as the first option, namely, a `sed` script.

@TMS brain child (`sed 's/^ *//'`) only strips spaces from the beginning of each line (`^` matches the beginning of the line - 'pattern space' in `sed` -speak).

If you additionally want to remove newlines, the easiest way is to append

```
| tr -d '\n'
```

to the command pipes. It functions as follows:

- `|` feeds the previously processed stream to this command's standard input.
- `tr` is the **tr**anslate command.
- `-d` specifies deleting the match characters.
- Quotes list your match characters - in this case just newline (`\n`). Translate only matches single characters, not sequences.

`sed` is uniquely retarded when dealing with newlines. This is because `sed` is one of the oldest `unix` commands - it was created before people really knew what they were doing. Pervasive legacy software keeps it from being fixed. I know this because I was born before `unix` was born.

The historical origin of the problem was the idea that a newline was a line separator, not part of the line. It was therefore stripped by line processing utilities and reinserted by output utilities. The trouble is, this makes assumptions about the structure of user data and imposes unnatural restrictions in many settings. `sed` 's inability to easily remove newlines is one of the most common examples of that malformed ideology causing grief.

It is possible to remove newlines with `sed` - it is just that all solutions I know about make `sed` process the whole file at once, which chokes for very large files, defeating the purpose of a stream editor. Any solution that retains line processing, if it is possible, would be an unreadable rat's nest of multiple pipes.

If you insist on using `sed` try:

```
sed -z 's/\n//g'
```

`-z` tells `sed` to use nulls as line separators.

Internally, a string in `C` is terminated with a null. The `-z` option is also a result of legacy, provided as a convenience for `C` programmers who might like to use a temporary file filled with `C` -strings and uncluttered by newlines. They can then easily read and process one string at a time. Again, the early assumptions about use cases impose artificial restrictions on user data.

If you omit the `g` option, this command removes only the first newline. With the `-z` option `sed` interprets the entire file as one line (unless there are stray nulls embedded in the file), terminated by a null and so this also chokes on large files.

You might think

```
sed 's/^/\x00/' | sed -z 's/\n//' | sed 's/\x00//'
```

might work. The first command puts a null at the front of each line on a line by line basis, resulting in `\n\x00` ending every line. The second command removes one newline from each line, now delimited by nulls - there will be only one newline by virtue of the first command. All that is left are the spurious nulls. So far so good. The broken idea here is that the pipe will feed the last command on a line by line basis, since that is how the stream was built. Actually, the last command, as written, will only remove one null since now the entire file has no newlines and is therefore one line.

Simple pipe implementation uses an intermediate temporary file and all input is processed and fed to the file. The next command may be running in another thread, concurrently reading that file, but it just sees the stream as a whole (albeit incomplete) and has no awareness of the chunk boundaries feeding the file. Even if the pipe is a memory buffer, the next command sees the stream as a whole. The defect is inextricably baked into `sed` .

To make this approach work, you need a `g` option on the last command, so again, it chokes on large files.

The bottom line is this: don't use `sed` to process newlines.

answered Oct 15 '17 at 21:12

I_always_RTFM_and_STFW

11 1

OMG you are so awesome... – TMS Jan 18 at 13:55