

# Assignment 1

cpe 357 Fall 2020

## Note

This initial assignment is intended as a warm up to get everyone started quickly in this class. It shouldn't cause anyone too much pain, but it should knock off some of the rust and reduce the shock to the system brought on by starting a new term. It should not be taken as an indication of the difficulty of the other assignments in the course.

C Code.

C Code Run.

Run, Code, RUN!

PLEASE!!!

— /usr/games/fortune<sup>1</sup>

Due by 11:59pm, Wednesday, September 23rd.

This assignment is to be done individually.

See the Instructions for Homework Submission for information on how to use `handin`.

## Program: detab

(This is exercise 1-20 in Kernighan and Ritchie with more explanation.)

Write a program called `detab` that replaces tabs in its input with the proper number of blanks to space to the next tab stop. For the purpose of this assignment, assume tab stops are every 8 spaces.

Your program should work as a filter, taking its input from the standard input (stdin) and writing its output to the standard output (stdout).

This program should do the same thing as `expand(1)`, so you can test your program by comparing its output to the output of `expand` on the same file, as in the following example. (If you are not familiar with IO redirection, see the FAQ entry).

`diff(1)` is a program that reports differences between two files. If it says nothing, the files are the same.

```
% gcc -o detab -Wall detab.c
% ./detab < TestInput > detab.out
% ~pn-cs357/demos/detab < TestInput > reference.out
% diff detab.out reference.out
%
```

---

<sup>1</sup>Berkeley Unix (BSD) distributions have traditionally included a collection of games stored in `/usr/games`. `fortune(1)` prints a randomly selected adage ranging from the humorous to the truly strange. Sadly, these days, many system administrators choose not to install `/usr/games` in order to save space or to encourage the doing of actual work.

Since `diff(1)` reported no differences, the files must be the same. If there had been differences `diff` would have displayed the lines that differ between the two files.

In the compile-line above, I used two switches for `gcc`. The first, `-o detab`, tells the compiler to name the output “detab” rather than “a.out”. The second switch, `-Wall`, turns on verbose warnings for “shaky” C programming practices. This catches many common programming errors. All the code you write in this class should compile cleanly with `-Wall` turned on.

## Further Discussion

`detab` is supposed to read text from its input (stdin) and copy it to its output (stdout), expanding tabs as it goes. Some confusion tends to arise when it comes to the meaning of “expanding tabs.”

A tab character causes the cursor to move to the next tab-stop. For this program, tab stops are defined to be every eight columns, so a tab will move the cursor from wherever it is to the next column that is a multiple of 8 (if you start counting from 0, or  $8n + 1$  if you start at 1). The result is that, depending on where it is typed, a tab can expand to anywhere between 1 and 8 spaces.

Because both tabs and spaces are invisible, it’s hard to give an example here, but if we write tab as “>” and blank spaces as “\_”, you can see the effect of `detab` in Figure 1.

Input	After Expansion
>after_tab	_____after_tab
0>after_tab	0_____after_tab
01>after_tab	01_____after_tab
012>after_tab	012_____after_tab
0123>after_tab	0123_____after_tab
01234>after_tab	01234_____after_tab
012345>after_tab	012345_____after_tab
0123456>after_tab	0123456_____after_tab
01234567>after_tab	01234567_____after_tab

Figure 1: `detab` in action

If you have any doubt about what your program is doing relative to what it should do, check its output against the output of `/usr/bin/expand` on the same input file. They should do the same thing.<sup>2</sup>

## Tricks and Tools

Remember, this assignment is fairly straightforward, but there are a few tricky edge cases to watch out for. (E.g., one should not be able to backspace beyond the left margin.) Pay close attention to these. Also, be careful not to impose artificial limits on the program’s functionality. There is no reason, for example, to limit either line length or file size. There are a few library routines that may help with implementing `detab` listed in Figure 2.

Note also that any character that changes the column could have an effect on your tab-stop calculations. Other than “ordinary” letters, the ones of note are tab (`'\t'`), backspace (`'\b'`), newline (`'\n'`), and carriage return (`'\r'`).

<sup>2</sup>**Note:** the behavior of `expand(1)` appears to have changed with respect to carriage return (`'\r'`). Compare your output to that from my demo version, `~pn-cs357/demos/detab`.

<code>getchar(3)</code>	read a character from stdin
<code>putchar(3)</code>	write a character to stdout
<code>getc(3)</code>	read a character from the specified stream
<code>putc(3)</code>	write a character to the specified stream
<code>gets(3)</code>	never use this function. Not ever.
<code>feof(3)</code>	test to see if a given stream has encountered an end of file

Figure 2: Some potentially useful library functions

## Coding Standards

See the pages on coding standards on the cpe 357 class web page.

## What to turn in

Submit via `handin` in the CSL to the `asgn1` directory of the `pn-cs357` account:

- Your well-documented source file, called **detab.c**.
- A README file that contains:
  - Your name, including your login name(s) in parentheses (e.g. “(pnico)”).
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

The README file should be **plain text**, i.e, **not a Word document**, and should be named “README”, all capitals with no extension.

## Sample runs

I would generally put some sample runs below, but it’d be silly because in printed form the tabs and spaces would look exactly the same. I will place an executable version of `detab` in `~pn-cs357/demos` so you can run it yourself and compare your output with `diff(1)`.