Assignment 1 Solutions

cpe 357 Fall 2023

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[1]

Due by 11:59pm, Monday, October 9th.
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
%
```

[1] Berkeley Unix (BSD) distrubutions 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.[2]

Tricks and Tools

Remember, this assignment is fairly straightforward, but it 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[3] 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'`).

[2] **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`.

[3] Ok, I'll let you assume that your column number will fit in the range of an `int`.

`getchar(3)`	read a character from stdin
`putchar(3)`	write a character to stdout
`getc(3)`	read a character from the specified stream
`putc(3)`	write a character to the specified stream
`gets(3)`	never use this function. Not ever.
`feof(3)`	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)`.

I have also published a partial test harness in `~pn-cs357/demos/tryAsgn1`. When you run this, it will try to compile your program and try running it on a few sample inputs.

Also, **and this comes with absolutely zero support,** `tryAsgn1` is just a shell script meaning you can read it. Why would you want to beyond curiosity? It could lead you to the path where the actual test inputs are.

Solution:

```
#
# This Makefile is more complex than the ones I expect you to generate,
# but it gives an example of some of the things you can do with one.
# In particular, note the targets "all", "clean", and "test".  These
# don't correspond to actual files to create, but provide extra
# functionality.
#

SHELL  = /bin/sh

CC     = gcc

CFLAGS = -Wall -ansi -pedantic

MAIN   = detab

OBJS   = $(MAIN).o

TESTDIR   = ./inputs

TESTINPUTS = $(TESTDIR)/*

all:    $(MAIN)

clean:
	rm -f $(OBJS) *~

$(MAIN): $(MAIN).o
	$(CC) $(CFLAGS) -o $(MAIN) $(OBJS)

detab.o: detab.c
	$(CC) -c $(CFLAGS) detab.c

#
#  It's generally a good idea to have a "test" target to test your
# program.  This one uses the Bourne shell(/bin/sh) for loop
# to test the program on a number of inputs.  For each test,
# it uses the exit code of diff (the value its main() returns) to
# determine whether the test succeeded.
#   This also demonstrates the Bourne shell's conditional syntax.
#
#   For make, each command can only be a single line.  The backslashes
# below allow many real lines to be treated as one logical line.
#
#   All the details you want (and more) about /bin/sh programming can
# be found in the manual.
#

test: $(MAIN)
	@echo "Testing $(MAIN):"
	@cd $(TESTDIR); for input in $(TESTINPUTS) ; do \
	  if [ `uname -s` = "SunOS" ]; then             \
	    echo "         $$input...\c";              \
	  else                                         \
	    echo -n "         $$input...";             \
	  fi;                                          \
	  expand  < $$input > expand.output;           \
	  $(PWD)/$(MAIN) < $$input > detab.output ;          \
	  if diff expand.output detab.output > /dev/null ; then \
	    echo "ok.";                                  \
	  else                                          \
	    echo "FAILURE." ;                             \
	  fi;                                           \
	  rm -f expand.output detab.output;                    \
	done
```

```
/*
 * detab.c is an implementation of a subset of expand(1).  Detab
 * copies stdin to stdout expanding tabs to the next tabstop.
 * Tabstops are defined by the macro TAB_WIDTH.
 *
 * -PLN
 *
 * The below revision history is provided automatically by the
 * revision control system (RCS).  rcs(1) allows you to keep track of
 * previous versions of a file in order to track changes (or recover
 * from bonehead mistakes).
 *
 * Revision History:
 *
 * $Log: detab.c,v $
 * Revision 1.5  2003-04-14 12:53:48-07  pnico
 * kogorman added that \r should return to column zero,
 * and \b should not back up to before column zero
 *
 * Revision 1.4  2002-01-22 15:03:54-08  pnico
 * changed over to a case-based structure and added support for backspace
 *
 * Revision 1.3  2002-01-12 11:42:15-08  pnico
 * fixed the second bonehead error of forgetting to
 * increment the column when printing each character
 *
 * Revision 1.2  2002-01-12 11:12:21-08  pnico
 * fixed the traditional off-by-one error brought on by counting
 * the first column, but neglecting to print the first space when
 * expanding tabs
 *
 * Revision 1.1  2002-01-12 10:39:15-08  pnico
 * Initial revision
 *
 *
 */
#include <stdio.h>

#define TAB_WIDTH 8

int main(int argc, char *argv[]) {
  int c;                    /* the character */
  int col;                  /* keep track of the current column */

  col=0;                     /* start at column 0 */
  while ( (c=getchar()) != EOF ) {
    switch ( c ) {
    case '\t':              /* if it's a tab, expand it */
      /* to expand the tab, we continue until the column is an
        * integral multiple of TAB_WIDTH.  A tab always produces
        * at least one space, so the do-while and pre-increment
        * make sure the place we stop will be the _next_ tab-stop
        * in case we're already at one.
        */
      do {
        putchar(' ');
      } while( (++col % TAB_WIDTH) != 0);
      break;
    case '\n':/*fall through*/  /* if c was newline, we're back at column 0 */
    case '\r':                  /* carriage return also puts us back at 0   */
      col = 0;
      putchar(c);
      break;
    case '\b':                  /* print backspaces, but decrement col */
      if (col)                 /* (but only so far as column zero      */
        col--;
      putchar(c);
      break;
    default:                    /* otherwise, just print it and count up */
      col++;
      putchar(c);
      break;
```

```
    }
  }

  return 0;                    /* everything's ok... */
}
```