Laboratory Exercise 2

Solutions

cpe 357 Fall 2023

Due date TBA.
The Written Exercises (problems) are to be done individually.
The Laboratory Exercises are also to be done individually.

This looks kind of long, but none of the tasks below are terribly large.

Problems

These problems are designed to provoke some thought (could there be a quiz coming?):

1. (Warm up) Please provide declarations for the following data:
   (a) a pointer *cp* that points to a `char`.
   Solution

   *char *cp;*

   (b) a pointer *ap* that points to an array of `char`s.
   Solution

   *char *ap;*

   (c) a pointer *pp* that points to a pointer that points to an `int`.
   Solution

   *int **pp;*

2. Is it possible in C to declare and initialize a pointer that points to itself? Why or why not? (And, if so, *how,* of course.)
   Solution

   Sure, but you won't be able to get it to typecheck (because the pointer to whatever type will be a pointer to that type). You can do it with `void *`*, though:*
   *void *p = &p;*

3. What is the fundamental problem with the following code fragment intended to print out a string:

```
char s[] = "Hello, world!\n";
char *p;
for(p = s; p != '\0'; p++)
  putchar(*p);
```

   What will happen when this is executed? How can it be fixed?
   Solution

   It'll crash, because it's comparing the pointer to `'\0'`*, not the pointee.*

4. C programmers often say "arrays are the same as pointers". In one sense this is true. In another, more correct, sense they are fundamentally different.

   (a) In what ways is this statement correct?

   (b) How is it in error? That is, what makes a pointer fundamentally different from an array?

   Solution

   (a) Both pointers and arrays can be indexed. The name of an array is the address of the first element of the array.

   (b) A pointer can be an lvalue. (it can be changed) An array name is a constant.

5. Exercise 1.3 from Stevens APUE.

   Solution

   The argument to `perror(3)` *is a pointer. The* `const` *attribute guarantees that the called routine will not change the characters to which it points, guaranteeing that* `perror(3)` *will not alter the string passed to it. The parameter of* `strerror(3)` *is an* `int`. *Since C is call-by-value, all* `strerror(3)` *gets is a copy, so the caller already knows that* `errnum` *will not be altered.*

6. Exercise 1.4 from Stevens APUE, 3rd ed. (Ex. 1.5 in the 2nd.)

   Solution

   Time on a UNIX *machine is counted as the number of seconds since the Epoch, midnight January 1st, 1970. A 32-bit integer can represent numbers from* -2^{31} *to* $2^{31}-1$.

$$\begin{aligned} 2^{31}-1 seconds &= 2147483647.00 seconds \\ &\approx 335791394.12 minutes \\ &\approx 596523.24 hours \\ &\approx 24855.13 days \\ &\approx 68.05 years\ (w/leap\ years) \end{aligned}$$

   So, $1970 + 68.05$*years gives us: January 18th, 2038, around 3am.*

7. A subcase of Ex. 2.2 from Stevens: On `unix5`, what is the actual data type of a `size_t` (the type of `malloc(3)`'s argument)? In what header file is it defined?

   Solution

   The answer depends on various configuration parameters, but you'll find that it's either an `unsigned int` *or an* `unsigned long`.

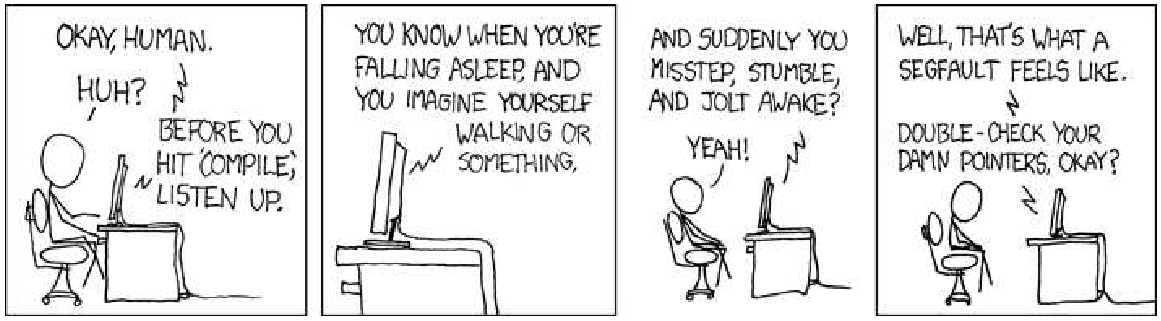
OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Figure 1: `http://xkcd.com/371/`

Laboratory Exercises

This hodgepodge of laboratory exercises designed to provide you with useful tools.

1. To help me with sorting out who's who in the class (and three years from now when you write me out of the blue asking for a recommendation) I am asking you to submit a digital photo of yourself to the `lab02` directory of the `pn-cs357` account. I will not be sharing these with anybody so it doesn't have to be glamorous, but it does need to be recognizable. That is, that great photo from last Halloween of you in your mummy costume really isn't the thing for this lab.

   Please make the root of the file name your login name. E.g., mine would be `pnico.jpg`.

2. GDB Tutorial.

   Read and understand the *Quick Introduction to GDB* linked from the main cpe357 web page.

   Once you have done this, write a small program, compile it, and run it within gdb. At the very least, you should

   (a) set a breakpoint by function name,
   (b) print the value of some variable in the function,
   (c) examine a backtrace to see how the function was called, and
   (d) step through a loop.

   You might want to try this on the program from step 4.

3. Learn to use `make(1)`.

   You'll want to do this in conjunction with step 4.

   `make(1)` is a wonderful build-management tool that will help you keep your programs up to date while minimizing compile time by only recompiling those components for which it is necessary[1]. Besides, there is a direct benefit to this because every assignment from here on out will require you to submit a functional Makefile.

   Your task:

[1]That is, it will do that if you set up your dependencies correctly. If you get those wrong, you can be in for a world of hurt.

(a) Read the *Quick notes on make* from the cpe357 web page.

(b) Read and study the sample makefiles published with the solutions to Asgn1, also linked from the cpe357 page.

(c) Write a small program—the one from step 4—and create a `Makefile` for it. This makefile must compile uniq when no argument is given to make, and support the following targets:

all	Build the program.
uniq	Build the program.
test	Builds the program and runs it with a sample input. For example, it could run your uniq on a test file and compare the result to that produced by `/bin/uniq`
clean	Removes all non-essential files generated during the build. Generally this means the intermediate object (.o) files.

Your goals when writing this makefile should be to minimize the amount of duplication. That is, `all`, `uniq`, and `test` should not each have separate instructions for building the program. You should also make sure to include appropriate dependency information so that if one of your source files changes `make` will only recompile those files that need to be recompiled.

Bonus: Now—and only now—look into the man page for `makedepend(1)`, but be warned: `makedepend` will include all the system files, too, making makefiles that are not portable. Clever application of the `-Y` option can get around this.

4. Program: `uniq`

This program is an exercise with dynamic data structures as a warm-up for Assignment 2.

Write a version of the unix utility program `uniq(1)`. This program will act as a filter, removing adjacent duplicate lines as it copies its stdin to its stdout. That is, any line that is identical to the previous line will be discarded rather than copied to stdout.

Your program may not impose any limits on file size or line length.

To get started, I highly recommend writing a function `char *read_long_line(FILE *file)` that will read an arbitrarily long line from the given file into newly-allocated space. Once you have that, the program is easy.

Be careful to free memory once you are done with it. A memory leak could be a real problem for a program like this.

Note: even a utility as simple as this has decisions that must be made and specifications that must be negotiated. For example, a line that does not end with a newline is considered the same as one that does. From the specification[2]:

Also Note: For this, you may not use `getline(3)` or equivalent, since the whole point of the exercise is to learn to do dynamic memory management.

> "The *uniq* utility shall read an input file comparing adjacent lines, and write one copy of each input line on the output. The second and succeeding copies of repeated adjacent input lines shall not be written. The trailing <newline> of each line in the input shall be ignored when doing comparisons.
>
> "Repeated lines in the input shall not be detected if they are not adjacent."

[2]`https://pubs.opengroup.org/onlinepubs/9699919799/utilities/uniq.html`

Tricks and Tools

There are some library routines with which you might want to be familiar before working on `uniq` listed in Figure 2.

`strlen(3)`	calculate the length of a string
`strcmp(3)`	compare two strings
`fgets(3)`	read a string into a buffer (may or may not be of use to you here)
`malloc(3)`	allocate a given number of bytes of memory from the heap
`realloc(3)`	Change the size of a previously allocated chunk of memory
`free(3)`	return a malloc()ed region of memory to the heap

Figure 2: Some potentially useful commands and library functions

What to turn in

For the Written Problems: Include then in the a README file described below.

For the Laboratory Exercise: Submit via `handin` to the `lab02` directory of the `pn-cs357` account:

- your picture, named after yourself,
- your well-documented source file(s) for `uniq`, and
- A makefile (called `Makefile`) that will build `uniq` from your source when invoked either with no target or with the target "`uniq`".
- A README file that contains:
  - Your name(s). In addition to your names, please include your Cal Poly login names with it, 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 answers to the written questions above.

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

Solution:

File	**Where**
Makefile	p.6
readline.h	p.7
readline.c	p.8
uniq.c	p.9

```
MAIN=uniq

CC = gcc

CFLAGS = -g -Wall

OBJS = $(MAIN).o readline.o

$(MAIN): $(OBJS)
	$(CC)  −o $(MAIN) $(OBJS)

clean:
	rm −f $(OBJS) core *~
```

```
#ifndef READLINEH
#define READLINEH

#include <stdio.h>

extern char *readline(FILE *);
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

#define CHUNK 80

extern char *readline(FILE *infile) {
  /* Read a string from given stream.  Returns the string, or NULL if
   * EOF is encoutered.  Approach: allocate a buffer of size CHUNK,
   * and as the string grows expand the buffer as necessary
   */
  int i;
  char *buff;
  char *ret;
  int size=0;
  char c;

  size=CHUNK;
  if(NULL==(buff=(char*)malloc(size * sizeof(char)))) {
    perror(_FUNCTION_);
    exit(−1);
  }
  for(i=0,c=getc(infile); c!=EOF ;c=getc(infile)) {
    if(i>=size−1) {     /* buffer is too small, expand it. */
      size+=CHUNK;
      if(NULL==(buff=(char*)realloc(buff,size * sizeof(char)))) {
          perror(_FUNCTION_);
          exit(−1);
      }
    }
    if ( c == '\n' )
        break;
    buff[i++]=c;
  }

  if ( i || c=='\n' ) {          /* if there was a string read, copy it
                                  * into a new buffer.  Otherwise, return
                                  * NULL to signal EOF
                                  */
      buff[i]='\0';     /* final nul */
      if(NULL==(ret=(char*)malloc(i+1))) {
          perror(_FUNCTION_);
          exit(−1);
      }
      strcpy(ret,buff);
  } else {
      ret = NULL;
  }

  free(buff);
  return ret;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "readline.h"

int main(int argc, char *argv[]){
    /* read lines from stdin until there are no more lines.  For each line,
     * compare it to the previous line.  If they are the different, print
     * the previous line.  If the same, discard the previous line.
     */
    char *last, *next;

    last = readline(stdin);     /* read an initial line */

    /* now, keep reading lines until there are no more lines */
    while ( (NULL != last) && (NULL != (next=readline(stdin)))) {
        if ( strcmp(last, next) ) { /* print the old line if different */
            puts(last);
        }
        free(last);             /* we're done with last now */
        last = next;
    }

    if ( last )                 /* print the last line if there is one */
        puts(last);

    return 0;
}
```