CSC 352, Fall 2015
Assignment 7
Due: Friday, October 23 at 23:59:59

The Usual Stuff

All the usual stuff on assignments applies to this one, too: make an `a`*N* symlink, use our `gcc` alias, use the Tester, the Tester runs `gcc` first, etc.

Refer to previous write-ups if you need a refresher.

This Assignment Has Restrictions

To focus you on the elements of C that we have studied, THE FOLLOWING RESTRICTIONS APPLY TO ALL SOLUTIONS for this assignment:

- **Pointer variables may not be used. We'll be talking about them soon but in short, their declaration has an asterisk, like `int *p`. (Yes, with a function such as `f(int a[])`, `a` is essentially a pointer, but that's ok.)**

- **The only library routines that may be used are `strlen`, `strcpy`, `strcat`, `printf`, `getchar`, and `putchar`. (See below re "C Library String Routines".)**

Problem 1. (3 points) `gdb.txt`

This problem is essentially a "lab". Its purpose to encourage you to get up and running with `gdb`. Work through the steps below and then copy/paste a transcript of your full session with `gdb` into `gdb.txt`.

1. Start by copying `a7/gdb1.c` into your `a7` directory and compiling it into an executable named `a.out`. Start up `gdb` on `a.out`.
2. Using the `list` command one or more times, display all the code in both `gdb1.c` and `strfuncs.h`, too. (Note that it's relatively uncommon to put source code for functions in a header file, but it's convenient in this case.)
3. Set a breakpoint in `main`, and then do `run`.
4. Using `next`, step over the declaration of `s`.
5. Examine `s` using both "`p s`" and "`p/c s`".
6. Using a single `p` (print) command and an artificial array, examine `s` and the ten chars on each side of `s` (120 characters in all).
7. Use `s` (`step`) to step into the call of `copy` at line 7. Use `where` to see where you are, then print the values of `i`, `from`, and `to`.
8. Using `next` once, and then ENTER to repeat that last command, `next`, a number of times, run the `while` loop to completion, examining `to` one or more times. You'll eventually get back to `main`.
9. Back in `main`, use `next` to step over the first call to `concat` and then use `step` to step into the second call to `concat`.
10. Do some stepping through the loops in `concat`, and then use `finish` to return to

`main`.

11. Step over the `printf`, causing its output to appear
12. That's it! Type `q` to exit `gdb`.

Note: We won't be exacting when grading your `gdb.txt`; the purpose of this problem is just to be sure that everybody has run `gdb` and has tried a wide variety of commands. If your `gdb.txt` shows that you put forth a good effort to work through the steps above, you'll get full credit. We'll be happy to take a look at your `gdb.txt` before it's due.

Don't bother with editing errors out of your `gdb.txt`. It's educational for me and the TAs to see what sort of mistakes you make with `gdb`.

Problem 2. (14 points) `edit.c`

Text editors come in all sizes. Emacs and Vim are big editors. In this problem you are to write a tiny text-editing function named `edit`. Here's the prototype: `void edit(char s[], char ops[])` `edit` applies the sequence of operations in `ops` to the string `s`.

Each operation is designated by a single character:

`=`	Move to the next character.
`#`	Delete the current character.
`&`	Duplicate the current character.
`~`	Swap the current character and the next character.
`|`	("or"-bar) Discard the rest of the string, including the current character.

Here is a test program that prompts for a string to operate on, and then repeatedly prompts for operations, applies them, and prints the resulting string.

```
#include <stdio.h>

int main()
{
    char line[100];

    printf("String? ");
    gets(line);

    char ops[100];
    while (printf("Ops? ") && gets(ops)) {
        edit(line, ops);
        puts(line);
        }
}
```

An example of operation follows, accompanied by commentary.

```
String? abcdef
Ops? #       (delete first character: a)
bcdef
Ops? =#==#   (skip b, delete c, skip de, delete f)
bde
Ops? =&      (skip b, duplicate d)
bdde
Ops? &====&  (duplicate b, skip bbdd, duplicate e)
bbddee
Ops? =~      (skip b, swap bd)
bdbdee
Ops? ==|     (skip bd, delete the rest)
bd
```

Note that the operations are cumulative on `line`—we start with `abcdef` and end with `bd` after six calls to `edit`, but the "cursor" starts at 0 on each call to `edit()`. Also note that the only operation that advances the "cursor" is =.

It's important to recognize that the above interaction with the test program is equivalent to this sequence of code:

```
char line[100] = "abcdef";

edit(line, "#");
edit(line, "=#==#");
edit(line, "=&");
edit(line, "&====&");
edit(line, "=~");
edit(line, "==|");
```

`edit` ignores operations that have no meaning, such as doing more deletes (#) or moves (=) than there are characters, or a swap (~) when positioned on the last character of the string. Unknown characters appearing in the operation string are silently ignored. `edit` assumes `s` has enough space.

Implementation notes

It is important to understand that **what you are to write is a function, not an entire program**. The file `edit.c` that you submit for grading will probably look something like this:

```
#include <string.h>

void edit(char s[], char ops[])
{
    ...your code...
}
```

If you wish to write "helper" functions to simplify `edit`, that's fine. `See a7/edit-hints` for the two helpers I use.

Note that there is no `main` in `edit.c`. To combine your implementation of `edit` with the test program, and produce an executable named `edit1`, do this:

```
% gcc -o edit1 edit.c a7/edit1.c
```

The Tester does the same thing for `edit`: it combines your `edit.c` with `a7/edit1.c` and produces an executable named `edit1`.

A reference version of `edit1`, which is `a7/edit1.c` combined with my version of `edit.c`, is in `a7/edit1`. Experiment with `a7/edit1` to help clarify your understanding of the operations.

If you look at `a7/edit1.c`, and I recommend you do study it, you'll see this `main`:

```
int main()
{
    if (isatty(0))
        tty_main();
    else
        no_tty_main();
}
```

The C library function `int isatty(int fildes)` is used to determine whether standard input ("file descriptor" zero) is the keyboard. (Do `man isatty`.) If so, `tty_main()` is called. It is essentially the main program shown above—it prompts the user for input. If standard input is not a terminal, which implies that standard input is a file or a pipe, `no_tty_main()` is called. It does not prompt, but it does echo input lines so that the correspondence between input and output can easily be seen.

Observe the behavior of `edit1` when reading from a file (which is not a "tty"/terminal).

```
% cat a7/edit.1
abcdef
#
=#==#
=&
&====&
=~
==|
% a7/edit1 < a7/edit.1
Input:  abcdef
Ops:    #
Result: bcdef
Ops:    =#==#
Result: bde
Ops:    =&
Result: bdde
Ops:    &====&
Result: bbddee
Ops:    =~
Result: bdbdee
```

```
Ops:    ==|
Result: bd
```

Contrast that with the behavior when standard input is the keyboard:

```
% a7/edit1
String? abcdef
Ops? #
bcdef
Ops? =#==#
bde
Ops? ...and so forth...
```

Problem 3. (14 points) `cset.c`

The Icon programming language has a data type named "cset". A cset represents a set of characters. Like a mathematical set, a `cset` has no duplicates, and there is no notion of ordering.

In this problem you are to write several C functions that perform various manipulations on csets of ASCII characters.

A simple way to represent a cset in C that holds ASCII characters is with an array of 128 integers:

```
int set1[128] = {}; // 128 elements with no initializers = 128
0s
```

The presence of the ASCII character with code N is represented by a non-zero value in `set1[N]`. As declared above, `set1` is empty.

The characters '`a`', '`A`', and '`1`' can be added to the set by assigning `1` to the corresponding elements in the array:

```
set1['a'] = 1;  // equivalent: set[97] = 1;
set1['A'] = 1;
set1['1'] = 1;
```

The function `int cset_size(int cset[])` returns the number of elements in the cset:

```
printf("set1 has %d elements\n", cset_size(set1));
          // Output: "set1 has 3 elements"
```

A cset can be made from a string with `void str_to_cset(char str[], int cset[])`:

```
int set2[CSET_SIZE];  // CSET_SIZE is #defined in a7/cset.h
str_to_cset("A-Zccbaba0-9", set2);
```

Note that a range of characters can be specified using a *`FIRST-LAST`* notation similar to that

used by `tr(1)`, shell wildcards, and regular expressions.

The resulting cset, `set2`, has 39 characters: `A-Z`, `0-9`, and `a`, `b`, `c`. (Remember: no duplicates.) Note what `gdb` shows for `set2`:

```
(gdb) p set2
$1 = {0 <repeats 48 times>, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 1 <repeats 26 times>, 0, 0, 0, 0, 0, 0, 1,
1, 1, 0 <repeats 28 times>}
```

A specification such as "`-a`" or "`a-`" creates a set with two elements: '`a`' and '`-`'. Assume that the first character of a range is less than the ending character of a range—we won't test with something like "`9-0`".

Note that we cannot use `str_to_cset()` to create a cset that contains ASCII `NUL` (code 0). We might try to make a cset with two characters, a `NUL` and an `'a'`, like this: `str_to_cset("a\0");` but we'd find that the cset had only one member, `'a'`, because that `'\0'` would be interpreted as the end of the string!

The function `void cset_to_str(int cset[], char str[])` fills in `str` with a string that represents the characters in cset. Example, assuming `set2` from above:

```
char s[CSET_SIZE+1];
cset_to_str(set2, s);
printf("s = %s\n", s);
```

Output:

```
s = 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabc
```

The string produced by `cset_to_str` always orders characters by their ASCII code.

Just like `str_to_cset()` can't be used to create a cset that contains `NUL`, `cset_to_str()` isn't useful with a cset that contains a `NUL`: it would produce a string that begins with a `NUL`, which is effectively an empty string!

The function `void cset_complement(int to[], int from[])` produces a complemented copy of a cset. The cset `to` is a copy of the cset `from` that holds the characters not in `from`.

Example:

```
int set1[CSET_SIZE], set2[CSET_SIZE];

str_to_cset("0-9", set1);

cset_complement(set2, set1); // produce complement of set1 in
set2

printf("set1: %d chars\n", cset_size(set1));
```

```
printf("set2: %d chars\n", cset_size(set2));
```

Execution:

```
set1: 10 chars
set2: 118 chars
```

The function `void cset_strip(char s[], int cset[])` removes the characters in `cset` from the string `s`.

Example:

```
int vowels[CSET_SIZE];

str_to_cset("aeiou", vowels);

char s[] = "Some text to strip";
char s2[] = "Some text to strip";

cset_strip(s, vowels);
printf("s minus vowels: '%s'\n", s);
          // Output: s minus vowels: 'Sm txt t strp'

int not_vowels[CSET_SIZE];
cset_complement(not_vowels, vowels);
cset_strip(s2, not_vowels);
printf("s2 minus not-vowels: '%s'\n", s2);
          // Output: s2 minus not-vowels: 'oeeoi'
```

Implementation notes

It is important to understand that in this problem you are implementing a collection of related functions. The file you turn in, `cset.c` should look something like this:

```
#include "/cs/www/classes/cs352/fall15/a7/cset.h"

void str_to_cset(char s[], int cset[])
{
    ...
}

void cset_to_str(int cset[], char str[])
{
    ...
}

int cset_size(int cset[])
{
    ...
}

void cset_complement(int to[], int from[])
```

```
{
    ...
}

void cset_strip(char s[], int cset[])
{
    ...
}
```

The order of the functions is not important. As with `edit.c`, there is no `main` in `cset.c`. You may have helper functions in your `cset.c`, too.

Test programs based on the examples shown above are in `a7/cset[1-5].c`.

To test your implementation with one of the test programs, specify both the test program and your solution as `gcc` arguments:

```
% gcc a7/cset1.c cset.c
% a.out
set1 has 3 elements
```

Problem 4. (14 points) `path.c`

In this problem you are to write a function named `path_elem` that extracts a specified element from a UNIX path such as `/home/whm/352/args.java`.

A path that specifies a file can be considered to have three elements: a directory, a filename, and a filename extension. For example, the path `/home/whm/352/args.java` can be broken down as follows:

> Directory: `/home/whm/352`
> Filename: `args.java`
> Extension: `java`

The path "`data`" breaks down as follows:

> Directory: `.` (just a dot)
> Filename: `data`
> Extension: *(none)*

Here are the rules:

> Directory:
>
> Everything from the start of the path up to (but not including) the last slash, if any. If the path has no slash, the directory is "`.`". As a special case, a path such as "`/x`" has the directory "`/`".
>
> Filename:
>
> Everything to the right of the last slash, if any. If there is no slash, the whole path

is the filename.

Extension:

Everything to the right of the last dot in the filename. The filename "x.java.ok" has the extension "ok". The filename ".emacs.old" has the extension "old". If the only dot in a hidden file is the initial dot, the name has no extension. For example, ".emacs" has no extension. The name "..xy" has the extension "xy".

Assume that a path never contains two consecutive slashes ("x//x", for example) and that a path never ends with a slash ("/a/b/c/", for example).

Here is the prototype for path_elem:

```
void path_elem(char path[], char which, char result[])
```

The second parameter, which, is assumed to be one of 'd', 'n', 'e', indicating that the directory, name, or extension, respectively is to be found in path and copied to result.

path_elem never modifies the values in path. The behavior of path_elem is undefined if which is not one of 'd', 'n', or 'e'.

Here is an example of usage:

```
char p[] = "/home/whm/352/args.java";
char result[50];

path_elem(p, 'd', result);
printf("Directory: '%s'\n", result);

path_elem(p, 'n', result);
printf("     Name: '%s'\n", result);

path_elem(p, 'e', result);
printf("Extension: '%s'\n", result);
```

Output:

```
Directory: '/home/whm/352'
     Name: 'args.java'
Extension: 'java'
```

A copy of the above code can be found in a7/path0.c. a7/path1.c is a test program that reads paths from standard input and prints each element of each path. An executable is in a7/path1. a7/path1.[123] are text files that each contain a number of paths. Another good source of input for a7/path1 is "find -type f".

Implementation note

Just as for edit.c and cset.c, combine your solution with a test program by naming both on

the `gcc` command line:

```
% gcc -o path1 a7/path1.c path.c
% path1 < a7/path.1
...
```

Problem 5. (8 points) `csort.c`

Write a C program named `csort` that reads ASCII characters on standard input and sorts the input on a character by character basis.

Example:

```
% echo Testing | csort

Teginst% echo Testing | csort | od -tu1 | head -1
0000000  10  84 101 103 105 110 115 116
% echo pack my box with five dozen liquor jugs | csort

       abcdeefghiiijklmnooopqrstuuvwxyz%
```

Note the percent sign that appears at the end of the `csort` output is the `bash` prompt. The last character that is output by `csort` is the input character with the highest ASCII code.

Restriction: Your sorting algorithm may use no more than 1024 bytes of storage. Exceeding that limit will result in a score of zero for this problem.

You may assume that each ASCII character will appear no more than two billion times in the input; this in turn implies a maximum input size of 256 billion characters.

Hint: If you think problem is hard, you haven't thought about it enough.

Problem 6. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "`Hours:`". There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a7/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `aN.YYYYMMDD.HHMMSS.tz` You can run `a7/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `352f15` if you need to recover a file—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a7/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a7/delivs)
  60  135 1213 edit.c
  65  174 1092 cset.c
  56  137 1266 path.c
  13   29  203 csort.c
 194  475 3774 total

% for i in $(grep -v txt a7/delivs); do echo $i: $(tr -dc \; <
$i | wc -c); done
edit.c: 22
cset.c: 30
path.c: 27
csort.c: 5
```

There are no comments in my code.

Miscellaneous

This assignment is based on the material on C slides 1-246, plus whatever slides follow before the material on pointers begins. That will probably be around slide 255.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them and has completed the previous assignments will need 8-10 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. **If you put eight hours into this assignment and seem to not be close to completing it, it's probably time to touch base with us.** Specifically mention that you've reached eight hours. Give us a chance to speed you up! **Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)