CSC 352, Fall 2015
Assignment 8
Due: Friday, October 30 at 23:59:59

The Usual Stuff

All the usual stuff on assignments applies to this one, too: make an `aN` symlink, use our `gcc` alias, use the Tester, the Tester runs `gcc` first, etc. Refer to previous write-ups if you need a refresher.

Introduction

When writing code in C there is commonly a choice between writing code in an array-based style or a pointer-based style. Slides 309-313 show string manipulation routines written in both styles. In assignment 7 you solved several problems using an array-based style. Experienced C programmers often favor writing code using a pointer-based style than an array-based style. Also, some general wisdom about programing is that it's better to use iterators than it is to index through arrays. These lines of thinking somewhat intersect at the observation that a pointer in C can be thought of as a very simple iterator that specifies a position in a sequence of values.

One of the goals of this assignment is to help you shift into a pointer-based style of programming in C. The first problem, `warmup.c`, asks you to write several very simple functions using pointers. The next two problems ask you to rewrite `cset.c` and `path.c` from assignment 7 using pointers. Your first thought on those two might be to simply transliterate your assignment 7 solutions but, as the example at the bottom of slide 310 shows, that can lead to bad code.

To help you break out of an array-based mindset, those first three problems have restrictions. The final two problems have no restrictions whatsoever, but if the first three problems have been effective in rewiring your brain for pointers, you'll find yourself naturally using pointers on those last two problems.

Problems 1, 2, and 3 Have Restrictions

Instead of this two-page section on restrictions, I'd like to write just one sentence: "Use pointers as much as you possibly can on this assignment." However, experience tells me that not everyone will take that to heart, and then they'll not have the experience with pointers that they'll need to do well on later assignments, where fluency with pointers is essential.

To help you start thinking in terms of pointers rather than arrays and indices, **the first three problems (`warmup.c`, `cset.c`, and `path.c`) have two restrictions:**

1. **ALL VARIABLES MUST BE POINTERS, function parameters excepted.**
2. **Square brackets (`[` and `]`) may not appear in your solutions.**

Let's first talk about restriction 1, "**ALL VARIABLES MUST BE POINTERS, function parameters excepted.**"

Here are some declarations that are not allowed on problems 1-3:

```
int i;
char c;
```

```
int a[5];
double x[10];
```

The declarations above are not allowed because the variables they declare are not of pointer type.

Here are some declarations that are allowed:

```
char *p1;
int *p2;
double *p3;
```

You may be tempted to bypass the only-pointer-variables requirement by employing casts to essentially use pointer variables as integers, but that is not allowed! For example, consider the following version of a `sum` routine that declares `i` and `sum` as `char *`, but casts them to `int`s as needed.

```
int sum(int a[], int nelems)
{
    char *i, *sum = 0;

    for (i = 0; (int)i < nelems; i++)
        sum += a[(int)i];

    return (int)sum;
}
```

The routine works just fine, at least with `gcc` on lectura, but by using casts, the pointer variables `i` and `sum` are essentially being used as integers.

You are allowed to write helper functions that have non-pointer parameters as long as they are not used to bypass the restrictions. A helper routine I use in `path_elem` is `char *rpos(char *s, char c)`. It returns the address of the rightmost occurrence of `c` in `s`.

Below is an example of an example of a helper function that's **NOT ALLOWED**, because **it has an extra parameter that's really just a thinly-disguised local variable.**

```
int count(char c, char *s, int n)
{
    while (*s)
        if (*s++ == c)
            n++;

    return n;
}
```

`count` returns the number of occurrences of `c` in `s`. It might be used like this:

```
iprint(count('t', line, 0));
```

Observe is that the third argument initializes the parameter `n` to zero, and then `n` is used as a counter, much like declaring and initializing a local variable: "`int n = 0;`". The key point/red flag is this: The parameter `n` is not used to convey any necessary information to `count`.

In normal circumstances it would, of course, be perfectly reasonable to simply have `int n = 0;` in `count` but instead I'm asking you to contort things a little bit, to learn a little more: See if you can think of a way to write `count` by introducing another pointer variable or two, instead of that third parameter. My solution is in `a8/count-hint.c`.

In contrast to restriction 1, restriction 2 is dead simple: "**Square brackets (`[` and `]`) may not appear in your solutions.**" In effect, this restriction prevents you from treating pointers like arrays, and falling back into array-based programming.

Here's a statement that's prohibited by restriction 2—it contains square brackets:

```
if (p < &cset[CSET_SIZE]) ...
```

Instead, you might do this:

```
if (p < cset + CSET_SIZE) ... // OK
```

Violating either of the two restrictions may result in a score of zero for a problem. The TAs and I are happy to inspect your solutions prior to the deadline to be sure that you're in compliance—ask us during office hours or mail to `352f15`.

The script `a8/ckrestrict` does some simple-minded checking for compliance with the restrictions. Try it before requesting an inspection, but note that `ckrestrict`'s output is non-guaranteed—it's subject to false positives and, worse, false negatives. Here's a sample of its output on a clean run:

```
% a8/ckrestrict cset.c
Checking cset.c for non-pointer variables...
   none detected

Checking cset.c for casts...
   none detected

Checking cset.c for square brackets...
   none detected
```

Note that the two restrictions above apply only to problems 1-3; they do not apply to problems 4 and 5 (`getnth.c` and `skyline.c`)

No restrictions on library routines

My solutions use only `strlen`, `strcpy`, `printf`, `putchar`, and `puts` but you're free to use any C library routines that you wish.

Problem 1. (9 points) `warmup.c`

As the name implies, the purpose of this problem is to get you warmed-up writing some pointer-based code by implementing several very simple functions.

No full examples of operation of the functions are included in this write-up, but you can examine

a8/warmup1.c, and its output, to clarify how things work. Use gcc warmup.c a8/warmup1.c for experimentation.

The grading set for this problem will be only a8/warmup1.c. It won't change after 3:00pm on Friday, October 23, so there's no need to worry about cases not exercised in a8/warmup1.c.

Below are the nine functions you are to implement. Prototypes for all are in a8/warmup.h.

`void from_to(int from, int to, int by, int *result)`
stores an arithmetic progression of integers into consecutive ints starting at result. The first integer stored is from, the second is from + by, the third is from + by * 2, etc., through but not exceeding to. Assume that from <= to and by > 0.

Examples:
from_to(10,12,1,avals) stores 10, 11, and 12 in avals.
from_to(1,12,3,bvals) stores 1, 4, 7, and 10 in bvals.

Remember that you can't declare any non-pointer variables, but it's ok to change from, to, and/or by.

`void print_ints(int *first, int n)`
outputs n consecutive ints starting at first. Values are separated by commas, and the entire sequence is wrapped in curly braces and followed by a newline.

`void reverse(int *first, int *last)`
reverses the sequence of integers from first through last, inclusive.

My implementation swaps int values but a typical code for a swap uses a temporary variable, like this:

```
int tmp = i; // Non-pointer variable--NOT ALLOWED
i = j;
j = tmp;
```

But, as the comment indicates, that declaration of tmp is not allowed—tmp is not a pointer variable!

Instead of using a temporary variable, use an "XOR swap"—Google for it! For those who haven't had 252, or haven't heard of an "exclusive OR", see a8/xorswap.c for a working example.

Incidentally, "swap two values without using a temporary variable" is a well-known interview question.

`void merge(int *a, int *b, int n, int *result)`
merges the int sequences starting at a and b, both assumed to have n values, into a single sequence in result, simply alternating between values in a and b. Assume n >= 0.

Example: If n is 4, a is {1,2,3,4} and b is {10,20,30,40}, result would be

`{1,10,2,20,3,30,4,40}`.

`split(int *vals, int n, int *a, int *b)`
is the opposite of `merge`, copying the `n` ints in `vals` into `a` and `b` alternately.

Example: If `n` is 6 and `vals` is `{1,2,3,4,5,6}`, then a would be `{1,3,5}` and `b` would be `{2,4,6}`. Assume `n >= 0` and is an even number.

`char *merge_s(char *s1, char *s2, char *result)`
is a string-based analog of `merge`, above. It copies characters into `result` alternately from `s1` and `s2`. If `s1` and `s2` are not the same length, remaining characters of the longer one are appended to the merged characters. `merge_s` returns its third argument.

Example: Merging `"abc"` with `"12345"` produces `"a1b2c345"`.

`void split_s(char *s, char *r1, char *r2)`
is a string-based analog of `split`, above. It copies characters from `s` alternately into `r1` and `r2`. If `s` contains an odd number of characters then `r1` will be one character longer than `r2`.

`void blend(int *vals, char *s)`
prints integers from `vals` with characters from `s` interleaved between them. For example, if `vals` is `{10,20,30,40,50}` and `s` is `"xyz"` then this output is produced:

```
{10x20y30z40}
```

The length of `s` determines how many integers from `vals` are output. If `s` is zero-length, only the first integer in `vals` is produced.

Like `print_ints` above, `blend` wraps its output with curly braces and follows it with a newline.

`void average(int *first, int *last, double *avg)`
computes the average of the `int`s between `first` and `last`, inclusive and stores it in the double pointed to by `avg`.

Your first thought might be to do this: `int n; double total;`, but those are not pointer variables, and are not allowed! There's a little bit of a puzzle here, but don't let it become frustrating or turn into a time sink.

I'm hereby declaring a Piazza BLACKOUT on `average`—no posts about `average`, please. Minimum penalty: one point. If you're stuck on `average`, mail to `352f15` for a hint.

Note that because there's only one test case, `warmup1.c`, grading for this problem, `warmup.c`, is all-or-nothing. If the Tester shows you passing `warmup1.c` and you don't violate either restriction, you're guaranteed all nine points; if not, it'll be a zero. Note that it's far better to violate a restriction on a function, and lose a point for it, rather than leaving a function unfinished, causing the `warmup1.c` test to fail, resulting in a zero on the whole nine-point problem.

Problem 2. (11 points) `cset.c`

Implement the following cset routines, as described in assignment 7:

```
void str_to_cset(char *str, 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);
```

Note that `cset_size` returns an `int`. The character-counting example in the restrictions, and my solution, in `a8/count-hint.c`. may be helpful when implementing `cset_size`.

The above prototypes can be found in `a8/cset.h`.

Problem 3. (11 points) `path.c`

Implement `void path_elem(char *path, char which, char *result)` using the specification of `path_elem` in assignment 7.

Problem 4. (12 points) `getnth.c`

One way to have a string contain other strings is to use a delimiter to separate the contained strings, like `"red,green,blue"`. However, an obvious shortfall of such a representation is that the contained strings can't contain the delimiter. Another approach is to precede each string with its length, like this: `"3.red5.green4.blue"`. Using a period to signal the end of the length allows it to be many digits long. The string `"4.4you3.2me"` contains two strings: `"4you"` and `"2me"`. Take a look at `a8/getnth3.c` for more examples.

In this problem you are to write a function that extracts a specified string from a string in the length-encoded form shown above. Here is the prototype:

```
int getnth(char *string, int N, char *result);
```

`getnth` finds the `N`th string (zero-based) and copies it to the address specified by `result`; it assumes the caller has provided sufficient space. If `N` is negative or too large, `getnth` returns `-1` and does not change the buffer specified by `result`. If `getnth` is successful, it returns the length of the string copied into `result`.

Below is a test program, `a8/getnth1.c`. Note that `s` is specified as two consecutive string literals—recall from our study of the `iprint(e)` macro that the compiler treats juxtaposed string literals as a single string literal.

```
int main()
{
```

```
    char *s = "3.red5.green4.blue4.4you3.2me0."
              "10.12.18.201515.now and then...7.the end";

    for (int i = 0;; i++) {
        char value[100];
        int len;
        if ((len = getnth(s, i, value)) >= 0)
            printf("%d: '%s' (%d chars)\n", i, value, len);
        else
            break;
        }
}
```

Execution:

```
% gcc getnth.c a8/getnth1.c && a.out
0: 'red' (3 chars)
1: 'green' (5 chars)
2: 'blue' (4 chars)
3: '4you' (4 chars)
4: '2me' (3 chars)
5: '' (0 chars)
6: '12.18.2015' (10 chars)
7: 'now and then...' (15 chars)
8: 'the end' (7 chars)
```

`getnth` assumes that the string is well-formed. For example, a string such as `"2xy"` is invalid because the length is not followed by a period. The string `"2.ab3.c"` is invalid because a length of three is specified for the second string but only one character follows the period. `getnth` assumes that the length of a contained string can be represented with an `int`.

There are no restrictions on your `getnth` implementation—use any elements of the language and the libraries that you desire.

STRONG Recommendation: This problem is simpler than problems 1-3 but do this problem AFTER you've done problems 1-3. If problems 1-3 have caused the expected alterations in your brain you should find yourself naturally thinking about a pointer-based solution for this problem, too.

Problem 5. (12 points) `skyline.c`

In this problem you are write a function that builds a string that represents a simple "skyline". For example, the string `"1231234"` represents this skyline:

```
      x
  x   xx
 xx xxx
xxxxxxx
```

Here is the prototype: `void skyline(char *s, char *result).` It is important to understand that `skyline` produces no output—it constructs a string with embedded newlines (`'\n'` values) in the buffer specified by `result`. Example: (`a8/skyline1.c`)

```
int main()
{
    char buf[100];
    skyline("32023", buf);
    printf("%s", buf);
}
```

Execution:

```
% gcc skyline.c a8/skyline1.c && a.out
x   x
xx xx
xx xx
%
```

`skyline` assumes that the result buffer is big enough and that the specification contains only digits. `skyline` does not modify the characters that `s` points to. (If it did, the above example would fault!)

There are no restrictions on `skyline`—use any elements of the language and the libraries that you desire.

STRONG Recommendation: This problem is simpler than problems 1-3 but do this problem AFTER you've done problems 1-3. If problems 1-3 have caused the expected alterations in your brain you should find yourself naturally thinking about a pointer-based solution for this problem, too.

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 `a8/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N`*`.`*`YYYYMMDD`*`.`*`HHMMSS`*`.tz` You can run `a8/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.

`a8/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 < a8/delivs)
  90  237 1557 warmup.c
  62  162 1108 cset.c
  56  141 1297 path.c
  30   78  549 getnth.c
  28   69  521 skyline.c
 266  687 5032 total

% for i in $(grep -v txt a8/delivs); do echo $i: $(tr -dc \; < $i
| wc -c); done
warmup.c: 35
cset.c: 28
path.c: 22
getnth.c: 12
skyline.c: 14
```

There are no comments in my code, and no incorrect comments, either.

Miscellaneous

This assignment is based on the material on C slides 1-317.

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 don't seem to 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.)