CSC 352, Fall 2015
Assignment 6
Due: Wednesday, October 7 at 23:59:59

Introduction

The `a2` write-up started with 3+ pages of various information about assignments. None of that is repeated here, but all of it applies to this assignment, too.

Be sure to make the appropriate a*N* symlink for this assignment.

This Assignment Has Restrictions

If you've ever done any strength training, you know that particular exercises like presses, curls, and squats develop particular muscle groups. If you've had a basketball coach, I bet he or she made you learn to dribble, pass and shoot with both hands. I imagine that any soccer coaches in your past made you learn to shoot with both feet. I use restrictions on assignments in hopes of producing a similar well-roundedness with various course-related skills.

Unless a problem-specific exemption is specified, THE FOLLOWING RESTRICTIONS APPLY TO ALL SOLUTIONS for this assignment:

- **Arrays may not be used.**
- **"Pointers" may not be used.**
- **The only library routines that may be used are `getchar`, `putchar`, and `printf`.**

You'll likely never see such restrictions in the workplace but my job is to teach you a set of skills that may help you get a great job. Restrictions are one of the tools I use to help focus you on learning certain skills.

A solution that doesn't honor all restrictions will earn a greatly reduced score, typically a zero. The TAs and I are always happy to take a look at a solution in progress to be sure it doesn't violate any restrictions, but watch out for a crunch as the deadline approaches.

It's unlikely you'd use pointers or arrays by accident in C but to ease any concerns let me say a couple of things about pointers and arrays. First of all, note that we've been using `main`s that look like this:

```
int main(int argc, char *args[]) { ...
```

We'll be learning that `char *args[]` declares `args` to be an array of pointers. It's fine to have that line present but don't make any use of `args`. (I mention that only because I fear a student who's desperately wanting to use an array might try to press `args` into service! That'd be pretty silly, but I've seen far sillier.)

In general, variables of pointer type are declared with *TYPE *VAR*, like `int *p`. Array declarations look similar to Java: `int a[...]`. One way to create a pointer is with `&`*VAR*. Stay away from those three constructs, minus an exception that's provided for `rectangle.c`, and you should be fine! If any doubts, just mail us, and be sure to send your code.

Use our `gcc` alias!

Use our `gcc` alias! Here's the latest version:

```
alias gcc="gcc -Wall -g -std=c1x -I/cs/www/classes/cs352/fall15/h"
```

You'll also want to enable core dumps with this `ulimit` command:

```
ulimit -c 1000000
```

If you followed the instructions on UNIX slides 147-160 for setting up your `~/.profile` and `~/.bashrc`, just put the `alias` and `ulimit` lines above in your `~/.bashrc` and then type `restart`. That should produce no error messages. When done, check them like this:

```
% alias gcc
alias gcc='gcc -Wall -g -std=c1x -I/cs/www/classes/cs352/fall15/h'
% ulimit -c
1000000
```

If dot is not in your path (`$PATH`) (and I recommend it not be—see UNIX slides 157-158), I suggest putting your `a6` directory in your path. Here's what you might add to your `.bashrc` to achieve that:

```
PATH=$PATH:~/352/a6
```

If any trouble, mail to `352f15`!

CMD* && *CMD

I've suggested that when working with Java you use bash's `&&` operator to combine compilation and conditional execution like this:

```
% javac x.java && java x
```

Here's the obvious analog in C:

```
% gcc -o x x.c && x
```

The Tester runs `gcc` first

The Tester always compiles your code before running the tests. There's no need to run `gcc` before you run the Tester.

Note that although I recommend `-Wall` in your `gcc` alias, the Tester uses `-Werror`, which turns all warnings into errors.

`a6/tester` is in place.

Problem 1. (3 points) `rectangle.c`

For this problem you are to write a C program that reads one or more pairs of `int`s from standard input and outputs a rectangle drawn with asterisks. Example:

```
% echo 15 5 | rectangle
***************
*             *
*             *
* 15x5        *
***************
```

The top and bottom lines above have 15 asterisks. You can see that *`WIDTH`*x*`HEIGHT`* is shown in the left-hand corner.

Assume that rectangles will be at least three lines high, and wide enough to show the width and height without a bulge.

`rectangle` will always be given at least one pair of `int`s but it can handle any number of pairs. Behavior is undefined if the input is anything but pairs of `int`s.

Implementation notes

For this problem, `rectangle.c`, you are permitted to use `scanf` and the `int` pointers that `scanf` requires. I recommend that your `main` look like this:

```
int main(int argc, char *args[])
{
    int width, height;
    while (scanf("%d %d", &width, &height) == 2)
    {
        ...
    }
}
```

As a learning step, print `width` and `height` in that loop.

Don't overlook this: `printf` returns the number of characters output.

Problem 2. (5 points) `matched.c`

Write a C program that reads lines on standard input and determines whether the parentheses on the line are matched. The program simply reads and writes lines, following each line with either "`: matched`" or "`: NOT matched`".

Examples:

```
% cat a6/matched.1
(a)
(((a)b)()c)()
)(
((( )) () (  ((()))((())) ))
((()))((((()))((()))))
% matched < a6/matched.1
(a): matched
(((a)b)()c)(): matched
)(: NOT matched
((( )) () (  ((()))((())) )): matched
((()))((((()))((()))): NOT matched
% echo | matched
: matched
% (cat /bin/date; echo) | matched | grep -a ": NOT" | wc
      55     679   38742
%
```

Assume the last character of standard input is a newline.

If there is no input, `matched` produces no output.

The last example above emphasizes that `matched` should be able to handle any input stream, as long as it ends with a newline. (The construction "**`(cat /bin/date; echo) | matched`**" uses a *subshell*—a shell command line in parentheses—to group the output of `cat` and `echo`, which produces the required final newline, into an input stream for `matched`.)

Problem 3. (5 points) `sumnums.c`

Write a C program named `sumnums` that reads lines on standard input, considers every contiguous sequence of digits to be a whole number, and writes on standard output the sum of those numbers.

```
% echo x1x2x3-4-5 | sumnums
15
% cal -h 10 2015 | sumnums
2511
% cal -h 11 2015 | head -4 | tail -2 | sumnums
105
% echo Tue Sep 29 02:56:58 MST 2015 | sumnums
2160
% echo hello | sumnums
0
%
```

Assume that individual numbers, as well as the sum, can be represented with an `int`. Assume the last character of standard input is a newline.

If there is no input, or no digits in the input, `sumnums` outputs 0.

Implementation note: A simple way to convert an ASCII digit to the numeric quantity it represents is by subtracting `'0'` (a character constant) from it. For example `'7' - '0'` is `7`.

Here's a C program that reads digits and prints the numeric value that corresponds to each digit:

```
% cat a6/digits.c
#include <stdio.h>

int main(int argc, char *args[])
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if (c == '\n')
            continue;
        printf("%c -> %d\n", c, c - '0');  // Equiv: c - 48
    }
}
```

Interaction:

```
% echo 734 | digits
7 -> 7
3 -> 3
4 -> 4
```

This technique, subtracting `'0'`, works because the ASCII digits are contiguous: they're in the range 48 through 57. The above code is in `a6/digits.c`. Try some non-digits as input and study the results.

Problem 4. (7 points) `vis.c`

Write a C program named `vis` that prints on standard output a more visual representation of characters read on standard input. In terms of function, `vis` falls between `cat -A` and `od`: it doesn't have the possible ambiguities of `cat` but is more readable in some cases than `od`.

Here is a simple example:

```
% cat a6/text
one
    2  3 4
three
% vis < a6/text
one<NL><SPC><SPC><SPC><SPC>2<SPC><SPC>3<SPC>4<NL>three<NL>%
%
```

We can see that a newline follows "`one`" and that the second line starts with a tab character. Note that the output of `vis` does not have a newline at the end. The last character in `text` is a newline, and the `%` following the final `<NL>` is the `bash` prompt. (The second `%` following `<NL>` was produced by hitting ENTER.)

For each input character `vis` outputs a representation that is one or more characters in length. Here are the rules:

(1) Some characters are special-cased:

Input	Output
`NUL` (0)	`<#0>`
tab	`<TAB>`
newline	`<NL>`
carriage return	`<RET>`
ESC (27)	`<ESC>`
space	`<SPC>`
<	`<LT>`
`DEL` (127)	`<DEL>`

Each of the above characters causes the output of the four or five characters shown above. For example, if a zero-valued character is read, four characters are output: <, #, 0, >.

(2) If a character is not one of the special cases above and has a value less than 27 it is shown as `<C-?>`. For example, control-A, which has an ASCII code of 1, is printed as `<C-a>`, a five-character sequence. ASCII character 26 is printed as `<C-z>`.

(3) Characters in the range 28 through 31 are shown as follows:

Input	Output
28	`<C-\>`
29	`<C-]>`
30	`<C-^>`
31	`<C-_>`

Observe that each of the above characters produces a five-character sequence.

(4) Except for the less-than sign, characters in the range 33 through 126 inclusive are simply printed. For example, if an `a` is read, an `a` is printed.

(5) Characters in the range 128-255 are shown as `<#`*N*`>` , where `N` is the decimal representation of the value. For example, 200 is shown as `<#200>`.

`vis` is obviously not portable to a non-ASCII machine—this specification is packed with

specific behaviors based on ASCII character codes.

The file `a6/256.chars` contains each 8-bit character value from 0 to 255. It provides a simple full test for `vis`. In the following example the output of `vis` is piped through `fold` to facilitate display in this document.

```
% vis < a6/256.chars | fold -60
<#0><C-a><C-b><C-c><C-d><C-e><C-f><C-g><C-h><TAB><NL><C-k><C
-l><RET><C-n><C-o><C-p><C-q><C-r><C-s><C-t><C-u><C-v><C-w><C
-x><C-y><C-z><ESC><C-\><C-]><C-^><C-_><SPC>!"#$%&'()*+,-./01
23456789:;<LT>=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
klmnopqrstuvwxyz{|}~<DEL><#128><#129><#130><#131><#132><#133
><#134><#135><#136><#137><#138><#139><#140><#141><#142><#143
><#144><#145><#146><#147><#148><#149><#150><#151><#152><#153
><#154><#155><#156><#157><#158><#159><#160><#161><#162><#163
><#164><#165><#166><#167><#168><#169><#170><#171><#172><#173
><#174><#175><#176><#177><#178><#179><#180><#181><#182><#183
><#184><#185><#186><#187><#188><#189><#190><#191><#192><#193
><#194><#195><#196><#197><#198><#199><#200><#201><#202><#203
><#204><#205><#206><#207><#208><#209><#210><#211><#212><#213
><#214><#215><#216><#217><#218><#219><#220><#221><#222><#223
><#224><#225><#226><#227><#228><#229><#230><#231><#232><#233
><#234><#235><#236><#237><#238><#239><#240><#241><#242><#243
><#244><#245><#246><#247><#248><#249><#250><#251><#252><#253
><#254><#255>%
```

Another example:

```
% vis < a6/256.chars | wc
       0       1    1033
```

`wc` reports a line count of zero because `vis` never outputs a newline character.

Problem 5. (11 points) `siv.c`

Write a C program named `siv` that reverses the work of `vis`. It reads on standard input a stream of data produced by `vis` and outputs the data originally read by `vis`.

In the following example, `a6/256.chars` is processed by `vis`. The resulting file is then processed by `siv`, which in turn produces a file named `new` that `cmp` shows to be identical to `256.chars`.

```
% vis < a6/256.chars > out
% siv < out > new
% cmp a6/256.chars new
%
```

Any number of passes through `vis` can be undone by an equal number of passes through `siv`. Example:

```
% vis < a6/512.chars | vis | vis | siv | siv | siv > x
% cmp a6/512.chars x
%
```

`a6/vis` is my implementation of `vis`. You can use it to test your `siv`:

```
% echo $(seq 5) | a6/writebytes | a6/vis | siv | a6/vis
<C-a><C-b><C-c><C-d><C-e>%
```

IMPORTANT: Assume that the input of `siv` was produced by `vis`. For example, you don't need to be sure that a `"<"` is followed by one of the expected characters, or that digits follow `"<#"`, or that `"<SPC>"` has the closing bracket. The behavior of `siv` is "undefined" if it is given input other than data produced by `vis`.

Implementation notes for `siv`:

(1) Consider a function `int getnum(void)` that is called after `"<#"` has been read. It then reads digits until `">"` is encountered and returns the `int` that corresponds to the value represented by the digits.

(2) Note that the character immediately following `"<"` uniquely determines what the next element is. For example, if you see `"<S"` you can be sure that `"PC>"` comes next. Consider a function `void skip(int N)` that simply calls `getchar() N` times and discards the results.

Problem 6. (2 points Extra Credit) `rev.c`

Write a C version of `rev.java` from `a2`. You may not assume any specific limit on the number of lines or the length of lines. You may assume the input ends with a newline.

Note: Assuming you keep the assignment-wide restrictions in mind, this is perhaps the hardest problem on this assignment. There is a one-word hint in `a6/rev-hint`.

Problem 7. 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 "`Hours:`" lines.

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 `a6/turnin` to submit your work. Each run creates a time-stamped "tar file" in your `a`*`N`* directory with a name like `a`*`N`*`.`*`YYYYMMDD`*`.`*`HHMMSS`*`.tz` You can run `a6/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.

`a6/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 < a6/delivs)
  30   81  646 rectangle.c
  30   67  529 matched.c
  25   68  476 sumnums.c
  32   96  914 vis.c
  63  149 1418 siv.c
  24   39  298 rev.c
 204  500 4281 total
% grep -c \; $(grep -v txt < a6/delivs)
rectangle.c:14
matched.c:10
sumnums.c:8
vis.c:13
siv.c:27
rev.c:7
```

There are no comments in my code.

Miscellaneous

This assignment is based on the material on C slides 1-138, plus maybe a little in the slides on functions that will follow 138 but have not yet been written.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond 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 reach the eight-hour mark, regardless of whether you have specific questions, it's probably time to touch base with us. 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.)