baAssignment 5: Banner Builder

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Updated: Tuesday, October 26th, 2021 @ 11:00AM

Due: Monday, November 1st, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. Start early. **You MUST run the assignment on the pi cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on EdStem.

Mid Quarter Survey (2pts)

Please fill out this survey: https://forms.gle/AM7PR81Rf4HNURHw9

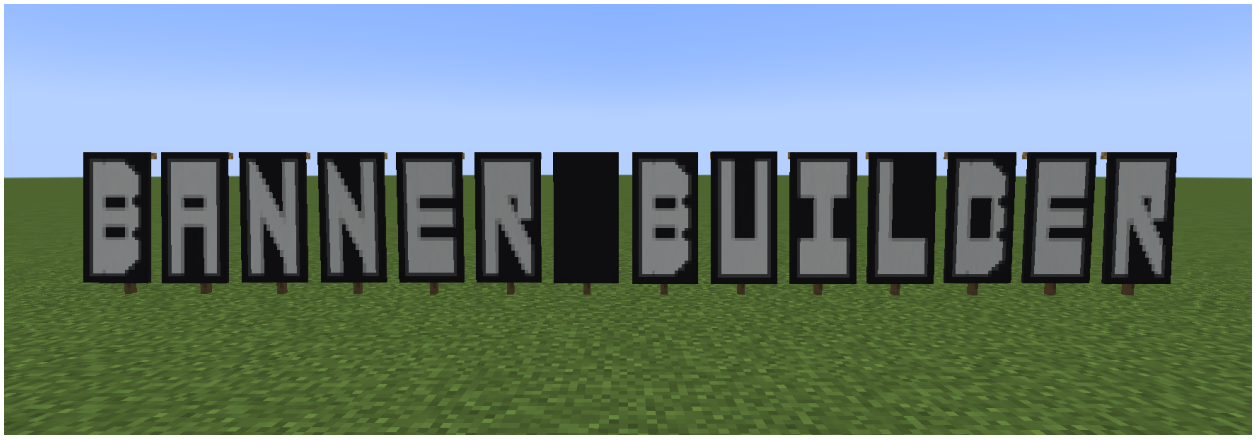
Table of Contents

Learning Goals

- Learn to implement a C program that takes arguments/flags
  - Learn to use File IO
  - Learn to manipulate pointer arrays
  - Learn to read and write 2D Arrays
  - Learn to interpret flags and optional arguments

Assignment Overview

For this assignment, we will be creating a banner builder program! What is a banner? A banner is a block in Minecraft that is often used to decorate things as it is able to have custom patterns painted on it. A very common pattern used by players are letter patterns, which are ones in which, as the name suggests, are patterns where there looks to be letters displayed on the banners.



In this assignment, we've provided many "banner patterns" in the starter code, and your job is to be able to translate inputted text into a "banner build". Your program will build the banners according to the inputted text using the provided patterns.

Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See Edstem FAQ or watch CSE30 Development Tools Tutorial
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following GitHub repository.

1. Clone/download the GitHub repository.[1]
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/bannerbuild-a5-ref`.
You can directly use `bannerbuild-a5-ref` as a command.

Makefile: The `Makefile` provided will create a `bannerbuild` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead. Run `make clean` to remove files generated by `make`.

[1] If you use your own Github repo, make sure it is private. Your code should never be pushed to a public Github repository. **A public Github repo is an AI violation.**

How the Program Works

Given a message string, the program prints large ASCII-art character banners for every character in the input.

Input :

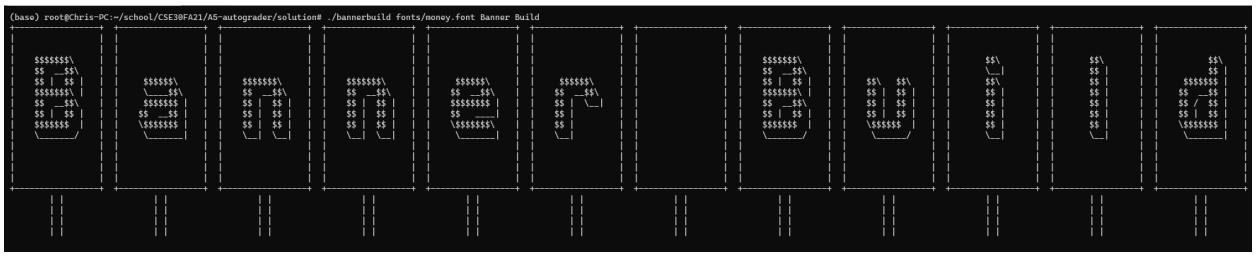
- A font file
- A message string

Output :

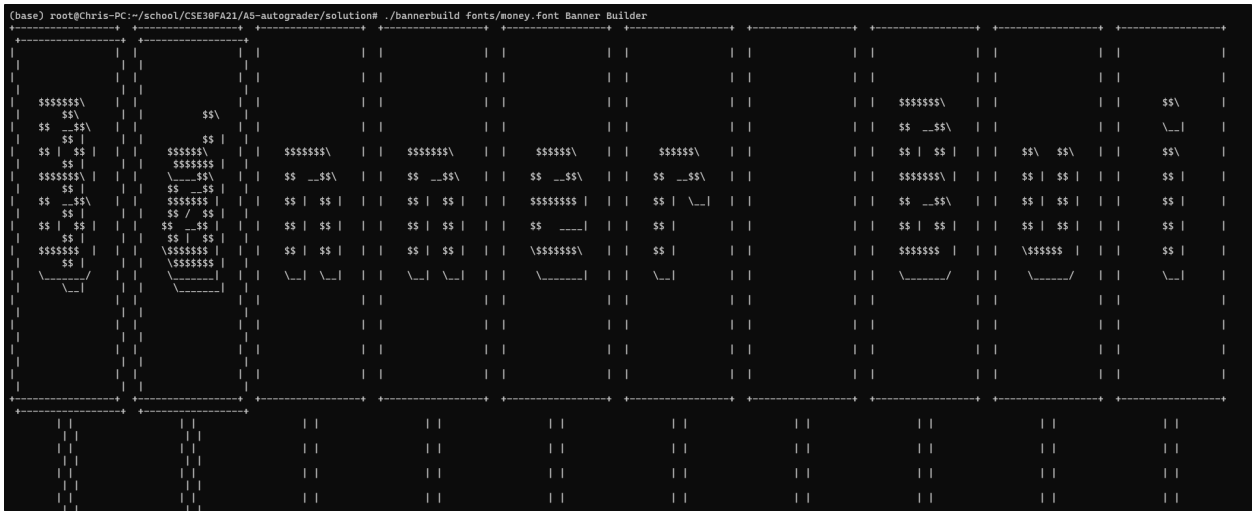
- A `MAXHEIGHT` × `MAXWIDTH` size symbol message printed using cBanners (ASCII-art)

Make sure your terminal window is large enough to hold the entire buffer or it may wrap around and mess up the display (256x24 chars by default; defined in `globals.h`). Typically your terminal will allow zooming out/in with `'CTRL'+'-'` and `'CTRL'+'+'` or on Mac, `'⌘'+'-'` and `'⌘'+'+'`. You cannot specifically zoom out/in on the terminal in VSCode, so I would recommend using a separate terminal.

Ex: `./bannerbuild fonts/money.font Banner Build`



WRONG DISPLAY (WHEN TERMINAL IS TOO SMALL):



Banners Arguments

Format for calling this executable with arguments (the flags cannot be clubbed with each other):
`./bannerbuild [-u] [-h <height>] [-w <width>] [-f <fillSymbol>] <font file> <words ...>`

Argument(s)	Description
`-u`	Prints a usage message and exits the program.
`-h height` (optional argument)	`height` is an integer specifying the height of the banner build in characters. If `-h` is not specified, the default is 24. If the argument to `-h` is larger than 24, it is ignored.
`-w width` (optional argument)	`width` is an integer specifying the width of the banner build in characters. If `-w` is not specified, the default is 256. If the argument to `-w` is larger than 256, it is ignored.
`-f fillSymbol` (optional argument)	Specifies a `char` (`fillSymbol`) to use as the background of the build. The default `fillSymbol` is a single space character (`' '` or ASCII 32). To debug we recommend using the `'-'` as `fillSymbol`.
`<font file>`	The path of a font file (more on this later).
`<words ...>`	The words you want to appear in the output. If the words are not surrounded by single quotes (i.e. they are separate arguments), then each word is printed with one "space" between them. If the words are surrounded by single quotes (i.e. they are a single argument), then each word is printed with the same amount of whitespace as in the original. This word should only contain characters between the ASCII values of 32 (space `' '`), and 126 (tilde `'~'`). You can assume that the word will not contain any characters outside the mentioned ASCII range. No other whitespace characters but space should appear in the input text. **If no words are provided, the program prints nothing.**

You can assume that arguments will be in this exact order. The first optional argument is `-u`, then optionally `-h`, then optionally `-w`, then optionally `-f`, then a required font file, and then any input text. Note that if no input text is provided (whether or not `-h` and/or `-w` were used), your program should print absolutely nothing. You can assume that a valid font file will be included in any test command. See the final page of this write-up for example code for parsing this input.

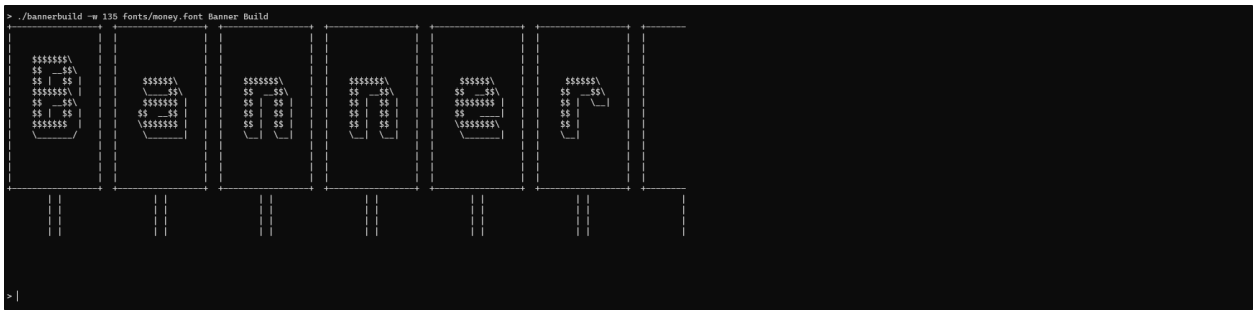
Examples:

$ ./bannerbuild -u

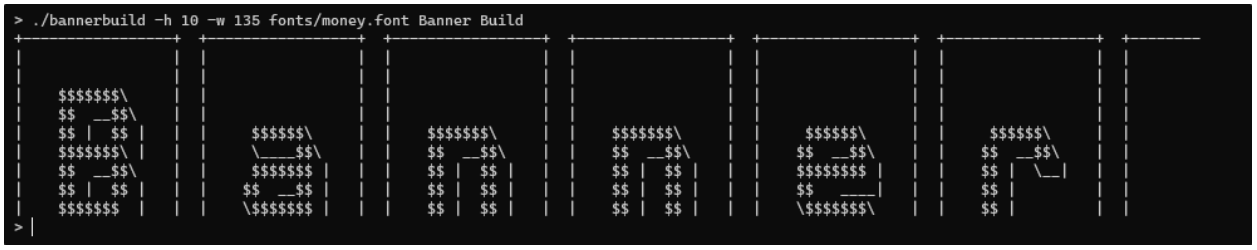
```
> ./bannerbuild -u
Usage: ./bannerbuild [-u] [-h <height>] [-w <width>] [-f <fillSymbol>] <font file> <input text>
  [-u]: Print the usage
  [-h <height>]: Specify the height of the banner build
  [-w <width>]: Specify the width of the banner build
  [-f <fillSymbol>]: Specify the fillSymbol of the banner build
  <font file>: The path of a font file
  <input text>: The words you want to appear in the output
```

$ ./bannerbuild -h 10 fonts/money.font Banner Build

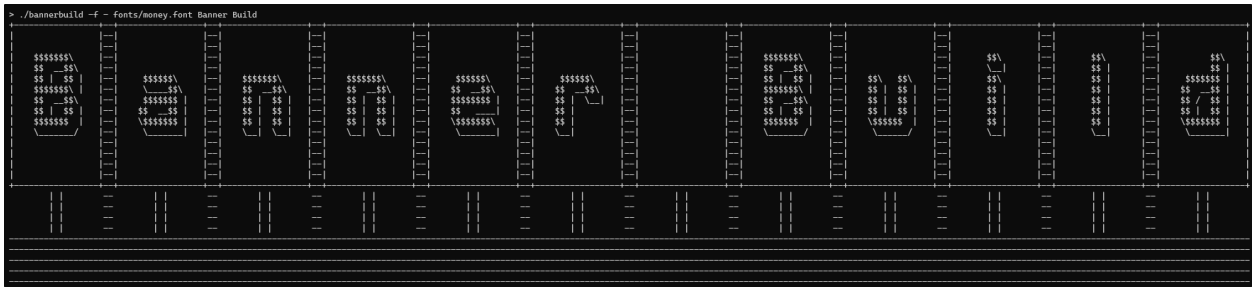
$ ./bannerbuild -w 135 fonts/money.font Banner Build



$ ./bannerbuild -h 10 -w 135 fonts/money.font Banner Build



$ ./bannerbuild -f - fonts/money.font Banner Build



Required Files

The following files are required to compile and run. `globals.h` **SHOULD NOT** be edited, as your submission **WILL NOT** use your file. For `fontBuf.c` and `displayBuf.c`, the function signatures are already written in their respective files and `globals.h`. There will be unit tests that will be looking for these specific signatures.

Filename	Description
`globals.h` (**DO NOT EDIT**)	Defines global constants, and externs variable and function declarations to interlink all of the other `.c` files (so that they can use each other's variables). Defined global constants: ● `MAXFONTSIZE` - Max size of fontBuffer ● `MAXFONTLINESIZE` - Max size of one line in font file ● `MAXNUMCBANNERS` - Number of distinct cBanners ● `MAXWIDTH` - Max width of the 2D `displayBuffer` ● `MAXHEIGHT` - Max height of the 2D `displayBuffer` ● `FONTDELIM` - `char` used to indicate end of cBanner ● `SPACING` - # of cols between cBanners in `displayBuffer` ● `SMILEY` - Represents emoji '`:)`' as ASCII value 127 ● `FIRSTCHAR` - `char` of first cBanner in font file ● `USAGE` - Usage string to be used for `printUsage()` Externed global variables from other files (functions found below): `fontBuf.c`: ● `char fontBuffer[MAXFONTSIZE]` ○ Holds all of the cBanners in flattened 1D form ● `char *cBannerLookup[MAXNUMCBANNERS]` ○ Maps a `char` to its cBanner's location in fontBuffer ● `int cBannerWidth` ○ Holds the width of the cBanners in the font file `displayBuf.c`: ● `char displayBuffer[MAXHEIGHT][MAXWIDTH]` ○ A 2D array for displaying cBanners ● `int displayHeight` ○ Holds the height of the final display (obtained from `-h`) ● `int displayHeight` ○ Holds the width of the final display (obtained from `-w`)

`fontBuf.c`	• `int readFontBuffer(const char *fontFile)`
	  ○ Reads font file
	  ○ Updates `cBannerLookup`
	• `void printCBanner(const char c)`
	  ○ Looks up and prints out appropriate cBanner

`displayBuf.c`	• `int copyCBanner(const char c, int xPos)`
	  ○ Starting at col `xPos`, looks up and copies the input `char c`'s corresponding cBanner char by char into `displayBuffer` (not exceeding `MAXWIDTH`)
	  ○ Returns where the next cBanner should start (i.e. the next `xPos`)
	• `void fillDisplayBuffer(const char c)`
	  ○ Fills `displayBuffer` with the given `char c`
	• `void printDisplayBuffer()`
	  ○ Prints the entire display buffer to `stdout`

`main.c`	• `void printUsage()`
	  ○ Prints the usage message
	  ○ Called when the `-u` flag is provided
	• `int main(int argc, char** argv)`
	  ○ Processes options
	  ○ Reads in the font fill
	  ○ Reads in the input string
	  ○ Prints out the banners

`README.md`	• Include name and hours spent in the fields provided

ASCII cBanners

We have implemented font files in the directory `fonts/`. These files contain ASCII-art of all the printable ASCII characters, displayed on an ASCII-art Minecraft banner. We will call them **cBanners**. In all the font files, each cBanner is separated by a single line containing **ONLY** the `'#'` delimiter character followed by a `'\n'` newline character. (Note: There is no delimiter character at the start of the file). This delimiter line is **NOT considered part of the cBanner**. Additionally, in between each cBanner and `'#'` is a line only containing the `'\n'` character. This is **NOT considered part of each cBanner**.

Below is a subset of one of the font files (`fonts/bannerbox.font`) with two cBanners shown, where `'↲'` represents the `'\n'` character. This font file uses `'#'` characters in the banners, so **you will need to handle** knowing whether or not encountering a `'#'` is the start of a cBanner.

`fonts/bannerbox.font`:

```
+-----------+↲
|           |↲
|    ###    |↲
|   #   #   |↲
|  #     #  |↲
|  #     #  |↲
|  #     #  |↲
|   #   #   |↲
|    ###    |↲
|           |↲
+-----------+↲
      | |     ↲
      | |     ↲
      | |     ↲
↲
#↲
+-----------+↲
|           |↲
|     #     |↲
|    ##     |↲
|   # #     |↲
|     #     |↲
|     #     |↲
|     #     |↲
|   #####   |↲
|           |↲
+-----------+↲
      | |     ↲
      | |     ↲
      | |     ↲
↲
#↲
```

Program Implementation

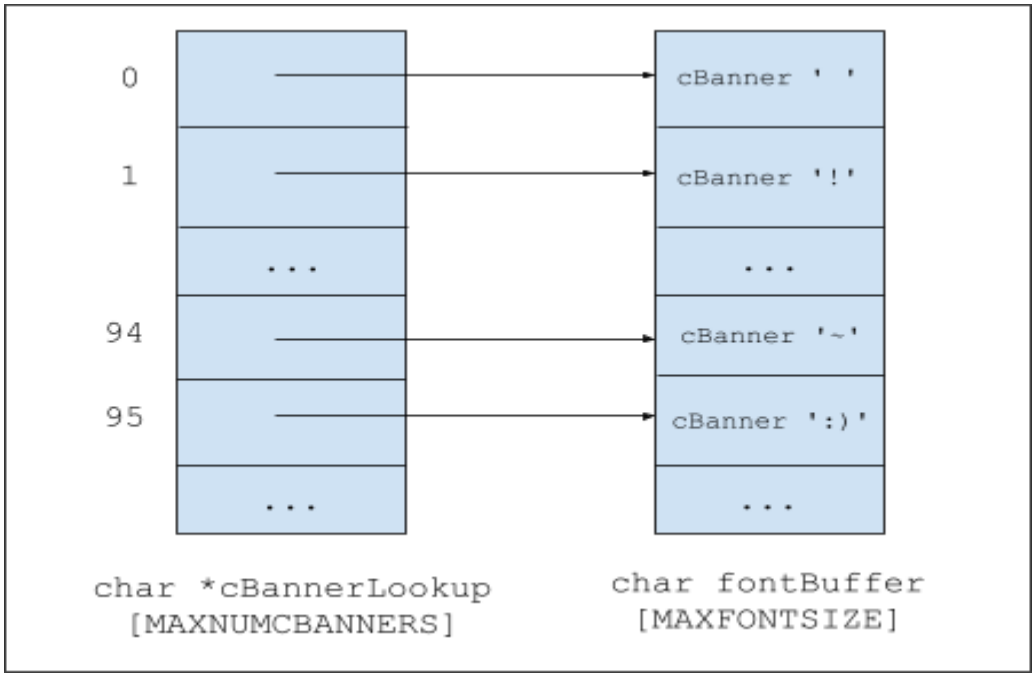
`fontBuf.c` Global Buffers (Arrays) and Variables

`char fontBuffer[]`: A big `char` array defined in `fontBuf.c` that is populated with the constituent symbols of all cBanners by the `readFontBuffer()` function, given some font file. The 2D cBanners are **flattened** in the `fontBuffer`, stored as sequences of concatenated rows.

- Note: cBanners are the same size and occupy the same amounts of storage in `fontBuffer`. However, the cBanner sizes differ BETWEEN font files, and your program must be able to handle all font files.

`char *cBannerLookup[]`: An array of `char` pointers defined in `fontBuf.c`. Each pointer points to the start of a cBanner in `fontBuffer`. This is effectively a char-to-cBanner map.

- Indexing: Our font file contains ASCII values 32 - 126 (`' '` - `'~'`), so the location of the cBanner for `' '` is stored in `cBannerLookup[0]`.
  We thus define `FIRSTCHAR` to be `' '`, and subtract its ASCII value (32) when indexing into `cBannerLookup`. For some `char c`, its cBanner data will be at the address given by `cBannerLookup[c - FIRSTCHAR]`.



`int cBannerWidth`: An integer defined in `fontBuf.c` that contains the width of the cBanners, given some font file. You will need to figure out the appropriate value for cBannerWidth in your code.

- This is used in `displayBuf.c` in order to determine what `xPos` to return from `copyCBanner()`.
- Be sure to account for `fgets()` returning the line with the `'\n'` newline character.

`displayBuf.c` Global Buffers (Arrays) and Variables

`char displayBuffer[][]`: A 2D `char` array defined in `displayBuf.c` used as a canvas to store the individual symbols that make up a series of cBanners.

- Dimensions: This array MUST be defined using the dimension order with height first and width second, as `displayBuffer[MAXHEIGHT][MAXWIDTH]` is how it is defined in `globals.h`. Thus, the 1st index pertains to rows, and the 2nd index pertains to cols.
- Filling: This array should be filled in only after the font buffers are completely finished. You will be filling this buffer in one char at a time, starting from the first character being printed from the string. When a character is being filled, you should only access the columns between [(`cBannerWidth + SPACING) * charNum`, ((`cBannerWidth + SPACING) * (charNum + 1) - SPACING`).
  For example, for the first character (numbered 0), if `cBannerWidth` is 19, the only columns that should be accessed are columns 0 through 18. However, the second character accesses columns 21 through 39. This is because columns 19-20 are skipped due to spacing, and should be filled by `fillDisplayBuffer()`. The correct column (`xPos`) should be returned from `copyCBanner()`.

`int displayHeight`: An integer defined in `displayBuf.c` that contains the height of the final printed display.

- This is obtained solely from `main.c` from the `getopt()` loop, specifically the `-h` flag.
- By default, it should be equal to `MAXHEIGHT`.

`int displayWidth`: An integer defined in `displayBuf.c` that contains the width of the final printed display.

- This is obtained solely from `main.c` from the `getopt()` loop, specifically the `-w` flag.
- By default, it should be equal to `MAXWIDTH`.

Functions and Behavior to Implement

You need to implement **all of the functions provided in all of the files** in the starter code **without changing their signatures**. Your functions must work with our provided `globals.h`. We propose the following workflow and guidelines.

1) `int readFontBuffer()` [10 points]

This function is partially completed in the starter code. The font file is read into the `FILE* fontFilePtr` pointer. Line 32 assigns the beginning of the 0th cBanner (the character `' '`) to be at index 0 of the `fontBuffer` array for convenience; feel free to change this if you wish.

The while loop starting on line 34 is the bulk of this code. The loop iterates over every line in the font file, copying the line ***into*** the `fontBuffer` array ***at the location of*** the `fontBufferPtr` pointer, and additionally updating the `fontLine` pointer to point to the same thing. For the moment, `fontBufferPtr` and `fontLine` are pointing to *exactly the same memory*, but they serve logically different purposes so we keep both: think of `fontBufferPtr` as the *location* in `fontBuffer` that should be updated, and think of `fontLine` as the *string* that just got copied.

Additionally, make sure to save the correct value to `cBannerWidth` as described above at any point during or after the loop.
Process the `fontLine` to correctly update the `cBannerLookup` array when necessary.
(How do you know when you've reached the end of a cBanner?)

After the `readFontBuffer()` function returns:

- The cBanner for `char c` must begin at the address given by `cBannerLookup[c - FIRSTCHAR]`, stored as a concatenated sequence of rows from top to bottom (including `'\n'` characters) and ending on the `FONTDELIM` char, which is just `'#'` defined in `globals.h`.
- The width of the cBanners for the font file must be given by `cBannerWidth`.
- The actual return value of this function is handled in the starter code.

2) `void printCBanner()` [2 points]

Given a char `c`, this function should print its cBanner to stdout. You can then use it to check that your `readFontBuffer()` is working correctly, for instance with a simple loop like below:

```
for(char c = ' '; c < '~'; c++) {
    printf("\n\'%c\' :\n", c);
    printCBanner(c);
}
```

```
$ ./bannerbuild money.font > foobar
$ less foobar
```

Once you correctly parse the font file, you are ready to move onto displaying a given message!

3) `int copyCBanner()` [10 points]

This function will copy the cBanner representing `char c` into the `outputBuffer` starting at row 0 and column `xPos`. It returns an `int` representing what the `xPos` of the next cBanner should be, which should account for the width of the cBanners and the additional inter-cBanner `SPACING`.

4) `void fillDisplayBuffer()` [2 points]

This function must fill the entire 2D `displayBuffer` array with the specified `char`. This `char` will serve as a background symbol for the cBanners. Hint: use a non-`' '` character (i.e. a dash `'-'`) to verify that your `copyCBanner()` function is positioning the cBanners correctly!

5) `void printDisplayBuffer()` [4 points]

Prints the `displayBuffer` to stdout. Since the terminal prints along a row, you should print one row of the `displayBuffer` at a time followed by a `'\n'` newline to return the terminal's cursor to the leftmost position on the next line. This is done for `DISPLAYHEIGHT` rows.

When printing the `displayBuffer`, you may reach the `DISPLAYWIDTH` of the buffer before printing out all of the symbols of the cBanners in a line. Nothing should be printed beyond these dimensions; any remaining cBanner symbols should just be simply cut off!

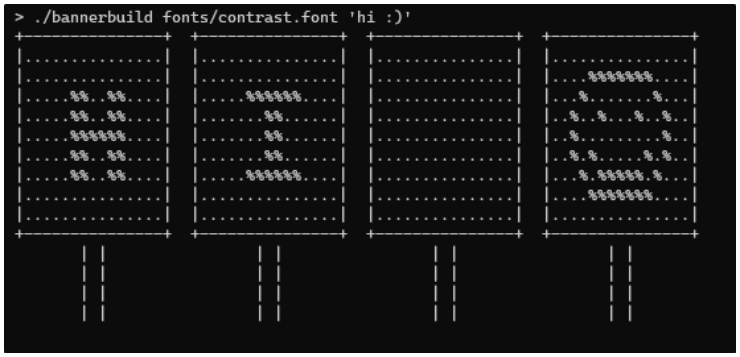
6) `int main()` [18 points]

You're ready to put it all together now! `main()` should process the options (see the [getopt example] on the next page), fill in the `displayBuffer` with the appropriate symbol (`' '` by default if not specified), read the font file into the `fontBuffer`, process the input string copying its cBanners into the `displayBuffer`, and print the `displayBuffer` out to stdout.

7) Support for a multi-char custom cBanner [2 points]

At the end of font files, we have defined a custom SMILEY cBanner. If you see the string `':)'` in the input (must be enclosed in single quotes), instead of printing the two cBanners for `':'` and `')'`, print the cBanner pointed to by `cBannerLookup[SMILEY]`.

Ex: `$ ./bannerbuild fonts/contrast.font 'hi :)'`



`getopt()` Example

You will be parsing flags (options) to your program. Some options are just a letter (e.g. `-u`) while others flags will have a value associated with them (e.g. `-h 10`). . Below is an example using the library function getopt(), which illustrates how you can parse two flags `-a` and `-c`, where `-a` takes an int argument, `-c` takes a char argument, a font filename, and then input text.

`getopt()` has an external variable called `optind` that stores the index of the argument to an option that it uses to parse `argv`, so you can access the argument with `argv[optind]` as you are currently processing an option flag. The rest of the input arguments will be in order after index `optind`. So, for example, if the command was:

```
$ ./myProgram -x -a 5 -c c money.font Hello there
```

first processes the -x flag; then as you process the `-a` flag, `argv[optind]` will be `5`; then as you process the `-c` flag, `argv[optind]` will be `'c'`, and after `getopt` finishes parsing the flags, `optind` will point to the next value of `"money.font"`, meaning that `argv[optind+1]` is `"Hello"` and `argv[optind+2]` is `"there"`.

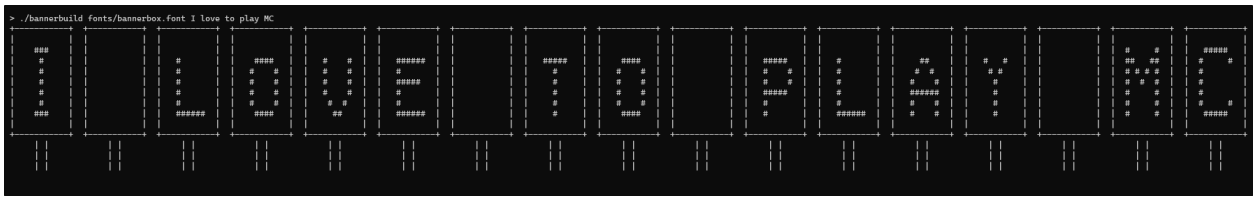
`optarg` will store the integer values passed in for `-a` and `-c`.
`strtol` converts a string to a `long`.
The string `"a:c:x"` in the while loop means that there are `-x, -a`, and `-c` optional flags of which `-a` and `-c` have an accompanying argument passed afterwards (an integer or a character in our example).

```
// parse optional commands
int opt;
 while((opt = getopt(argc, argv, "a:c:x")) != -1){
   switch (opt){
     case ('x'):
       printf("Got the option -x\n");
       break;
     case ('a'):
       printf("Value of arg a is: %d", (int) strtol(optarg, NULL, 0));
       // do something for case a with optarg
       break;
     case ('c'):
       printf("Value of arg c is: %c", optarg[0]);
       // do something for case a with optarg
       break;
     default:
       break;
   }
 }
// access font file name
char* fontFile = argv[optind];
```

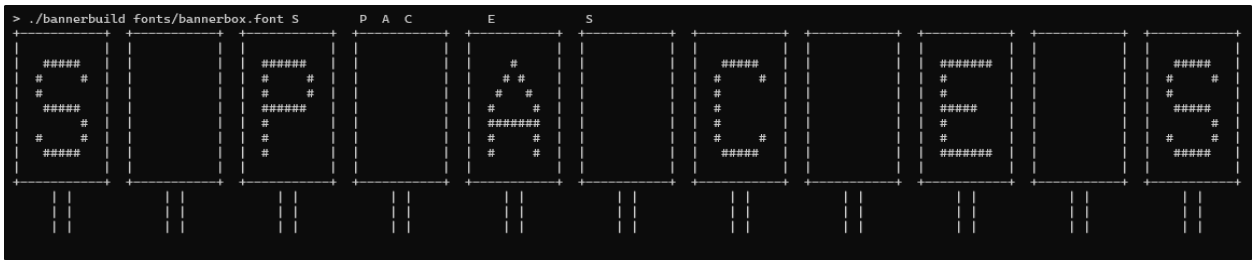
Extra Spacing Notes

Spaces Between Words

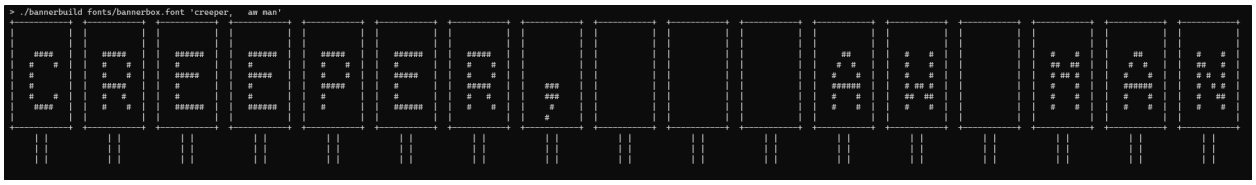
Separate words are printed with **exactly one cBanner of the space char `' '` between them**:



Note that this happens regardless of the number of spaces in your input, as the shell (bash / terminal) removes all whitespace between strings before passing them into your program's argv:



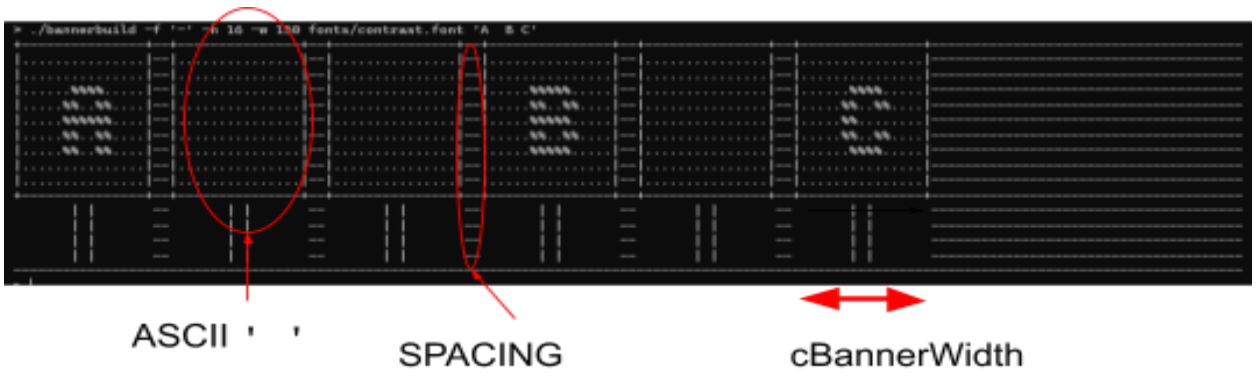
If you do want to print *multiple spaces* however, then surround the input by single quotes to force the shell to treat your entire quoted string *literally*. In the example below, the shell will remove the single quotes from `'creeper,  aw man'` and pass the *exact* string `creeper,  aw man` as one argument (one single argv entry) to your program:



Spaces Between Consecutive cBanners

Between any two consecutive cBanners (including the cBanner for the space character `' '` itself), there should be `SPACING` number of columns of the unaltered background character in the `displayBuffer`. To observe this, examine the following example:

Ex. `./bannerbuild -f '-' fonts/contrast.font 'A  B C'`



Summary of Specifications For Banner Builder

- Do not add other `.h` or `.c` files; complete all work in the provided starter code files.
- Do not edit `globals.h`, function signatures, or code that is outside of TODO sections.
- Ensure that your code compiles and runs on the pi-cluster with the commands used in the provided `Makefile` with no warnings.
- If no input text is given, your program should not print anything.
- You can assume a valid font file will be provided as an argument to your program.
- `-u, -h`, `-w`, and `-f` are optional, and if included, will be in the order specified on page 5.
- The size of the buffer printed should be the dimensions set by the default height and width (`MAXHEIGHT` and `MAXWIDTH`) or the height and width set by `-h` and `-w`, if they are provided.
- The background of the buffer printed should be the character set by the default fillSymbol (`space character ' '`) or the fillSymbol set by `-f`, if it is provided.
- Each cBanner in your input text should be printed with a two-character/column separation (as defined by the global `SPACING` variable).
- If your input text is enclosed in single quotes, your program should treat any extra spaces literally (see example on page 15).
- Any time the substring `':)'` appears in your input text, it should render as the final cBanner in the font file (in the font files, this is a smiley emoji); any input text with `':)'` needs to be enclosed in single quotes for it to work in the shell.
- We will be comparing your standard output (what is printed by your program) to ours.

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled “A5: Banners”.
   You will submit the following files:

   ```
   main.c
   fontBuf.c
   displayBuf.c
   README.md
   ```

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
   a. Check that all required files were submitted.
   b. Check that `main.c`, `fontBuf.c`, and `displayBuf.c` compile without warnings.
   c. Runs some tests on the resulting `bannerbuild` executable.

Grading Breakdown [2 + 48 pts]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

The assignment will be graded out of 50 points and will be allocated as follows:

- **Mid-Quarter survey: 2 points**
- `readFontBuffer()`: 10 test cases, each worth 1 point. 3 public, 7 hidden.
- `printCBanner()`: 1 test case, worth 2 points, hidden
- `copyCBanner()`: 3 test cases, 1 point each, hidden and 2 test cases, 3.5 points each, hidden
- `fillDisplayBuffer()`: 1 test case, 2 points, public
- `printDisplayBuffer()`: 1 test case, 4 points, hidden
- `main()`/End to end tests: 6 test cases, 2 points each, hidden and 6 test cases, 1 point each, hidden
- Support for special character: 2 test cases, 1 point each, hidden

NOTE: The end to end tests expect an EXACT output match with the reference binary. There will be NO partial credit for any differences in the output.

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster without warnings. **Any assignment that does not compile will receive 0 credit.**

Checking for exact output match:
A common technique is to redirect the outputs to files and compare against the reference solution:

```
./your-program args > output; our-reference args > ref

diff output ref
```

If the second command outputs nothing, there is no difference. Otherwise, it would output lines that differ with a < or > in front to tell which file the line came from.

Writeup Version

1.0 Initial release.
1.0.1 Fix line number typo in `readFontBuffer()` and text explaining the format for running the executable on Page 5.