A3: GDB and Printify

Due: Wednesday, Jan 29 @ 11:59PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. **Your program must be able to run on ieng6**

Part 0: Degree Planning Assignment [1 pt]
Please complete the degree planning assignment found here:
https://docs.google.com/document/d/1vrYQav_7lBePZpHp9KkTE2kCha8p_Q8uyvfdypEF8jI/edit

Part 1: GDB Core Dump [4 pts]

- What is GDB?
- Getting Started
- How the program works
- Submission and Grading

Part 2: Printify [45 pts]

- Getting Started
- Description
- Arguments
- Source Files
- ASCII cGrams
- Global Buffers
- Functions and Behavior to Implement
- Submission and Grading
- Extra Notes on Printify

Part 1: GDB Core Dump

When dealing with tools such as GDB you are free to search online to find any information that may help you. GDB is an extremely important tool and will save you HOURS of time debugging.

Learning Goals

- Gain practice using the gdb interface and commands
- Understand how useful gdb is
- Use gdb in Part 2 of this assignment and all future work in C and Assembly!
- Reading and understanding code written in C

What is GDB?

GDB is the GNU debugger. It allows you to see what is going on inside another program while it executes or the moment the program crashed. (https://www.gnu.org/software/gdb/)

Why use GDB? Can't I just use print statements?

Well, you can and should use print statements in some cases BUT gdb is an extremely useful tool for debugging that can help you determine exactly where errors are occuring. It allows you to execute a program step by step, print intermediate values, and set breakpoints to pause code execution at a specific line.

Getting Started

For this assignment you will need to run a simple program and determine what is causing a segmentation fault when you run it.

Compile the program:

```
$ gcc -g -O0 -Wall -Wextra -o <exe file> <src files>
```

Argument	Description
`gcc`	Runs the GNU Compiler Collection (GCC)
`-g`	Flag to produce debugging information

`-O0`	Used to reduce compiler optimizations
`-Wall` (optional)	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid
`-Wextra` (optional)	This enables some extra warning flags on top of `-Wall`
`-o <exe file>` (optional)	Name of the output executable (will be `a.out` if unspecified)
`<src files>`	Name of the source code files (just `coredump.c` in our case)

How the program works

For this assignment, you will run the program located in `coredump.c`. This is a small program that, when run, will lead to a segmentation fault. It is your job to use gdb to debug this program and determine what is causing the program to segfault. There starter code can be found here: https://github.com/cse30-wi20/hw3gdb

When you compile and run this program, the output will be:

```
Segmentation fault (core dumped)
```

We have provided you with an example of the commands you should run in gdb to find the issue but expect you to spend time playing around with gdb as well.

GDB Commands You Will Use

https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gnat_ug_unx/Introduction-to-GDB-Commands.html

Argument	Description
`run`	executes the program from the beginning
`where`	shows the next statement that will be executed
`up`	used to examine the content of frames
`p (or print)`	print the value of a given expression
`q (or quit)`	used to quit out of gdb

What You Should Run

```
$ gdb a.out

(gdb) run

Starting program: /home/core/a.out

Program received signal SIGSEGV, Segmentation fault.
strlen () at ../sysdeps/x86_64/strlen.S:106
106   ../sysdeps/x86_64/strlen.S: No such file or directory.
(gdb)

(gdb) where

#0  strlen () at ../sysdeps/x86_64/strlen.S:106
#1  0x00007ffff7a7c69c in _IO_puts (str=0x0) at ioputs.c:35
#2  0x00005555555546ee in printListFwd (curPt=0x555555756010) at coredump.c:26
#3  0x0000555555554786 in main (argc=1, argv=0x7fffffffdc78) at coredump.c:41

(gdb) up

#1  0x00007ffff7a7c69c in _IO_puts (str=0x0) at ioputs.c:35
35    ioputs.c: No such file or directory.


(gdb) up

#2  0x00005555555546ee in printListFwd (curPt=0x555555756010) at coredump.c:26
26    //! This function returns a pointer to the new record which becomes

(gdb) print curPt->name

… what is printed here?
```

In the above example, the `where` command prints which functions have been called to get to the failure point. Typing `up` moves gdb to the function that called a function. The error occurs in an internal library's functions (`strlen()` and `_IO_puts`), which are called from our program. So we type `up` twice to get to our code.

Submission and Grading

Submitting

You will submit a PDF to the assignment the **"HW 3: GDB"** assignment on Gradescope. The PDF should contain:

1. Take a screenshot of the terminal where you ran gdb (it should look similar to what is shown above). It is okay if you need multiple screenshots to show everything you ran.

2. Provide a short explanation on what is causing program to have a segmentation fault. What does the final command you run (`print curPt->name`) print? What does it mean? What does it say about the issue with our program? How can the program be fixed?

Grading Breakdown

- [2 point] Screenshots of GDB output
- [2 point] Explanation of why the segmentation fault is occurring

Part 2: Printify

Learning Goals:

- File IO
- Pointer arrays
- 2D Arrays
- Flags with optional arguments
- Multiple .c files and .h files

Getting Started

The starter code is at https://github.com/cse30-wi20/hw3starter. You can either clone the repo or download the zip.

The Makefile provided will create a `printify` executable from provided source files. Compile it by typing `make` into the terminal. Warnings are on by default (run `make no_warnings` to compile without warnings). Run `make clean` to remove all files generated by `make`.

Description

Printify is a commonly used spell that generates fancy announcements at venues like Quidditch matches and tournaments. Given a message string, it conjures large ASCII-art characters for every character in the input. (Handy ASCII Table.) **Make sure your terminal window is large enough to hold the entire buffer or it may wrap around and mess up the display (150x16 chars by default; defined in `globals.h`).**

Input:

- A font file
- A message string*

Output:

- A `MAXWIDTH` x `MAXWIDTH` symbol message printed using cGrams (ASCII-art)

Ex. `./printify fonts/cse30.font Printify!`

```
../printify: ./printify fonts/cse30.font Printify!
ooooooooo.                 o8o                        .      o8o     .o88o.                  .o.
`888   `Y88.               `"'                      .o8      `"'     888 `"                  888
 888   .d88'   oooo d8b   oooo    ooo. .oo.     .o888oo   oooo    o888oo    oooo    ooo   888
 888ooo88P'    `888""8P   `888    `888P"Y88b      888     `888     888       `88.  .8'    Y8P
 888            888         888     888   888      888      888     888        `88..8'     `8'
 888            888         888     888   888      888 .    888     888         `888'      .o.
o888o          d888b       o888o   o888o o888o     "888"   o888o   o888o         .8'       Y8P
                                                                                .o..P'
                                                                                `Y8P'
```

Printify Arguments

Format for calling this executable with arguments (up to one flag at a time):
`./printify [-h] [-f fillSymbol] <font file> <text to use for printify>`

Argument	Description
`-h`	Prints a help message and exits the program.
`-f fillSymbol`	Specifies a char (fillSymbol) to use as background. The default fillSymbol is a single space character (ASCII 32). To debug we recommend using the `'-'` as fillSymbol
`<font file>`	The path to a font file.
`<text to use for printify>`	The words you want to appear in the output. **If no input text provided, the program prints nothing.**

Source Files

The following files are required to compile and run, but `globals.h` must not be edited.

Filename	Description
`globals.h` **do <u>NOT</u> 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). • `MAXFONTSIZE` - max size of `fontBuffer` • `MAXFONTLINESIZE` - max size of one line in font file • `MAXNUMCGRAMS` - number of distinct cGrams • `MAXWIDTH` - max width of the 2D `displayBuffer` • `MAXHEIGHT` - max height of the 2D `displayBuffer` • `FONTDELIM` - char used to indicate end of cGram • `SPACING` - # of cols between cGrams in `displayBuffer` • `SMILEY` - represents emoji :) as ascii value 127 • `FIRSTCHAR` - char of first cGram in font file *continued on the next page*

	Externed globals from other files: • `char fontBuffer[MAXFONTSIZE]` - holds all of the cGrams in flattened 1D form • `char *cGramLookup[MAXNUMCGRAMS]` - maps a char to its cGram's location in `fontBuffer` • `int cGramWidth[MAXNUMCGRAMS]` - maps a char to its cGram's width • `char displayBuffer[MAXHEIGHT][MAXWIDTH]` - a 2D array for displaying cGrams
`fontBuffer.c`	• `int readFontBuffer(const char *fontFileName)` - reads font file; updates `cGramLookup`, `cGramWidth` • `void printCGram(const char c)` - looks up and prints out appropriate cGram
`displayBuffer.c`	• `int copyCGram(const char c, int xPos)` - starting at column `xPos`, looks up and copies the appropriate cGram char by char into displayBuffer (not exceeding `MAXWIDTH`) - returns where *the next cGram* should start (i.e. the next `xPos`) • `void fillDisplayBuffer(const char c)` - fills the display buffer with the given symbol • `void printDisplayBuffer()` - prints the entire display buffer to stdout
`main.c`	• `void printHelp()` - prints the help message; called with the -h flag • `int main(int argc, char** argv)` - processes options, reads in the font file, reads in the input string, and prints out the message

ASCII cGrams

We have implemented a font file in `fonts/cse30.font`. This file contains ASCII art of all the printable ASCII characters, which we call **cGrams**. Our `cse30.font` cGrams are separated by a single line containing the `'#'` delimiter character (no delim at the start of the file), which is **not considered part of the cGram**. Here is a subset of the file with two cGrams shown, where ↲ represents the `'\n'` character:

```
  .oooo.↲
 d8P'`Y8b↲
888    888↲
888    888↲
888    888↲
`88b  d88'↲
 `Y8bd8P'↲
↲
↲
↲
#↲
  .o↲
o888↲
 888↲
 888↲
 888↲
 888↲
o888o↲
↲
↲
↲
#↲
```

Global Buffers (Arrays)

`fontBuffer`: A big char array defined in `fontBuffer.c` that is populated with the constituent symbols of all cGrams by the `readFontBuffer()` function, given some font file. The 2D cGrams are **flattened** in the `fontBuffer`, stored as sequences of concatenated rows.
Note: cGrams are *of varying sizes* and occupy different amounts of storage in `fontBuffer`.

`cGramLookup`: An array of char pointers defined in `fontBuffer.c`. Each pointer points to the start of a cGram in `fontBuffer`. This is effectively a char-to-cGram map.

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

`cGramWidth`: An array of ints defined in `fontBuffer.c` that specify the width of each cGram.

- Indexing: The indexing scheme here is identical to that of `cGramLookup`.

`displayBuffer`: A 2D char array defined in `displayBuffer.c` used as a canvas to store the individual symbols that make up a series of cGrams. To print a message “ab”, the symbols that make up the cGrams for ‘a’ and ‘b’ are copied into `displayBuffer`, with additional SPACING between them. Then `displayBuffer` is printed to stdout.

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 cGram (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. Process the `fontLine` to correctly update the `cGramLookup` and `cGramWidth` arrays when necessary. (How do you know when you've reached the end of a cGram?)

After this `readFontBuffer()` function returns:
- The cGram for char `c` must begin at address given by `cGramLookup[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 cGram for char `c` must be given by `cGramWidth[c - FIRSTCHAR]`

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

Given a char `c`, this function should print its cGram 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);
    printCGram(c);
}
```

```
$ ./printify cse30.font > foobar
$ less foobar
```

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

3) `void copyCGram()` [10 points]

This function will copy the cGram representing `char c` into the `outputBuffer` starting at row 0 and column `xPos`. It returns an int representing what the `xPos` of the next cGram should be, which should account for the current cGram's width and the additional inter-cGram `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 cGrams. Hint: use a non-`' '` character (like a dash `'-'` for example) to verify that your `copyCGram()` function is positioning the cGrams 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 `MAXHEIGHT` rows.

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

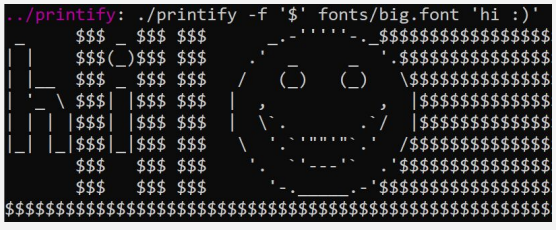
6) `int main()` [15 points]

You're ready to put it all together now! `main()` should process the options (see the getopt example at bottom of doc), 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 cGrams into the `displayBuffer`, and print the `displayBuffer` out to stdout.

7) Support for a custom cGram [2 points]

At the end of `cse30.font`, we have defined a custom SMILEY cGram. If you see the string `':)'` in the input (must be enclosed in single quotes), instead of printing the two cGrams for `':'` and `')'`, print the cGram pointed to by `cGramLookup[SMILEY]`.

Ex: `$ ./printify fonts/big.font 'hi :)'`



Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled HW3. You will submit the following files (you don't need to submit `globals.h`):

    - `main.c`
    - `fontBuffer.c`
    - `displayBuffer.c`

    To upload multiple files to gradescope, zip all of the files and upload the zip to the assignment. Ensure that the files you submit are not in a nested folder. Gradescope can extract from a zip, but not a folder.

2. After submitting, the autograder will run a few tests:
    a. Checks that all required files were submitted
    b. Checks that your source code compiles
    c. Runs functional tests on all of the functions you need to complete
    d. Compares output from your program to one from a solution implementation

Grading Breakdown

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade.

We will test your code both at a function wise level and at an input--output level. You will receive points for each test that you pass on Gradescope. Ensure your code compiles and runs on ieng6 with the provided `Makefile` and `globals.h`. **Any assignment that does not compile will receive 0 credit.**

Style Guidelines

No points are given for style, but teaching staff won't be able to provide assistance or regrades unless code is readable. Please take a look at the following Style Guidelines.

Extra Notes for Printify

Spaces between words

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

*note that for the long string above, we temporarily increased the `MAXWIDTH` value in `globals.h`.

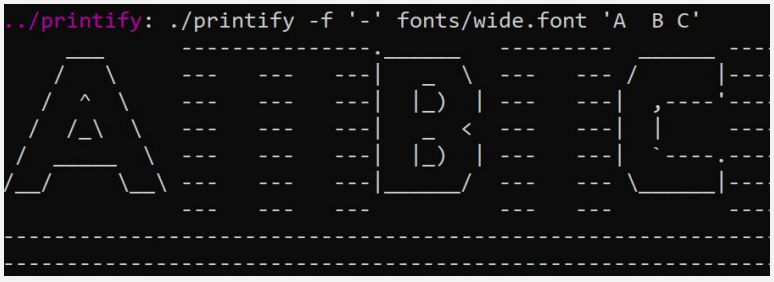
Note that this happens regardless of the number of spaces in your input, as the shell (bash / terminal) removes all whitespace between strings before passing them into your program's argv:

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 and pass the *exact* string `a     wizard` as one argument (one single argv entry) to your program:

Spaces between consecutive cGrams

Between any two consecutive cGrams (including the cGram for the sapce 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. `./printify -f '-' fonts/big.font 'a  b c'`



A note on `getopt()`

You will be parsing not just flags, but also values associated with the flags whose types differ depending on the flags. Below is an example of how you can parse two flags `-a` and `-c`, where `-a` takes an int argument, and `-c` takes a char argument, a font file, 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 -a 5 -c c  cse30.font Hello there
```

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 `"cse30.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`.
`atoi` converts an int to a char.
The string `"a:c:"` in the while loop means that optional flags `-a` and `-c` will 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:")) != -1){
   switch (opt){
   case ('a'):
     printf("Value of arg a is: %d", atoi(optarg));
     // do something for case a with optarg
     break;
   case ('b'):
     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];
```