COM S 229 Project 1 Spring 2014

Part A. Assigned Monday, January 27th. Due Friday, February 21st, 11:59pm.

Part B. Assigned Monday, February 17th. Due Wednesday, March 12th, 11:59pm.

1 Project Summary (Part B)

You will use your existing data structures and procedures from Part A in order to create an *internet meme generator*. For our purposes, an *internet meme* is a picture with silly phrases written over the image. In order to write text, you will be given a bitmap font that is determined by our font definition format. Specifically, you will read a text file containing the specification of the font, which includes the SIMP file containing the bitmap data, and the ranges of the picture that includes each letter.

(a) The Impact Font Image.

(b) The Impact Font, each letter framed.

Figure 1: An example of a font and the letter boundaries.

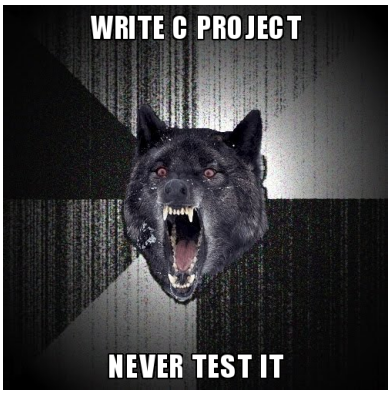
You will collect the letters that you need for the text snippets and overlay these on the specified meme image. Thus, you will create an executable called `meme` that takes the following command-line arguments.

./meme memefile.mem action.act

Where `memefile.mem` is a text file that contains the specifications of the available meme images and fonts, and the `action.act` file contains the specifications for which meme image, font, and phrases to use.

More details on the exact specification for `meme` is included in Section 4.

You will also be required to make all of your software, including the commands from Part A to be free of memory leaks. Thus, every command will be tested using `valgrind` to ensure that no memory errors occur, and that no memory remains reachable at the end of the main method.



(a) Insanity Wolf

(b) Confession Bear

Figure 2: Two memes you can make with the `meme` command.

2 Project Goals

There are several learning objectives for this project. In earlier programming classes, all assignments were described in a detailed manner. One of our goals in this class is to keep the assignment slightly open-ended and underspecified. We want you to learn to think of possibilities and to ask for clarifications (to specify the problem better). Also, we want you to learn to handle complexity by dealing with slightly larger projects with more lines of code. We want to wean you from being too dependent on IDEs by being able to compile and run code from command line and to use makefiles. In addition, of course, we want to make sure you learn to use C.

In addition, you will demonstrate proficiency in the following C programming language features: structures, pointers, memory-management, (binary and text) file input/output, command-line arguments, source code organization, compilation, makefiles.

You may also find the following concepts helpful: unions, debuggers (such as `gdb`), memory error detectors (such as `valgrind`),

3 Grading

The following distribution of points will be used for this part of the project.

Category	**Points**
`meme`	150
Memory Test of Part A	50
`Makefile`	20
`README`	30
Total (Part A):	250

Note that if your code does not build because of compiler or linker errors (either using make or by hand), you will likely lose much more than 20 points (if not all 200 points for the compiled programs).

Effectively: the TAs reserve the right to not grade your project at all if it fails to compile or link. **Test early, and test often** *on pyrite.*

4 Detailed Project Specification

Each text file using the FSF, MEM, or ACT format has data on each line, where the first word of the line contains a set of variable names (separated by spaces), then a colon, then the rest of the line (up to but not including the newline character) is the value of the "deepest" variable. For example, if there is a line of the format

```
VAR1 VAR2 VAR3:This is a line of text.
```

Then there must be a variable of name `VAR1` that contains a variable of name `VAR2` that contains a variable of name `VAR3` whose value should be "This is a line of text."

If the line starts with something other than an allowed variable, that line should be ignored. Some variables include special data that should be parsed specific to that format.

4.1 The FSF Format for specifying fonts

A font is specified by a file using the FSF format. The font consists of an image saved in SIMP format, and then gives a list of characters and the sub-rectangles for each of those characters. You can assume that all of these characters have the same height value, but they may have variable widths.

- `NAME` — Gives the name of the font and will be used as an identifier for the font.
- `IMAGE` — Gives the file name of a SIMP file that contains the bitmap font.
- `CHARACTERc: x y w h` — For the character `c` (which could be any letter, number, or punctuation mark), it specifies the x, y, w, h position to perform a crop operation on the base image in order to get the bitmap image for that character.

Example.

```
NAME:IMPACT
IMAGE:impact.simp
CHARACTERa:0 0 10 20
CHARACTERb:10 0 10 20
```

```
CHARACTERc:20 0 10 20
CHARACTER :30 0 5 20
CHARACTER':35 0 5 20
CHARACTERd:0 20 10 20
```

(This example is incomplete. See the examples on the project web page for more information.)

4.2 The MEM Format for specifying Memes

A meme database is specified by a file using the MEM format. Such a file lists the available memes and fonts. In addition, each meme has a set of locations for the text messages to go, and these are stored as identifiers that are paired with each meme image. The variables for a MEM file are specified below.

- `MEMES` — Gives a list of identifiers, separated by whitespace, of the available memes.
- `FONTS` — Gives a list of FSF files, separated by whitespace, for the available fonts.
- `MEMEID FILE` — Gives the file name for a SIMP file containing the bitmap for that meme.
- `MEMEID TEXTID:x y` — Gives a position in the specified meme for text given by the `TEXTID`, and the string should include the x and y coordinate. Text that is placed at this position should be center-aligned at the x value with the bottom of the text set to the y value. The `TEXTID` will not be `FILE`, as that value is reserved.

Example.

```
MEMES:INSANITY DOGE KEANU
FONTS:impact.fsf comicsans.fsf
INSANITY FILE:insanity.simp
INSANITY TOP:250 30
INSANITY BOTTOM:250 490
```

(This example is incomplete. See the examples on the project web page for more information.)

4.3 The ACT Format for specifying Actions

A meme action is specified by a file using the ACT format. The variables for an ACT file are specified below.

- `OUTFILE` — Gives the name of the SIMP file location for saving the final image.
- `MEME` — Gives the id of the meme to use.

- `FONT` — Gives the id of the font to use.
- `TEXTID` — Gives the string to use at the position specified by the `TEXTID` (specified by the meme).

Example. The action below creates Figure 1(a).

```
OUTFILE:insanity_project.simp
MEME:INSANITY
FONT:IMPACT
TOP:WRITE C PROJECT
BOTTOM:NEVER TEST IT
```

4.4 Detailed Executable Specifications

Consider now the program `meme` with arguments

./meme memefile.mem action.act

This program should load the MEM file specified by argument 1, and the ACT file specified by argument 2. The MEM file contains references to FSF files, so these should be loaded as well. The program should then perform the action specified by the ACT file, to load the given meme image, the specified font, and to overlay the specified text over the meme as specified. The final meme should be saved to the SIMP file specified by `OUTFILE` in the ACT file.

4.5 README

As part of your source-code deliverable, you will write a text file called `README` that contains a description of your project design. The `README` file should contain the following elements:

1. Your Name and NetID
2. A description of your data structure for storing a font, the meme data, for a specific action.
3. A description of any extra details assumed due to ambiguous requirements in the project specification.
4. A file listing of the source code files, specifically mentioning the purpose of each code file.
5. A description of any extra features you added to the programs (or any extra programs you created).

4.6 Compilation and Makefile Requirements

All software must compile and run on `pyrite.cs.iastate.edu` without error. Simply typing `make` should build all of your executables. Also, `make clean` should remove the executables and any object files. You may include other targets for your convenience as you choose (e.g., "`make tarball`")

Your source code should be delivered as a gzipped tarball with the file name `netid.tar.gzip` (where `netid` is replaced by your NetID). Inside the tarball should be *source code only*, no compiled binaries. The grader will type the following commands to compile the software:

```
tar xvf netid.tar.gzip
make clean
make
cat README
```

Thus, your tarball should contain the `Makefile` in the root directory, and compile all binaries to that same directory. In addition, your makefile should use the `-ansi` and `-pedantic` flags when compiling via `gcc`.

If all of your source code is given in `.c` and `.h` files in one directory, you can compress your tarball using the command

```
tar czf netid.tar.gz *.c *.h Makefile
```

4.7 Submitting your work

You should turn in a gzipped tarball containing your source code, makefile, and a README file that documents your work. The tarball should be uploaded in Blackboard.

Your executables will be tested on `pyrite.cs.iastate.edu`. You should test early, and test often on pyrite. We will use some scripts to check your code; this means you should not change the name of the executables or the order of the command-line arguments.

Since all input in this project is given in the form of command-line arguments and by file input/output, your binaries should not output anything to the shell unless there is an error condition. Likewise, they should not take any input from standard in.

5 More Resources

http://en.wikipedia.org/wiki/Computer_font#Bitmap_fonts

http://en.wikipedia.org/wiki/Typesetting