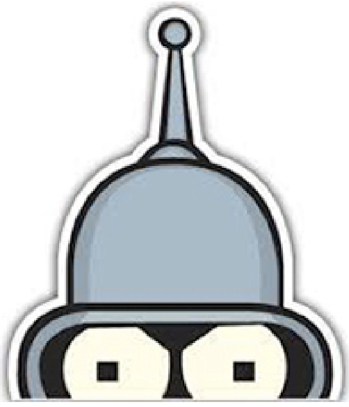
B1 - Elementary Programming in C

B-CPE-110

Bootstrap

Antman



1.0

Bootstrap

binary name:	antman, giantman
language:	any, C
compilation:	via Makefile, including re, clean and fclean rules

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

The challenge of the antman project is to achieve lossless compression of a file.

We will guide you through two exercices. The first one will help you evaluate the efficiency of a compression and could be used as a base for your functional tests. The second one will be a series of thinking and small coding exercises to help you handle compressed data.

If you still haven't, you must start by reading the project's subject. The bootstrap's exercices could be used a the basis for your project.

Functionnal Testing

This part can be done in any language. A scripting language should be prefered.

Step 1 - Mockups

Write the code for a mockup of your `antman` binary. It will not do anything other than printing "COMPRESSED DATA".

Then, write the code for a mockup of your `giantman` binary. It will not do anything other than printing "ORIGINAL UNCOMPRESSED FILE"

Add a folder at the root of your program for test files. In it, you should put a file containing "ORIGINAL UNCOMPRESSED FILE".

This is the moment where you can test your Makefiles and check your repository structure

Step 2 - Script

Write a script that executes your mockup `antman` binary with a test file and redirects its output to a temporary file. It then executes your mockup `giantman` binary with the temporary file as a parameter and redirects the output to another temporary file. Finally it compared the second temporary file to the original test file.

Change your `giantman` output to include an error. Modify your script so that it can detect the error.

Step 3 - Evaluation

Add a functionnality to your script: It should compare the first temporary file (the one containing the compressed data) and the original file and display stats about the compression performance. Try printing "The file was compressed and reduced to X% of its original size."

Step 4 - Automation

Add subfolders to your test files folder, each one corresponding to filetypes. Add multiple test files inside of each of them.

Modify your script so that it runs on every file in every subfolder, displaying stats and errors for each of them.

You can modify your mockups' behavior to react differently to different parameters, but most things at this stage should be hardcoded. This is fine.

Step 5 - Final touches

Add more stats at the end of your script. What filetype can your program handle best? Which one produces the most errors? How do you compare to other compressors you can find online?

You can even add more filetypes or 'tags' if you want. Do you handle small files or big files better? Rap songs or Pop songs? Have you tried compressing audio files? Try thinking of fun challenges for yourself and your classmates.

Reading and Writing

Step 1 - Read and write

Write a first version of your `antman` and `giantman` binaries which read the data and print it back without compressing/uncompressing it.

If you worked on the functionnal tests, there should be no errors and you should get a "The file was compressed and reduced to 100% of its original size." message for every file

Step 2 - Peas in a pod

Let's assume your file will only contain numbers. The entire man ascii, comprised of 127 different characters can now be reduced to only 10 of them. This means that a single char can hold not 1 but 2 symbols. This could represent a 50% compression!

Write a `my_putcstr` function that prints a compressed version of a string of numbers as a compressed version where each char represents two digits.

Write its counterpart `my_readcstr` that reads a compressed string and prints back the original one.

Plug each of them in your binaries and test your first compression attempt.

Think very hard about the values your chars will take. The character storing the "22" sequence should not have a value of 4

Step 3 - Bitwise Operations

The previous method works well because you can fit two 4-bit encoded values in a single char, and then print it. But if you want to work with 5-bit encoded values, your code will be very different. It would be easier to print your bits directly... But you can't write a single bit in a file. The smallest unit you can write is a char (which is comprised of 8 bits)...
Actually, you're a programmer! Everything is possible! In this case, you will have to use bufferization and the `static` keyword. Basically, when you call your `write Bit` function, it saves the bit in a char. When the char is full (you called your function 8 times) you can print it. There will be a delay between the moment you call your function and the moment your bit is written but the result will be the same.
Write this function, and try to use it for the next step.

Step 4 - ABC

The technique from step 2 works when you know you will not use all of the man ascii characters. It means that for each char stored in the original file, some values will never be used. This is a bad thing, because every bit counts and should be optimised. Let's work with files with letters (in both lower and upper case), spaces, newlines, and simple punctuation (,.;!.'?-). This represents only 61 possible values, which can be stored on 6 bits of data. This means that 24 bits can store 4 of such values instead of 3.
Rewrite your functions from step2 to allow them to work with this bigger alphabet.

You might encounter some problems related to padding. Be careful.

Step 5 - Adapt

If you look at the examples we gave you, you'll see that even in the simplest file type, there are some characters that are not in the list of the previous step. But this is fine. The previous method applied to 99% of the file will still yield some good results. You can be less stingy about bit saving wih rare cases and anomalies. For example, having a value 62 (which fits in our 6-bit encoded system) in your file could mean 'The next bytes should be read as a group of 8 because the symbol doesn't fit in our 6-bit encoded alphabet'.

You've already encountered this logic in printf with the % char, with ctrl in shortcuts, with backslashes... Those are symbols that mean 'What comes next should not be interpreted in the same way the other symbols were interpreted'.

Step 6 - Think outside of the box

You should now get consistently good results for the first type of files. Almost 25% reduction in some of the cases. But you may not get results as good with the others. Images will not be composed of many values that are letters or punctuation... Could you generate your alphabet after reading the file to choose the most common values to be encoded in the smallest number of bits? If you have a file that is only numbers on the first half and only letters on the second half, can you use two alphabets? Should you? Can you understand the logic behind the compression in the subject example? How can you improve it? What about filetypes makes them different from each other?

You have a lot to think about, good luck.