p1: Fortune

1. Overview

The Caesar cipher is a very simple way of encoding messages. The idea is that each character (that is a lowercase letter) in the original (or plaintext) message is shifted forward or backward by some number to get the encoded message(or ciphertext). So for instance, given the plaintext

```
attack at dawn!
```

a forward shift of 3 for every character would result in the ciphertext

```
dwwdfn dw gdzq!
```

To decode this message we simply shift back every character by 3.

For this assignment you will compile and run a program that has been provided for you to decode a mystery message that was encoded specifically for you. The original plaintext is a fortune cookie message which has been encoded by a backward shift of all characters by some number. The exact number of shifts to encode your fortune cookie is determined by some computation on your CS login username.

2. Learning Goals

1. Learn some basic Linux commands.
2. Start using a text editor.
3. Become familiar with the build process.
4. Learn how to submit your assignments.

3. Logistics

1. This project must be done individually.
2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or putty as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

4. Linux commands

We'll first use some basic commands to set up the workspace on the CS Linux machines and get the source code for this assignment. This reference (Links to an external site.)Links to an external site. on basic Linux commands is helpful.

1. Open the terminal (Ctrl+Alt+T in Ubuntu) or remotely connect to a CS Linux machine.
2. Make sure you're in your home directory (the directory having your CS username that stores all your files). The command used for finding out where you are in the directory tree is "pwd" (print working directory). The command to change back to your home directory is "cd ~" (change directory)
3. List all the files and directories under your home directory. Try to find out the Linux command to do this in the reference linked above. Find a directory named private among the files and directories listed. Notice there is a directory named 'Public' and another named 'public'. Unlike in Windows, filenames are case-sensitive.
4. Change your current directory to private.
5. Make a new directory named cs354 inside your private directory. Find the appropriate Linux command to do this in the reference linked above.
6. Change your current directory to cs354.
7. Make a new directory named p1 (for project 1) inside your cs354 directory.
8. Change your current directory to p1.
9. Copy the file **decode.c** from the location

```
/p/course/cs354-skrentny/public/code/p1/decode.c
```

to your p1 directory. Find the appropriate Linux command to do this in the reference linked above. You will be using this file to build and run your first program as explained in the following sections.

5. Text editor

Vim is a very popular text editor used in the Linux OS environment, which we recommend you learn to use. Run "vimtutor" on a CS Linux machine to learn the basics. This cheatsheet (Links to an external site.)Links to an external site. to vim keyboard shortcuts is helpful. Other popular text editors are:

1. gedit
2. emacs
3. nano

If you're transitioning from Windows to Linux gedit is the easiest text editor to use when you're in the CS Computer Labs. Using it remotely requires additional configuration of your machine that you'll need to figure out on your own (hint: x forwarding).

6. The Build Process:

The purpose of this part of the assignment is for you to understand the different steps involved in building an executable file from a C program.

6.1. Preprocessing Phase

Build the intermediate file after preprocessing and store it in a file named **decode.i**. The command used to build the intermediate file after preprocessing is:

```
gcc -E -o decode.i decode.c -m32 -Wall -std=gnu99
```

1. Try to understand the meaning of the gcc's command line option "-E". You can use the manual page for gcc to understand this, by typing "man gcc" in the terminal.
2. The option "-m32" is used to generate code for a 32-bit environment since we'll be using this environment to study assembly language.
3. The "-Wall" gcc option is suggested to be used so that you see all of the warnings resulting from your programs.

In preprocessor stage the lines in decode.c beginning with a "#", called preprocessor directives, which are included header files and defined macros, are expanded and merged within the source file to produce an updated source file. Open the file decode.i and you will see that those lines have been replaced with intermediate code. You don't need to understand the intermediate code, just know that the preprocessing step substitutes preprocessor directives like #include<stdio.h> to let the compiler know that the definitions of library functions like printf are defined elsewhere.

6.2. Compilation Phase

The next stage of the process is the actual compilation of the preprocessed source code to assembly language, for a specific processor. So now we are going to stop our compilation after the compiler generates the assembly file. There is an option to let gcc know that it should stop the build process after the compilation phase (named as "compilation proper" in the man page).

Run one of the following commands at the command prompt:

```
gcc <option> decode.c -m32 -Wall -std=gnu99
```

OR

```
gcc <option> decode.i -m32 -Wall -std=gnu99
```

Your task is to find the correct <option> to stop the build process after compilation. HINT: Take a look at gcc's man page!

Open and inspect the generated **decode.s** file in a text editor.

As of now, don't worry about understanding the contents of this file. Just get a feel of how assembly language code looks like. By the end of this semester, you'll be able to understand what most of these lines mean.

6.3. Assembling Phase

We know that computers can only understand machine-level code (in binary). This requires an assembler that converts assembly code in the decode.s file into machine code that the computer can understand.

You are now going to stop the build process after the assembling phase and before the linking phase.

Execute one of the following commands to create the object file **decode.o**:

```
gcc -c decode.c -m32 -Wall -std=gnu99
```

OR

```
gcc -c decode.s -m32 -Wall -std=gnu99
```

Again note that the input to gcc can either be the C source file (decode.c) or the Assembly Code file (decode.s) that was generated from the previous step.

Try opening the binary file (decode.o) in your text editor and see what happens.

View the contents of the object file (decode.o) using a tool named objdump (object dump) as shown below:

```
objdump -d decode.o
```

objdump is a disassembler which converts the machine code to assembly code. A disassembler does the inverse operation of an assembler which converts the the assembly code into machine code.

Understand the use of the command objdump and the meaning of the option “-d” by looking at the man page for objdump or typing “objdump --help” at the terminal.

Save the disassembled output of the object file object.o in a file named **objfile_contents.txt**. One easy way to do this is to redirect the output of the command “objdump -d decode.o” to a file named objfile_contents.txt as follows:

```
objdump -d decode.o > objfile_contents.txt
```

The other way to do this is to simply copy the contents of the output from the terminal and paste it in your text file.

6.4. Linking Phase

This is the final phase of the build process where your object file will be linked with some files in the standard C library to create the final executable file. Execute one of the following commands to create the final executable file.

```
gcc -o decode decode.c -m32 -Wall -std=gnu99
```

OR

```
gcc -o decode decode.o -m32 -Wall -std=gnu99
```

As you might have already guessed, you can provide the original C file or the object file (that we generated in the previous phase) as an input to gcc to generate the final executable file.

Use objdump to view the disassembled contents of the executable file (which is also a binary file) as we did for the object file decode.o.

Redirect the disassembled output that you got to a file named **exefile_contents.txt**. This file should be much larger than the disassembled output of the decode.o file since decode is an executable file which has information combined from decode.o and many library functions like printf.

6.5. What's Typically Used

Even though we have seen each and every intermediate file that was created while we try to compile a C program, the two files that you'll most often use in your real life are the following:

1. C Source File (decode.c)
2. Executable File (decode)

In most cases you would be directly compiling the source file to the executable file using the command with the following format

```
gcc -o <executable-name> <source-file-name> -m32 -Wall -std=gnu99
```

7. Running your first C program

Next we'll run the executable file to decode your encoded fortune cookie.

First copy the file **ciphertext.txt** from the location:

```
/p/course/cs354-skrentny/public/students/<your-cs-login>/p1
```

to your p1 directory. This file contains the fortune cookie encoded under your CS login.
The **ciphertext.txt** and the executable **decode** should both be present in the same directory at this point.

Run the decode executable file and you will be prompted for you CS login. Entering your CS login correctly will give you the decoded fortune cookie which should be a valid phrase in English. See the sample run as shown below.

```
[haseeb@dingo] (8)$ ./decode
Ciphertext:
c eqpenwukqp ku ukorna vjg rnceg yjgtg aqw iqv vktgf qh vjkpmkpi.
Enter your CS login: haseeb
Plaintext:
a conclusion is simply the place where you got tired of thinking.
```

Create a new file named **fortune_cookie.txt** and save your decoded fortune cookie output string as the first and only line of this file.

Make sure that the contents of fortune_cookie.txt are exactly the same as the decoded string and nothing else. We will be matching this string with the original to grade your submission. Include all the punctuation marks which are present in the output plaintext (including the trailing period, exclamation or question mark). It is recommended that you directly "copy" the output from the terminal instead of retyping, to avoid trivial spelling mistakes. In the above sample run,

the fortune_cookie.txt file should only contain the string "a conclusion is simply the place where you got tired of thinking."

Inspect the the code in decode.c and see if you can get an overview of the major steps that the program takes to decode the ciphertext. Understanding the basic primitives and functions being used in the program will help you on the upcoming assignments ahead.

8. Submitting the assignment

The files you need to submit are the following:

1. decode.i (the intermediate file after preprocessing)
2. decode.s (the assembly file after compilation proper)
3. decode.o (the object file after assembling)
4. decode (the executable file after linking with standard libraries)
5. objfile_contents.txt (disassembled output of decode.o object file)
6. exefile_contents.txt (disassembled output of decode executable file)
7. fortune_cookie.txt (decoded plaintext)

You'll need to upload all of the above files **with their names matching exactly those listed above** under Project 1 in Assignments on Canvas on or before the due date. Please DO NOT compress all your files into a single file. Note it is your responsibility to ensure your submission is complete with the correct file names having the correct contents before the due date.

Resubmission: You may resubmit your work repeatedly until the deadline has passed. We strongly encourage you to use Canvas as a place to store a back up copy of your work. If you resubmit, you must resubmit all of your work rather than updating just some of the files. When you resubmit, Canvas will modify the file names by appending a hyphen and a number (e.g. decode-1.s) and these Canvas modified names are accepted for grading.