


p1: A Fortune to Get You Started

[OVERVIEW](#)[LINUX](#)[EDITING](#)[BUILDING](#)[EXECUTING](#)[SUBMITTING](#)

Announcements: Name work log as log-p1.pdf as per rubric

- Released Friday of Week 1. Due on or before Friday of Week 3 but you should complete and submit by Friday of Week 2, so you can work on p2A in Week 3.
- Create a copy of the Google Sheet [Work Session Log](http://tiny.cc/work-log)  (<http://tiny.cc/work-log>)
 - You may remove Deb's sample entries or edit for yourself.
 - Record date and time of each work sessions for this project.
For example we would expect time it took to do the following for this assignment:
 - remote connect,
 - download p1 provided code,
 - read assignment,
 - figure out how to create intermediate build files
 - how to submit
 - Add lines for each internet search or AI chat you make use of.
 - Include Name of any course staff that assist you during that work session and the topic they assisted you with.
 - Include Name and email of any other person that assists you during a work session and the topic they assisted you with.
- When ready to submit
 - Download a pdf of your work log:
File -> Download -> PDF -> Export -> log-p1.pdf
 - Submit your log-p1.pdf with your assignment files.
- It is academic misconduct to falsify this record of your work, and where you sought help. Please keep it as complete as possible, so that it is clear what work and in what order you did to solve this assignment.

OVERVIEW

The Caesar cipher is a simple way to make secret messages. To encode a message, each character is shifted forward or backward in the alphabet. For the message (aka plaintext):

attack at dawn!



a forward shift of 3 for each character results in the secret message (aka. the cipher text):

dwdfn dw gdzq!

To decode this secret message, we simply shift back every character by 3. Note that punctuation is not encoded for this simple approach and we've restricted the characters to lower case.

For this assignment, you'll use a C program we've written to decode a fortune that was encoded specifically for you. The exact shift we've used is based on a calculation using your CS login username. You'll need to get your fortune, get a copy of our source file, add a file header comment to that source file, determine your fortune, and submit it along with some files you'll produced during the build process.

Learning Objectives:

1. Able to use Linux command-line development tools and commands.
2. Able to use built-in Linux text editor(s).
3. Able to manage, create, copy, move, files within filesystem and between systems. Copy files to and from your CSL account.
4. Identify and label the steps and sequence of transformations required to build an executable program from C source code. See this diagram of the [build process](https://www.linkedin.com/pulse/c-build-process-details-abdelaziz-moustafa/)  [\(https://www.linkedin.com/pulse/c-build-process-details-abdelaziz-moustafa/\)](https://www.linkedin.com/pulse/c-build-process-details-abdelaziz-moustafa/).
5. Able to use Ubuntu's Linux gcc to build executable for IA-32 machines that are running Linux.
 1. Understand use and meaning of these gcc options: **-Wall -m32 -std=gnu99**
 2. Able to rename the output of gcc using **-o**
6. Able to submit work as directed for assignment.
 1. Download (copy) files from CSL to local machine
 2. Upload (copy) files from local machine to course filesystems including: cs354-deppeler/handin, Canvas, GradeScope, or other.
7. Record your progress on <http://tiny.cc/work-log>  [\(http://tiny.cc/work-log\)](http://tiny.cc/work-log) for your work on this and each project
 1. If your work is not specific to this project,
8. Submit your work-log to show your work and how your learning progresses.

LINUX Commands

We'll start by using some Linux commands to set up your workspace on the CS Linux machines and get the source file for this project (for more info see [linux_reference.pdf](#)

<https://canvas.wisc.edu/courses/359992/files/33859609?wrap=1>) 

https://canvas.wisc.edu/courses/359992/files/33859609/download?download_frd=1).

1. Remotely connect to a CS Linux machine.

When do your work in person on a CSL Linux machine, you will be able to use this shortcut to open the terminal: **Ctrl+Alt+T** in Ubuntu.
2. Make sure you're in your home directory where your files are located. Change to your home directory using **cd ~** (change directory). Recall that you can find where you're at with **pwd** (print working


`directory`).

3. List all the files and directories under your home directory. See if you can remember the command to do this (if not, look 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`. These are two different directories. Unlike in Windows, filenames are case sensitive.
4. Change to your `private` directory.
5. If you do not have a `cs354` directory inside your `private` directory, create one now. Find the appropriate Linux command in the reference linked above.
6. Change to `cs354` as your working directory (aka current directory).
7. Make a new directory named `p1` inside `cs354`, which you'll use for this project.
8. Change your working directory to `p1`.
9. Copy the file `decode.c` from the location:

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

to your `p1` directory. Find the appropriate Linux command to do this in the reference linked above. This is the source file you'll be using to build and run your first program, which we'll explain below.

EDITING C Source Files

Use **vim**, which is a popular text editor used in the Linux OS environment. Any time you want to learn more vim commands, run **vimtutor** on a CS Linux machine and continue where you left off from the last time. Here's a link to a useful [cheatsheet](https://www.maketecheasier.com/vim-keyboard-shortcuts-cheatsheet/)  (<https://www.maketecheasier.com/vim-keyboard-shortcuts-cheatsheet/>) of vim keyboard shortcuts. Other common text editors are: gedit, emacs and nano. If you're transitioning from Windows to Linux nano and pico are the easiest editors to use when you're working remotely. When you're working on a CS lab computer, using gedit is easiest. To use gedit remotely requires additional configuration of your machine that you'll need to figure out on your own (hint: x forwarding).

We've provided the code for this assignment, but you must edit that source file and add your file header comment as specified in this [Program Commenting Guide](https://canvas.wisc.edu/courses/359992/pages/program-commenting-guide) (<https://canvas.wisc.edu/courses/359992/pages/program-commenting-guide>).

BUILDING C Executable File

Source files can not be executed (run) directly. We must translate the source code into machine code, called an *executable*. This process is typically called compiling. Actually, compiling is just one phase of the process we call "building", which is described below.

1. Preprocessing Phase

Preprocessing is the first phase of the build process, which prepares a C source file for compiling. We can just preprocess the `decode.c` source file and store its result in a file named `decode.i` using the command:

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

- Learn more about gcc's "-E" option by looking at the manual page for gcc. Type `man gcc` at the Linux prompt.
- Recall the `-Wall` option is recommended to be used so that all of the warnings are displayed during the build process.
- Recall the `-m32` option is used to generate code for a 32-bit environment, which we'll be using to study assembly language.

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'll 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>` so that the compiler knows the definitions of library functions like `printf` that are defined elsewhere.

2. Compilation Phase

The next phase of the build process is the compilation of the preprocessed source code. Compiling translates this source to assembly language for a specific processor. Let's stop after compilation to see the generated the assembly file. The option to let gcc know it should stop the build process after compilation can be discovered in the man page under "compilation proper".

Run one of the following commands at the command prompt:

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

OR

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

Protip: Find the correct `<option>` to stop the build process after compilation by taking a look at gcc's `man` page.

Next, open and inspect the generated `decode.s` file in a text editor.

Don't worry about understanding the contents of this file right now; we'll learn more about it after the midterm. For now, just get a feel of how assembly language code looks. By the end of this semester, you'll be able to understand much more of this file.

3. Assembling Phase

Computers can only understand machine-level code (in binary), which requires an assembler to convert the assembly code into machine code that the computer can execute.

Let's now stop the build process after the assembling phase by entering one of the following commands to create the object file `decode.o`:

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

OR

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

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. If you use the source file then all the prior phases will be repeated.

Try opening the `decode.o` file in your text editor and see what happens.

You can view the contents of an object file (`decode.o`) using a tool named `objdump` (object dump) as shown below:

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

`objdump` is a disassembler that converts the machine code to assembly code, which is the inverse operation of the assembler. Understand the use of the command `objdump` and the meaning of the option "-d" by looking at its man page or by typing `objdump --help` at the Linux prompt.

Next, save the disassembled output of the object file `decode.o` in a file named `objectfile_contents.txt`. An easy way to do this is to redirect the output of the command to a file as follows:

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

You could also do this by copying what `objdump` displays on the terminal and paste it in a text file, but that is more error prone.

4. Linking Phase

The last phase of the build process combines your object file with other object files such as those in the standard C library to create the executable file. Execute one of the following commands to create the executable file.

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

OR

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

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 `execfile_contents.txt`. This file should be much larger than the disassembled output of the `decode.o` file since it's an executable file, which has information combined from `decode.o` and library functions like `printf`.

What's Typically Used

We've now seen the steps of the build process and generated intermediate files for each. You'll find that the two files that you'll most often use are:

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

In most cases, you would compile the source file directly to the executable file using a command in this form:

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

CAUTION: the order of the options to C commands can vary, but some options have multiple parts (like the `-o <executable-name>`). The command below also works to build an executable program from the source file.

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

EXECUTING C Programs (executables)

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

First, copy the file `cipher.txt` from your own personal **p1** directory:

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

to your own CS account **p1** directory. This file contains the fortune encoded using your CS login. The `cipher.txt` and the executable `decode` must both be present in the same directory at this point.

Note: If you add CS 354 late, there will be a delay before you get your cipher.txt file. It can take 1-3 weekdays for late adds to appear on the CS department course rosters, which are used to generate these files. Email deppeler@wisc.edu you have added course more than 2 days ago and your p1 directory is still not available.

Run the `decode` executable file and you will be prompted for your CS login. Correctly enter your CS login to get your decoded fortune, which should be a valid phrase in English. See the sample run shown below.

```
[deppeler@liederkranz] (33)$ ./decode
Your cipher text:
c eqpenwukqp ku ukorna vjg rnceg yjgtg aqw iqv vktgf qh vjkpmkpi.
Your CS login: deppeler
Plaintext:
a conclusion is simply the place where you got tired of thinking.
```

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

Make sure that the contents of **myfortune.txt** are exactly the same as the decoded string and nothing else. We'll use a script to automatically match this string with our answer key to grade your submission. Include all the punctuation marks that are present in the plaintext output (including the trailing period, exclamation or question mark). Copy the output from the terminal instead of retyping, to avoid trivial spelling mistakes. In the above sample run, the **myfortune.txt** file should have only one line that is "a conclusion is simply the place where you got tired of thinking." without the quotes.

Before you finish, take a look at the code in **decode.c** and take notes on the major steps this **decode** program. Understanding this can help you with your C programming and upcoming assignments.

SUBMITTING & VERIFYING

Leave plenty of time before the deadline to learn about and complete ALL steps for submission found below. If you can not submit, close all browsers windows, reopen assignment and try again.


[Copy files from CSL file system to your personal computer](https://canvas.wisc.edu/courses/359992/pages/copy-files-from-csl-file-system-to-your-personal-computer)

<https://canvas.wisc.edu/courses/359992/pages/copy-files-from-csl-file-system-to-your-personal-computer>

1.) Always refresh your assignment page before trying to submit your work.

- "Stale" pages, (pages that have been open for more than 12 hours) may have "timed-out".
- Pages that have timed-out will not permit you to upload files.
- To refresh
 - Close all browsers windows
 - Relaunch Chrome
 - Reopen the assignment submit page (with active Submit file button).
 - Submit files

2.) Submit the files listed below under Project p1 in Assignments on Canvas as a single submission. **Do not zip, compress, submit your files in a folder, or submit each file individually.**

- **log-p1.pdf** (a pdf download of your work-log spreadsheet <http://tiny.cc/work-log> 
(<http://tiny.cc/work-log>))
- **decode.c** (the source file with your file header comment added)
- **decode.i** (the intermediate file after preprocessing)

- **decode.s** (the assembly file after compilation proper)
- **decode.o** (the object file after assembling)
- **decode** (the executable file after linking with standard libraries)
- **objectfile_contents.txt** (disassembled output of decode.o object file)
- **execfile_contents.txt** (disassembled output of decode executable file)
- **myfortune.txt** (decoded plaintext)

Repeated Submissions are Encouraged: You may resubmit your work. Each submission must contain all files. We strongly encourage you to submit and use Canvas to store a backup of your current work.

Note: If you resubmit, Canvas will modify your file names by appending a hyphen and a number (e.g., **myfortune-1.txt**). This is expected and does not cause any problems for us or penalty for you. It does cause us problems if you submit files with any suffixes or other names. We can fix wrong file names, but we will deduct 10% for each such fix.

2.) Verify your submission to ensure it is complete and correct. If it is not correct in any way, you must resubmit **all** of your files rather than updating just some of the files.

- **Make sure you have submitted ALL the files listed above.** Forgetting to submit or not submitting one or more of the listed files will result in you losing credit for the assignment.
- **Make sure the files that you have submitted have the correct contents.** Submitting the wrong version of your files, empty files, skeleton files, executable files, corrupted files, or other wrong files will result in you losing credit for the assignment. You must check and submit the correct file before the due date and time. To check your submitted files are correct, you must go to your submission and download the files and examine the contents. It is not enough to see the correct filename in your submission folder.
- **Make sure your file names exactly match those listed above.** If you resubmit your work, Canvas will modify your file names as mentioned in **Repeated Submission** above. These Canvas modified names are accepted for grading.

LATE WORK: Work that is not submitted prior to the DUE Date and Time but is submitted while the assignment is still available, before the availability date and time is marked LATE by Canvas. Work that is submitted LATE is subject to 10% or 20% late penalty depending upon if it is less than or more than 24 hours late.

If you are unable to submit before availability date and time, you may opt to forfeit your Oops point. See [Oops \(https://canvas.wisc.edu/courses/359992/assignments/2045683?wrap=1\)](https://canvas.wisc.edu/courses/359992/assignments/2045683?wrap=1) for information on how to do this.

Points 60

a file upload

Submitting

Due	For	Available from	Until
Sep 22, 2023	Everyone	Sep 9, 2023 at 8am	Sep 24, 2023 at 11:59pm

Project p1

Criteria	Ratings				Pts
Submission contains "decode.c"; "decode.c" has file header comment	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "decode.i"; "decode.i" contains "extern int printf"	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "decode.s"; "decode.s" contains "main"	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "decode.o"; "decode.o" contains correct contents	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "decode"; "decode" contains correct contents	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "objectfile_contents.txt"; "objectfile_contents.txt" contains "main>:" and doesn't contain "_start>:"	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "execfile_contents.txt"; "execfile_contents.txt" contains "_start>:"	5 pts Full Marks	3 pts Any of these two is incorrect		0 pts No Marks	5 pts
Submission contains "myfortune.txt"; get correct contents of "myfortune.txt"	20 pts Full Marks	12 pts A few characters are incorrect		0 pts Wrong or no submission	20 pts
Submission contains "log-p1.pdf" Record the work you do to complete p1. Give estimates to the best of your ability and approximate times.	5 pts Full Marks There should be a log-p1.pdf with log entries and time estimates showing the time it took student to do things like:	4 pts wrong file name must be	3 pts insufficient detail	0 pts Wrong or no submission	5 pts

Criteria	Ratings				Pts
Record any queries and URLs that help you.	remote connect, download p1 provided code, read assignment, figure out how to create intermediate build files. It can be one work session, with multiple lines outline or multiple lines, and it should outline the sequence of work	named log-p1.pdf			
and results. If student worked on p1 on different days and times, that should show up in the log for p1 work.					Total Points: 60