

Assignment 5. Low-level refactoring and performance

[[course home](#) > [assignments](#)]

Introduction

This assignment is designed to give you some skills with low-level programming, which is used in later courses like the operating system class, as well as in real-world applications like the [Internet of things \(IoT\)](#). You'll start with a working program; you'll add a few features, and [tune](#) and [refactor](#) the program to make it better.

Note: Use a private local Git repository (not a repository host like GitHub) to keep track of your work in this assignment when you're modifying code, data, or `notes.txt`. Don't put big output files into your repository; use it only for sources that you maintain by hand.

Useful pointers

- Jeremy W. Sheaffer *et al*, [C tutorial for C++ programmers](#) (2022)
- [Debugging with GDB](#) (2022)
- [Valgrind Quick Start Guide](#) (2022)
- [Valgrind User Manual](#) (2022)

Homework: Tuning and refactoring a C program

Keep a log in the file `notes.txt` of what you do in the homework so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

You're trying to generate large quantities of random numbers for use in a machine-learning experiment. You have a program `randall` that can generate random byte streams, but it has problems. You want it to be (a) faster and (b) better-organized.

You can find a copy of a repository for the `randall` source code in the tarball [randall-git.tgz](#). Unpack that tarball, clone the resulting repository, and look at the resulting source code. It should contain:

- A `.gitignore` file listing built files that Git should ignore (and that you should not commit). Feel free to update this file if it's incomplete for your work.
- A `COPYING` file giving license permissions for this program.
- A `Makefile` that can build the program `randall` and can build the tarball.
- A file `randall.c` that is a single main program, which you would like to modularize.

Add `notes.txt` to your clone of the repository, and commit changes to it as needed while you work on this assignment.

Read and understand the code in `randall.c` and in `Makefile`.

Modify the `Makefile` so that the command `'make check'` tests your program. You can supply just a simple test, e.g., that the output is the correct length. You're doing this step first because you believe in [test-driven development \(TDD\)](#).

Next, split the `randall` implementation by copying its source code into the following modules, which you will need to likely need to modify to get everything to work:

- `options.c` with header `options.h`, which does command-line options processing. Initially there isn't very much of this.
- `output.c` with header `output.h`, which does the output.
- `rand64-hw.c` with header `rand64-hw.h`, which does the hardware-based random-number generation.
- `rand64-sw.c` with header `rand64-sw.h`, which does the software-based random-number generation.
- `randall.c` should contain the main program that glues together everything else. It should include the `.h` files mentioned above, and use their APIs to do its work.

You may add other modules if you like. Each module should `#include` only the files that it needs; for example, since `rand64-hw.c` doesn't need to do I/O, it shouldn't `#include <stdio.h>`. Also, each module should keep as many symbols private as it can.

Next, modify the Makefile to compile and link your better-organized program.

Next, add some options to your program to help you try to improve its performance. Redo the program so that it has an option `-i input`, where *input* is one of the following:

- `rand` – the hardware random-number generation supported by x86-64 processors if available; `randall` should issue an error message and fail if it is not available. This option is the default.
- `mrnd48_r` – the [mrnd48_r](#) function of the GNU C library.
- `/F` (i.e., any argument beginning with `/`) – use the file `/F` as a source of random data, instead of using [/dev/random](#).

Also, redo the program so that it has an option `-o output`, where *output* is one of the following:

- `stdio` – Use `stdio` output, as before. This is the default.
- `N` (a positive decimal integer) – Output `N` bytes at a time, using the `write` system call. If the `write` call reports a successful write of fewer than `N` bytes, do not consider this to be an error; just add the successfully-written number of bytes to your count of bytes written. The last output block might be smaller than usual, if needed to get the total size to be correct. You can use the [malloc](#) function to allocate your output buffer.

You can use [getopt](#) to implement your option processing.

Add some `'make check'` tests to check your additions to `randall`.

When debugging, you may find the `valgrind` program useful. Also, the [AddressSanitizer \(asan\)](#) and the [Undefined Behavior Sanitizer \(ubsan\)](#) may be useful; these can be enabled with the GCC options [-fsanitize=address](#) and [-fsanitize=undefined](#), respectively.

If the program encounters an error of any kind (including option, output and memory allocation failures), it should report the error to `stderr` and exit with status 1; otherwise, the program should succeed and exit with status 0. The program need not report `stderr` output errors.

Finally, time your implementation as follows ...

```
# This is a sanity check to test whether you're in the right ballpark.  
time dd if=/dev/urandom ibs=8192 obs=8192 count=16384 >/dev/null
```

```
time ./randall 133562368 >/dev/null
time ./randall 133562368 | cat >/dev/null
time ./randall 133562368 >rand.data
```

... except that you may need different numbers if your implementation is faster or slower. Also you should try various combinations of the above options to see which gives you random data the fastest. One option that you should try is '[-i /dev/urandom](#)'.

Record your results (including your slow results) in `notes.txt`.

Submit

Submit two files:

1. The file `randall-submission.tgz`, which you can build by running the command `"make submission-tarball"`. Test your tarball before submitting it, by extracting from it into a fresh directory and by running `'make check'` there.
2. The file `randall-git.tgz`, which is a gzipped tarball of your private local Git repository and configuration, created by the command `"make repository-tarball"`.

Neither submitted file should be all that large, since it should contain only information about source files maintained by hand, as opposed to generated files.

All source files should be ASCII text files, with no carriage returns, and with no more than 100 columns per line. The shell command

```
expand Makefile notes.txt *.c *.h |
awk '/\r/ || 100 < length'
```

should output nothing.

© 2020–2023 [Paul Eggert](#). See [copying rules](#).

\$Id: assign5.html,v 1.57 2023/02/16 22:51:46 eggert Exp \$