ECE 2400 Computer Systems Programming, Fall 2017
Programming Assignment Assessment Rubric

`http://www.csl.cornell.edu/courses/ece2400`
School of Electrical and Computer Engineering
Cornell University

revision: 2017-09-21-17-25

This document describes what students are expected to submit for the programming assignments, and how their submissions will be evaluated. A handout is provided for each programming assignment that describes the motivation for the assignment and provides background on the required implementations, testing strategy, and evaluation. Each programming assignment requires you to submit two major parts: the actual code itself (worth 70% of your grade) and a short report (worth 30% of your grade).

Any programmer can write code. *Good programmers* can write code for multiple implementations, use a testing strategy to verify these implementations are correct, and evaluate the performance of these implementations. *Great programmers* can do all of these things, but can also explain their how their implementations work at a high level, justify the specific design choices used in their implementations, use an evidence-based approach to make a compelling argument that their code is correct, and both qualitatively and quantitatively compare and contrast different implementations. Doing well on the code means you are making progress towards being a good programmer. Doing well on both the code and the report means you are making progress towards being a great programmer. By the end of the semester, we hope every student will have evolved from simply being a programmer to being a *great programmer*.

1. Programming Assignment Code Release

Initial code for each programming assignment will be released through GitHub, and students will be using GitHub for all development related to the programming assignments. Every student will have his or her own private repository for the first four programming assignments. Each group of two students will have their own group repository for the last two programming assignments. All of these repositories will be part of the `cornell-ece2400` GitHub organization, and all development must be done in these specific repositories. **You should never fork your remote repository! If you need to work in isolation then use a branch within your remote repository.** The course instructors will merge new code into the each of these remote repositories, and then students simply need to pull these updates.

2. Programming Assignment Code Submission

The code will be submitted via GitHub. You just need to make sure that the final version of your code is pushed to your remote repository on GitHub before the deadline. Automated scripts will clone the `master` branch of each repository at 11:59pm on the due date, and then create an annotated tag to unambiguously denote what version of the code was collected. If you are trying to push last minute changes then it is likely our automated scripts may clone the wrong version. You should make sure your final code is pushed to GitHub at least five minutes before the deadline.

You should browse the source on GitHub to confirm that the code in the remote repository is indeed the correct version. Make sure all new source files are added, committed, and pushed to GitHub.

You should not commit the `build` directory or any generated content (e.g., object files, executable binaries, unit test outputs). Including generated content in your submission will impact the grade for the assignment. **You should confirm that a clean clone of your programming assignment correctly builds and passes all of the tests you expect to pass and also completes the evaluation using the following process:**

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf <repo>
% git clone git@github.com:cornell-ece2400/<repo>

% cd ${HOME}/ece2400/submissions/<repo>/<paname>
% mkdir -p build
% cd build
% ../configure
% make check

% cd ${HOME}/ece2400/submissions/<repo>/<paname>
% mkdir -p build-eval
% cd build-eval
% ../configure --enable-eval
% make eval
```

where `<repo>` is either your NetID for the first four programming assignments or your group repo for the final two programming assignments, and `<paname>` is the name of the programming assignment (e.g., `pa1-math` for the first assignment). If, for any reasons, the above steps do not work, then the score for code functionality will be reduced. For example, students occasionally forget to commit new source files they have created in which case these new files will not be in the remote repository on GitHub.

We will be using TravisCI to grade the code functionality for the programming assignments. So in addition to verifying that a clean clone works on the `ecelinux` machines, you should also verify that all of the tests you expect to pass are passing on TravisCI by visiting the TravisCI page for your repository:

- `https://magnum.travis-ci.com/cornell-ece2400/<repo>`

where `<repo>` is either your NetID for the first four programming assignments or your group repo for the final two programming assignments. If your code is failing tests on TravisCI, then the score for code functionality will be reduced. Keep in mind that in the final few hours before the deadline, the TravisCI work queue can easily fill up. You should always make sure your tests are passing on the `ecelinux` machines and not rely solely on TravisCI to verify which tests are passing and failing.

The functionality of the first and second implementations is worth 40% of the grade for the programming assignment. Verification quality is worth 20% and code quality is worth 10%. Overall, the code portion is worth 70% of the grade for the programming assignment.

3. Programming Assignment Code Revision

We will be incorporating a new aspect into the programming assignment submission process this year. After the deadline, the course instructors will branch your submission and create a pull request on GitHub. The instructors will then commit the instructor tests and evaluation program into this pull request which will trigger a TravisCI build. This will enable the instructors and the students to immediately see how their submission does on both the student tests and the instructor tests. If the student's programming assignment fails some of the instructor tests, then the students are free to fix bugs and commit these changes as part of the pull request. The students are encouraged to add comments to the pull request indicating what they had to change to pass the instructor tests, and why the student tests did not catch this bug. This code revision *will not* mitigate a reduction in the code functionality score due to failing instructor tests, but it *will* enable the course staff to judge how severe a penalty to access. If it turns out that after the students fix a very small mistake in their code, their programming assignment now passes all of the tests then this will result in a small penalty. If it turns out that the students have to fix a major mistake, then this will result in a larger penalty, but at least the students will have figured out what is wrong. Such code revisions will need to be made within a few days of the deadline.

4. Programming Assignment Report Submission

In addition to the actual code, we also require students to submit a programming assignment report. The report offers an opportunity for students to convey the high-level implementation approach, motivation for specific design decisions, evidence for a compelling testing strategy, and evaluation of performance trade-offs. We would argue that the ability to convey this information via a technical report is just as important, or potentially even more important, than simply writing code. The report is worth 30% of the grade for the programming assignment.

The programming assignment report should be written assuming the reader is familiar with the lecture material, *but do not assume that the reader has read the programming assignment handout*; thus you might need to paraphrase some of the content in the handout in your own words to demonstrate understanding. Details about the actual code should be in the code comments. The report should focus on the high-level design, implementation, verification, and evaluation aspects of the assignment. All reports should include a title and the name(s) and NetID(s) of the student(s) which worked on the assignnent at the top of the first page. Do not put this information on a separate title page. The report should be written using a serif font (e.g., Times, Palatino), be single spaced, use margins in the range of 0.5–1 in, and use a 10 pt font size. All figures must be legible. Avoid scanning hand-written figures and do not use a digital camera to capture a hand-written figure. Clearly mark each section with a *numbered* section header. You should include the following sections:

- **Section 1. Introduction –** Students must summarize the purpose of the programming assignment. Why are we doing this assignment? How does it connect to the lecture material? There are often many purposes. Think critically about how the assignments fits into the other programming assignments. Students can paraphrase from the handout as necessary. Students must describe their progress on the assignment. Did you complete the first implementation? the second implementation? Students must include a sentence or two that describes at a very high-level their implementations. Students must include a brief qualitative *and* quantitative overview of the evaluation results (Which implementation performed best? By how much? On which inputs?). Students must include some high-level conclusions they can draw from their qualitative and quantitative evaluation. Do not over-generalize. The introduction should be brief but still provide a good summary of the programming assignment.
- **Section 2. First Implementation –** Students must describe their first implementation. Think critically about what are the key items to mention in order for the reader to understand how the first implementation works. Recall that you cannot assume the reader has read the programming assignment handout. You will likely need to summarize some information from

the handout in your own words. Examples are usually great to include here to illustrate how the first implementation works. Students are highly encouraged to include pseudo-code where appropriate. Do not include C code; your report should be at a higher level. Students must explain why this implementation is interesting to study in the context of the programming assignment. **If the first implementation required some creativity, then students must provide a balanced discussion of not just the implementation itself, but why you chose to take this approach.**

- **Section 3. Second Implementation –** Students must describe their second implementation. Think critically about what are the key items to mention in order for the reader to understand how the second implementation works. Recall that you cannot assume the reader has read the programming assignment handout. You will likely need to summarize some information from the handout in your own words. Examples are usually great to include here to illustrate how the second implementation works. Students are highly encouraged to include pseudo-code where appropriate. Do not include C code; your report should be at a higher level. Students must explain why this implementation is interesting to study in the context of the programming assignment. *Students are encouraged to compare and contrast their implementations to demonstrate understanding of various implementation trade-offs.* **If the second implementation required some creativity, then students must provide a balanced discussion of not just the implementation itself, but why you chose to take this approach.**

- **Section 4. Testing Strategy –** Students must describe the overall testing strategy (e.g., unit testing, directed testing, random testing, whitebox vs. blackbox testing, assertion-based testing, integration testing). Simply saying the students used unit testing is not sufficient; be specific and explain *why* you used a specific testing strategy (e.g., why use directed testing? why use random testing?). Students must explain at a high-level the kind of directed tests cases they implemented and why they used these test cases. Consider including a table with a test case summary, or some kind of quantitative summary of the number of test cases that are passing. Consider including results from code coverage analysis. Note that students are not required to achieve 100% code coverage. Students are trying to provide a compelling, evidence-based argument that their implementations are functionally correct. Code coverage is just one piece of evidence which should be integrated with the types of evidence (e.g., number of tests, types of tests) in this section. We recommend students start this section with a short paragraph that provides an overview of your *strategy* for testing (so how all of the testing fits together). Then you might have a short paragraph for each kind of testing. Each paragraph starts with the "why" (why that kind of testing) and then goes on to the "what" (what did you actually test using that kind of testing). Then you can end with a paragraph that pulls it all together and tries to make a compelling case for why you believe your design is functionally correct. Do not include the actual test code itself; your report should be at a higher level. Do not include the output from running the tests (we can see that on TravisCI). **Remember to provide a balanced discussion between how you tested your design *and* why you chose that testing strategy and test cases.**

- **Section 5. Evaluation –** Students must report their performance results using a table and (if appropriate) a plot. Do not simply include the text output from running the evaluation programs. Format the data so it is appropriate for a report. You must explain how you collected this data (number of subtrials? number of trials? what was the variance?) You must include some kind of analysis of the results: Why is one implementation better or worse than another? Can you predict how the results might change for other inputs? What can we learn from these results? There is not a separate conclusion section, so the big picture summary should really be in the evaluation. **Remember to provide a balanced discussion between what the results are *and* what those results mean.**

It is also always great to include extra material to help demonstrate your understanding. You could include an example of a stack frame diagram like we do in lecture for a small example. You could implement a third implementation to gather additional data points to make for a richer comparative analysis in the evaluation section. You could include a particularly clever test case and reference it in the testing strategy section. If you used a new kind of testing technique then make sure you highlight that in the testing strategy. You could try different evaluation inputs to illustrate a point. Be sure to highlight "extra" work you did in your implementation, testing, or evaluation. There are many creative things you can do to set your report apart!

Sections 1–5 (including the title and author list) should be two to four pages. The maximum is four pages. We do not recommend including diagrams, plots, and tables throughout your discussion since this means you will have less room for text (and puts pressure on making the diagrams, plots, and tables too small). Instead, you can have an appendix at the end of you report which includes all pseudocode, diagrams, plots, and tables. The appendix can be as many pages as you want. Be sure to number your pseduocode, diagrams, plots, and tables and reference them throughout your discussion.

Many students initially struggle with the idea of preparing the programming assignment report. In previous courses, students often simply describe their code at a low level in a programming assignment report. In this course, we are challenging students to prepare reports that better demonstrate the student's understanding of the course content. Before starting to write the report, we encourage students to prepare a detailed outline. The outline should include one section for each of the five sections that will eventually make up the report. Under each section, there should be one bullet for each paragraph the student is planning to include in that section. This bullet should describe the topic of the paragraph. Under each bullet there should be several sub-bullets, one for each topic to be discussed in that paragraph. The outline should also explicitly include references to the figures, tables, and plots the student plans to include in the report. This is called a *structured approach* to technical writing. Students are strongly discouraged from "just starting to write". Just like we should always plan our approach before starting to write our programs, we should plan our approach before writing the report. Students are encouraged to review their outline with the course staff several days before the deadline.

To help students understand our expectations, we have prepared a rough outline for a report for the first programming assignment. It is available on the public course webpage here:

- `http://www.csl.cornell.edu/courses/ece2400/handouts/ece2400-pa1-outline.txt`

Students do not need to follow this outline, nor should students expect a similar outline for future programming assignments. Keep in mind that any outline will evolve as you start writing the report. This outline is simply meant as an example to demonstrate our expectations and our suggested approach for writing great reports; which ultimately will enable students to become great programmers.

5. Programming Assignment Assessment Rubric

The programming assignment assessment is divided into five criteria weighted as follows:

• Code: First Impl Functionality	20%
• Code: Second Impl Functionality	20%
• Code: Verification Quality	20%

• Code: Code Quality	10%
• Report	30%

The assessment for the report is further divided into the following six subcriteria:

• Report: Introduction	($\times 1$)
• Report: First Implementation	($\times 2$)
• Report: Second Implementation	($\times 2$)
• Report: Testing Strategy	($\times 2$)
• Report: Evaluation	($\times 2$)
• Report: Writing Quality	($\times 1$)

In other words, the introduction and writing quality subcriteria are worth half as much as the other report subcriteria. As discussed in the syllabus, each criteria/subcriteria is scored on a scale from 0 (nothing) to 4.25 (exceptional work). The functionality of the implementations is assessed based on the number of test cases that pass in both the student and instructor test suites in combination with the severity of any errors. The verification quality is based on the judgment of the instructor in terms of how well the students' test cases actually test the design. The code quality is based on: how well the code follows the course coding guidelines; inclusion of comments that clearly document the structure, interfaces, and implementation; following the naming conventions; decomposing complicated monolithic expressions into smaller sub-expressions to increase readability. Overall, good code quality means little work is necessary to figure out how the code works and how we might improve or maintain the design.

The following table illustrates a couple different scenarios. In scenario (a), the student did not add any new tests, but (surprisingly) the code still passes all of the instructor tests. The student's code quality is quite poor, and the student did not turn in the report. The final grade is a C/C-. In scenario (b), the student still has not added any new tests, but now the code quality is much improved. The final grade is a C. In scenario (c), the student has added a nice selection of tests but still did not turn in a report. The final grade is a B. In scenario (d), the student completes all parts of the programming assignment, and the report is average. The final grade is an A/A-. In scenario (e), the student completes all parts of the programming assignment, and the report is above average. The final grade is an A. Scores of 4 on the report are relatively rare and are usually reserved for exceptional work.

Criteria	**Weight**	**(a)**	**(b)**	**(c)**	**(d)**	**(e)**
Code: First Impl Functionality	20%	4.25	4.25	4.25	4.25	4.25
Code: Second Impl Functionality	20%	4.25	4.25	4.25	4.25	4.25
Code: Verification Quality	20%	0	0	4.00	4.00	4.00
Code: Code Quality	10%	1.00	4.00	4.00	4.00	4.00
Report	30%	0	0	0	3.50	3.50
Score		1.80	2.10	2.90	3.80	3.95
Grade		C/C-	C	B	A/A-	A

6. GitHub and Academic Integrity Violations

Students are explicitly prohibited from sharing their code with anyone that is not within their group or on the course staff. This includes making public forks or duplicating this repository on a different repository hosting service. Students are also explicitly prohibited from manipulating the Git history or changing any of the tags that are created by the course staff. The course staff maintain a copy of all repositories, so we will easily discover if a student manipulates a repository in some inappropriate way. Normal users will never have an issue, but advanced users have been warned.

Sharing code, manipulating the Git history, or changing staff tags will be considered a violation of the Code of Academic Integrity. A primary hearing will be held, and if found guilty, students will face a serious penalty on their grade for this course. More information about the Code of Academic Integrity can be found here:

- `http://theuniversityfaculty.cornell.edu/dean/the-rules/academic-integrity`