CSI 333 – Programming at the Hardware/Software Interface
Information Regarding Lab Exercises and Programming Assignments

1 Lab Exercises

There will be five equally weighted lab exercises. Each lab exercise will have three parts, namely pre-lab, in-lab and post-lab. The pre-lab part will generally be studying portions of the text or other material plus preparing notes and code samples to enable you to begin the in-lab part. The pre-lab material will be checked early in the lab session. Satisfactory pre-lab preparation is necessary for any credit for the exercise and will be checked in lieu of lab attendance.

The in-lab part must be at least started during the lab session. Credit (“checkoffs”) will be awarded when you demonstrate to any of the course staff the specific exercises according to the instructions printed on the assignment sheet. The exercises will introduce you to specific programming methods, tools and standards. They include documentation and modularization rules and the use of a syntax aiding editor, debugger, version control system and build automation tool (“make”) which are required by the course curriculum. (Professional programmers are often required by their management to follow the “process rules.” Refusal to follow the project process requirements is grounds for getting fired!)

The post-lab parts consist of checkoffs whose completions are optional during the session plus possibly assignments to be done after the session. How they are to be submitted after the session (on paper, electronically or demonstrated to a lab instructor or assistant) will be specified in the assignment. For pre-lab help and post-lab help or checkoffs, you can see any of the lab instructors or assistants, not just your own.

Which checkoffs MUST be demonstrated in the lab session you attend and which can be done before your next lab session will be specified in the lab or in the written assignment.

To get credit, lab checkoffs must be completed PRIOR to the BEGINNING of the NEXT lab session for which you are REGISTERED. You may participate in any lab session in which there is room after REGISTERED students have been given a chance to be seated. In the event of overcrowding in a lab session, students with completed pre-lab material will be given priority. Among those, the students registered in the session will given priority over others.

2 Programming Projects

There will be 4-5 programming projects. Note that the projects are different from lab exercises. The due date and and lateness penalty factor assignment will be indicated on the assignment sheet. No program will be accepted beyond 4-5 days after the deadline. They are really “projects”. That means they entail analysis of the problem parts, learning topics relevant to their solutions, planning for incremental development, design, coding, testing, debugging and the solution of unanticipated problems spread out over the two to three week period over which the project is assigned. There will be pitfalls and problems to solve that you will not know about until you work through the project details! Most students who do no substantial work on a project before a couple of days before the due date will encounter frustration and failure.

2.1 Submission Information

For each program, you must electronically submit the required file(s). The procedure for the electronic submission of your source programs will be documented in the first assignment. (Please note that you should not mail the files to the professor or to any of the teaching assistants.)
Each program should be submitted electronically to the lab section in which you are registered, so your identity will be recorded automatically by lab username. Failure to do so will result in a 5 point penalty.

2.2 Testing

Project assignments will be accompanied by a suite of test case and scripts to automatically run a program against the test cases. More test cases will be added as they are developed and the course staff gains experience with typical problems.

You are expected to learn to use the script to test your own program. Part of the curriculum is to gain experience with using software created by others and written in (script) languages you are not taught explicitly. The scripts will be internally documented (with comments or options to print help). Help with the scripts should be sought through on-line references, questions in the lecture, newsgroup or the course staff, knowledgeable friends, etc.

The tests will be given in a sequence that encourages incremental development. You are also expected to write and submit your own test cases for requirements and implementation issues not covered by ours.

Unlike the rule against sharing solution code, students are encouraged to post test cases on the class newsgroup.

Later projects may require you to code or modify scripts or to write makefile rules to orchestrate automated testing.

Each revision of your program is to be tested with all the tests. The process to test each new revision against all the tests that previous revisions passed, even if you believe the revisions are irrelevant to the old tests, is called regression testing. The testing done by a script makes such regression testing easy and practical.

The 70 points that count for program functionality will be awarded on the following basis:

1. Your report on which of our tests your program passed, verified by our testing.
2. Your own submitted test cases and your report of their results, subject to our verification.
3. Results on secret tests. The secret cases will be published when the grading is completed.

2.3 Code Editing and Documentation

For each programming assignment, approximately 70% of the grade will be for correctness and the remaining 30% will be for structure and documentation.

If your C++/C or assembly code is not consistantly indented to show syntactic structure, 5% will be deducted automatically. It is easy to get your C++/C code properly indented automatically when you use emacs. (Hint: C-C C-Q will re-indent the entire function that the cursor is currently in.)

The documentation comments to be written into each program must include:

(a) You must give your name as the programmer at the beginning of each program file.

(b) An overall description of what the program does must be given, however, it might simply refer specifically to a version or part name or number in the assignment. This is OK for assignments that specify the description in detail.

Any assumptions that you make with respect to the inputs (in addition to those stated in the assignment specification) must be clearly stated. In other words, if you can make your program give correct output under limited conditions, explain them in detail to maximize partial credit.
When you invent yourself what the program does, which may happen in later projects or more advanced courses, that must be written in terms of

1. what the inputs are, and
2. what the outputs are and how they depend on the inputs.

(c) The purpose of every constant, data type and variable declared in your program must be stated at the point of declaration if a declaration is required. For assembly language programs this applies

- to memory variables, both static (allocated in the .data segment) and automatic (allocated in a stack frame) AND
- to registers. Document the purpose of every register you use.

(d) For each function you must provide a description of what the function does in terms of a description of all the parameters, what the values returned (as return value or through reference parameters) are, and how they and any other effects depend on the parameters.

(e) You must also include in-line comments (i.e., comments interspersed with code) to explain any logic that is not obviously expressed by the code itself, such as loop invariants and explanations of how program operation depends on assumptions about the inputs.

Document key uses of variables with invariant comments about relationships among data that are clearly stated and absolutely true whenever control passes the invariant comment. For example, instead of the rather uninformative comments in

```assembly
1bu $s3, str($s2)  #Use string index in $s2 to load a byte into $s3
addi $s2,$s2,1   #Increment counter
```

instead write

```assembly
1bu $s3, str($s2)  #$s3 contains the INPUT char to be processed now.
addi $s2,$s2,1   #$s2 is the number of chars that had been loaded.
```

The better choice for $s3 reminds us that the input string is in memory beginning at address `str` and tells us the significance of the current value in `$s3`. Everybody already knows `1bu` loads a byte. The comment for the `addi` instruction documents the details of how `$s2` is managed. Everybody already knows `addi` increments something!

A useful feature of an invariant comment is it can sometimes be expressed as a test you can program into your code to test itself for internal consistency (also known as “sanity checks”). The C/C++ standard `assert` macro (whose definition comes from `#include <assert.h>`) evaluates a boolean expression and makes the program print the assertion if it is false (“fires”) and then crash. The resulting “core dump file” can be analyzed with a debugger. If the program is run under debugger control, it will stop at the fired assertion.

### 2.4 Modularization Rules

Some of the projects will explicitly require that the program be designed and implemented with separately compiled modules whose interfaces are defined in header files. For those projects, the criteria for the structure, documentation and “Makefile” will be described in the assignment and in the class. These “process requirements” do count for grading purposes. See my remarks about process rules in the lab exercise information above.

The fundamental idea for logical organization of software is to organize it into modules: Each module solves a separate problem and the modules cooperate by using each other’s services or data. Modularization helps make the creation of highly complex software feasible. Now that you know from the prerequisite courses CSI201 and CSI310 some of the computer problems and algorithms that are
feasible to handle in small programs, this course will introduce practices that make feasible more
ambitious, realistically sized projects.

One of these practices is to distribute the code over several different files. Major practical programs
may be written in many thousands of files! The distribution of the code into different files (or directo-
ries, libraries or networked systems too) is called the physical organization. This practice is most
beneficial when:

**The physical organization reflects the logical organization.**

Different programming languages and environments (for example, C, C++, Java, and assembly
languages; and separated tool Unix-like, integrated, and computer aided software design tools) provide
different styles and levels of support for modularization expressed through physical organization. This
course will cover the support provided by traditional C/C++ header, implementation and separately
compiled object files with the C/C++ preprocessor, plus some object orientation.

The course will provide further explanation. Most projects assignments will include or reference
specific rules about what should be in each kind of file. Conformance with these CSI333 modularization
standards will count in the 30 points for structure and documentation.

2.5 Review Questions

1. What is the maximum number of points you will get for a project submission that we cannot
   compile and run?

2. What is a process rule?

3. Give some reasons why it’s not a good idea to start a programming project 2 days or less before
   it is due.

4. What is regression testing? Why do you think it is not done as much as it should be in some
   organizations?

5. What is special about an “invariant comment”?

6. If you read an invariant comment aloud, would it always be a complete sentence? Can it be a
   question?

7. Programming textbooks and lecture material often have code examples with comments to explain
   something about the programming language feature that is being illustrated. Are comments of
   this nature required in this course? Think of some reasons to write them into you project or lab
   exercise code, or into a production project anyway. Think of some reasons to omit them.

8. Experiment with the assert macro in a simple C++ program. Remember to “*include <assert.h>*”
   in the source file. Get the assertion to “fire” while running the program under the gdb or ddd
debugger.

9. Generally speaking, what should go into a header file? Base your answer on what is written
   above.

10. Discuss these terms, preferably in your own words: module, logical organization, physical orga-
    nization.

11. Where are static variables allocated? Where are automatic variables allocated? Why is a stack
    sometimes called a LIFO store? In which course did you learn about stacks? Implement a stack
    in C++. 

4