CS 70: Homework Policy

Melissa O'Neill

Spring 2002

1 Submitting Homework Assignments

Homework assignments are divided into two parts, a written part and a coding part. The written part must be handed in on paper, whereas the coding part must be submitted electronically.

1.1 Submitting Written Work

Written work must be submitted by placing it under my office door (Olin 1258) on or before the deadline. It is acceptable to turn in homework early (for example, in class prior to the deadline), but homework will not be accepted late.

Written homework should be printed on letter-sized paper and be written and printed with readability in mind. You should not waste time on cover pages, presentation folders, and so forth. If you do use such embellishments they will be detached, discarded, and not returned.

1.2 Submitting Code

All coding exercises must be submitted on turing using `cs70submit` (or, alternatively, `cs70submitall`, which is equivalent to `cs70submit *`). You are not required to do all your code development on turing (for example, you could use `g++` and GNU `make` on another platform), but you are required to test your code to ensure that it compiles and operates correctly on turing before submitting your code. We use an automated system on turing to test your code; if your code does not compile it will fail all tests, resulting in a zero score for that portion of the assignment.

Unless otherwise specified, you *must* use the filenames specified in the assignment.

1.3 Frequent Submissions

To paraphrase Al Capone, "submit early and submit often". Every time you reach a milestone (header files completed, code written, code compiles, something starts working even a little bit, etc.) take a break and run `cs70submitall` to submit what you have so far. Doing so has several advantages:

- If the system crashes, you won't have lost everything.
- If you really break your program later, you can easily demonstrate that it used to work.

- If you come to us for help, we can look at earlier versions of your program to find out what went wrong.
- If you get sick or have an emergency that prevents you from completing the assignment on time, you will be able to convince us that you didn't just put things off until the last minute, and you will thus have better grounds for getting an extension.

Similarly, you may ask me or the graders for feedback on your preliminary answers to written problems before the assignment is due.

2 Late Submissions

No work will be accepted late. A submission that is even one second past the deadline, by turing's clock, is considered late. We will make appropriate arrangements for extended or last-minute system crashes, bugs in assignments, vague or broken specifications, or other factors beyond everyone's control. Your responsibility is to inform the course staff promptly if you detect or suspect such a problem. If a problem emerges at the last minute such that we cannot correct it, clearly describe the problem you encountered and what assumptions you made to work around it.

3 Presentation and Style

Your grade is determined in part by your writing and coding style. I value simplicity, conciseness, and readability.

3.1 Written Work

All written work (code documentation and answers to questions) should be written in your best English, using correct spelling and grammar and laid out sensibly on the page. You should already know how to write well-structured English from high school and your humanities classes. If you are in any doubt about any aspects of written English, Harvey Mudd College's Writing Center will be able to help. In addition, the book *BUGS in Writing*[1] can be an invaluable aid to for developing your ear for written English. It is a fun, easy-to-read book targeted at people writing in the sciences, and is available on reserve in Sprague Library.

- Spelling

  Use a spelling checker such as `ispell` to help eliminate typos, and try to be aware of homonyms and use them correctly. We will be relatively lenient about misused homonyms, but we will deduct points for any misspelling that would have been flagged by a spelling checker.

1. Lyn Dupré. Addison Wesley, 1998.

- Apostrophes

  Make sure you use apostrophes correctly: an apostrophe means *ownership*, not plural, except that pronouns never take apostrophes (i.e., we say that John lost *his* textbook, not *hi's* or *his's*; similarly for *hers* and *its*). As for contractions, if you have trouble with them, simply avoid using them. If you write "it is" you may sound a little stuffy but you won't misspell *it's*. (While we're talking about contractions, *can't* is short for *cannot*, not *can not*.)

- Layout

  Your work must be laid out sensibly. In text, lines should be short enough to be readable (i.e., don't fill the entire width of the page with text—appropriate use of margins and other blank space on a page can dramatically improve the readability of your work).

  You can use this handout as an indication of what I consider a sensible font size and line length.

  Unless there is a compelling reason to do otherwise, all pages should be on letter-size paper, with content in the portrait orientation. Pages should be held together with a single staple in the top left-hand corner.

- Whitespace

  Use sufficient whitespace. When writing text files that will be submitted electronically, leave a blank line between paragraphs—whitespace is cheap, and makes your text *much* easier to read. Similarly, leave space between questions on written assignments.

- Consistency

  Whatever writing conventions you use, use them consistently. For example, be consistent about your use of parentheses when referring to functions. If you prefer writing `make_list` to `make_list()`, or vice versa, stick to that style throughout.

- Focus

  Stay on topic. For example, when asked to describe your code, describe it as it was when you documented it and submitted it. It should not be a story about how you constructed the code or a saga about your battles with `emacs`.

Some allowances will be made for special writing difficulties, such as dyslexia or English as a second language, but only if we are informed of these issues beforehand.

3.2 Coding Style

In CS 70, a significant portion of the grade for your code is based on its readability and style. Code that is readable—even if it contains bugs—may score better than obfuscated code that operates flawlessly. To obtain a good style grade, your code should

- Be simple and elegant
- Be neat and clear
- Be consistent and standard

Code should be written so that it may be read and understood by another C++ programmer with as little effort as possible. Affectations that severely impact code readability will be severely penalized. For example, code that shows no attempt at correct indentation, has woefully inadequate comments, or uses ugly/sloppy coding hacks (e.g., using a `goto` statement rather than a **`while`** loop) would be considered unreadable. Similarly, extremely verbose code (e.g., two pages of code for code that can be written in two simple lines) is also considered unreadable.

There is one *instance when* `goto` might *be okay—breaking out of a deeply nested loop.*

3.2.1 Indentation Style

In CS 70, the preferred indentation style is "Stroustrup style" (i.e., the C++ variant of "Kernighan & Ritchie" style), with a block indent of four spaces. You will find it easiest if you code in this style (even though it may take a little getting used to) because example code will always be provided in this style.

Other styles are allowed, provided that you include at the top of the file the popular name of the style (i.e., the name used within `emacs` to refer to the style). The style used in the Weiss textbook is known as `bsd` style within `emacs`. If you wish to use a style not directly supported by `emacs` please see me—I will require a carefully reasoned explanation for why you wish to "reinvent the wheel".

Regardless of your coding style, each file must use the same style throughout. In assignments where you are required to make small modifications to an existing file, this requirement means that you must code that file in Stroustrup style.

3.2.2 Line Length

To encourage you to write code that is elegant and simple, you are required to format your code so that it can be easily read in a window that is 80 columns wide. Excessively long lines are usually a symptom of wider problems with your code.

3.2.3 Comments

Your code needs to be adequately commented. Our requirements for adequate commenting mandate that each source file have a header comment block that includes the file's name and describes its purpose. It is also wise to include your name (preceded by `Author:` or `Modified by:` as appropriate), the date, the course, and the assignment number in this header.

Within your code, you should assume that the person reading the code can understand well-written C++ code and is familiar with common basic data structures, C++ loop idioms, and so forth. A clear coding style, together with informative variable and function names, will reduce the number of comments required. Obscure code or cryptic function names will cause loss of points (for bad style) and also require more extensive comments. We reserve the right to grade excessively impenetrable code as if it did not work.

In addition, your comments need to be laid out consistently with readability in mind. Remember,

- Comments should convey useful information to guide the reader: Don't restate the obvious (e.g., a direct translation of C++ code into English isn't likely to be useful).
- You are not required to comment every line of code. If your functions are straightforward and your identifiers are named meaningfully, you may only need to write a single block comment describing the function.
- When // C++ comments are used to annotate code in the file, the comments should line up on the same column rather than being arranged randomly.
- When a comment consists of one or more full sentences, it should be written using the usual rules for English, including capitalization and spelling. Thus you should write

```
normalizeForm(k);  // Divide coefficients so that k[0] == 1.
```

  and not

```
noFo(k); //       / 2 MAKE 1ST COFF 1.
```

- Keep on topic. Avoid comments and code such as

```
--muZakSUks;  // MY ROOMMAT3 ZACK L15ENS TO YANN1!!!
++r0ck_rul3z;   // AN I WANNA PLAY M3TAL1CA!!! BOO HOO!!!
```

3.2.4 Code Formatting

You may be able to remember the C++ precedence rules well enough to know how the expression `w & x = y << 2 * z` will be evaluated, but you shouldn't assume that your readers can (even experienced C++ coders frequently need to check on the precedence of more esoteric operator combinations). When writing code,

- Order your code so that the most fundamental functions are at the top
- Use case conventions to distinguish between classes (e.g., `StudentGrade`), constants (e.g., `MAX_STUDENTS`), and variables (e.g., `topStudent`)
- Choose sensible and consistent names for functions, classes, and variables
- Avoid using global variables wherever possible
- Name any constants used in the code (avoid "magic numbers")

  Exceptions are `0` *and* `1`.
- Keep data members of classes **private**

- Only provide accessors that make sense
- Follow standard C++ idioms for **for** loops, object construction, and so forth
- Define or disable the standard constructors and assignment operators
- Don't use the C preprocessor to define macros or constants
- Put space around binary (and ternary) operators (+, –, ==, etc.)
- Leave a gap between code and comments. Avoid code such as `a[(-x)++]=1//:-)`
- Leave a gap between multiline comment blocks and code