Program Format Guidelines

Kaminski / cs3310

A good program:

- produces correct output (by the deadline) according to the requirements specifications (as determined by the designer/developer, in consultation with the client), including using:
  - the correct input
  - the proper processing of that input
  - the output format as specified
  - handles all user/input errors properly
- is thoroughly tested (both good and bad data/input, both normal and unusual cases)
- is efficient in terms of run time
- is efficient in terms of memory space (since space translates to run time for OS overhead for paging)
- is well-written in terms of format, style, documentation - to make it:
  - efficient to write/test/debug
  - "easy" to maintain (often by another programmer) – e.g., bug-fixes, future enhancements.

Regarding this last item, this document provides guidelines for commonly accepted practices of "good programming style", regardless of which programming language is used. Use them when writing programs for this course and in future. Develop good habits now – they'll stay with you! Their purpose is not just to provide a "nice, consistent presentation" of a program. They will lead to better, less buggy software in the long run, as well as save a significant amount of time for development, debugging and maintenance – that's **your** time as well as **future programmers'** time and **users'** time.

Consider how much someone would dislike you if they had to make a change in your program a month after you have left the company. Is it readable? Does it use commonly acceptable conventions? (We all prefer the intuitive, "everybody knows" approach, rather than having to read manuals). Does the code "speak for itself" (as much as possible – i.e., self-documenting)? Are the comments clear and helpful (without re-stating the obvious – don't over-comment!)? Do the comments match any newly changed code? Does the code formatting visually reflect the logic of what the program does? Is it easy to see where a modification should be made? Will the effects be local rather than global? Are modules clearly and accurately named (i.e., "truth in advertising")? Are modules physically demarcated and easy to find?

1) Write well-structured, modular, hierarchical programs, e.g.,
   - Create classes and methods (OOP) or separate callable functions (Procedural Progr.) when appropriate
   - Make each method/function/procedure do only one "function"
   - Generally make a module (a method/function/procedure) 1 page/screen or less
   - Main should show the "big picture" of what the program does without including many action details. It the driver/controller (unless the project is an event-controlled program). It should be mainly a series of calls to other modules to do the actual work, plus the overall control structure. It's generally not just a single call.
   - Use a controller module and worker modules rather than using recursive calls to handle "control" (i.e., where each worker module calls the next worker module in turn).
   - Pass parameters to and from generic modules rather than writing near-duplicate code
   - Use information hiding - local variables, variables inside classes, parameter passing vs. global variables
   - Keep local subroutines hidden/private within a module if they're only used there
   - Group variables together to match the conceptual view of the problem (e.g., in record struct's, classes)
   - Use appropriate control structures, e.g.,
     - a `for` loop for count-controlled tasks (vs. a `while` loop with a counter),
     - a `while` loop for event-controlled tasks (vs. a `for` loop with a break out before completion),
     - an `if` statement for conditional execution (vs. a pre-test `while` loop with a `break`)
     - a pre-test `while` loop (vs. adding an `if` to guard a post-test `do…while` loop)
     - a `switch` statement or `if/else if/else if/…` if it's a mutually exclusive set of options (vs. a series of `if`'s)
     - a `for/while/do` structures for looping (vs. a `goto` or recursive calls involving > 1 method)

2) Program for generality e.g.,
   - Define constants at the top of the program/module, then use the constant name within the program
   - Allow the program to handle any number of input records, not just a set number for a test data set

- Use a tester program (driver) to run the real program repeatedly with different test data sets specified as parameters
- Check for overflow when using fixed array storage (regardless of what N the client promises)
- Don't duplicate large chunks of similar code – instead write a generic method which is repeatedly called with the appropriate parameters passed in/out

3) <u>Use a consistent, human readable, commonly-accepted style, so the visual layout reflects the program logic, e.g.,</u>
   - Consistently indent using 4 spaces (or 3 spaces or a tab), but not just 2 spaces (i.e., barely visible)
   - Write one statement per line, generally
   - Align paired things like `if` and `else`, `'{'` and `'}'` etc.
   - Align statements which are at the same level of nesting
   - Don't nest too deeply - call a separate module instead for doing the detailed work
   - It's not necessary to use `//end while` or `//end if` comments unless the body of the structure is long or there are a lot of }'s right together. Proper alignment & indentation makes it visually clearer
   - Put all internal in-line comments way off on the right end (e.g., last 1/3 of the line) rather than embedding them in the code (so they can be visually ignored when examining the code)
   - Use blank lines for visual clarity (without over-doing it)
   - Use a visual separators between modules – e.g., a blank line and a comment line of all *'s between methods/modules and a boxed comment for classes (as below)
   - Use the standard capitalization conventions for your language for naming variables, methods…

4) <u>Provide good internal "documentation"</u>
   - Use the naming which is in the project's Requirements Specifications so it's easier for everyone involved to understand things and communicate about the project
   - Use self-documenting code vs. excessive commenting, e.g., good naming, visual display of logic
   - Choose descriptive variable/constant/class/method/function names, e.g.,
     - n, i, & j are about the only obvious 1 character variable names with particular meanings in CS
     - flag1, flag2 are not descriptive enough, use something like dayFlag and monthFlag instead
     - include "in" and "out" as a prefix of file/record/field names if input and output files are used or if several files are involved; use a prefix like "master"(or m or mast), "error" (or e or err), etc.
   - Use method names which describe the method's function – generally verb&objectItActsOn like WriteLine or ReadARecord
   - Write an initial boxed comment for any program and any physically separate module, including:
     - Name, class, date, assignment# and assignment name
     - Name & brief description of each file used
     - Brief description of what the program does
   - Include a boxed, descriptive comment before each module (which supplements the descriptive module name & parameters) – e.g.,

```
//******************************************************************************
// include such things as (for example):
//     this module's external file name (except a class's name should be the same)
//     brief description of what module does (unless the name "says it all")
//     brief description of parameters (& function/method return value)
//          including which are IN vs. OUT/RETURNED (unless the code "says it all")
//     brief description of any files read/written
//             or interaction with keyboard/screen
//     anything unusual about the module's handling of things
//     any side effects (e.g., accessing global variables)
//     kinds of error checking done or not done
//******************************************************************************
```

   - Line-by-line commenting is generally NOT needed – it makes code harder to read - do it only as needed as on "tricky code", unusual ways handling of things (e.g., to speed up run time), etc.
   - Do not over-comment – comments should help understanding, not just restate the obvious

5) <u>Use good judgment and efficient algorithms:</u>
   - Avoid using tricky, hard-to-understand code, unless it's part of system software or in heavily used parts of the program (i.e., in the 20% of the code which gets 80% of the execution time) - in which case, carefully comment the code and explain how the tricky part helps run-time
   - When faced with choosing between human-efficient code (i.e. design/test/debug/maintenance time) vs. machine-efficient code (run-time, program size) consider how often the program (and this portion of the code) will be run. Sometimes clarity may be more important than run-time efficiency.