Programming Requirements and Guidelines

Every major project has its own coding requirements and style guidelines, which are conventions about how to write code for that project. It is much easier to understand a large codebase when all of the code in it is in a consistent style. Many organizations, such as Google[1], require that all of the code written for its projects conform to these guidelines. In this course, you must write your programs so that they conform to several requirements. Some requirements are related to how and what to submit; others have to do with the form of the code.

The requirements stated in this document are implicitly part of every programming assignment and must be satisfied by your solution to that assignment. They are arranged by category.

1 Requirements

1.1 Submission, Correctness, and Authenticity

1. The submitted program must be free of all errors when it is compiled, linked, and executed on any of the department's `cslabXX` computers. These are the machines in the walk-in lab, 1001B, which are named `cslab1`, `cslab2`, and so on. All of these machines have identical architectures and software, so if a program runs correctly on one, it will run correctly on any other[2]. In general, a program's behavior on one computer may be different than on another because of differences in the installed program libraries, compilers, and the operating system kernels. This requirement stipulates that it must run correctly on these lab machines regardless of what it does on any other computer.

2. ***Every program must be correct to receive full credit.*** "Correct" means that for every possible input, it produces output that is consistent with the specification. If the program produces correct results for some, but not all, inputs, it is not correct. Since there may be an unbounded number of possible inputs, you cannot possibly establish your program's correctness by running it on all inputs. You must use a combination of sampling (i.e., testing) and logical analysis to convince yourself of its correctness. A very common mistake is for a student to hand in a program that does not even run correctly on the input file distributed by the instructor. In other words, the student failed to check the outputs of the program before submitting it. This is either laziness or hubris. It is also very common to make a "small" last-minute change to a program and fail to re-test the program on all inputs, only to learn later that the change "broke" the program completely. Test the exact version that you submit!

3. You must submit all of the source code and absolutely nothing else, unless the assignment states otherwise. Do not submit any executable code, data files, or output files.

4. For full credit, a program must be submitted in the manner described in the assignment by the stated due date. Whether or not it is accepted after the deadline, and if so, how much it will be penalized for lateness, is a rule that will vary from one course to another; whatever is stated in that course's syllabus is the determining rule.

5. *The program must be your work, and your work alone.* You are not free to share solutions or parts thereof with anyone else unless this has been explicitly stated by the instructor. If you do not understand what it does or why it works because someone else's hand is in it, this will be discovered one way or another. You are forewarned that your instructor might ask you to explain how your program works and that you should be able to do so without advance preparation. If you cannot explain it,

[1] The C++ guidelines for Google can be found here: `https://google.github.io/styleguide/cppguide.html`

[2] Unless you are so clever that you have figured out how to make it behave differently depending on which host it is running, in which case you might have "shot yourself in the foot."

then it is not "yours". Representing someone else's work as your own is *plagiarism*, and it is a violation of Hunter College policy. We will file an official complaint against any student who we believe has committed plagiarism.

1.2 Documentation Requirements

Every program must be thoroughly documented. In particular it must satisfy the following documentation rules:

1. Every distinct source code file must contain a preamble with the file's title, author, brief purpose and description, date of creation, and a revision history. It must also contain detailed instructions on how to build the executable from all of the source code files. If the build instructions require multiple commands, creating a Makefile and submitting it with the source code is the best possible solution. The description must be a few sentences long at the minimum. A revision history is a list of brief sentences describing revisions to the file, with the date and author (you) of the revision. This is an example of an acceptable file preamble:

```
/*****************************************************************************
    Title :       draw_stars.c
    Author :      Stewart Weiss
    Created on :  April 2, 2010
    Description : Draws stars of any size in a window, by dragging the mouse to
                  define the bounding rectangle of the star
    Purpose :     Demonstrates drawing with the backing-pixmap method, in which the
                  application maintains a separate, hidden pixmap that gets drawn
                  to the drawable only in the expose-event handler. Introduces the
                  rubber-banding technique as well.
    Usage :       draw_stars
                  Press the left mouse button and drag to draw a 5-pointed star
    Build with :  gcc -o drawing_demo_03 draw_stars.c \
                     `pkg-config --cflags --libs gtk+-2.0`
    Modifications: April 29, 2010
                   Improved efficiency of the algorithm a bit. See the code.
    ****************************************************************************/
```

2. All function prototypes in your program, whether members of a class or not, must have a prologue containing a description of each parameter and the return value, if it has one, as well as appropriate pre- and post-conditions. ***These prologues must not be in the implementation files of classes, but in the class interfaces (i.e., .h files).*** Example:

```
/** get_next(istream & in) resets the values of the command object on
 * which it is called to the values found at the current read pointer of
 * the istream in, provided in.eof() is false.
 * @param istream in [inout]  an istream already opened for reading
 * @pre   istream in is open for reading and in.eof() is false
 * @post  If in.eof() is false on entry to this constructor, then
 *        the command is re-initialized to the values found in the input
 *        stream in, and the istream pointer is advanced to the next line.
 *        If the command is invalid, then when typeof() is called on it,
 *        it will return bad_command.
 *        If in.eof() is true on entry, then the Command_type is set
 *        to null and the remaining values are undefined.
 * @return true if the command was initialized to something other than a
```

```
 *         bad_command, and false otherwise.
 */
bool get_next (istream & in );
```

3. Functions that have non-trivial algorithms must be documented in plain English in a multi-line comment block. All non-trivial declarations must have adjoining, brief comments.

4. Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used. An example adapted from Google's style guidelines:

```
// Iterates over the contents of a BigTable.
// Example:
//     BigTableIterator* iter = table->NewIterator();
//     for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//       process(iter->key(), iter->value());
//     }
//     delete iter;
class BigTableIterator {
    ...
};
```

5. In general, variable names should be descriptive enough to give a good idea of what the variable is used for. Sometimes, more comments are required. Example:

```
int       entrance_id;   // unique positive identifier for this station
string    entrance_url;  // URL of service info webpage for this entrance
string    entrance_name; // string defining location of entrance
GPS       gps_location;  // the GPS location of the station
line_set  entrance_mask; // a bitmask describing lines accessed at this
                         // station
bool      exit_only;     // true if this entrance is only an exit
```

1.3 Style Requirements

1. The names of variables, including function parameters and data members of classes, must be all lowercase, with underscores between words. For example, `line_count` or `table_size` meet this requirement whereas `Big_num` does not. Do not use `camelCase` or `PascalCase` for variable names,

2. Use ***PascalCase*** for class and structure names and type names in general: `TreeInfo` or `ListObject`. Pascal case is the equivalent to Camel case with a starting uppercase letter.

3. Function names can use either method 1 or 2 but must be consistent throughout the entire project.

4. Declared constants should use PascalCase, such as `const int MaxListSize = 100;`

5. Every program must follow commonly accepted stylistic guidelines regarding the use of blank lines, white space, and indentation. In particular

   (a) Each line of text in your code should be at most 80 characters long.

   (b) Use only spaces, not tabs, and indent 4 spaces at a time.

   (c) When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines. In this example, the logical AND operator is always at the end of the lines:

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
    ...
}
```

1.4 Performance

1. Every program must satisfy specified performance requirements if these are stated. This means that it uses an amount of storage and running time within specified or reasonable limits.

2 Miscellaneous Guidelines

There is a difference between requirements and guidelines. A requirement must be followed. A guideline is a suggestion; it is strongly encouraged but does not have to be followed.

1. Programs should avoid error-prone syntax as much as possible. For example, it is better to write the condition

   ```
   if ( 0 == N )
   ```

   than the condition

   ```
   if ( N == 0 )
   ```

   because of the very common mistake of writing `( N = 0 )` instead. Similarly, one should always use braces with compound statements such as if's and while's, even if they are not necessary, as the following example demonstrates:

   ```
   count = N;
   while ( 0 < count ) {
       a[count--]++;
   }
   ```

   because if you make a habit of doing this, you will not be in the situation where you inadvertently write this:

   ```
   count = N;
   while ( 0 < count )
       cout << a[count] << endl;
       count--;
   ```

   and waste time trying to figure out why your program is stuck in an infinite loop.

2. The output of any program should be readable and understandable by ordinary human beings who lack mind-reading capability and who have not read the assignment or the program, unless specified otherwise. For example, the output

   ```
   The file "playlist1" contains 6 songs that won Grammies in 2010.
   ```

   is more understandable than the output

   ```
   6 songs
   ```

   or this

   ```
   playlist1: 6
   ```