Coding Style Guidelines

Following these coding style guidelines will make your code more readable for yourself, your partner in the final project and the people who read and mark your code. This document contains the style we expect you to maintain in all code you submit in CS 246. Some of the following are guidelines to make your code more readable to others and some will make you more efficient. If you do deviate from these guidelines, keep your documentation style consistent [1].

The two guiding principles behind your coding style should be:

Document your code to the style you would want to see others use.

Be consistent. Always.

1 General Information

The following are good practices for programming in any language:

1.1 Naming

You should always choose variable, class, function, and parameter names that describe the purpose of the value it contains or represents. Choosing meaningful variable names will make your code easier to understand when you haven't looked at it for a while and to keep straight what all your variables are used for. Names like $found$, $flag$ and *done* do not describe what each does. While names like i, j, and n do not meet these guidelines, they serve common and understood roles (as indices for i and j, and the number of elements for n).

A related matter is the use of magic numbers (i.e. literal numbers that reappear in your program). Make these constants.

1.2 Whitespace

Proper, consistent indentation emphasizes program structure. It is also important to maintain a consistent format to aid in searching for the beginning and end of control structures. Suggested indentation levels are multiples of 2, 3 or 4. These may also be tabs or whitespaces. However, whitespaces allow much more flexibility and are not editor dependent (e.g. a tab could be the equivalent of 4 or 8 spaces depending on your editor).

[1] This document extensively references material found at:
`https://www.student.cs.uwaterloo.ca/~cs246/current/C++CodingGuidelines.shtml`

Blank lines should be used between control structures, logical breaks in code, and function definitions to improve readability.

1.3 Control Structures

Control structures, such as loops, if statements, and functions, should have a preceding comment describing what it does. The possible states of the program should be listed in these comments.

Closing lines of control structures should have the same indentation of the opening lines and be followed by a comment stating what is being closed.

1.4 Documentation

A good rule of thumb regarding documentation is that you should attempt to answer either "What?" or "Why?" and not "How?". Comments that answer "How?" are often not informative (e.g. "I am looping through this list by incrementing 1"). On the otherhand, answering "What?" and "Why?" offer you the chance to *briefly* describe the algorithm you're using or why you made particular decisions.

Routine documentation should describe what a routine does and not how it does it. Write a summary of the routine. Describe what each meaningful variable is used for (we can figure out what an index i does). State what assumption a routine makes and what it does if it detects an error.

Variables should be accompanied by a comment describing their purpose, if it is not obvious from the variable's name.

A group of statements that perform a non-trivial task should be preceded by a comment summarizing the task. Alternatively, break them into their own function but keep functions simple and to the point.

1.5 Line Length

A line should be no longer than 120 characters long, including comments.

2 C++ Programming Style

2.1 Case Conventions

The following are reasonable (and common) naming conventions for C++:

- Completely in upper-case (VARNAME): macro name, constants
- First letter capitalized (VarName): structures, classes, and typedef
- First letter not capitalized (varName): variable, routines, method names

2.2 Comments

Single line comments should appear before what they are describing if they are long or by an end-of-line.

```
//This is a long comment so not making it an end-of-line comment
statement1;
statement2;                // simple descriptive comment
```

A good multi-line comments looks like:

```
/*
   This comment is longer than one line and must be written
   over multiple lines. Writing it this way increases the readability.
*/
```

The following format is also widely popular:

```
/*
 * This comment is longer than one line and must be written
 * over multiple lines. Writing it this way increases the readability.
 */
```

3 Example

The following is an example of ideal documentation.

```
/*********** findOrInsert ***********
     Purpose: An element, key, is looked up in an unsorted list of items,
          if it does not appear in list, it is added to the end of the list,
          if it exists in list, its associated list counter, listSize, is incremented.

     Returns: Position in list of key.

     Errors: List is full and cannot insert item. Program is terminated.
***********************************/

int findOrInsert( Elem list[ ], int  &listSize, int key ){
    int pos;

    // loop has two exits: search does not find key and key is found
    for ( pos = 0; ; pos += 1 ) {

        if ( pos >= listSize ) {                    // if key not found, insert
            listSize += 1;
            if ( listSize > MaxListSize ) {      // list full ?
                cerr << "ERROR: List is full." << endl;
                exit(EXIT_FAILURE);              // TERMINATE PROGRAM
            } // if
            list[pos].count = 1;
            list[pos].data = key;
            break;
        } // exit

        if ( key == list[pos].data ) {              // if key found, increment counter
            list[pos].count += 1;
            break;
        } // exit
    } // for

    return pos;                                     // return position of key in list
} // SearchInsert
```