A C++ Programming Style Guide for COSC 1030

This information is excerpted from geosoft.no/development/cppstyle.html. We will use it as a basis for our code style for the remainder of the course. These are not meant to be all-encompassing or to be overly restrictive. However, following them will make everyone's life easier.

Naming conventions:

1. Variable names are mixed case starting with lower case. Ex: `line`, `savingsAccount`
2. Type names, like for classes and structs, are mixed case starting with upper case. Ex: `Line`, `SavingsAccount`
3. Named constants (including enumeration values) must be all uppercase using underscore to separate words. `MAX_ITERATIONS`, `COLOR_RED`, `PI`
4. Names representing methods or functions should be verbs and written in mixed case starting with lower case. Ex: `getName()`, `computeTotalWidth()`
5. Functions (methods returning something) should be named after what they return and procedures (*void* methods) after what they do. Ex: `getDate()`, `setDay()`.

   Note: Increases readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.
6. Variables with a large scope should have long names, variables with a small scope can have short names.

   Note: Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside of a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

Files:

1. A class should be declared in a header file and defined in a source file where the name of the files match the name of the class. Ex: MyClass.hpp, MyClass.cpp

   Note: Makes it easy to find the associated files of a given class. An obvious exception is template classes that must be both declared and defined inside a .h file.

2. File content must be kept within 80 columns.

   Note: 80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

3. Special characters like TAB and page break must be avoided. These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

   Note: In VisualStudio, select Tools→ Options → Text Editor → C/C++ → Tabs. Under **Indenting** select *Smart*. Then under **Tab** set the *Tab size* to 2, the *Indent size* to 2, and select *Insert spaces*. This keeps the editor from inserting tab characters.

4. The incompleteness of split lines must be made obvious.

```
totalSum = a + b + c +
           d + e;

function (param1, param2,
          param3);

setText ("Long line split"
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep)
{
...
}
```

   Note: Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.

   In general:

   - Break after a comma.
   - Break after an operator.
   - Align the new line with the beginning of the expression on the previous line.

5. *Include* statements must be located at the top of a file only.

6. The parts of a class should be sorted *public*, *protected* and *private*. All sections should be identified explicitly. Not applicable sections should be left out.

7. Type conversions should always be done explicitly. Never rely on implicit type conversion.

```
floatValue = static_cast<float>(intValue);  // NOT: floatValue = intValue;
```

   Note: By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

8. Variables should be initialized where they are declared. This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared:

```
int x, y, z;
getCenter(&x, &y, &z);
```

   Note: In these cases it should be left uninitialized rather than initialized to some phony value.

9. Use of global variables should be minimized.

   Note: If you cannot justify a global variable as anything other than expedient (simple, easy), do not use it. Find some other way.

10. Class variables should never be declared public.

11. C++ pointers and references should have their reference symbol next to the type rather than to the name.

```
float* x // NOT: float *x;
int& y   // NOT: int &y;
```

    If declaring multiple variables of the same as in

```
float* x, y;
```

    "y" will be of type **float**, not type **float***. You must do the following to have two pointers.

```
float* x, *y;
```

Loops:

1. Only loop control statements must be included in the for() construction.

```
sum = 0;                          //NOT for(i = 0, sum=0; i< end; i++) {
for(i = 0; i< end; i++) {         //      sum += value[i];
  sum += value[i];                //    }
}
```

2. Loop variables should be initialized immediately before the loop.

```
done=false;
while(!done) {
  doSomething();
  done=thingsToDo();
}
```

3. The form while(true) should be used for infinite loops.

4. Executable statements in conditionals must be avoided.

   Note: Conditionals with executable statements are just very difficult to read. Not to mention that this sometimes gives unexpected results. This is especially true for programmers new to C/C++.

5. Functions must always have the return type and value explicitly listed.

```
int checkValue(int value)          \\NOT
  if ( value > 5 ) return 1;       \\  checkValue(int value)
  else return 0;                   \\    if ( value > 5 ) return 1;
}                                  \\  }
```

Layout:

1. Basic indentation should be 2 spaces.

   Note: Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increases the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines.

2. Use one of the two following block layout methods:

```
while(!done) {
  dosomething();
  done = moreToDo();
}
```

```
while(!done)
{
  dosomething();
  done = moreToDo();
}
```

3. Use alignment wherever it enhances readability.

Comments:

1. Tricky code should not be commented but rewritten!

2. All comments should be written in English.

3. Use // for all comments, including multi-line comments.

   Note: This allows the use of the C-style comments (/* */) for quickly commenting out large blocks of code for debugging. Unfortunately, C-style comments cannot be nested.

4. Files, regardless of type, **will** have heading comments. Classes and methods should have heading comments as well.

   File headings should be formatted like

```
// filename.extension
// Student Name
// Date
// Course and Assignment
//
// Further comments describing the use of the file contents.
```

Or

```
/*
 * filename.extension
 * Student Name
 * Date
 * Course and Assignment
 *
 * Further comments describing the use of the file contents.
 */
```

Class comments:

```
// Used to do something. An explanation. Basic
// discussion of its use/reason for existing
//
```

Function comments, if function is non-trivial. A function like "int getVal()" is trivial if it simple returns a element of a class with no other operations:

```
// Returns something.
// Brief discussion of algorithm
// MAKE SURE that you mention any possible side effects.
//
```