Introduction

This document is used to convey common flaws seen during assessment of computer laborations. Note that this is written with generality in ind and to cover all courses we have. We attempt to write what you *should* do rather than what you should not, with variable success. Some points may be irrelevant to your course.
This document is structured in categories (see 2-3) to easier navigate it.

1 General coding style

You can override several recommendations in this section if you *consistently* follow your own style. If you follow the recommendations in a published coding style we recommend you to refer to it in a comment at the beginning of each source file. If your course require a specific coding style you should of course use that.

1-1. Only declare one variable on each line.
This will for example avoid erroneous pointer declaration and initiations.

1-2. Only write one statement per line.
Make sure each line accomplish only one thing, lines with side effects are harder to follow.

1-3. Break up long lines in a suitable way.
At most 80 characters per line is a common assumption in several tools.

1-4. Adjust your formatting to show the semantic meaning. Separate operators from their operands with a space.
Declarations of pointers (*) and references (&) have the operator attached to the data type.
Common binary operators (+, &&) are spaced on each side.
The address-of operator (&) and dereference operator (*) are attached to the expression.

1-5. Use parentheses to control priority and associativity, or simply to improve readability and avoid ambiguity, but avoid needless parentheses.

1-6. Use consistent indentation style.
Avoid double indentation of braces ({}, also referred to as pointy parentheses).

1-7. Start new blocks on its own line and always use blocks in combination with control statements.

1-8. Use English language in code.
Local language is okay in comments and literal strings if used consistently.

1-9. Use a consistent naming style.
This concerns language, use of upper and lower case letters as well as separation of words.
Words are separated by underscore (_), variable names are lower case, constants are upper case and custom types use camel case of each word (My_Own_Data_Type).

1-10. Include C libraries according to C++ rules.
Remove ".h" and prefix the C library name with a 'c'.
Example: <stdio.h> is written as <cstdio> in C++

1-11. Write clear and relevant error messages to users of your programs.

1-12. Avoid exit.
Program start and exit in main - code and resource cleanup will easily become untenable if your program terminate somewhere and other.

1-13. Follow POSIX conventions for main program termination.
Your program (main) will according to convention return zero (0) when the program was successful in its operation, and a non-zero error code for each kind of error the program may detect. Error codes above 255 should be avoided.

1-14. Use constants instead of repeated literal values (DRY - Don't Repeat Yourself).
Constants are allowed in global scope if you need them at several places.

1-15. Use std::cout for normal user hints and std::cerr for error messages.
Example of error situations: missing command line arguments, system errors (for example out of memory), unreadable files etc.

1-16. Avoid unused parameter warnings by omitting the parameter name.

1-17. Use extra indentation for lines following the first line when an expression spans several lines.

2 Program structure

2-1. Follow conventions for program structure.
Includes appear first in programs followed by definitions of important data types and global constants. Then follow declarations of help functions, definition of the main program and last definitions of help functions (in the order declared). It is encouraged to place declarations and definitions of help functions in separate files.

2-2. Use an empty line to separate two logically different parts in the code.
Use empty lines to convey the program structure, but avoid needless emptiness.
Example of different parts: includes, declarations, class definitions etc.

2-3. Groups related functions. See also 2-2.

2-4. Declare variables immediately before the section of code that uses them.
In short functions ($<$ 10 lines), declare all variables at the beginning of the functions. In large functions ($>$ 20 lines), group the statements according to logically different tasks and place declarations at start of each task.

2-5. Avoid complete enumeration of all cases.
Similar tests enumerated in a long ($>$ 2) list of if-statements or conditions should be written in a general way. Use for example a loop, recursion or look-up in a data structure. Have the program compute all cases required, do not do it yourself. To have the program handle yet another case it should be enough to change an input value or add a value to a data set. If you need to add another condition or statement the program is flawed.

2-6. Avoid long or complicated lambda functions.
Implement large or complicated lambda functions as a regular function (when possible) or as a suitable named function object (always possible). This will abstract the complexity of the implementation to the clear and simple name of your function or object.

2-7. Include every library required by the code written in each file.
Specifically, do it even if one include (<string>) happen to be included by another library you use (<iostream>).

2-8. Avoid explicit jumps in your code.
Statements performing jumps hinder readability.
Do not use goto.
In rare cases you can find good motivation to use continue or break.
Avoid more than one return-statement in long (> 5 lines) functions.

2-9. Avoid deep nesting of statements (blocks inside other blocks).
Use functions to move code away from deeply nested blocks or restructure your code.

2-10. Only include header files, not implementation files.
Choose the correct implementation file during compilation.

3 Emphasize the important

3-1. Write function definitions in the implementation file.
Do not write them in the header file.

3-2. Write public members first in class definitions.
Only the public members are interesting to the programmer using your class.

3-3. Remove obtrusive code only present for debugging purposes.
Learn how to trace the execution with a debugger instead of writing a lot of debug messages.

3-4. Move complicated calculations to functions.
Your calculation is thus abstracted to a clear and simple name.

4 Unnecessary code

4-1. Avoid needless variables.
Variables who's purpose is to label intermediate calculations or split a large calculation in simpler steps are okay.

4-2. Do not use the keyword "this" needlessly.
Use "this" only at utmost need (for example "return *this") or to avoid ambiguity (after considering renaming).

4-3. Do not repeat manipulators with permanent effect.
For example setprecision(n) and fixed.

4-4. Avoid specification of class name in front of members in member functions.
Class_Name:: are only required in rare inheritance situations.

4-5. Only include libraries actually in use.

4-6. Do not repeat similar code.
Similar code can be abstracted to a function or otherwise restructured.

4-7. Remove code not required for program functionality.
Code never executed hinder readability. Relate to 3-3.

4-8. Use exceptions to *reduce* complexity of error handling.
If the error can be handled locally you do not need a catch block. Do not catch and re-throw, any uncaught error will automatically be passed down the next catch in your call chain.

5 Self-explanatory code

5-1. Choose names to clearly indicate your code's intention.

5-2. Choose control statements to best match your code's intention.

5-3. Use the standard library.
Use the algorithms and data structures available in the language instead of your on implementations.

5-4. Use reference to constant for input parameters of class type.
Other programmers will then know it's only used as input parameter and not as output parameter.

5-5. Use documenting comments when it is not immediately apparent what your code will achieve.
Comments should describe your code's goal, not how it gets there. The latter is shown by the statements and calculations performed (the code itself). Comments should add information not already immediately apparent. Short comments that abstract the overall purpose of the following 5-10 lines may be suitable. Comments are placed above the code section it regards.

5-6. Avoid negating (!) large or complicated conditions.
Negations make compound conditions harder to comprehend correctly. Relate to 3-4

5-7. Avoid complicated "clever" constructions.
Rather write slightly more but simpler and straightforward code. A clever or tricky optimization is only called for once you proved the straightforward solution is not sufficient to fulfill all requirements.

6 Construction of classes

6-1. Expose only the functions intended for external use in he public part.

6-2. Place members common to related classes in a common base class.

6-3. Base classes with virtual member functions should have virtual destructor.

6-4. Use the “virtual” keyword only in your base class. Use “override” in your subclass to shadow a definition in your base class.

6-5. Always consider all six special member functions.
They are default constructor, destructor, copy constructor, move constructor, copy assignment and move assignment. If you want the compiler-generated defaults (and they are sufficient) you should be explicit with “= default” declaration. If not, use a “= delete” declaration or implement them properly.

6-6. Declare member functions not changing the state of the object as constant.
This will tell other programmers that the object is untouched and allow use of the member function also on constant objects.

6-7. Avoid use of “static”.
Data members declared static correspond to global variables.

6-8. Avoid use of “mutable”.
The “mutable” keyword is used only for data members describing an internal state not visible externally. If you want to change a state visible externally you should use a member function that’s not declared constant.

7 Code safety

7-1. Initialize variables using braces when possible.
The exeption is “auto” that must be initialized by =.

7-2. Use member initialization lists to initialize members in constructors when possible.

7-3. Do not open namespaces in header files (using namespace).
Everyone (including yourself) that include your header do not want that namespace open.

7-4. Use the member versions that give constant iterators.
By using cbegin, cend etc. you improve clarity both for programmers and compiler, which can avoid errors.

7-5. Do not use globally reachable variables.
Global constants are motivated in some cases. Relate to 1-14.

7-6. Check whether file open succeeded and handle and error.
In this case it may sometimes be okay to print the name of the file and terminate the program.

7-7. Use C++ syntax for type conversions.

Always prefer to use data types that avoid all need of conversions. Do not use the crossed out versions:

C-style (unsafe): ~~(int)my_double_var~~

Function-like (unsafe): ~~int(my_double_var)~~

C++; normal conversion (type safe): static_cast<int>(my_double)

C++; forced reinterpretation (use with special care): reinterpret_cast<char*>(&my_double)

C++; conversion between base- and subclass pointers or references (type safe, check result): dynamic_cast<Sub*>(base_ptr)

C++; conversion to change constness (should almost never be called for): const_cast<Type>(my_constant_expression)

7-8. Return resources obtained, for example open files and allocated memory as soon as possible.

7-9. Exceptions should be thrown as value and caught as (const) reference.

Dynamic allocation should not be used (to avoid resource leaks and catch complexity). Catching references will make any subclass caught correctly too.

7-10. Write exception-safe code.

Specifically consider how to handle allocated resources when exceptions are raised.

7-11. Write exception-neutral code.

Your code should be neutral to exceptions. Do not change the exception type or raise new exceptions, only catch exceptions where you can handle it properly.

7-12. Use the input operation as condition in loops.

Usage of functions to check for error bits in the input stream will probably cause one extra input operation.

8 Logic errors

8-1. Possible memory leak.

8-2. Possible use of uninitialized variable.

8-3. Possible use of out of bound index.

8-4. Possible division by zero.

8-5. Possible attempt to store a value outside the types range.

For example overflow in int.

8-6. Possible dereference of an end-iterator or pointer with nullptr-value.

Before you dereference an iterator or pointer you have ensure its validity.

8-7. Input error handling is not correctly accomplished.

8-8. The code do not fulfill set specification.