Technische Universität Carolo-Wilhelmina zu Braunschweig

Fakultät für Architektur, Bauingenieurwesen und Umweltwissenschaften

Masterarbeit

Vorname Nachname

Matrikelnummer

Titel der Arbeit

Institut für rechnergestützte Modellierung im Bauingenieurwesen

Erstprüfer:	Erstgutachter
Zweitprüfer:	Zweitgutachter
Betreuer:	Betreuer

Braunschweig 2017

Contents

List of Figures

Listingsverzeichnis

Abkürzungsverzeichnis

Formelverzeichnis

Chapter 1
Introduction

This document lists C++ coding recommendations common in the C++ development community. The recommendations are based on established standards collected from a number of sources, individual experience, local requirements/needs, as well as suggestions given in [1] - [4].

There are several reasons for introducing a new guideline rather than just referring to the ones above. The main reason is that these guides are far too general in their scope and that more specific rules (especially naming rules) need to be established. Also, the present guide has an annotated form that makes it far easier to use during project code reviews than most other existing guidelines. In addition, programming recommendations generally tend to mix style issues with language technical issues in a somewhat confusing manner. The present document does not contain any C++ technical recommendations at all, but focuses mainly on programming style. For guidelines on C++ programming style refer to the C++ Programming Practice Guidelines.

While a given development environment (IDE) can improve the readability of code by access visibility, color coding, automatic formatting and so on, the programmer should never rely on such features. Source code should always be considered larger than the IDE it is developed within and should be written in a way that maximize its readability independent of any IDE.

1.1 Layout of the Recommendations

The recommendations are grouped by topic and each recommendation is numbered to make it easier to refer to during reviews.

Layout of the recommendations is as follows:

1. Guideline short description
Example if applicable
Motivation, background and additional information.

1.2 Recommendation Importance

In the guideline sections the terms *must*, *should* and *can* have special meaning. A must requirement must be followed, a should is a strong recommendation, and a can is a general guideline.

Chapter 2
Recommendations

2.1 General Recommendations

1. Any violation to the guide is allowed if it enhances readability.
The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

2. The rules can be violated if there are strong personal objections against them.
The attempt is to make a guideline, not to force a particular coding style onto individuals. Experienced programmers normally want to adopt a style like this anyway, but having one, and at least requiring everyone to get familiar with it, usually makes people start thinking about programming style and evaluate their own habits in this area. On the other hand, new and inexperienced programmers normally use a style guide as a convenience of getting into the programming jargon more easily.

2.2 Naming Recommendations

2.2.1 General Naming Conventions

3. Names representing types must be in mixed case starting with upper case (Pascal Casing).

Line, SavingsAccount

Common practice in the C++ development community.

4. Variable names and attribute names must be in mixed case starting with lower case (Camel Casing).

line, savingsAccount

Common practice in the C++ development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

5. Named constants (including enumeration values) must be all uppercase using underscore to separate words.

MAX_ITERATIONS, COLOR_RED, PI

Common practice in the C++ development community. In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:

```
int getMaxIterations() // NOT: MAX_ITERATIONS = 25
{
    return 25;
}
```

This form is both easier to read, and it ensures a unified interface towards class values.

6. Names representing methods or functions must be verbs and written in mixed case starting with lower case (Camel Casing).
getName(), computeTotalWidth()
Common practice in the C++ development community. This is identical to variable names, but functions in C++ are already distingushable from variables by their specific form.

7. Names representing namespaces should be all lowercase.
model::analyzer, io::iomanager, common::math::geometry
Common practice in the C++ development community

8. Names representing template types should be a single uppercase letter.
template<class T>...
template<class C, class D>...
Common practice in the C++ development community. This makes template names stand out relative to all other names used.

9. Abbreviations and acronyms must not be uppercase when used as name.
exportHtmlSource(); // NOT: exportHTMLSource();
openDvdPlayer(); // NOT: openDVDPlayer();
Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readbility is seriously reduced; the word following the abbreviation does not stand out as it should.

10. Generic variables should have the same name as their type.

```
void setTopic(Topic* topic)           //NOT: void setTopic(Topic* value)
                                      //NOT: void setTopic(Topic* aTopic)
                                      //NOT: void setTopic(Topic* t)
void connect(Database* database)      //NOT: void connect(Database* db)
                                      //NOT: void connect (Database* oracleDB)
```

Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.
If for some reason this convention doesn't seem to fit it is a strong indication that the type name is badly chosen.
Non-generic variables have a role. These variables can often be named by combining role and type:
Point startingPoint, centerPoint;
Name loginName;

11. All names must be written in English.

```
fileName; // NOT: dateiName
```

English is the preferred language for international development.

12. Variables with a large scope should have long names, variables with a small scope can have short names.

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*.

13. The name of the object is implicit, and should be avoided in a method name.

```
line.getLength(); // NOT: line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

2.3 Specific Naming Conventions

14. The terms *get/set* must be used where an attribute is accessed directly.

```
employee.getName();
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

Common practice in the C++ development community. In Java this convention has become more or less standard.

15. The term compute can be used in methods where something is computed.

```
valueSet->computeAverage();
matrix->computeInverse()
```

Give the reader the immediate clue that this is a potentially time-consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.