Department of Computer Science
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS121 Lecture Notes: L11

Coding Standards

22 August 2014

Contents

L11.1 Introduction

A coding standard can broadly be defined as a set of programming styles and practices to which a group of people adhere, in the belief that such adherence contributes the overall efficiency in producing high quality code that is understandable and maintainable.

There is ample literature that discusses the benefits of having uniform coding styles and standards. Scott Ambler, the Practice Leader Agile Development at IBM Corporation in the IBM Methods group once said:

> *Inexperienced developers, and cowboys who do not know any better, will often fight having to follow standards. They claim they can code faster if they do it their own way. Pure hogwash. They MIGHT be able to get code out the door faster, but I doubt it. Cowboy programmers get hung up during testing when several difficult-to-find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they're the only ones who understand their code. Is this the way that you want to operate? I certainly do not.* [1]

Coding standards are laid down to achieve robust and error free code that is easy to use, understand and maintain. Style is a crucial component of professionalism in software development. Clean code that follows stylistic conventions is easier to read, maintain, and share with colleagues. When a consistent style is used throughout a project, it makes it easier for the developers working on the project to understand each others code. Oman and Cook [6] found through empirical studies that the style used when writing or maintaining a program has a direct impact upon the quality of the software and the comprehensibility and maintainability of a program.

Adhering to styles and standards based on good programming practices is not only beneficial for sharing and understanding code among programmers. These practices can also increase the productivity of individual programmers and simplify the evaluation of code written by students. Programmers who code in good style are less likely to have silly bugs and will most likely spend less time developing and debugging their code.

L11.2 Classification of coding standards

Coding standards and guidelines can be classified into the following five broad categories [7]:

Style: Guidelines and standards in this category deal with layout issues.

Clarity: These are about measures, besides typographic rules, that can enhance the readability and understandability of code.

Flexibility: In the software engineering industry it is paramount that software should be agile and portable. These standards are guidelines to enhance the adaptability and portability of the code.

Reliability: Reliability concerns the production of user-friendly, robust and error-free code. These are guidelines and practices that are aimed at reducing the chances of making common silly programming errors and reducing chances of program malfunction due to user actions.

Efficiency: Efficiency is about writing elegant code that uses its resources sparingly. Standards aimed at efficiency are rules to follow to utilise resources such as memory, CPU time, disk space, etc. efficiently without compromising other resources such as programmer effort and money.

In the following sections each of these categories are discussed in more detail. Specific standards are specified. Many of these have been adapted form coding standards specifications by Horstmann [2].

You are required to faithfully adhere to these standards, not only in COS132, but in all other Computer Science courses taken at Pretoria University. In any commercial enterprise where software development is important, you will find a similar set of coding standards, probably differing in some of the details, but addressing the categories mentioned below. Because the standards that have been recommended in this text have evolved over several years, they occasionally differ from those followed in the text book.

L11.3 Style

Standards related to style prescribe typographical requirements. The purpose of these standards is to improve consistency and neatness in the appearance of the code. Adherence to these standards enhances program readability.

L11.3.1 Naming conventions

- Use ALL_CAPS for named constants, and camelCase for all other identifier names.
- Identifiers of variables, functions and methods should start with a lowercase letter.
- Class names should be capitalized (start with an initial capital letter).

L11.3.2 Layout rules

- Use indentation and blank lines to reveal the subordinate nature of blocks of code. Each line which is part of the body of a control structure (if, while, dowhile, for, switch) is indented one tab stop from the margin of its controlling line. The same rule applies to function, struct, or union definitions, and aggregate initialisers.
- Use blank lines freely to separate parts of a function or method that are logically distinct.
- Use a blank space around binary operations.
- Leave a blank space after (and not before) each comma, colon or semicolon.

- Lines of code should never extend beyond the right hand side of a resonable window width on the screen. Limiting each line of code to 80 columns will ensure that this is achieved.
- If a hard copy of a program list is made, insert page-breaks to avoid code blocks from spanning over page breaks.
- We do not require specific placement of opening and closing braces. We do, however, require consistency. According to Sutter and Alexandrescu [8], a professional programmer will not have difficulty in reading any of the following styles. Choose one of them and use it consistently throughout all the code of one project.

```
int main()
{
    cout << "Programming is great fun!" << endl;
    return 0;
}
```

```
int main(){
    cout << "Programming is great fun!" << endl;
    return 0;
}
```

```
int main()
    {
        cout << "Programming is great fun!" << endl;
        return 0;
    }
```

L11.4 Clarity

Clarity is about measures, besides typographic rules, that can enhance the readability and understandability of code. The organisation and order of presentation, the careful selection of identifier names, and the content and writing style of comments play an important role in the clarity of code.

L11.4.1 Organisation and Order of presentation

- When the main function calls other functions, they may be defined in the same file. In this case list all function prototypes above the definition of the main function. Their definitions should follow the main function in the same order that their prototypes are listed.
- Functions that are called in the main function may be defined in a different file. In this case list the function prototypes in a header file that is included in the main function's file. Their definitions should be included in a separate source file in the same order as the list of prototypes in the header file.
- In a class definition list all its public members, then all its protected members and lastly all its private members. List instance variables before methods in each section.

- For each class, place the class definition in a header file that is included in the source file that implements the methods of the class. The implementation should be presented in the order in which they are listed in the header file.
- Program sections should be listed and grouped in a logical order that will enhance comprehension.
- The grouping of program sections should maximise cohesion of groups, and minimise coupling between groups.
- The beginning and end of a program block should fit on one screen. Long code sections can always be defined in terms of a number of smaller functions. Between seven and 15 lines of code in a block is a good norm. Do not exceed 30 lines of code in one block.

L11.4.2 Selection of identifier names

Although the compiler only needs a unique character string to identify an entity, programmers also rely on their meaning. Identifier names serve the convenience of readers, and should not serve as a shortcut for the writer.

- Use nouns to name classes and variables.
- Use verbs to name functions and methods.
- Apart from being of the correct word type (noun or verb), it should be reasonably long and descriptive of its purpose in the program.
- Avoid the use of names that are too general.
- Avoid the use of abbreviations (e.g. calc for calculate). Use dictionary words[1]. Exceptions to this rule are using single characters or very cryptic variable names for loop counters and for the parameters of a constructor, provided that these variables are used in an accompanying initialiser list.
- Never use the single characters "O", "o" or "ℓ" (which normally shows on displays as "l"), and avoid using them as the last character in an identifier—these two characters can easily be confused with 0 and 1.

L11.4.3 Commenting practices

Comments are included in code to clarify code and give additional information that cannot be included in the code. The principle is rather to write *self documenting code* than to over comment. It is important to realise that comments cannot rectify bad code. As Oman and Cook [6] put it *"Don't comment bad code - rewrite it"*.

[1]Note that this standard may be violated in handwritten code snippets

Comments are used to enhance the clarity of automatically generated documentation. For this reason comments that are embedded in the code should follow the syntax specified by the documentation generator[2] of your choice.

- Avoid redundancy and duplication of what is already clear in the code. This rule is often violated by programmers who are under the impression that the mere presence of comments serves a purpose. More often than not extra comments obscures more than clarifies.
- Make sure comments and code agree. Often programmers change code without updating the accompanying comments. This is unacceptable. Inaccurate comments are worse than no comments.
- Use a formal writing style to state facts in full sentences that are concise and to the point. Writing concise explanations is often trickier than writing code!
- Every file containing code should start with a comment containing the name(s) and student number(s) of the author(s), the date of last edit as well as the purpose of the file. Use the proper tags for author and date as specified by the documentation generator of your choice.
- Every function definition should be preceded by comments that briefly describe what the function does. Bear in mind that the documentation generator of your choice uses these comments when generating the documentation. Therefore you should use the proper tags as required by the generator. The best way of specifying what a function does is to provide the following:
  - Give the function's *precondition*. Do this by describing each function parameter. For each parameter, indicate any restrictions on the values it may assume in order to guarantee that the function will work correctly. For example, if an integer parameter called `income` is required to be positive for the function to work properly, this should be stated explicitly
  - Similarly, give the function's *postcondition*. Do this by describing what the function will return, or how the function will change the state of the code.

  The pre- and postconditions can be seen as a contract between the person who wrote the function and the person who use the function in a program: if the programmer that uses the function ensures that the parameters comply with the preconditions, then the writer of the function guarantees that the function will comply with its postcondition.

[2] Javadoc can be used when writing Java code and a package like Doxygen is suitable for C and C++ code

L11.4.4 Control structure style

It is extremely important that other programmers (and yourself after a while) are able to follow the program flow of your code. To aid in this respect, you will find it useful to adhere to the following conventions that are aimed at simplicity and clarity of program flow:

- A function or method should be a pure accessor or a pure mutator. Avoid using reference parameters in value returning functions/methods.
- Use jump statements responsibly. These are `goto`, `break`, `continue` and `return` statements that are intended to short circuit a loop or to leave a function or structure at a point other than its end. Minimise the use of `break` statements in loops, and avoid the use of `continue` and `goto` statements altogether. It is preferable to avoid using `return` to break out of a loop—rather terminate the loop gracefully (by having a suitable loop condition) and then `return` after the loop's termination.
- Give preference the use of `while` loops. Use `for`-loops only when a variable runs from somewhere to somewhere with some constant increment/decrement.
- Avoid confusing programming tricks [5].
- Avoid deep nesting of loops and conditionals [5].

L11.5 Flexibility

Flexibility standards are guidelines to assist programmers in building adaptable and portable code. If code is adaptable it can easily be changed and re-used. If code is portable, it can easily be moved to another platform or environment.

- Avoid the use of "magic numbers". A magic number is a numeric constant embedded in code. Rather introduce a named constant. (Recall that named constants should be in capital letters.) An example of the use of a magic number is when you hard-code 3.14 where π is used in a formula. Rather introduce a named constant (e.g. PI).
- Write your programs in a modular fashion. Functions should be used to split up functionality. This splitting should be done in a logical fashion, grouping similar functionality (that makes sense as a unit) together. Avoid too few, as well as too many functions.
- Strive to maximise cohesion within a function and minimise coupling between functions.
- Apply the object oriented programming principles such as modularity, encapsulation and independence.
- Apply the appropriate design patterns wherever possible.

L11.6 Reliability

Code that is written to be flexible is normally also more reliable. This is because when you enhance flexibility (according to the guidelines below) you also contribute to localisation — i.e. information relevant to particular parts of the code is close together. As a result, it becomes easier to isolate possible errors, which in turn renders the code less error prone. Furthermore, code that adheres to standards that are aimed at eliminating human error and enhancing usability will contribute to its user-friendliness and will consequently be more robust and reliable.

L11.6.1 Avoiding logical and runtime errors

The following good habits may lead to code that are less likely to contain logical errors. It may also contribute to the reduction of common runtime errors such as reference to uninitialised objects (segmentation faults) and overflow or underflow.

- Make sure that variables are initialised before they are used. Best is to provide default values upon declaration [4].
- Test your program with data that includes all possible extreme cases as well as all conceivable user misinterpretation.
- Take compiler warnings seriously [4]. It is important to make sure that before you dismiss a warning, you understand exactly what it's trying to tell you.
- For every class with dynamic instance variables explicitly declare a default constructor, copy constructor, assignment operator and destructor.

L11.6.2 Scope and accessibility

Wikipedia [9] describes a side-effect as follows:

> *a function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world.*

Side effects are unexpected behaviour originating from unintended changes of the values of variables in the program. This should be avoided as far as possible.

- All non-final variables (interim values) should be private.
- Keep accessibility as private as possible. Avoid global variables and minimise the use of static variables.
- All features should be explicitly tagged public, protected or private — avoid using the default visibility.
- Define each variable just before it is used, rather than defining all variables at the beginning of a block.
- Avoid having instance variables of a class that could have been defined as non-final variables within the implementation.

L11.6.3 User orientation

The robustness of a program is dependent on how well the user understands its use. For this reason one should strive to write user friendly code.

- Introduce the user to the purpose of the program.
- Avoid clutter on the screen.
- When prompting a user:
  - Be as exact and complete as possible with regard to what would be acceptable input.
  - In command line prompts, end the prompt string with a colon and a space and **no** new line. The user input should be typed on the same line as the prompt.
- When displaying results
  - Display the result in a complete, grammatically correct, sentence
  - Be as exact and complete as possible with regard to the meaning of the result.
  - If applicable include the input values that contributed to the result in the output.
- When displaying an error message
  - Be consistent in the appearance of different error message throughout the program
  - Be as exact and complete as possible with regard to what went wrong. For example, if a file could not be opened, include the name of the file that could not be opened in the error message.

L11.7 Efficiency

Efficiency is about writing code that is elegant and at the same time aware of resource usage. You are advised to read *Effective C++* [4] for a comprehensive discussion of specific ways to improve the effectiveness of your code. Adhering to these guidelines will greatly improve your code. However, one should be aware that even when following the guidelines to the letter, it can not guarantee good code. Efficiency is mostly algorithmic based and is inherently situational. Here we mention only a few very prominent ways to avoid gross inefficiency.

- Never declare variables that are not used.
- For each variable use the smallest data type that will comfortably hold the expected extreme values.
- Avoid the need to apply type casting.

- If the same (or very similar) code appears in more than one place in the program, put it in a function that can be called more than once.
- If a number of consecutive lines of code are the same (or very similar) find a way to specify the operation performed by the code using a loop structure.

L11.8 Conclusion

The purpose of these coding standars is to introduce students to a representative set of coding standards that are typical to professional programming practices and to help students develop the habit of good coding style, which is necessary to successfully complete large programs.

Complying with given coding standards is a vital professional skill required by the software industry. The point of the coding standard is to enhance code quality and uniformity. When complying with such standards code becomes a little easier for everyone to read, and simpler for other people to analyze, debug, and maintain in general.

Li and Prasad [3] observed that most students believe coding standards are important in programming courses but tend not to comply with them. We hope that that this document not only convince students of the benefits of compliance with coding standard but also to provide guidelines to enhance the quality of the code they write and to inspire them to strive for excelence.

References

[1] Ambler SW (2000) Writing robust java code, `http://www.ambysoft.com/downloads/javaCodingStandards.pdf`. [Online; accessed 2008-03-29].

[2] Horstmann CS (2003) *Computing concepts with Java essentials*, Hoboken, NJ: Wiley, 3 edn. Appendix A1.

[3] Li X and Prasad C (2005) Effectively Teaching Coding Standards in Programming, in: *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, 239–244, New York, NY, USA: ACM, URL `http://0-doi.acm.org.innopac.up.ac.za/10.1145/1095714.1095770`.

[4] Meyers S (2005) *Effective C++: 55 specific ways to improve your programs and designs*, Upper Saddle River, NJ 074548: Pearson Education Inc, 3^{rd} edn.

[5] Oman PW and Cook CR (1990) A taxonomy for programming style, in: *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, 244–250, New York, NY, USA: ACM.

[6] Oman PW and Cook CR (1990) Typographic style is more than cosmetic, *Communications of the ACM*, 33(5):506–520.

[7] Pieterse V (2008) Reflections on coding standards in tertiary Computer Science education : festschrift : dedicated to Derrick Kourie, *South African Computer Journal*, 41:29–37.

[8] Sutter H and Alexandrescu A (2004) *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*, Addison-Wesley Professional.

[9] Wikipedia (2013) Side effect (computer science) — Wikipedia, The Free Encyclopedia. [Online; accessed 10-February-2013].