C/C++ Coding Guidelines

Contents

1 Preamble **1**
- 1.1 Copyright Notices 1
- 1.2 Feedback 1
- 1.3 Referenced Documents 1
- 1.4 Definitions 1

2 Introduction **2**
- 2.1 Summary of Guidelines 2

3 Lexical Issues **3**
- 3.1 Naming Conventions 3
- 3.2 Preprocessor 4
- 3.3 Parentheses, Braces and Comma 4
- 3.4 White Space and Indentation 4

4 Statements **4**
- 4.1 Flow Control 4
- 4.2 Return 5
- 4.3 Switch 5
- 4.4 Goto, continue, and break 6

5 Variables **6**
- 5.1 Global Variables 6
- 5.2 Constants 6
- 5.3 Numeric Literal 6
- 5.4 Initialisation 6

6 Functions **7**
- 6.1 Function Length 7
- 6.2 Function Comment 7
- 6.3 Parameters 8

7 ADT and Classes **8**
- 7.1 Abstract Data Type 8
- 7.2 Classes 8

8 Memory **8**

9 Exceptions **8**

10 File Organisation **9**
- 10.1 Introductory Comment 9
- 10.2 Source Files 9
- 10.3 Header Files 9
- 10.4 Functions and Classes 10

11 The Readme File **10**

1 Preamble

1.1 Copyright Notices

1.2 Feedback

Please submit all additions and corrections to Mark Upston (m.upston@curtin.edu.au)

1.3 Referenced Documents

These guidelines are a simplified version of the following documents:

- *Programming in C/C++, Rules and Recommendations*, (1992), by Ellemtel Telecommunications Systems Laboratories.
- *C/C++ Coding Standard*, tmh@possibility.com
- Reiss, S.P., (1999), *A Practical Introduction to Software Design with C/C++*, John Wiley & Sons.
- Deitel, H.M. & P.J., (2000), *C/C++ How To Program* ,Third Edition, Prentice Hall.
- ESA, (1991), *ESA Style Guide for 'C' coding*, Expert Systems Australia Pty. Ltd.
- Horstmann, C.,(1999), *Computing Concepts with C/C++ Essentials*, 2nd Edition, Wiley.

1.4 Definitions

The term *function* as used in this document to mean a callable block of code that performs one or more actions. This is a general definition and makes no distinction whether or not the "block of code" returns a value. If this distinction is important it will be explicitly stated. Based on this definition the term *function* is a synonym for a function, a procedure, subroutine or a method.

2 Introduction

Coding guidelines define a standard for the look and feel of your code. These guidelines are a set of rules describing how your code should look, which features of the programming language you will use and how. The important thing isn't which set of conventions is better, but rather to have standard and use it consistently.

These guidelines will not only help you in reading code from your instructor and classmates[1] but will help your tutors as they read (and grade!) your code. These guidelines should make you a more productive programmer because you do not have to make decisions about trivial matters, you can spend your energy on the solution of real problems. It is also important to remember that most software firms follow strict coding standards, and this is one of the ways to simulate the real world as closely as possible. You are expected to conform to these Coding Guidelines in all your programming.

Deviations from this standard are acceptable if they enhance readability and code maintainability. Major deviations required a explanatory comment at each point of departure so the person assessing your code will know that you didn't make a mistake, but purposefully are doing a local variation for a good cause. If you ignore these guidelines, then your code will be branded as "unacceptable" and will not be marked. Although the language C/C++ is used throughout this document, every attempt has been made to make the guideline as generic as possible. Most of the concepts mentioned here are valid for and can be applied to many programming languages[2] . There is a companion document for the Java language which can be found on the COMP1001 (OOPD) web area.

Should this document contradict or vary from the lecturer or Unit Coordinator, the lecturer or Unit Coordinator shall always take precedence.

2.1 Summary of Guidelines

These guidelines are necessarily somewhat long and dull. They may also mention features that you may not have seen in class. Here are the most important highlights (see relevant section for details):

- tab are set every three or four spaces (3.4);
- variable names are lowercase (3.1);
- classes , functions, enumerators and typedefs are mixed case (3.1);
- constants are uppercase (3.1);
- use a space after keywords, after every comma, after a begin parenthesis and before an end parenthesis (3.3);
- there are spaces between binary operators (3.3);
- place pointer * and reference & signs adjacent to the identifier (3.3);
- braces must lines up (3.3);
- no magic numbers may be used (5.2);
- every function must have a comment (6.2);
- functions should be at most 30-50 lines in length (6);
- no global variables are allowed (5.1);
- no goto, continue, break or multiple returns are allowed (4.4, 4.2); and
- lines should not exceed 78 characters (3.4).

[1]This is when you *help;* not *copy* someone else's code, as that would be misconduct
[2]Future versions may isolate the C/C++ unique components into an Appendix

3 Lexical Issues

3.1 Naming Conventions

One way to improve the readability of your code is to follow certain naming conventions that show what category an identifier is in. You should be able to tell at first glance if an identifier is a typedef, constant or a variable.

Generally, all names must be written in English. Avoid names that don't mean anything (e.g. *foo*) and single character variable names. The exception being loop counters which may use variable names such as *i,j,* and *k*. Any numbers in a name should be written using letters, not numbers, (e.g. *ValueOne*) unless there is a good reason not to. The following rules specify when to use upper- and lowercase letters in identifier names.

- **Files.** All filenames are lowercase. Header files are suffixed with .h (or .hpp for C++), implementation fileswith .c (or .cpp for C++), and class files with .cxx. (The use of the .hpp suffix enables us to differ between C and C/C++ header files.)
- **Functions.** Functions can be made up from several words, the first letter of each word, except the first, in uppercase and the rest in lowercase. The first word must be all lowercase. Acronyms should not beused.

```
void setText( const char *text );
```

- **Classes** Class names can be made up from several words, the first letter of each word in uppercase and the rest in lowercase.

```
class Dialog;
class ModuleManager;
```

- **Constants.** Constants are written in uppercase, multiple words divided by underscore.

```
#define PI 3.1415
const int MEANING_OF_LIFE = 42;
```

- **Enumerators & Typedefs.** Enumerators and Typedefs can be made up from several words, the first letter of each word in uppercase and the rest in lowercase. They should normally not consist of more than three words. Acronyms can be used as long as the meaning is obvious.

```
typedef enum DayType
{
      Monday,
      Tuesday,
      ...
};
```

- **Variables.** Local variables can consist of several words, using only lowercase. Use capitalisation to divide words (camelCase). Use as few words as possible, and use acronyms when needed. Special acronyms like i (index) and x,y (position) are allowed. Parameters should be named in the same way as local variables.

```
int i, index, x, y;
int xAadd;
```

3.2 Preprocessor

Macros are to be avoided, use inline functions where ever possible. Macros can be used when there are good reasons to do so, therefore make sure you include a comment.

3.3 Parentheses, Braces and Comma

Use a space after every comma. Use a space after a begin parenthesis, and a space before an end parenthesis. Do not use such spaces for square brackets. Use spaces between binary operators and no spaces between unary operators. Put pointer * and reference & signs adjacent to the variable they belong to and not the type identifier.

Brace Placement Put begin and end braces on the same column, on separate lines directly before and after the block. For example:

```
void meaningOfLife( int age, int year )
{
    if ( age == year )
    {
       ...
    }
    else
    {
       ...
    }
}
```

3.4 White Space and Indentation

White Space An important way of making you code easy to read is to space it out in a neat and orderly manner.

Empty Lines Use empty lines to group variables and code lines that logically belongs together. Vari able names that are grouped this way must be on the same column. Empty lines must be empty, containing no TABs or spaces. Allow one blank line between a block comment and the function is belongs to. Allow four to six lines at the end of a function before the next block comment. Separate logical chunks of code within a function with blank lines.

Tab Use 3 or 4 character indentation, this is commonly used and allows for easy recognition of the indentation level. Use SPACEs (NOT TABs) to assure the code looks the same whether printed or viewed on the screen.

Line Length Lines should not exceed 78 characters this will ensure that printed output will match the screen on all terminals and printers.

4 Statements

4.1 Flow Control

if, else, while, for, and *do* should be followed by a block (i.e. must use braces {}). If you have many consecutive *if* and *else if* statements, consider using *case* statements instead, or redesign using a more object oriented approach.

if statements, *while*, *do/while* and *for* loops can only execute boolean expressions, never assign or change any variables. The ONLY exception to this is the *for* loop when it modifies the loop counter. Do not check if a boolean expression is equal to true or false as part of the boolean expression!

```
/* BAD EXAMPLE */
   if ( x <= y == TRUE )
   {
      process(x, y);
   }
   else if ( a >= b == FALSE )
   {
      process(a, b);
   }
```

```
/* GOOD EXAMPLE */
   if ( x <= y )
   {
      process(x, y);
   }
   else if ( !(a >= b) )
   {
      process(a, b);
   }
```

4.2 Return

Having a single return statement at the end of a function has the advantage that there is a single, known point which is passed through at the termination of execution of the function. If forces you to write structured code, using *if-then-else* statements “properly”.

Avoid using multiple returns if the code can be just as clearly written with one return. Do not use multiple returns in COMP1000-UCP.

```
/* BAD EXAMPLE */
for ( int i = 0 ; i < max ; i++ )
{
   if ( record[i] == key )
   {
      return TRUE;
   }
}
return FALSE;
```

```
/* GOOD EXAMPLE */
found = FALSE; int i = 0;
while ( ( i < max ) && ( !found ) )
{
   if ( record[i] == key )
   {
      found = TRUE;
   }
   i++
}
return found;
```

The *GOOD EXAMPLE* has the advantage that if we later decide that there is more to do after the search, we do not need to modify the *while* loop but just place the new code between the while loop and the returnstatement.

4.3 Switch

In a *switch* statement each *case* should be terminated with a *break* statement. All *switch* statements should have a default case. All “drop through” *cases* should be clearly documented in a comment before the switch statement. This should be only used when it makes the code simpler and clearer.

4.4 Goto, continue, and break

Goto, continue and *break* statements should not be used. The only exception is using the *break* keyword within a switch statement (see section 4.3).

5 Variables

5.1 Global Variables

Global variable are not to be used.

5.2 Constants

In C use *#define or enum* to define constants.
In C++, do not use *#define* to define constants, use *const* or *enum* instead.

5.3 Numeric Literal

Avoid the use of numeric literals. A numeric literal is an integer constant used in source code without a constant definition. For example:

```
/* BAD EXAMPLE */
if ( customer == 16 )
{
   processTicket();
}
else if ( customer == 7 )
{
   processRefund();
}
else
{
   opps("How did I get here?");
}
```

```
/* GOOD EXAMPLE */
#define HAVEPAID 16
#define COMPLAINED 7

if ( customer == HAVEPAID )
{
   processTicket();
}
else if ( customer == COMPLAINED )
{
   processRefund();
}
else
{
   opps("How did I get here?");
}
```

In the above example what do “16” and “7” mean? If there was a number change or the number were just plain wrong how would you know? Instead of numbers use a real name that means something, as shown in the *GOOD EXAMPLE.*

5.4 Initialisation

Variables should be declared at the top of the function they are to be used in. Ensure every variable is given a value before use, and wherever possible, initialise variables closer to where the decision is made about what its value will be, or where it is used, rather than where it is declared.

This makes it clear that you have considered all paths through the code and the variable is initialised correctly, ensures that variables are not unnecessarily allocated and reduces the risk of a variable being inadvertently hidden.

```
/* BAD EXAMPLE */
int max = 0;
:
:
for ( int i = 0 ; i < n ; i++ )
{
   if ( age[i] > max )
   {
      max = age[i];
   }
}
```

```
/* GOOD EXAMPLE */
int max;
:
:
max = 0;
for ( int i = 0 ; i < n ; i++ )
{
   if ( age[i] > max )
   {
      max = age[i];
   }
}
```

The assignment of *max* is located close to where it is algorithmically significant in the *GOOD EXAMPLE*, whereas it is harder to notice in the *BAD EXAMPLE* (given that there could be a large number of lines of code between the declaration of *max* and the for loop).

6 Functions

6.1 Function Length

Functions should be short and do one task. The length of the function depends on how complex it is. The more complex the function is, the shorter it should be. Too long functions should be split into several functions each doing a minor task. As a general guide, a function should not be longer than 25 lines. If similar chunks of code appear multiple time in your code, you should write a function to handle that operation. Note that good use of stepwise refinement will result in achieving the above requirement as a natural part of the design process.

When following an Object Oriented methodology all functions must be placed inside classes. Exceptions are main() and functions used as an interface between classes and a C API.

6.2 Function Comment

For every function, write a comment that explains:

- what it does;
- any hard-to-understand parameters;
- pre- and post-conditions;
- what input/output and output argument will be set to; and
- what it returns.
- any other general

  comments For example:

```
// NAME:      DateToJulian
// PURPOSE:   Convert calendar date into Julian date
// IMPORTS:   d, m, y the day, month and year
// EXPORTS:   The Julian day number that begins at noon of the given
//            calendar date
// ASSERTIONS:
//     PRE:   d, m, y all represent a valid date
//     POST:  Julian date will be valid
// REMARKS:  This algorithm is from Press et all., Numerical Recipes
//           in C, 2nd ed., Cambridge University Press, 1992.
```

6.3 Parameters

If a function has many parameters it might be wise to split the parameter line into several lines, place the parameters with a close relation to each other on the same line. More than five (5) parameters is often a sign that your function is too big or that an ADT is required. Objects should be passed into a function by using a pass-by-reference or simulate pass-by-value by passing *const* references.

If a function returns exactly one value then define it as a non-void function returning the value. If a function returns more than one value (for example it modifies two parameters) then define it as a *void* function with reference parameters or pointers; or define it as an `int` function where the returned `int` is an error code.

7 ADT and Classes

7.1 Abstract Data Type

A data structure defines the information about some thing (aka an object). All information about an object should be grouped into a struct. This make it obvious the they relate to the same object. Each struct should have its own set of manipulation functions. These functions should have as its first argument the structure it is manipulating.

7.2 Classes

Classes should do only one thing. Split it into subclasses if it's too big. Use multiple inheritance sparingly, preferably use a pointer to an object instead of inheriting it.

All member variables must be private. When access to member variables is required use functions, never use public or protected variables.

Helper functions should be private. Functions that don't access member variables or functions can be made static.

8 Memory

If a function allocates memory for some object, then there should be a matching function to free the memory.

9 Exceptions

Use exceptions to control things that don't work as they were expected to, like parameters having values they should never have. Don't use exceptions as a way of leaving loops or function calls, or signaling an expected outcome. Be aware of memory leaks that may happen when using exceptions.

Functions that use exceptions should explicitly state so in the documentation. When using a function with exceptions, always use a try/catch statement to handle it. Use a try/catch statement in main() to tell the user/developer of any fatal errors that occur.

10 File Organisation

Generally your files should be functionally organised. A file consists of various sections that should be separated by several blank lines. It is important that, within each file you keep a consistent ordering of code components.

10.1 Introductory Comment

Every file must be documented with an introductory comment that provides:

- information on the file;
- author, user name and unit;
- its purpose;
- references;
- other required files;
- date of last modification;
- general comments for example:
  - command line options;
  - file formats;
  - pictures of main data structures;

For example:

```
// FILE:       widget.cpp
// AUTHOR:     Jon Post
// USERNAME:   postj
// UNIT:       IPE152
// PURPOSE:    Event Based Widget manipulation
// REFERENCE: Cohoon J.P. & Davidson J.W., (2000), C/C++ Program
Design,
//             2nd Edition, McGraw-Hill
// LAST MOD:   2001-03-04
// COMMENTS:   The following diagram describes the interaction
between
//             the widget and a user program
//
//             +--------------+             +--------------+
//             |              |Mouse Click |              |
//             |              |----------->|              |
//             | SimpleWidget |             | User Program |
//             |              |Timer Tick   |              |
//             |              |----------->|              |
//             +--------------+             +--------------+
```

10.2 Source Files

The suggested order of sections for a program file is as follows:

- Introductory Comment;
- Header file includes;
- Function implementations (note these should be in some sort of meaningful order).

10.3 Header Files

Every header file should have a header guard.
The suggested order of sections for a header file is as follows:

- Introductory Comment;

- Header file includes;
- constants;
- typedefs, enums;
- structs, classes;
- function prototypes.

10.4 Functions and Classes

Within function and class definitions certain other rules of order apply. For example, within a class it is often useful to keep accessors, mutators, helper functions together within their own respective groups. However, the final grouping/ordering is up to you, as long as it is logical and useful.

11 The Readme File

A *README* file should accompany each of your assignments. This is a **text** file that explains the overall design of your project. When presenting your design, you should discuss why you made the design choices you did. Anyone should be able to explain and defend your design by only reading your *README* (without looking at your code). You should also discuss all known bugs, speculate about their cause, and explain how they could be fixed. Finally you should include a description of any functionality that goes beyond the required specifications (i.e all the bells and whistles you added).