TI C++ software rules

Hogeschool Utrecht, Technische Informatica

Wouter van Ooijen

Version 1.0, 2014-09-30

1. Contents

2. Context

This document gives a rule for C++ code software written by TI students for course modules in the 2nd year of the curriculum or later. They can be used as requirements in the course description, as input for a Fagan inspection, and as motivation for grading code work.

To facilitate the use of this document as input for a Fagan inspection the rules are numbered. In some cases a defect can be covered by more than one rule, in which case the more specific one should be used. When referring to a rule (for instance in a Fagan defect list) both the rule number and the title should be mentioned, and the version of this document must be clear too (probably implicitly because it is an input document).

All rules in this document can be overruled by the course module, either on a module-wide base, or on request from the student or team and approved by the course teacher, on a case-by-case base. For a Fagan inspection, such exceptions should be explicitly provided to the inspectors as input.

This document is not a tutorial: it is up to the appropriate course modules to explain the rules in this document detail, and also to explain why they are worth following.

3. Rules

3.1. Fundamentals

These rules are the central ones. They are more or less in the order of importance. All other rules can be traced back to one of these. If possible, use a more specific rule. When a more specific rule seems to contradict one in this section, this might be an indication that the more specific rule should be overruled (for that specific case).

3.1.1. Original

All submitted code must be the work of the student or team that submits the code, unless

- The assignment allows the use of 3d party code, and
- The 3d party code is clearly recognizable as such from the comment header at the start of the file.

3.1.2. Correct

Code must be correct, that is: it must satisfy its requirements. This applies to all requirements, functional and non-functional (e.g. resource limits).

Note that this rule cannot be checked without explicit requirements.

3.1.3. Readable

After being correct, code must be readable. Code must be written to aid readers and users of the code.

3.1.4. Maintainable

After being readable, code must be maintainable. This implies for instance that small changes in the requirements should cause only small and localized changes in the code.

3.1.5. Efficient

”correct is better than fast”

After being maintainable, code must not be inefficient (use more resources than needed) when this can easily be avoided. Resources include (but are not limited to) CPU time, memory use, and power consumption.

Note that explicit timing requirements do not fall under this rule, they fall under the (more important) rule that software must be correct (= satisfy its requirements).

Remember Hoare’s dictum: Premature optimization is the root of all evil. Don’t optimize at the cost of the higher goals (correctness, readability, maintainability), and if you must optimize (because you don’t meet explicit speed requirements) get some measurements first. Programmers are nearly always wrong when guessing which part of the code needs optimization.

3.2. Comments

3.2.1. English

All comments must be in English.

3.2.2. File header

Each file must have a comment header that states at least

1. The author (name and student number)
2. The context (year, module, assignment)

3.2.3. No magic values

Magical values must be explained. This applies to any value other than 0 and 1, and in some contexts even those values might need explaining. Note that, as always, a well-chosen name is better than a comment.

When possible, literal values must be declared as constexpr. (Not yet implemented in MSVC 2013)

3.2.4. Comment WHY

Comments must focus on WHY a piece of code does what it does, not HOW it does it.

When a piece of code is read the reader has immediate access to that piece of code, so he can (in theory) deduce what it does. But he has no (immediate) access to the places where the code is used, nor to motivation of the programmer to write the code the way he did. Hence comments must explain the WHY. The HOW needs commenting only if it is not immediately obvious, but the need for lots of HOW-type comments is often an indication of other problems. Meaningful names and recognizable code constructs (idioms, patterns) that need no explanation other than the mentioning of the construct are better than comments.

3.2.5. Use //

The // style comments must be used instead of the /* … */ style. The /* ... */ style doesn't nest, and in the middle of a large block it is not immediately apparent whether a line is commented out or not.

If a piece of code must be disabled for some temporary reason, use #ifdef with a meaningful name, like

```
#ifdef NOT_TESTED_YET
. . .
#endif
```

This requirement (and all others) do not apply to temporary code that is not part of a deliverable.

3.3. Layout

3.3.1. Indentation

A recognizable indentation style must be used consequently, and its appearance must not depend on a particular editor or printer.

Recommended: 1TBS, 3 spaces per indent.

3.3.2. No tabs

Tabs must not be used to format code, because different editors and printers have different ways to interpret tabs.

Makescripts are an exception because they can can’t be written without TABs.

3.3.3. ASCII

A source fill must contain only printable ASCII characters (including en-of-line delimiters).

3.3.4. Line length

The line length must be limited to 80 characters. A line that is folded is less readable, and the fold will appear at different places depending on the editor, windows size, or printer used.

3.3.5. Parentheses

Parentheses must be used to avoid reliance on the more obscure priority rules (which is almost anything beyond the familiar MVDWOA).

3.3.6. Full bracing

The body of a loop (for, while, or do-while) and the alternatives of an if statement must be enclosed in parenthesis, except when an else clause contains (only) an if statement (chained if’s).

```
while( s[ i ] != ’\0’ ){
   i++;
}

if( a > b ){
   max = a;
} else {
   max = b;
}

if( line == ”melon” ){
   color = yellow;
} else if( line == ”cherry” ){
   color = red;
} else {
   throw data_error( ”unknown product” );
}
```

3.3.7. Complex expressions

If it reasonably fits, a function call must be all on one line. If not, it must be broken at or near the opening parenthesis, and the arguments and closing brace must be formatted according to your indent style. Complex expressions must be formatted likewise.

```
draw_circle( 12, 48, 15, color::white );

draw_circle(
   x_root + 4,
   y_root + 40,
   color::invert( color::black )
);
```

3.4. Architecture

This section has a large overlap with design decisions that should be made before any code is written (which is outside the scope of this document). When rules in this section are violated this is an indication that those design decisions must be reconsidered, in particular the class structure (or maybe its realization in code).

3.4.1. Cohesion

Code elements that share a responsibility must be grouped together. Suitable grouping mechanisms include a function, a class, a method within a class, a template, or even a file. Responsibilities must not be stretched wider than necessary.

3.4.2. Layering

Code that operates at different levels of abstraction must not be mixed. An extreme example would be mixing hardware pin access and game strategy.

3.4.3. Separation

Code that operates on different entities must not be mixed. An example would be mixing the implementation of writing to a LCD with the implementation of beeping on a speaker.

3.4.4. Balance

A design must strive to have components of roughly equal size and complexity. Entities (classes, functions, files, etc.) that are either very small or very large can be an indication of a bad architecture.

3.4.5. No repetition

Identical or near-identical code fragments must be factored out into common-path code or (probably better) a separate function, method, class or template (or, but only as last resort, as a macro).

3.4.6. Hide details

Information that is not needed by the user of an interface must be hidden. Hence methods and attributes that are not part of the interface of a class must be private, and declarations that are not part of an interface must not appear in an interface header file. Likewise, declarations that are not part of the interface of a source code file must not appear in its header.

One example of failure to hide details is providing set and get methods for every attribute of a class. Such access methods must only be provided when they are part of the user interface of the abstraction that the class implements.

For technical reasons, templates must be fully in header files.

3.4.7. Explicit interfaces

Communication between separate parts of code must be explicit. If a function or method needs outside information it is better to pass it in parameters than to store it in a global variable. For a method such data, should be part of the object’s state only when it is really part of the objects state.

The std::ostream formatting mechanism is a an example of how this can wrong: formatting settings should be part of the print operation, not of the ostream. Another infamous example is errno.

3.4.8. Localization

The scope, visibility and lifetime of an item must not be larger than needed. In particular, the index variable of a for loop must be declared in the for statement, and variables must be declared when they are first used (not in a cluster at the beginning of a method or function).

3.5. Naming

3.5.1. English

Names must be in English.

3.5.2. Informative names

The name of an item must strive to provide all relevant information about the function/purpose/use of the item, but without being overly long.

In general, an item with a larger scope needs a long name, an item with a small scope should have a short name. (Which is an additional argument for limiting the scope of an item.)

3.5.3. Separation

Distinct parts of a name must be separated. One separation style must be used. Some separation styles are:

1. camelCaseIsNamedAfterTheHumpsOfACamel
2. PascalCaseAlsoCapitalizesTheFirstLetter
3. Adding_Underscores_Makes_Reading_Easier
4. but_with_underscores_who_needs_the_captials

The last style (underscores between the words, no capitals) is recommended. (But use all UPPERCASE for macro's, if you must use a macro.)

3.5.4. Correct names

Names must reflect the kind of item:

1. The name of a predicate must contain a predicate-like work: is, has, contains, etc.
2. The name of an object must contain a noun.
3. The name of a function or method that primarily does something must contain a verb.
4. The name of a function that primarily computes something and returns it must be named after what it computes and returns.

3.5.5. Name in context

A name must always be understood in its context (enclosing class or namespace). Do not repeat this context in the name. (Note that this argues against the use of 'using'directives.)

3.5.6. Units

When any doubt is possible, the unit must be included in the name of an item, like temperature_in_celcius or speed_kmh.

3.5.7. Get and set

Names that contain (end in) get and set must be used (only) for methods that retrieve and change values that behave in a value-like manner, that is: getting always returns the last value that was set (directly or indirectly), and a sequence of set calls has the same effect as just the last set call. Use names that contain verbs like 'insert' or 'add' for methods that add data.

3.5.8. Plurals

Names that represent a collection of things must be plural.

3.5.9. File names

File names must be informative. C++ files must have the extension .cpp or .cc. C++ header files must have the extension .h or .hpp.

A header file must It must be possible to include a header files multiple times in a project, hence it must not contain code.

All C++ code files must be compiled separately: a code file must not be included by another code file.

3.6. Doxygen

The rules in this section apply if Doxygen documentation of certain parts of the code is required. Doxygen is meant to be used for documenting interfaces, don’t use it to document the internal working of a piece of code.

3.6.1. Short description

The short description must facilitate the reader in selecting a library element from a long list of elements. Hence it must convey the maximum information in the minimum of words. In general this means that the short description is not a full sentence.

3.6.2. Long description

The long description must consist of full sentences.

3.6.3. Completeness

The long description, combined with the (optional) parameter descriptions and the class description, must provide enough information for the user to use the element.

3.6.4. Compactness

Doxygen is used to document an interface, hence it must NOT be used to document the internal working of a piece of code. A consequence is that a header file must contain ONLY those items are intended to be used by the user of that header. (For technical reasons, this might not be possible for templates.)

3.7. OO specifics

3.7.1. Inheritance is substitutability

Inheritance must be used only when the derived class is substitutable for the superclass. In other words: the derived class can be used in all places where the superclass is required. (Liskov's substitution principle)

Note that the ”is-a”, ”kind-of” and ”is-a-subset-of” tests give too many false positives because our brain processes them in an loose, every day sense. They are correct, but only when applied in a very strict way.

Another way to formulate this rule is that a derived class must:

- promise no less, and
- require no more.

3.7.2. Either ADT or object or container

In general, a class must be either

1. a value class (Abstract Data Type), or
2. an identity class (describing a real object), or
3. a container (which holds objects without interfering with them).

Do not mix these type into one classes.

A value class has value semantics, an assignment operator and a copy constructor, and the appropriate comparison operators.

An identity class has no assignment operator, no comparison operators, and no copy constructor.

3.7.3. Rule of three

A class that has subordinate objects (= contains pointers to objects created by its code) must have (besides a constructor):

1. A copy constructor
2. An assignment operator
3. A destructor

These methods must take care that no memory problems (leaks, double deallocations) occur, and that object integrity is maintained (no aliasing).

Note that for identity classes it is often a good idea to block the copy constructor and assignment operator instead of providing an implementation.

3.7.4. No global objects

Global objects that require run-time initialization by calling their constructors must not be used. The problem is that the order of initialization is not defined over multiple files. Note that (mutable) global objects are a bad idea for other reasons too.

Global objects that are declared constexpr are allowed because their initialization is done at compile time. (consexpr is not implemented in MSVC 2013)

3.7.5. Use initialization lists

Attributes of an object must be initialized in the initializer list of the constructor, not in a statement in the body.

3.8. General quality

Rules in this section strive to improve the correctness of the code, both in its original form and after some (small) modifications.

3.8.1. C++ 0x11

Code must conform to the C++0x11 standard. Use the appropriate compiler switches to let the compiler check this (GCCC: -std=c++11).

3.8.2. No resource leaks

Code must not leak memory or other resources.

This applies especially to C++ classes that create subordinate objects: such classes must have a destructor that destructs such objects. (And such classes need to pay attention to the copy constructor and assignment operator.)

3.8.3. Const correctness

A parameter that is by design not changed must be marked as const. A method or function that does not change its context must be marked as constexpr (when possible) or const.

3.8.4. Multiple inclusion guards

Including a header file multiple times must not create a problem. In practice this means that a header file must contain the standard #ifdef/#define/#endif lines with a name that is derived from the file name.

3.8.5. No Implicit conversions

Implicit conversions must be disabled by declaring all one-parameter (except the copy constructor) constructors as explicit.

3.8.6. Zero

The value 0 must be used only for integer values. Use 0.0 for reals, nullptr for pointers, false for bool, and '\0' for chars.

3.8.7. Use enum class

The enum class must be used instead of the plain enum.

3.8.8. Use bool

The bool type (not int or char) must be used for truth values.

3.8.9. Don't return an invalid reference

A function or method must not return a reference (or pointer) to a location that is no longer valid after the return of the function, like a local variable.

3.8.10. Don't compare floats for equality

Floating point (or double) values are not exact, hence they must not be compared for equality (or inequality). Likewise, code must not rely on the difference between < and <= (or > and >=) for such values.

3.8.11. Use cin & cout

The C++ std::cout and << operators must be used for formatted output (instead of the C-style printf). Likewise, the >> operators must be used for formatted input.

3.8.12. No memcpy, memcmp, realloc, etc.

The C-library functions for bit/byte-wise copying (memcpy etc) and comparing must not be used. Instead, use the methods defined for the objects at hand.

3.8.13. No malloc/free

The C-library memory management functions (malloc, free, etc) must not be used. Use the C++ new and delete.

3.8.14. New and new[]

Objects allocated with new must be deleted (once!) by delete, objects allocated by new[] must be deleted (once!) by delete[].

For some implementations this is not critical, so hence the argument ’but it works’ fails (as it does almost every time).

3.8.15. No variable arguments lists

Function or method variable argumenst list must not be used. Template variable arguments lists and initializer_list<> are OK.

3.8.16. Use ++i

When either ++i or i++ can be used, ++i must be used because in general the implementation of ++i is simpler (shorter and faster) than of i++.

3.8.17. Use array<>

Avoid the use of C-style arrays, use the array<> template.

3.8.18. The for(:) variable must be &

Always declare the loop variable in a for(:) loop as reference. Add the const qualifier when you don’t intend to modify the elements that you loop over.

3.9. Readability

The rules in this section strive to improve the readability of the code. As a side effect, these rules tend to improve correctness, because it makes it easier for the author to read his code to detect bugs.

3.9.1. No warnings

Code must compile without warnings. Use the appropriate compiler settings to make sure your code complies with this rule (GCC: -Wall –Werror).

Most warnings indicate either an error, or something that a reader would find strange too. Most warnings can be avoided by a suitable extra piece of code, which satisfies both the compiler and the reader.

3.9.2. No surprises

Code must strive to minimize surprises for the reader. This rules against silly things like using operator+ to do multiplication.

3.9.3. Function size

A function or class must not be unduly large or complex. A function or method must certainly be limited to one screen, but preferably much smaller.

3.9.4. Avoid macro's

Macro's definitions must only be used when no alternatives exist.

Alternatives for constants are enum class, const, constexpr. For more complex cases templates should be considered.

When a macro is still necessary its name must be in UPPERCASE. In the macro body, enclose instances of a parameter in the body parenthesis to avoid unpleasant surprises.

3.9.5. No goto

Goto must not be used.

The main problem with a goto is reading the code after a label. When reading such a label, the reader must find all goto's that jump to that location, to find out in what states the application can be after the label.

3.9.6. Switch and break

Each alternative in a switch statement must end with a break: fall-through to the next alternative is not allowed.

3.9.7. ? operator

Use the ? operator only if it makes the code more readable (which in most cases means shorter). People often find ? expressions difficult to read, so put parenthesis around the condition (this helps a little bit). Format more complex ? expressions like an if statement.

```
maximum = ( x > y ) ? x  : y;

fac =
         ( n > 1 )
            ? fac( n - 1 )
            : 1;
```

3.9.8. Avoid pointers

Pointers must be used only to store an address which must be subject to change and/or pointer arithmetic. Use a reference when possible, or use an array (as parameter specification) when a parameter is intended to be an array.

3.9.9. No octal literals

Octal constants (constants that begin with a 0 digit) must not be used. Few people even know that a literal that starts with a 0 digit is in octal.

3.9.10. Char

Use the char type (only) to represent ASCII values. The char type can be either signed or unsigned (depending on the compiler), so the guaranteed range is 0..127. Hence char expressions, including intermediate values, must be limited to that range.

Char should not be used for small integers, use signed char or unsigned char instead (or better: one of the types from the header <cstdint>). Char should not be used for Boolean values: use bool instead.

3.9.11. Explicit conversions

Type conversions, even between floating point and integer types, must be made explicit.

3.9.12. Do not hide

Identifiers in an inner scope must not use the same name as an identifier in an outer scope and thus hide that (outer) identifier.

4. For the future

Aspects that might be added:

- Testability, test reports
- Be const correct
- Do not hide names
- Use RAI (Resource Acquisition is Initialization)
- * for pointer parameters, [] for arrays
- No using, especially not in headers
- Excessive number of parameters
- Avoid high McCabe
- Make methods static if possible
- Use override when possible
- Avoid globals (testability)
- Bundle parameters that belong together (like x,y)
- Use unsigned?
- Exception safety (except for RAII?)
- Multithreading safety
- Resource Standard Metrics
- Automate copy-paste detection
- Cyclo / n-path complexity
- Fan out
- IFSQ level 2 (joost heeft pdf) Bolton
- Throw by value, catch by reference:
- Emphasize ownership of resources
- Prefer compile/link errors over runtime errors
- Avoid cyclic dependencies
- Headers must be self-sufficient (not require another header first)
- Parameters (including return) by value, pointer (juk!) or reference
- If +, then also +=
- Avoid slicing
- Commit to abstractions, not to details
- Sutter's STL advices
- One Definition Rule: shared stuff must be in headers

Either final, or no virtuals, or virtual destructor

A class must either be declared final, or have no virtual methods, or have a virtual destructor.

The or is inclusive: more than one is OK, although both final and virtual methods is a strange combination.

The reason is that a derived class that allocates memory, and is passed around as (pointer or reference to) the base class must get a chance to free that memory when it is destructed in a situation where it is only known to be of the base class. Hence a base class that does not have a virtual destructor cannot safely be used to derive from, and there is no language mechanism that warns against such a situation.

5. Sources

1. Google C++ Style Guide 3.274, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml : usefull and surprisingly short
2. C++ Coding Standard 2008-03-01, http://www.possibility.com/Cpp/CppCodingStandard.html : seems to be written by one ex-Java programmer, overly detailed in some aspects, empty in others
3. High Integrity C++ coding standard version 4.0, http://www.codingstandard.com/ : good, but large and detailed
4. GCC C++ Coding Conventions 2012-08-16, http://gcc.gnu.org/wiki/CppConventions : rather detailed in most aspects, takes a lot for granted, geared towards GNU projects and low-level efficiency
5. C++ FAQs, Cline & Lomow, ISBN 0-201-58958-3 : not a coding standard, but it states many things that should or should not be done in C++ code. Pre-ANSI.
6. Geosoft C++ programming style guidelines Version 4.9, January 2011, http://geosoft.no/development/cppstyle.html : usefull, but a bit too detailed in some place (e.g. punctuation).
7. JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS revision C December 2005 : very detailed, reads almost like a book of law, but has some useful ideas.
8. C++ Coding Standards, Sutter & Alexandrescu, 2005 : high-level rules that are also detailed, good motivations, many are too advanced for our students (PIMPL, policy classes, user-defined new, etc)