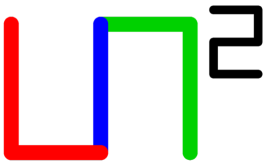

Programming style guide

how to write C++ programs

LIA
Universidade de Vigo
Escola Superior de Enxeñaría Informática
E–32004 Ourense

http://lia.ei.uvigo.es
mailto:formella@uvigo.es

Contact: Arno Formella

Reference: LIA-DOC-PRG-STYLE
Version: 1.3
Date: 15/03/2013
Pages: 39

Contents

1 Introduction

This document is about writing *nice* C++ programs.

It is not about software design in general nor about software engineering in C++ in particular, i.e., issues such as "how to design a class or class hierarchy" or "how to implement a certain algorithm, pattern, flow, or whatsoever" are beyond its scope. You should be already quite familiar with most of the features of C++. This document is not a C++ tutorial, nor an introduction to the programming language C++, neither a help on compiling C++ programs on some platform using some compiler.

Why should you read on? Because a common style really helps to achieve your programming goals, especially when you work in a multi–user, multi–platform environment.

Certainly, many of the recommendations gathered in this document are different from or even contrary to other style guides, especially the one you may already may use. (There is a short overview to other style guides in the reference section.) Most of the statements presented here are based on reasons, excluding explicitly the following one: *being compatible to legacy code and programming habits*. If you need such a compatibility in a specific case: document it and use the style you are required to use. But avoid sticking to traditions just because they are traditions, especially in new projects; rather draw reasonable decisions.

Writing programs, in any programming language, is not an easy task. Besides implementing all requirements in a correct, efficient and robust manner, there are more aspects to be addressed:

- The code should be reusable by yourself and others, either in an API–like way or directly through the copy&modify approach.
- The code should be portable to other environments including where people speak a different language.
- The code should be written in a way that reduces the probability of errors.

A common programming style among people working as a developing team on the same project, at least, helps to achieve such goals. Further positive aspects of a common programming style include:

- New people can start programming quickly.
- People new to C++ are spared the need to develop their own style.
- People new to C++ are spared making the same mistakes over and over again.
- The number of mistakes and misinterpretations is reduced in a consistent environment.
- All programmers adhering to more or less the same style can read any code easily and figure out what is going on without additional effort, especially when they are familiar with the underlying implicitly provided information.
- With consistently formatted documents other tools work much easier and their output is again much more readable for instance, linewise comparison with diff.

Clearly, there are some negative aspects of a common programming style that should be mentioned:

- You have to learn it.
- You might be obliged to use it.
- There are always exceptions and errors.
- Automatic code generation tools often don't produce code that adheres to any useful programming style.
- Your current editor or its default settings do not support directly the recommendations.

However, most of the arguments against a certain style are subjective and reflect personal opinions, such as:

- I have already my nice personal style and don't want to change.
- I don't like the proposed style.

If you go–on reading you should put yourself in the role of someone who never has written a medium or large size C++ program, but who is interested in writing such a program in a clear and commonly acceptable style.

Modern integrated development environments (IDEs) can improve the readability of code by access visibility, color coding, automatic formatting and so on. However, a programmer should never rely on such features only. Source code should always be considered independent from the IDE where it is or was developed. It should be written in a way that maximizes its readability and that allows to edit and modify the code on any other IDE.

This document distinguishes between style guide and programming discipline. The style guide just states how certain things should be written. The programming discipline gives some advice which constructs of the programming language should be used preferably in a certain context and how additional, implicit information can be passed to a programmer reading the code.

1.1 Purpose and scope

This style guide is intended as the base for all programming using the C++ programming language in the **LIA** research group. Although written specifically for C++, most of its content is easily ported to other programming languages such as C, Java, C#, Python or Perl.

All researchers of **LIA**, students realizing their final projects within the research group, and everyone else who likes to, can use this style guide as a base for writing C++ programs.

Comments on the document are very welcome.

1.2 Revision history

Version 1.3: A revised version with small changes and spelling corrections. Some issues of C++11 introduced.
Version 1.2: A revised version being more complete and consistent, written in the end of 2009 and beginning of 2010.
Version 1.1: Adapting the layout to the **LIA**-document style.
Version 1.0: This is the initial document written in July and August of 2009.

1.3 References

The document was developed with a long term programming experience in C++. The following references have been a great help:

- Todd Hoff's C++ coding standard located at `http://www.possibility.com/Cpp/CppCodingStandard.html` in its version from March 01, 2008.
- C++ Programming Style Guidelines. Version 4.7, October 2008. Geotechnical Software Services Copyright © 1996–2008. This document is available at `http://geosoft.no/development/cppstyle.html`.

- The C++ rules and recommendations by Mats Henricson and Erik Nyquist, Ellemtel Telecommunication Systems Laboratories, Box 1505, 125 25 Älvsjö, Sweden, Copyright © 1990–1992. http://www.doc.ic.ac.uk/lab/cplus/c++.rules
- Google C++ Style Guide, Revision 3.133 by Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai and Tashana Landray, 2009.

1.4 Writing conventions

The style guide follows the strategy of first stating some recommendations, then (maybe not in all cases) giving some positive examples, and afterward providing some arguments why the recommendations are given that way. Possibly, additional notes provide even more information about the issue treated in the section.

Recommendations

- The recommendations summarize the main aspects.
- A recommendation with the terms "*must*" or "*must not*" states a requirement that *must* be followed.
- A recommendation with the terms "*should*" or "*should not*" states a strong recommendation that *should* be followed, but that can be moderated or interpreted according to some specific needs or circumstances.
- A recommendation with the terms "*can*" or "*cannot*" states a general guideline that can be followed additionally, if you like.
- The recommendations are roughly ordered from "*must*"– over "*should*"– to "*can*"–recommendations.
- You *must* provide either a new or an alternative reason before you change a recommendation.

```
// The code sections are given as positive examples, only.
// To highlight the examples, they are surrounded by a rounded box.
```

Reasons

- The reasons summarize some insight why a specific recommendation has been stated.
- The main arguments for a common style can be summarized with: simple, uniform, tool adequate, editor adequate, human readable, pretty printable, easy documentable.

Notes

- If there are notes, they usually describe in some detail specific cases, exceptions, or give other worthwhile comments.
- A note may recall a specific issue of the language standard a novice programmer might not be aware of.

2 Style guide

As already stated in the introduction, the style guide concentrates on *how* different elements of the programming language should be *written* in a program file.

Recommendations

- You *must not* violate the recommendations as long as you do not have any reason to do so.
- You *should* state a reason whenever you violate the recommendations.
- You *can* change the recommendation whenever you have a clear reason to do so.

2.1 Comments

A comment should be useful, at least for two people: the writer and the reader.

As you know, a comment can appear at all places in a C++ program where a whitespace is accepted as well, but where should you place it?

Recommendations

- A comment *must* precede the item being commented, with only one exception: a comment continuing in the very same line as the item being commented (so called in–line comments).
- Comments on the same line *must not* extend over more than that line.
- You *must* leave one blank after the `//`–comment indicator or the comment introduction of the documentation tool, e.g., after the `///`–comment.
- You *must* use the additional features of your documentation tool (e.g., Doxygen).
- You *must not* use a `/*..*/`–comment which separates the code within a line.
- You *should* place the comments as close as possible to the item being commented.
- You *should* give preference to the `//`–comment on each line rather than to the `/*..*/`–comment.
- You *should* use correct sentences including correct punctuation in your comments.
- Abbreviated comments *should* be used only in in–line comments.
- You *should not* use relative references in your comment, such as “the following class” or “the next group of methods”.

2.1.1 Comments for code and classes

Recommendations

- You *must* indent the comment either at the same level or one level deeper than the subsequent block structure of the item being commented, whatever you find more readable in the corresponding situation.
- You *must not* use a blank line between the comment and the block being commented, rather use a blank line before the comment, if necessary.

```
/// class description
class SomeClass {
  ...
    /// This constructor uses a somewhat rare argument that ...
  SomeClass(const Rare& argument);
};

// Explain the purpose of the while loop.
while(SomeCondition()) {
  // Describe the loop invariant.
  some code;
  ...
}
```

2.1.2 Comments for variable and constant declarations or definitions

Recommendations

- You *must* indent the comment either at the same level or one level deeper than the subsequent block structure of the item being commented, whatever you find more readable in the corresponding situation; or continue with the comment on the same line.
- You *must* use a preceding comment whenever a comment following a declaration or definition does not fit into the line.

```
unsigned int start(size/2); // start at the center of the array

// The magic_constant is merely used for irrational purposes.
const unsigned int magic_constant(42);
```

2.1.3 Comments for parameters

Recommendations

- You *must* indent the comment either at the same level or one level deeper than the subsequent block structure of the item being commented, whatever you find more readable in the corresponding situation; or continue with the comment on the same line.
- You *must* use a comment with forward reference, e.g., `///<`, whenever you decide to place the comment after the parameter.
- You *should not* continue a comment following a declaration or definition in the subsequent line, rather use a preceding comment.

```
void DoSomeThing(
  const int iteration,  ///< current index of iteration
  SomeThing& some_thing ///< the modified thing
) {
  ...
}
```

2.1.4 Special purpose comments (gotchas)

The word “gotcha” comes from the relaxed pronunciation of “I got you” or “I’ve got you” usually referring to an unexpected capture or discovery.

In programming, a “gotcha” is a feature of a system, a program, or a programming language that works in the way it is documented but is counter–intuitive and almost invites mistakes, because it is both easy to invoke and unexpected and/or unreasonable in its outcome.

Recommendations

- You *must* place the gotcha keyword as the first symbol in the comment.
- You *must* write the gotcha keyword all uppercase and with surrounding colons.
- You *should* use marks or commands of the documentation tool to document the information (e.g. with Doxygen use `todo` for to–do information).
- The first line *should* be a self–containing, meaningful summary.
- The author and the date of the remark *should* be part of the comment.
- More information *can* be added in subsequent comment lines.

Here are examples of some commonly used “gotchas”:

Example

- `:BUG: [bugID] describe`
  Means that there is a known bug here, optionally give a bug ID, explain the bug and possibly give a workaround.

- `:COMPILER: state`
  Sometimes you need to work around a compiler problem. Document it here. The problem may go away eventually.
- `:KLUDGE: explain`
  When you have done something ugly, say so, and explain how you would do it differently next time if you had more time.
- `:TODO: list`
  Means that there is more to do here, don't forget.
- `:TRICKY: explain`
  Tells somebody that the following code is very tricky, so don't go changing it without thinking.
- `:WARNING: warn`
  Beware of something.

Reasons

- Often, gotchas stick around longer than they should. So make sure that even in the far future another programmer understands what is said.
- Embedding author and date information allows other programmers to draw the necessary decisions.
- Embedding author information indicates who to ask in case of doubts.

Note that documentation tools usually provide methods to include these special comments into the automatically generated documentation. As an example Doxygenprovides the `bug`, `note`, `todo`, and `warning` comments.

2.2 Names

One good name tells you more than a thousand comments.

Names are, besides the underlying algorithms and design, the heart of programming. A name should be the result of a sufficiently long thought process about the context the name will live in. A programmer who tries to build a system that is understandable as a whole would create a name that "fits" with the system or even beyond the system. If the name is appropriate, everything fits together naturally, relationships become clear, meaning becomes derivable, and reasoning from common human expectations works as expected.

2.2.1 Type related names

2.2.1.1 Namespaces

Recommendations

- You *must* use all lowercase letters for a namespace name.
- You *must not* use the underscore as word separator.
- You *must* use a namespace alias if you need to work with a foreign library that does not comply with the above recommendations.
- You *should* use short namespace identifiers.
- You *can* use short abbreviations for namespaces.

```
namespace lia {
 ...
}

lia::Vector ...
```

```
namespace some_WEIRED_company_NAMEspace {
  ...
}

namespace swc = some_WEIRED_company_NAMEspace;
```

2.2.1.2 Classes

Name a class after what it *is*. If you can't think of a name for the objects you are about to create, you probably have not thought through the design well enough.

Recommendations

- You *must* use an uppercase letter for the first character.
- You *must* use uppercase letters as word separators, lowercase letters for the rest.
- You *must not* use underscores.
- You *should* use nouns (or verbs or adjectives used as nouns) as class names.
- You *should not* use compound names of more than three words.
- You *should not* bring the name of the class a class derives from into the derived class's name. But:
- You *can* use suffixes which are helpful, e.g., if your system uses agents (class `Agent`) then naming something `DownloadAgent` conveys real information.

```
class Vector {
  ...
};

class ParticleStore {
  ...
};
```

Notes

- A class should stand on its own. If you feel you need to use the same name twice, it is time to think of introducing namespaces.
- Prefixes are sometimes helpful, e.g., if you want to make clear that certain classes belong to the same group of objects, but probably namespaces are the better option.

2.2.1.3 Templates

Name a template after what it *stands for*. Usually a template is an abstract entity, such as a container, collection, action, or concept. If you can't think of a name for the template you are about to implement, you probably have not thought through the design well enough.

Recommendations

- Template names *must* follow the class naming conventions.
- Template names *must* terminate in `T`.
- You *should* avoid an uppercase letter in front of the terminating `T`.
- Template parameters *should* be single uppercase letters.
- You *should* use `typename` instead of `class` in the template parameter section.

```
template <typename T>
BoxT {
  T t;
  ...
}
```

2.2.1.4 Types

Name a type after what it *is*. Often, a type is an abbreviation for a complex class of template instantiation. If you can't think of a name for the type you are about to define, you probably have not thought through the design well enough.

Recommendations

- `typedef`s *should* follow the class naming conventions.
- If the `typedef` defines sized data, the number of bits or bytes *should* be given in the type name.

```
typedef unsigned int Uint32;
typedef unsigned char Block512[512];

BoxT<double> box;

typedef BoxT<double> BoxD;
```

2.2.2 Class methods and functions

Name a method or function after what it *does*. Usually methods and functions perform actions. So the name should make clear what that action does. If you can't think of a name for the functions you are about to implement, you probably have not thought through the design well enough.

Recommendations

- You *must* use an uppercase letter for the first character.
- You *must* use uppercase letters as word separators, lowercase letters for the rest.
- You *should* use a verb as first word of the name.
- You *should* use the same name (i.e., use overloading) whenever the methods indeed realize sufficiently similar tasks.
- You *should* use common standards. Document to which standard you adhere, if a class is designed in the field of this standard.
- You *should* try to use readable names.
- You *should not* use compound names of more than three words.
- You *should not* use abbreviations, spell things out, especially you should not abbreviate words by simply omitting its tail.

```
bool CheckForErrors(void);
void DumpDataToFile(const Data& data);
```

Recommendations

- With the first recommendation and the first one for class attributes (see next section), an access method to a class attribute is just distinguished from the attribute by the uppercase letter, which underlines the strong relationship between the attribute and the method.

2.2.3 Class attributes and variables

Recommendations

- You *must* use a lowercase letter for the first character.
- You *must not* use an underscore for the first character.
- You *must* use uppercase letters as word separator for class attributes.
- You *must not* use additional prefixes for class attributes.
- You *must* use the underscore as word separators for local variables.

- You *must* use math standards, i.e., write angles with words for Greek letters, write coefficients with letters from the beginning of the alphabet, write variables with letters from the end of the alphabet.
- You *should* use single letter variable names for standard type variables in a consistent way, possibly with numerical suffix (e.g., `x0`,`x1`), some examples:
  - `i,j,k`: integer loop variables
  - `n,m`: integer limit variables
  - `a,b,c`: floating point coefficients
  - `p,q`: points
  - `r,s,t`: parameter values
  - `v,w`: values
  - `x,y,z`: coordinates
- You *should* use the general rule: the longer the name the larger the lifetime.
- You *should* use for deeper nested variables letters from later in alphabet. There is one exception: if you use a variable for a dimension, e.g., `i` for rows and `j` for columns, you *should* rather stick to this naming convention, rather than to the nesting rule.
- You *should* use commonly used addressing modes, e.g., `i` for rows and `j` for columns.
- You *should not* bring the type of the variable into the variable's name, but
  - Generic variables *should* have the same name as their type.
  - Non–generic variables *should* have the name of their type added as suffix or as a prefix (the former especially when the variable represents a GUI component). Such variables usually play a role, its semantics is easier captured if the type is seen at the same time.
- You *should* use the plural form on names representing a collection of objects.
- You *should* use the prefix `n` for variables representing a number of objects.
- You *should* use the prefix `i`, respectively `j` and `k`, for variables representing elements in a loop (named iterators), e.g., `it`, `jt`, and `kt` for iterators in nested loops.
- You *should* use a suffix to express a certain variant, e.g., the unit used for the value represented by this variable.
- You *should not* use prefixes to indicate pointer or reference type of a variable, unless you need to point out something.
- You *should not* use compound names of more than three words.
- You *should not* use abbreviations, spell things out.

```
Vector vector;
ParticleStore particleStore;
double phi;

void HighlightAll(Topic* topic)
void Open(Database& database)

Database backupDatabase;

Point startingPoint;
Point center_point;

Dialog preferenceDialog;
Scrollbar buttomScrollbar;

std::vector<Point> points;
unsigned int nPoints;

for(vector<Point>::iterator it(list.begin());
  it!=list.end();
  ++it
) {
  Point iPoint(*it);
  ...
}
```

```
Segment* segment;

Segment segment;
// if you use at the same time segment and p_segment.
Segment* p_segment;
```

Reasons

- Repeating the same name reduces complexity by reducing the number of terms and names used.
- The type can be deduced often by the name of the variable.
- Using the commonly used ways of how things are named increases the readability and understandability of the code.
- You can always prepend the classname as classifier if you need to make a distinction.

2.2.3.1 Global variables

Recommendations

- Global variables *must* follow the naming convention for member variables of classes.
- Global variables *must* always be referred to using the `::`–operator.
- Global variables *should not* have a specific prefix.

Reasons

- Variables may change from global to local/class–local, hence a prefix might become obsolete.
- You can think of global variables as static variables of your “application class”.

2.2.3.2 Local variables

Recommendations

- You *should* write local variables with all lowercase letters and the underscore as word separator.

```
int NameOneTwo::HandleError(
  const int errorNumber
) {
  const int error(OsErr());
  const Time time_of_error;
  ErrorProcessor error_processor;
  Time* p_out_of_time=0;
}
```

2.2.3.3 Argument names

Recommendations

- You *must* write a variable name in the declaration of the formal parameters of a function or method.
- You *must* write `void` in the parameter list, whenever a function or method does not have parameters, with the only exceptions of the default constructor and destructor.
- Generic variables *should* have the same name as their type.
- You *can* use for the parameters of a constructor the same names as the member variables appending an underscore in the definition (but not in the declaration).

```
class SomeClass {
  unsigned int maxCount;
  ...
  SomeClass() : maxCount(0) { }
  ~SomeClass() { }
  SomeClass(
    const unsigned int maxCount_
  ) :
    maxCount(maxCount_)
  { }
  unsigned int MaxCount(void) const { return maxCount; }
  ...
  void LinkDatabase(Database* database);
};
```

2.2.3.4 Pointer and reference variables

Recommendations

- You *must* write the pointer symbol and reference symbol next to the type, not next to the variable.
- Pointer and reference variables *can* have a type indicating prefix, e.g., `p` for pointer and `r` for reference.

```
Particle* pP=0;
Particle& rQ(someParticle);
```

2.2.3.5 Static variables

Recommendations

- Static variables *should not* have an indicating prefix.

Reasons

- There might be a shift during development from static to non–static and vice versa, hence, maintaining consistent prefixing becomes difficult.
- The distinction should come from the name, i.e., it should be clear that the variable is a class component and not an instance component.

2.2.4 Enumerations

Recommendations

- You *must* follow the class name conventions for the name of the enumeration.
- You *must* use all uppercase letters with the underscore as word separator for the enumeration members.
- You *should* consider namespaces to group enumeration values that are used as globally defined constants.
- You *should not* use a plural noun as name for an enumeration.
- You *can* use common prefixes to differentiate the names in an enumeration type.

```
enum Color {
  COLOR_RED,
  COLOR_GREEN,
  COLOR_BLUE
};
```

2.2.5 Constants

Recommendations

- You *must* use all uppercase letter for constants.
- You *must* use the underscore as word separator.
- You *must* write floating point constants always with decimal point and at least one decimal, or with exponent.
- You *must* write floating point constants always with a digit before the decimal point.
- You *should not* use compound names of more than three words.

```
const int SOME_CONSTANT;
const unsigned int MAX_ITERATIONS(100);

double total(0.0);
const double SPEED_OF_LIGHT(3e8);
```

2.2.6 Labels

Recommendations

- Besides labels placed within `switch`–statements, all labels *must* have the prefix `Label`.
- The colon *must* follow immediately after the label.
- The label *must* be placed starting at the very left end of a line, independently whether the label is nested within some block structure.
- Labels *should* indicate in their names what is the reason for being there.

```
...
LabelError:
...
```

Reasons

- Labels should be very easy to find in the code, because they usually are reached through a abrupt change in program flow.

2.2.7 File guards

A file guard in a header file is a mechanism that prevents multiple inclusions of the same file.

Recommendations

- You *must* use the exact filename as part of the header file guard, where the dot is replaced by an underscore.
- You *must* include a comment explaining why you do not use a file guard in a header file, i.e., the specific file might be included more than once.
- You *can* prepend a significant prefix indicating the company, project, or namespace.

```
#ifndef Vector_h
#define Vector_h
...
#endif

#ifndef mathlib_Vector_h
#define mathlib_Vector_h
...
#endif
```

2.2.8 Macro definitions

The use of preprocessor macros can introduce unwanted side effects, so special care must be taken of.

Recommendations

- You *must* use all uppercase letters for macro definitions.
- You *must* use the underscore as word separator.
- You *should* undefine a macro definition when you do not need it any more.
- You *should* avoid macro definition, there are better possibilities available in C++.
- You *should* consider the `do-while` construct for the macro definition.

```
#define FOO(what_you_want) do { what_you_want } while(0)
```

Reasons

- You can use a complex expression and sequences of statements as argument of the macro.
- You don't run into problems with empty statements.
- You don't run into problems with a semicolon placed after the usage of the macro, for instance as in `FOO(WriteSomething);`

2.2.9 C functions

Recommendations

- You *must* precede a declaration of a C–function with `extern "C"` or gather them in a block with `extern "C"{ }`.
- If the declaration is used both in a C and in a C++ file, you *must* guard the declaration with an appropriate preprocessor directive.
- You *should* follow for naming C–functions the same recommendations as for C++ functions and methods.

```
extern "C" void ALonelyCFunction(void);

#if defined(__cplusplus}
extern "C" {
#endif

void SomeCFunction(void);
int AnotherOne(const int n);

#if defined(__cplusplus}
}
#endif
```

Notes

- Note that the `__cplusplus` macro is defined by all reasonable C++ compilers.

2.2.10 Abbreviations

Recommendations

- You *must not* use all uppercase abbreviations, instead you should use an initial uppercase letter followed by all lowercase letters or all lowercase letters.
- You *should* avoid abbreviations, use only really commonly established abbreviations.

```
class XmlReader {
  ...
};

int xml_header_offset;
```

Reasons

- If you don't write uppercase abbreviations, the building of the names follows the general rule for building names.
- If two or more abbreviations are connected, it is difficult to separate them in their individual parts, as in `STDHTMLInterface`.
- Different programmers might abbreviate differently which adds complexity to the project.

2.2.11 Files

Recommendations

- You *must* name header files with the extension `.h`.
- You *must* name source files with the extension `.cpp`.
- You *must* name the files implementing a class or template exactly as the class name or the template name.
- You *should* implement at most one class in one file.
- You *should* name implementation files for inline functions and templates with the extension `.hpp`.

2.3 Formatting

The formatting style is the programmer's corporate identity.

Recommendations

- Code lines *must not* exceed 80 characters.
- You *must* split lines larger the 80 characters in readable parts.
- You *must* use indentation, blank lines, and white spaces to enhance the readability of the program.

2.3.1 Block structure and white space

2.3.1.1 Placement of braces, parenthesis, and the like

Recommendations

- You *must* place an opening brace in the same line as its dominating control structure.
- You *must* place a closing brace either as a single symbol on a line, possibly followed by a semicolon or a comment, or in the same line as its corresponding open brace.
- You *should* place one blank between a pair of braces to indicate an empty block.
- You *should* place all non–braces parenthesis pairs such as `[]`, `()`, `<>` either on the same line or you *should* follow the recommendations for braces.
- You *should* use blanks sparsely close to parenthesis of any kind.

```
class XmlReader {
  ...
  XmlReader(void) { }
};

switch(state) {
  case INITIAL: {
    ...
    break;
  }
  case
    ...
  default:
    throw Exception(...);
} // end of state machine switch
```

Reasons

- The more important brace is the closing brace, because it indicates where a block ends. The beginning of a block can be guessed in almost all case by analyzing the identifier/keyword in front of the brace.
- Placing opening braces alone in their own lines just adds almost totally blank lines which disturb the block structure, because it is difficult to distinguish between blank lines separating blocks and the line containing just the brace.
- Braces, parenthesis and the like serve two purposes: they group certain entities as one element (e.g. a parameter list, a block of instruction, a first–to–evaluate–expression, etc.) and they allow for a nested structure of blocks (e.g., nested loops, nested classes, nested control structures etc.).
- The purpose of such parenthesis always should be the separation of entities without destroying the perception of the groups.

2.3.1.2 Indentation

Recommendations

- You *must not* use tabulators to indent.
- You *must not* indent a namespace content, use blank lines instead to highlight the start and end of a namespace.
- You *must not* indent a preprocessor directive, they always start at the first column.
- You *must not* indent `public`, `protected` and `private` keywords further than the `class` keyword.
- You *must not* indent labels.
- You *should* indent with two (2) spaces (increasing on all levels).
- You *should not* try to maintain vertically aligned blocks.
- You *can* indent with three or four spaces.

Reasons

- Huge indentation levels leave simply no room to place interesting information in that line.
- Wrapping of lines in the visualization tool, i.e., continuing the line at the beginning of the next line, just renders the indentation intent ridiculous.
- Even though with big monitors we can stretch windows wide, printers or printing software usually only print roughly 80 characters wide.
- The wider the window the fewer windows we can have on a screen. More windows is better than wider windows.
- diff output is viewed and printed correctly on all terminals and printers.
- It is difficult and a lot of typing work to maintain vertically arranged blocks when changes are necessary or things get copy&pasted.
- Indenting by itself does not enhance the logical structure of the code, it just adds white space on the left.
- Helper tools, like diff, or version control systems, like git, are line-oriented. Hence, with shorter lines clearly dedicated to one purpose, tracking of the on-going development is much easier and difference files are much smaller.
- Automatic documentation tools, like Doxygen, might get confused with the indentation level if the tab–size does not coincide or whenever indentation with blanks and tabs are mixed.

2.3.1.3 Blank spaces

Recommendations

- You *must not* use trailing whitespaces.
- You *must not* use tabulators as whitespaces.
- You *must not* use pagebreaks as whitespaces.

- You *must not* use blanks between a function name and the opening parenthesis.
- You *must not* use blanks between a keyword of a control structure and the opening parenthesis of its condition.
- You *should* use whitespace sparsely, its main purpose is to group in readable units (words).
- You *can* put blanks after the opening parenthesis and before the closing parenthesis of a condition.
- You *can* put blanks around operators.
- You *can* put blanks after a coma.

Reasons

- Too much separation in a text between letters, words, sentences, paragraphs etc. does not enhance but disturbs a fluent reading.
- Modern editors and print software highlight keywords anyway, so separation is not needed.
- Blanks and blank lines disturb the visual block structure.
- Tabulators and pagebreaks cause problems for editors, printers, terminal emulators and/or debuggers when used in a multi–programmer, multi–platform environment.

Notes

- Note that the C/C++ standard allows the following six digraph symbols `<:`, `:>`, `<%`, `%>`, `%:`, `%:%:` which behave exactly like the six tokens `[`, `]`, {, }, `#`, `##`, hence, you have to place in certain expressions a blank (or another separator).

```
a=b% ::global;  // here the blank is needed, avoids %: digraph
if(a< ::global) // here the blank is needed, avoids <: digraph
```

2.3.1.4 Blank lines

Recommendations

- A blank line *must not* contain any character (neither blank spaces, nor comment symbols).
- You *should* add a blank line to separate logical parts of the code.
- You *should* use a blank line both before and after a namespace opening and closing.
- You *can* use up to three blank lines to separate definitions of functions/methods.

2.3.2 Layout of control structures

Recommendations

- You *should* place the opening brace in the same line as the corresponding keyword, and the closing brace right below, i.e., at the same indentation as the keyword.
- You *should* use the same convention for the parenthesis of long conditions.
- You *can* use end–markers for closing braces, if the corresponding opening brace is difficult to find.

Reasons

- You spare a line, hence you see more code at once.
- There is a clear difference what a function, class, or indented block is. For long conditions you would end up with two almost blank lines.

2.3.2.1 if–then–else

Recommendations

- You *must* avoid assignment statements in conditions.
- You *must* use either braces in both the `if` and the `else` part or in none of them.
- You *must not* use a blank after the keyword, rather use blanks in front and after the condition.

- You *should* avoid evaluation statements in conditionals.
- You *should* put the condition on a separate line, rather than breaking it in two parts.

```
if(condition) {
  statements;
}

if(condition) {
  statements;
}
else {
  statements;
}

if(
  condition1 &&
  condition2
) {
  statements;
}

if( condition ) {
  statements;
}
else if( condition ) {
  statements;
}
else {
  statements;
}

const File* fileHandle=open(fileName,"w");
if(!fileHandle) {
  ...
}
```

Reasons

- For debugging purposes, when writing on a single line, it is not apparent whether the test is really true or not. Remember that setting breakpoints and tracing in most debuggers is still line oriented.
- The parenthesis around a condition are—speaking somewhat imprecisely—redundant, you should make clear what parenthesis are used to group the logical parts of a condition and what parenthesis are just part of the syntax of the C++ language.
- Conditionals with executable statements are just very difficult to read.

2.3.2.2 while and do–while

Recommendations

- You *must* follow the sames rules as given for the `if`.
- You *must* write the `while` in a `do`–loop in the same line as the closing brace of the block.

```
while(condition) {
  statements;
}

while(
  condition1 &&
  condition2
```

```
) {
  statements;
}

do {
  statements;
} while(condition);

do {
  statements;
} while(
  condition1 &&
  condition2
);
```

2.3.2.3 for

Recommendations

- You *must* write the loop controlling code either in one line or at least in three lines, one for each part.

```
for(initialization; condition; update) {
  statements;
}

for(
  initialization;
  condition;
  update
) {
  statements;
}
```

2.3.2.4 switch

Recommendations

- You *must* mark a fall through, which is done on purpose, with an explanatory comment.
- You *must* mark a missing `default`–case, which is done on purpose, with an explanatory comment.

```
switch(selection) {
  case ABC:
    statements;
    // fall through is on purpose
  case DEF: {
    declarations;
    statements;
    break;
  }
  case XYZ:
    statements;
    break;
  default:
    statements;
    break;
}
```

2.3.2.5 Try–catch

The try–catch–block follows the spirit already given for the loop structures.

```
try {
  statements;
}
catch(const Exception& exception) {
  statements;
}
```

2.3.2.6 Conditional expression

Recommendations

- You *must* write the expression either in one line or use at least three lines, one for each part.

```
(condition) ? funct1() : func2();

   or

(condition)
  ? long statement
  : another long statement;
```

2.3.3 Block layout

2.3.3.1 Header file layout

Recommendations

- You *must* use an include protector or comment why you don't use one.
- You *must* group include statements.
- You *must* sort include statements alphabetically within a group.
- You *must not* use absolute file names in include statements.
- You *must* include all files at the top of a file.
- You *should* order the include files putting first the local includes, then the project/library includes, that the system includes.
- You *should* use as general order of the parts in a typical header file: general comments, include files, forward declarations, local defines, classes, inline operations (possibly as include of an external file), external references.

2.3.3.2 Class layout

Recommendations

- The general layout of a class *should* follow the scheme:
  friends: first all friend classes then all friend methods; recall that friend methods are implicitly `public`.
  statics: first all static attributes then all static methods.
  classes: all nested classes.
  members: all member attributes.
  methods: all methods organized as follows:

lifecycle: construction and destruction
operators: overloaded operators
access: access methods and operators
inquiry: methods that query the state of the object
operations: methods that operate on/with the object

- You *must* order the entries in a reasonable part of a class: first `private`, then `protected`, then `public`.
- You *must* use the `virtual` keyword whenever you overwrite a virtual method.
- You *should* group the entries in each part with common sense from a programmer's point of view.
- Within each group, you *should* sort the entries alphabetically.
- You *should* be aware of data alignment issues on the target architecture.

Reasons

- The order in which items appear in a file is relevant for a programmer which works directly with the given code.
- You end up scrolling up and down if you want to learn in detail what the class does, because public interfaces usually use private parts.
- The automatic documentation tools rearrange everything anyway, so a user of the class interested only in the public interface has a standard way of looking at the class, which does not depend much of the real order.
- If things are sorted, they are easier to find.
- Memory aligned data might be faster to access on a given processor.

```
class SomeClass : public BaseClass {
  ...
};

class SomeClass :
  public AClass,
  public BClass,
  public CClass
{
  ...
};
```

2.3.3.3 Source file layout

Recommendations

- You *must* include all files at the top of a file.
- You *should* group the include files reasonably.
- Within a group, you *should* sort the include files according to their names.
- You *should* include first the specific local files, then other library files, and finally system header files.
- You *should* maintain the same order while implementing the methods as you used in the header file.

2.3.3.4 Method, variable, and parameter layout

Recommendations

- You *must* declare/define each variable on one line.
- You *must* write the reference and pointer symbols next to the type rather than next to the name.

- You *must* use `const` where ever possible.
- You *should* write the type and the name of the variable or method/function on the same line, both in declarations and in definitions.
- You *should not* pass simple types by reference, if possible.
- You *should* prepend a `/* static */`–comment right before the definition of a static method.

```
double* d;
int& j(i);

/* static */
void MyClass::StaticMethod(
) {
}

int MyClass::AnyMethod(
  const int arg1,
  const int arg2,
  const int arg3,
  const int arg4
) const {
}

int MyClass::AnyMethod(
  const double arg1,
  const double arg2
);

int MyClass::AnyMethod(
  const SomeType& in,
  OtherType& out
);
```

Reasons

- Editing is still line oriented. Moving entire lines is much easier and less error prone than copying words.
- With common file comparing tools (such as diff) it is easy to find out when variables or parameters differ.
- You avoid pointer declaration errors (`int* i, j`, here, only `i` is a pointer, `j` is not).
- The pointer–ness or reference–ness of a variable is a property of the type rather than the name.
- It is easy to add specific comments to each entity.
- The return type of a method or function is related to the function in the same way as it is the case for a simple variable.
- Human reading is line oriented, so things being in the same line are usually perceived as closer related than things on separate lines.

2.3.4 Splitting lines

Recommendations

- Lines *must not* exceed 80 characters.
- You *must* split lines in readable and logically coherent parts.
- You *must* indent and align the parts to enhance readability.

```
RunMethod(par1,par2,par3,par4);
RunMethod(
  par1,par2,par3,par4
);
```

```
RunMethod(
  par1,par2,
  par3,par4
);
RunMethod(
  par1,
  par2,
  par3,
  par4
);

if(
  conditionOne &&
  conditionTwo
) {
  statements;
}
else if(
  conditionThree ||
  conditionFour
) {
  statements;
}
else {
  statements;
}
```

Reasons

- Writing the parameters each one at its own line, allows to add comments for each parameter easily.
- Writing the conditions each one on its own line, allows to add comments for each condition easily.
- Helper tools, like diff, or version control systems, like git, are line-oriented. Hence, with shorter lines clearly dedicated to one purpose, tracking of the on-going development is much easier and difference files are much smaller.
- External references to specific lines of the code are easier to use, because there is only one significant entity on that line.

3 Programming discipline guide

As already stated in the introduction, the programming discipline concentrates on *how* different elements of the programming language should be *used* (or should not be used).

Recommendations
- You *should* write your programs completely in English.
- You *should* be aware that a good programming discipline is even more important than a nice programming style.

Reasons
- English is the preferred language for international development.
- Almost all libraries are written in English.

3.1 Comments

Comment the necessary, not the obvious.

Recommendations
- You *must* use automatic documentation tools, such as Doxygen.
- You *should* try to write your program that the code is self-explaining, especially by using names that through their context based semantics tell the right story.
- You *should* comment the borderline and exceptional cases.
- You *should* comment the invariants of the algorithm.
- You *should* comment any grouping and general assumptions.
- You *should* comment preconditions and postconditions.
- You *should* comment side effects.
- You *should* comment why you don't initialize a variable.
- You *should* comment early, best when you write the code. No one goes back and documents old code.
- You *should* take into account that comments in header files are for users of the class and comments in source files are for implementers of the class or of derived classes.
- You *should* use additional blocks to highlight comments for such a block (and possibly confine the lifetime of the variables).
- You *should* comment type conversions and give arguments why you don't expect loss of precision or why you don't care of such a loss.
- You *should* make sure that comments are easily added/modified/deleted whenever the entity is added/modified/deleted.
- You *should* stick to the notations of the source documents for your coding effort (even if this up to a certain degree vulnerates the style conventions given in this document).
- You *should* avoid using code to explain code.
- You *should* comment tricky implementations especially explaining non–obvious decisions being drawn for performance reasons.
- You *should* give references to additional information and avoid copying information that appears better in its original format. Note that documentation tools allow to include elements from complex text processors.
- You *should* give a comment on copyright issues and on code implicitly or explicitly used in the file with fulfilling their own copyright requirements.

3.2 Const correctness

The compiler can do more than you might think.

Recommendations
- You *must* use `const` where ever you can.

- You *must* declare variables that are not changed after initialization as `const`.
- You *must* declare methods that do not change the object as `const`.
- You *must* declare parameters that are not changed, i.e., input parameters, either as `const`–parameters (simple type) or as `const`–references.
- You *must not* use magic constants literally, rather you *should* define useful names either as constants or as enumerations of constants.

Reasons

- The compiler can statically check that the constant variable is not changed, whenever the explicit possibilities are not employed.
- A reader of the code can rely that the state of the object represented by the variable is not changed inbetween two usages.

3.3 Ordering

Think of searching a name in an unordered phone book...

Recommendations

- Whenever there is no apparent reason against it, you *must* order the entries alphabetically, i.e., once the parts of your code have been grouped (possibly in nested groups) within each basic group all entries *must* be sorted in ascending order.
- The ordering of `operator`–implementation *must* follow descending order in priority level, i.e., high priority operators *must* be declared or defined before low priority operators.

operator	description	associativity	over-loadable
`::`	scope resolution	left-to-right	no
`::*`	pointer to member		no
`++ --`	postfix increment and decrement		yes
`()`	function call		yes
`[]`	array subscripting		yes
`.`	element selection by reference		no
`->`	element selection through pointer		yes
`typeid()`	run–time type information		no
`const_cast`	type cast		no
`dynamic_cast`	type cast		no
`reinterpret_cast`	type cast		no
`static_cast`	type cast		no
`++ --`	prefix increment and decrement		yes
`+ -`	unary plus and minus		yes
`! ~`	logical NOT and bitwise NOT		yes
`(type)`	type cast		yes
`*`	indirection (dereference)		yes
`&`	address–of (reference)		yes
`sizeof`	size-of		no
`new new[]`	dynamic memory allocation		yes
`delete delete[]`	dynamic memory deallocation	right-to-left	yes
`.*`	pointer to member by reference		no
`->*`	pointer to member by pointer	left-to-right	yes
`* / %`	multiplication, division, and remainder		yes
`+ -`	addition and subtraction		yes
`<< >>`	bitwise left–shift and right–shift		yes
`< <=`	less–than and less–than–or–equal–to		yes
`> >=`	greater–than and greater–than–or–equal–to		yes
`== !=`	equal–to and not–equal–to		yes

`&`	bitwise AND		yes
`^`	bitwise XOR (exclusive or)		yes
`|`	bitwise OR (inclusive or)		yes
`&&`	logical AND		yes
`||`	logical OR		yes
`c?t:f`	ternary conditional	right-to-left	no
`=`	direct assignment		yes
`+= -=`	assignment by sum and difference		yes
`*= /= %=`	assignment by product, quotient, and remainder		yes
`<<= >>=`	assignment by bitwise left–shift and right–shift		yes
`&= ^= |=`	assignment by bitwise AND, XOR, and OR		yes
`throw`	throw operator	(not available)	
`,`	Comma	left-to-right	yes

Table 1: C++ operators and their priorities

Notes

- A precedence table, while mostly adequate, cannot resolve a few details. In particular, note that the ternary operator allows any arbitrary expression as its middle operand, despite being listed as having higher precedence than the assignment and comma operators. Thus `a ? b , c : d` is interpreted as `a ? (b, c) : d`, and not as the meaningless `(a ? b), (c : d)`.
- Also note that the immediate, unparenthesized result of a C cast expression cannot be the operand of `sizeof`. Therefore, `sizeof(int)*x` is interpreted as `(sizeof(int))*x` and not `sizeof((int)*x)`.

3.4 Namespaces

They provide some privacy for names.

Recommendations

- You *must not* declare anything in the standard namespace `std`.
- You *must not* place a `using` declaration in a header file.
- You *should* avoid the `using` declaration, rather use a short namespace alias and write the namespace classifier.
- You *should* use an unnamed namespace if you need to use file local declarations in a source file.
- You *should not* use the `using`–directive to make all names from a namespace available, use the corresponding classifiers.
- You *should not* use unnamed namespaces in header files.
- You *can* use the `using`–directive on the block level of functions to introduce individual names.

Reasons

- Declarations in the `std`–namespace result in undefined, and unportable behavior.
- If there is a name clash within two namespaces you end up writing the classifiers anyway.
- If the clash appears during on–going development, you may end up calling the wrong function.

3.5 Construction, assignment, and destruction

Think twice, it's crucial.

Recommendations

- You *must* always declare the default constructor, the destructor, the copy–constructor, and the assignment operator.

- If you want the compiler generated versions, you *must* comment the declarations, to make implicitly clear that the omission is intended.
- You *must* use private declaration if the compiler generated versions for constructors and assignment operator are not sufficient, i.e., when you don't want the compiler generate the copy–constructor and/or the assignment operator.
- You *must* declare the destructor virtual when you have at least one virtual member or you expect the class being extended.
- You *must* declare the return type of the assignment operator to be a constant reference.
- You *must* initialize the objects after the `:` in the constructor.
- You *must* use the `explicit` construction whenever the constructor takes only one argument, or clearly comment why the constructor is intended to be used for type conversion.
- You *must not* throw exceptions in a destructor.
- You *should* limit the work done in a constructor to such operations that do not throw exceptions, i.e., that are guaranteed to terminate (for instance they should not wait for input of any kind), and that do not call virtual functions.

```
// We can live with the compiler generated versions.
class SomeThing {
public:
  // SomeThing(const SomeThing& S);
  // const SomeThing& operator=(const SomeThing& S);
  ...
};

// We cannot live with the compiler generated versions
// and we think we don't need an implementation.
class SomeThing {
private:
  SomeThing(const SomeThing& S);
  const SomeThing& operator=(const SomeThing& S);
  ...
};
```

Reasons

- Private declarations generate compile time errors when copying or assigning is needed but (still) not implemented.
- With a virtual destructor you make sure that the destructor of the derived class is called when you delete the base class object.
- With a constant reference as return type of the assignment operator you make sure that the difficult to interpret `(a=b)=c`–construction can be detected.
- Direct correct construction of members is more efficient.
- It is difficult to report errors from a constructor.
- An exception may leave the object in an undefined state which may cause problems later on.
- Virtual functions are still not dispatched to the according level, which might result in unexpected behavior.
- If you need complex initialization, consider to construct the object first the easy way and then call a corresponding initialization method, e.g., instead of passing a file to the constructor where the object should be constructed from, construct the object and call a read method.
- Use of `explicit` avoids implicit type conversions.

3.6 Name coherence

A monkey is a monkey, it doesn't matter where.

Recommendations

- You *must* use the same name if you mean the same thing.
- You *must* use the same name for the parameters both in declaration and in definition.

3.7 Class and template design

It is not only a matter of taste.

3.7.1 Abstract classes, Do–ables

Recommendations

- You *should* use a verb converted to an adjective to define an abstract class.

```
class Comparable {
  ...
};

class Insertable {
  ...
};
```

3.7.2 Liskov's substitution principle

The Liskov's substitution principle states that all classes derived from a base class should be interchangeable when used as a base class.

The idea is that users of a class should be able to count on similar behavior from all classes that derive from a base class. No special code should be necessary to qualify an object before using it.

3.7.3 Open/Closed principle

The Open/Closed principle states that a class must be open and closed where

- open means a class has the ability to be extended.
- closed means a class is closed for modifications other than extension.

The idea is once a class has been approved for use having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

The Open/Closed principle is a pitch for stability. A system is extended by adding new code not by changing already working code. Programmers often don't feel comfortable changing old code, because it works!

In practice the Open/Closed principle simply means making good use of abstraction and polymorphism; abstraction to factor out common processes and ideas, and inheritance to create an interface that must be adhered to by derived classes.

3.8 Declaration

Recommendations

- You *must* declare and define classes and variables in the smallest possible scope.
- You *must* declare loop variables in the `for()`–construction whenever their lifetime ends right after the loop.
- You *must not* declare any other variable in the `for()`–construction.
- You *should* declare loop variables for `while`–loops as close as possible before the loop.

```
for(unsigned int i(0);i<N;++i) ...

bool isDone(false);
while(!isDone) {
  ...
}
```

3.9 Template definitions

Recommendations

- You *must* comment what operations you expect a template parameter to have implemented. C++ does not enforce such contracts.
- You *must* use the `this`–pointer to access member functions in templates. This avoids the possibility that an unwanted, for instance, in the context available global, function is called.

3.10 Types and conversions

- Type conversions *must* always be done explicitly. Never rely on implicit type conversion.
- Types that are local to one file only *should* be declared inside that file.

```
floatValue = static_cast<float>(intValue);
```

3.11 Methods and functions

The methods and functions are the verbs of the programming language.

Recommendations

- Whenever you define a function or method, you *should* think carefully about the scope of the functions: global, class static, class member, or class non–member.
- You *should* declare file local global functions as `static`.
- You *should* avoid a previous declaration of a file local function, when the function can be defined right away.

3.11.1 Names

Recommendations

- The name of the class is implicit, and you *should* avoid it in a method name.

```
class Segment {
  ...
  double Length(void) const;
}
```

3.11.2 Access methods

Recommendations

- You *must* use the name of a class attribute written with leading uppercase letter as the name of a member function to access the attribute of a class by reference.

```
class Person {
private:
  unsigned int age;
  String name;
public:
  const unsigned int& Age() const { return age; }
  unsigned int& Age() { return age; }

  const String& Name() const { return name; }
  String& Name() { return name; }
}
```

3.11.3 Get and set

Recommendations

- You *must* use for `Set/Get` for setting and getting of static variables.
- You *should* use `Set/Get` only if really something is computed.
- You *should* use `Set/Get` only if side–effects take place.

```
class Particle {
private:
  double radius;
  double mass;
  double density;
public:
    // Sets new radius and recomputes mass.
  void SetRadius(const double radius_) {
    radius=radius_;
    mass=4.0/3.0*pi*radius*radius*radius*density;
  }
}
```

3.11.4 Friend declarations

Friend declarations are implicitly public (even if the private keyword is placed before!).

- You *should* avoid any side effect in a friend method.

3.11.5 Verbs

Methods should use standardized verbs to pass implicit information to the programmer.

- `Compute`: indicates that possibly something time consuming is computed.
- `Find`: indicates that possibly a time consuming search operation takes plane, similarly for `FindFirst...`, `FindLast...`, `FindClosest...`
- `Lookup`: indicates that a fast search operation takes place.
- `Initialize`: indicates that a one time operation is taken place.

- `Can`: indicates whether the object has a specific capability.
- `Has`: indicates whether the object has a specific property.
- `Is`: indicates whether the object is of a certain kind.
- `Should`: indicates whether the object provides some hints.

- You *must not* use negated boolean variable names.

```
const bool can_handle(CanHandle());
const bool has_cache(HasCache());
const bool is_visible(IsVisible());
const bool should_sort(ShouldSort());

const bool isError;
```

Reasons
- The programmer is forced to use meaningful names.
- Double negation in logical expressions is avoided (What does `!is_not_error` actually mean?).

Recommendations
- You *should* use complementary names for complementary actions.
- You *should* avoid abbreviations of verbs.
- You *should* use commonly used abbreviations of non–verbs.

```
// examples of complementary verbs include:
// get/set, add/remove, create/destroy, start/stop,
// insert/delete, increment/decrement, old/new, begin/end,
// first/last, up/down, min/max, next/previous, old/new,
// open/close, show/hide, suspend/resume

void ReadCommand(void);        // NOT: ReadCmd(void);
void CopyCommand(void);        // NOT: CpCommand(void);
void InitializeStorage(void); // NOT: InitStorage(void);

void HtmlReader(void);         // NOT: HypertextMarkupLanguageReader()
```

Reasons
- Complexity is reduced by symmetry.
- You automatically add documentation to your code.

3.12 Preprocessor usage

Useful and evil, take care.

3.12.1 `include` directive

Recommendations
- You *must not* use relative file names in include directives.
- You *must* use the directive `#include "filename.h"` for user–prepared include files and the directive `#include <filename.h>` for include files from system libraries.
- You *must* give preference to forward declarations instead of including the entire class declaration, whenever the header file just uses this class with pointers or references.
- You *should* give preference to forward declarations instead of prefixing the point or reference with the `class` keyword.
- You *should* include the files in the following group order: local include files, library include files, C++ include files, C include files. Within each group, you *should* order alphabetically.

Reasons
- The compiler usually has the ability to locate the include files. Relative filename would require a certain directory structure in the file system which might not be convenient in certain environments.
- A forward declaration does not produce a dependency in the file graph, which might reduce the recompilation effort whenever the class is changed.
- Forward declarations grouped at the beginning of the file document what other classes are using.

3.12.2 `define` directive

Recommendations

- You *must* undefine macro definitions that are intended for file local use only.
- You *must* document side effects of macro usage.
- You *should* undefine macro definitions as early as possible.
- You *should* use `#if defined` instead of `#ifdef`.
- You *should* prefer `#if SOMETHING` over `#ifdef SOMETHING`.
- You *should* avoid side effects when defining macros.
- You *should* avoid macro definitions, rather use inline functions.
- You *should* consider templates and template specifications for such inline functions.
- You *should* concentrate the defines in a single file which is included where needed.

```
#if NOT_YET_IMPLEMENTED

#if OBSOLETE

#if TEMP_DISABLED
```

Reasons

- The `#if defined`–form requires the macro to be defined, whereas the simple `#ifdef`–form silently ignores code lines.
- The `#if`–form allows a definition of the macro to 0 with the effect that the section is skipped as well, whereas the simple `#ifdef`–form always would include the section.
- Having macros defined in a scope larger than a file may easily produce name clashes with other packages or libraries, if the macro is not undefined properly.

3.13 Variables and parameters

Recommendations

- Variables *must* be initialized where they are declared, whenever this is possible.
- You *must not* use static or global variables of class type (including standard library objects). Be aware that in C++ there is no order defined for constructor calls of global variables across compilation units.
- Variables *must not* be initialized to a merely phony value.
- Variables *must not* have a dual meaning. If you need a new variable with a new meaning, declare and use a new one.
- You *must* use the same names for parameters in declarations and definitions of functions.
- You *should* use the `auto`-type in C++11 only when the type is not known, otherwise a typedef to shorten names can be used.
- You *should* minimize the use of global variables.
- You *should* use `assert` for parameter checking.
- You *should* specify first input and than output parameters in the parameter list.
- You *should* give preference to reference parameters rather than pointer parameters.

```
SomeObject obj(SomeMethod());

int x, y, z; // here it makes no sense to initialize
getCenter(&x, &y, &z);
```

Reasons

- Constructing objects during declaration is more efficient, otherwise default construction takes place, and later a copy operation is to be performed. (Note that the gcc compiler offers the `-felide-constructors` option which avoids the unnecessary temporary object.)
- All variables/objects are in a consistent states unless for well known reasons. Consider writing a comment if you cannot initialize a variable to a known state.
- References guarantee that the object already exists whereas pointers may be 0, indicating that the object is still not constructed.
- Parameter ordering in function calls according type can have a performance impact.

3.14 Constants

Recommendations

- You *must* declare everything `const` that you can.
- You *must* you `0` not `NULL` for pointer values.
- You *should* use the `nullptr` keyword of the C++11 standard (possibly with an appropriate define for older standards).
- You *should* use the `constexpr` keyword of the C++11 standard.
- You *can* use the implicit conversion of integral, floating point or pointer types to a boolean value.

```
if(errorNum==6) ...

if(nLines) ...
if(value) ...
if(pointer) ...
```

Reasons

- The compiler helps you to find side effect.
- More efficient code can be generated often by the compiler.
- The C++–standard states that zero ints and floats are converted to false. The same is true for the pointer types.
- There is a clear visual difference between comparing to zero and comparing to any other value.

3.15 Loops

Without loops no general computability.

Recommendations

- You *should* use the loop your algorithm needs (loops have a semantic).
- You *should* include the lower limit and exclude upper limit in intervals and iterations.
- You *should* avoid the use of `continue` and `break`.
- You *should* use `for(;;)` (read: forever) for an infinite loop.
- You *should* use `for` loops for counting loops.
- You *should* use `while` loops for conditions that change during the iterations.
- You *should* use `do-while` loops when you require that the iteration is performed at least once.
- You *can* use `while(1)` for an infinite loop.
- You *can* use `continue` and `break` if you really need.
- You *can* use `goto` if you really need it.

```
  for(...) {
    while(...) {
      ...
      if(disaster)
        goto LabelDisaster;
    }
  }
  ...
LabelDisaster:
   // code to clean up the mess
```

Reasons

- The visual impact of `for(;;)` is sufficiently clear that there happens something special here. Consider writing an appropriate comment.
- A clear semantic of a loop is more important than a limited use of loop constructs.
- Unstructured code sometimes is more efficient and shorter than structured code.

3.16 Conditions

Clear conditions reveal clear understanding.

Recommendations

- The nominal case *should* be put in the `if`-part and the exceptional case in the `else`-part of an `if` statement.
- You *must* not use assignments in conditions.
- You *should* avoid complex conditional expressions. Introduce temporary boolean variables instead.

```
std::ofstream out_file(fileName);
if(out_file) {
  ...
}

const bool is_finished((element_no<0)||(element_no>max_element));
const bool is_repeated_entry(element_no==last_element);
if(is_finished||is_repeated_entry) {
  ...
}
```

Reasons

- For debugging purposes, when writing on a single line, it is not apparent whether the test is really true or not. Remember that setting breakpoints and tracing in most debuggers is still line oriented.
- Conditionals with executable statements are just very difficult to read.

3.17 Overloading and overwriting

Talking is talking, and walking is walking.

Recommendations

- You *must* overload operators only with operations that make common sense, in most cases just mathematical operations.
- You *must* overload methods only with variations that have the same semantics, i.e., that can be used for the same purpose.
- You *must* maintain the semantics when you overwrite a method in the derived class, i.e., the derived class should at most specialize the semantics of the method in the base class.
- You *should* avoid overloading of methods with template parameters, whenever there is a risk that the template instantiation might produce the same signature more than once, rather use template specialization.
- You *should* be aware that classes overloading the `&`–operator cannot be declared forwarded any more.

```
class String {
  ...
  bool Contains(const String& s) const;
  bool Contains(const char c) const;
  ...
};

template <typename T>
class Container {
  ...
  void Insert(const int element);
  void Insert(const T element);   // what happens if T==int
  ...
}
```

3.18 Units

Recommendations

- If the program deals with physical units, use a unit suffix in case the unit is not the base unit.

```
double distance;    // here distance is implicitly in [m]
double distance_km; // here is it made clear distance in in [km]
```

3.19 Default values

Recommendations

- You *must* make a label for an error state.
- You *should* make it the first label if possible.

```
enum { STATE_ERR, STATE_OPEN, STATE_RUNNING, STATE_DYING};
```

Reasons

- It is often useful to be able to say an enum is not in any of its valid states.

3.20 Test code

First thought, first fault.

Recommendations

- You *should* first think on how you would implement test code that checks that your algorithm to be implemented is correct.
- You *should* think of starting coding by implementing the test code.

Reasons

- If you plan to implement a sorting algorithm, first implement the code that checks that the output of your algorithm is really sorted.

3.21 Exception handling

An exception should be an exception.

Recommendations

- Exception classes *must* have `Exception` as suffix.
- You *must* document that an exception is being thrown in your method comment header.
- You *must* draw a clear decision what is considered an exception and what can be a simple assert. The main difference, obviously, is the instant in time when the things are checked: asserts at compile time during debugging, exceptions at run time in production code.
- You *should not* use sophisticated exception hierarchies.
- Instead, you *should* create one exception per library or namespace and have an exception reason within that exception to indicate the type of the exception.
- You *should* create a macro for throwing the exception so that you can transparently include `__FILE__` and `__LINE__` in the exception.
- The exception *should* take one string argument so developers can include all the information they need in the exception without having to create a derived exception to add a few more pieces of data.
- Throwing an exception *should* take only one line of code.
- You *can* also automatically take a time stamp and get the thread ID.

- You *can* use derived exceptions, if you think you need to create derived exceptions, but you *must* derive them from your libraries' base exception class.
- You *can* include a stack trace of where the exception happened to provide additional information in complex call graphs.

Reasons

- Creating very elaborate exception hierarchies is a waste of time. Nobody ends up caring and all the effort goes to waste.
- For example, for your operation system encapsulation library, make an exception called `OsEncapException`.
- By using just one exception you make it easy for people using your code to catch your exception.
- If you have many exceptions it is difficult for anyone to handle those, especially in the right order.
- Whenever you add more exceptions to existing ones, you will break existing code that thinks they are handling all your exceptions.
- Having one base class exception, allows your library users to catch the base class exception if they wish.
- Most exceptions are thrown in situations where the code can't do anything about it anyway, so creating lots of exceptions to express the very little thing that something specific went wrong with a separate class is time consuming and confusing for the user.
- Exceptions are mostly used for dealing with abnormal situations, they should not be thought of as normal program flow. The additional information provided by throwing the exception should be useful information.

```
// The condition causes the exception to be thrown when true.
// msg is a local context.
// reason is the specific error code for the exception,
// if you think you need it.
THROW_NAMESPACE_EX_IF(cond,msg,reason);
```

3.22 Version control systems

Recommendations

- You *must not* use RCS/CVS/SVN keywords.

Reasons

- Using keyword substitution changes the file in places which generate unuseful hits in file comparisons.
- The file's content should indicate a revision, the meta data in the revision control system should describe further issues.

4 Documenting guide

This sections is not a tutorial, not even a short description, of Doxygen. Please refer to its documentation for more details. The purpose of this section is to explain how to use the very flexible tool in a manner being consistent with the style and programming discipline described earlier.

4.1 Documentation tool

Recommendations

- You *should* use the Doxygen documentation tool to document your sources.
- You *should* use the ///–style when you use the Doxygen documentation tool.
- You *should* use the \–form of structural commands, rather than the @–form.
- You *should* use the java–doc–autobrief–feature of Doxygen, i.e., the first part of the documenting block until a dot followed by a blank or newline is encountered is automatically considered the brief description.
- You *should not* put documentation after members, variables, and parameters, i.e., restrain from using the //<–form of comments, whenever you need more than just a brief comment.
- You *should not* use the HTML-tags to introduce documentation format.

```
/// This is an example class.
/// There is not much to say about it, but we could write
/// a whole novel about documenting classes.
class Example {
  ...
}
```

Reasons

- The ///–style gives the most uniform visual effect and does not introduce additional almost empty lines.
- In an after member documentation block you cannot use certain structural commands.
- HTML-tags spoil the fluent reading of the comment. The built-in formating rules of Doxygen, for instance for lists, are much simpler and do not consume much space.

4.2 Specific requirements

The documentation tool Doxygen imposes certain requirements such that all documentation blocks are visible in the generated documentation.

Recommendations

- You *must* include a documentation block on file level to obtain the documentation of the global objects in this file.
- You *must* include a documentation block on namespace level to obtain the documentation of its contained objects.
- You *must* include a documentation block on class level to obtain the documentation of its members.