EPITA C99/C++ Coding Style Standard

Edition 30 November 2015 f124591

C/Unix Assistants *et al.*

This document is intended to uniformize the coding styles of EPITA engineering students during their second term.

Covered topics:

- Naming conventions
- Object Oriented considerations
- Local layout (block level)
- Global layout (source file level), including header files and file headers
- Project layout, including `Makefile`'s

The specifications in this document are to be known in detail by all students.

Some rules differ from C99 to C++. The rules with a name starting wth *c++.* only apply to C++. These may be additions, exceptions, or contradictions to previous rules. A summary of the major differences is present at the end of this document.

If you are a beginner in C++, you are **strongly suggested** to entirely read **really** read the C++ Super-FAQ[1].

During the second period, all submitted projects must comply **exactly** with the standard; any infringement causes the mark to be at most 0.

[1] https://isocpp.org/faq.

1 How to Read this Document

This document adopts some conventions described in the following nodes.

1.1 Vocabulary

This standard uses the words *MUST*, *MUST NOT*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY* and *OPTIONAL* as described in RFC 2119.

Here are some reminders from RFC 2119:

MUST This word, or the terms *REQUIRED* or *SHALL*, mean that the definition is an absolute requirement of the specification.

MUST NOT
This phrase, or the terms *PROHIBITED* or *SHALL NOT*, mean that the definition is an absolute prohibition of the specification.

SHOULD This word, or the adjective *RECOMMENDED*, mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understod and carefully weighted before choosing a different course.

SHOULD NOT
This phrase, or the phrase *NOT RECOMMENDED*, mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY This word or the adjective *OPTIONAL*, mean that an item is truly optional. One may choose to include the item because a particular circumstance requires it or because it causes an interesting enhancement. An implementation which does not comply to an *OPTIONAL* item *MUST* be prepared to be transformed to comply at any time.

1.2 Rationale — Intention and Extension

Do not confuse the intention and extension of this document.

The intention is to limit obfuscation abilities of certain students with prior C experience, and uniformize the coding style of all students, so that group work does not suffer from style incompatibilities.

The extension, that is, the precision of each "rule", is there to explain how the automated standard verification tools operate.

In brief, *use your common sense and understand the intention*, before complaining about the excessive limitations of the extension.

1.3 Beware of Examples

Examples of this standard are there for illustratory purposes *only*. When an example contradicts a specification, the specification is authoritative.

Be warned.

As a side-note, do not be tempted to "infer" specifications from the examples presented, or they might "magically" appear in new revisions.

2 Naming Conventions

Names in programs must comply to several rules. They are described in the following nodes:

2.1 General Naming Conventions

`name.lang` [Rule]

Names *MUST* be expressed in English. Names *SHOULD* be expressed in correct English, i.e. without spelling mistakes.

`name.gen` [Rule]

Entities (variables, functions, macros, types, files or directories) *SHOULD* have explicit and/or mnemonic names.

```
#define MAX_LINE_SIZE               1024
#define COMMENT_START_DELIMITER     '#'
#define MAX_FILE_NAME_LENGTH        2048
```

`name.abbr` [Rule]

Names *MAY* be abbreviated, but only when it allows for shorter code without loss of meaning. Names *SHOULD* even be abbreviated when long standing programming practice allows so:

```
maximum ↦ max
minimum ↦ min
length  ↦ len
...
```

`name.sep` [Rule]

Composite names *MUST* be separated by underscores ('`_`').

`name.reserved` [Rule]

You *MUST NOT* define any kind of name that begins with an underscore. They are reserved by the implementation of the compiler and standard libraries. This also applies for preprocessor macros (including header guards).

`c++.name.members` [Rule]

You *MUST* name public members '`like_this`'. No upper case letters, and words are separated by an underscore.

You *MUST* name private and protected members '`like_this_`', with a trailing underscore.

Rationale: it is extremely convenient to have a special convention for private and protected members: you make it clear to the reader, you avoid gratuitous warnings about conflicts in constructors, you leave the "beautiful" name available for public members etc.

For instance, write:

```
class IntPair
  {
  public:
    IntPair(int first, int second)
      : first_(first)
      , second_(second)
    {}
```

```
protected:
  int first_;
  int second_;
}
```

2.2 Letter Case

`name.case` [Rule]

Variable names, C function names and file names *MUST* be expressed using lower case letters, digits and underscores **only**. More precisely, entity names *MUST* be matched by the following regular expression:

```
[a-z][a-z0-9_]*
```

Rationale: while this is a technical requirement for C code, it is not for filenames. Filenames with uncommon characters or digit prefixes are inelegant.

`c++.name.case` [Rule]

You *MUST* use title case (otherwise known as "upper camel case") for your class names (defined by `class` *and* `struct`); for instance '`Exp`, `StringExp`, `TempMap`, `InterferenceGraph`' etc. This also applies to class templates.

For instance, write:

```
using baz_type = OtherClass;

my_type c_function(my_type my_argument)
{
  return bar_foo(my_argument);
}

MyClass OtherClass::cpp_function(MyClass my_object)
{
  return foo_bar(my_object);
}
```

`c++.name.file` [Rule]

The declaration of a class '`LikeThis`' *MUST* be placed in a header file named `like-this.hh`. Note that the mixed case class names are mapped onto lower case words separated by dashes.

There can be exceptions, for instance auxiliary classes used in a single place do not need a dedicated set of files.

If there is only one implementation file (`.cc`, `.hxx`) related to a header file `like-this.hh`, it *MUST* be named accordingly; only the extension will differ. Otherwise, files *MUST* be placed into a folder named `like-this`. In the latter case, no naming constraint is applied.

`c++.name.space` [Rule]

You *MUST* name your namespaces using lowercase only: '`likethis`'.

`name.case.macro` [Rule]

Macro names *MUST* be upper case. Macro arguments *MUST* be in title case:

```
#define MAX(Left, Right) ((Left) > (Right) ? (Left) : (Right))
```

2.3 Name Prefixes

`type.typedef` [Rule]

Structure and union names *SHOULD NOT* be aliased using '`typedef`' for aesthetic reasons.

Rationale: '`typedef`' is an often misused keyword that serves its purpose only on fully abstracted types (i.e. types manipulated only through functions).

`name.prefix` [Rule]

Structure and union names *MAY* be aliased using '`typedef`'. It is mandatory to define shortcut names to structures, unions and enumerations using the significant prefix: '`s_`' for structures, '`u_`' for unions and '`e_`' for enumerations.

```
typedef struct text
{
  t_string title;
  t_string content;
} s_text;

typedef union
{
  char      c;
  short     s;
  int       i;
  long long l;
} u_cast;

typedef int (*f_open)(char*, int, int);
```

Rationale: for using prefixes: they are the first characters read while the eye is parsing, and allow to tag the identifier without need to read it entirely.

`c++.name.prefix` [Rule]

You *MUST NOT* use C prefix in C++ sources.

`c++.type.suffix` [Rule]

When declaring types (eg., when using '`using`', type names *MUST* be suffixed with '`_type`'.

```
using value_type = typename MyTraits<T>::res;
using map_type = std::map<const Symbol, Entry_T>;
using symtab_type = std::list<map_type>;
```

Rationale: this is a common idiom in C++ (which happens to be used by the STL)

Rationale: for not using '`_t`': identifiers ending with '`_t`' are reserved by POSIX.

`name.prefix.redef` [Rule]

When redefining a '`typedef`' on an type that is already a '`typedef`', the prefix of the type *MUST* be preserved if the original type is prefixed by '`e_`', '`f_`', '`s_`', '`t_`' or '`u_`'.

```
typedef struct foo  s_foo;
typedef s_foo       s_foo_rename;
```

`name.prefix.global` [Rule]

Global variable identifiers (variable names in the global scope), when allowed/used, *MUST* start with '`g_`'.

3 Preprocessor-level Specifications

The global layout of files, and sections of code pertaining to the C preprocessor, including file inclusion and inclusion protection, must comply to specifications detailed in the following sections.

3.1 File Layout

`file.80cols` [Rule]
Lines *MUST NOT* exceed 80 columns in width, excluding the trailing newline character.

`file.indentation` [Rule]
Indentation *MUST* be done using whitespaces only, tabulations *MUST NOT* appear in your source code.

Rationale: Some automatic tools send code snippets by mail.

`file.terminate` [Rule]
The last line of the file *MUST* end with a line feed.

`file.dos` [Rule]
The DOS CR+LF line terminator *MUST NOT* be used. Hint: do not use DOS or Windows standard text editors.

`file.trailing` [Rule]
There *MUST NOT* be any whitespace at the end of a line.

Rationale: although this whitespace is usually not visible, it clobbers source code with useless bytes.

`file.spurious` [Rule]
There *MUST NOT* be any blank lines at the beginning or the end of the file.

`file.deadcode` [Rule]
In order to disable large amounts of code, you *SHOULD NOT* use comments. Use '`#if 0`' and '`#endif`' instead. Delivered project sources *SHOULD NOT* contain disabled code blocks.

Rationale: C comments do not nest.

3.2 Preprocessor Directives Layout

`cpp.mark` [Rule]
The preprocessor directive mark ('`#`') *MUST* appear on the first column.

`cpp.if` [Rule]
Preprocessor directives following '`#if`' and '`#ifdef`' *MUST* be indented by one character:

```
#ifndef DEV_BSIZE
# ifdef BSIZE
#  define DEV_BSIZE BSIZE
# else /* !BSIZE */
#  define DEV_BSIZE 4096
# endif /* BSIZE */
#endif /* !DEV_BSIZE */
```

As shown in the previous example, '`#else`' and '`#endif`' *MUST* be followed by a comment describing the corresponding condition. Note that the corresponding condition of the '`#else`' is the negation of that of the '`#if`'.

Rationale: In large files, the developer might lose sight of the '`#if`' test.

`cpp.linebreak` [Rule]

When a directive must span over multiple lines, escaped line breaks ('\'-newline) *MUST* appear on the same column. As a reminder, this *MUST* be done with whitespaces only.

This is wrong:

```
#define SWAP(A, B) \
  do { \
    A ^= B; \
    B ^= A; \
    A ^= B; \
  } while (0)
```

This is correct:

```
#define SWAP(A, B) \
  do {             \
    A ^= B;        \
    B ^= A;        \
    A ^= B;        \
  } while (0)
```

`cpp.token` [Rule]

The pre-processor *MUST NOT* be used to split a token. For instance, doing this:

```
in\
t ma\
in ()
{
}
```

is forbidden.

3.3 Macros and Code Sanity

As a general rule, preprocessor macro calls *SHOULD NOT* break code structure. Further specification of this point is given below.

`cpp.macro` [Rule]

Macro call *SHOULD NOT* appear where function calls wouldn't otherwise be appropriate. Technically speaking, macro calls *SHOULD parse* as function calls.

This is bad style:

```
#define FOO(Arg)      \
  return (Arg) + 1;   \
}

int main()
{
  FOO(42)
```

This is more elegant:

```
#define FOO(Arg)    \
  (Arg) + 1

int main()
{
  return FOO(42);
}
```

Rationale: macros should not allow for hidden syntactic "effects". The automated standard conformance tool operates over unprocessed input, and has no built-in preprocessor to "understand" macro effects.

`cpp.macro.case` [Rule]

Macro names *MUST* be entirely capitalized.

`cpp.macro.style` [Rule]

The code inside a macro definition *MUST* follow the specifications of the standard as a whole.

`cpp.macro.variadic` [Rule]

Macros *MAY* be variadic:

```
#define EPRINTF(...)                                          \
  do {                                                        \
    fprintf(stderr, "%s:%d: error: ", __FILE__, __LINE__);    \
    fprintf(stderr, __VA_ARGS__);                             \
  } while (0)
```

3.4 Comment Layout

`comment.lang` [Rule]

Comments *MUST* be written in the English language. They *SHOULD NOT* contain spelling errors, whatever language they are written in. However, omitting comments is no substitute for poor spelling abilities.

`comment.single` [Rule]

Single-line comments *MAY* be used, even outside the functions' body.

`comment.multi` [Rule]

The delimiters in multi-line comments *MUST* appear on their own line. Intermediary lines are aligned with the delimiters, and start with '`**`':

```
/*                                  /*
 * Incorrect                        ** Correct
 */                                 */

/* Incorrect                       /**
 */                                ** Correct
                                   */

                                   /* Correct */

                                   // Correct
```

`doc.obvious` [Rule]

You *SHOULD NOT* document the obvious.

`doc.style` [Rule]

You *SHOULD* use the imperative when documenting, as if you were giving order to the function or entity you are describing. When describing a function, there is no need to repeat "function" in the documentation; the same applies obviously to any syntactic category.

`doc.doxygen` [Rule]

Documentation *SHOULD* be done using *doxygen*.

In order to use *doxygen*, the following type of comments is allowed:

```
/**
** Doxygen comment.
** Notice the additional '*' in the first line.
*/
```

For additional specifications about comments, see Section 3.1 [File Layout], page 6 and Chapter 6 [Global Specifications], page 26.

3.5 Header Files and Header Inclusion

`cpp.guard` [Rule]

Header files *MUST* be protected against multiple inclusions. You *MAY* use the `#pragma once` directive in order to achieve this. Otherwise, you *MUST* use `#include` guards. In the latter case, the protection "guard" *MUST* contain the name of the file, in upper case, and with (series of) punctuation replaced with single underscores. It *MAY* also contain additional upper case letters and underscores. For example, if the file name is `foo++.h`, the protection key *MUST* contain '`FOO_H`':

```
#ifndef FOO_H
# define FOO_H
/*
** Contents of foo++.h
*/
#endif /* !FOO_H */
```

For instance, in case `foo++.h` is in a `modules` directory, the guard *MAY* be as follow:

```
#ifndef MODULES_FOO_H_
# define MODULES_FOO_H_
/*
** Contents of foo++.h
*/
#endif /* !MODULES_FOO_H_ */
```

`cpp.include` [Rule]

When including headers, **all** inclusion directives ('`#include`') *SHOULD* appear at the start of the file. Inclusion of system headers *SHOULD* precede inclusion of local headers.

This is bad style:

```
#ifndef FOO_H
# define FOO_H

int bar();

# include "baz.h"

int foo();

# include <stdio.h>
#endif /* !FOO_H */
```

This is elegant:

```
#include <stdio.h>

#include "foo.h"
#include "baz.h"

int
foo()
{
  return baz();
}

int
bar()
{
  return foo();
}
```

`cpp.include.filetype` [Rule]

Included files *MUST* be header files, that is, files ending with a '`.h`' (a '`.hh`' or a '`.hxx`' for C++) suffix. You *MUST NOT* include source files. In order to use X-Macros, you *MAY* include '`.def`' files in your code.

`cpp.digraphs` [Rule]

Digraphs and trigraphs *MUST NOT* be used.

4 Writing Style

The following sections specify various aspects of what constitutes good programming behaviour at the language level. They cover various aspects of C constructs.

4.1 Blocks

`braces` [Rule]

All braces *MUST* be on their own line.

This is wrong:

```
if (x == 3) {
  // Something
}
```

This is correct:

```
if (x == 3)
{
  // Something
}
```

`c++.braces.empty` [Rule]

As an exception, you *MAY* put an opening and a closing brace together on a single line, with no space (or anything) between them, to denote an empty function body.

These are wrong:

```
foo::Foo()
{  }

foo::~Foo()
{ /* Comment. */ }
```

These are correct:

```
foo::Bar()
{}
foo::~Bar()
  {}
```

`c++.braces.comment` [Rule]

As another exception, you *MAY* add a comment after a closing brace in order to specify what is getting closed.

```
namespace foo
{
  // 500 lines here.
} // namespace foo
```

But *SHOULD NOT* state the obvious.

`c++.braces.try` [Rule]

As yet another exception, you *MAY* put the opening brace after a try or a do on the same line. This code is correct:

```
int foo()
{
  int c = 0;
  int r = 0;

  do {
    try {
      r = bar(++c);
    }
    catch (...)
    {
      r = 0;
    }
```

```
  } while (c < 42 && !r);
  return r;
}
```

`braces.close` [Rule]
Closing braces *MUST* appear on the same column as the corresponding opening brace.

`braces.open` [Rule]
Opening braces *SHOULD* appear on the same column as the text before, or indented by 2 spaces.

`braces.indent` [Rule]
The text between two braces *MUST* be indented by a fixed, homogeneous amount of whitespace. This amount *SHOULD* be 2, but *MAY* be 4 spaces. As a reminder of a previous rule, tabulations *MUST NOT* be used.

`braces.indent.style` [Rule]
In other words, the indentation style *MUST* be Allman (also known as "ANSI" or "BSD") or GNU (prefer the former), and *MUST NOT* be any one of the following: K&R, 1TBS, Whitesmith, Horstmann, Lisp.

These are wrong:

```
if (x == 3) {
  foo();
  bar();
}

if (x == 3)
  {
  foo();
  bar()
  }

if (x == 3)
{ foo();
  bar();
}

if (x == 3)
{
  foo();
  {
    bar(); } }
```

These are correct:

```
if (x == 3)
{
  foo3();
  bar();
}

if (x == 3)
  {
    foo();
    bar();
  }
```

`braces.dowhile` [Rule]
As an exception to the previous rules, the opening brace of a '`do...while`' block *SHOULD* be on the same line as the '`do`'.

This is wrong:

```
do
{
  // Something
} while (0);
```

This is correct:

```
do {
  // Something
} while (0);
```

`braces.useless` [Rule]

Useless braces *SHOULD* be omitted.

```
if (x <= tree->key)
  tree = insert_left(tree,
                     make_node(x));
else
  tree = insert_right(tree,
                      make_node(x));
```

4.2 Structures Variables and Declarations

4.2.1 Alignment

`decl.align` [Rule]

Declared identifiers *MAY* be aligned with each other when they are at the start of a block.

This is correct:

```
int foo(void)
{
  int         i = 0;
  const char *s = NULL;
  // some code
  return i;
}
```

This is correct too:

```
int foo(void)
{
  int i = 0;
  const char *s = NULL;
  // some code
  return i;
}
```

Rationale: Some might argue that one style offers more clarity than the other. However, there is no universal consensus as to which one that is.

4.2.2 Declarations

`decl.single` [Rule]

There *MUST* be only one declaration per line.

`decl.point` [Rule]

The pointer symbol (‘`*`’) in declarations *MUST* appear next to the variable name, not next to the type.

The following is incorrect:

```
// and probably does not have
// the intended meaning either
const char* str1, str2;
```

The following is correct:

```
const char *str1;
const char *str2;
```

`c++.decl.point` [Rule]

The pointer symbol (‘`*`’) *MUST* be put near the type, not near the variable.

```
const char* p;  // not ‘const char *p;’
std::string& s; // not ‘std::string &s;’
void* magic();  // not ‘void *magic();’
```

`decl.fields` [Rule]

When declaring a structure or a union, there *SHOULD* be only **one** field declaration per line. Enumeration values *SHOULD* also appear on their own lines.

This is bad style:

```
struct s_point
{
  int x, y;
  long color;
};
```

This is more elegant:

```
struct s_point
{
  int x;
  int y;
  long color;
};
```

`decl.enum` [Rule]

Enumeration values *SHOULD* be upper case.

`decl.inner` [Rule]

Inner declarations (i.e. at the start of inner blocks) *SHOULD* be used (they can help improve readability, or compiler optimizations). Variables *SHOULD* be declared close to their first use and their scope must be kept as small as possible.

`decl.init` [Rule]

Variables *SHOULD* be initialized at the point of declarations, by expressions that are valid according to the rest of the coding style.

The following is wrong:

```
char c = (str++, *str);
int  i = tab<:c:>
```

These are correct:

```
int bar = 3 + 6;
char *opt = argv[cur];
unsigned int *foo = &bar;
unsigned int baz = 1;
static int yay = -1;
int *p = NULL;
char *opt = argv[2];
int mho = CALL(x);
size_t foo = strlen("bar");
```

Hint: to detect uninitialized local variables, use the '`-Wuninitialized`' flag with GCC.

`decl.init.multiline` [Rule]

When initializing a local structure (a C99 feature) or a static array, the initializer value *MUST* start on the line after the declaration:

This is wrong:

```
s_point p1 = { .x = 0, .y = 1, .color = 42 };
```

This is correct:

```
s_point p1 =
{
  .x = 0,
  .y = 1,
  .color = 42
};
```

This is wrong:

```
static int primes[] = { 2, 3, 5, 7, 11 };
```

These are correct:

```
static const int primes[] =
{
   2, 3, 5, 7, 11
};

static const struct
{
  char c;
  void (*handler)(void *);
} handlers[] =
{
  { 'h', &left_handler },
  { 'j', &up_handler },
  { 'k', &down_handler },
  { 'l', &right_handler },
  { '\0', 0 }
};
```

`decl.type` [Rule]

When negative values are impossible, you *MUST* specify '`unsigned`'. Appropriate typedef (such as '`size_t`') *SHOULD* be used when available.

`decl.vla` [Rule]

When declaring an array of automatic storage duration, its size specifier *MUST* be a *constant expression*, whose value can be computed at *compile time*. Variable-length arrays are therefore not allowed.

4.3 Statements

`stat.single` [Rule]

A single line *MUST NOT* contain more than one statement.

This is wrong:

```
x = 3; y = 4;
```

This is correct:

```
x = 3;
x = 4;
```

`stat.sep` [Rule]

Commas *MUST NOT* be used outside the '`for`' construct. The following is wrong:

```
x = 3, y = 4;
```

`comma` [Rule]

The comma *MUST* be followed by a single space, *except* when they separate arguments in function (or macro) calls and declarations and the argument list spans multiple lines: in such cases, there *MUST NOT* be any trailing whitespace at the end of each line.

`semicolon` [Rule]

The semicolon *MUST NOT* be preceded by a whitespace, except if alone on its line. However, a semicolon *SHOULD NOT* be alone on its line. If the semicolon is followed by a non-empty statement, a single whitespace *MUST* follow the semicolon.

`stat.asm` [Rule]

The asm declaration MUST NOT be used.

For a detailed review of exceptions to the three previous rules, See Section 4.5 [Control Structures], page 16.

`keyword` [Rule]

Keywords *MUST* be followed by a single whitespace, *except* those without arguments, which *MUST NOT* be separated from the following semicolon by a whitespace.

This is wrong:

```
if(true)
  break ;
```

This is right:

```
if (true)
  break;
```

`c++.keyword.template` [rule]

As an exception of the previous rule, you are allowed (even *RECOMMENDED*) to write '`template<>`' for explicit specializations.

```
template<>
class Foo<int>
{
  ...
};
```

`keyword.return` [Rule]

When the '`return`' statement takes an argument, this argument *MUST NOT* be enclosed in parenthesis, unless that argument spans multiple lines, in which case it *MUST* be enclosed in parenthesis, and the following lines *MUST* start on the same column as the opening parenthesis. When there is no argument, there *MUST NOT* be any parenthesis.

These are wrong:

```
return (0);
return (a + b);
```

These are right:

```
return 0;
return a + b;
```

These are wrong:

```
return ();

return a && b ||
  c && d;
```

These are correct:

```
return;

return (a && b
        || c && d);
```

`keyword.goto` [Rule]

The '`goto`' statement *MUST NOT* be used.

4.4 Expressions

`exp.padding` [Rule]

All binary and ternary operators *MUST* be padded on the left and right by one space, **including** assignment operators.

Prefix and suffix operators *MUST NOT* be padded, neither on the left nor on the right.

When necessary, padding is done with a single whitespace.

`exp.nopadding` [Rule]

The '`.`' and '`->`' operators *MUST NOT* be padded.

This is wrong:

```
x+=10*++ x;
y=a?b:c;
```

This is correct:

```
x += 10 * ++x;
y = a ? b : c;
```

`c++.exp.padding` [Rule]
The '`operator++`', '`operator()`', '`operator[]`', '`operator&`' (etc.) operators *MUST* be written thusly. You *MUST NOT* insert any whitespace. This is wrong:

```
operator ()
```

`exp.linebreak` [Rule]
Expressions *MAY* span over multiple lines. When a line break occurs within an expression, it *MUST* appear just before a binary operator, in which case the binary operator *MUST* be indented with at least an extra indentation level.

This is wrong:

```
if (a &&
    b)
  // something
```

This is correct:

```
if (a
    && b)
  // something
```

`exp.parentheses` [Rule]
There *MUST NOT* be any whitespace following an opening parenthesis nor any whitespace preceding a closing parenthesis.

`exp.args` [Rule]
There *MUST NOT* be any whitespace between the function or method name and the opening parenthesis for arguments, either in declarations or calls.

This is wrong:

```
int foo ();
int bar (int baz)
{
  return bar (baz);
}
```

This is correct:

```
int foo();
int bar(int baz)
{
  return bar(baz);
}
```

`keyword.arg` [Rule]
"Functional" keywords *MUST* have their argument(s) enclosed between parenthesis. Especially note that '`sizeof`' *is* a keyword, while '`exit`' *is not*.

This is wrong:

```
p1 = malloc (3 * sizeof(int));
p2 = malloc(2 * sizeof p1);
```

These are correct:

```
p1 = malloc(3 * sizeof (int));
p2 = malloc(2 * sizeof (p1));
```

4.5 Control Structures

4.5.1 General Rules

`ctrl.single` [Rule]
The conditional parts of if and while control structures, and the `else` keyword, *MUST* be alone on their line.

These constructs are incorrect:

```
while (*s) write(1, s++, 1);

if (x == 3) {
  foo3();
  bar();
} else {
  foo();
  baz();
}
```

These are correct:

```
while (*s)
  write(1, s++, 1);

if (x == 3)
{
  foo3();
  bar();
}
else
{
  foo();
  baz();
}
```

4.5.2 'else if'

`ctrl.elif` [Rule]

As an exception to the previous rule, an '`else if`' construct *SHOULD* be written on a single line.

This is bad style:

```
if (x == 0)
  foo0();
else
  if (x == 3)
    foo3();
  else
    foo();
```

This is better:

```
if (x == 0)
  foo0();
else if (x == 3)
  foo3();
else
  foo();
```

4.5.3 'for'

Exceptions to other specifications (See Section 4.3 [Statements], page 14, see Section 4.2 [Structures Variables and Declarations], page 12) can be found in this section.

`ctrl.for` [Rule]

Multiple statements *MAY* appear in the initial and iteration part of the '`for`' structure. For this effect, commas *MAY* be used to separate statements. This is correct:

```
for (yyp0 = yystackp->yyitems, yyp1 = yynewItems, yyn = yysize;
     0 < yyn;
     yyn -= 1, yyp0 += 1, yyp1 += 1)
  {
    /* ... */
  }
```

`ctrl.for.split` [Rule]

The three parts of the '`for`' construct *MAY* span over multiple lines.

`ctrl.for.init` [Rule]

Variables *SHOULD* be declared in the initial part of the '`for`' construct.

This is wrong:

```
int i;
for (i = 0; i < 42; ++i)
  {
    /* ... */
  }

/* 'i' is not used outside
    the loop's scope.  */
```

These are correct:

```
for (int i = 0; i < 42; ++i)
  {
    /* ... */
  }

int j;
for (j = 0; j < 42; ++j)
  {
    /* ... */
  }
printf("j = %d\n", j);
```

`ctrl.for.empty` [Rule]

Each of the three parts of the '`for`' construct *MAY* be empty. Note that more often than not, the '`while`' construct better represents the loop resulting from a '`for`' with an empty initial part.

When a part of the '`for`' construct is empty, the whitespace following the previous semicolon *SHOULD* be omitted.

These are wrong:

```
for ( ;;)
for ( ; ; )
for ( ;i < 42; )
```

These are correct:

```
for (;;)
for (; i < 42; )
```

This is not elegant:

```
for (; ; )
```

4.5.4 Loops, General Rules

`ctrl.empty` [Rule]

To emphasize the previous rules, even single-line loops ('`for`' and '`while`') *MUST* have their body on the following line. This *MAY* be a single semicolon, but *SHOULD* be the '`continue`' statement.

This is wrong:

```
for (len = 0; *str; ++len, ++str);
```

This is correct:

```
for (len = 0; *str; ++len, ++str)
  continue;
```

Rationale: the semicolon at the end of the first line is a common source of hard-to-find bugs, such as:

```
while (*str);
  ++str;
```

Notice how the discreet semicolon introduces a bug.

4.5.5 The '`switch`' Construct

`ctrl.switch` [Rule]

Incomplete '`switch`' constructs (that is, which do not cover all cases), *MUST* contain a '`default`' case, unless when used over an enumeration type, in which case it *SHOULD NOT* contain one.

`ctrl.switch.cross` [Rule]

Non-empty '`switch`' condition blocks *SHALL NOT* crossover. That is, all non-empty '`case`' blocks *SHOULD* end with a '`break`', **including** the '`default`' block. This restriction is tampered by some particular uses of '`return`', as described below.

`ctrl.switch.control` [Rule]

Control structure *MUST NOT* span over several '`case`' blocks.

This is very wrong (but valid C):

```
switch (c)
{
case c_x:
  while (something)
  {
     foo();
case c_y:
      bar();
  }
}
```

`ctrl.switch.indentation` [Rule]

Each '`case`' conditional *MUST* be on the same column as the construct's opening brace, which *MAY* be indented from the '`switch`', and the code associated with the '`case`' conditional *MUST* be indented from the '`case`'.

This is wrong:

```
switch (c)
{
  case c_x: foo(); break;
  case c_y:
    bar();
    break;
  default:
    break;
}
```

This is correct:

```
switch (c)
{
case c_x:
  foo();
  break;
case c_y:
  bar();
  break;
default:
  break;
}
```

This is also correct:

```
switch (c)
  {
  case c_x:
    foo();
    break;
  case c_y:
    bar();
    break;
  default:
    break;
  }
```

`ctrl.switch.return` [Rule]

When a '`case`' block contains a '`return`' statement at the same level than the final '`break`', then all '`case`' blocks in the same '`switch`' (*including* '`default`') *SHOULD*

end with '`return`', too. In this particular case, the '`return`' statement *MAY* replace the '`break`' statement.

This is inelegant:

```
switch (color)
{
case BLACK:
  return foo();
  break;
case WHITE:
  return bar();
  break;
default:
  break;
}
return foobar();
```

This is elegant:

```
switch (color)
{
case BLACK:
  return foo();
case WHITE:
  return bar();
}
return foobar();
```

Rationale: when using '`switch`' to choose between different return values, no condition branch should allow to "fall off" without a value.

`ctrl.switch.merge` [Rule]

Empty '`case`' blocks *MAY* be put on a single line. This is correct:

```
switch (direction)
{
case d_left: case d_right:
  break;
}
```

`ctrl.switch.padding` [Rule]

There *MUST NOT* be any whitespace between a label and the following colon (":"), or between the '`default`' keyword and the following colon.

5 Object Oriented Considerations

5.1 Lexical Rules

`lexical.order` [Rule]

You *MUST* order class members by visibility first.

`lexical.order.visibility` [Rule]

When declaring a class, start with `public` members, then `protected`, and last `private` members. Inside these groups, you are invited to group by category, i.e., methods, types, and members that are related should be grouped together.

Rationale: People reading your class are interested in its interface (that is, its `public` part). `private` members should not even be visible in the class declaration (but of course, it is mandatory that they be there for the compiler), and therefore they should be "hidden" from the reader. This is an example of what should **not** be done:

```
class Foo
{
public:
  Foo(std::string, int);
  virtual ~Foo();

private:
  typedef std::string string_type;
public:
  std::string bar_get() const;
  void bar_set(std::string);
private:
  string_type bar_;

public:
  int baz_get() const;
  void baz_set(int);
private:
  int baz_;
}
```

instead, write:

```
class Foo
{
public:
  Foo(std::string, int);
  virtual ~Foo();

  std::string bar_get() const;
  void bar_set(std::string);

  int baz_get() const;
  void baz_set(int);

private:
  typedef std::string string_type;
```

```
  string_type bar_;
  int baz_;
}
```

and add useful Doxygen comments.

`lexical.inherit` [Rule]

You *SHOULD* keep superclasses on the class declaration line. Leave a space at least on the right hand side of the colon. If there is not enough room to do so, leave the colon on the class declaration line.

```
class Derived: public Base
{
  // ...
};

/// Object function to compare two Temp*.
struct temp_ptr_less:
  public std::binary_function<const Temp*, const Temp*, bool>
{
  bool operator()(const Temp* s1, const Temp* s2) const;
};
```

`lexical.inline` [Rule]

You *SHOULD* use `inline` in implementations (i.e., `*.hxx`, possibly `*.cc`), not during declarations (`*.hh` files).

`lexical.virt` [Rule]

You *MUST* repeat `virtual` in subclass declarations. If a method was once declared `virtual`, it remains virtual. Nevertheless, as an extra bit of documentation to your fellow developers, repeat this `virtual`:

```
class Base
{
  public:
    // ...
    virtual foo();
};

class Derived: public Base
{
  public:
    // ...
    virtual foo();
};
```

`lexical.template` [Rule]

You *MUST NOT* leave spaces between template name and effective parameters:

```
std::list<int> l;
std::pair<std::list<int>, int> p;
```

with a space after the comma:

```
std::list<std::list<int>> ls;
```

These rules apply for casts:

```
// Come on baby, light my fire.
int* p = static_cast<int*>(42);
```

`lexical.template.space` [Rule]

You *MUST* leave one space between `template` and formal parameters. Write

```
template <class T1, class T2>
struct pair;
```

with one space separating the keyword `template` from the list of formal parameters. For explicit specializations you *MAY* (and are *RECOMMENDED* to) write `template<>`:

```
template<>
struct pair<int, int>
{ ... };
```

`initialization` [Rule]

You *MUST* put initializations below the constructor declaration.

Don't put initializations or constructor invocations on the same line as the constructor. As a matter of fact, you *MUST NOT* even leave the colon on that line. Instead of `A::A(): B(), C()`, write either:

```
A::A()
  : B()
  , C()
{
}
```

or

```
A::A()
  : B(), C()
{
}
```

Rationale: the initialization belongs more to the body of the constructor than its signature. And when dealing with exceptions leaving the colon above would yield a result even worse than the following.

```
A::A()
try
  : B()
  , C()
{
}
catch (...)
{
}
```

5.2 Do's and Don'ts

`use` [Rule]

Read the C++ Super-FAQ[1]. This is by far the most useful document you can read while learning C++.

`flags` [Rule]

Trust your compiler. Let it be your friend. Use the warnings: `-Wall -Wextra`.

`read` [Rule]

Read what Tiger Assignment says about the use of STL[2].

[1] https://isocpp.org/faq.

[2] http://lrde.epita.fr/~akim/ccmp/assignments.split/Use-of-STL.html.

`const.arg` [Rule]

You *SHOULD* declare your constructors with one argument `explicit` unless you know you want them to trigger implicit conversions.

`init.default` [Rule]

You *MUST* initialize the attributes of an object with the constructor's member initialization line whenever it's possible.

Don't write:

```
class Foo
{
public:
  Foo(int i);
private:
  int i_;
};

Foo::Foo(int i)
{
  i_ = i;
}
```

But instead write:

```
class Foo
{
public:
  Foo(int i);
private:
  int i_;
};

Foo::Foo(int i)
  : i_(i)
{
}
```

`const.fail` [Rule]

When a constructor fails, it *MUST* throw an exception instead of leaving the newly created object in a zombie state (see this link[3]).

`destr.fail` [Rule]

A destructor *MUST NOT* fail, that is, it *MUST NOT* throw an exception (see this link[4]).

`destr.virt` [Rule]

Your destructor *MUST* be `virtual` if there is at least one virtual method in your class (see When should my destructor be virtual?[5]).

`op.overload` [Rule]

You *MUST NOT* overload `operator,`, `operator||` and `operator&&`.

[3] https://isocpp.org/wiki/faq/exceptions#ctors-can-throw.

[4] https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw.

[5] https://isocpp.org/wiki/faq/virtual-functions#virtual-dtors.

`op.overload.binand` [Rule]
You *SHOULD NOT* overload `operator&`.

`friend.bad` [Rule]
You *MUST NOT* use `friend`s to circumvent bad design.

`throw` [Rule]
You *MUST NOT* throw literal constants. You *SHOULD NOT* throw anything else than a temporary object. In particular, you *SHOULD NOT* throw objects created with `new`. See this FAQ[6].

`throw.paren` [Rule]
You *MUST NOT* put parenthesis after a `throw` because some compilers (including some versions of GCC) will reject your code. Here is an example of what should not be done:

```
int main()
{
  throw (Foo());
}
```

`throw.catch` [Rule]
You *SHOULD* catch by reference.

`faq.read` [Rule]
Once again, you *SHOULD* **really** read the C++ Super-FAQ[7]. This is by far the most useful document you can read while learning C++.

[6] https://isocpp.org/wiki/faq/exceptions#what-to-throw.
[7] https://isocpp.org/faq.

6 Global Specifications

Some general considerations about the C sources of a project are specified in the following sections.

6.1 Casts

`casts` [Rule]

As a general rule, C casts *MUST NOT* be used. The only exception to this requirement is described below.

Rationale: good programming behavior includes proper type handling.

`casts.exception` [Rule]

For the purpose of so-called genericity, explicit conversion between *compatible pointer types* using casts *MAY* be used, but *only* with the explicit allowance from the assistants. "Compatible" pointer types are types accessible from one another in the subtyping or inheritance graph of the project.

Hint: if you do not know what are subtyping nor inheritance, avoid using casts.

`c++.casts` [Rule]

Use the C++ casts instead: '`dynamic_cast`', '`static_cast`', '`const_cast`', '`reinterpret_cast`'. However, the use of '`reinterpret_cast`' is *NOT RECOMMENDED*.

6.2 Functions and Prototyping

`fun.proto` [Rule]

Any exported function *MUST* be properly prototyped.

`fun.proto.layout` [Rule]

Prototypes for exported function *MUST* appear in header files and *MUST NOT* appear in source files. The source file which defines an exported function *MUST* include the header file containing its prototype.

This layout is correct:

`my_string.h`:

```
#ifndef MY_STRING_H
# define MY_STRING_H

# include <stddef.h>

size_t my_strlen(const char *);
char   *my_strdup(const char *);

#endif /* !MY_STRING_H */
```

`my_strlen.c`:

```
#include "my_string.h"

size_t my_strlen(const char *s)
{
  // Implementation
}
```

`my_strdup.c`:

```
#include "my_string.h"

char *my_strdup(const char *s)
{
  // Implementation
}
```

`fun.proto.type` [Rule]

Prototypes *MUST* specify **both** the return type and the argument types.

`fun.proto.gnu` [Rule]

You *MAY* specify the return type of a function or method on its own line. This is the *RECOMMENDED* way described in the gnu Coding Standards.

```
int
foo(int n)
{
  return bar(n);
}
```

`fun.proto.arg` [Rule]

Prototypes *SHOULD* include argument names (in addition to their type).

These are bad style:

```
int foo(int, long);

int bar();
```

These are better:

```
int foo(int x, long y);

int bar(void);
```

`fun.proto.void` [Rule]

You *MUST* specify '`void`' if your function does not take any argument. **Don't** do this:

```
int main()
{
  return 0;
}
```

Instead, write:

```
int main(void)
{
  return 0;
}
```

`c++.fun.proto.void` [Rule]

You *MUST NOT* specify '`void`' if your function or method does not take any argument. The C++ way of doing is to simply write that your function does not take any argument. **Don't** do this:

```
int main(void)
{
  return 0;
}
```

Instead, write:

```
int main()
{
} // In C++, main implicitly returns 0
```

`fun.proto.align` [Rule]

Within a block of prototypes, function names *SHOULD* be aligned.

This is inelegant:

```
unsigned int  strlen(const char *);
char *strdup(const char *);
```

This is elegant:

```
unsigned int strlen(const char *);
char         *strdup(const char *);

/* or */

unsigned int
strlen(const char *);

char
*strdup(const char *);
```

`fun.arg.count` [Rule]
Functions *MUST NOT* take more than four arguments.

`c++.fun.arg.count` [Rule]
Functions *MAY* take any number of arguments, as long it is deemed reasonable.

`fun.arg.qual` [Rule]
Arguments passed by address *SHOULD* be declared '`const`' unless actually modified by the function. Arguments passed by address *SHOULD* be declared '`restrict`' when they point to data that can be accessed only through that pointer.

6.3 Global Scope and Storage

`export.fun` [Rule]
There *MUST* be at most **five** exported functions per source file.

`export.other` [Rule]
There *MUST* be at most **one** non-function exported symbol (such as a global variable) per source file.

`c++.global` [Rule]
You *SHOULD NOT*be using global objects or objects with static storage. Whenever you do, you must be aware that their construction order is subject to the so-called *static initialization order fiasco* (see this link[1]).

`unused.local` [Rule]
There *SHOULD NOT* be any unused local (tagged with '`static`') functions in source files. Hint: hunt unused functions with `gcc -Wunused`.

`file.fun.count` [Rule]
There *MUST NOT* appear more than **ten** functions (exported + local) per source file.

`c++.file.fun.count` [Rule]
There *MAY* be as many functions (exported or local) as necessary per source file. Common practice includes implementing all the methods of a class in the same file.

[1] https://isocpp.org/wiki/faq/ctors#static-init-order.

6.4 Code Density and Documentation

`fun.doc` [Rule]

Function declarations *SHOULD* be preceded by a comment explaining the purpose of the function. This explanatory comment *SHOULD* contain a description of the arguments, the error cases, the return value (if any) and the algorithm realized by the function.

This is recommended:

```
/*
** my_strlen: "strlen" equivalent
**   str: the string
**   return value: the number of characters
** precondition: [str] != NULL
** my_strlen counts the number of characters in [str], not
** counting the final '\0' character.
*/
size_t    my_strlen(const char *str);
```

`fun.innercomment` [Rule]

Function bodies *MAY* contain comments although any useful notice should appear before the function.

`fun.innerblank` [Rule]

Functions bodies *MAY* contain blank lines to make the code more understandable.

`fun.length` [Rule]

Functions' body *MUST NOT* contain more than 25 lines. The enclosing braces are excluded from this count as well as the blank lines and comments.

Rationale: functions should be kept short.

`c++.fun.length` [Rule]

Functions' body *MUST NOT* contain more than 50 lines. The enclosing braces are excluded from this count as well as the blank lines and comments.

`fun.error` [Rule]

Many functions from the C library, as well as some system calls, return status values. Although special cases *MUST* be handled, the handling code *MUST NOT* clobber an algorithm. Therefore, special versions of the library or system calls, containing the error handlers, *SHOULD* be introduced where appropriate.

For example:

```
void *xmalloc(size_t n)
{
  void *p = NULL;

  p = malloc(n);
  if (p == NULL)
  {
    fprintf(stderr, "Virtual memory exhausted.\n");
    exit(1);
  }
  return p;
}
```

7 Project Layout

Specifications in this chapter are to be altered (most often relaxed) by the assistants on a per-project basis. When in doubt, follow the standard.

7.1 Directory Structure

`proj.directory` [Rule]

Each project sources *MUST* be delivered in a directory, the name of which shall be announced in advance by the assistants. In addition to the usual source files, and without additional specification, it *SHOULD* contain a number of additional files:

`configure`

When the project contract allows so, *and only then*, the script `configure` is automatically run before running the `make` command. It *MAY* create or modify files in the current directory or subdirectories, but *MUST NOT* expect to be run from a particular location.

Rationale: allow for site configuration with Autoconf or similar tools.

`Makefile*`

Unless explicitly forbidden, the project directory *MAY* contain an arbitrary number of files with names derived from "Makefile". These files are optional, although a `Makefile` *MUST* be present at the time the command `make` is run.

Rationale: the `Makefile` may include `Makefile-rules.make` or similar files for architecture-dependent compilation options.

`proj.authors` [Rule]

`AUTHORS`

This file *MUST* contain the authors' names, one per line. Each line *MUST* contain an asterisk, then a login name. The first name to appear is considered as the head of the project.

Rationale: this file is to be `grep -E`'ed over with a

```
^\* ([a-z-]+_[a-zA-Z0-9_-]).*$
```

regexp pattern to extract login names.

It is especially important to note that this specification allows for *documenting* a project by using actual text in the `AUTHORS` file: the regexp will only extract the relevant information. For example, consider the following text:

```
This project was written with the help of:

* foo_b (Foo Bar), main developer;
* baz_y (Baz Yay), code consultant;
* chiche_f (Chichery Florent), coffee maker;

Many thanks to them for their contribution to the project.
```

Because the regex only matches the relevant information, it constitutes a valid `AUTHORS` file. Hint: `AUTHORS` file format *MAY* change on a project basis.

7.2 Makefiles and Compilation Rules

`mk` [Rule]

– The input file for the `make` command *MUST* be named `Makefile`, with a capital "M".

Rationale: although the latter name `makefile` is also valid, common usage prefers the former.

– The Makefile rules produced by Autotools by default are always correct. You *MUST NOT* change their behaviour. If something about them contradicts the EPITA Coding Style, follow the Autotools.

Rationale: these rules are a standard.

– To emphasize the previous rule, files automatically distributed by an Autotools `Makefile`'s `dist` rule *SHOULD* be so. For example, `aclocal.m4` *SHOULD* be distributed.

– The `Makefile` (provided or generated by `configure`) *SHOULD* contain the `all`, `clean` and `distclean` rules. The default rule *MUST* be the `all` rule.

`mk.rules.mostlyclean` [Rule]

The `mostlyclean` rule *SHOULD* remove compilation files (such as .o).

`mk.rules.clean` [Rule]

The `clean` rule *MUST* depend on the `mostlyclean` rule, and *SHOULD* remove the targets of compilation (such as executable binairies, shared objects, library archives, .pdf and .html manuals, etc.): this is "opposite" of '`make all`'.

`mk.rules.distclean` [Rule]

The `distclean` rule *MUST* depend on the `clean` rule, and *SHOULD* remove generated files that aren't usually shipped in a release (or handin), such as `Makefile` from `Makefile.am`, etc.: this is the "opposite" of '`configure && make all`'.

`mk.rules.maintainer-clean` [Rule]

The `maintainer-clean` rule *MUST* depend on the `distclean` rule, and *SHOULD* remove all generated files, even when they *MAY* be present in public releases (such as `Makefile.in` that are usually shipped so that the end-users don't depend on Autotools, .c from .y, etc.): this is the "opposite" of '`bootstrap && configure && make all`'.

`mk.rec` [Rule]

– So-called "recursive `Makefile`s" *MAY* be used. That is not considered, however, a good programming practice.

Rationale: The use of "recursive `Makefile`s" may slow compilations down a lot: recursively calling `make` involves non negligible and resource consuming tasks, such as creation of new processes, I/O operations, `Makefile` parsing and dependencies computations. Besides, the dependency tracking will not traverse Makefile boundaries, which may result in incorrect builds.

– When "recursive `Makefile`s" are used, there *SHOULD* be a low amount of redundancy between `Makefile`s.

Hint: Use the `include` directive properly.

Rationale: Code duplication is evil.

`proj.mk.flags` [Rule]

– C sources *MUST* compile without warnings when using strict compilers. The GNU C compiler, when provided with strict warning options, is considered a strict compiler for this purpose.

Especially, when GCC is available as a standard compiler on a system, source code *MUST* compile with GCC and the following options:

```
-Wall -Wextra -std=c99 -pedantic -Werror
```

Additionally, it *SHOULD* compile without warnings with GCC and the following options (all documented in the GCC manual page):

```
-Wall -Wextra -std=c99 -pedantic
-Wfloat-equal -Wundef -Wshadow -Wpointer-arith
-Wbad-function-cast -Wcast-qual -Wcast-align
-Waggregate-return -Wstrict-prototypes -Wmissing-prototypes
-Wmissing-declarations -Wnested-externs
-Wunreachable-code -Wwrite-strings
```

– The previous requirement does *not* imply that the `Makefile` must actually use these flags. It does *not* imply that GCC must be always used: only the command `cc` is guaranteed to be available, and may point to a different compiler.

– C compilation rules *SHOULD* use the warning flag specifiers when possible.

8 Differences Between C99 and C++

This is a non exhaustive list of the differences between the guidelines for C++, relative to those for C99.

- You don't have to and even *MUST NOT* add `s_`, `u_`, `e_`, `t_` before your type names.
- Global variables don't have to be prefixed by `g_`.
- You can write your comments the way you want instead of having to stick to

```
/*
** Comment.
*/
```

- Pointerness is to be expressed as part of the type (`int* p`) despite the (sad) fact that `int* p, q` declares a pointer and an integer. You *MUST NOT* declare two things on the same line anyway.
- Functions can take more than 4 arguments although it is usually not necessary.
- You are not limited to any number of functions per file. A common practice in C++ is to define all the methods of a class in the same file.
- Function bodies *MAY* contain up to 50 (useful) lines maximum instead of 25.

9 Differences with Previous Versions

The following changelog is just here for information's sake, knowledge of this section is **not** mandatory.

9.1 Differences with Year 2014

Four major changes:

- VLA are now forbidden.
- `typedef` keyword should not be used abusively.
- Explicit the C++ naming convention of files.
- Using the directive `#pragma once` is authorized.

9.2 Differences with Year 2007's C99 Coding Style

In 2012, there was only one C coding style, and it had been decided that preparations were needed for a merging of the C, the C++, and the Tiger assignment coding styles. The following changes were made:

- Properly capitalize section names, and add labels to identify parts of the coding style more precisely than the subsections numbering did.
- Reorder this changelog, it was reversed (older was first).
- Remove some weird rationales.
- Fix or improve many examples.
- No longer reference the "EPITA header" of files. This was already moot.
- Indentation with tabulations is now prohibited, whitespaces are mandatory.
- Explicit the interdiction of digraphs and trigraphs, and of preprocessor abuse.
- Explicit the obligation of using doxygen for documentation.
- Change the format of file guards (FOO_H_ is now FOO_H).
- Change the position of the opening braces of `do...while` constructs (the indentation of that construct is now K&R).
- Change `for` construct (`for (;;)` instead of `for (; ; )`, and promote the use of `continue;` over semi-colons alone on their line).
- Change `else if` construct (allow and recommend it on a single line).
- Change `switch` construct (allow it over non-enumeration types, recommend not using `default` over enumerations).
- Variables should now be initalized on declaration (even with non-constant values).
- Lines now break before binary operators (it used to be after).
- Change `return` parenthesis (remove them when the expression is a single line).
- Stop aligning variables, and allow declarations anywhere in the block.

9.3 Differences with Year 2007's C89 Coding Style

9.3.1 Comments

see Section 3.4 [Comment Layout], page 8

- This was changed :

  - There *SHOULD NOT* be any single-line comment, excluding the case of comments inside the functions' body.

    Rationale: if the comment is short, then the code should have been self-explanatory in the first place.

- Into :
  - Single-line comments *MAY* be used, even outside the functions' body.

    Rationale: C99 introduces C++ like comments with //

- Two new examples have been added

9.3.2 Variable Declarations

see Section 4.2 [Structures Variables and Declarations], page 12

- This has been changed :
  - Declared identifiers *MUST* be aligned with the function name, using tabulations *only*.
- Into :
  - Declared identifiers *MUST* be aligned with the function name, using tabulations *only*, even for identifiers declared within a block.
- The example has been changed.
- This has been added :
  - When initializing a local structure (a C99 feature), the initializer value *MUST* start on the line after the declaration:
- An example has been added.

9.3.3 For Construct

see Section 4.5 [Control Structures], page 16

- This has been changed :
  - Variables *MUST NOT* be declared in the initial part of the '`for`' construct.
- Into :
  - Variables *MAY* be declared in the initial part of the '`for`' construct.
- An example has been added.

9.3.4 Prototypes

see Section 6.2 [Functions and Prototyping], page 26

- This has been changed :
  - Prototypes *MUST* conform to the ANSI C standard: they must specify **both** the return type and the argument types.
- Into :
  - Prototypes *MUST* conform to the C99 standard: they must specify **both** the return type and the argument types.
- This has been added :
  - Function arguments passed by address *SHOULD* be declared '`restrict`' when they point to data that can be accessed only through that pointer.

9.3.5 Compilation Flags

see Chapter 7 [Project Layout], page 30

- This has been changed :

```
-Wall -Wextra -ansi -pedantic -Werror
```

- into

```
-Wall -Wextra -std=c99 -pedantic -Werror
```

9.4 Differences with Year 2006

The following changes were introduced during the year 2007.

Here is a summary of the major changes.

- When creating a '`typedef`' from a type that is already an EPITA-style '`typedef`', the prefix of the type must be preserved.

```
typedef struct foo      s_foo;
typedef s_foo           *s_foo_ptr;
```

Other minor changes this year include:

- The rule introduced last year which required that function arguments to be spanned over multiple lines (one argument per line) was cancelled.
- The EPITA-style header is no longer mandatory because it generates spurious conflicts with Source Control Management softwares and provides little value added.

9.5 Differences with Year 2005

The following changes were introduced during the year 2006.

Here is a summary of the major changes. These modifications were introduced to make the whole coding style standard more coherent.

- Functions' body can contain comments and blank lines to make the code more understandable and clearer. These additional lines are not counted in the function's body 25 lines limit.

```
int foo(char *s)
{
  int length;

  /* sanity check */
  if (s == NULL)
    return (-1);

  /* do the job */
  length = strlen(s);

  return (length);
}
```

- Function argument lists must be split between each argument, after the comma and the arguments must be properly aligned.

```
int        open(const char *pathname,
int        flags);

void        bar(char        *s,
int         flags)
```

```
{
  int   fd;

  if ((fd = open(s, flags)) == -1)
    return;

  /* ... */
}
```

- When the '`return`' statement contains an argument, this argument must be enclosed in parenthesis.
- Structures, unions and enumerations should be aliased using typedefs and using specific prefixes: '`s_`' for structures, '`u_`' for unions, '`e_`' for enumerations and '`f_`' for function pointers.

```
union                   cast
{
  char                  c;
  short                 s;
  int                   i;
  long                  l;
};

typedef union cast      u_cast;

int         main(int         argc,
                 char        **argv)
{
  union cast    cast1;
  u_cast        cast2;

  /* ... */
}
```

- Global variable names must be prefixed by '`g_`'.
- Enumerations values must be capitalized.
- Enumeration value alignment with the enumeration name is no longer mandatory.

9.6 Differences with Year 2004

No significant changes: minor corrections to some examples.

9.7 Differences with Year 2003

Starting from the year 2004, this english version is the official specification document used.

The major changes are:

- Expressions tolerated at initialisation are clearly specified.
- English comments are now mandatory.
- Comments for functions are on declarations and not definitions.
- There can be only '`static const`' variable, no more static variables are tolerated.
- Auto-generated headers can now have more than 80 columns.
- Structure and unions must be returned and passed by address.

9.8 Differences with Year 2002

Starting with 2003, the assistants decided to revert to a short specification written in French, for readability convenience.

The English document you are now reading was a complement that could *optionally* used as a substitute.

Here is a summary of the major changes:

- Names are required to match a regular expression.
- Preprocessor directives indentation between '`#if`' and '`#endif`' is now mandatory.
- Header protection tags now have a '`_`' appended.
- Multiple declarations per line are now forbidden, due to abuses during the past year.
- Cumulated declaration and initialization is now explicitly authorized.
- Local external declarations ('`extern`' in block scope) are now implicitly authorized.
- Statement keywords without argument are not followed by a white space anymore.
- '`else if`' cannot appear on a single line any more.
- Single-line empty loops are now forbidden (the trailing semicolon must appear on the following line).
- Return-by-value of structures and unions is now implicitly authorized.
- '`typedef`' of structures and unions is now disallowed.
- Line count for function bodies is now absolute again (empty lines, '`assert`' calls and '`switch`' cases are counted).
- Project recommendations now insist on the fact that GCC must not always be used and that the '`configure`' script is not always allowed.
- Sample '`Makefile`' and '`configure`' scripts are not provided anymore.

9.9 Differences with the Legacy Version

The 2002 document was intended to supersede the legacy '`norme`', first written in (??), and last updated in October, 2000.

It was based on the previous version, adding finer distinctions between requirements and recommendations, updating previous specifications and adding new ones.

Here is a summary of the major changes:

- Specification of the differences between '`requirements`' and '`recommendations`' was added.
- Indentation requirements were clarified.
- Header file specifications were clarified and updated to match modern conventions.
- The '`switch`' construct is now allowed under special circumstances.
- Prototyping specifications were clarified and detailed.
- Naming conventions were clarified.
- Declaration conventions were clarified and relaxed for some useful cases.
- Line counting of function bodies was relaxed. The limit on the number of function arguments was explained and relaxed.
- Comment specifications, including standard file headers, were clarified and detailed.
- Project layout specifications were added. Default '`Makefile`' rules and rule behaviors were updated to match modern conventions.
- Special specifications for C++ were added.

In addition to these changes, the structure of the standard itself has been rearranged, and an index was added.

Index and Table of Contents

I

K

L

M

N

O

P

R

S

T

U

Table of Contents