EPITA Coding Style Standard

Edition 19 September 2003

C/Unix Assistants *et al.*

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

Covered topics:

- Naming conventions
- 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.

During the initial period, all submitted projects must comply **exactly** with the standard; any infringement causes the mark to be multiplied by 0.

This standard is usually relaxed during the second period (starting in january), mainly because of the evolution of project requirements: the use of Automake leverage constraints over `Makefile`'s, and the use of languages other than C imply their own, different, coding styles. However, this **does not** mean that introducing new tools or language requirements in project during the first period automatically relaxes the standard: this has to be negociated on a per-case basis with the assistants.

Note that this document is complementary to the official document, which is written in french and is available on the assistants web site.

1 How to read this document

This documents 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 incompatibilites.

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.

1.4 Correlation with the authoritative document

The authoritative document, called “norme” and written in french, only covers a subset of the mandatory items (denoted by *MUST*, *SHALL*, *REQUIRED*). Compliance to that document is checked thoroughly and strictly.

This document is more general. It expresses rules of thumbs and guidelines for elegance in C writing. It was extensively checked for compliance with the authoritative “norme”, and can therefore be used as a substitute.

2 Naming conventions

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

2.1 General naming conventions

– 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
```

–

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
...
```

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

– Names *MUST* be expressed in english.

– Names *SHOULD* be expressed in correct english, i.e. without spelling mistakes.

2.2 Name capitalization

– 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-z.A-Z0-9]
```

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

–

C macro names *MUST* be entirely capitalized.

– C macro arguments *MUST* be capitalized:

```
#define XFREE(Var)       \
  do                     \
  {                      \
    if (Var)             \
      free(Var);         \
  }                      \
  while (0)
```

2.3 Name prefixes

– When declaring types, type names *MUST* be prefixed according to the group they belong to: structure names *MUST* start with '`s_`', typedefs with '`t_`', union names with '`u_`', enumeration names with '`e_`'. Beware, the prefix is *not* part of the identifier, thus "anonymous typedefs" of the form `typedef int t_;` are *PROHIBITED*.

```
typedef unsigned char           t_cmap[COLOR_WIDTH * NCOLORS];
typedef unsigned char           t_pixel;

struct                          s_picture
{
  int                           width;
  int                           height;
  t_cmap                        cmap;
  t_pixel                       *picture;
};
```

Rationale: for not using suffixes instead: identifiers ending with '`_t`' are reserved by POSIX (beside others).

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.

- Structure and union names *MUST NOT* be aliased using '`typedef`'. It is therefore not correct to defined shortcut names to structures and unions prefixed with '`t_`'.

  Rationale: '`typedef`' hides the compound nature of structures and unions.

-

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

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

– Lines *MUST NOT* exceed 80 characters in width, including the trailing newline character.
– The DOS CR+LF line terminator *MUST NOT* be used. Hint: do not use DOS or Windows standard text editors.
– All source and header files *MUST* start with a file header, which *MUST* specify the file name, the project name, an optional location, the author's name and login name, and the creation and last modification timestamps.
– File headers *MUST* comply to the following template:

```
/*
** <filename> for <project> in <location>
**
** Made by <author>
** Login   <login name@site>
**
** Started on  <date> <author>
** Last update <date> <author>
*/
```

  Hint: this layout can be obtained at EPITA with `C-c C-h` in Emacs.
– When instantiating the previous template, the '`for <project> ...`' part *MUST NOT* be omitted. Hint: if the '`<project>`' field is irrelevant (i.e. the file is independent), fill it with "self".
– In order to disable large amounts of code, you *SHOULD NOT* use comments. Use '`#if 0`' and '`#endif`' instead.

  Rationale: C comments do not nest.
– Delivered project sources *SHOULD NOT* contain disabled code blocks.

3.2 Preprocessor directives layout

– The preprocessor directive mark ('`#`') *MUST* appear on the first column.
– 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 initial condition.
– When a directive must span over multiple lines, escaped line breaks ('\'-newline) *MUST* appear on the same column. For this purposes, tabulations *MUST* be used.

This is wrong:

```
#define XFREE(Var) \
  do \
  { \
    if (Var) \
      free(Var); \
  } \
  while (0)
```

This is correct:

```
#define XFREE(Var)          \
  do                        \
  {                         \
    if (Var)                \
      free(Var);            \
  }                         \
  while (0)
```

Hint: use *C-\* and *C-u C-\*, or *M-i* under Emacs.

3.3 Macros and code sanity

- C macro names *MUST* be entirely capitalized (see Section 2.2 [Name capitalization], page 4).
- As a general rule, preprocessor macro calls *SHOULD NOT* break code structure. Further specification of this point is given below.
- 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 MY_CASE(Name)        \
 case d_ # Name:             \
   return go_to_ # Name();

[...]

switch (direction)
{
   MY_CASE(left)
   MY_CASE(right)
   default:
     break;
}
```

  This is more elegant:

```
#define MY_CASE(Action)    \
  return go_to_ # Action();

[...]
switch (direction)
{
   case d_left:
     MY_CASE(left);
     break;
   case d_right:
     MY_CASE(right);
     break;
   default:
     break;
}
```

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

3.4 Comment layout

- Comments *SHOULD* be written in the english language.
- Comments *SHOULD NOT* contain spelling errors, whatever language they are written in. However, omitting comments is no substitute for poor spelling abilities.
- There *SHOULD NOT* be any single-line comment.

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

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

```
/*
 * Incorrect
 */

/* Incorrect
 */
```

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

For additional specifications about comments, see Section 3.1 [File layout], page 6 and Chapter 5 [Global specifications], page 17.

3.5 Header files and header inclusion

– Header files *MUST* be protected against multiple inclusion. The protection "key" *MUST* be the name of the file, entirely capitalized, which punctuation replaced with underscores, and an additional underscore appended. For example, if the file name is `foo.h`, the protection key *SHALL* be '`FOO_H_`':

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

– 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 "bar.h"

int foo();

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

This is elegant:

```
#ifndef FOO_H_
# define FOO_H_

# include <stdio.h>
# include "bar.h"

int bar();
int foo();

#endif /* !FOO_H_ */
```

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

– All braces *MUST* be on their own lines.

  This is wrong:

```
if (x == 3) {
  x += 4;
}
```

  This is correct:

```
if (x == 3)
{
  x += 4;
}
```

– Closing braces *MUST* appear on the same column as the corresponding opening brace.
– The text between two braces *MUST* be indented by a fixed, homogeneous amount of whitespace. This amount *SHOULD* be 2 or 4 spaces.
– Opening braces *SHOULD* appear on the same column as the text before. However, they *MAY* be shifted with a fixed offset after control structures, in which case the closing brace *MUST* be shifted with the same offset.

  These are wrong:

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

if (x == 3)
{
  foo3();
  {
    inner();
   }
  }

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

  These are correct:

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

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

– In C functions, the declaration part *MUST* be separated from statements with one blank line. Note that when there are no declarations, there *MUST NOT* be any blank line within a block.

  An example is provided in the following section.

4.2 Structures variables and declarations

4.2.1 Alignment

– Declared identifiers *MUST* be aligned with the function name, using tabulations *only*. Hint: Emacs users, use *M-I*.

  The following is wrong:

```
int foo()
{
  int i = 0;
  return i;
}
```

  The following is correct:

```
int     foo()
{
  int   i = 0;

  return i;
}
```

– In C, pointerness is not part of the type. Therefore, 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):

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

  The following is correct:

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

– Structure and union fields *MUST* be aligned with the type name, using tabulations.

– When declaring a structure or an union, there *MUST* be only **one** field declaration per line.

  This is incorrect:

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

  This is correct:

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

– Enumeration values *SHOULD* be capitalized or reasonably prefixed.

  Rationale: the use of common lowercase identifiers is discouraged because it clobbers the namespace.

– Enumeration values *MUST* appear on their own lines, properly aligned with the name of the enumeration.

  This is incorrect:

```
enum e_boolean
{ true, false };
```

  This is correct:

```
enum    e_boolean
{
        b_true,
        b_false
};
```

4.2.2 Declarations

– There *MUST* be only one declaration per line.

– Inner declarations (i.e. at the start of inner blocks) are *RECOMMENDED* when they can help improve compiler optimizations.
– Declaration blocks in functions *SHOULD NOT* contain '`extern`' declarations.
– Variables *MAY* be initialized at the point of declarations. However, for this purpose function and macro calls and composite expressions *MUST NOT* be used.
  The following is wrong:

```
int    foo = strlen("bar");
char   c = (str++, *str);
```

  This is correct:

```
unsigned int    *foo = &bar;
unsigned int    baz = 1;
static int      yay = -1;
```

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

4.3 Statements

– A single ligne *MUST NOT* contain more than one statement.
  This is wrong:

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

  This is correct:

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

– Commas *MUST NOT* be used on a line to separate statements.
– 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.
– The semicolon *MUST* be followed by a newline.
– For a detailed review of exceptions to the three previous rules, See Section 4.5 [Control structures], page 12.
– Statements keywords *MUST* be followed by a single whitespace, *except* those without arguments. This especially implies that '`return`' without argument, like '`continue`' and '`break`', *MUST NOT* be separated from the following semicolon by a whitespace.
– When the '`return`' statement takes an argument, this argument *MUST NOT* be enclosed in parenthesis.
  This is wrong:

```
return (0);
```

  This is correct:

```
return 0;
```

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

4.4 Expressions

– 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.
- The '`.`' and '`->`' operators *MUST NOT* be padded, neither.

  This is wrong:

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

  This is correct:

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

- There *MUST NOT* be any whitespace between the function and the opening parenthesis for arguments in function calls.
- "Functional" keywords *MUST* be followed by a whitespace, and their argument(s) *MUST* be 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 char);
  ```

  This is correct:

  ```
  p = malloc(3 * sizeof (int));
  ```

- Expressions *MAY* span over multiple lines. When a line break occurs within an expression, it *MUST* appear just after a binary operator, in which case the binary operator *MUST NOT* be padded on the right by a whitespace.

4.5 Control structures

4.5.1 General rules

- Control structure keywords *MUST* be followed by a whitespace.

  This is wrong:

  ```
  if(x == 3)
    foo3();
  ```

  This is correct:

  ```
  if (x == 3)
    foo3();
  ```

- The conditional parts of algorithmic constructs ('`if`', '`while`', '`do`', '`for`'), 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();
}

do {
  ++x;
} while (x < 10);
```

These are correct:

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

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

do
{
  ++x;
}
while (x < 10);
```

4.5.2 ‘while’ and ‘do ... while’

– The ‘`do ... while`’ construct *MAY* be used, but appropriate use of the ‘`while`’ and ‘`for`’ constructs is preferred.

4.5.3 ‘for’

Exceptions to other specifications (See Section 4.3 [Statements], page 11, see Section 4.2 [Structures variables and declarations], page 10) can be found in this section.

– Multiple statements *MAY* appear in the initial and iteration part of the ‘`for`’ structure.
– For this effect, commas *MAY* be used to separate statements.
– Variables *MUST NOT* be declared in the initial part of the ‘`for`’ construct.

This is wrong:

```
for (int i = 0, j = 1;
     p = i + j, p < 10;
     ++i, ++j)
{
  /* ... */
}
```

This is correct:

```
int   i;

for (i = 0, j = 1, p = i + j;
     p < 10;
     ++i, ++j, p = i + j)
{
  /* ... */
}
```

– As shown in the previous examples, the three parts of the ‘`for`’ construct *MAY* span over multiple lines.
– 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.

These are wrong:

```
for (;;) ;

for ( ; ; ) ;
```

This is correct:

```
for (; ; )
  ;
```

4.5.4 Loops, general rules

– To emphasize the previous rules, single-line loops (‘`for`’ and ‘`while`’) *MUST* have their terminating semicolon on the following line.

  This is wrong:

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

  These are correct:

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

  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

– The ‘`switch`’ *MUST* be used **only** over enumeration types.

– Incomplete ‘`switch`’ constructs (that is, which do not cover all cases of an enumeration), *MUST* contain a ‘`default`’ case.

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

– Control structure *MUST NOT* span over several ‘`case`’ blocks.

  This is very wrong:

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

– Each ‘`case`’ conditional *MUST* be indented from the associated ‘`switch`’ once, 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;
  }
```

– 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 (direction)
{
  case d_left:
    return go_to_left();
    break;
  case d_right:
    return go_to_right();
  case d_down:
    printf("Wrong\n");
    break;
  default:
    break;
}
return do_it();
```

This is elegant:

```
switch (direction)
{
  case d_left:
    return go_to_left();
  case d_right:
    return go_to_right();
  case d_down:
    printf("Wrong\n");
    return do_it();
  case d_up:
    return do_it();
}
```

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

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

4.6 Trailing whitespace

– 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.
– There *SHOULD NOT* be any empty lines at the end of a source file. Emacs users should be careful not to let Emacs add blank lines automatically.
– When it is not a requirement, contiguous whitespace *SHOULD* be merged with tabulation marks, assuming 8-space wide tabulations.
– (Reminder, see Section 3.1 [File layout], page 6) The DOS CR+LF line terminator *MUST NOT* be used. Hint: do not use DOS or Windows standard text editors.

5 Global specifications

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

5.1 Casts

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.

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.

5.2 Functions and prototyping

- Any exported function *MUST* be properly prototyped.
- 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:

  File `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_ */
```

  File `my_strlen.c`:

```
#include "my_string.h"

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

  File `my_strdup.c`:

```
#include "my_string.h"

char *my_strdup(const char *s)
{
/* definition of my_strdup */
}
```

- Prototypes *MUST* conform to the ANSI C standard: they must specify **both** the return type and the argument types.

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

  These are invalid prototypes:

```
foo();

bar(int, long);

int baz();
```

  These are valid prototypes:

```
int  foo(int x);

void bar(int x, long y);

int  baz(void);
```

– 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 *);
```

– Function names in prototypes *SHOULD* be aligned with other declarations.

  This is not recommended:

```
int                    gl_counter;

struct s_block *allocate(unsigned int size);
void    release(struct s_block *);
```

  This is recommended:

```
int                    gl_counter;

struct s_block         *allocate(unsigned int size);
void                   release(struct s_block *);
```

– Function argument lists *MAY* be broken between each argument, after the comma. When doing so, the arguments *MUST* be properly aligned.

  This is correct:

```
type foo(type1 p1, type2 p2);
```

  This is also correct:

```
type bar(type1 p1,
         type2 p2,
         type3 p3);
```

– Functions *MUST NOT* take more than 4 arguments.

  Rationale: the C ABI on Unix specifies that the first 4 function arguments are always passed in registers, which yields more performance. In addition, if more arguments are needed, usually the modelling of the project is wrong.

– An argument name *MAY* be omitted at the point of *definition* of a function, in the special case where it is not used by the function.

Rationale: by omitting an argument name, the compiler warning saying that it is unused is inhibited.

- Functions *SHOULD NOT* return structures or unions by value. Structures or unions *SHOULD NOT* be passed by value as function arguments, either. The use of dynamic memory management is encouraged instead.
- Function arguments passed by reference *SHOULD* be declared '`const`' unless actually modified by the function.

5.3 Global scope and storage

- There *MUST* be at most **five** exported functions per source file.
- There *SHOULD* be only **one** non-function exported symbol per source file.

  Rationale: when statically linking executables against libraries, most linker algorithms operate with object file granularity, not symbol granularity. With only one exported symbol per source file, the link process has the finest granularity. Hint: track exported symbols with `nm`.
- There *SHOULD NOT* be any unused local (tagged with '`static`') functions in source files. Hint: hunt unused functions with `gcc -Wunused`.
- In order to block known abuses of the previous rules, there *MUST NOT* appear more than *ten* functions (exported + local) per source file.
- Static declarations are *NOT RECOMMENDED*. When required by a particular circumstance, there *MUST* be **only one** static variable per line.
- When initializing a static array or structure with const elements, the initializer value *MUST* start on the line after the declaration:

  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 }
};
```

5.4 Code density and documentation

- (Reminder, see Section 3.1 [File layout], page 6) Lines *MUST NOT* exceed 80 characters in width, including the trailing newline character.

– Function definitions *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
** my_strlen counts the number of characters in [str], not
** counting the final '\0' character.
*/
size_t    my_strlen(const char *str);
{
   /* definition of my_strlen */
}
```

– Function bodies *MUST NOT* contain comments. Any useful notice should appear before the function.

– There *MUST NOT* be any blank line elsewhere than between declarations and statements within a function body.

– Function bodies *MUST NOT* contain more than 25 lines, enclosing braces excluded.

  Rationale: function bodies should be kept short.

– 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;

  p = malloc(n);
  if (p == 0)
  {
    fprintf(stderr, "Virtual memory exhausted.\n");
    exit(1);
  }
  return p;
}
```

6 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.

6.1 Directory structure

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

'`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`'ed over with a

```
^\* \([a-z-][a-z-]*_[a-z]\).*$
```

regexp pattern to extract login names.

It is especially important to note that this specifications 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;
* franco_l (Ludovic François), tutor

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

Because the regex only matches the relevant information, it constitutes a valid '`AUTHORS`' file.

'`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 explicitely 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.

6.2 Makefiles and compilation rules

– 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 prefer the former.

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

- The 'Makefile' *MUST NOT* use non-standard syntax. In particular, it *MUST NOT* expect to be parsed by GNU make ("gmake").
- The default rule *MUST* be the `all` rule.
- The `clean` rule *SHOULD* clear object files, temporaries and automatic editor backups from the source tree.
- The `distclean` rule *MUST* depend on the `clean` rule, and *SHOULD* clear executables, shared objects and library archives from the source tree.
- The use of so-called "recursive 'Makefile's" is discouraged; when used, the amount of redundancy between 'Makefile's *SHOULD* be kept low by proper use of `include` directives.
- 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 -W -ansi -Werror
  ```

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

  ```
  -Wall -W -ansi -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
  ```
- 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.
- 'Makefile' rules *MUST NOT* expect the presence of GCC on all target architectures.
- As a side effect of the two previous rules, compiler differences and architecture-dependent flags *MUST* be handled by appropriate use of the `uname` command. In particular, the environment variable `HOSTTYPE` *MUST NOT* be used for this purpose, since it has a shell-dependant and architecture-dependent behaviour.

7 Differences with previous versions

7.1 Differences with the legacy version

The 2002 document was intended to supercede 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 difference 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.

7.2 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 is now a complement that can *optionnally* 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 explicitely authorized.
- Local external declarations (‘`extern`’ in block scope) are now implicitely 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 traling semicolon must appear on the following line).
- Return-by-value of structures and unions is now implicitely 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.

Index and Table of Contents

Table of Contents