EPITA C++ Coding Style Guidelines

Edition 21 February 2007

EPITA Assistants

This document intends to establish a common coding style for the C++ projects of the EPITA students.

Covered topics:

- Naming conventions
- Lexical layout (block level)
- Object Oriented (OO) consideration.
- Global layout (source file level), including header files and file headers
- Project layout

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

All submitted projects must comply **exactly** with these rules.

WARNING: Despite the fact that this document looks pretty much like that of the C Coding Style, it is **different** and specifies **different rules**. Read it carefully.

Some sections of this document are identical to the corresponding ones of the C Coding Style. In this case, they will not be repeated and you are asked to refer to the C Coding Style guidelines.

The rules specified in this document give you much more freedom than the C Coding Style one, use this freedom wisely.

The Tiger Compiler Assignment also has a section about Coding Style, you *MUST* follow it for TC. If the Tiger Assignment contradicts this document, then it is authoritative for the Tiger Compiler.

If you are a beginner in C++, you are **strongly suggested** to entirely read The C++ FAQ light of comp.lang.c++.

1 How to read this document

This document adopts some conventions described in the following nodes.

1.1 Vocabulary

These guidelines use 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 understood 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 mix up the intention and extension of this document.

The intention is to limit obfuscation abilities of certain students with prior C++ experience, and make uniform 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 coding style 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 the examples

Examples in these guidelines 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.

2 Naming conventions

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

2.1 General naming conventions

The recommendations provided by the C Coding Style guidelines also apply in C++: give your {files,variables,classes,namespaces,etc} relevant (English) names.

Additionally, the following rules apply:

- 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).
- You *MUST* name your classes `LikeThis`.

  Class should be named in mixed case; for instance `Exp`, `StringExp`, `TempMap`, `InterferenceGraph` etc. This also applies to class templates.
- 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_`. 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. We used to write `_like_this`, but this goes against the standard (see [Stay out of reserved names], page 3). For instance, write:

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

- You *MUST* declare one class `LikeThis` per files '`like-this.*`'. Each class `LikeThis` is implemented in a single set of file named '`like-this.*`'. 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.
- You *MUST* name your namespaces `likethis`.

This section is incomplete and you must follow the guidelines specified in the section Name Conventions of Tiger Assignment.

2.2 Name capitalization

Preprocessor macro names *MUST* be entirely capitalized.

2.3 Typedef suffix

When declaring types, type names *MUST* be suffixed with `_type`

```
typedef typename MyTraits<T>::res value_type;
typedef std::map<const Symbol, Entry_T> map_type;
typedef std::list<map_type> symtab_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 (beside others).

3 Preprocessor-level specifications

The global layout of files, and sections of code pertaining to the C preprocessor (which happens to be used in C++), 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 text editors.
- 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 *MUST 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`' *SHOULD* be followed by a comment describing the corresponding initial condition.
- The pre-processor *MUST NOT* be used to split a token. For instance, doing this:

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

  is strongly forbidden.
- Trigraphs and digraphs *MUST NOT* be used.

3.3 Macros and code sanity

- Preprocessor macro names *MUST* be entirely capitalized.
- 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.

  Rationale: macros should not allow for hidden syntactic "effects".
- The code inside a macro definition *MUST* follow the specifications of these guidelines as a whole.

– The code inside a macro definition *MUST NOT* contain unbalanced parentheses or unbalanced braces.
– The code inside a macro definition *MUST NOT* break the control flow using `break`, `continue`, `return` or `throw`.

3.4 Comment layout

- Comments *MUST* be written in English.
- You *SHOULD* prefer C-style (‘`/** ... */`’) comments for long comments and C++/C99-style (‘`/// ...`’) comments for one-line comments.
- You *MUST NOT* document the obvious.

  Don’t write:

  ```
  /// Declaration of the Foo class.
  class Foo
  {
    ...
  };
  ```

  It is so obvious that you’re documenting the class and the constructor that you should not write it down. Instead of documenting the *kind* of an entity (class, function, namespace, destructor...), document its *goal*.
- 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. For instance, instead of:

  ```
  /// \brief Swap the reference with another.
  /// The method swaps the two references and returns the first.
  ref& swap (ref& other);
  ```

  write:

  ```
  /// @brief Swap the reference with another.
  /// Swap the two references and return the first.
  ref& swap (ref& other);
  ```

  The same rules apply to ChangeLogs.
- You *SHOULD* write your documentation in Doxygen.
- You *MUST* document your extern functions, your class and their methods, in the header that declares them, not in the file that implement them.

  Rationale: The clients of your interfaces will read your headers to know how your library or module works. They don’t want to dig in the implementation to find the documentation.

  Rationale: The code is much more read that written and you should favor the (many) readers instead of the (often only) implementer.

  Rationale: Doing so leaves you some room to comment your implementation.

3.5 Header files and header inclusion

– Header files *MUST* be protected against multiple inclusions.
– Inclusion of system headers *SHOULD* precede inclusion of local headers.
– There *MUST NOT* be any `using namespace` in a header.
– You *MUST* document classes in their ‘`*.hh`’ file.

4 Writing style

The following sections specify various aspects of what constitutes good programming behavior at the lexical level.

4.1 Blocks

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

This is wrong:

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

This is correct:

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

– As an exception, you are allowed to add a comment after a closing brace in order to specify what is getting closed.

```
namespace foo
{
  // FIXME: Add 500 lines here.
} // namespace foo
```

But *SHOULD NOT* state the obvious.

– As another exception, you are allowed to 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;
}
```

– 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 an indentation level after control structures, in which case the closing brace *MUST* be shifted with the same offset.

4.2 Declarations

4.2.1 Alignment

Pointers and references are part of the type, and *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();’
```

4.2.2 Declarations

- There *MUST* be only one declaration per line.
- Inner declarations (i.e. at the start of inner blocks) are *RECOMMENDED*. Variables *SHOULD* be declared close to their first use and their scope must be kept as small as possible.
- Variables *SHOULD* be initialized at the point of declarations.

  Hint: Your compiler can help you to detect uninitialized local variables.

4.3 Statements

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

  This is wrong:
  ```
  x = 3; y = 4;
  ```
  This is correct:
  ```
  x = 3;
  y = 4;
  ```

- Commas *MUST NOT* be used on a line to separate statements.
- The commas *MUST* be followed by a space character.
- The semicolons *MUST* be followed by a newline, and *MUST NOT* be preceded by a whitespace, except if alone on the line.
- Keywords *MUST* be followed by a single whitespace, *except* those without arguments.
- The `goto` statement *MUST NOT* be used.

  Of course, this rule is not applicable on source files generated by external tools (like `bison` or `flex`).
- The `asm` declaration *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 `.`, `->`, `::`, `operator[]` and `operator()` 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;
  ```

- In C++ you *SHOULD* write `0` instead of `NULL` for NULL pointers.
- There *MUST* be a space between the function name arguments.

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

– 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.

```
std::cout << "Hello!" << std::endl
  << "World!\n" << std::endl;
```

4.5 Control structures

4.5.1 General rules

– The following keywords *MUST* be followed by a whitespace:

```
do if typeid sizeof case catch switch template for throw while try
```

This is wrong:

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

This is correct:

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

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

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

– 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 (;;)
  ;
```

4.5.2 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)
  ;
```

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.

– While it is not a requirement, contiguous whitespace *MAY* be merged with tabulation marks, assuming 8-space wide tabulations.

– (Reminder, see Section 3.1 [File layout], page 5) The DOS CR+LF line terminator *MUST NOT* be used. Hint: do not use DOS or Windows text editors.

5 Object Oriented considerations

5.1 Lexical Rules

- You *MUST* order class members by visibility first.

  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.

- 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;
};
```

- 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 ();
};
```

- 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, and of course between two closing '>':

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

  These rules apply for casts:

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

- 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>
{ ... };
```

- You *MUST* leave a space between function name and arguments. The `()` operator is not a list of arguments.

```
class Foo
{
  public:
    Foo ();
    virtual ~Foo ();
    bool operator() (int n);
};
```

- 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;
}
```

  Write this instead (without the comments, obviously):

```
int
main () // Not like C: main can't be passed any argument
{
} // In C++, main implicitly returns 0
```

- 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);
}
```

- 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

- Read The C++ FAQ light of comp.lang.c++. This is by far the most useful document you can read while learning C++.
- Trust your compiler. Let it be your friend. Use the warnings: `-Wall -W`.
- Read what Tiger Assignment says about the use of STL.
- You *SHOULD* declare your constructors with one argument `explicit` unless you know you want them to trigger implicit conversions.
- 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)
{
}
```

- When a constructor fails, it *MUST* throw an exception instead of leaving the newly created object in a zombie state (see this link).
- A destructor *MUST NOT* fail, that is, it *MUST NOT* throw an exception (see this link).
- Your destructor *MUST* be `virtual` if there is at least one virtual method in your class (see When should my destructor be virtual?).
- You *MUST NOT* overload `operator,`, `operator||` and `operator&&`.
- You *SHOULD NOT* overload `operator&`.

- You *MUST NOT* use `friend`s to circumvent bad design.
- 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.
- 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 ());
}
```

- You *SHOULD* catch by reference.
- Once again, you *SHOULD* **really** read The C++ FAQ light of comp.lang.c++. This is by far the most useful document you can read while learning C++.

6 Global specifications

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

6.1 Casts

- As a general rule, C casts *MUST NOT* be used.

  Use the C++ casts instead: `dynamic_cast`, `static_cast`, `const_cast`, `reinterpret_cast`.

  Rationale: good programming behavior includes proper type handling.
- Use of `reinterpret_cast` is *NOT RECOMMENDED*.

6.2 Global scope and storage

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).

6.3 Code density and documentation

– (Reminder, see Section 3.1 [File layout], page 5) Lines *MUST NOT* exceed 80 characters in width, including the trailing newline character.

– Function declarations *SHOULD* be preceded by a Doxygen 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 implemented by the function.

– Function bodies *MAY* contain comments and blank lines.

– 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.

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 these guidelines.

7.1 Directory structure

This section is identical to the corresponding one described in the C Coding Style.

Note, however, that many people tend to use `make tests` instead of `make check` or to put their tests in a directory named `test` or `check` whereas it *SHOULD* be named `tests` in order to comply with the GNU Coding Standards.

8 Differences with the C Coding Style

This is a non exhaustive list of the differences between the guidelines provided by this document and that provided by the C Coding Style.

- 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 don't have to and even *SHOULD NOT* add the EPITA header to all your files. It is deprecated and useless. SVN does a better and more useful job if you want to track modifications.
- You can write your comments the way you want instead of having to stick to

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

- You don't have to and even *SHOULD NOT* align everything (declarations, arguments, local variable names, member fields in structures, enums or unions, etc).
- 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.
- When declaring **and** initializing your variable you can initialize them the way you want, not only with simple expressions as required by the C Coding Style.
- The `return` value doesn't have to be in parentheses. Actually, it *SHOULD NOT* unless adding parentheses makes it easier to read (e.g, because it spans over more than one line).
- Almost all parentheses must be preceded by a whitespace.
- When breaking a long expression so that it spans over more than one line, the break must occur **before** a binary operator, not after. This is what the GNU Coding Standards suggest (and justifies).
- You are allowed and advised to write `else if` on the same line.
- 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.

Index and Table of Contents

P

R

S

T

U

V

W

Table of Contents