1 C++ coding style guide 3
  1.1 Formatting ......................................................... 3
  1.2 Naming .......................................................... 5
  1.3 Error handling .................................................. 6
  1.4 Misc ............................................................. 7
  1.5 TODO ............................................................. 7

2 Makefile style guide 9
  2.1 Naming .......................................................... 9
  2.2 Special targets .................................................. 9

3 CMake coding style guide 11
  3.1 Naming .......................................................... 11

Bibliography 13
This repository contains my personal misc coding conventions.
C++ coding style guide

Why yet another C++ coding style? Because there is no standard style. Each company, organization has its own. Most of them don’t satisfy me. They are similar to java coding style. Personally, I want to stay close to Bjarne Stroustrup’s and STL, Boost coding styles. Why? Because this looks more like C++ and code becomes more consistent when it integrates nicer with standard libraries.

1.1 Formatting

1.1.1 Lines

Maximum of 80 characters should be used on a single line. Why?:

- Humans read narrower columns faster.
- [http://www.emacswiki.org/emacs/EightyColumnRule](http://www.emacswiki.org/emacs/EightyColumnRule)

1.1.2 Indentation

Tabs. Tab size is 8 spaces.

1.1.3 Namespaces

```cpp
namespace log {
...
}
```

1.1.4 Classes

```cpp
class employee : public person {
public:
    employee(const std::string& name, const std::string& profession);
protected:
    ...
```
private:
...

1.1.5 Enums

```cpp
eenum severity_level {
    debug, info, warning, error
}
```

Or

```cpp
eenum severity_level {
    debug, // Most verbose logs.
    info, // General info logs.
    warning, // Logs that need attention.
    error // Something unexpected happened.
}
```

1.1.6 Try, catch

```cpp
try {
    new int[10000000000];
} catch (std::bad_alloc e) {
    cout << e.what() << '\n';
}
```

1.1.7 If, else

```cpp
bool success = false;
...
if (success) {
    // on success.
} else {
    // on error.
}
```

1.1.8 Switch

```cpp
switch (http_method) {
    case HTTP_GET:
        break;
    case HTTP_POST:
        break;
    default:
```

Chapter 1. C++ coding style guide
1.2 Naming

1.2.1 Files

To easier distinguish between C and C++ code header files should be named *.hpp and source files *.cpp.

1.2.2 Macros

In general, macros should be avoided, but if you have ones, you should capitalize them:

```cpp
#define VERSION 0x010A03
```

1.2.3 Classes, enums

Their names consist of all lower case letters and words are separated with an underscore. underscore_based_classes simply read easier than CamelCaseClassNames.

```cpp
class http_server {
    ...
    
    enum http_methods {
        ...
    }
};
```

Class fields, methods

They start with lower case letters and each word is separated with underscore.

```cpp
class http_server {
    public:
        void set_uri_handler(...);
};
```

Private fields

Private class fields end with underscore:

```cpp
class http_server {
    private:
        unsigned int port_;  
};
```

Constants

Use same naming convention as for usual variables, no UPPER CASE NAMES:

```cpp
class http_server {
    public:
        static const std::string protocol_version = "1.1";
    ...;
};
```
Setter, getter methods

Setters and getters have the same name. They are named after the variable they set. Setter accepts parameter to set. Getter method does not accept any parameters.

```cpp
class http_server {
public:
    void port(unsigned int port_);
    unsigned int port(void) const;

private:
    unsigned int port_; }
```

1.3 Error handling

Different forms of error reporting should be used as follows:

- **Static assertions** To prevent invalid instantiations of templates and to check other compile-time conditions.
- **Exceptions** To let some calling code know that a function was unable to fulfil its contract due to some run-time problems.
- **Error codes** To report run-time conditions that are part of a function’s contract and considered normal behavior.
- **Run-time assertions** To perform sanity checks on internal operations at run-time and ensure that major bugs do not enter production builds.

1.3.1 Exceptions

Catch exceptions by reference:

```cpp
try {
    // ...
} catch (const std::runtime_error& e) {
    // ...
}
```

[^1]: [http://josephmansfield.uk/articles/exceptions-error-codes-assertions-c++.html](http://josephmansfield.uk/articles/exceptions-error-codes-assertions-c++.html)
1.4 Misc

1.4.1 Accessing class members

When accessing private class members always refer to them via this:

```cpp
class person {
public:
    std::string name() const {
        return this->name_;}

private:
    std::string name_;}
```

This makes it clear where variable `name_` came from without further code investigation. And avoids errors in some situations.²

1.5 TODO

- In source documentation: do not document what’s obvious. E.g. `std::string get_name();`

References

Makefile is a text file that defines targets and rules which are executed by Make utility. This document describes conventions for writing the Makefiles.

## 2.1 Naming

### 2.1.1 Files

The recommended name for make files is *Makefile*. Misc make files with common targets or variables should have extension `.mk`. This helps text redactors to identify that this is a makefile and enable syntax highlighting.

### 2.1.2 Targets

Target names should use lower case letters. Words are separated with a hyphen `-' . E.g.:

```make
  test-debug:
    $(build_dir)/debug/bin
```

### 2.1.3 Variables

Variables which are not special to make or inherited from the environment should be in lowercase. Words should be separated with underscore symbol `_' . E.g.:

```make
  src_dir = $(CURDIR)/src
  build_dir = $(CURDIR)/build
```

## 2.2 Special targets

### 2.2.1 Phony targets

Phony target declarations should follow appropriate target declarations rather than be defined in one place [f1]. This way it's easier to maintain targets.

Good:
all: build test
.PHONY: all

Bad:

.PHONY: all build test

all: build test

References
3.1 Naming

3.1.1 Commands

Use lowercase letters:

`add_executable(main main.cpp)`

3.1.2 Variables

Local variable names should use all lowercase letters:

`set(src_dir "${CMAKE_CURRENT_SOURCE_DIR}/src")`

References
[f1] https://techbase.kde.org/Policies/CMake_Coding_Style