style-guides Documentation

Release latest

January 13, 2015

Contents

This repository contains my personal misc coding conventions.

CHAPTER 1

C++ coding style guide

Why yet another C++ coding style? Because there is no standard style. Each company, organization has it's 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 charactes should be used on a single line. Why?:

- Humans read narrower columns faster.
- http://www.emacswiki.org/emacs/EightyColumnRule

1.1.2 Indentation

Tabs. Tab size is 8 spaces.

1.1.3 Namespaces

```
namespace log
{

...

}
```

1.1.4 Classes

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

protected:
        ...
```

```
private:
        ...
};
```

1.1.5 Enums

```
enum severity_level {
        debug, info, warning, error
}
```

Or

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

1.1.6 Try, catch

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

1.1.7 If, else

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

1.1.8 Switch

```
switch (http_method) {
HTTP_GET:
        break;

HTTP_POST:
        breaj;

default:
}
```

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:

```
#define VERSION 0x010A03
```

1.2.3 Classes, enums

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

```
class http_server {
...
};

enum http_methods {
...
};
```

Class fields, methods

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

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

Private fields

Private class fields end with underscore:

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

Constants

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

```
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.

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

private:
        unsigned int port_;

};


void
http_server::port(unsigned int port_)
{
        this->port_ = port_;
}

unsigned int
Http_server::port(void) const
{
        return this->port_;
}
```

1.3 Error handling

Different forms of error reporting should be used as follows:[1]

- *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:

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

[1] 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`:

```
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

[2] http://www.parashift.com/c++-faq-lite/nondependent-name-lookup-members.html

CHAPTER 2

Makefile style guide

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* [f2]. 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.:

```
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.:

```
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

CHAPTER 3

CMake coding style guide

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

Bibliography

[f1] http://clarkgrubb.com/makefile-style-guide#phony-targets

[f2] https://www.gnu.org/software/make/manual/html_node/Makefile-Names.html

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