ViewPoint C++ Style Guide and Core Guidelines

C++ is a powerful, complex, feature-rich language. The goal of this guide is to provide a set of conventions and best practices designed to help you produce code that is not only correct but also maintainable, efficient, readable and consistent.

The guide is supposed to be a living document under continuous improvement and that's why it will never be complete. Try to follow the proposed recommendations but also stay creative.

Some coding styles are ~~stolen from~~ based on Google C++ Style Guide. There are also numerous books and internet resources for best practices of modern C++ (e.g. C++ Core Guidelines).

Let's go…

Key Principles

- **Consistency**: Consistent code is easier to understand and maintain. We'll cover naming conventions, formatting, and other stylistic choices to ensure uniformity across projects.
- **Safety and Performance**: Striking the right balance between safety and performance is crucial. We'll explore best practices for memory management, exception handling, and optimization.
- **Modern C++ Features**: C++ evolves, and so should our code. We'll discuss how to leverage modern features effectively.
- **Code Organization**: Well-organized code reduces complexity. Learn about project structure, header files, namespaces, and separation of concerns.
- **Documentation**: Clear comments and documentation enhance code readability. We'll emphasize the importance of documenting code.

Naming Convention

As well known, naming is one of the hardest thing in Computer Science. Having a naming convention is a cornerstone of good coding practices, ensuring that code is self-explanatory and accessible to all team members. While naming things: follow the agreed style/format and find names that tell a story about the code and explain the purpose of the named objects.

(The rules must be applied to all new projects. For old projects you might follow the naming schema of the given project if this leads to better readability.)

Project Names

Project names should be all lowercase and can include underscores (`_`) or dashes (`-`).

```
vpmain
vpimagecache
vp-image-repo
```

File Names

File names should be all lowercase and can include underscores (`_`) or dashes (`-`).
C++ files should end with `.cpp` and header files should end with `.h`.

```
main.cpp
htpp_request.cpp
htpp_request.h
```

Type Names

We use upper camel case (`UpperCamelCase`) for type names. This rule is applied to all types - classes, structures, enums, type aliases… Don't add the prefix 'VP'.

```
class ConnectionPool;
struct ConnectionOptions;
enum class ServerStatus {
    NotRunning,
    Starting,
    Running,
    Stopping
 };
```

Variable Names

Variable names follow the snake case (`snake_case` - all lowercase, with underscores between words) schema. Data members of classes(but not structs) have trailing underscores (`_`) .

```
int subscription_id {0};
QString procedure_name;
bool finished {false};

struct ImageId
{
    QString study_uid;
    QString series_uid;
    QString image_uid;
};

class ImageLocation
{
...
private:
    ImageId id_ {-1};
    QString file_path_;
}
```

Constant Names

Global variables declared `constexpr` or `const`, and whose values are fixed for the duration of the program, are named with a leading "k" followed by upper camel case.

```
namespace {
constexpr auto kPublishTimeout = std::chrono::seconds(10);
}
```

Function Names

Regular function names (free functions or class methods) are:

- lower camel case (`lowerCamelCase`)
- snake case (`snake_case`)

```
void addWorklistItem();
void add_worklist_item();
```

Please use one or another option consistently within a project.

Namespace names

Namespace names are all lower case, with words separated by underscores (snake case).

```
namespace vp {
  namespace http {
  ...
  }
}

namespace vp::dicom {
...
}
```

Macro names

In case you cannot avoid using macros, they are upper case with words separated by underscores.

```
#define VPLOG_MODULE_NAME "Main"
```

Formatting

Along with naming convention, code formatting is essential for clear readability, easier maintenance, and efficient collaboration.

Line Length

We don't set any hard limitation to the line length in your code. The code with a hard limit(e.g. 80 characters long) would not use the whole width of modern monitors and would remind on the old VGA resolution. From the other side don’t make your text extremely wide. Remember that one of the main goals of the style guide is readability.

Tabs vs. Spaces

For indentation we use spaces and not tabs. An indent is 4 spaces wide. Set your editor to replace tabs with spaces automatically.

Parenthesis and curly braces

- open parenthesis is on the same line as the function name
- close parenthesis is on the same line with the last function arguments
- open curly brace of a function/class/struct/enum/namespace is either on a new line or on the end of the last line of the declaration
- close curly brace is either on a new line or on the same line as the corresponding open curly brace
- open curly brace of looping or branching statement is on the same line as the statement

```
void start_server(const QString& server_name);

int addPatient(const QString& last_name, const QString& first_name, const QString& middle_name,
               const QString& dicom_id)
{
...
}

enum class ProcessingState { ReadyForExecution, Processing, Done }

class Request
```

```
{
public:
  void send();
}
```

```
if (condition) {
  ...
}
else {
  ...
}

for (auto const item: items) {
  log(item);
}
```

Order of Includes

Headers are included in the following order:

- precompiled header - *stdafx.h* (if it’s used)
- system and standard libraries headers
- third party or other VP components headers
- headers of the given project

```
#include "stdafx.h"

#include <chrono>
#include <mutex>
#include <condition_variable>

#include <QMetaObject>
#include <QThread>

#include <pets/classification.h>

#include "dog.h"
#include "cat.h"
```

Core Guidelines

Using Namespaces

Namespace provides a scope to the identifiers(classes, functions, variables). It’s used to organize code into logical groups and to prevent name collisions. Namespaces can be nested within each other, allowing for a hierarchical organization of code.

If you develop a “public” interface that might be used by other components put all declaration into a namespace. There is a root namespace in ViewPoint - ‘vp’. All other namespaces must be nested into the root namespace. Try to invent a good name for a namespace that best describes the component. Examples of ViewPoint namespaces:

```
namespace vp::http {
}

namespace vp::dicom {
}

namespace vp::dicom::repo {
}
using namespace vp::dicom::repo;
```

Function/method declarations

- First of all a function must have a good name reflecting the purpose of the function.
- Ideally a function must perform a single well defined action without hidden side affects.
- For all public functions add a short function description as a comment.
- Returning a value directly from a function (using the return statement) is generally preferred over using output parameters.
- If a function allocates(and returns) a resource think about the resource's ownership (who owns the resource? who destroys the resource?).
- Define proper function parameters (const vs non const, by pointer vs by reference vs by value)
- Always declare whether a function is guaranteed not to throw using *noexcept* keyword. If function can throw exceptions - specify which exceptions it can throw and at what conditions this happen(as a comment to function description).

```
class Repository final
{
public:
    // 'http' - fully initialized HttpClient (don't forget to set the base URL since Repository
    // works internally with relative endpoints only).
    // 'cache' - fully initialized ImageCache.
    // The 'thread' argument is optional and may be passed to supply an external thread
    // that provides the event loop needed for internal Repository implementation.
    // If thread argument is null the Repository will create its own thread.
    Repository(const std::shared_ptr<http::HttpClient>& http, const std::shared_ptr<dicom::ImageCache>& cache, QThread* thread = nullptr) noexcept;

    // Returns the thread in which the repository lives.
    // It might be used to change thread affinity for client objects (e.g. objects implementing slots)
    QThread* thread() const noexcept;

    // Returns shared pointer to ImageCache the Repository was initialized with.
    std::shared_ptr<dicom::ImageCache> cache() noexcept;

    // Returns an ImageLocation by image identifier 'id' (all UIDs must be present - study, series and image UIDs).
    // Optional callback 'callback' is called when the image is found in the cache or successfully downloaded
    // from the server or in case any errors occurred (e.g. HTTP error or server error) and the search stopped.
    // The callback is called in the thread of 'context' if it's provided, otherwise in the working thread of Repository.
    // @exception: throws CVPGenericExc if any file operation failed
    [[nodiscard]] std::unique_ptr<ImageLocation> get(const ImageId& id, const OnFinishedCallback& = {}, const QObject* context = nullptr);
...
}
```

Exception Handling

- Exception specification is essential part of function declaration/documentation (at least we should follow this rule for all public/interface functions/methods)
- Use `noexcept` specifier for the functions that cannot throw (avoid using throw() specifier because it's deprecated).

- As a rule don't catch `std::bad_alloc`, let application crash nicely.
- Catch `bad_alloc` only if you can handle this situation and let program run further. (E.g. it's certainly makes sense to catch this exception in case an attempt to allocate a huge chunk of memory has failed, let user know about that or log the error to the file and continue if it's possible. But if the program cannot allocate 200 Bytes it's probably better to fail immediately).
- Avoid using `catch(...)`. Prefer catching exceptions by their types wherever it’s possible.
- Catch an exception as soon as possible if you can handle it, otherwise re-throw.
- Catch exception object by `const reference`.

```
// @exception: throws CVPGenericExc if configuration file cannot be opened
Options read_configuration()
...
try {
...
    const auto options = read_configuration();
    perform_action(options);
...
}
catch (const CVPGenericExc& exc) {
 VPLOG_VPEXC(exc) << "Cannot perform action";
}
```

Logging

Logging is a crucial aspect of any software for a few reasons. In ViewPoint we mainly use logs for troubleshooting/debugging and performance optimization. There are 6 logging severity levels:

- **Critical**(aka **Alarm**)- serious conditions that could lead to system failure if not addressed immediately (this level is rarely used in ViewPoint. As practice has shown, it is not entirely obvious how to distinguish **Critical** and **Error** problems. Until we have an automatic log post-processing mechanism, for simplicity it is preferable to use the **Error** level instead)
- **Error** - any errors that affect functionality
- **Warning** - potential issues that could cause problems in the future or unexpected but not erroneous behavior
- **Info** - messages that record normal product operations
- **Debug** - detailed information for debugging purposes
- **Verbose**(aka **Trace**) - more detailed information for debugging purpose

Logging must be seen as essential part of any software components and the log itself as a product. Think carefully what you are going to log at every level. While implementing the code, try to reproduce the error situations and verify whether your log provides enough information to understand and fix the problem. Write main steps of a bigger action to the log at Info or Debug level. This will help to understand possible performance issues or localize any troubles up to the step.

The header file ..\Common\Base\VPLogMacros.h contains convenient macros that might be used for logging (VPLOG_ERROR, VPLOG_WARN, VPLOG_INFO, VPLOG_DEBUG…).

```
QString read_file(const QString& file_name)
{
  QFile file(file_name);
  if (!file.open(QIODevice::ReadOnly)) {
      // throw CVPGenericExc("Cannot open file");                                         // not good
      // throw CVPGenericExc("Cannot open file: %1", file_name);                          // better
      throw CVPGenericExc("Cannot open  file: [%1]. Error: %2", file_name, file.errorString());    // good!
      ...
  }
}
....
try {
```

```cpp
  auto content = read_file(gateway_config_file);
  ...
}
catch (const CVPGenericExc& exc) {
  VPLOG_EXC(exc) << "Cannot read Gateway configuration."
}

// ------------------------------

for(auto task: tasks) {
  VPLOG_INFO << "Start the task: " << task.info();
  ...
  VPLOG_DEBUG << "Parsing the DICOM file: " << source_file;
  ...
  VPLOG_DEBUG << "Convert raw data to MF image: " << mf_image_file;
  ...
  VPLOG_DEBUG << "Export MF image to PACS: " << pacs.info();
  ...
  if(pacs.export( mf_image_file)) {
      VPLOG_DEBUG << "Exported to PACS as " << pacs.image_location();
      ...
  }
  else {
      VPLOG_ERROR << "Cannot export to PACS [" << task.info() << "]. Error: "  << pacs.error_string();
      ...
  }
}

```

While reviewing the code pay attention to logging too!

Private Implementation (PIMPL)

The PIMPL (Pointer to IMPLementation) design pattern helps manage dependencies and improve encapsulation. In large codebase the PIMPL leads to faster build time. Don’t hesitate to use this pattern where it’s appropriate.

```cpp
// dog.h
namespace vp::pets;
{
class DogPrivate;
class Dog
{
public:
  Dog() noexcept;
private:
  std::unique_ptr<DogPrivate> pimp;
}
}
```

```cpp
// dog.cpp
using namespace vp::pets;

Dog::Dog() noexcept
: pimpl(std::make_unique<DogPrivate>())
{
}
```

Precompiled Headers

Precompiled headers (PCH) are a powerful tool in C++ that might significantly speed up compilation process of a large project. Depending on the content the compilation time of a typical VP middle size project can be reduced up to 4 times by introducing PCH. We use stdafx.h as a name for precompiled header. This header is also a perfect place to define commonly use macros like VP asserts or logging macros.

```
// stdafx.h
#pragma once

#include <Base/VPBase.h>
#define VPLOG_MODULE_NAME "Dogs"
#include <Base/VPLogMacros.h>
```

```
# dogs.pro
...
PRECOMPILED_HEADER = "stdafx.h"
...
```

Use this technique if it’s appropriate for the given project.

Variable initialization

All variables must be initialized before usage.
Prefer to initialize member variables directly in the class/struct definition.

```
namespace vp::dicom {
    struct CacheOptions
    {
        bool encrypt{ true };
        bool delete_on_destroy{ false };
        std::chrono::hours study_expiration{ 8 };
        std::chrono::hours cleanup_every{ 1 };
        ...
    };
}
```

Using auto

There were dark times in VP when `auto` was outlaw. It’s not the case anymore. In general `auto` makes your code more efficient and easier to maintain.

Comments

Using comments in C++ code is essential for several reasons, all of which contribute to better code quality and maintainability. The key benefits are:

- Improves Readability
- Facilitates Maintenance
- Aids Collaboration
- Documents Assumptions and Decisions
- Helps Debugging

Best practices for writing comments:

- **Be Clear and Concise**. Write comments that are easy to understand and to the point.
- **Avoid Redundancy**. Do not state the obvious. Comments should add value, not repeat what the code already clearly states.

- **Keep Comments Updated**. Ensure comments are updated when the code changes to avoid misleading information.
- **Explain the Why, Not Just the What.** Focus on explaining why certain decisions were made, not just what the code is doing.

However, the best code is self-documenting. Think about how to restructure and format code to make it more readable and understandable.

All public functions and methods must have comments describing what the function does and how to use it. If the function might throw exceptions, specify which exceptions and under what conditions. Comments might be omitted if the function usage is obvious.

Static code analysis tool

SonarQube - powerful tool designed for continuous code quality and security analysis - is integrated into VP CI pipeline. SonarQube usually produce meaningful issues. Take them seriously. Additionally SonarQube provide excellent explanations for the issues. This is a prefect possibility to learn language and best practices.

Improvements, suggestions, ideas ?

Please add your comments here - Discussion Page - "ViewPoint C++ Style Guide and Core Guidelines" or drop me a line @Viktor Chernenko