ab-vm Documentation

Release 0.0.1

Robert Young

Dec 14, 2019

Contents

1 Table of Contents 1

2 Indices and tables 5

CHAPTER 1

Table of Contents

1.1 Building Ab VM

1.1.1 Requirements

- CMake version 3.2 or greater
- Clang or GCC with support for C++17
- Git
- Ragel
- Python 3
- PyYaml
- Jinja2

1.1.2 The Basics

```
# Clone the project
git clone https://github.com/ab-vm/ab

# Create a build directory
mkdir ab/build
cd ab/build

# Run cmake and configure the build
cmake ..

# build it!
cmake --build .
```

(continues on next page)

(continued from previous page)

```
# Test it!
ctest .
```

1.2 License

Ab is made available under both the APL 2.0 and EPL 2.0.

1.3 Coding Style Guidelines

1.3.1 Choice of Language

Ab is written in C++14 with minimal standard library usage. In particular, be suspicious of newer standard library APIs introduced since C++11. Ab is written without exceptions.

Tooling and scripts should be written in python3.

1.3.2 Formatting Code

Ab uses clang-format. Always run clang-format on every commit.

1.3.3 Namespaces, Includes, and Project Layout

Project Configuration Headers

Configuration headers must be included at the top of every single source.

Configuration headers will change system APIs, and MUST be included BEFORE any system header.

Config.hpp is generated by cmake, and contains all configuration defines.

Namespaces and Nesting

All code should be under the top level project namespace. Use inner namespaces sparingly. Namespaces should almost exactly mimic the project layout.

Avoid nesting indentifers. Try to use globally unique indentifiers instead.

Example:

```
// Everything under the project namespace

namespace Ab {

class HeapConfig {}; // good - not nested

class Heap {
        class Config {}; // bad - nested and nonunique name.
};
```

(continues on next page)

(continued from previous page)

```
} // namespace Ab
```

Includes and Headers

There is one top-level include directory per project. All public headers are grouped in a project subdirectory. Always write include directives relative to the top-level include directory. The project name must prefix every include.

Example:

```
#include <Ab/Config.hpp> // Configuration first
#include <Ab/*.hpp>      // Internal headers, alphabetical, with the project prefix
#include <external.hpp>    // External headers, alphabetical
#include <system>          // System and standard headers, last.
```

1.3.4 Writing Classes

Naming

Classes, namespaces, and structs are all TitleCased. Only the first letter of an acronym should be uppercased.

Structs vs Classes

Use struct for simple POD-style objects. Use class when the object is complex.

Fields, Getters, and Setters

private and protected fields are named with trailing underscores. public fields don't have underscores.

Name the accessors after the field, without trailing underscore. Setters should take a single value, and return `*this`, implementing a fluent interface.

Example:

```
class Array {
public:
        inline constexpr auto size() const -> std::size_t {
                return size_;
        }

        inline auto size(std::size_t x) -> Array& {
                size_ = x;
                return *this;
        }

private:
        std::size_t size_;
};
```

1.3.5 Constants, Enums, and Macros

Code should be const by default.

Constants and macros are written as `MY_CONSTANT`. Use `static const constexpr` for constants, do not use macros.

Use `enum class` whenever possible.

Place compilation flags in the config header, do not add definitions to the command line.

Do not introduce a macro without a very good reason. Since macros cannot be namespaced, prefix all macros with the project name.

CHAPTER 2

Indices and tables

- genindex
- search