Strawhacks

Rock Chalk Rendezvous

Source Code Standards

Version 0.1

Rock Chalk Rendezvous	Version: 0.1
Source Code Standards	Date: 4/11/2024
code_stds1	

Revision History

Date	Version	Description	Author
03/21/2024	0.1	Initial structure and some standards included.	Ben Phillips

Rock Chalk Rendezvous	Version: 0.1
Source Code Standards	Date: 4/11/2024
code_stds1	

Table of Contents

Rock Chalk Rendezvous	Version: 0.1
Source Code Standards	Date: 4/11/2024
code_stds1	

1. Introduction

<describe the purposes of this document>

2. Practical

2.1. Include Guards

All header files should use an include guard of the following format. The first two lines of the file must be:

#ifndef RCR_<FILENAME>_DEFINITIONS
#define RCR_<FILENAME>_DEFINITIONS

The last line must be the #endif directive. All contents must be between the #define and #endif directives.

2.2. Encoding & Decoding

Encoding methods should have the function signature:

std::string encode() const

It is encouraged to use an std::ostringstream internally when encoding to avoid the $O(n^2)$ overhead of repeated string concatenation, especially for larger data structures. A static encode function is necessary in order to pass it as a function pointer and call it in generalized contexts, like encode_vector. In many cases a non-static method will be more convenient, and should be written as:

static inline std::string encode_static(const T& t) const { return t.encode(); }

Decoding methods should have the function signature:

Status decode(std::istream&, T&)
static inline Status decode_static(std::istream& s, T& t) { return t.decode(s); }

The TypeName& reference should have the decoded value written to it in the case of a successful read. Decoding methods should return a failure status if the data isn't formatted as expected or if stream.fail() is true (or stream to boolean coercion is false).

2.3. Structs

Functions should be methods of a struct only if their functionality is inherent to the data being stored by the struct.

2.4. Inheritance

Never inherit from types that will be constructed. It creates ambiguity about which version of a method is called in different scenarios, and there are many better methods of code reuse. Abstract types (that will never be constructed) may be inherited from as they can be used as interfaces.

2.5. Function Overloading

Never overload functions. If it is a different function, it should have a different name. Using default arguments is function overloading.

In most cases, operator “overloading” is just operator defining since most operators have no definitions for custom types by default. This is good when those operators have an inherent meaning for the type, like mathematical structures or structures that conceptually represent numerical values in a way that C++ can’t infer.

Defining a static method for a non-static counterpart or vice versa is acceptable as long as one is inlined and only calls the other version with the “this” argument provided differently. C++ really should just allow calling non-static methods statically since that is how they work internally, but C++ is silly so we have to compensate.

3. Formatting

3.1. Naming

Names for values should suitably describe how their purpose differentiates them from other values of the similar types. Longer names, especially function names, improve the ability of code to accurately explain what it is doing without the need for comments. Single letter names should be avoided unless the type or context gives enough information about it.

Abbreviations should never be used in names in order to avoid ambiguity. For example, “function” should never be shortened to “funct”, “func”, “fn”, or “f”, it should be spelled out entirely. If a name is used in a frequent and widespread manner, like the “int” keyword, abbreviations are acceptable.

Names of values should adhere to the following:

Snake case for variables and fields: days_this_month

Snake case for functions and methods: find_days_in_year

Capitalized camel case for struct names, enum names, and enum variants: ServerResponse::UserCalendarWritten
Upper snake case for static constants: MONTH_NAMES

3.2. Comments

Comments should only be used to clarify code whose intent is not already clear and not able to be made clear. Names that follow the guidelines in section 3.1 should make comments unnecessary in many places. Signposting comments, combined with proper spacing, also help when locating different sections of a longer file.

Todo comments can be used to mark places in the code where features will be implemented in the future.

3.3. Auto / Let

The “auto” keyword, also defined as “let” in the codebase, should not be used unless the type has a long name and either the type is unimportant and a more descriptive name is used or the type is obvious from the initialization.

“let value = typename(args);” is preferred over “typename value(args); because the latter looks very similar to a function declaration and puts visual distance between the name of the type being constructed and the arguments to the constructor.

“typename value;” is acceptable for calling a default constructor because this will happen all the time regardless.

The two keywords are not interchangeable: “let” should express that the type is obvious and that stating the type would be redundant, while “auto” should indicate that the type is not obvious but that it isn’t important and would clutter up the code to include, and the name of the variable aptly describes it.

3.4. Indentation

Spaces vs tabs, size

3.5. Curly Braces

When to use / not use where optional

3.6. Number representations

When numeric literals are the result of combining multiple values together, they should always be expressed in the terms of those operations. For example, five hours in minutes should be written as 5 * 60 rather than 300.

When the same numeric value is used in many places whose meaning is not otherwise clear, it should be represented with a macro using a #define directive in order to label and potentially facilitate modifying this value.

3.7. Structs

All data fields of a struct must be specified before any methods.

All fields and methods of a struct should be explicitly marked as either public or private (i.e. the struct declaration should always begin with a visibility specifier) when at least some part of the struct is to be private.

4. Testing

4.1.