Naivi C++ Style Guide

Loosely based on the Google C++ style guide

This document outlines the do's and don't's when writing C++ code for any Naivi related software product. Not following the rules will result in a food enforced penalty, enjoyed by the rest of the development team.

Header Files

- In general, every .cpp file should have an associated .h file. There are some common exceptions, such as unittests and small .cpp files containing just a main() function
- Try to make every header **self contained**, ie: can be included without having to rely on other headers.
- A header should have header guards and include all the other headers it needs
- Functions are declared in the header and defined in the cpp file. One liners are an exception to this rule, ie, can be declared and defined on the same line, but should not exceed a single line. Template definitions are defined at the end of a file or in a separate include file, ie, _include.hpp
- At the beginning of a header file: don't use #ifndef #define, **use: #pragma once**
- Try to avoid forward declarations, unless it's absolutely necessary to avoid cyclic dependencies. If you have to use forward declarations to avoid a cyclic dependency it's probably best to rethink your code design.
- **Don't use inline**, have faith in your compiler
- Use **quotes " " for local includes** and **< > for external includes**. Say you have a file with the following path: c:/awesome_project/include/base.h, when you want to include base.h in a file that resides in the same folder, use quotes, this makes base.h local to that file. If the file that includes base.h resides in a different directory (within the same or external project), use brackets.
    - coreattributes.h: *#include "attribute.h"*
    - transform.h: *#include <nap/coreattributes.h>*
- Start with local includes followed by external includes

Scoping

Namespaces

- With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Using namespaces in .cpp files is encouraged. **Do not use using directives (e.g. using namespace foo). If so, only in .cpp files**
- Do not use inline namespaces
- Try to end a namespace with a comment: // namespace
- Do **not** use namespace aliases: `namespace foo_bar = ::foo`

Nonmember, Static Member, and Global Functions

- Prefer placing nonmember functions in a namespace.
- Try to avoid global functions
- Group functions using a namespace instead of a class as if it were a namespace
- Global functions should start with a lowercase 'g': std::string gToString<T>(const T& value)
- Static members of a class should be closely related to instances of that class
- Static members of a class start with a lowercase 's': sRegisterCreateFunctions()
- If you define a non member function and it's only needed in it's .cpp file, use static linkage, ie: static int sFoo() to limit it's scope.

Local Variables

- Place a function's variables in the narrowest scope possible, and initialize variables in the declaration
- Declare variables as close to the first use as possible, except in loops:
```
Foo f;  // My ctor and dtor get called once each.
        for (int i = 0; i < 1000000; ++i)
{
        f.DoSomething(i);
}
```
- Prefer initialization using braces: `vector<int> v = { 1, 2 };`

Static and Global Variables

- Variables of class type with static storage duration are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such variables are allowed if they are constexpr: they have no dynamic initialization or destruction.
- Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only ints, chars, floats, or pointers, or arrays/structs of POD.
- If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your main() function or from pthread_once(). Note that this must be a raw pointer, not a "smart" pointer, since the smart pointer's destructor will have the order-of-destructor issue that we are trying to avoid.
- **Don't use** #define to declare constants, use const or constexpr instead: http://stackoverflow.com/questions/15218760/should-i-avoid-using-define-in-c-why-and-what-alternatives-can-i-use

Classes

With the introduction of move constructors and move assignment operators, the rules for when automatic versions of constructors, destructors and assignment operators are generated has become quite complex. Using `= default` and `= delete` makes things easier.

Provide the copy and move operations if their meaning is clear to the user and the copying/moving does not incur unexpected costs. If you define a copy or move constructor, define the corresponding assignment operator, and vice-versa. If your type is copyable, do not define move operations unless they are significantly more efficient than the corresponding copy operations. If your type is not copyable, but the correctness of a move is obvious to users of the type, you may make the type move-only by defining both of the move operations.

Use the new C++ 11 = default and = delete keywords to create or remove the default:

- Constructor
- Destructor
- Copy Constructor
- Copy Assignment Operator
- Move Constructor
- Move Assignment Operator

Constructors

- Avoid virtual method calls in constructors
- Initialize member variables using the { } in the header
- Use the =default directive to add a default constructor

Destructors

- Always make the base class constructors virtual
- If no specific behaviour is necessary, declare the destructor to be =default

Copy

C++ defines it's own copy constructor and assignment operator if not defined explicitly. Try to be as obvious as possible by using the =default or =delete keywords to either allow, disallow copy functionality. Implement your own if the object manages some dynamic allocated memory or has pointer variables and you want to be able to copy it.

- **When defining a copy constructor, also define a copy assignment operator.**

Move

C++ defines its own move constructor and assignment operator in a very limited set of circumstances. For more information read up on it over here: http://en.cppreference.com/w/cpp/language/move_constructor. It's never safe to assume that the move operator isn't automatically implemented, it's therefore wise to show your intention using the =default or =delete keywords.

- **When defining a move constructor, also define a move assignment operator.**

Inheritance

- Composition is often more appropriate than inheritance. When using inheritance, make it public.
- Try to avoid multiple inheritance

Operator Overloading

Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g. ==, <, =, and <<), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with libraries that expect those names.

- Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use | as a bitwise- or logical-or, not as a shell-style pipe.

Declaration Order

Use the specified order of declarations within a class: public: before private:, methods before data members (variables), etc.

Your class definition should start with its public: section, followed by its protected: section and then its private: section. If any of these sections are empty, omit them.

Within each section, the declarations generally should be in the following order:

- Using-declarations, Typedefs and Enums
- Constants (static const data members)
- Constructors and assignment operators
- Destructor
- Methods, including static methods
- Data Members (except static const data members)

Functions

Parameter Ordering

- When defining a function, parameter order is: inputs, then outputs.

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be non-const pointers. When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

Write Short Functions

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

Reference Arguments

- const arguments passed by reference are always non modifiable input arguments
  - const float& intensity
- Arguments passed by reference are modifiable and should be preceded by 'out':
  - float& outIntensity

C++ Features

Exceptions

- We do not allow exceptions

Casting

- Be as explicit as possible, something like this is not ok and most compilers will throw a warning
  - float f(0.0f); int x = f;
  - **Instead use**: float f(0.0f); int x = static_cast<int>(f);
- Use C++-style casts like static_cast<float>(double_value) as much as possible
- Try to avoid c style casts such as: (int)3.5f or int(3.5f)
- Never use dynamic_cast
- Try to avoid reinterpret_cast

The problem with C casts is the ambiguity of the operation; sometimes you are doing a *conversion* (e.g., (int)3.5) and sometimes you are doing a *cast* (e.g., (int)"hello"). Brace initialization and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

For more information:
http://stackoverflow.com/questions/103512/in-c-why-use-static-castintx-instead-of-intx

Const

- Use const whenever it makes sense
- We don't actively encourage the use of constexpr (C++ 11), although it might be preferred in some const cases
- We encourage putting const first, so:
  - NOT: int const * foo
  - BUT: const int * foo

Some variables can be declared constexpr to indicate the variables are true constants, i.e. fixed at compilation/link time. Some functions and constructors can be declared constexpr which enables them to be used in defining a constexpr variable.

Constexpr therefore defines a more robust specification of the constant parts of an interface. Use constexpr to specify true constants and the functions that support their definitions. **Avoid complexifying function definitions** to enable their use with constexpr. Do not use constexpr to force inlining.

Integer Types

- The only actively used signed or unsigned integer type is int (compiler specific, minimum of 32 bits)
- Use standard library <stdint.h> integer types for different sizes of int
- NAP defines its own set of integer types based on the types in <stdint.h>
- Never use short, long etc.
- When iterating, use size_t or auto. This helps avoid platform specific warnings

Macros

- Try to avoid them as much as possible
- Prefer inline functions, enums and const variables to macros
- Don't rely on macros to define pieces of a C++ API
- Don't use macros to store a constant, use a const variable

Use nullptr

- Always use nullptr, never use NULL
- Use 0 for integers and 0.0 for reals

Auto

Auto is permitted, for local variables only, when it increases readability, particularly as described below. Do not use auto for file-scope or namespace-scope variables, or for class members. Never initialize an auto-typed variable with a braced initializer list.

Programmers have to understand the difference between auto and const auto& or they'll get copies when they didn't mean to.

- We encourage the use of auto where the type doesn't aid in clarity for the reader
  - for (const auto& v : vector) or auto iterator = std::find_if(vector, ..)
- Use type declarations when it helps readability
  - for(size_t i=0; i<100; ++i) or float v =static_cast<float>(d)

Lambdas

- Use lambda expressions where appropriate.

- Prefer explicit captures when the lambda will escape the current scope.
    - Instead of:
      ```
      Foo foo;
      executor->Schedule([&]
      {
              Frobnicate(foo);
      }
      ```
    - Use:
      ```
      Foo foo;
      executor->Schedule([&foo]
      {
              Frobnicate(foo);
      })
      ```

Aliases

- Use `using` instead of `typedef`
- Try to limit the number of aliases in the public (header) scope. This will make them available to users of the API and creates unnecessary clutter. Only put it in your public API is you intend it to be used by your clients.

Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

Tips:
- Functions and methods should be **verbs**, describing an *action*.
- Variables should be **nouns**, representing *state*.
- Classes are nouns as well, describing a well defined **concept**

General Naming Rules

- Names should be descriptive; avoid abbreviation.
- Do not worry about saving horizontal space, it's more important to make your code understandable
- Do not use abbreviations that are ambiguous to readers outside of the project
- Do not abbreviate by deleting letters within a word

Good:

```
int price_count_reader;       // No abbreviation.
    int num_errors;                   // "num" is a widespread convention.
    int num_dns_connections;          // Most people know what "DNS" stands for.
```

Bad:

```
int n;                        // Meaningless.
    int nerr;                         // Ambiguous abbreviation.
    int n_comp_conns;         // Ambiguous abbreviation.
    int wgc_connections;              // Only your group knows what this stands for.
    int pc_reader;                    // Lots of things can be abbreviated "pc".
    int cstmr_id;                     // Deletes internal letters.
```

File Names

- Should all be lowercase and preferably not contain any underscores or dashes
  - attributes.h
  - component.h
  - rectangle.h
  - numerictypes.h
- Try to make your filenames very descriptive. For example, use: httpclientlogs.h rather than logs.h

Type Names

- Start with a capital letter and have a capital letter for each new work, with no underscores:
  - MyNewClass
  - MyNewEnum
  - MyNewStruct
- Types include:
  - Classes
  - Structs
  - Enums
  - Aliases
  - Template Parameters

Variable Names

- The names of class data members are all camelcase, preceded by the letter 'm' if not an Attribute<T> or public:
  - `Attribute<float> intensity            //< Public Attribute`
  - `Attribute<bool> enableDamping  //< Public Attribute`
  - `float intensity                       //< Public member`

  - `bool mCached                  //< Private member`
  - `float mTime                           //< Private member`

- Struct members are all public and follow the class variable naming convention.
- Variables in function scope are all lowercase, with underscores between words.
    - int current_cycle;
    - vec3f new_point_position
- Static variables all start with the letter 's'
    - std::string sDefaultOperatorName
    - bool sRegistered

Function Names

- Function names are camelcase
    - getName()
    - getCount()
    - setCount()
    - updateCache()
- Global functions are preceded by the letter 'g', also when declared in a namespace
    - gGetString()
    - gClamp()
- Static functions are preceded by the letter 's'
    - sRegisterType()
    - sGetRegisteredTypes()

Namespace Names

- Namespace names are all lower case
- Top level names are based on the project name
- Avoid nested namespaced that match well known top level namespaces

Enumerator Names

- Start with the letter 'e':
    - eDays
        - Monday
        - Tuesday
        - Wednesday
        - ...
    - eDrawMode
        - WireFrame
        - Shaded
        - ...
- Are always based on the C++ 11 enumerator classes
    - enum class eDays : public int

Macro Names

- If you define one use all uppercase with underscores between words:
    - MY_HORRIBLE_MACRO
    - RTTI_DEFINE
    - RTTI_DECLARE

Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

Comment Style

- Use either the `//` or `/** */` syntax, as long as you are consistent.

Class Comments

Every class declaration should have an accompanying comment that describes what it is and how it should be used!

The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. Document the synchronization assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

The class comment is often a good place for a small example code snippet demonstrating a simple and focused usage of the class.

When sufficiently separated (e.g. .h and .cc files), comments describing the use of the class should go together with its interface definition; comments about the class operation and implementation should accompany the implementation of the class's methods.

```
/**
* Attribute
```

```
*
* Attribute is a system wide available parameter that holds a name and contains a value
* This value......
*/
class Attribute
{};
```

Function Comments

Almost every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g. simple accessors for obvious properties of the class). These comments should be descriptive ("Opens the file") rather than imperative ("Open the file"); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Use the Javadoc style decorators to explain input parameters and return values: @param and @returm

Types of things to mention in comments at the function declaration:

- What the inputs and outputs are.
- For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
- If the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.
- If there are any performance implications of how a function is used.
- If the function is re-entrant. What are its synchronization assumptions?

If the function is complex and needs a lot of documentation, use the Class style comments to explain functionality. Otherwise use the double slashes // preceding the function or, if it's a one-liner, place it after the the function declaration

```
/**
 * gSmoothDamp
 *
 * Interpolates a value over time from @currentValue to @targetValue using a
   damping
 * algorithm
 *
 * @param: currentValue: the previously computed value, starting at 0.0
 * @param: targetValue: the value that @currentValue needs to become
 * @param: velocity: the current velocity that is updated and used to blend the
   value
 * @return: the dampened (interpolated)  value, also the next @currentValue
 */
float gSmoothDamp(float currentValue, float targetValue, float& velocity)


// Returns the total number of vertices the curve currently has
Int getCount()


std::string getName()   //< Returns the object's current name
```

Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should *not* just repeat the comments given with the function declaration, in the .h file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

- The actual name of the variable should be descriptive enough to give a good idea of what the variable is used for.
- Sometimes more comments are required
- All global variables should have a comment describing what they are and why it's global
- Don't state the obvious, if the code is self explanatory, don't add a comment
- with multiple comments on subsequent lines, aligning them makes the code easier to read.

```
// Used to bound-check table access, -1 means that we don't know how many entries we have
int mTotalEntryCount

int mVertCount            //< Total number of vertices the curve has
int mPolyCount            //< Total number of polygons the curve has
```

Line Comments

- Lines that are non obvious should get a comment at the end of the line.
- Always separate these lined with the code using a minimum of 2 spaces
- If you have several comments on subsequent lines, it can often be more readable to line them up

TODO comments

- Use TODO comments for code that is temporary, a short-term solution or not perfect
- TODO comments should include the string TODO followed by a name, email or bug ID

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
    // TODO(Zeke) change this to use relations.
    // TODO(bug 12345): remove the "Last visitors" feature
```

Deprecation Comments

- Use @DEPRECATED to mark deprecated code
- @DEPRECATED comments should include the string @DEPRECATED followed by a name, email or bug ID and potential suggestions for refactoring the call site
- Try to find the time to actually remove / change the code that is deprecated

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Line Length

- Each line of text in your code should be at most 120 characters long
- Don't break lines

Spaces vs. Tabs

- Use only tabs

Function Declarations and Definitions

- Return type on the same line as function name
- Parameters on the same line if they fit
- Wrap parameters if they do not fit on a single line
- Opening brace after function declaration / definition
- Closing brace on seperate last line
- Align multiline parameters

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2)
{
		  DoSomething();
	...
	}
```

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
	  Type par_name1,
  Type par_name2,
	  Type par_name3)
{
		DoSomething();  // 2 space indent
		...
	}
```

Lambda Expressions

- Format parameters and bodies as for any other functions
- Format capture lists like other comma separated lists
- Don’t leave a space between the & and variable name for reference captures

```
// Example One
int x = 0;
    auto add_to_x = [&x](int n)
{
x += n;
};


// Example Two
set<int> blacklist = {7, 8, 9};
    vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
    digits.erase(std::remove_if(digits.begin(), digits.end(), [&blacklist](int i)
{
    return blacklist.find(i) != blacklist.end();
}), digits.end());
```

Function Calls

- Write a call on a single line or start on a new line properly indented and aligned
- Split multiple arguments when complex or confusing, create variables that capture that argument in a descriptive name

```
// Example One
bool result = DoSomething(argument1, argument2, argument3);


// Example Two
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

Conditionals

- No spaces inside parenthesis
- The if and else keywords belong on a separate line
- Use braces with multiple line execution statements, otherwise not
- Always place braces on a separate line
- Short statements can be on one line

```
    // Example One
    if (condition)
{
        ....
        ....
}
    else
    {
            ...
        ...
        }

    // Example Two
if (condition) {  // Good - proper space after IF and before }.
```

Switch Statements

- You may use braces for blocks, following the conditional statement guidelines
- Braces are optional for single line expressions
- If not conditional or an enumerated value, switch statements should always have a default case

```
    // Example One
    switch (var)
    {
        case 0:
{
        ...         // 4 space indent
        break;
}
 case 1:
{
        ...
        break;
}
 default:
        assert(false);
}
```

Loops

- Curly braces on separate line
- If a loop is a single statement, line is placed under loop expression

```
// Example One
for (int i = 0; i < kSomeNumber; ++i)
        printf("I love you\n");


// Example Two
for (int i = 0; i < kSomeNumber; ++i)
{
        printf("I take it back\n");
    }
```

Pointer and Reference Expressions

- No spaces around period or arrow
- Pointer operators do not have trailing spaces
- We do not use Hungarian declaration

```
// Example One
x = *p;
    p = &x;
    x = r.y;
    x = r->y;
```

Variable and Array initializations

- Use =, () or {}
- Be careful with braced initialization lists when the type has a std::initializer_list constructor, which is always preferred by the compiler
- Use parentheses instead of braces to force a non initializer_list constructor!

```
// Example One
vector<int> v(100, 1);     // A vector of 100 1s.
   vector<int> v{100, 1};  // A vector of 100, 1.
int pi(3.14);          // OK -- pi == 3.
   int pi{3.14};           // Compile error: narrowing conversion.
```

Class and Struct Format

- In order: public, protected and private
- Declarations only in header, no implementation, except for one-liners
- Use the end of a file or a hpp file for template definitions
- Braces go on a separate line
- Column for readability are encouraged
- The public, protected and private key words should be on the same line as the class keyword.
-

```
// Example One
class MyClass
{
   Public:
   MyClass() = default;
   MyClass(int var);

         Virtual ~MyClass() = default;
```

```
        void someFunction();
        void someFunctionThatDoesNothing()  { }

        void setSomeVar(int var)            { mSomeVar = var; }
        Int getSomeVar() const              { return mSomeVar ; }

    private:
        bool someInternalFunction();
        int mSomeVar = 0;
        Int mSomeOtherVar = 0;
    };
```

Namespace Formatting

- The contents of namespaces are indented 1 tab for every namespace

```
// Example One
namespace
{
        void foo() {}
    ...

    }  // namespace
```