Coding Guidelines

# Introduction

All code in the dhorn namespace (should) follow a general set of coding standards/guidelines. This document describes naming conventions for types, functions, members, and namespaces among other things. In general, naming follows that of the STL.

# Namespaces

All code (types, globals, etc.) exist in the dhorn namespace (note that this does not imply that dhorn is the parent namespace; e.g. there could be the type dhorn::foo::bar). All *public* code that is not OS specific (e.g. the dhorn::service\_container type) exists with the dhorn namespace as its parent. Code that is OS specific (e.g. dhorn::win32::window) exists one level removed from the dhorn namespace with its parent namespace appropriately named. For example, all DirectX code exists in the dhorn::d3d namespace whereas all Windows specific code exists in the dhorn::win32 namespace. Again, this applies to *public* types/functions/etc.

## Garbage Namespace

Code that is *not* public exists in the garbage namespace existing in the same namespace as the code that uses it. For example, the type dhorn::foo::bar may reference the meta-function dhorn::foo::garbage::doit. In other APIs, the equivalent of the garbage namespace is usually named details. So why is details not used here? The simplest answer is that I was unaware of the details naming convention when I created the garbage namespace. Even then, I kind of like the name garbage better since it is more explicit to client code that they should not use such functions.

This should be obvious, but clients should never reference anything existing in the garbage namespace directly as types can freely be removed and/or renamed.

# Type Names

As mentioned earlier, *public* type names follow the same naming convention as the STL: entirely lowercase with words separated by underscores. For those of you coming from a camel-casing world, something like FooBar would instead be named foo\_bar. This naming convention applies to type definitions as well.

Private types (this includes types in the garbage namespace) do not necessarily need to follow these conventions and may differ. E.g. some use STL naming, whereas others may use Pascal casing. In general, anything observable to the user, public or not, uses the STL naming convention, whereas anything not observable to the user (e.g. private member types) use Pascal casing.

## basic\_\* Type Names

Additionally, any type that is heavily templated should be named as basic\_\* with numerous associated type definitions, dropping the basic\_ prefix and changing the name to fit what it logically represents. For example, both std::string and std::wstring are type definitions of std::basic\_string; one using char as a template argument and the other using wchar\_t as the template argument.

# Function Names

Public function names should, for the most part, follow the same naming conventions as type names. I.e. lowercase with underscores as separators. This applies to both public member functions as well as global functions.

## Operator Overloading

In general, all types should prefer the use of operator overloading as opposed to explicit function names. For example, there should never be the function named get\_at; use operator[] instead. It may be the case where both functions exist (e.g. std::vector has both operator[] as well as at), and this is okay, but it should be used only if it adds value.

## Private Member Functions

Private member functions deviate from the STL naming convention; they should use Pascal case. E.g. the function private\_function would instead be named PrivateFunction. The reason for this is that it is desirable for public and private functions to be easily distinguished from one another. It’s common practice to have both public and private functions only call private functions, and this makes that visually obvious. Note that for all naming purposes, protected functions/members should follow the same rules as private functions/members.

## Static Member Functions

The naming of static member functions should follow the same guidelines as that of their non-static counterparts. I.e. public static member functions should follow STL naming conventions and private/protected static member functions should be Pascal case. No indication that the function is static is necessary (i.e. prefix or postfix) as the function name should be indicative of whether it is static or not, and if the name is not clear, IDEs make the problem irrelevant.

# Variable Names

Public variable names should follow STL naming conventions as well. This applies to global const values as well as struct members. Note member variables should only be made public for POD types. For non-POD types, use a function and/or operator that returns a reference to the member variable.

## Private Member Variables

Similar to private member functions, private member variables follow a different naming scheme than their public counterparts. All private member variables should begin with an underscore followed by the variable name in Camel case. E.g. a private member variable might be named \_isFoo.

# STL Conventions

In general, all code should be written as if it were part of the STL. That is, any collection data type exposes an iterator, anything that can be generalized uses templates. This rule is occasionally violated when it adds questionably little/no benefit, or when inclusion will make the code difficult to write and/or maintain (e.g. the use of allocator types). Additionally,