The following standards and patterns are used throughout the Standard Extensions library (STDE).

## Library Structure

The STDE library is designed to look and feel like a natural extension of the Standard Template Library (STL). Like the STL, STDE is designed to be used as headers only. There are no libs or dlls to consume. The headers contain all the relevant pieces to make them functional, so there’s never a need to include anything else besides the individual headers you need.

## Naming conventions and Namespaces

In the spirit of feeling like a natural extension of the STL, STDE’s functions and classes follow the naming conventions of the STL whenever possible. Similarly, everything in the STDE is in the stde namespace. Following the naming conventions means using lowercase throughout, and using underscores to separate words. Here are some example naming rules to follow:

1. \_ptr suffix on a class means that this type represents a pointer. Examples: shared\_ptr, com\_ptr, unique\_ptr
2. unique\_ prefix on a class means that the resource contained in this class is not meant to have multiple copies floating around. This means that this class is the sole owner of that object, and should delete the resource when it goes out of scope.
3. shared\_ prefix on a class means that this object supports sharing, normally by means of some sort of ref counting mechanism. The object does not necessarily own the resource, just a reference to the resource. If it’s the last owner of the object when it goes out of scope, then the object should be deleted.

## Calling Convention

In general, the **fastcall calling convention is preferred** for most functions. Both stdcall and fastcall have the benefit of generating a single copy of the stack cleanup code (in the function itself), which reduces duplication of that code at each call site. However, fastcall will also try and fit the first 3 DWORD sized parameters into registers and bypass stack variables altogether when possible, making it an ideal choice for most scenarios.

Note, however, that there are many valid reasons to use the other calling conventions. For instance, variadic parameters cannot possibly be cleaned up properly by the function, as there is no single correct clean up code that will suffice for all possible callers. For that reason, the caller MUST be the owner for stack cleanup, and that limits these calls to **cdecl only**.

## Template Use and Guidelines

C++ templates are a great way to improve code reuse while still benefiting from type safety, compile time checks, compile/link time optimizations, and more. When using templates, the following concepts should at least be considered:

1. traits are a mechanism for maximizing code reuse when you have a class of types that are nearly identical, but with some subtle differences in semantics. The classic example (already in the STL) are the string objects. They are both strings of characters, they have lengths, can be appended, etc… The difference is just that one has wide characters and the other not. That difference is captured in a traits object which is used to instantiate the templates. See the unique\_handle type in stde for another example.
2. The <type\_traits> header includes a class of compile time checks that can be performed using static\_assert. This allows validation of non-trivial checks within templates at compile time. We should try and use these to call out invalid uses with very clear error messages when possible. They can be used at the class or method level.
3. Default template parameters should be provided when they make sense, to help make the types easy to use.

## Class design considerations

Does it make sense to make copies of this class? Does it make sense to assign it to something? Does it make sense to construct it with another type that’s coercible to the constructor’s types? These are all questions that we often forget to ask ourselves when quickly writing up a class definition. But it’s these very subtle behaviors that can often lead to difficult to diagnose bugs, particularly when used in conjunction with STL and, more specifically, STL containers. The containers all have specific requirements of the types put in them. It’s possible to get them to compile, but completely break at runtime if care is not taken to design classes correctly for the containers. In general:

1. Unless it makes sense to ‘copy’ or ‘assign’ your object, you should declare a private copy constructor and assignment operator to prevent the compiler from generating public ones for you. They don’t need bodies, just declarations.
2. If you plan on using your type in containers, though, then you *must* support copy construction and assignment. In these cases, be sure your object correctly implements them so that your contracts are not broken.
3. If you do not support taking in another coercible type during construction, then let the compiler enforce that for you by declaring the constructor as explicit. For example, if you take in a HANDLE as a parameter to your constructor, the compiler might normally allow someone passing in 47, as it’s implicitly convertible to a HANDLE value. However, this doesn’t make any sense, and should be safeguarded against.