Design Overview

Assertions

Stroika makes extensive use of assertions to help assure correct code, and to document how to use the Stroika library.

Nearly every API has pre-requisite ‘Require’ statements, and ‘Ensure’ statements which make promises about the state of the object (in methods) or return values. This provides not only documentation, but executable documentation – which makes it much easier to develop correct Stroika applications.

Debug Builds

In debug builds, all Stroika Assertions are checked – evaluated at runtime, and problems cause the program to abort, or drop into a debugger. These are *not* recoverable (not subject to turning into exceptions or ignoring).

Debug builds are typically 2 or 3 times slower than release builds (this factor can vary a great deal depending on your program).

It is recommended programs be developed mostly with Debug builds, and transition to release builds more towards the end of a release cycle (but always use a mix of both).

Release Builds

Release builds have zero overhead from assertions. There is no runtime or space cost.

This is very important to understand, because it the zero cost of assertion checking in the final delivered product helps encourage more use of assertions (by removing the excuse that the check would make the program seem slow for users). And it contributes to why Stroika is a very high performance framework.

Copy by Value

Objects in Stroika are overwhelmingly copy-by-value in semantics, though often copy-by-reference internally, for performance reasons.

For example

```
String a = L"a";
String b = a;
b += L"a";
Assert (a == L"a");
```

This principle – copy by value almost everywhere – except where clearly marked by names to the contrary – helps make Stroika semantics visually obvious.

Ptr objects are something of an exception

There are a few kinds of objects in Stroika that violate this rule about copy-by-value. Some objects only make sense to share. For example, Sockets don't make sense to copy by value. They are intrinsically associated with network endpoints that cannot be duplicated, and so 'copies' are copies of the **pointer** to that object. To emphasize this, Stroika uses the naming convention of Ptr at the end of the name, or as the name, of such objects.

These objects often have methods of the thing they point to – with a variety of convenient overloads etc, but copying those objects logically just copies a reference (pointer) to the underlying shared object.

Examples of such things that are intrinsically (and named) 'Ptr' objects include:

- Thread (Thread::Ptr)
- Socket (Socket::Ptr, as well as ConnectionlessSocket::Ptr etc)
- InputStream::Ptr, OutputStream::Ptr, etc…

Iterator objects – halfway between by value and by reference/Ptr

Stroika also makes extensive use of Iterator objects – and these are logically copied by value, but they are also logically pointers 'into' some associated container.

SharedByValue<T>

A very helpful template class used internally in Stroika is SharedByValue<T>. You don't need to know about this, but it may be helpful in efficiently implementing by-value semantics with nearly by-reference performance impact.

SharedByValue implements 'copy-on-write', fairly simply and transparently. So you store your actual data in a 'rep' (letter part of letter-envelope pattern), and when const methods are accessed, you simply dereference the pointer. Such objects can be copied for the performance cost of copying a shared_ptr<> - fairly cheap. The only time you pay (a significant) cost, is when you mutate one of these objects (which is already shared) – then you do the copying of the data behind the object.

Const and Logical Const

Generally Stroika uses the idea of logical const for its objects, and freely uses mutable for fields to enforce that notion.

But there is one case where this is slightly vague, and at first glance, may appear not fully adhered to: Ptr objects.

Ptr objects are really combinations of two kinds of things – smart pointers – and short-hand accessors for the underlying thing.

Because of the C++ thread safety rules (always safe to access const methods from multiple threads at once so long as no writers, and the need for synchronization on writes) – and because these rules only apply literally and directly to the ‘envelope’ part – or the smart-pointer part of the object, we use the constness on Ptr objects to refer to the ptr itself, and not thing pointed to.

We arguably COULD get rid of PTR objects and just use shared_ptr<T> or shared_ptr<const T> - but then we would lose the convenience of having simple interfaces for reps, and more complex, overloading etc interfaces for calling.

Synchronization (thread safety)

One reason it's very important to understand what values are copy-by-value, and what are copy-by-reference, is because of understanding thread safety.

All of Stroika is built to be extremely 'thread safe', but automatically synchronizing all operations would create a high and almost pointless performance penalty.

Instead, Stroika mostly follows the C++ STL (Standard Template Library) thread safety convention, of having const methods always safe for multiple readers, and non-const methods ONLY safe with a single caller at a time.

But that only goes one level deep – the outer object you are accessing. For the special case of these 'Ptr' objects, the user must also worry about synchronizing the internal shared 'rep' objects. The way this is done varies from class to class, and look at the particular 'Ptr' classes you are using to see. For example, Thread::Ptr internal rep objects are always internally synchronized (meaning the caller only need worry about synchronizing the Ptr object). Stream internal rep objects are by default, *not* externally synchronized, but you can easily construct a synchronized internal stream with InputStream<>::Synchronize () – for example.

And to synchronize any c++ object, you can always use the utility template Synchronized<T> - to wrap access to the object. You can also use lock_guard<> etc, but Synchronized<> makes accessing shared data in a thread-safe way MUCH simpler and more transparent (but only synchronizes the 'envelope' – not the 'shared rep' of 'Ptr' objects.

See 'Thread Safety.md' for more details.

Ptr vs. Rep (e.g Streams) thread safety

In the several families of classes, such as Threads, Streams (InputStream, OutputStream etc), Sockets, and others using the letter-envelope paradigm, users must separately consider the thread safety of the letter and the envelope.

The envelope typically follows C++-Standard-Thread-Safety, but the thread safety rules applying to the letter (shared rep object) – depend on how that object was created. So see its Object::New () method for documentation on this.

Debug::AssertExternallySynchronized<T>

To help document, and to help ensure that Stroika classes are used in a thread safe manner, the helper class Debug::AssertExternallySynchronized<T> is used fairly consistently throughout Stroika to 'wrap' objects in a thread-safety-checking envelope. This has no performance cost (space or runtime) in release builds, but has a significant (roughly 2x slowdown) in debug builds.

But it means that if your code runs correctly (without assertion errors) in Debug builds, it's probably thread safe.

This doesn’t completely replace tools like thread-sanitizer, and valgrind/helgrind, but it does help provide simpler, and clearer diagnostics directly when you are running your threaded applications.

External Validation Tools

Tools like valgrind (helgrind and memcheck), and sanitizers (address, undefined behavior, and soon thread sanitizer) are all regularly run as part of the Stroika regression test suite, and are a sensible addition Stroika-based development process.

They are especially useful to help validate that any subtle bugs aren't present ONLY in release builds, but not in debug builds (extremely rare, but it can happen).

‘Quasi-namespace’

Why are Ptr objects ‘struct’ / ‘class’ instead of actual namespaces?

Things like ‘Stream’ or ‘Thread’ or ‘Socket’ – are just logical groupings

These logical groupings could have implemented using actual namespaces or just struct’s acting as ‘quasi’ namespaces.

Advantages of using ‘namespace’:

- This is logically the best fit
- There are a number of namespace specific features – inlining and importing namespaces to leverage

Advantages of using struct/class

- Namespaces offer no mechanism for private or protected access control, which for many of our uses is very helpful
- Namespaces cannot be templated (classes/structs can) – which makes clearer the grouping between classes (e.g. InputStream<char>::Ptr and InputStream<char>::_IRep more clearly related than InputStream::Ptr<char> and InputStream::_IRep<char>).
- Classes provide their own mechanism for automated related lookup (nested classes see members from parents) – which is helpful

In the end – no very strong arguments, but for now I’ve gone with ‘struct/class’