Coding Conventions

Formatting

- Run astyle script
  make format-code
  I'm not even slightly happy about the way this looks but I've found no better alternative.
- Spacing in header files
  - No whitespace before or after text of file
  - One space between leading include #define and includes
  - One space between related sections of #includes
  - 3 spaces between major file sections (before \file comment and after)
  - Inside namespaces (where code declared) – two spaces between major related declarations.
  -
  - EXAMPLE

```
/*
 * Copyright(c) Sophist Solutions, Inc. 1990-2013.  All rights reserved
 */
#ifndef _Stroika_Foundation_Configuration_Enumeration_h_
#define _Stroika_Foundation_Configuration_Enumeration_h_  1

#include    "../StroikaPreComp.h"

#include    "Common.h"



/**
 *  \file
 *
 * TODO:
 *      @todo - maybe stuff like Add(ENUM, ENUM), and DIFF (ENUM,ENUM) to
workouarnd
 *              issues with too-strong typing with enum class?? (avoid so many
casts)
 */



namespace   Stroika {
    namespace   Foundation {
        namespace   Configuration {


            /**
             *  \brief  Increment the given enumeration safely, without a bunch
of casts.
             *
             *      \req    ENUM uses  Define_Start_End_Count() to define eSTART,
eND
             *      \req    e >= typename ENUM::eSTART and e < typename
ENUM::eEND
```

```
            */
            template    <typename   ENUM>
            ENUM    Inc (ENUM e);


        }
    }
}


/*
 ********************************************************************************
 ***************************** Implementation Details ***************************
 ********************************************************************************
 */
#include    "Enumeration.inl"

#endif  /*_Stroika_Foundation_Configuration_Enumeration_h_*/
```

STL naming versus "Studly Caps" or "CamelCase"

I personally prefer the style "CamelCase" – probably because I first did object oriented programing in Object Pasca/MacApp – a few years back. Maybe there is another reason. But now it's a quite convenient – providing a subtle but readable visual distinction.

All (or nearly all) Stroika classes, and methods use essentially the same 'Studly Caps' naming styled from MacApp, except that we use 's' prefix for static variables. We use the 'f' prefix for data members. We use the 'k' prefix for constants. We use the 'e' prefix for enumerators.

However, STL / stdc++ - has its own naming convention (basically all lower case, and _), plus its own words it uses by analogy / convention throughout (e.g. begin, end, empty).

Stroika methods will start with an upper case letter, EXCEPT in the case where they method mimics for follows an existing STL pattern. If you see lower case, assume the function follows STL semantics. If you see CamelCase, you can assume it follows Stroika semantics.

For example:

> String:: SubString () follows Stroika semantics (asserting if values out of range).
>
> String::substr() follows the semantics of STL's basic_string<>::substr().

Note – this 'convention' doesn't replace documentation (the behavior of each method is documented). It just provides the user/reader a quick subtle convenient visual cue which semantics to expect without reading the docs.

Examples of common STL methods which appear in Stroika code (with STL semantics):

- clear()
- empty()
- push_back
- erase
- c_str()
- length()
- size()
- compare()
- begin() // sort of – usually using Stroika iterators
- end() // ditto

Begin/End versus start/length

STL is reasonably consistent, with most APIs using T* start, T* end, but some APIs use length instead of end. The Stroika convention is to always use T* start, T* end.

Rationale

One, this gives more consistent expectations. That's especially important for APIs that use offsets (like String) – so that it's obvious the meaning of integer parameters.

And it avoids problems with overflow. For example, if you had an API like:

```
basic_string substr(
   size_type _Off = 0,
   size_type _Count = npos
) const
```

To map this to an internal representation you have todo:

```
char* s = m_bufPtr + _Off;
char* e = m_bufPtr + _Off + _Count;
```

but if count was numeric_limits<size_t>::max(), then the e pointer computation would overflow. There are ways around this, but mixing the two styles creates a number of problems - but for implementations – and for use.

mk Factories

Stroika doesn't make much use of the factory pattern, but occasionally – it is useful. If the type provided by the factory is exactly the type of a given class, then we generally use

```
struct T {
    static T mk();
};
```

Of course in this case, there was little obvious motivation to use a factory instead of regular constructor. However, if the class T is effectively a smart-pointer wrapper on some underlying dynamic 'rep' – this pattern may make sense.

But – for shared_ptr types, and typedefs, we generally use

```
struct                      X;
typedef  shared_ptr<X>      XPtr;
XPtr                        mkXPtr ();
```

Compare () and operator<, operator>, etc…

For types Stroika defines, it generally uses the convention of providing a compare function:

```
int T::Compare (T rhs);    // sometimes with additional optional
                           // arguments for how to compare
```

and provides

bool operator<, operator<=, operator>, operator>=, operator==, operator!= which inline trivially maps to this.

Stroika code which COUNTS on comparison doesn't directly call Compare(), but instead uses 'a < b', etc. This applies to things like Stroika containers. The reason for this later choice include:

- Working with builtin types (e.g. in)
- Working with STL types, and 3rd-party libraries
- Probably more likely to seamlessly fit with user code

Note that we choose to use non-member operator overloads for these comparison functions because putting them in the namespace where the class is defined provides the same convenience of use (name lookup) as member functions, but allows for cases like C < O where C is some time convertible to O, and O is the class we are adding operator< support for.

So for example:

```
if (L"aa" < String (L"ss")) {
}
```

Works as expected, so long as either the left or right side is a String class, and the other side is convertible to a String.

Using T= versus typedef

C++11 now supports a new typedef syntax – using T=…. This is nearly the same as typedef in terms of semantics.

Stroika code will generally use the using T = syntax in preference to typedef for two reasons:

- The using = syntax is slightly more powerful, in that it supports defining derivative template typdefs.
- And more importantly, I believe it makes code more readable, because the type of INTEREST is the one being defined = which appears first. What it maps to is often more complicated (why we define the typedef) – and one can often ignore that detail (or skim it).