Coding Conventions

## Formatting

* Run AStyle script

RunAstyle.pl

I’m not even slightly happy about the way this looks but I’ve found no better alternative.

* Spacing in header files
  + No whatspace before or after text of file
  + One space between leading include #define and includes
  + One space between releated 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, eEND

\* \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\_\*/

## 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 member function operators for comparison – instead of global (namespace) functions (with two arguments) – because

* The namespace based ‘global’ operators only get overloaded if you import the entire namespace (or at least import those functions)
  + This is either very awkward to use or encourages namespace conflicts
* The namespace/global approach CAN lead to confusing conflicts of inappropriately colliding chained conversions

The downside of this approach is that stuff like:

if (L”aa” < String (L”ss”)) {

}

Fails. You must have the left most object be already a String (or other stroika) object. Sigh. Seems like the best compromise?

## 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).