Cpp0x

Development Plan

The following development plans apply to 4.3.x series of releases leading up to the 5.x major release tentatively scheduled sometime after the current C++ draft standard is ratified.

Configuration

The release distribution(s) will provide a configuration option that allows users to explicitly enable or disable support for C+0x library specifications and extensions. In 4.3.x releases, C+0x support is disabled by default if the option is not explicitly specified by the user.

If C+0x support is enabled, an additional configuration macro (or macros) will be defined within the library that specifically indicates whether C+0x specifications should be enabled. (The exact name of this macro and how and where it is defined is unspecified for purposes of this document.)

Affected Headers

New headers specified by the C++0x extensions will reside in the `$TOPDIR/include` directory (where `$TOPDIR` indicates the source distribution and/or installation directory) with the filenames specified by the standard. (Note especially, there is no `std::tr1` namespace or associated `tr1` directory.) Consequently, compiler search paths for header files are unchanged.

The new headers specified by the C++0x draft standard are shown in the following list:

	New C++0x Headers	
`<type_traits>`	`<random>`	`<tuple>`
`<array>`	`<unordered_set>`	`<unordered_map>`
`<regex>`	`<ccomplex>`	`<complex.h>`
`<cfenv>`	`<fenv.h>`	`<cinttypes>`
`<inttypes.h>`	`<cstdbool>`	`<stdbool.h>`
`<cstdint>`	`<stdint.h>`	`<ctgmath>`
`<tgmath.h>`		

Note, some of the new headers – the C headers with a `.h` suffix – are actually required by ISO/IEC 9899:1999 (a.k.a. C99) and consequently specified as part of the C+0x draft standard. While these headers are technically a part of the C+ standard library, they do not fall within the scope of this development plan.

Modified headers – existing headers for which changes are mandated by the standard – will contain the appropriate conditional guards (utilizing the configuration macro(s) specified above) to enable the implementation of requirements specific to the C+0x draft standard. The modified headers specified by the C+0x draft standard are shown in the following table:

	Modified C++0x Headers	
`<functional>`	`<memory>`	`<utility>`
`<complex>`	`<cctype>`	`<ctype.h>`
`<cfloat>`	`<float.h>`	`<ios>`
`<climits>`	`<limits.h>`	`<locale>`
`<cmath>`	`<math.h>`	`<cstdarg>`
`<stdarg.h>`	`<cstdio>`	`<cstdlib>`
`<stdlib.h>`	`<ctime>`	`<wchar.h>`
`<cwctype>`	`<wctype.h>`	

Note again, some of these headers – the C headers in particular – contain normative changes specified by C99 and are thus outside the scope of this development plan.

Components

The new C++0x features are classified according to components shown in the following list:

- General utilities library (Chapter 20, Clause [utilities])
    - Tuples (Section 3, Clause [tuple])

    - Type traits (Section 4, Clause [meta])
    - Function objects (Section 5, Clause [function.objects])
        - Reference wrappers (Section 5.5, Clause [refwrap])
    - Smart pointers (Section 6, Clause [memory])
- Numerical library (Chapter 26, Clause [numerics])
    - Random number generation (Section 4, Clause [rand])
- Containers (Chapter 23, Clause [containers])
    - Fixed-size arrays (Section 2.1, Clause [array])
    - Unordered associative containers (Section 4, Clause [unord])
- Regular expressions (Chapter 28, Clause [re])
- Atomic operations (Chapter 29, Clause [atomics])
- Thread support (Chapter 30, Clause [thread])

Note, this is not a complete list of C++0x features: these are only the components that fall within the scope of this development plan. Also, the last two components – atomic operations and multithread support – are not planned until the 5.0 release timeframe.

Tuples

Tuples are basically the same thing as `std::pair` except that tuples have a variable number of type parameters (or an implementation-defined number of parameters with default types) . The class template and associated `std` namespace members are sufficiently simple enough so that two implementations – one for compilers that support variadic templates and another for all other compilers that don't – can be written at the same time. Due to resource constraints however, only the variadic templates version will be implemented initially. The specification for tuples in the latest draft standard also assumes variadic templates. Consequently, a tuple implementation that does not utilize variadic template would be considered a library extension.

Tuples can have no type parameters; e.g., `std::tuple<>` is a valid type. Consequently, this particular tuple type should have no constructors that accept (or other members that operate on) any values (other than the value of the tuple itself).

Tuples with exactly two type parameters have conditional constructors and operators for conversions from values of the `std::pair` class template. This means the class template needs to be specialized when instantiated with exactly two types to define the additional members.

Issues

The current implementation places the `std::tuple` class template in the `<rw/_tuple.h>` header file. I'm not convinced that an internal tuple, i.e. `_RW::+rw_tuple`, is needed though if so, it would be placed in this header and the standard tuple would be moved out of this header into the standard `<tuple>` header. (That's a lot of essentially duplicated constructors due to three different tuples – one internal tuple, one standard generic tuple, and one standard pair tuple – if this proves to be the case.) Should probably also move the internal `+rw` namespace members from the standard `<tuple>` header to the internal header.

The tuple specialization for two element types is currently essentially a duplicate of `std::pair`. This might not be the most appropriate solution (due to additional helper specializations) and may prove that the internal tuple mentioned above really is needed.

The tuple_cat() function overloads are REALLY tricky to implement.

The tuple tests programs are trivial. Currently they only contain a bare minimum of rudimentary tests to verify basic functionality. (Need to become more familiar with general test patterns for various C++ constructs and features explified by more exhaustive test programs. For some C++0x features, these patterns have not yet been explored.)

Online Resources

- ISO/IEC TR 19768: C++ Library Extensions TR1. The original draft specification. (Now obsolete since modifications have been made in the latest C++ draft standard.)
- Technical Report 1 (Wikipedia). General overview of the TR1 extensions.