ISO/IEC TR 19768

C++ Standard Library
Technical Report 1

What *is* Library TR1?

- Report on useful extensions to C++ standard library
- Non-normative
  - No requirement to implement it
  - No requirement to implement all of it
- Not a mandate for the next standard ...
- ... although existing practice is much easier to standardise
- Breaking News from Berlin: TR1 was adopted for the next language standard, minus one component

Goals

- Improve library usage
- Improve library writing
- Extend the reach of the library
- Improve C compatibility

Goals

- Improve library usage
  - Fix function binders
  - Container friendly smart pointer
  - Refine iterator categories / algorithms
- Improve library writing
  - Template metaprogramming aids
  - Iterator framework
- Extend the reach of the library
  - New containers
  - New problem domains
- Improve C compatibility
  - Adopt missing C99 library functions

Contents

- Utilities for general users
  - General purpose smart pointer
  - More containers
  - Function wrappers
  - C99 Standard Library
- Tools for library writers
  - Type traits
  - Tuples
- Domain specific
  - Regular Expressions
  - Random number generator
  - Engineering/scientific math functions

Implementation details

Backwards compatible

- All new names declared in namespace `std::tr1`
- Or even a nested namespace below that
- Must actively enable TR1 in some implementation defined manner
  - Separate headers and search paths for tr1
  - Macros to conditionally include TR1 code
  - Etc.
- In general, provide no-throw swap for classes

User Oriented Components

C99 library

Array ‘container’

Hashing containers

Reference counted smart pointer

Improved function binders

Function-wrapper

C99 Library Compatibility

Language compatibility conventions:

- long long -> typedef _Longlong
- unsigned long long -> typedef _ULonglong
- **restrict** keyword dropped from all function declarations
- Function-like macros wrapped as function templates
- Floating point functions overloaded on ‘primary’ name, as well as providing suffixed version

C99 Library Compatibility

- New headers — *Mostly* new typedefs and macros
  - <ccomplex> / <complex.h> — Forward to <complex>!
  - <cfenv> / <fenv.h> — FPU control e.g. rounding modes
  - <cinttypes> / <inttypes.h> — <cstdint> + a utility functions
  - <cstdbool> / <stdbool.h> — macro `__bool_true_false_are_defined`
  - <cstdint> / <stdint.h> — fixed size integer types eg uint_fast32_t
  - <ctgmath> / <tgmath.h> — <ccomplex> + <cmath>
- Enhanced headers
  - <cctype> <cfloat> <climits> <cmath> <cstdarg>
  - <cstdio> <cstdlib> <ctime> <cwchar> <cwtype>
  - <complex> <ios> <locale>

C99 Library : notable headers

- <complex>
  - Trig functions — asin acos atan
  - Hyperbolic functions — asinh acosh atanh
  - Complex manipulation — norm arg imag real conj
- <cmath>
  - trig, hyperbolic, exponential — acos, asinh, exp log10 …
  - Scientific — erf, lgamma, tgamma …
  - floating point manipulation — nextafter, nexttoward, fmod …
- <cstdio>
  - New format flags for printf family of functions

Language features that may improve this component : C99 Library

- (unsigned) long long type
- restrict keyword
- __func__ identifier for assert macro

array

- Declare size with type, not object
```
std::tr1::array< int, 5 > x;
```
- Familiar container-like interface
```
std::find( a.begin(), a.end(), 3 );
```
- No pointer decay, so we keep type information
- No pointer decay, can pass by-value or by-reference
- As-efficient as a ‘regular’ array, for efficient compilers.
- data() member to pass to C APIs

array : definition

```
template <class T, size_t N >
struct array {
        // types:
        typedef T &                                                 reference;
        typedef const T &                           const_reference;
        typedef implementation defined                              iterator;
        typedef implementation defined                              const_iterator;
        typedef size_t                              size_type;
        typedef ptrdiff_t                           difference_type;
        typedef T                                   value_type;
        typedef std::reverse_iterator<iterator>     reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        T elems[N]; // Exposition only

        // No explicit construct/copy/destroy for aggregate type
        void        assign(const T& u);
        void        swap( array<T, N> &);

        // iterators:
        iterator                begin();
        const_iterator          begin() const;
        iterator                end();
        const_iterator          end() const;
        reverse_iterator        rbegin();
        const_reverse_iterator  rbegin() const;
        reverse_iterator        rend();
        const_reverse_iterator  rend() const;

        // capacity:
        size_type  size() const;
        size_type  max_size() const;
        bool       empty() const;

        // element access:
        reference               operator[](size_type n);
        const_reference         operator[](size_type n) const;
        const_reference         at(size_type n) const;
        reference               at(size_type n);
        reference               front();
        const_reference         front() const;
        reference               back();
        const_reference         back() const;
        T *                     data();
        const T *               data() const;
};
```

array in action

```
#include <array>
#include <functional>
#include <iostream>
#include <ostream>
using namespace std;

template< class T, size_t N >
void WriteArray( const tr1::array< T, N > & a, std::ostream & os ) {
  copy( a.begin(), a.end(), ostream_iterator< T >(os, “, ”) );
  os << endl;
}

template< class T, size_t N >
void TestArray( tr1::array< T, N > a, std::ostream & os ) {
  transform( a.begin(), a.end(), a.begin(), tr1::bind( plus< T >, _1, _1 ) );
  copy( a.begin(), a.end(), ostream_iterator< T >(os, “, ”) );
  os << endl;
}

int main() {
    tr1::array< int, 5 > x = { 1, 2, 3, 4, 5 };
    WriteArray( x, cout );                  // 1, 2, 3, 4, 5,
    TestArray( x, cout );                   //  2, 4, 6, 8, 10,
    WriteArray( x, cout );                  //  1, 2, 3, 4, 5,
}
```

Dangers of misuse : array

- No default initialization for POD types

```
std::tr1::array< double, 5 > x;
std::tr1::array< double, 5 > y = x;
```

- Empty arrays have front and back members

```
std::tr1::array< int, 0 > x = {};
if( !x.empty() ) { return x.front(); }
```

- NOT a container
  - no push/pop/insert members
- Aggregate type means no user declared constructors
  - hard to initialize

```
class test {
private:
    std::tr1::array< double, 5 > x;
public:
    test() : x()  {  // value initialization is the best we can manage
        std::fill( x.begin(), x.end(), 42 );
    }
};
```

Language features that may improve this component : array

- Concepts
  - Will ensure only ‘usable’ types are stored
  - Will type-check when used inside another template
- New initialization syntax
  - Will be able to initialize array objects as data members

Hashing containers

What's in a name?

'unordered associative containers'

- unordered_set
- unordered_map
- unordered_multiset
- unordered_multimap

Why?!

- 'obvious' names hash_set etc. already taken
- Emphasises interface over implementation

Hashing container requirements

- key type must support a hashing function
- Complexity guarantees on element lookup and removal
  - ‘average case’ constant time
  - worst case linear with size of container
- Erase operations will not invalidate iterators
- Insert operations invalidate iterators if-and-only-if they force a rehash
- Operations offer strong exception safety guarantee if hash function throws, often strengthened to no-throw otherwise
- Cannot compare containers
  - No support for operators ==, <, <=, >=, >

Hashing

- #include <functional>
- Standard hash functor

```
template <class T>
struct hash : public std::unary_function<T, std::size_t>
{
  std::size_t operator()(T val) const;
};
```

- Support for:
  - Integral types
  - Floating point types
  - Pointer types
  - std::string / std::wstring

Sets

#include <unordered_set>

unordered_set stores unique objects

unordered_multiset allows duplicates

Main operations

- insert / erase / find
- load_factor / max_load_factor / rehash

Maps

#include <unordered_map>

unordered_map requires unique keys

unordered_multimap allows duplicate keys

Stores std::pair< key_type, value_type >

Main operations

- insert / erase / find
- only multi_map supports operator[]
- load_factor / max_load_factor / rehash

Dangers of misuse

- Bad hashing functions
  - Good hashes are vital, but hard to write
  - Research good algorithms rather than invent
  - Google is your friend!
- Applying to problems where ordering matters
  - Use ‘classic’ set or map instead
- Passing iterators to algorithms requiring sorted ranges
  - E.g. lower_bound,
- Assuming all duplicate values in a multi-container will form a single range
  - Iterator ranges yield all objects that hash to the same bucket

Language features that may improve this component : hashing containers

- Concepts
  - Will ensure only ‘usable’ types are stored
  - Will type-check when used inside another template
- Move semantics

shared_ptr

Designed to solve many of the problems associated with std::auto_ptr in an efficient package.

- Can be safely stored in a container
- Can store ‘incomplete types’
- Can handle arrays or objects allocated in a different heap e.g. on a DLL

shared_ptr

Solves many of the problems addressed by policy-based frameworks, with less baggage

- Runtime customisation of deleter ‘policy’
- Simple interface without waiting for typedef templates
- Single pointer type for all policies, for convenient use in APIs (the *vocabulary* problem)

shared_ptr by example

- Template declared in header <memory>
- Useful return type for factory functions

```
std::tr1::shared_ptr< interface > > MakeObject()
```

- Perfect to store in containers

```
std::vector< std::tr1::shared_ptr< MyClass > >
```

- Handles incomplete types, so perfect to implement the pImpl idiom

```
class Facade {
public:
  // interface details
private:
  struct IMPL;
  std::tr1::shared_ptr< IMPL > m_pImpl;
};
```

- shared_ptr< void > makes an interesting ‘handle’ type

```
std::tr1::shared_ptr< void > MyClass::Lock() {
  return std::tr1::shared_ptr< void >( new Lock( m_mutex ) );
}
```

shared_ptr by example

Custom deleter can handle arrays

```
struct delete_array {
  template< typename T >
  void operator()( T * target ) const {
    delete [] target;
  }
};

std::tr1::shared_ptr< double > x( new double[ 42 ], delete_array() );
```

Custom deleter can handle special allocation/release requirements, eg. COM

```
std::tr1::shared_ptr< IUnknown > x( factory.Create()
                                  , std::tr1::mem_fn( &IUnknown::Release )
                                  );
```

weak_ptr

- Intrusive solution for circular references
- When you think two shared_ptr objects may be mutually dependent, make one of them a weak_ptr

```
std::tr1::shared_ptr< MyClass > x( new MyClass );
std::tr1::weak_ptr< MyClass > wptr( x );
```

- weak_ptr shares the same ‘control block’ as the referenced shared_ptr, but has no ownership
- lock the weak_ptr to obtain a shared_ptr when you need to access the target object
- lock returns an empty shared_ptr if the shared count has been reduced to zero

```
if( std::tr1::shared_ptr< MyClass > y = wptr.lock() ) {
   // Can now use y safely
}
```

Dangers of misuse : shared_ptr

- Circular references
  - Solution is use weak_ptr …
  - … when you are aware you have the problem
- Multiple shared_ptrs owning same object
  - Don’t do this!
- Storing arrays without special deleter
  - Use delete_array class or similar
- Holding memory allocated in a different memory manager
  - E.g. allocated inside a dll
  - Store a deleter that releases object back to original memory manager

Language features that may improve this component : shared_ptr

- Right angle bracket hack
  ```
  vector<shared_ptr<MyType>>
  ```
- Move Semantics
  - Will make copies as efficient as auto_ptr
  - Thread-safe copying without spinlocks!

bind

"A function that returns another function"

- Extensible replacement for std::bind_1st / std::bind_2nd
- Simple to use syntax
- Returns 'adaptable functors' as per STL specification
  - Will derive from std::unary_function / std::binary_function when appropriate

bind by example

```
std::tr1::array< int, 5 > data = { 13, 42, 5, 69, 8 };
Std::find( data.begin()
         , data.end()
         , std::bind1st( std::greater<int>(), 33 )
         );

std::find( data.begin
         , data.end()
         , std::tr1::bind( std::greater< int >(),  _1, 42 )
         );

std::transform( data.begin()
              , data.end()
              , data.begin()
              , bind( multiplies<int>()
                    , bind( plus<int>(), _1, 2)
                    , 3
                    )
              );
```

Reference Wrapper

- Created to solve ‘reference to reference’ problem
- Also delivers ‘rebindable’ reference objects
- Callable if reference to a function type, pointer to function, or other callable object
- Factory functions to create wrapper
  - #include <functional>
  - ref(T &) creates a wrapper for reference
  - cref(const T &) wraps a reference-to-const
- Danger : be careful not to hold references beyond an object’s lifetime

mem_fn

Simpler and more flexible replacement for:

- std::mem_fun
- std::mem_fun_t
- std::mem_fun1_t
- std::const_mem_fun_t
- std::const_mem_fun1_t
- std::mem_fun_ref
- std::mem_fun_ref_t
- std::mem_fun1_ref_t
- std::const_mem_fun_ref_t
- std::const_mem_fun1_ref_t

mem_fn

- All can be replaced by std::tr1::mem_fn
- Also handles member functions with more than one argument
- Can also store pointer-to-data-member

```
struct Demo {
 void DoIt() { std::cout << “Doing it now” << std::endl; }
 void Echo( int arg ) { std::cout << arg << std::endl; }
};

std::vector< Demo > v( 25 );

std::for_each( v.begin(), v.end(), std::mem_fn( &Demo::DoIt ) );
std::for_each( v.begin(), v.end(), std::bind( &Demo::Echo, _1, 42 ) );
```

Language features that may improve this component : enhanced function binders

- Lambda would almost entirely replace the need
- Reference-collapsing solves many reference-to-reference issues
- Concepts to define ‘callable’

function

std::tr1::function is to callbacks what smart pointers are to regular pointers

- value type
- strongly typed
- Stores anything matching the callable interface
- Returns ‘adaptable functors’ as per STL specification
  - Will derive from std::unary_function / std::binary_function when appropriate
- Interesting syntax ...

```
std::tr1::function< void( int, double ) > x;
```

Function : example

An ‘undo’ facility using tr1::function to implement the Command pattern

```
#include <functional>

typedef function< void() > Action;
typedef std::pair< Action, Action > Command;

std::stack< Action > undo_list;
std::stack< Action > redo_list;

void Do( Command cmd ) {
  cmd.first();
  undo_list.push( cmd );
  clear( redo_list );
}

void Undo() {
  assert( !undo_list.empty() );
  Command cmd = undo_list.top();
  cmd.second();
  undo_list.pop();
  redo_list.push( cmd );
}

void Redo() {
  assert( !redo_list.empty() );
  Command cmd = redo_list.top();
  cmd.first();
  redo_list.pop();
  undo_list.push( cmd );
}
```

Function : example

An implementation of the ‘observer’ pattern

```
using namespace std::tr1::placeholders;
typedef std::tr1::function< void() > Event;

std::vector< Event > callbacks;

// Note upper-case ‘R’ or we are using a keyword
void Register( Event ev ) {
  callbacks.push_back( ev );
}

void call( Event ev ) {
  ev();
}

void FireEvents() {
  std::for_each( callbacks.begin()
               , callbacks.end()
               , tr1::bind( call( _1 ) )
               );
}
```

Language features that may improve this component : function objects

- Rvalue references and ‘perfect forwarding’
- Variadic templates
- Lambda

An Integrated Example

- Storing a function object as a shared_ptr custom deleter, to log each release

```
template< typename T >
void LoggedDeleter(T * target, ostream & os )
{
  os << “Logging Customer deleter” << endl;
  delete target;
}

template< typename T >
tr1::shared_ptr< T > MakeLoggedPtr( ostream & os )
{
  tr1::function< void( T * ) > deleter = tr1::bind( LoggedDeleter< T >
                                                  , _1
                                                  , tr1::ref( os ) );
  return tr1::shared_ptr< T >( new T(), deleter );
}
```

Library Oriented Components

(Library authors are users too!)

Type Traits

result_of

Tuples

Type Traits

- A library interface to compiler information
  - Implemented in header <type_traits>
- Mostly implementable as metafunction templates today ...
  - E.g. is_reference
- ... but a few will require compiler magic
  - E.g. is_union
- type traits support:
  - Deducing information about a type
  - Relationships between type
  - Transforming or manipulating types

Type Traits are metafunctions

- Metafunctions are the compile-time equivalent of runtime functions
- A way to perform a computation at compile time
- Typically, metafunctions use types as arguments, where runtime functions use objects
- Arguments passed as template parameters, results are typedefs

```
template < typename T >
struct identity {
  typedef T type;
};
```

- Always return types, even when numeric values are expected!

Type Traits : integral_constant

- A unique type for each value

```
template <class T, T v>
struct integral_constant
{
 typedef T value_type;
  typedef integral_constant<T,v> type;
  static const T value = v;
};
```

- Predefined types

```
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

Type Traits : type categorisation

Primary type categories:

is_void
is_integral
is_floating_point
is_array
is_pointer
is_reference
is_member_object_pointer
is_member_function_pointer
is_enum
is_union
is_class
is_function

Composite type categories:

is_arithmetic
is_fundamental
is_object
is_scalar
is_compound
is_member_pointer

Type traits : type information

type properties:

is_const
is_volatile
is_pod
is_empty
is_polymorphic
is_abstract
has_trivial_constructor
has_trivial_copy
has_trivial_assign
has_trivial_destructor
has_nothrow_constructor
has_nothrow_copy
has_nothrow_assign
has_virtual_destructor
is_signed
is_unsigned
alignment_of
rank
extent

type relations:

is_same< T, U >
is_base_of< T, U >
is_convertible< from, to >

Type traits : transformations

const-volatile modifications:
remove_const
remove_volatile
remove_cv
add_const
add_volatile
add_cv

reference modifications:
remove_reference
add_reference

array modifications:
remove_extent
remove_all_extents

pointer modifications:
remove_pointer
add_pointer

other transformations:
aligned_storage

Type Traits : example

Optimised copy algorithm

```
template < typename T >
void copy( T * begin, T * end, T * dest )  {
  if( std::tr1::type_traits::is_POD< T >::value )  {
    std::memcpy( dest, begin, (end-begin) * sizeof( T ) );
  }
  else  {
    while( begin != end ) {
      *dest = *begin++;
    }
  }
}
```

Optimal version would use is_POD to specialize function template, and deduce at compile time

Language features that may improve this component : type traits

- Concepts!
  - Type traits are generally NOT concepts
  - Concepts will find type traits extremely useful

result_of metafunction

If F is not a function object defined by the standard library, and if either the implementation cannot determine the type of the expression `f(t1, t2, ..., tN)` or the expression is ill-formed, the implementation shall use the following process to determine the type member:

1. If F is a function pointer or function reference type, type shall be the return type of the function type
2. If F is a member function pointer type, type shall be the return type of the member function type
3. If F is a possibly cv-qualified class type with a member type `result_type`, type shall be `typename F::result_type`
4. If F is a possibly cv-qualified class type with no member named `result_type` or if `typename F::result_type` is not a type:
   a) If N=0 (no arguments), type shall be `void`
   b) If N>0, type shall be `typename F::template result<F(T1, T2,..., TN)>::type`
5. Otherwise, the program is ill-formed

tuple

- Listed in TR1 under containers
  - ‘A container of heterogeneous values’
  - Container size is fixed at compile time
  - As are the value types
- Essentially an anonymous struct type
  - Elements accessed by index not name
  - typedef the tuple if want to name for specific use
- Very useful for generic code, in order to construct types on demand

tuple interface

The main interface consists of free functions, and metafunctions

```
typedef std::tr1::tuple< int, double, MyClass >  Example;
```

- `tuple_size` metafunction
  - Returns number of elements in tuple

```
tuple_size< Example >::value == 3;
```

- `tuple_element` metafunction
  - Returns type of element at index

```
tuple_element< 1, Example >::type is double
```

- `get` element accessor
  - Free function returns value of element at index

```
Example ex( 1, 2.0, MyClass() );
double d = std::tr1::get<2>( ex );
```

- `make_tuple` factory function
  - Overload set of factory functions to create a tuple based on arguments
  - Template type deduction will make the right tuple type for you

```
Example ex = std::tr1::make_tuple( 1, 2.0, MyClass() );
```

- tie
  - Bind existing variables into a tuple (by reference)

Extending tuple interface

- std::pair is a tuple of length 2
- std::tr1::array is a homogeneous tuple
- As tuple interface is a collection of free functions, easy to overload those for pair and array types

```
map< Key, Value >::iterator SetValue( Key k, Value v, map< Key, Value > & dest ) {
  bool inserted;
  map< Key, Value >::iterator result;
  tie( result, inserted ) = dest.insert( make_pair( k, v ) );
  if( !inserted ) {
    throw runtime_error( “failed!” );
  }
  return result;
}
```

Language features that may improve this component : tuple

- variadic templates
- rvalue references and move semantics
- Improved initialization syntax

Domain Specific Libraries

Scientific / Engineering math functions

Random Number Framework

Regular Expression parser

Special Math Functions

- <cmath> covers basic math
- C99 advances just short of undergraduate level
  - trigenometic sin, cos, tan, asin, acos, atan, atan2
  - hyperbolic sinh, cosh, tanh, asinh, acosh, atanh
  - exponential exp, exp2, frexp, ldexp, expm1
  - logarithmic log10, log2, logb, ilogb, log1p
  - power pow, sqrt, cbrt, hypot
  - ‘special’ erf, erfc, tgamma, lgamma
- TR1 special math adds advanced functions found useful in Scientific and Engineering applications
- List of functions draws on existing ISO standard
  - ISO:31 Quantities and units

Special Math Functions

- Interface chosen to be compatible with C
  - declared as overloads, not templates
  - suffixed names according to data type
    - expint(double)
    - expintf(float)
    - expintl(long double)
  - For C++, additional overloads on primary name
    - expint(float)
    - expint(long double)
- Will also be available as a C Technical Report
- C++ and C committee working in parallel on this proposal

Special Math Functions

assoc_laguerre assoc_legendre beta

comp_ellint_1 comp_ellint_2 comp_ellint_3

conf_hyperg cyl_neumann

cyl_bessel_i cyl_bessel_j cyl_bessel_k

ellint_1 ellint_2 ellint_3

expint hermite hyperg

Laguerre legendre riemann_zeta

sph_bessel sph_legendre

sph_neumann

Dangers of misuse : Special Math Functions

- The domain over which these functions give useful answers is more limited than it appears from signatures ...
- ... but is certainly useful when constrained by real-world problems.
- Always read the friendly manual!

Random Number Framework

- Good random number generators are hard to write
- Even harder to validate, without a clear specification to measure against
- TR1 provides a framework for generating random sequences
- Engines produce a random sequence of bits
- Distributions provide a ‘shape’ to the numbers emerging from the sequence
- variate_generator links a generator to a distribution

Random Number Engines

- Engines are the source of randomness
- Can be created with an initial ‘seed’ to guarantee repeatable (testable) sequences
- A variety of well-known engines supplied in TR
  - linear_congruential
  - mersenne_twister
  - subtract_with_carry
  - subtract_with_carry_01
- Additional engines adapt others
  - discard_block
  - xor_combine
- Support for external hardware as true source of randomness
  - random_device

Engines with predefined parameters

9 predefined engines with known-to-be-good characteristics

- typedef linear_congruential< *implementation-defined*, 16807, 0, 2147483647> minstd_rand0;
- typedef linear_congruential< *implementation-defined*, 48271, 0, 2147483647> minstd_rand;
- typedef mersenne_twister< *implementation-defined*, 32, 624, 397, 31, 0x9908b0df, 11, 7, 0x9d2c5680, 15, 0xefc60000, 18> mt19937;
- typedef subtract_with_carry_01<float, 24, 10, 24> ranlux_base_01;
- typedef subtract_with_carry_01<double, 48, 10, 24> ranlux64_base_01;
- typedef discard_block<subtract_with_carry< *implementation-defined*, (1<<24), 10, 24>, 223, 24> ranlux3;
- typedef discard_block<subtract_with_carry< *implementation-defined*, (1<<24), 10, 24>, 389, 24> ranlux4;
- typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 223, 24> ranlux3_01;
- typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 389, 24> ranlux4_01;

Random Number Distributions

Distributions ‘shape’ the sequence of numbers, and may turn integral values from engine into floating point values

- uniform_int
- uniform_real
- Bernoulli
- binomial
- geometric
- Poisson
- exponential
- gamma
- Normal

Random Number Distributions

Link Engine to Distribution with variate_generator functor

```
template<class Engine, class Distribution>
class variate_generator {
public:
  typedef typename Distribution::result_type result_type;
  // … and other typedefs for distribution type, engine type etc.

  variate_generator(engine_type eng, distribution_type d);

  result_type operator()();
  // … and further access functions to ask about generator
   properties
};
```

Dangers of misuse : Random Number Framework

Using non-standard engines

- **Very** easy to produce a bad engine
- This is a problem domain for true experts
- Although it might be educational to experiment...
- ... never let such experiments leak into a production system

Random Numbers in C++0x

- Simplified API to connect engine to distribution
  - variate_generator is gone
- Separate Random Number Engine Adapter concept
- More Distributions
  - negative binomial
  - Weibull
  - extreme value
  - lognormal
  - chi-squared
  - Cauchy
  - Fisher’s F
  - Student’s t
  - histogram
  - ...

Regular Expressions

- Possible the most useful feature of TR1
- (after shared_ptr)
- Also the largest feature of TR1
- Would take another session to describe completely ,so I won’t even try

Regular Expressions

- Regular expressions are interpreted programs in a text parsing language
- Actually, a family of parsing languages
  - ECMAScript
  - basic
  - extended
  - awk
  - grep
  - egrep
  - Set
- Language describes rules for pattern matching and searching text

Applying regular expressions

- Build a parser object:

```
std::tr1::regex gearbox("~?GEARBOX/GEARS/GEAR_[0-9]/RATIO");
```

- Use it to search text, or replace values

```
std::string sTest( “this should fail” );
if( !std::tr1::regex_match(sFieldName.begin(), sFieldName.end(), gearbox) )
{  // Failed regex comparison
    throw std::runtime_error("'" + sTest + "' - not a gear ratio field");
}
```

- Library template depends on char type and char traits, as per iostream
- Standard typedefs regex/wregex, analogue to std::string / std::wstring

Where can I get it?

- Dinkumware
  - Tested and available ‘Real Soon Now’
- Boost
  - Boost lib 1.34 in final testing now
  - TR1 implementation on top of existing lib
  - No math functions, C99, unordered containers
- GCC/libstd++
  - Available since 4.1
  - No random or regular expressions yet
  - Math functions a work-in-progress

Where can I learn more?

"The standard is not a tutorial"

Books

- Pete Becker (out this Summer)
  The C++ Standard Library Extensions: a Tutorial and Reference
- Bjorn Karlsson (out now!)
  Beyond the C++ Standard Library : An Introduction to Boost

Online

- TR1 available in PDF from ISO web page
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf
- Boost documentation
  http://www.boost.org/libs/libraries.htm
- Active newsgroups/mailing lists
  - Boost at gmane
  - Comp.lang.c++.moderated
- Dinkumware
  - Will supply online docs when TR1 product is officially released

Technical Report 2?

Focussed more on application developers than library builders

- Portable file system access
- Threads
- Networking API
- Consistent reporting of system errors.
- Consistent use of string types
- Date and time
- ‘Optional’ Values
- ‘Any’ Values
- Value conversion
  - Numeric cast
  - Lexical cast
- Interval arithmetic
- Infinite precision integer
- Range-types and algorithm overloads
- String processing algorithms
- ‘missing’ algorithms e.g. copy_if, minmax, mean, variance

Open Questions

- Will it ship before or after the next language standard?
- Can/Should we use language featured only found in the next standard?

Deadline : New proposals must be evaluated at the next meeting in Portland, 15 – 20 October 2006

How can I get involved?

- Join your Standards Body
  - BSI standards@acu.org
  - ANSI
  - DIN
- Read the papers

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- Boost www.boost.org
- comp.std.c++