Doc No:	N1756=05-0016
Date:	2005-01-17
Reply to:	Matt Austern
	austern@apple.com

Library Extension Technical Report — Issues List

Revision 7: January 2005 midterm mailing

1 TR Introduction issues

1.1 How to disable TR features

Section: 1 [tr.intro]
Submitter: Matt Austern
Status: NAD

The TR says that implementers should not enable the TR by default, and should hide TR features more thoroughly than just putting them in another namespace. It's vague on exactly what implementers should do: have files in another directory (perhaps even shadow headers, like an alternate version of <functional>), or use a macro, or something else. Should we be more specific?

Resolution:
The LWG decided that the current text is satisfactory.

1.2 Feature test macros for the TR

Section: 1 [tr.intro]
Submitter: Beman Dawes
Status: Closed

How can users determine whether or not a particular compiler/library implementation supports the components described in the library extension TR? Should we have a coarse-grained macro (yes or not), or should we have a fine-grained facility so users can perform feature tests for individual pieces? (See http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1558.html)

Rationale:
There was some discomfort with putting these macros in a header that's associated so strongly with the operating system. Additionally, it isn't clear whether these macros would give us as much of a portability benefit as one might hope, and they might present a legacy issue. (If code has come to rely on their presence, there might be pressure to keep them in even after these components have been standardized.)

1.3 Reference to clause 17 should be stronger

Section: 1 [tr.intro]
Submitter: Beman Dawes
Status: TR

The TR working paper makes clear that the "Methods of description" (17.3) of the C++ Standard also apply to the TR.

But it appears to me that all of clause 17 should apply to the TR.

Proposed Resolution:
Replace:

> 1.1 Method of description [tr.description]
>
> The structure of clauses in this technical report, the elements that make up the subclauses, and the editorial conventions used to describe library components, are the same as described in clause 17.3 of the C++ standard.

with:

> 1.1 Relation to C++ Standard Library Introduction
>
> Unless otherwise specified, the whole of the ISO C++ Standard Library introduction (clause 17) is included into this Technical Report by reference.

1.4 Meaning of “impure extension”

Submitter: Pete Becker
Status: TR

TR1 asserts that "additions to standard library components" are "impure" extensions, and cites as an example the additions to std::pair in tr.tuple.pairs. For example, TR1 adds tuple_element<0, pair<T1, T2> >, which extracts the type T1.

On the other hand, operator<<(basic_ostream<...>, tuple<...>) is not an "impure" extension, even though it is an addition to basic_ostream.

I don't see the difference between these two: in both cases, if you use code from TR1 you can do things that you can't do with something from the current standard library. More important, I don't see the point of this distinction: what should I do differently when something is labeled an "impure" extension versus when it's labeled a "pure" extension? Looks to me like we should remove this distinction. (especially since these "additions" to pair are the only things where this distinction is applied).

Proposed resolution
Remove paragraph [tr.intro.ext]/2 ("The first three categories are collectively ...")

Change [tr.tuple.pairs]/1 from:

This is an impure extension (as defined in section 1.2) to the standard library class template std::pair.

to:

These templates are extensions to the standard library class template std::pair.

2 Smart pointer issues

2.1 shared_ptr constructor from auto_ptr missing postcondition

Submitter: Beman Dawes
Section: 2.2.3 [tr.util.smartptr.shared.const]
Status: TR

For

```
template <class Y> shared_ptr<Y>(auto_ptr<Y> & r)
```

The Postcondition clause says:

```
use_count() == 1
```

Resolution:
change it to

```
use_count() == 1 && r.get() == 0
```

2.2 Error in shared_ptr constructor

Submitter: Pete Becker
Section: 2.2.3 [tr.util.smartptr.shared.const]
Paper: c++std-lib-11461, 11463-11510, 11512-11524
Status: Closed

template<class Y> explicit shared_ptr(Y *p);
template <class Y, class D> shared_ptr(Y *, D d);
template <class Y> shared_ptr<auto_ptr<Y> & r);

The Effects clauses for the first two ctors say:

Constructs a shared_ptr that owns the pointer p [and the deleter d].

and their Postconditions clauses say:

use_count() == 1 && get() == p

Similarly, the Effects clause for the third ctor says:

Constructs a shared_ptr that stores and owns r.release()

and the Postcondition clause says:

use_count == 1

Issues:

- Is this the correct behavior when p or r.release() is a null pointer? Consistency with the default constructor would suggest that use_count() == 0 for a null pointer, i.e. the result is an empty shared_ptr.

- If use_count() should be 0, this raises the lesser issue of whether smart_ptr(null, Dtor) should remember _Dtor, or should be equivalent to smart_ptr(). I'm pretty sure I prefer the latter, 'cause it's the way I've implemented it. (It's also simpler and more efficient to treat all null pointers the same way).

Resolution:
Discussed at Kona. There are several ways of phrasing this issue: Do we reference-count null pointers? Are null pointers a special case? What is the deleter argument good for? There wasn't consensus for changing what the TR already says, but it was agreed that this exposed another issue (2.3, see below).

2.3 shared_ptr equality and operator<

Submitter: Beman Dawes
Section: [tr.util.smartptr.shared]
Status: Closed

When two shared_ptrs p1 and p2 are constructed from the same underlying pointer, the behavior of operator== and operator< is surprising. We will have p1 == p2, but also either p1 < p2 or p1 > p2. We thus violate the usual trichotomy condition. For example, if you have a whole bunch of shared_ptrs in a set, you can't search for it by constructing a new shared_ptr.

It may seem that this is irrelevant because it's never correct to have two shared_ptrs with the same underlying pointer, but that's wrong. It's valid in two cases: (1) when the underlying pointer is null; or (2) when you're using a user-defined deleter object that doesn't do deletion.

Further discussion: See N1590=04-0030, "Smart Pointer Comparison Operators", for a justification of the current behavior.

Rationale:
General agreement that this behavior is surprising. Also general agreement that (a) we're scared of violating Peter Dimov's recommendation; and (b) the proposed change hasn't been tested; and (c) the surprising behavior is only in a few corner cases. Straw poll 4-1-1: keep as is vs do the comparison based on the underlying pointer vs eliminate comparison operators entirely

2.4 shared_ptr::operator<() not a strict weak ordering

Submitter: Joe Gottman
Status: TR

According to the draft Technical Report on Standard Library Extensions, two shared_ptr's are equivalent under the !(a < b) && !(b < a) relationship if and only if they share ownership. But an empty (default constructed) shared_ptr does not share ownership with anything, not even itself. This means that if a is an empty shared_ptr, it will not be equivalent to itself, so operator< is not a strict weak ordering. The same holds true for weak_ptr's.

Peter Dimov comments (c++std-lib-12700):

> Technically, this is not a defect. There is an explicit requirement in 2.2.3.6 and 2.2.4.6 for operator< to be a strict weak ordering. This requirement implies that every smart pointer is equivalent to itself under !(p < q) && !(q < p).

In the current text, the equivalence relation is not required to yield true for two different empty pointers p and q in order to allow implementations that use several statically allocated control blocks for empty pointers. In such an implementation, two empty pointers may or may not share a control block.

The original version of the proposal allowed implementations where it is not possible to detect whether a given smart pointer is empty. The revised version in the TR, however, does not permit such implementations, since it requires use_count() to return zero for empty pointers. Therefore, it is possible to tighten the specification of operator< as proposed.

Proposed Resolution:
Change the specification of shared_ptr::operator<() to say two shared_ptr's are equivalent if and only if they share ownership or are both empty.

Change the specification of weak_ptr::operator<() to say two weak_ptr's are equivalent if and only if they share ownership or are both empty.

2.5 May smart pointers point to incomplete types?

Submitter: Peter Dimov
Status: TR

In clause 17 (specifically, 17.4.3.6), we say that we get undefined behavior "if an incomplete type is used as a template argument when instantiating a template component." Should we make an explicit exception to that general rule for `shared_ptr`?

Sydney: Intuitively it makes sense to say that you should be able to have a shared_ptr where T is incomplete. However, we do not have a proposal saying what you can and can't do with such a shared_ptr. We can't specify that it works until we identify those limitations, and this issue has no proposed resolution. However, ruling out incomplete types is a serious limitation. There are many use cases where this feature is important.

Straw poll, close as NAD vs leave open: 2-6. Alisdair will try to come up with a proposed resolution for the Redmond meeting.

Additional comments from Peter Dimov:
All of the proposed text was written as if there were an explicit requirement that T is allowed to be incomplete. Otherwise void would not have been allowed, either.

There are no additional restrictions on what you can do with shared_ptr<T> (and weak_ptr<T>), where T is incomplete. The Boost implementation supports it fully, and this is a very important feature of shared_ptr

Also note that the template parameter of enable_shared_from_this is always incomplete, as its intended use is:

```
class X: public enable_shared_from_this<X>
{
};
```

and since enable_shared_from_this<T> typically contains a weak_ptr<T> member, it is effectively rendered useless by an implementation that does not support incomplete types.

Proposed Resolution:
Add to 2.2.3: "The template parameter T of shared_ptr can be an incomplete type."

Add to 2.2.4: "The template parameter T of weak_ptr can be an incomplete type."

Add to 2.2.5: "The template parameter T of enable_shared_from_this can be an incomplete type."

2.6 dynamic_ptr_cast and deleters

Submitter: Alisdair Meredith
Status: NAD

c++std-lib-12600:
The following example appears to meet the explicit reqirements for 2.2.3.9. Not sure if I am missing some implicit reqs.

```
class base
{
};

class derived : public base
{
  void foo();
};

struct DeleteDerived
{
  template< class T >
  void operator()( T *pt )
  {
    if( pt ) pt->foo();
    delete pt;
  }
};

int main()
{
  shared_ptr< base > pb;
  {
     shared_ptr< derived > pd( new derived, DeleteDerived() );
     pb = dynamic_pointer_cast< base >( Make );
  }
}
```

I suspect this kind of functor-deleter should be disallowed, but appears to pass the conditions in 2.2.3.1 Assuming DeleteDerived is rewritten as a straight function:

```
void DeleteDerived( derived *pd )
{
  if( pd ) pd->foo();
  delete pd;
};
```

What are the implications on shared_ptr< base > calling DeleteDerived? The pointer points to the correct object type, but is only known to be of base type. However, I am not clear what happens dispatching all this through a function pointer. Are the function pointer type assignment compatible?

Again, this simpler deleter appears to meet the explicit requirements in 2.2.3.9.

Rationale:
We have a pointer to the correct type internally. The control block knows the correct type internally. Even a shared pointer to void works correctly.

2.7 weak_ptr and deleters

Submitter: Alisdair Meredith
Status: NAD

c++std-lib-12601:
Deleters again: what happens in the following case?

```
void array_deleter( int *p )
{
  delete []p;
}

int main()
{
  shared_ptr<int> p1;
  {
    shared_ptr<int> p2( new int[1], &array_deleter );
    weak_ptr< int > pw( p2 );
    p1 = pw.lock();
  }
}
```

Rationale:
NAD for the same reason asd issue 2.6.

2.8 Need equivalent of shared_ptr for arrays

Submitter: Alisdair Meredith

Status: NAD

The lack of shared_array is a problem, as undefined behavior storing arrays in smart pointers is a frequent problem when learning. This problem is worse when our only advice is "don't do that"

IIUC the recommended solution is to use shared_ptr with a deleter object that will delete arrays instead. Why not put that deleter into the TR as well, to make this clear?

Proposed Resolution:
Add a deleter class that's appropriate for arrays:

```
struct array_deleter
{
template<class T>
void operator()( T *pt ) const { delete []pt; }
};
```

Then add a non-normative example showing how this can be used:

```
struct junk  {
  static int i;
  ~junk() {
    std::cout <<  "deleting object number "
              <<  ++i
              << std::endl;
   }
};

int junk::i = 0;
int main()
{
  std::tr1::shared_ptr<void> p(new junk[5], array_deleter());
  return 0;
}
```

Rationale:
The argument for it: it would be of value for users. The argument against it: it's not at all hard to do, so all we need is user education.

2.9 Proposed addition: const_pointer_cast

Submitter: Peter Dimov
Status: TR

N1450 says in III.B.11 that "reinterpret_cast and const_cast equivalents have been omitted since they have never been requested by users."

This was true at the time, but I was shown a legitimate use case for const_pointer_cast; a library returns shared_ptr<X const> "read handles" and provides a separate "lock" function that converts a read handle to a write handle (shared_ptr<X>).

On most (all?) existing implementations, shared_ptr<X const> is layout-compatible with shared_ptr<X>, so it is possible to achieve the desired effect with a reinterper_cast, but a portable mechanism would be better.

Proposed resolution:
Add to 2.2.3.9:

```
template<class T, class U>
shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);
```

Requires: The expression const_cast<T*>(r.get()) is well-formed.

Returns: If r is empty, an empty shared_ptr<T>; otherwise, a shared_ptr<T> object that stores const_cast<T*>(r.get()) and shares ownership with r.

Throws: nothing.

Notes: the seemingly equivalent expression shared_ptr<T>(const_cast<T*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice.

2.10 Missing converting constructor requirements

Submitter: Peter Dimov
Status: TR

The following constructors:

```
template<class Y> shared_ptr(shared_ptr<Y> const& r);
template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);

template<class Y> weak_ptr(shared_ptr<Y> const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);
```

are missing a requirement that Y* needs to be convertible to T*.

Proposed resolution:

In 2.2.3.1, replace:

```
shared_ptr(shared_ptr const& r);
template<class Y> shared_ptr(shared_ptr<Y> const& r);
```

Effects: If r is empty, constructs an empty shared_ptr; otherwise, constructs a shared_ptr that shares ownership with r.
Postconditions: get() == r.get() && use_count() == r.use_count().
Throws: nothing.

with:

```
shared_ptr(shared_ptr const& r);
template<class Y> shared_ptr(shared_ptr<Y> const& r);
```

Requires: for the second constructor Y* shall be convertible to T*.

Effects: If r is empty, constructs an empty shared_ptr; otherwise, constructs a shared_ptr that shares ownership with r.
Postconditions: get() == r.get() && use_count() == r.use_count().
Throws: nothing.

Add:
Requires: Y* shall be convertible to T*.
after:
```
template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
```

In 2.2.4.1, replace:
```
template<class Y> weak_ptr(shared_ptr<Y> const& r);
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);
```
Effects: If r is empty, constructs an empty weak_ptr; otherwise, constructs a weak_ptr that shares ownership with r and stores a copy of the pointer stored in r.
Postconditions: use_count() == r.use_count().
Throws: nothing.
with:
```
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(shared_ptr<Y> const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);
```
Requires: for the second and third constructors Y* shall be convertible to T*.
Effects: If r is empty, constructs an empty weak_ptr; otherwise, constructs a weak_ptr that shares ownership with r and stores a copy of the pointer stored in r.
Postconditions: use_count() == r.use_count().
Throws: nothing.

3 Type traits issues

3.1 Use of Language in type transformations

Submitter: Pete Becker
Status: TR

See N1519 for discussion of the issue.

Resolution:
Accept the proposed resolution for N1519. *[but editorial change: also add a non-normative note pointing out what it means for cv-qualified types]*

3.2 Why three headers?

Submitter: Pete Becker
Status: TR

Three headers seems excessive. Why not put them all into <type_traits>? That would simplify things for users, who wouldn't have to remember which of the three headers defines the template

they're interested in. Currently, <type_traits> has 33 templates (not counting helpers), <type_compare> has 3, and <type_transform> has 11. The classification is reasonable in itself, but I don't think it's particularly helpful.

A number of people expressed support for one header on the LWG reflector.

Resolution: Combine the three type traits headers into a single header named <type_traits>.

3.3 Is integral_constant an implementation detail?

Submitter: Pete Becker
Status: NAD

See N1519 for discussion of the issue.

Resolution:
NAD. We accepted several changes that require integral_constant to be exposed explicitly.

3.4 Revising the Unary Type Traits Requirements

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue.

Resolution: Accept the proposed resolution from N1519.

3.5 New type trait: alignment_of

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue.

Resolution: Accept the proposed resolution from N1519.

3.6 New type trait: has_virtual_destructor

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue.

Resolution: Accept the proposed resolution from N1519, but add a proviso that **false** is the fallback position if the compiler can't determine an exact answer.

3.7 New type trait: is_safely_destructible

Submitter: Bronek Kozicki
Status: NAD

See N1508 for discussion of the issue.

Resolution: The LWG decided not to accept this proposal. If we accepted it, it would be better for the template to have two parameters: can class **D** be safely destroyed via a pointer to class **B**? But as is, the trait seems too high level: it answers a complicated compound question, not an atomic question.

3.8 New type trait: rank

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue.

Resolution:
Discussed at Kona. The LWG wasn't sure whether this was useful; the few people who could use it reliably for metaprogramming would probably find it just as easy to write it themselves.

3.9 New type trait: dimension

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue. At Sydney the LWG decided that this was a good idea but that "dimension" was a confusing word. We agreed to use "rank" and "extent," since those two words are unambiguous.

Resolution:
(This proposed resolution is N1620.)

```
4.2  Header <type_traits> synopsis:

Add under:

// type properties:
...
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

Change:

template <class T> struct remove_dimension;
template <class T> struct remove_all_dimensions;

to:

template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

4.3.4 Type properties
```

Add:

```
template <class T> struct rank {
 static const std::size_t value = implementation_defined;
  typedef std::size_t                          value_type;
 typedef integral_constant<value_type,value> type;
 operator type()const;
};
```

value: An implementation-defined integer value representing the rank of objects of type T (8.3.4). [Note - the term "rank" here is used to describe the number of dimensions of an array type - end note]

[example -

```
 // the following assertions should hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);
```

- end example]

...

```
template <class T, unsigned I = 0> struct extent {
 static const std::size_t value = implementation_defined;
  typedef std::size_t value_type;
 typedef integral_constant<value_type,value> type;
 operator type()const;
};
```

value: An implementation-defined integer value representing the extent (dimension) of the I'th bound of objects of type T (8.3.4). If the type T is not an array type, has rank of less than I, or if I == 0 and is of type "array of unknown bound of T", then value shall evaluate to zero; otherwise value shall evaluate to the number of elements in the I'th array bound of T. [Note - the term "extent" here is used to describe the number of elements in an array type - end note]

[example -

```
 // the following assertions should hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[][4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
```

```
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[][4], 1>::value) == 4);

- end example]
```

4.5.3 Array modifications:

Change:

```
template <class T> struct remove_dimension{
   typedef T type;
};
template <class T, std::size_t N> struct remove_dimension<T[N]>{
   typedef T type;
};
template <class T> struct remove_dimension<T[]>{
   typedefs T type;
};
```

to:

```
template <class T> struct remove_extent {
   typedef T type;
};
template <class T, std::size_t N> struct remove_extent<T[N]> {
   typedef T type;
};
template <class T> struct remove_extent<T[]> {
   typedef T type;
};
```

Change:

```
[example
  // the following assertions should all hold:
  assert((is_same<remove_dimension<int>::type, int>::value));
  assert((is_same<remove_dimension<int[2]>::type, int>::value));
  assert((is_same<remove_dimension<int[2][3]>::type,
int[3]>::value));
  assert((is_same<remove_dimension<int[][3]>::type,
int[3]>::value));
Ñend example]

template <class T> struct remove_all_dimensions {
   typedef T type;
};
template <class T, std::size_t N> struct
remove_all_dimensions<T[N]> {
   typedef typename remove_all_dimensions<T>::type type;
};
```

```
template <class T> struct remove_all_dimensions<T[]> {
   typedef typename remove_all_dimensions<T>::type type;
};

to:

[example
  // the following assertions should all hold:
  assert((is_same<remove_extent<int>::type, int>::value));
  assert((is_same<remove_extent<int[2]>::type, int>::value));
  assert((is_same<remove_extent<int[2][3]>::type,
int[3]>::value));
  assert((is_same<remove_extent<int[][3]>::type,
int[3]>::value));
Ñend example]

template <class T> struct remove_all_extents {
   typedef T type;
};
template <class T, std::size_t N> struct
remove_all_extents<T[N]> {
   typedef typename remove_all_extents<T>::type type;
};
template <class T> struct remove_all_extents<T[]> {
   typedef typename remove_all_extents<T>::type type;
};

Change:

[example
  // the following assertions should all hold:
  assert((is_same<remove_all_dimensions<int>::type,
int>::value));
  assert((is_same<remove_all_dimensions<int[2]>::type,
int>::value));
  assert((is_same<remove_all_dimensions<int[2][3]>::type,
int>::value));
  assert((is_same<remove_all_dimensions<int[][3]>::type,
int>::value));
Ñend example]

to:

[example
  // the following assertions should all hold:
  assert((is_same<remove_all_extents<int>::type, int>::value));
  assert((is_same<remove_all_extents<int[2]>::type,
int>::value));
  assert((is_same<remove_all_extents<int[2][3]>::type,
int>::value));
```

```
  assert((is_same<remove_all_extents<int[][3]>::type,
int>::value));
Ñend example]

4.5.4 Pointer modifications

Change:

template <class T> struct add_pointer {
   typedef typename remove_dimension<
      typename remove_reference<T>::type
               >::type*
           type;
};

to:

template <class T> struct add_pointer
{
   typedef typename remove_extent
   <
      typename remove_reference<T>::type
   >::type* type;
};

4.6 Implementation requirements

Change:

is_pod<T>::value == is_pod<remove_dimension<T>::type>::value

to:

is_pod<T>::value == is_pod<remove_extent<T>::type>::value

Change:

has_trivial_*<T>::value ==
has_trivial_*<remove_dimension<T>::type>::value

to:

has_trivial_*<T>::value ==
has_trivial_*<remove_extent<T>::type>::value
```

3.10 New type trait: aligned_storage

Submitter: John Maddock

Status: TR

See N1519 for discussion of the issue.

Resolution:
Accept the proposed resolution from N1519, but say "unspecified" instead of "implementation defined."

3.11 New type trait: remove_all_dimensions

Submitter: John Maddock
Status: TR

See N1519 for discussion of the issue.

Resolution:
Accept the proposed resolution from N1519.

3.12 Conversion of traits to integral_constant

Submitter: Dave Abrahams
Status: TR

Every traits class **X** has a nested typedef **type**, and has a conversion operator, **operator type() const**. Automatic conversions are useful and important, but a conversion operator is the wrong way to do it. Instead, we should say that **X** inherits from **type**. This would be consistent with actual implementation practice.

3.13 is_base_of<X,X>

Submitter: Dave Abrahams
Status: TR

Currently, `is_base_of<X,Y>` returns false when X and Y are the same. This is technically correct (X isn't its own base class), but it isn't useful. The definition should be loosened to return true when X and Y are the same, even when the type isn't actually a class.

Note: the LWG agreed that this behavior is more useful than what's currently in the TR. We were uneasy about changing the behavior will keeping the name is_base_of, but nobody thought of a better name. We will consider changing the name if someone can come up with a better one.

3.14 Type_traits specifications could be simpler

Submitter: Pete Becker
Status: TR

In 4.4.2 (for example), we say:

```
template<class T> struct remove_const{
        typedef T type;
```

};
template<class T> struct remove_const<T const>{
	typedef T type;
};
type: defined to be a type that is the same as T, except that any top level const-qualifier has been removed....

The use of two structs is an implementation technique. The description is the actual behavior. It should be written like this:

template<class T> struct remove_const{
	typedef T1 type;
};
The type T1 is the same as T, except that any top level const-qualifier has been removed.

This form of change is needed for all of the type transformations in clause 4.4.

Proposed resolution:
Accept the changes described in N1713=04-0153, “Proposed Resolution to TR1 Issues 3.12, 3.14, and 3.15,” by Pete Becker.

3.15 Inconsistent non-normative note for has_virtual_destructor

Submitter: John Maddock
Status: TR

The entry for has_virtual_destructor has a non-normative note that reads: "[Note: An implementation that cannot determine whether a type has a virtual destructor, e.g. a pure library implementation with no compiler support, should return false. -end note]".

However none of the other template that require compiler support have such notes, instead they have an entry in section 4.6 (which has_virtual_destructor does not), we should try to be consistent.

Proposed resolution:
Accept the changes described in N1713=04-0153, “Proposed Resolution to TR1 Issues 3.12, 3.14, and 3.15,” by Pete Becker.

3.16 aligned_storage underspecified?

Submitter: Pete Becker
Status: TR

This type trait produces "an implementation defined POD type with size Len and alignment Align, and suitable for use as uninitialized storage for any object of a type whose size is Len and whose alignment is Align."

Is the intention to permit arbitrary values for the Align argument? I hope not: I don't know how to implement that. <g> Seems like there ought to be a constraint that Align must be one of some implementation-specific set of values. Probably the best way to say that is to require that it be one of the values that can be returned by align_of. (No, the set shouldn't be implementation defined -- that's a nightmare, because it depends on compiler switches.)

Proposed resolution:
Replace [tr.meta.trans.other] paragraph 1 with:

Preconditions: Len shall be nonzero. Align shall be equal to alignment_of<T>::value for some type T.

type: an unspecified POD type suitable for use as uninitialized storage for any object whose size is at most Len and whose alignment is a divisor of Align.

[Note: a typical implementation would define type as:

```
union type
{
   unsigned char __data[Len];
   /Aligner/ __align;
};
```

where /Aligner/ is the smallest POD type for which alignment_of</Aligner/>::value is Align. -- end note.]

3.17 type_traits Compromise Requirements

Submitter: Pete Becker
Status: TR

[tr.meta.req] specifies compromise requirements when implementations can't get the exact answer. Paragraphs 3, 4, and 5 say that if an implementation can't get 'em right, is_class, is_union, is_polymorphic, and is_abstract "need not be provided." Should this be "shall not be provided"? Alternatively, is there something useful we can say about a conservative fallback for implementations that can't get the right answer?

Proposed resolution:
Change [tr.meta.req]/3 from:

> If there is no means by which the implementation can differentiate between class and union types, then the class templates is_class and is_union need not be provided.

to:

> If there is no means by which the implementation can differentiate between class and union types, then the class templates is_class and is_union shall be defined as follows:
>
> template <class T> struct is_class {};
> template <class T> struct is_union {};

Change [tr.meta.req]/4 from:

> If there is no means by which the implementation can detect polymorphic types, then the class template is_polymorphic need not be provided.

to:

If there is no means by which the implementation can detect polymorphic types, then the class template is_polymorphic shall be defined as follows:

template <class T> struct is_polymorphic {};

Change [tr.meta.req]/5 from:

If there is no means by which the implementation can detect abstract types, then the class template is_abstract need not be provided.

to:

If there is no means by which the implementation can detect abstract types, then the class template is_abstract shall be defined as follows:

template <class T> struct is_abstract {};

3.18 What should is_class do for unions?

Submitter: Discussion on the reflector
Status: NAD

The type traits classes are supposed to model clause of the standard, and the bulleted list in paragraph 1 of [basic.compound] lists "classes" and "unions" separately; most users probably think of classes and unions as different things. On the other hand, the standard is quite clear that ([class], paragraph 4) a union is just "a class defined with the class-key `union`," and even that bulleted list in [basic.compound] says that a union is a kind of class. Should we change `is_class` so that it returns true for classes defined with any of the three class-keys? If so, should we provide any mechanism for users to find out if a type is a class-but-not-union?

Rationale:
The LWG believed that it would be a usability problem, and a violation of user expectation, to return "true" for unions. Most of the use cases for which anyone would want to query a type to see if it's a class, you don't want to do the same thing for unions as for ordinary classes.

3.19 Incorrect normative description of type properties

Submitter: Martin Sebor
Status: TR

I don't think the statement in [tr.meta.unary.prop]/1 is completely accurate or appropriate:

> These templates provide access to some of the more important properties of types; they reveal information which is available to the compiler, but which would not otherwise be detectable in C++ code.

The type properties exposed by several of the templates, (e.g., is_const, et al) are easily detectable in C++ code. Additionally, I don't think the reference to the compiler is appropriate since, at least in theory, C++ code need not be translated by a compiler.

Proposed resolution:

I propose to drop the part of the sentence after the semicolon since it doesn't carry any information relevant to the rest of the section.

3.20 Is_convertible and void

Submitter: Dave Abrahams
Status: New

What should `is_convertible<From, To>` do if `To` is `void`? Currently the `TR` is quite clear: it's ill formed. Perhaps this should be changed so that `is_convertible<From, void>` is well formed and returns false. After all, `void` isn't convertible to anything else.

3.21 Incorrect implementation requirements

Submitter: John Maddock
Status: New

In 4.9 it says that:

> It is unspecified under what circumstances, if any, is_pod<T>::value evaluates to true, except that, for all types T:
>
> is_pod<T>::value == is_pod<remove_extent<T>::type>::value
> is_pod<T>::value == is_pod<T const volatile>::value

However the implication of 3.9 para 2 appears to be that volatile qualified types can not be POD's. Likewise volatile qualified types probably can not have trivial any operations (construct, copy, assign etc), or at least whether they can or not is implementation defined.

Likewise in 4.9 it also says:

> It is unspecified under what circumstances, if any, has_trivial_*<T>::value evaluates to true, except that:
>
> has_trivial_*<T>::value == has_trivial_*<remove_extent<T>::type>::value
> has_trivial_*<T>::value >= is_pod<T>::value

In addition to the problem with volatile qualifiers outlined above, there is one operation (assignment) that can not be applied to const-qualified types, so even if a const-qualified type is considered a POD, has_trivial_assign for that type must surely be false.

Finally it also says:

> If the implementation cannot detect abstract types, then the class template is_abstract shall be defined as follows:
>
> template <class T> struct is_abstract {};

But as Peter Dimov has pointed out, we now have an implementation technique for implementing is_abstract entirely within the language, so this latitude can be removed.

Proposed resolution:

In section 4.9, replace:

> is_pod<T>::value == is_pod<T const volatile>::value

with:
is_pod<T>::value == is_pod<T const>::value

and replace:
It is unspecified under what circumstances, if any, has_trivial_*<T>::value evaluates to true, except that:
has_trivial_*<T>::value == has_trivial_*<remove_extent<T>::type>::value
has_trivial_*<T>::value >= is_pod<T>::value
with:
It is unspecified under what circumstances, if any, has_trivial_*<T>::value evaluates to true, except that:
has_trivial_*<T>::value == has_trivial_*<remove_extent<T>::type>::value
has_trivial_assign<T>::value == is_pod<T>::value && !is_const<T>::value

and for all other has_trivial* other than has_trivial_assign:
has_trivial_*<T>::value >= is_pod<T>::value

Finally remove paragraph 5 of 4.9, which reads:
If the implementation cannot detect abstract types, then the class template is_abstract shall be defined as follows:
template <class T> struct is_abstract {};

4 Random number generator issues

4.1 Confusing Text in Description of v.min()

Submitter: Pete Becker (see N1535)
Status: TR

In "Uniform Random Number Requirements" the text says that v.min() "Returns ... l where l is ...". This is the letter ell, which is too easily confused with the numeral one. Can we change it to something less confusing, like "lim"?

Resolution:
Change the first sentence of the description of v.min() in 5.1.1 [tr.rand.req], Table 5.2 (Uniform random number generator requirements) from:
Returns some l where l is less than or equal to all values potentially returned by operator().
to:
Returns a value that is less than or equal to all values potentially returned by operator().

4.2 Confusing and Incorrect Text in Description of v.max()

Submitter: Pete Becker (see N1535)
Status: TR

In "Uniform Random Number Requirements" the text says that v.max() "returns l where l is less than or equal to all values...". Should this be "greater than or equal to"? And similarly, should "strictly less than" be "strictly greater than."?

Resolution:
Change the first sentence of the description of v.max() in 5.1.1 [tr.rand.req], Table 5.2 (Uniform random number generator requirements) from:

> If std::numeric_limits<T>::is_integer, returns l where l is less than or equal to all values potentially returned by operator(), otherwise, returns l where l is strictly less than all values potentially returned by operator().

to:

> If std::numeric_limits<T>::is_integer, returns a value that is greater than or equal to all values potentially returned by operator(), otherwise, returns a value that is strictly greater than all values potentially returned by operator().

4.3 Table "Number Generator Requirements" Unnecessary

Submitter: Pete Becker (see N1535)
Status: TR

The table "Number Generator Requirements" has only one entry: X::result_type. While it's true that random nunber generators and random distributions have this member, it doesn't seem like a useful basis for classification -- there's nothing in the proposal that depends on knowing that some type satisfies this requirement. I think the specification of X::result_type should be in "Uniform Random Number Generator Requirements" and in "Random Distribution Requirements."

Resolution:
Copy the description of X::result_type from 5.1.1 [tr.rand.req], Table 5.1 (Number generator requirements) to 5.1.1 [tr.rand.req], Table 5.2 (Uniform random number generator requirements) and to 5.1.1 [tr.rand.req], Table 5.4 (Random distribution requirements) and remove 5.1.1 [tr.rand.req], Table 5.1 (Number generator requirements).

4.4 Should a variate_generator Holding a Reference Be Assignable?

Submitter: Pete Becker (see N1535)
Status: TR

The third paragraph says, in part:

> Specializations of variate_generator satisfy the requirements of CopyConstructible. They also satisfy the requirements of Assignable unless the template parameter Engine is of the formU&.

This looks like an implementation artifact. Is there a reason that variate_generators whose engine type is a reference should not be copied?

Resolution:
Change the first two sentences of the third paragraph of 5.1.3 [tr.rand.var] from:

> Specializations of variate_generator satisfy the requirements of CopyConstructible. They also

satisfy the requirements of Assignable unless the template parameter Engine is of the form U&.

to:

Specializations of `variate_generator` satisfy the requirements of CopyConstructible and Assignable. [*Note:* If the template parameter `Engine` is of reference type it is the reference, not the object referred to, that is copied. —*End Note]*

4.5 Normal Distribution Incorrectly Specified

Submitter: Pete Becker (see N1535)
Status: TR

For normal_distribution, the paper says that the probability density function is1/sqrt(2*pi*sigma) * exp(- (x - mean)^2 / (2 * sigma^2)). The references I've seen have a different initial factor, using 1/(sqrt(2*pi) * sigma). That is, sigma is outside the square root.

Resolution:
Change the first paragraph of 5.1.7.8 [tr.rand.dist.norm] from:

A normal_distribution random distribution produces random numbers x distributed with probability density function $(1/\mathrm{sqrt}(2*\mathrm{pi}*\mathrm{sigma}))e^{-(x-\mathrm{mean})2/(2*\mathrm{sigma}2)}$, where mean and sigma are the parameters of the distribution.

to:

A normal_distribution random distribution produces random numbers x distributed with probability density function $(1/(\mathrm{sqrt}(2*\mathrm{pi})*\mathrm{sigma}))e^{-(x-\mathrm{mean})2/(2*\mathrm{sigma}2)}$, where mean and sigma are the parameters of the distribution.

4.6 Should Random Number Initializers Take Iterators by Reference or by Value?

Submitter: Pete Becker
Status: TR

See N1547 for a full discussion. Summary: when engines are seeded, the seed may be arbitrarily large. For compound engines we use a range where the first iterator is taken by reference and updated. This is an unconventional interface and will invite bugs. The obvious solution would be to have a function that takes iterators `first` and `last` by value and returns the updated version of first. However, this is an awkward solution for constructors. One possibility would be to abandon range constructors, and rely instead on two-phase initialization where the iterators are passed to a member function.

Notes: Sydney: the LWG agrees that this is an improvement. The only discomfort is that it would be nice to have the glue code to turn a pair of iterators into a generator, instead of asking each user to write it for themself. This may be a TR2 candidate.

Resolution:
In section 5.1.1 [tr.rand.req], replace in the paragraph before table 5.2

... it1 is an lvalue and it2 is a (possibly const) value of an input iterator type It having an unsigned integral value type, ...

by

... g is an lvalue of a zero-argument function object returning values of unsigned integral type, ...

In the same section, replace in table 5.2 the table row for X(it1, it2) by

expression: X(g)

return type: (none)

pre/post-condition: creates an engine with the initial internal state given by the results of successive invocations of g. Throws what and when g throws.

complexity: O(size of state)

In the same section, replace in table 5.2 the table row for u.seed(it1, it2) by

expression: u.seed(g)

return type: void

pre/post-conditions: sets the internal state of u so that u == X(g). If an invocation of g throws, that exception is rethrown, and further use of u (except destruction) is undefined until a seed member function has been executed without throwing an exception.

complexity: same as X(g)

After table 5.2, add a new paragraph following the one starting "Additional Requirements":

For every pseudo-random number engine defined in this clause:

- the constructor

template<class Gen> X(Gen& g)

shall have the same effect as

X(static_cast<Gen>(g))

if Gen is a fundamental type.

- The member function of the form

template<class Gen> void seed(Gen& g)

shall have the same effect as

X(static_cast<Gen>(g))

if Gen is a fundamental type.

[Note: The casts make g an rvalue, unsuitable for binding to a reference.]

[Note to editor: The following changes intend to morph "dereferencing *first" to "invoking g" only. However, complete text has been given.]

In section 5.1.4.1 [tr.rand.eng.lcong], replace the constructor template<class In> linear_congruential(In& first, In last) by

template<class Gen> linear_congruental(Gen& g)

Effects: If c mod m = 0 and g() mod m = 0, sets the state x(i) of the engine to 1 mod m, else

sets the state x(i) of the engine to g() mod m.

Complexity: Exactly one invocation of g.

Furthermore, adjust the class synopsis accordingly.

In section 5.1.4.2 [tr.rand.eng.mers], replace the description of the constructor template<class In> mersenne_twister(In& first, In last) by

template<class Gen> mersenne_twister(Gen& g)

Effects: Given the values z[0] ... z[n-1] obtained by successive invocations of g, sets x(-n) ... x(-1) to z[0] mod 2w ... z[n-1] mod 2w.

Complexity: Exactly n invocations of g.

Furthermore, remove the description of the seed(first, last) function, it is subsumed by table 5.2 and the description of the constructor, and adjust the class synopsis accordingly.

In section 5.1.4.3 [tr.rand.eng.sub], replace the description of the constructor template<class In> subtract_with_carry(In& first, In last) by

template<class Gen> subtract_with_carry(Gen& g)

Effects: With n=(w+31)/32 (rounded downward) and given the values z[0] ... z[n*r-1] obtained by successive invocations of g, sets x(-r) ... x(-1) to (z0 * 232 + ... + zn-1 * 232*(n-1)) mod m ... (z(r-1)*n * 232 + ... + zr-1 * 232*(n-1)) mod m. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

Complexity: Exactly r*n invocations of g.

Furthermore, remove the description of the seed(first, last) function, it is subsumed by table 5.2 and the description of the constructor, and adjust the class synopsis accordingly.

In section 5.1.4.4 [tr.rand.eng.sub1], replace the description of the constructor template<class In> subtract_with_carry_01(In& first, In last) by

template<class Gen> subtract_with_carry_01(Gen& g)

Effects: With n=(w+31)/32 (rounded downward) and given the values z0 ... zn*r-1 obtained by successive invocations of g, sets x(-r) ... x(-1) to (z0 * 232 + ... + zn-1 * 232*(n-1)) * 2-w mod 1 ... (z(r-1)*n * 232 + ... + zr-1 * 232*(n-1)) * 2-w mod 1. If x(-1) == 0, sets carry(-1) = 2-w, else sets carry(-1) = 0.

Complexity: Exactly r*n invocations of g.

Furthermore, remove the description of the seed(first, last) function, it is subsumed by table 5.2 and the description of the constructor, and adjust the class synopsis accordingly.

In section 5.1.4.5 [tr.rand.eng.disc], replace the description of the constructor template<class In> discard_block(In& first, In last) by

template<class Gen> discard_block(Gen& g)

Effects: Constructs a discard_block engine. To construct the subobject *b*, invokes the b(g) constructor. Sets n = 0.

Furthermore, remove the description of the seed(first, last) function, it is subsumed by table 5.2 and the description of the constructor, and adjust the class synopsis accordingly.

In section 5.1.4.6 [tr.rand.eng.xor], replace the description of the constructor template<class In> xor_combine(In& first, In last) by

template<class Gen> xor_combine(Gen& g)

Effects: Constructs a xor_combine engine. To construct the subobject *b1*, invokes the b1(g) constructor. Then, to construct the subobject *b2*, invokes the b2(g) constructor.

Furthermore, remove the description of the seed(first, last) function, it is subsumed by table 5.2 and the description of the constructor, and adjust the class synopsis accordingly.

4.7 Are Global Operators Overspecified?

Submitter: Pete Becker (see N1535)
Status: TR

See N1535 for a full discussion. Summary: Do we literally want to require the existence of a namespace-scope `operator==`, or do we just want to say that when `x` and `y` are engines, `x == y` is required to work?

Resolution:

In section 5.1.1 [tr.rand.req], table 5.2, replace in the pre-/post-condition column for x == y

== is an equivalence relation. The current state x(i) of x is equal to the current state y(j) of y.

by

== is an equivalence relation. Given the current state x(i) of x and the current state y(j) of y, returns true if x(i+k) is equal to y(j+k) for all integer k >= 0, false otherwise.

In section 5.1.4.1 [tr.rand.eng.lcong], remove the prototypes for operator== and operator!= from the synopsis.

In section 5.1.4.2 [tr.rand.eng.mers], remove the prototypes for operator== and operator!= from the synopsis.

In section 5.1.4.3 [tr.rand.eng.sub], remove the prototypes for operator== and operator!= from the synopsis.

In section 5.1.4.4 [tr.rand.eng.sub1], remove the prototypes for operator== and operator!= from the synopsis.

In section 5.1.4.5 [tr.rand.eng.disc], remove the prototypes for operator== and operator!= from the synopsis.

In section 5.1.4.6 [tr.rand.eng.xor], remove the prototypes for operator== and operator!= from the synopsis.

4.8 Should the Template Arguments Be Restricted to Built-in Types?

Submitter: Pete Becker (see N1535)
Status: TR.

See N1535 for a full discussion. Summary: Generators and distributions are parameterized on arithmetic types. The TR tries to allow user defined number-like types, but it's very hard to get that sort of thing right. We should restrict it to the built-in arithmetic types.

Resolution:
Replace in 5.1.1 [tr.rand.req], last paragraph

> Furthermore, a template parameter named RealType shall denote a type that holds an approximation to a real number. This type shall meet the requirements for a numeric type (26.1 [lib.numeric.requirements]), the binary operators +, -, *, / shall be applicable to it, a conversion from double shall exist, and function signatures corresponding to those for type double in subclause 26.5 [lib.c.math] shall be available by argument-dependent lookup (3.4.2 [basic.lookup.koenig]). [Note: The built-in floating-point types float and double meet these requirements.]

by

> Furthermore, the effect of instantiating a template that has a template type parameter namedRealType is undefined unless that type is one of float, double, or long double.

Delete from 5.1.7 [tr.rand.dist]

> A template parameter named IntType shall denote a type that represents an integer number. This type shall meet the requirements for a numeric type (26.1 [lib.numeric.requirements]), the binary operators +, -, *, /, % shall be applicable to it, and a conversion from int shall exist. [Footnote: The built-in types int and long meet these requirements.]
>
> ...
>
> No function described in this section throws an exception, unless an operation on values of IntType or RealType throws an exception. [Note: Then, the effects are undefined, see [lib.numeric.requirements].]

Add after 5.1.1 [tr.rand.req], last paragraph

> The effect of instantiating a template that has a template type parameter named IntType is undefined unless that type is one of short, int, long, or their unsigned variants.
>
> The effect of instantiating a template that has a template type parameter named UIntType is undefined unless that type is one of unsigned short, unsigned int, or unsigned long.

4.9 Do Engines Need Type Arguments?

Submitter: Pete Becker (see N1535)
Status: Closed

See N1535 for a discussion. Summary: engines are parameterized by type, but this is pretty much redundant. The appropriate type can be deduced from the template arguments.

Resolution: Discussed at Kona. No consensus that this change would be a good idea. That's still the status from Sydney.

4.10 Unclear Complexity Requirements for variate_generator

Submitter: Pete Becker (see N1535)
Status: TR

The specification for variate_generator says

> Specializations of variate_generator satisfy the requirements of CopyConstructible. They also satisfy the requirements of Assignable unless the template parameter Engine is of the formU&. The complexity of all functions specified in this section is constant. No function described in this section except the constructor throws an exception.

Taken literally, this isn't implementable. operator() calls the underlying distribution's operator(), whose complexity isn't directly specified. The distribution's operator() makes an amortized constant number of calls to the generator's operator(), whose complexity is, again, amortized constant. So the complexity ofvariate_generator::operator() ought to also be amortized constant.

variate_generator also has a constructor that takes an engine and a distribution by value, and uses their respective copy constructors to create internal copies. There are no complexity constraints on those copy constructors, but given that the default constructor for an engine has complexity O(size of state), it seems likely that an engine's copy constructor would also have complexity O(size of state). This means thatvariate_generator's complexity is at best O(size of engine's state), not constant.

I suspect that what was intended was that these functions would not introduce any additional complexity, that is, their complexity is the "larger" of the complexities of the functions that they call.

Resolution:
Replace in 5.1.3 [tr.rand.var]

> The complexity of all functions specified in this section is constant.

by

> Except where otherwise specified, the complexity of all functions specified in this section is constant.

Add for variate_generator(engine_type e, distribution_type d)

> **Complexity:** Sum of the complexities of the copy construtors of engine_type anddistribution_type.

Add for result_type operator()()

> **Complexity:** Amortized constant.

Add for result_type operator()(T value)

> **Complexity:** Amortized constant.

4.11 xor_combine Over-generalized?

Submitter: Pete Becker (see N1535)
Status: Editorial

For an xor_combine engine, is there ever a case where both s1 and s2 would be non-zero? Seems like this would produce non-random values, because the low bits (up to the smaller of the two shift values) would all be 0.

If at least one has to be 0, then we only need one shift value, and the definition might look more

like this:
template <class _Engine1, class _Engine2, int _Shift = 0> ...

with the output being (_Eng1() ^ (_Eng2() << _Shift)).

Resolution: Discussed at Kona. The LWG felt that this interface is still the simplest. The right solution is to add a non-normative note advising users that only one of these parameters should be nonzero. The project editor is directed to add that note.

4.12 xor_combine::result_type Incorrectly Specified

Submitter: Pete Becker (see N1535)
Status: TR

xor_combine has a member
typedef typename base_type::result_type result_type;

However, it has no type named base_type, only base1_type and base2_type. So, what should `result_type` be?

Resolution:
In 5.1.4.6 [tr.rand.eng.xor] replace
typedef typename base_type::result_type result_type;
by
typedef /* see below */ result_type;

and add at the end of the paragraph below the class definition
The member result_type is defined to that type ofUniformRandomNumberGenerator1::result_type andUniformRandomNumberGenerator2::result_type that provides the most storage [basic.fundamental].

4.13 subtract_with_carry's IntType Overpecified

Submitter: Pete Becker (see N1535)
Status: TR

The IntType for subtract_with_carry "shall denote a signed integral type large enough to store values up to m - 1." The implementation subtracts two values of that type, and if the result is < 0 it adds back the m, which makes the result non-negative. In fact, this also works for unsigned types, with just a small change in the implementation: instead of testing whether the result is < 0 you test whether it's < 0 or greater than or equal to m. This works because unsigned arithmetic wraps, and it makes the template a bit easier to use.

I suggest that we loosen the constraint to allow signed and unsigned types. Thus the constraint would read "shall denote an integral type large enough to store values up to m - 1."

Resolution:
In 5.1.4.3 [tr.rand.eng.sub], replace

The template parameter IntType shall denote a signed integral type large enough to store values up to m-1.

by

The template parameter IntType shall denote an integral type large enough to store values up to m.

4.14 subtract_with_carry_01::seed(unsigned) Missing Constaint

Submitter: Pete Becker (see N1535)
Status: TR

The specification for subtract_with_carry::seed(IntVal) has a *Requires* clause which requires that the argument be greater than 0. This member function needs the same constraint.

Resolution:
Add:

Requires: value > 0

to the description of subtract_with_carry_01::seed(unsigned) in 5.1.4.4 [tr.rand.eng.sub1]. (See resolution of issue 4.19, which also affects the wording in this area.)

4.15 subtract_with_carry_01::seed(unsigned) Produces Bad Values

Submitter: Pete Becker (see N1535)
Status: TR

subtract_with_carry_01::seed(unsigned int) uses a linear congruential generator to produce initial values for the fictitious previously generated values. These values are generated as(y(i)*2^-w) mod 1. The linear congruential generator produces values in the range [0, 2147483564), which are at most 31 bits long. If the template argument w is greater than 31 the initial values generated by seed will all be rather small, and the first values produced by the generator will also be rather small. The Boost implementation avoids this problem by combining values from the linear congruential generator to produce longer values when w is larger than 32. Should we require something more like that?

Resolution:
In 5.1.4.4 [tr.rand.eng.sub1] replace

void seed(unsigned int value = 19780503)

Effects: With a linear congruential generator l(i) having parameters m = 2147483563, a = 40014, c = 0, and l(0) = value, sets x(-r) ... x(-1) to (l(1)*2-w) mod 1 ... (l(r)*2-w) mod 1, respectively. If x(-1) == 0, sets carry(-1) = 2-w, else sets carry(-1) = 0.

Complexity: O(r)

With

void seed(unsigned long value = 19780503ul)

Effects: With n=(w+31)/32 (rounded downward) and given an iterator range [first, last)that refers to the sequence of values lcg(1) ... lcg(n*r) obtained from a linear congruential generator lcg(i) having parameters mlcg = 2147483563, alcg = 40014, clcg = 0, and lcg(0) =value, invoke seed(first,last) .

Complexity: O(r*n)

4.16 subtract_with_carry_01::seed(unsigned) Argument Type Too Small

Submitter: Pete Becker (see N1535)
Status: TR

subtract_with_carry_01::seed(unsigned) has a default argument value of 19780503, which is too large to fit in a 16-bit unsigned int. Should this argument be unsigned long, to ensure that it's large enough for the default?

Resolution:
In 5.1.4.2 [tr.rand.eng.mers], change the signature of a constructor and a seed function from
```
explicit mersenne_twister(result_type value);
void seed(result_type value);
```
to
```
explicit mersenne_twister(unsigned long value);
void seed(unsigned long value);
```

In 5.1.4.3 [tr.rand.eng.sub], change the signature of a constructor and a seed function from
```
explicit subtract_with_carry(IntType value);
void seed(IntType value = 19780503);
```
to
```
explicit subtract_with_carry(unsigned long value);
void seed(unsigned long value = 19780503ul);
```

In 5.1.4.4 [tr.rand.eng.sub1], change the signature of a constructor and a seed function from
```
subtract_with_carry_01(unsigned int value);
void seed(unsigned int value = 19780503);
```
to:
```
subtract_with_carry_01(unsigned long value);
void seed(unsigned long value = 19780503ul);
```

4.17 subtract_with_carry::seed(In&, In) Required Sequence Length Too Long

Submitter: Pete Becker (see N1535)
Status: TR

For both subtract_with_carry::seed(In& first, In last) andsubtract_with_carry_01::seed(In& first, In last) the proposal says: "With n=w/32+1 (rounded downward) and given the values z0 ... zn*r-1." The idea is to use n unsigned long values to generate each of the initial values for the generator, so n should be the number of 32-bit words needed to provide wbits. Looks like it should be "n=(w+31)/32". As currently written, when w is 32, the function consumes two 32-bit values for each value that it generates. One is sufficient.

Resolution:
Change
```
With n=w/32+1 (rounded downward) and given the values z0 ... zn*r-1
```

to

With n=(w+31)/32 (rounded downward) and given the values z0 ... zn*r-1

in the description of subtract_with_carry::seed(In& first, In last) in 5.1.4.3 [tr.rand.eng.sub] and in the description of subtract_with_carry_01::seed(In& first, In last) in 5.1.4.4 [tr.rand.eng.sub1].

4.18 linear_congruential -- Giving Meaning to a Modulus of 0

Submitter: Pete Becker (see N1535)
Status: TR

Some linear congruential generators using an integral type _Ty also use a modulus that's equal to numeric_limts<_Ty>::max() + 1 (e.g. 65536 for a 16-bit unsigned int). There's no way to write this value as a constant of the type _Ty, though. Writing it as a larger type doesn't work, because the linear_congruential template expects an argument of type _Ty, so you typically end up with a value that looks like 0.

On the other hand, the current text says that the effect of specifying a modulus of 0 forlinear_congruential is implementation defined. I decided to use 0 to mean max()+1, as did the Boost implementation. (Internally, the implementation of mersenne_twister needs a generator with a modulus like this). Seems to me this is a reasonable choice, and one that users ought to be able to rely on. Is there some other meaning that might reasonably be ascribed to it, or should we say that a modulus of 0 meansnumeric_limits<_Ty>::max() + 1 (suitably type-cast)?

Resolution:
Replace in 5.1.4.1 [tr.rand.eng.lcong], in the paragraph after the class definition

If the template parameter m is 0, the behaviour is implementation-defined.

by

If the template parameter m is 0, the modulus m used throughout this section isstd::numeric_limits<IntType>::max() plus 1. [*Note*: The result is not representable as a value of type IntType. —*end note*]

4.19 linear_congruential::seed(IntType) -- Modify Seed Value When c == 0?

Submitter: Pete Becker (see N1535)
Status: TR

When c == 0 you get a generator with a slight quirk: if you seed it with 0 you get 0's forever; if you seed it with a non-0 value you never get 0. The first path, of course, should be avoided. The proposal does this by imposing a requirement on seed(IntType x0), requiring that c > 0 || (x0 % m) > 0. The boost implementation uses asserts to check this condition. The only reservation I have about this is that it can only be checked at runtime, when the only suitable action is, probably, to abort. An alternative would be to force a non-0 seed in that case (perhaps 1, for no particularly good reason). I think the second alternative is marginally better, and I suggest we change this requirement to impose a particular seed value when a user passes 0 to a generator with c == 0.

Resolution:

Replace in 5.1.4.1 [tr.rand.eng.lcong]

explicit linear_congruential(IntType x0 = 1)

Requires: c > 0 || (x0 % m) > 0
Effects: Constructs a linear_congruential engine with state x(0) := x0 mod m.

void seed(IntType x0 = 1)

Requires: c > 0 || (x0 % m) > 0
Effects: Sets the state x(i) of the engine to x0 mod m.

template linear_congruential(In& first, In last)

Requires: c > 0 || *first > 0
Effects: Sets the state x(i) of the engine to *first mod m.
Complexity: Exactly one dereference of *first.

by

explicit linear_congruential(IntType x0 = 1)

Effects: Constructs a linear_congruential engine and invokes seed(x0).

void seed(IntType x0 = 1)

Effects: If c mod m = 0 and x0 mod m = 0, sets the state x(i) of the engine to 1 mod m, else sets the state x(i) of the engine to x0 mod m.

template linear_congruential(In& first, In last)

Effects: If c mod m = 0 and *first mod m = 0, sets the state x(i) of the engine to 1 mod m, else sets the state x(i) of the engine to *first mod m.
Complexity: Exactly one dereference of *first.

Replace in 5.1.4.2 [tr.rand.eng.mers]

void seed()

Effects: Invokes seed(4357).

void seed(result_type value)

Requires: value > 0
Effects: With a linear congruential generator l(i) having parameters $m_l = 232$, $a_l = 69069$, $c_l = 0$, and l(0) = value, sets x(-n) ... x(-1) to l(1) ... l(n), respectively.
Complexity: O(n)

by

void seed()

Effects: Invokes **seed(0)**.

void seed(result_type value)

Effects: If value == 0, sets value to 4357. In any case, with a linear congruential generator lcg(i) having parameters $m_{lcg} = 232$, alcg = 69069, $c_{lcg} = 0$, and lcg(0) = value, sets x(-n) ... x(-1) to lcg(1) ... lcg(n), respectively.
Complexity: O(n)

Replace in 5.4.1.3 [tr.rand.eng.sub]

void seed(unsigned int value = 19780503)

Requires: value > 0

Effects: With a linear congruential generator l(i) having parameters ml = 2147483563, al = 40014, cl = 0, and l(0) = value, sets x(-r) ... x(-1) to l(1) mod m ... l(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

Complexity: O(r)

by

void seed(unsigned long value = 19780503ul)

Effects: If value == 0, sets value to 19780503. In any case, with a linear congruential generator lcg(i) having parameters m_{lcg} = 2147483563, a_{lcg} = 40014, c_{lcg} = 0, and lcg(0) =value, sets x(-r) ... x(-1) to lcg(1) mod m ... lcg(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

Complexity: O(r)

Replace in 5.4.1.4 [tr.rand.eng.sub1]

void seed(unsigned int value = 19780503)

Effects: With a linear congruential generator l(i) having parameters m = 2147483563, a = 40014, c = 0, and l(0) = value, sets x(-r) ... x(-1) to (l(1)*2-w) mod 1 ... (l(r)*2-w) mod 1, respectively. If x(-1) == 0, sets carry(-1) = 2-w, else sets carry(-1) = 0.

Complexity: O(r)

by

void seed(unsigned long value = 19780503ul)

Effects: If value == 0, sets value to 19780503. In any case, with a linear congruential generator lcg(i) having parameters m_{lcg} = 2147483563, a_{lcg} = 40014, c_{lcg} = 0, and lcg(0) =value, sets x(-r) ... x(-1) to lcg(1) mod m ... lcg(r) mod m, respectively. If x(-1) == 0, sets

Complexity: O(r)

4.20 linear_congruential -- Should the Template Arguments Be Unsigned?

Submitter: Pete Becker (see N1535)
Status: TR

This template takes three numeric arguments, a, c, and m, whose type is IntType. IntTypeis an integral type, possibly signed. These arguments specify the details of the recurrence relation for the generator:

x(i + 1) := (a * x(i) + c) mod m

Every discussion that I've seen of this algorithm uses unsigned values. Further, In C and C++ there is no modulus operator. The result of the % operator is implementation specific when either of its operands is negative, so implementing mod when the values involved can be negative requires a test and possible adjustment:

```
IntType res = (a * x + c) % m;
if (res < 0)
    res += m;
```

If the three template arguments can't be negative the recurrence relation can be implemented directly:

```
x = (a * x + c) % m;
```

This makes the generator faster.

Resolution:
In clause 5.1.4.1 [tr.rand.eng.lcong] replace every occurrence of IntType with UIntType and change the first sentence after the definition of the template from:

> The template parameter IntType shall denote an integral type large enough to store values up to(m-1).

to:

> The template parameter UIntType shall denote an unsigned integral type large enough to store values up to (m-1).

4.21 linear_congruential::linear_congruential(In&, In) -- Garbled Requires Clause

Submitter: Pete Becker (see N1535)
Status: TR

The **Requires** clause for the member template template <class In> linear_congruential(In& first, In last) got garbled in the translation to .pdf format.

Resolution:
Change the **Requires** clause for the member template template <class In> linear_congruential(In& first, In last) in 5.1.4.1 [tr.rand.eng.lcong] from:

> **Requires:** c > 0 — *first ¿ 0—

to:

> **Requires:** c > 0 || *first > 0

4.22 bernoulli_distribution Isn't Really a Template

Submitter: Pete Becker (see N1535)
Status: TR

The text says that bernoulli_distribution is a template, parametrized on a type that is required to be a real type. Its operator() returns a bool, with the probability of returning true determined by the argument passed to the object's constructor. The only place where the type parameter is used is as the type of the argument to the constructor. What is the benefit from making this type user-selectable instead of, say, double?

Resolution:
In 5.1.7.2 [tr.rand.dist.bern], change the section heading to "Class bernoulli_distribution", remove template <class RealType = double> from the declaration of bernoulli_distribtion, change the declaration of the constructor from:

```
explicit bernoulli_distribution(const RealType& p = RealType(0.5));
```

to:
explicit bernoulli_distribution(double p = 0.5);

and change the header for the subclause describing the constructor from:
bernoulli_distribution(const RealType& p = RealType(0.5))
to:
bernoulli_distribution(double p = 0.5)

4.23 Streaming Underspecified

Submitter: Pete Becker (see N1535)
Status: TR

See N1535 for a full discussion. Summary: the goal is for engines to be well enough specified so that the state of an engine can be streamed out on one system and read in on a different system, and so that the engine on the second system would produce the same sequence of values as it would on the first. Distributions are less clear-cut, but at least we want to be able to save and restore on the same system for the sake of checkpointing. Given that we don't care about portability, streaming of distributions may be adequately specified. However, we may not want to call it `operator<<` and `operator>>`, because implementers will probably want to use binary formats.

Proposed resolution:
After table 5.2, add a new paragraph following the one starting "Additional Requirements":
If a textual representation was written by os << x and that representation was read by is >> v, then x == v, provided that no intervening invocations of x or v have occurred.
In section 5.1.4.1 [tr.rand.eng.lcong], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state x(i) is 1.":
The textual representation is the value of x(i).

In section 5.1.4.2 [tr.rand.eng.mers], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state x(i) is n.":
The textual representation is the values of x(i-n), ..., x(i-1), in that order.

In section 5.1.4.3 [tr.rand.eng.sub], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state is r.":
The textual representation is the values of x(i-r), ..., x(i-1), carry(i-1), in that order.

In section 5.1.4.4 [tr.rand.eng.sub1], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state is r.":
With n = (w+31)/32 (rounded downward) and integer numbers z[k,j] such that x(i-k)*2w = z[k,0] + z[k,1] * 232 + z[k,n-1] * 232(n-1), the textual representation is the values of z[r,0], ... z[r,n-1], ... z[1,0], ... z[1,n-1], carry(i-1)*2w, in that order. [Note: The algorithm ensures that only integer numbers representable in 32 bits are written.]

In section 5.1.4.5 [tr.rand.eng.disc], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state is the

size of b plus 1.":

The textual representation is the textual representation of *b* followed by the value of n.

In section 5.1.4.6 [tr.rand.eng.xor], remove the prototypes for operator<< and operator>> from the synopsis. Also, remove the description of operator<<. Add after "The size of the state is the size of the state of b1 plus the size of the state of b2.":

The textual representation is the textual representation of *b1* followed by the textual representation of *b2*.

4.24 Garbled characters

Submitter: Jens Maurer
Status: Editorial

There are some places where the TR draft contains garbled characters. This issue points out the places where editorial changes to rectify this need to be performed.

- 5.1.4.3 [tr.rand.eng.sub], first paragraph
- 5.1.4.4 [tr.rand.eng.sub1], first paragraph
- 5.1.4.5 [tr.rand.eng.disc], after the class definition
- 5.1.4.5 [tr.rand.eng.disc], effects clause of operator()

4.25 class vs. type

Submitter: Jens Maurer
Status: TR

The wording in section 5.1.1 isn't parallel.

Resolution: Replace in section 5.1.1 [tr.rand.req], last paragraph
In the following subclauses, a template parameter named UniformRandomNumberGenerator shall denote a class **type** that satisfies all the requirements of a uniform random number generator.

4.26 Fix section reference

Submitter: Jens Maurer
Status: TR, Editorial

A section reference needs to be fixed.

Resolution:
Replace in section 5.1.4 [tr.rand.eng], second paragraph

The class templates specified in this section satisfy all the requirements of a pseudo-random number engine (given in tables in section ~~x.x~~ **5.1.1 [tr.rand.req]**), except where specified otherwise. Descriptions are provided here only for operations on the engines that are not described in one of these tables or for operations where there is additional semantic information.

4.27 Avoid confusion for "ell" and "one"

Submitter: Jens Maurer

Status: TR

We need to be careful with subscripts: “l” and “1” look very similar in most fonts, so “l” is a poor choice for a variable that will be used in subscripts.

Resolution:
Replace in 5.4.1.2 [tr.rand.eng.mers]

> Effects: With a linear congruential generator l(i) having parameters m_l = 232, al = 69069, c_l = 0, and l(0) = value, sets x(-n) ... x(-1) to l(1) ... l(n), respectively.

by

> Effects: With a linear congruential generator lcg(i) having parameters m_{lcg} = 232, a_{lcg} = 69069, c_{lcg} = 0, and lcg(0) = value, sets x(-n) ... x(-1) to lcg(1) ... lcg(n), respectively.

Replace in 5.4.1.3 [tr.rand.eng.sub]

> **Effects:** With a linear congruential generator l(i) having parameters m_l = 2147483563, a_l = 40014, c_l = 0, and l(0) = value, sets x(-r) ... x(-1) to l(1) mod m ... l(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

by

> **Effects:** With a linear congruential generator lcg(i) having parameters m_{lcg} = 2147483563, a_{lcg}= 40014, c_{lcg} = 0, and lcg(0) = value, sets x(-r) ... x(-1) to lcg(1) mod m ... lcg(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

Replace in 5.4.1.4 [tr.rand.eng.sub1]

> **Effects:** With a linear congruential generator l(i) having parameters m = 2147483563, a = 40014, c = 0, and l(0) = value, sets x(-r) ... x(-1) to (l(1)*2^{-w}) mod 1 ... (l(r)*2^{-w}) mod 1, respectively. If x(-1) == 0, sets carry(-1) = 2^{-w}, else sets carry(-1) = 0.

by

> **Effects:** With a linear congruential generator lcg(i) having parameters m_{lcg} = 2147483563, a_{lcg}= 40014, c_{lcg} = 0, and lcg(0) = value, sets x(-r) ... x(-1) to (lcg(1)*2^{-w}) mod 1 ... (lcg(r)*2^{-w}) mod 1, respectively. If x(-1) == 0, sets carry(-1) = 2^{-w}, else sets carry(-1) = 0.

[Note to editor: see issue 19 for another issue that touches these words.]

4.28 xor_combine: fix typo

Submitter: Jens Maurer
Status: TR

Resolution:
Replace in 5.1.4.6 [tr.rand.eng.xor]

> The template parameters UniformRandomNumberGenerator1 and UniformRandomNumberGenerator~~1~~**2** shall denote classes that satisfy all the requirements of a uniform random number generator, ...

[Replace "1" by "2" once.]

4.29 Require additional properties for Engine result_type

Submitter: Jens Maurer

Status: TR

Currently, there are no restrictions on UniformRandomNumberGenerator::result_type, althoughvariate_generator is supposed to possibly convert between integer and floating-point types.

Proposed resolution:
In 5.1.1 [tr.rand.req], replace the pre/post-condition for result_type:
    std::numeric_limits<T>::is_specialized is true
by
    T is an arithmetic type [basic.fundamental]

4.30 Garbled precondition for min()

Submitter: Jens Maurer
Status: TR

Proposed resolution:
In 5.1.3 [tr.rand.var], add the highlighted text for min():
    Precondition: distribution().min() is **well-formed**

4.31 xor_combine: Require additional properties for base*_type::result_type

Submitter: Jens Maurer
Status: TR

There are no restrictions on UniformRandomNumberGenerator1::result_type and UniformRandomNumberGenerator2::result_type that would ensure that << and ^ are available on them. That's well defined for unsigned integral types.

Proposed resolution:
Add in 5.1.4.6 [tr.rand.eng.xor] in the paragraph after the class definition
    Both UniformRandomNumberGenerator1::result_type
    andUniformRandomNumberGenerator2::result_type shall denote (possibly different) unsigned integral types. The size of the state ...

4.32 Be precise about the size of the state of xor_combine

Submitter: Jens Maurer
Status: TR

It is unclear what the "size of b1" and the "size of b2" mean, we only talk about the "size of the state".

Proposed resolution:
Add in 5.1.4.6 [tr.rand.eng.xor] in the paragraph after the class definition:
    The size of the state is the size **of the state** of b1 plus the size **of the state** of b2.

4.33 uniform_real should return open interval

Submitter: Jens Maurer
Status: TR

uniform_real was specified with a closed interval [min, max] range, but it should have a half-open interval [min, max) range to avoid lots of special cases in more complex distributions. (The boost implementation and documentation does this since ever.)

Proposed resolution:

In 5.1.7.6 [tr.rand.dist.runif], replace
   min <= x <= max
by
    min <= x < max

4.34 No complexity specification for copy construction and copy assignment

Submitter: Jens Maurer
Status: TR

In 5.1.1 [tr.rand.req], add a new paragraph after table 5.3 (pseudo-random number generator):
    Additional requirements: The complexity of both copy construction and assignment is O(size of state).

4.35 Insufficient preconditions on discard_block

Submitter: Jens Maurer
Status: TR

discard_block does not have sufficient requirements on the r and p template parameters.

Proposed resolution:

Replace in 5.1.4.5 [tr.rand.eng.disc]
   r <= q
by
    The following relation shall hold: 0 <= r <= p.

4.36 Insufficient preconditions on xor_combine

Submitter: Jens Maurer
Status: TR

xor_combine does not have any requirements for s1 and s2 template parameters.

Proposed resolution:

Add in 5.1.4.6 [tr.rand.eng.xor], paragraph after the class definition, before "The size of the state

..."

The following relation shall hold: 0 <= s1 and 0 <= s2.

4.37 Streaming operators must handle templatized streams

Submitter: Jens Maurer
Status: TR

Section 5.1.1 [tr.rand.req], table 5.2, specifies that the streaming operators take std::ostream and std::istream. This does not take wide streams nor the template nature of streams into account.

Resolution:
(The proposed resolution is N1621.)

Replace in section 5.1.1 [tr.rand.req] before table 5.2:
In table 5.2, [...], `os` is convertible to an lvalue of type `std::ostream`, and `is` is convertible to an lvalue of type `std::istream`.

by

In table 5.2, [...], `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`, and `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`, where `charT` and `traits` are constrained according to [lib.strings] and [lib.input.output].

Replace in section 5.1.1 [tr.rand.req] in table 5.2, in the row for `os << x`, the return type
`std::ostream&`
by
reference to the type of `os`
Also, replace in the row for `is >> v`, the return type
`std::istream&`
by
reference to the type of `is`
and the pre/post-condition
sets the state v(i) of `v` as determined by reading its textual representation from `is`. post: The `is`.*fmtflags* are unchanged.
by
sets the state `v(i)` of `v` as determined by reading its textual representation from `is`. pre: The textual representation was previously written using a `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same than those of `is`, respectively. post: The `is`.*fmtflags* are unchanged.

In section 5.1.1 [tr.rand.req], add at the end of the paragraph preceding table 5.3
`os` is convertible to an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`, and `is` is convertible to an lvalue of the type of some class template specialization `basic_istream<charT, traits>`, where `charT` and `traits` are constrained according to [lib.strings] and [lib.input.output].

Replace in section 5.1.1 [tr.rand.req] in table 5.3, in the row for `os << x`, the return type
`std::ostream&`
by

reference to the type of `os`
Also, replace in the row for `is >> v`, the return type
`std::istream&`
by
reference to the type of `is`
and the pre/post-condition
restores the parameters and additional internal data of the distribution u. pre: `is` provides a textual representation that was previously written by `operator<<`. post: The `is`.*fmtflags* are unchanged.
by
restores the parameters and additional internal data of the distribution u. pre: `is` provides a textual representation that was previously written using a `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same than those of `is`, respectively. post: The `is`.*fmtflags* are unchanged.

4.38 Seeding of random number generators

Subumitter: Matt Austern
Status: TR

All of the random number engine classes in clause 5.1.4 of the TR have two methods for setting the seed: an inconvenient but fully general method (passing in a zero-argument function object that returns unsigned integers) and a less general but more convenient method (passing in a single unsigned int). However, the engine requirements table (table 9, Pseudo-random number engine requirements) has only a single seeing interface: the general and inconvenient one. Should we put the simple interface in the requirements table?

Further comments from Jens Maurer:

> We would need to fix both the ctor and the seed() requirements, plus we need some language (currently in the individual sections for specific engines) that X(g) and X(unsigned int) are never ambiguous and magically resolve to the right thing.

Proposed resolution:
Modify section 5.1.1 as follows:

- In paragraph 2, add after "v is an lvalue of X"
  ..., s is a value of integral type, ...
- Add in table 9 before X(g):
  expression: X(s)
  return type: none
  pre/post-condition: creates an engine with the initial internal state determined by s
  complexity: O(size of state)
- Add in table 9 before seed(g):
  expression: u.seed(s)
  return type: void
  pre/post-condition: post: sets the internal state of u so that u == X(s).
  complexity: same as X(s)

- Merge paragraph 5 with paragraph 4 and remove the words at the beginning of paragraph 5: "For every pseudo-random number engine defined in this clause" so that the requirements in paragraph 5 become general "additional requirements".

In section 5.1.4.1 "linear_congruential", replace the declarations in both the synopsis and the detailed description:

```
explicit linear_congruential(UIntType x0 = 1);
...
void seed(UIntType x0 = 1);
```

by

```
explicit linear_congruential(unsigned long x0 = 1);
...
void seed(unsigned long x0 = 1);
```

In section 5.1.4.5 "discard_block", add (also to the synopsis)

explicit discard_block(unsigned long s)

Effects: Constructs a discard_block engine. To construct the subobject *b*, invokes the b(s) constructor. Sets n = 0.

In section 5.1.4.6 "xor_combine", add (also to the synopsis)

explicit xor_combine(unsigned long s)

Effects: Constructs a xor_combine engine. To construct the subobject *b1*, invokes the b1(s) constructor. Then, to construct the subobject *b2*, invokes the b2(s+1) constructor. *[Note: If both b1 and b2 are of the same type, both engines should not be initialized with the same seed.]*

4.39 Is uniform_real fully specified?

Submitter: Jens Maurer
Status: New

The uniform_real distribution is supposed to be able to handle any UniformRandomNumberGenerator engine by scaling the results appropriately. However, there's still a crucial link missing in the requirements, which makes the requirements on the individual distribution templates unduly burdensome,.

5.1.3p10 variate_generator has verbose language that ensures a mapping between the values that the engine returns and the values that we give to the distribution:

- if the distribution wants integers and the engine provides integers, we don't do anything at all (the distribution likely knows best what to do in this case)
- if the distribution wants real numbers, variate_generator provides real numbers in the range [0,1) to the distribution, adapting to both integer-valued and real-valued engines with funny ranges.

That's a nice service for all distributions, because they get a "reasonable" range as input. (The boost implementation relies on that.) 5.1p8 and table 10 say that e must be an engine returning values of type U, and X::input_type be U. Thus, nobody can call uniform_real's operator() with an integer-valued engine.

Unfortunately, there appears to be no language that says that the distributions wanting real numbers as input have additional restrictions on the urng parameter of their operator(), namely that urng.min() == 0.0 and urng.max() == 1.0 . That forces all the distributions to perform the division again (see my description of variate_generator above), and the only hope is that the compiler is clever enough to detect that the second division is redundant. Divisions are expensive.

Proposed resolution:
Add at the end of 5.1p9 "If std::numeric_limits<X::input_type>::is_integer is false, the value of e.min() shall be 0 and the value of e.max() shall be 1."

5 Special function issues

5.1 Clean up special function names and descriptions

Submitter: Bill Plauger, Walter Brown
Status: TR

The names of special functions should be cleaned up so they're all-lowercase and more spelled out (to make them more consistent with C naming style), there should be names with ***f*** and ***l*** suffixes for float and long double versions, and the behavior should be specified mathematically instead of by reference.

Resolution:
Accept the changes proposed in N1542, "Mathematical special functions, v3".

5.2 Assoc_legendre incorrectly requires a domain error

Submitter: Bill Plauger
Status: TR

assoc_legendre says "a domain error occurs if m is greater than l." But the value is well defined -- zero. Hence, a domain error should **never** occur.

Status: Sydney: 5.2 through 5.6 all go together. The general principle we want: if a function is defined with a real value we should return it. If it isn't we should give a domain error. The difficulty is that some formulations of these functions are applicable to a wider domain than others.

We will leave all of these issues open for now. Bill, Walter, and Marc will come up with a proposal by Redmond.

Resolution:
Accept the proposed resolution from N1665.

5.3 Assoc_legendre should require domain error when |x| > 1

Submitter: Bill Plauger
Status: TR

assoc_legendre says "a domain error may occur if the magnitude of x is greater than one." But the value is always imaginary. Hence, a domain error should **always** occur.

Resolution:
Accept the proposed resolution from N1665.

5.4 Beta should have domain error if x <= 0 or y <= 0

Submitter: Bill Plauger
Status: TR

beta says "a domain error may occur (a) if either x or y is a negative integer, or (b) if either x or y is zero." But the beta function is defined only for x,y > 0. Hence, a domain error should **always** occur if x <= 0 or y <= 0.

Resolution:
Accept the proposed resolution from N1665.

5.5 Legendre should always have domain error if |x| > 1

Submitter: Bill Plauger
Status: TR

legendre says "a domain error may occur if the magnitude of x is greater than one." But the value is always imaginary. Hence, a domain error should **always** occur.

Resolution:
Accept the proposed resolution from N1665.

5.6 Bessel should require domain error for x < 0

Submitter: Bill Plauger
Status: TR

Bessel functions all say "a domain error may occur if x is less than zero." The various Bessels can generally be extended to negative real x, but the functions are arguably undefined along the negative real axis. Hence, a domain error should **always** occur.

Resolution:
Accept the proposed resolution from N1665.

5.7 Order parameter in Bessel functions should be restricted

Submitter: Bill Plauger
Status: TR

Bessel functions have a parameter ν that describes their order. There are no known techniques for computing these functions for arbitrarily large ν with tolerable efficiency and accuracy. See N1697 for more details.

Proposed resolution:
Accept the proposed resolution from N1697.

6 Unordered associative container issues

6.1 Incorrect const qualification

Submitter: Rober Klarer
Status: TR

The parameters to the container swap functions are const-qualified, and I don't think they should be. For example the declaration for the swap function that appears in 6.2.4.3.2 is

```
template <class Value, class Hash, class Pred, class Alloc>
void swap(const unordered_set<Value, Hash, Pred, Alloc>& x,
          const unordered_set<Value, Hash, Pred, Alloc>& y);
```

I believe that x and y can't be references to const containers because the swap function needs to be able to modify both containers.

Resolution:
In section 6.4.2 [tr.unord.unord], remove the const qualification in the parameters of the nonmember swap functions for all four unordered associative containers, both in the header synopses and in the text.

6.2 Erase takes const iterator

Submitter: Rober Klarer
Status: NAD

The erase member functions with iterator parameters are declared as follows

```
void erase(const_iterator position);
void erase(const_iterator first, const_iterator last);
```

This is consistent with the requirements table, but I'm not sure that it's intentional.

Resolution: Not a defect. This was intentional. The other containers should probably be changed in a similar way in a future standard.

6.3 Bucket members not declared const

Submitter: Rober Klarer
Status: TR

The bucket(...) and bucket_size(...) members of each container template should be const, but they aren't declared const in the class definitions. The requirements table correctly implies that these functions are const members.

Resolution:
In section 6.4.2 [tr.unord.unord], in the class declarations of all four unordered associative containers, declare the bucket and bucket_size member functions as const.

6.4 Incorrect variable in requirements table

Submitter: Rober Klarer
Status: TR

All occurences of "for const a" in the "Return Type" column of the requirements table should actually read "for const b." Also, under the the "assertion/note/pre/postcondition" column, the phrase "out of which a was constructed" should be "out of which b was constructed" for b.hash_function() and b.key_eq(). Similarly, "a.end()" should be "b.end" for b.find(k), and "std::make_pair(a.end(), a.end())" should be "std::make_pair(b.end(), b.end())" for b.equal_range(k).

Resolution:
As above. (See N1549.)

6.5 Hashing strings

Submitter: Alan Stokes
Status: TR

N1518 at 6.2.3 requires the library to provide a specialisation of the hash template for basic_string instantiated with any valid set of charT, traits, and Alloc.

This is tricky, for two reasons:

1. charT can be any POD. It might therefore be a struct with padding for alignment. How does the implementation hash the value while skipping the unused bytes?

2. hash is required to return equal results for equal arguments. For basic_string equality is determined by traits::eq, so can be arbitrary. For example it could ignore case, or it could ignore some components of a POD struct. So the library doesn't know, when given an argument to hash, what other arguments it might compare equal to.

These problems are not insurmountable - hash<basic_string<...> > could just always return 0, or could just hash the string length. But neither would be very good hash functions for use in the unordered containers.

Perhaps only std::char_traits should be allowed; that limits you to hashing strings of char and wchar_t (but with any allocator).

Or we could require the supplied traits class to support hashing of individual characters, and add the necessary support to std::char_traits.

Resolution:
(From N1622)

In 6.3.3, change "for any valid set of charT, traits, and Alloc" to "for any valid set of charT, traits, and Alloc such that charT is an integer type."

Rationale:

Agreed that this is a problem. Hashing of fully general strings is probably unimplementable. We considered three choices: (1) hash only string and wstring. (2) hash basic_string for every integer type T. (3) hash basic_string for any T that can be cast to unsigned long. The LWG preferred choice 2. Straw poll: 2-5-0.

6.6 Unordered assoc containers not containers

Submitter: Beman Dawes
Status: TR

The TR does not explicitly say that unordered associative containers must meet the standard's requirements for containers. The phrase "(in addition to container)" is part of the title for table 6.1, but that is not explicit enough, and fails to make clear that all of 23.1's requirements have to be met, not just table 65's.

For consistency, the proposed resolution wording is similar to the way that std::basic_string (21.3, paragraph 2) references the Sequence requirements.

Proposed Resolution
To section 6.2.1, Unordered associative container requirements, add:

> Unordered associative containers conform to the requirements for Containers (C++ Standard, 23.1, Container requirements).

6.7 Exception safety of unordered associative container operations

Submitter: Matt Austern
Status: TR

The only unordered associative container members that provide anything other than the basic exception guarantee are clear(), erase(), swap(), and the single-element version of insert(). In particular, rehash() only provides the basic guarantee. This is correct as far as it goes, but we can do better.

Proposed resolution (N1622):
Add to the list of exception safety guarantees:
> "For unordered associative containers, if an exception is thrown from within a rehash() function other than by the container's hash function or comparison function, the rehash() function has no effect."

6.8 Equality-comparability of unordered associative containers

Submitter: Robert Klarer
Status: TR

The unordered associative containers were intended to satisfy all of the general container requirements, but they don't. In particular, the unordered associative containers are not equality-comparable.

Naively defining equality comparison for these containers doesn't solve this problem. According to the general Container requirements table, equality comparison for containers should work like

this:

== is an equivalence relation.
a.size()==b.size() && equal(a.begin(), a,end(), b.begin())

This definition of container equality is inadequate for unordered containers. Should an unordered_set A containing the elements {3, 2, 1} be considered equal to an unordered_set B containing the elements {1, 2, 3}? There is good reason to think so, especially since the order of the elements in a particular container will seem arbitrary to the user. This order will depend on the bucket count of the container, peculiarities of the implementation, etc. Unfortunately, if the unordered associative containers were equality-comparable in the way that is required by Container, then the containers A and B (from the examples above) will definitely not compare equal.

Resolution (N1622):
To section 6.2.1, Unordered associative container requirements, add:
Unordered associative containers conform to the requirements for Containers (C++ Standard, 23.1, Container requirements), except that the following expressions are not required to be valid, where a and b denote values of a type X, and X is an unordered associative container class:

unsupported expressions
a == b
a != b
a < b
a > b
a <= b
a >= b

Rationale:
This was discussed in Oxford. We are revisiting the issue. The fundamental problem: the equality function described in the container requirements makes no sense for hashtables. We considered four choices: close as NAD, put in a caveat saying we don't quite satisfy the container requirements, put in the operator== defined in terms of std::equal, or put in Howard's (more useful) operator==. By an 0-6-0-3 straw vote we chose the second.

6.9 Unordered_map and unordered_multimap don't have assignable value types

Submitter: Rober Klarer
Status: NAD

The container requirements say that a container's value type must be assignable. The value types of unordered_map and unordered_multimap violate that requirement. (note that the same problem is true of map and multimap in the standard.)

Rationale:
LWG issue 276 will remove the requirement that a container's value type be assignable.

6.10 Unordered_multimap shouldn't have operator[]

Submitter: Rober Klarer

Status: TR

According to subsection 6.3.4.6 of the TR, unordered_multimap possesses an overloaded operator []. This should be removed.

Resolution:
Remove it.

6.11 Rationale for rehash precondition

Submitter: Thomas Witt
Status: TR

I am wondering what the rationale for the rehash precondition is. It seems to be unneccessary strict to me. My naive approach would be to simply do nothing if the requested bucket numer is less then size()/max_load_factor() possibly returning the resulting bucket count.

Resolution (N1622):

```
Page 117 in requirements table  for a.rehash(n)
===============================================

change
    Pre: n > a.size() /
    a.max_load_factor().
    Changes the number of buckets
    so that it is at least n.

to
    a.bucket_count() > a.size() / a.max_load_factor().
    a.bucket_count() >= n.
```

6.12 Definition of load factor is overconstrained

Submitter: Bill Wade
Status: NAD Future

I suggest that implementations be allowed (not required) to define load factor in terms of the number of elements with unique hash() values and/or the number of equivalence classes in an unordered_multi{set,map}. Particularly in a multi-map or multi-set, growing the bucket count unnecessarily is an anti-optimization.

Rationale:
Unclear if this is a real problem (implementations already have a fair amount of freedom in how they use the load factor), and having the load factor be implementation defined made people uncomfortable.

6.13 When may an implementation change the bucket count?

Submitter: Bill Wade
Status: TR

Is an implementation ever allowed to reduce the bucket count (other than via swap)? Is an implementation allowed to rehash when an insertion does not cause a violation of max_load_factor? Is insert() allowed to invalidate existing iterators when no rehash occurs?

Note that one set of answers to these questions means that rehash() provides some guarantees analogous to vector::reserve().

Resolution (N1622):
In clause 6.3.1, after the first sentence of the last paragraph, add: "The insert members shall not affect the validity of iterators if (N+n) < z * B, where N' is the container's size, n is the number of elements inserted, B is the container's bucket count, and z is the container's maximum load factor."

6.14 Complexity of iterator increment

Submitter: Bill Wade
Status: NAD

When the current load factor is very small, O(bucket_count()) can be considerably worse than O(size()). On many implementations iterator increment (or begin()) will have this problem. Dinkum (the version with MS 7.1) keeps the iterator operations O(1), but insert or erase adjacent to a range of empty buckets is expensive (linear). This means that calling rehash(n) with the current Dinkum implementation, prior to performing a large number of insertions would result in quadratic behavior.

It is straightforward to make the cost of empty-bucket ranges no more than log-time (and no more than one-bit per bucket space), but if I were happy with log time I might as well use an ordered map.

It isn't obvious to me how you can implement iterator++, insert() and erase() so that they are all faster than logarithmic in the number of empty buckets, but if I give you a perfect hash function and you tell me to call rehash(1000000) before I add a million elements, I'm going to be upset if the call to rehash() changes what was linear behavior to quadratic behavior. Is there any way to specify the interaction between rehash() and insert(), or will prudent programs avoid calls to rehash()?

Rationale:
It's amortized constant time, and it'll be bad for degenerate hash functions, and that's just a characteristic of this data structure.

6.15 Hash functions and const containers

Submitter: Robert Klarer
Status: TR
Reflector message: c++std-lib-13458

Please consider:

```
  #include <functional>

  struct UserDefinedType { /* ... */ };

  struct my_hash<UserDefinedType>
    : public std::unary_function<UserDefinedType, std::size_t>
  {
    std::size_t operator()(UserDefinedType val) /* non-const!
*/;
  };
```

A hash function object such as this one can't be used as the hasher to an unordered associative container. Several of the unordered associative member functions, like find(), count(), equal_range(), and especially bucket(), need to be able to call the hash object's function-call operator. When the container is const-qualified, these member functions are const, too, so a hasher with no const function-call semantics won't work.

I have two questions:

- is there any explicit requirement that an unordered associative container's hasher type have const function-call semantics (eg. must the overloaded operator() be const-qualified)?
- if not, should there be (i.e. is this an issue)?

There may be similar issues with unordered associative containers' equality function objects, and with ordinary associative containers' comparison function objects.

Resolution:
In table 14, unordered associative container requirements, replace the assertion column for X::hasher with: "Hash is a unary function object type such that the expression hf(k) has type std::size_t".

6.16 Swap() missing from header synopses

Submitter: Matt Austern
Status: TR

The header synopses in 5.3.1.3 [tr.unord.syn.set] and 6.4.4.2 [tr.unord.syn.map] are supposed to contain all namespace scope declarations. The specializations of swap are missing.

Proposed resolution:
Add them.

6.17 Hashing strings, revisited

Submitter: Alan Stokes
Status: TR

Some time ago I raised the issue that the library TR originally required library implementers to support hashing of any valid instantiation of basic_string, which appears to be impossible. This is issue 6.5 in the list.

The resolution, proposed in N1622 and voted into the TR in Sydney, reduced the requirement to supporting hashing of instantiations of basic_string "for any valid set of charT, traits, and Alloc such that charT is an integer type."

I'm not convinced that that is sufficient. The library's hash function is required to ensure that equal inputs hash to the same value (TR 6.3.3/2). For basic_string equality comes down to what traits::eq does, and the implementation has no (easy) way to determine that.

Consider the common (if arguably misguided) case where charT is char and a user-defined traits class provides for case insensitive comparisons. How is the implementation to ensure that "cat" and "CAT" hash to the same value?

Or consider a basic_string<unsigned long> where each "character" consists of 8 bits of character data and 24 bits of other information (colour, say), and the user supplies a traits class that does comparisons looking only at the bottom 8 bits. Again the implementation will find it very difficult to ensure that strings that compare equal hash to the same value.

Proposed resolution:
1. Only hash string and wstring.

Rationale:
The LWG considered two other alternatives:
2. Only hash basic_string where charT is an integer type and traits is std::char_traits<charT>.
3. Only hash basic_string where charT is an integer type and traits::eq(c, d) returns c == d.

The LWG felt that either of those was unnecessarily complicated. String and wstring are the only important instantiations of basic_string. Anyone who is using other instantiations is certainly sophisticated enough to know how to provide their own hash function.

6.18 not enough support for hash functions on user-defined types

Submitter: Peter Dimov
Status: Closed

Currently, the unordered associative containers provide specializations for the hash<> class template for a selected list of built-in types and std::basic_string. However, no support is provided for writing quality hash functions for even the simplest user-defined types such as the equivalent of std::pair<int, int>.

Furthermore, the specialization interface through which users may supply a default hash function for a given type is less convenient than the alternative approach of providing a function overload reachable via ADL.

Proposed resolution:

Add the following to the synopsis in [tr.unord.fun.sys] before struct hash:

```
size_t hash_value(int v);
size_t hash_value(unsigned int v);
size_t hash_value(long v);
size_t hash_value(unsigned long v);

size_t hash_value(float v);
size_t hash_value(double v);
size_t hash_value(long double v);

template<class T> size_t hash_value(T * v);

template<class T> void hash_combine(size_t & seed, T const &
v);

template<class It> size_t hash_range(It first, It last);

template<class A, class B>
size_t hash_value(std::pair<A, B> const & v);

template<class E, class T, class A>
size_t hash_value(std::basic_string<E, T, A> const & v);

template<class T, class A>
size_t hash_value(std::vector<T, A> const & v);

template<class T, class A>
size_t hash_value(std::list<T, A> const & v);

template<class T, class A>
size_t hash_value(std::deque<T, A> const & v);

template<class K, class C, class A>
size_t hash_value(std::set<K, C, A> const & v);

template<class K, class C, class A>
size_t hash_value(std::multiset<K, C, A> const & v);

template<class K, class T, class C, class A>
size_t hash_value(std::map<K, T, C, A> const & v);

template<class K, class T, class C, class A>
size_t hash_value(std::multimap<K, T, C, A> const & v);
```

Add a new section [tr.unord.hashval], Template function hash_value, with the following contents:

```
size_t hash_value(int v);
```

```
size_t hash_value(unsigned int v);
size_t hash_value(long v);
size_t hash_value(unsigned long v);
```

Returns: v.
Throws: nothing.

```
size_t hash_value(float v);
size_t hash_value(double v);
size_t hash_value(long double v);

template<class T> size_t hash_value(T * v);
```

Returns: an unspecified value, except that equal arguments shall yield the same result.
Throws: nothing.

```
template<class A, class B>
size_t hash_value(std::pair<A, B> const & v);
```

Effects:

size_t seed = 0;
hash_combine(seed, v.first);
hash_combine(seed, v.second);
return seed;

```
template<class E, class T, class A>
size_t hash_value(std::basic_string<E, T, A> const & v);
template<class T, class A>
size_t hash_value(std::vector<T, A> const & v);
template<class T, class A>
size_t hash_value(std::list<T, A> const & v);
template<class T, class A>
size_t hash_value(std::deque<T, A> const & v);
template<class K, class C, class A>
size_t hash_value(std::set<K, C, A> const & v);
template<class K, class C, class A>
size_t hash_value(std::multiset<K, C, A> const & v);
template<class K, class T, class C, class A>
size_t hash_value(std::map<K, T, C, A> const & v);
template<class K, class T, class C, class A>
size_t hash_value(std::multimap<K, T, C, A> const & v);
```

Returns: hash_range(v.begin(), v.end()).

Add a new section, Template functions hash_combine and hash_range, with the following contents:

```
template<class T>
```

```
void hash_combine(size_t & seed, T const & v);
```

Effects: seed ^= hash_value(v) + (seed << 6) + (seed >> 2);
Notes: hash_value is called without qualification.

```
template<class It> size_t hash_range(It first, It last);
```

Effects:

```
size_t seed = 0;

for( ; first != last; ++first )
{
   hash_combine( seed, *first );
}

return seed;
```

[hash_combine is fully specified. This is a tradeoff that guarantees identical results on different implementations. The alternative would be "implementation defined". This allows implementations to do better; but it also allows them to do worse, and it means that performance tests on a certain key set will not be portable.]

Replace [tr.unord.hash]/2 with:

```
std::size_t operator()(T const & val) const;
```

Returns: hash_value(val).
Notes: hash_value is called without qualification.

[This mechanism allows whole hierarchies to be handled with a single hash_value overload, as in the following example:

```
struct Hashable
{
   virtual void size_t hashValue() const = 0;
};

size_t hash_value(Hashable const & v)
{
   return v.hashValue();
}
```

Now every type that derives from Hashable is automatically usable as a key in an unordered associative container.]

Remove the explicit specializations of the hash class template.

Rationale:
The proposed implementation of hash_combine is derived from a string hashing function from:

Methods for Identifying Versioned and Plagiarised Documents
Timothy C. Hoad, Justin Zobel

http://www.cs.rmit.edu.au/~jz/fulltext/jasist-tch.pdf

I have been able to (sometimes dramatically) improve the performance of some hash table implementations by replacing the built-in hash function with it.

It references:

M.V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In Proc. Int. Conf. on Database Systems for Advanced Applications, pages 215-223, Melbourne, Australia, April 1997.

http://www.cs.rmit.edu.au/~jz/fulltext/dasfaa97.ps
as a source.

Rationale:
The LWG agreed that there are usability problems with the current interface, but thought that this proposal was too large, and raised too many design issues, for it to be accepted for the TR. A library of hashing primitives might be a good candidate for TR2.

7 Regular expression issues

7.1 basic_regex should Not Keep a Copy of its Initializer

Submitter: Pete Becker (N1499)
Status: TR

The basic_regex template has a member function str which returns a string object that holds the text used to initialize the basic_regex object. It also provides a container-like interface to this text through the member functionsbegin and end, which return const_iterator objects that allow inspection of the initializer text. While it might occasionally be useful to look at the initializer string, we ought to apply the rule that you don't pay for it if you don't use it. Just as fstream objects don't carry around the file name that they were opened with, basic_regex objects should not carry around their initializer text. If someone needs to keep track of that text they can write a class that holds the text and the basic_regex object.

Resolution:
As described in N1551, Changes to N1540 to Implement N1499 Parts 1 and 2.

7.2 basic_regex Should Not Have an Allocator

Submitter: Pete Becker (N1499)
Status: TR

The basic_regex template takes an argument that defines a type for an allocator object. The template also has several member typedefs and one member function to provide information about the allocator type and the allocator object. This is because a basic_regex object "is in effect both a container of characters, and a container of states, as such an allocator parameter is appropriate." Calling it a container doesn't make it one. The allocator in basic_regex is not very useful, and it unduly complicates the implementation.

The cost of using an allocator is high. Every type that the basic_regex object uses internally must have its own allocator type and its own allocator object. A node based implementation might have a dozen or more node types, requiring a dozen or more allocator objects. Allocator objects can be created as local objects when needed, which effectively precludes allocators with internal state; they can be ordinary members of the basic_regex object, inflating its size; or they can be implemented as a chain of base classes (to take advantage of the zero-size base optimization), with a high cost in readability and maintainability. None of these options is attractive.

Further, it's not at all clear how a user can determine that a substitute allocator is appopriate or what characteristics such an allocator should have. The STL containers have clearly spelled out requirements for their memory usage;basic_regex objects have no such requirements (nor should they). The implementor of the basic_regex template knows best what its memory requirements are.

Resolution:
As described in N1551, Changes to N1540 to Implement N1499 Parts 1 and 2. Some memory management interface may be a good idea, but allocators aren't it.

7.3 The Interface to regex_traits Should Use Iterators, Not Strings

Submitter: Pete Becker (N1499)
Status: TR

The member functions of the regex_trait template support customization and internationalization for regular expressions. Of these, the member functions transform, transform_primary, lookup_collatename, andlookup_classname take string as input.

This interface is inherently inefficient -- it requires creating a string object from a sequence in order to pass that string to the function. Further, in the case of transform, the function typically extracts iterators from the string object. Passing the text as a pair of iterators avoids introducing unnecessary string objects.

Resolution:
Apply the resolution from N1623=04-0063, Resolutions to regular expression issues.

7.4 Regular expressions and internationalization

Submitter: Pete Becker (N1500)
Status: TR

See N1500 for a detailed description. Summary: We're basing regexps on ECMAScript. However, ECMAScript is entirely unicode and doesn't deal with multiple locales and such. We're using it in a non-unicode environment. Some of the lookups it's asking for, e.g. asking whether a character is a digit in a locale-dependent way, are very expensive.

We allow metacharacters to be remapped, and (via the *translate* member function) even ordinary characters may be remapped. Remapping metacharacters means you can't tell what a regexp does just be looking at it. Remapping ordinary characters means that we use an expensive code path for all matches, even ordinary case sensitive matches.

Suggestions:

- Don't use translate for case-sensitive matches. (Or at least only use it if we're using the *collate* option when compiling the regex string into the regex object.
- Get rid of the *syntax_type* function that allows you to remap the meaning of metacharacters.

Resolution:
Apply the resolution from N1623=04-0063, Resolutions to regular expression issues.

7.5 Bad rationale for regex_ prefixes

Submitter: Pete Becker (N1507)
Status: NAD

Pete writes:

> I'm not strongly for or against the regex_ prefixes. They may well be helpful in understanding code. But I'm strongly against the notion that the standard library should use prefixes because users abuse using declarations.

Resolution: NAD. The rationale isn't part of the TR. If we decide to change the names, that will be a separate issue.

7.6 Unintended occurrence of reg_expression

Submitter: John Maddock (N1507)
Status: TR

There is a systematic error in the "proposed text" section: the various algorithms have been defined to accept a type "reg_expression" which does not in fact exist in the proposal, and which should of course be called "basic_regex". This is an editing error that crept in when the name of that class was changed from reg_expression to basic_regex.

The fix is to just replace all occurrences of "reg_expression" with "basic_regex" throughout that section.

Resolution: As above.

7.7 Iterators have incorrect definitions of the types “reference” and “pointer”

Submitter: John Maddock (N1507)
Status: TR

In regex_iterator and regex_token_iterator the definitions given for the types "iterator" and "reference" are wrong: as given these types refer/point to the value_type of the underlying iterator type, but should of course refer/point to the actual value_type being enumerated (the two are not the same type).

Resolution:
Change:

```
typedef typename
iterator_traits<BidirectionalIterator>::pointer
        pointer;
typedef typename
iterator_traits<BidirectionalIterator>::reference
        reference;
```

To:

```
typedef const value_type* pointer;
typedef const value_type& reference;
```

In both the regex_iterator and regex_token_iterator definitions.

7.8 regex_iterator does not handle zero-length matches correctly

Submitter: John Maddock (N1507)
Status: TR

There is a subtle bug in regex_iterator::operator++; when the previous match found matched a zero-length string, then the iterator needs to take special action to avoid going into an infinite loop, the current wording does this but gets it wrong because it does not allow two consecutive zero length matches, for example iterating occurrences of “^” in the text “\n\n” yields just one match rather than three as it should. The actual behavior should be as follows:

When the previous match was of zero length, then check to see if there is a non-zero-length match starting at the same position, otherwise move one position to the right of the last match (if such a position exists), and continue searching as normal for a (possibly zero length) match.

Resolution:
Covered by the proposed resolution to issue 7.9.

7.9 Regex_iterator does not set match_results::postion correctly

Submitter: John Maddock (N1507)
Status: TR

As currently specified, given:

```
regex_iterator<something> i;
```

then `i->position() == i->prefix().length()` for all matches found.

This is correct for the first match found, but makes little sense for subsequent matches where the result of i->position() is only useful if it returns the distance from the start of the string being searched to the start of the match found.

(Recall that i->prefix() contains everything from the end of the last match found, to the start of the current match, this allows search and replace operations to be constructed by copying i->prefix() unchanged to output, and then outputting a modified version of whatever matched.)

For example this problem showed up when converting a boost.regex example program from the regex_grep algorithm (not part of the proposal) to use regex_iterator: the example takes the contents of a C++ source file as a string, and creates an index that maps C++ class names to file positions in the form of a std::map<std::string, int>. In order for the program to take a regex_iterator and from that add an item to the index, it needs to know how far it is from the start of the text being searched to the start of the current match: that was what regex_match::position() was intended for, but as the proposal stands it instead returns the distance from the end of the last match to the start of the current match.

Resolution:
[Note: Discussed at Kona. General agreement that this is a real issue, also that the proposed resolution in N1507 was not the right way to resolve it. This is the new proposed resolution.]

Change:

```
private:
match_results<BidirectionalIterator> what; // exposition only
  BidirectionalIterator end; // exposition only
  const regex_type* pre; // exposition only
  match_flag_type flags; // exposition only
};
```

To:

```
private:
// these members are shown for exposition only:
BidirectionalIterator begin, end;
regex_type *pregex;
regex_constants::match_flag_type flags;
match_results<BidirectionalIterator> match;
};
```

And then add the following immediately afterwards:

A regex_iterator object that is not an *end-of-sequence iterator* holds a *zero-length match* if match[0].matched == true and match[0].first == match[0].second. [Note: this occurs when the part of the regular expression that matched consists only of an assertion (such as '^', '$', '\b', '\B')].

Then change the following members as shown:

regex_iterator constructors [tr.re.regiter.cnstr]
regex_iterator();

Effects: Constructs the end-of-sequence iterator.
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
regex_constants::match_flag_type f = regex_constants::match_default);

Effects: Initializes begin and end to point to the beginning and the end of the target sequence, sets pregex to &re, sets flags to f, then calls regex_search(begin, end, match, *pregex, flags). If this call returns false the constructor sets *this to the *end-of-sequence iterator*.
regex_iterator comparisons [tr.re.regexiter.comp]
bool operator==(const regex_iterator& right);

Returns: true if *this and right are both *end-of-sequence iterators* or if begin == right.begin, end == right.end, pregex == right.pregex, flags == right.flags, and match[0] == right.match[0], otherwise false.
bool operator!=(const regex_iterator& right);

Returns: !(*this == right)
regex_iterator dereference [tr.re.regexiter.deref]
const value_type& operator*();

Returns: match
const value_type* operator->();

Returns: &match
regex_iterator increment [tr.re.regexiter.incr]
regex_iterator& operator++();

Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.

If the iterator holds a *zero-length match* and start == end the operator sets *this to the *end-of-sequence iterator* and returns *this.

Otherwise, if the iterator holds a *zero-length match* the operator calls regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous). If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a *zero-length match*.

If the most recent match was not a *zero-length match*, the operator sets flags to flags | match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the *end-of-sequence iterator*. The iterator then returns *this.

In all cases in which the call to regex_search returns true match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0,match.size()) for which match[i].matched is true, match[i].position() shall return

distance(begin, match[i].first).

[Note: this means that match[i].position() gives the offset from the beginning of the target sequence, which is often not the same as the offset from the beginning of the sequence passed in the call to regex_search.]

It is unspecified how the implementation makes these adjustments.

[Note: this means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of regex_search will not be called.]
regex_iterator operator++(int);

Effects:
regex_iterator tmp = *this;
++(*this);
return tmp;

7.10 Naming of basic_regex::getflags

Submitter: Pete Becker (N1507)
Status: TR

basic_regex has member functions named getflags and get_allocator. The latter is consistent with the use of the same name in STL containers. In general, it seems to me, the library tries to use an underscore to separate a verb from its object for names of this nature. That convention would mean that we should call the other one get_flags. On the other hand, we do have getline, but that's arguably different because it's not a state query. Do we have a general policy here? If so, what is it, and what should the name of getflags be?

Resolution:
Replace all occurrences of “getflags” in the document with “flags”.

7.11 Missing namespace prefix in regex_iterator description

Submitter: Pete Becker (N1507)
Status: TR

The definition of regex_iterator in RE.8.1 mentions
regex_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re,
match_flag_type m = match_default);

And
match_flag_type flags; // for exposition only

match_flag_type and match_default are defined in the nested namespace regex_constants, so these two names need to be qualified with regex_constants::. Same thing in the first RE.8.1.1.

Resolution:
Go through the text and replace all occurrences of:

`match_flag_type` with `regex_constants::match_flag_type`,
`match_default` with `regex_constants::match_default`,
`match_partial` with `regex_constants::match_partial`,
`match_prev_avail` with `regex_constants::match_prev_avail`,
`match_not_null` with `regex_constants::match_not_null`,
`format_default` with `regex_constants::format_default`,
`format_no_copy` with `regex_constants::format_no_copy`,
`format_first_only` with `regex_constants::format_first_only`,
except in the section which defines these (RE.3.1).

7.12 Unnecessary sub-section headers in regex_iterator

Submitter: Pete Becker (N1507)
Status: editorial, TR

The first clause labeled RE.8.1.1 has the title "regex_iterator constructors". It contains descriptions of the constructors, plus several operators. The second clause labeled RE.8.1.1 has the title "regex_iterator dereference". It contains operator*, operator->, and the two versions of operator++. Seems like both of these labels should be removed.

Resolution:
Rename the section “RE.8.1.1 regex_iterator constructors” as “regex_iterator members”, remove the section “RE.8.1.1 regex_iterator dereference”, rename the section “RE.8.2.1 regex_iterator constructors” as “regex_token_iterator members”, remove the section: “RE.8.2.1 regex_token_iterator dereference”.

7.13 Names of symbolic constants

Submitter: Pete Becker (N1507)
Status: TR

ECMAScript has five control escapes: t, n, v, f, r. The regex proposal has named constants for four of them: escape_type_control_f, _n, _r, and _t. escape_type_control_v seems to be missing. (Okay, that's not about names, but the next two are).

This is minor, but in C and C++ those five things are escape sequences, and using names that include 'control' is a bit confusing. Granted, it fits with the terminology in ECMAScript, but I'd lean toward more C-like names, on the line ofescape_type_f.

And finally, there's escape_type_ascii_control. (For those not familiar with the details of the proposal, this refers to things that we might write in ordinary text as <ctrl>-X, for example.) We've pretty much avoided the term "ascii" in the standard (it's only used twice, in footnotes, apologetically), and I'm a bit uncomfortable with its use here. I'd preferescape_type_control_letter, which picks up the name of the production in the ECMAScript grammar for the letter that follows the escape. I think it's pretty clear what it means, and it avoids "ascii".

Resolution:
Replace all occurrences of:

escape_type_control_f with escape_type_f
escape_type_control_n with escape_type_n
escape_type_control_r with escape_type_r
escape_type_control_t with escape_type_t
escape_type_ascii_control with escape_type_control

Then immediately after the line:
static const escape_syntax_type escape_type_t;
add the line:
static const escape_syntax_type escape_type_v;

Then immediately after the table entry:
escape_type_t t

Add the new table entry:
escape_type_v v

[Kona: in addition to the proposed resolution in this issue: the LWG felt that a review of names throughout the regex clause is in order: the names tend to be verbose. See issue 7.41.]

7.14 Traits class versioning incompletely edited in.

Submitter: Pete Becker (N1507)
Status: TR

The paper talks about versioning of regex_traits classes, and RE.1.1 (in table RE2) says that a traits class shall have a member X::version_tag whose type is regex_traits_version_1_tag or a class that publicly inherits from that. Neither the <regex> synopsis (RE.2) nor the description of regex_traits (RE.3.3) mentions either of these types. I can't tell whether this was partially edited in or partially edited out. <g> So, is regex_traits versioning part of the proposal?

Resolution:
Edit this feature out, by removing the entry of X::version_tag in table 7.1.

7.15 Specification of sub_match::length incorrect

Submitter: John Maddock (N1507)
Status: TR

The specification for sub_match::length has acquired a couple of typos (a misplaced static, and the logic in the effects clause is back-to-front)

Resolution:
Change it to:

```
difference_type length();
```

Effects: returns `(matched ? distance(first, second) : 0)`.

*[Note to editor: throughout the regex section, we see "**Effects:** returns…" This is unnecessarily convoluted, and should be replaced with plan "**Returns:** …"]*

7.16 Traits class sentry language

Submitter: Pete Becker (N1507)
Status: TR

The proposal says:

> "An object of type regex_traits<charT>::sentry shall be constructed from a regex_traits object, and tested to be not equal to null, before any of the member functions of that object other than `length`, `getloc`, and `imbue` shall be called. Type sentry performs implementation defined initialization of the traits class object, and represents an opportunity for the traits class to cache data obtained from the locale object."

The first sentence is in passive voice, and begs the question of who shall do it: the user of the regex instance that holds the regex_traits object, or the regex instance itself. Unless the user is hacking around with a standalone instance ofregex_traits, it probably ought to be the regex object that "shall" do this.

Second, sentry "performs implementation defined initialization." I think this ought to be implementation specific, not implementation defined. I don't want to have to document the details of the initialization that sentry performs.

Additional comment from Pete Becker:

I think the right answer is to remove sentry. It doesn't really do much.

It's there to provide a way for the various search functions to ensure that the traits object has done any needed initialization. It's appropriate to defer such initialization, since it can involve allocation and population of tables and perhaps other expensive operations, which would be wasted if the user subsequently imbued a different locale.

The sentry class, though, is overkill. It's there in part by analogy to iostreams, where each inserter constructs a sentry object and checks its state before inserting into the stream. But that's part of the semantics of streams: if the stream's state is bad, attempted insertions simply do nothing, and program execution continues. Regular expressions, on the other hand, don't have that requirement. (It's not clear what should happen if initialization fails; the current requirement is only that whoever constructs the sentry object should check whether it succeeded). Further, in iostreams, one of the purposes of sentry is to be able to provide thread locking, with a lock in the constructor and an unlock in the destructor. There's no analogous need in regular expressions.

I think it should be up to the traits implementor to get initialization right. That means lazy initialization, and checking flags to be sure that caches have been set up. When a new locale is imbued, cached data becomes invalid. I don't think we need a hook to tell the traits object that it's time to initialize.

Resolution:

* Remove the entries for "X::sentry", "X::sentry s(u);" and "X::sentry(u)" from table 7.1.
* Remove the nested type "struct sentry" from the regex_traits class synopsis (7.7).
* Remove the description of "struct sentry" from the regex_traits description.
* Remove the sentence "No member functions other than length, getloc, and imbue may be called until an object of type sentry has been copy-constructed from *this.", from the description of regex_traits::imbue (7.7).

Rationale:

From John Maddock:

It appears that the motivation for the sentry object (a means to signal to the traits class that it is about to be used, and should therefore initialize itself now, caching loaded data as appropriate), is unnecessary. There are other techniques available (such as not constructing a traits class instance until it is actually needed, or have the traits class load it's localization data on demand) that can deal with the issue just as well, I would therefore propose that we remove this type altogether:

7.17 Imprecise specification of regex_traits::char_class_type

Submitter: Pete Becker (N1507)
Status: TR

Roughly speaking, there are three categories of character class: the ones that are supported by C and C++ locales (alnum, etc.), the additional ones for the regex proposal (d s w) and user-supplied character classes (through extensions to regex_traits).

Is the intent of the proposal to require that for the first category, the value returned by, for example,lookup_classname("alnum") be the value alnum as defined by ctype_base::mask? (I don't care one way or the other, but we have to be clear about what's required).

Resolution:
Replace:

> "The type `char_class_type` is used to represent a character classification and is capable of holding an implementation defined superset of the values held by ctype_base::mask (22.2.1)."

with:

> "The type `char_class_type` is used to represent a character classification and is capable of holding the implementation specific set of values returned by `lookup_classname`."

7.18 Can anything other than basic_regex throw bad_expression objects?

Submitter: Pete Becker (N1507)
Status: TR

The text describing the class bad_expresions says it is the type of the object thrown to report errors "during the conversion from a string ... to a finite state machine." This suggests that it is not thrown by the functions that try to match a string to and a basic_regex object, and this is borne out by the throws clauses for the constructors and assignment operators for basic_regex, which say that they throw bad_expression if the string isn't a valid regular expression, and by the lack of throws clauses for regex_match, etc.

On the other hand, error_type has two values, error_complexity and error_stack, that only occur during matching. There's no other mention of these values, so the only thing that can be done with them is for the implementation to pass them to regex_traits::error_string, and the only way the user can see the resulting string is by catching an exception. This suggests that bad_expression can also be thrown by the match functions. And the text says, in the last paragraph of RE.4, that "the functions described in this clause can report errors by throwing exceptions of type bad_expression."

So: can the various match functions throw bad_expression, and, if so, is bad_expression the appropriate name?

Resolution:
Apply the resolution from N1623=04-0063, Resolutions to regular expression issues.

7.19 Unneeded basic_regex members

Submitter: John Maddock
Status: TR

The following basic_regex members are redundant and should be removed:

```
basic_regex(const charT* p1, const charT* p2, flag_type f =
regex_constants::normal,
                    const Allocator& a = Allocator());
basic_regex& assign(const charT* first, const charT* last,
                                    flag_type f =
regex_constants::normal);
```

Resolution: As above.

7.20 Missing basic_regex members

Submitter: Pete Becker (N1507)
Status: TR

The proposal has member functions named 'assign' that take argument lists that correspond to the argument lists for constructors, with two exceptions: there's `basic_regex(const charT *, size_type len,flag_type)`, but no `assign(const charT *, size_type, flag_type)`; and there's `basic_regex()`, but no`assign()`. Are these omissions intentional?

Resolution:
add the following member to the basic_regex class synopsis:

basic_regex& assign(const charT* ptr, size_type len, flag_type f = regex_constants::normal);

Then add the following description in the RE4.5 section:

basic_regex& assign(const charT* ptr, size_type len, flag_type f = regex_constants::normal);

Effects: Returns assign(string_type(ptr, len), f).

7.21 Types of match_results typedefs members

Submitter: Pete Becker (N1507)
Status: TR

The proposal says that match_results has a nested typedef

```
typedef const value_type& const_reference
```

Since match_results has an allocator, this should be
`typedef typename allocator::const_reference const_reference`

`Resolution: As above`

7.22 What does match_results::size() return?

Submitter: Pete Becker (N1507)
Status: TR

The member funtion size() returns "the number of sub_match elements stored in *this". Aside from the suggested implementation above, there are the prefix() and suffix() sub_match elements. Is the intention that size() should return the number of capture groups in the original expression, and not include those two extra sub_matches? (I think the answer is probably yes).

Resolution:
Replace:

```
size_type size()const;
```

Effects: Returns the number of sub_match elements stored in *this.

With:

```
size_type size()const;
```

Effects: Returns one plus the number marked sub-expressions in the regular expression that was matched.
[Note to editor: put in the missing "of"]

7.23 What does match_results::position return when passed an out of range index?

Submitter: Pete Becker
Status: TR

match_results::position() doesn't say what happens when someone asks for the position of a non-matched group. The specification says that it's distance(first1, first2), where first1 is the beginning of the target text and first2 is the beginning of the nth match. The specification for sub_match says that for a failed match the iterators have unspecified contents. Do we want this to be unspecified or undefined, or is there some meaningful value we can return?

Having looked ahead <g>, the match and search algorithms specify that non-matched groups hold iterators that point to the end of the target text. This conflicts with the specification for sub_match, which says they're undefined. Is that text in sub_match incorrect?

Resolution:
Changes to:

```
difference_type position(unsigned int sub = 0) const;
```

Effects: Returns `std::distance(prefix().first, (*this)[sub].first)`.

Are covered in “Regex_iterator does not set match_results::postion correctly”.

Delete the following paragraphs from the sub_match specification:

> When the marked sub-expression denoted by an object of type sub_match<> participated in a regular expression match then member `matched` evaluates to true, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is false, and members `first` and`second` contained undefined values.
>
> If an object of type `sub_match<>` represents sub-expression 0 - that is to say the whole match - then member`matched` is always true, unless a partial match was obtained as a result of the flag `match_partial` being passed to a regular expression algorithm, in which case member `matched` is false, and members `first` and `second`represent the character range that formed the partial match.

The add the following to the match_results specification, immediately after the sentence ending *“except that only operations defined for const-qualified Sequences are supported.”*:

> The `sub_match<>` object stored at index zero represents sub-expression 0; that is to say the whole match. In this case the `sub_match<>` member `matched` is always true, unless a partial match was obtained as a result of the flag `regex_constants::match_partial` being passed to a regular expression algorithm, in which case member `matched` is false, and members `first` and `second` represent the character range that formed the partial match.
>
> The `sub_match<>` object stored at index *n* denotes what matched the marked sub-expression *n* within the matched expression. If the sub-expression *n* participated in a regular expression match then the `sub_match<>` member `matched` evaluates to true, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is false, and members `first` and `second` point to the end of sequence that was searched.

7.24 What happens if match_results::operator[] is out of range?

Submitter: Pete Becker
Status: TR

With respect to match_results::operator[]: We need to say what happens for an index out of range. Seems to me there are two reasonable possibilities: undefined behavior, or returns a no-match object.

While I strongly favor undefined behavior over artificially well-defined results, I also favor well-defined behavior when it is not too artificial. Thus, the behavior of sqrt(-2.0) is undefined; free(0) does nothing. While undefined behavior provides a convenient hook for debugging implementations, that's not its purpose, and if we can specify reasonable (which includes inexpensive) behavior we ought to do it, rather than provide another place where users can go astray.

In this case, I think I prefer to view operator[] as indexing into an unbounded array of sub_match objects. The objects at match_results.size() and above would look like failed sub-matches: their boolean flag would be false, and both their iterators would point to the end of the target string. Since we've agreed that sub_match objects for failed sub-matches need not have distinct addresses, this can be implemented by simply adding one sub_match element beyond those needed for the actual results, and returning it for an index that's otherwise out of bounds.

Resolution:
replace:

```
const_reference operator[](int n) const;
```

Effects: Returns a reference to the `sub_match` object representing the character sequence that matched marked sub-expression *n*. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression.

With:

```
const_reference operator[](int n) const;
```

Effects: Returns a reference to the `sub_match` object representing the character sequence that matched marked sub-expression *n*. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression. If n >= size() then returns a sub_match object representing an unmatched sub-expression.

7.25 Incorrect case insensitive match specification

Submitter: John Maddock (N1507)
Status: closed

The following wording:

> "During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range c1-c2 against a character c is conducted as follows: if getflags() &regex_constants::collate is true, then the character c is matched if traits_inst.transform(string_type(1,c1)) <= traits_inst.transform(string_type(1,c)) && traits_inst.transform(string_type(1,c)) <= traits_inst.transform(string_type(1,c2)), otherwise c is matched if c1 <= c && c <= c2. During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to a sort keys using traits::transform_primary, and then comparing the sort keys for equality."

Is defective in that it does not take account of case-insensitive matches, all input characters, and all collating elements in the finite state machine should be passed through traits.inst.translate before being converted into a sort key.

Resolution: Closed, this is covered by the issue 7.26.

7.26 Character class extensions to ECMAScript grammar need a formal grammar

Submitter: Pete Becker (N1507)
Status: TR

The regex proposal adds to ECMAScript the ability to use named character classes through "expressions of the form":

[[:class-name:]]
[[.collating-name.]]
[[=collating-name=]]

This isn't sufficient. In ECMAScript the expression [[] is valid, and names a character set containing the character '['. Similarly, [[:] is also valid, and names a character set containing the characters '[' and ':'. We need to say whether these two expressions (and their analogs for collating names) are still valid. I suspect the answer is that they're not -- a '[' as the first character in a character class is a special character, which must be follwed by one of ':', '.', or '=', then a name that does not contain any of ']', ':', ".', or '=' (technically we could allow ']', but that seems unnecessarily baroque), then the appropriate close marker.

Resolution: Adopt the proposed resolution in N1507.

7.27 Imprecise Specification of regex_replace

Submitter: Pete Becker (N1507)
Status: TR

Finds all the non-overlapping matches 'm' of type match_results<BidirectionalIterator> that occur in the sequence [first, last).

Having found them or not, it then writes stuff depending on its arguments. It's not clear, though, what "non-overlapping matches" are. It took me about five minutes to convince myself that these are matches of the complete expression, and not matches of internal capture groups (which would always overlap the full match). I think a footnote is sufficient for this. More important, though, is what happens when matches overlap. Suppose we're searching for "aba" in the text "ababa". There are two matches: the first three characters match, and the last three match. These two matches overlap. Do we discard them both? Keep the first? Keep the second? My guess is that the intention is to keep the first one, but we need to say so.

Resolution:
Replace the following clause:

Effects: Finds all the non-overlapping matches *m* of type `match_results<BidirectionalIterator>` that occur within the sequence [first, last). If no such matches are found and `!(flags & format_no_copy)` then calls `std::copy(first, last, out)`. Otherwise, for each match found, if `!(flags & format_no_copy)` calls `std::copy(m.prefix().first, m.prefix().last, out)`, and then calls `m.format(out, fmt, flags)`. Finally if `!(flags & format_no_copy)` calls

`std::copy(last_m.suffix().first,last_m,suffix().last, out)` where `last_m` is a copy of the last match found. If `flags &format_first_only` is non-zero then only the first match found is replaced.

With:

Effects: Constructs an regex_iterator object: `regex_iterator<BidirectionalIterator, charT, traits, Allocator> i(first, last, e, flags)`, and uses *i* to enumerate through all of the matches *m* of type`match_results<BidirectionalIterator>` that occur within the sequence [first, last). If no such matches are found and `!(flags & format_no_copy)` then calls `std::copy(first, last, out)`. Otherwise, for each match found, if `!(flags & format_no_copy)` calls `std::copy(m.prefix().first, m.prefix().last, out)`, and then calls `m.format(out, fmt, flags)`. Finally if `!(flags & format_no_copy)` calls`std::copy(last_m.suffix().first, last_m,suffix().last, out)` where `last_m` is a copy of the last match found. If `flags & format_first_only` is non-zero then only the first match found is replaced.

7.28 What is an invalid/empty regular expression?

Submitter: Pete Becker (N1507)
Status: TR

Resolution:

In [tr.re.regex.construct], change paragraph 1 from:

> Effects: Constructs an object of class basic_regex. The postconditions of this function are indicated in Table 19.

to:

> Effects: Constructs an object of class basic_regex that does not match any character sequence.

Remove Table 19.

In the declaration of basic_regex in [tr.re.regex], change:

```
// capacity
bool empty() const;
unsigned mark_count() const;
```

to:

```
// capacity
unsigned mark_count() const;
```

Remove the entry for empty() in the following tables:

- Table 20 [basic_regex(const charT* p, flag_type f) effects]
- Table 21 [basic_regex(const charT* p1, size_type len, flag_type f) effects]
- Table 22 [basic_regex(const basic_regex& e) effects]
- Table 23 [basic_regex(const basic_string&) effects]

- Table 24 [basic_regex(ForwardIterator first, ForwardIterator last, flag_type f, const Allocator&) effects]
- Table 25 [basic_regex& assign(const basic_string<charT, string_traits, A>& s,flag_type f) effects]

[Also, the title of Table 24 still mentions an Allocator argument, which should have been removed in a previous edit]

Change the last sentence of [tr.re.regex.locale]/1 from:

Calls to imbue invalidate any currently contained regular expression.

to:

After a call to imbue the basic_regex object does not match any character sequence.

Remove [tr.re.regex.locale]/2, which reads:

Postcondition: empty() == true.

7.29 Regular expression constructor language

Submitter: Pete Becker (N1507)
Status: TR

For the basic_regex ctor that takes a const charT *p, the proposal says:
Effects: Constructs an object of class basic_regex; the object's internal finite state machine is constructed from the regular expression contained in the null-terminated string p...

p is not a null-terminated string. It is a pointer. The analogous phrasing for basic_string is:
Effects: Constructs an object of class basic_string and determines its initial string value from the array of charTof length traits::length(s) whose first element is designated by s ...

We need to maintain a similar level of formalism.

Resolution:
Replace the Effects clause for basic_regex(const charT*, flag_type) in tr.re.regex.construct with:

> Effects: Constructs an object of class basic_regex; the object's internal finite state machine is constructed from the regular expression contained in the array of charT of length char_traits<charT>::length(p) whose first element is designated by p, and interpreted according to the flags specified in f. The postconditions of this function are indicated in Table ??.

7.30 Incorrect usage of “undefined”

Submitter: Pete Becker (N1507)
Status: TR

In several places in the document the term “undefined” should be replaced by “unspecified”:

"Otherwise `matched` is false, and members `first` and `second` contained ***undefined*** values."

"If the function returns false, then the effect on parameter *m* is ***undefined***, otherwise the effects on parameter *m* are given in table RE18"

"If the function returns false, then the effect on parameter *m* is ***undefined***, otherwise the effects on parameter *m* are given in table RE19"

Resolution: As above

7.31 Incorrect usage of "implementation defined"

Submitter: Pete Becker (N1507)
Status: TR

In several places in the document the term "implementation defined" should be replaced by either "implementation specific" or "unspecified":

"Type sentry performs ***implementation defined*** initialization of the traits class object, and represents an opportunity for the traits class to cache data obtained from the locale object."
`"char_class_type lookup_classname(const string_type& name) const;`

Effects: returns an ***implementation defined*** value that represents the character classification `name"`

"Returns: converts f into a value `m` of type `ctype_base::mask` in an ***implementation defined*** manner"

"Effects: constructs an object `result` of type `int`. If `first == last` or if `is_class(*first,lookup_classname("d")) == false` then sets `result` equal to -1. Otherwise constructs a `basic_istream<charT>`object which uses an ***implementation defined*** stream buffer type which represents the character sequence [first,last), and sets the format flags on that object as appropriate for argument `radix`."

Resolution: As above

7.32 Are sub_match objects all unique?

Submitter: Pete Becker (N1507)
Status: NAD

Are sub_match objects for non-matched capture groups required to be distinct? I can picture amatch_type implementation that holds sub_match objects only for the capture groups that matched, and returns a generic no-match object for others. Is this intended to be legal? (My inclination is that it ought to be allowed, because I don't see any good reason not to allow it).

Resolution:

No, match objects are not guaranteed to be unique; the lack of a guarantee was intentional.
[Editorial issue: The editor should add a non-normative note pointing that out.]

7.33 How are Unicode escape sequences handled?

Submitter: Pete Becker (N1507)
Status: TR

ECMA-Script supports character escapes of the form "\uxxxx", where each 'x' is a hex digit. Each such escape sequence represents the character whose code point is the value of 'xxxx' translated to a number in the usual way. What do such character escapes mean when the character type for basic_regex is too small to hold that value? Do we intend to require multi-byte support here (I hope not)? Or is such a value invalid when the target character type is too small?

Resolution:
In tr.re.grammar, after the paragraph

> When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type *bad_expression*.

add the following paragraph:

> If the *CV* of a *UnicodeEscapeSequence* is greater than the largest value that can be held in an object of type `charT` the translator shall throw an exception object of type `bad_expression`. [Note: this means that values of the form "\uxxxx" that do not fit in a character are invalid.]

7.34 Meaning of the match_partial flag

Submitter: Pete Becker (N1507)
Status: TR

RE.3.1.2 says that the match_partial flag

> Specifies that if no match can be found, then it is acceptable to return a match [from, last) where from!=last, if there exists some sequence of characters [from,to) of which [from,last) is a prefix, and which would result in a full match.

Taking this literally, if I have the expression "a(?=b)(?!b)" and try to match it against "a", the partial match must fail, because the two assertions are contradictory. Is the matcher really required to do this sort of analysis of the expression, and determine that there is no possible continuation that could succeed?

From the name, I would think that partial_match would mean, roughly, that if you reach the end of the search text but are only partway through the regular expression, that's okay. So in the example above, the partial match would succeed. Is that what's intended here?

Comment from John Maddock, on use cases for this feature:

- Searching "infinite" texts: for example two real world use cases that Boost.regex has been put to, are searching a multi-gigabyte server log, and filtering the data passing through a socket. In these cases you can't possibly load all of the text into memory to search it, so

you load chunks into a buffer and search one chunk at a time. Then you need to know whether a match could have straddled two chunk boundaries: and that's what a partial match gives you, it tell you how much of the end of one chunk to hang onto before reading the next section.

- Data input validation: if the data in some field has to match some regex to be acceptable, some users what to check this character by character as it's entered - the question then becomes: “given some more input could we eventually match the expression,” again that's what a partial match gives you.

This still doesn’t give us a specification of the feature, but at least it gives us the motivation.

Resolution:
Remove `match_partial`.

Rationale:
The LWG agrees that this is a useful and implementable feature, but we have repeatedly tried and failed to give it an adequate specification. We hope that it will be possible to add this feature in a future version.

7.35 Name of regex_traits::is_class

Submitter: Pete Becker (N1507)
Status: TR

That name is confusing. I'd prefer inclass, or some variant. The function takes two arguments: a character and a character class, and tells you whether the character belongs to the class. is_class sounds too much like querying whether some object represents a character class.

Resolution:
Replace all occurances of "is_class" with "isctype".

7.36 Can traits::error_string be simplified?

Submitter: Pete Becker (N1507)
Status: TR

In the proposal, the template regex_traits has a member function error_string that takes an error code that indicates what error occurred and returns a string corresponding to that error, which is then used as the argument to the constructor for an exception object. Seems to me it would be simpler to have regex_traits simply provide a function that throws the exception, called with the error code. Is this string needed for anything else?

Resolution:
Covered by the resolution to 7.18.

Rationale:
The sense of the LWG is that we should rethink the error reporting policy. A bad_expression object should contain a flag that represents the error, not a string constructed from the flag. The string returned by what() should be left unspecified, and the error_string interface should

probably be thrown away entirely. (Programmers who want to test exception objects to find out the exact cause of the error find codes easier to work with than strings. Programmers who want to print diagnostics for users can supply their own code-to-string mechanism.)

7.37 Can traits::translate be improved?

Submitter: Pete Becker (N1507)
Status: TR

The regex_traits member function 'translate' is used when comparing a character in the pattern string with a character in the target string. It takes two arguments: the character to translate, and a boolean flag that indicates whether the translation should be case sensitive. So two characters are equal if

```
translate(pch, icase) == translate(tch, icase)
```

So with pattern text of "abcde", checking for a match would look something like this:

```
for (int i = 0; i < 5; ++i)
   if (translate(pch[i], icase) == translate(tch[i], icase))
      return false;
return true;
```

The implementation of regex_traits::translate in the library-supplied traits class is:

```
return (icase ? use_facet<ctype<charT> >(getloc()).tolower(ch)
: ch);
```

There's potential for a significant speedup, though, if case sensitive and case insensitive comparisons go through two different functions. The obvious transformation of the preceding loop would be:

```
if (icase)
   for (int i = 0; i < 5; ++i)
      if (translate_ic(pch[i]) == translate_ic(tch[i]))
         return false;
else
   for (int i = 0; i < 5; ++i)
      if (translate(pch[i]) == translate(tch[i]))
         return false;
return true;
```

For the default regex_traits class, the calls to translate in the second branch of the if statement would be inline calls to a translate function that simply returns its argument, so the loop turns into a sequence of direct comparisons, with no distractions from the possibility of case insensitivity. Further, since case sensitivity is determined by a flag that's set at the time the regular expression is compiled, one of the two branches of the outer if statement will always be unnecessary.

I made up the names 'translate_ic' and 'translate' for this e-mail. I'm not suggesting that we use them.

Resolution:

Closely related to issue 7.4, and covered by the resolution to that issue.

7.38 Improving on traits::toi

Submitter: Pete Becker (N1507)
Status: TR

It says, in part:

> If first == last or if is_class(*first, lookup_classname("d")) == false then sets result equal to -1.

And "d" by default is the digits 0-9. Since the radix for the conversion can be 8, 10, or 16, the condition involving "d" isn't right. For a hex value it precludes the value 'a0'. For an octal value it allows '90', but the ensuing conversion will fail. We need to find a different way to express this. The idea is to return -1 on a failed conversion, and the appropriate unsigned value on success.

And further: I'm starting to think that toi is too high level an interface. Regular expression grammars go character by character. For example, the value of a HexEscapeSequence (\xhh) is "(16 times the MV of the first hex digit) plus the MV of the second HexDigit". toi (hypertechnically) doesn't require that. In order to implement the specification literally, the regex parser needs to translate individual characters, not groups of characters, into values, and accumulate those values as appropriate. Thus, regex_traits ought to provide int value(charT ch),which returns -1 if isxdigit(ch) is false, otherwise the numeric value represented by the character.

And: I've just implemented it. Here are the changes I made:

- I removed escape_type_backref and escape_type_decimal
- I added escape_type_numeric (0-9)
- I added int regex_traits::value(charT ch, int base)

The first two aren't technically necessary for this change, but escape_type_backref is a bit misleading. ECMAScriptdoesn't restrict the number of capture groups, so \10 can be a valid back reference. This means thatescape_type_backref alone isn't sufficient. So I figured it's enough to know that you're starting a numeric constant (i.e.escape_type_numeric), and then you can use value() == -1 to determine when you've reached the end of a constant.

The second argument to value is needed in order to decide whether the character is a valid digit for the base. valuereturns -1 for an invalid digit, and the (unsigned) numeric value for a valid digit.

Resolution:
In tr.re.escsyn, remove escape_type_backref from the list of constants of type escape_syntax_type and from Table 7.5 (escape_syntax_type values in the C locale).

In tr.re.escsyn, change the "Equivalent characters" entry for escape_type_decimal from "0" to "0123456789".

In tr.re.req, Table 7.1 (regular expression traits class requirements), remove the entry for *v.toi(I1, I2, i)*.

In tr.re.req, add to Table 7.1 (regular expression traits class requirements)
the following entry:

v.value(c, I)	int	Returns the value represented by the digit *c* in base *I* if the character *c* is a valid digit in base *I*; otherwise returns -1. [Note: the value of *I* will only be 8, 10, or 16.]

In tr.re.traits, remove the declaration of the member function *toi* from
the definition of *regex_traits*.

In tr.re.traits, add the following declaration to the definition of *regex_traits*:

```
int value(charT ch, int radix) const;
```

In tr.re.traits, remove the synopsis

```
template<class InputIterator>
int toi(InputIterator& first, InputIterator last, int radix) const;
```

and the three following paragraphs (labeled Preconditions, Effects, and Postconditions).

In tr.re.traits, add the synopsis

```
int value(charT ch, int radix) const;
```

followed by the following text:

> Precondition: The value of *radix* shall be 8, 10, or 16.
> Returns: the value represented by the digit *ch* in base *radix* if the character *ch* is a valid digit in base *radix*; otherwise returns -1.

In tr.re.grammar, change the sentence

> Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling *traits_inst.toi*.

to

> Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling *traits_inst.value*.

7.39 Improving on traits::lookup_classname

Submitter: Pete Becker (N1507)
Status: Duplicate

I think this needs a change in specification. It returns a value that identifies the named character class identified by its string argument. The cases I'm concerned about are the ones with names like [:alnum:]. When the code encounters the opening [: it has to scan ahead for the matching :, pick up the characters in between, stuff them into a string, and call lookup_classname. This is a lot of wheel spinning. In particular, creating the string is expensive. If lookup_classname took

two iterators instead of a string it could simply look at the characters without the intervening string object.

Resolution:
This is a subset of something the LWG already agreed on in principle: using an iterator interface instead of a string interface. There's no need to discuss this subpart by itself.

7.40 match_results element access functions have incorrect parameter types

Submitter: Robert Klarer
Status: TR
Section: 7.9.3 [tr.re.results.acc]

The subscripting operator for match_results is declared as follows:

```
const_reference operator[](int n) const;
```

This declaration is inconsistent with std::vector<...>::operator[], and introduces the possibility that the function may be called incorrectly (using a negative argument).

A similar problem exists for the length(...), position(...), and str(...) members of match_results.

Proposed resolution:

change the declaraction of the subscripting operator for match_results from

```
const_reference operator[](int n) const;
```

to

```
const_reference operator[](size_type n) const;
```

change the declaration of the match_results member function length(...) from

```
difference_type length(int sub = 0) const;
```

to

```
difference_type length(size_type sub = 0) const;
```

change the declaration of the match_results member function position(...) from

```
difference_type position(unsigned int sub = 0) const;
```

to

```
difference_type position(size_type sub = 0) const;
```

change the declaration of the match_results member function str(...) from

```
string_type str(int sub = 0) const;
```

to

```
string_type str(size_type sub = 0) const;
```

7.41 Regex names should be reviewed

Submitter: Matt Austern
Status: Closed

This is an outgrowth of the Kona discussion of issue 7.13. Names throughout the regex section are rather verbose; this is partly, but not entirely, a result of the regex_ prefix that appears in so many places. We may want to consider a systematic renaming.

Resolution:
Possibly a good idea, but nobody has volunteered to do that review.

7.42 iterators have incorrect definition of difference_type

Submitter: Pete Becker
Status: TR

Issue 7.7 isn't quite complete. The fix that we made is to change the type of pointer from "iterator_traits<BidirectionalIterator>::pointer" to "const value_type*", and the corresponding change for reference. Looks like we missed difference_type, which needs a similar change.

Proposed resolution:
Change

```
typedef typename iterator_traits<BidirectionalIterator>::difference_type
difference_type;
```

to:

```
typedef ptrdiff_t difference_type;
```

in both [tr.re.regiter] and [tr.re.tokiter].

7.43 basic_regex::swap minor error

Submitter: Pete Becker
Status: TR

Postcondition: *this contains the characters that were in e, e contains the regular expression that was in *this.

Should be:

Postcondition: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

7.44 Too many syntax options

Submitter: Pete Becker
Status: TR

7.5.1 provides the following syntax options: normal, ECMAScript, JavaScript, JScript, basic, extended, awk, grep, egrep, sed, perl.

There are three issues here:

1. The first four mean the same thing, and sed is the same as basic. I think we ought to pick one name for each option, rather than have multiple ways of saying the same thing. I suggest that we remove normal, JavaScript, and JScript (this means changing the default 'normal' in a bunch of places to 'ECMAScript', but I think that's an improvement, since it no longer suggests that UNIX stuff is abnormal), and that we remove 'sed'.

2. basic, extended, awk, grep, egrep, sed, and perl are all optional. The requirement is that if the functionality is supported, then these are the names that should be used. I think this is too unpredictable; we should decide to require them, or to say nothing about them. Again, in the spirit of not demeaning UNIX, I think they ought to be required. (But see below)

3. perl "Specifies that the grammar recognized by the regular expression is an implementation defined extension of the normal syntax." The name is misleading, since such an extension doesn't have to be anything like perl. That aside, the option itself isn't useful, since it makes no portable guarantees. Conforming implementations can provide their own extensions with their own names, so reserving that name without detailed semantics doesn't benefit users. I think we should remove it.

Further comments from Pete Becker (paraphrased from c++std-lib-12781):

> ECMAScript is fundamentally different from the rest, all the others are fairly similar. Basic and extended have the same base syntax differ in a number of important ways. For example, basic has backreferences (like "\(abc\)d\1") and extended does not. Extended has alternation (like "a|b") and basic does not. Extended supports "*", "+", and "?" for repetition, basic only supports "*".
>
> Grep is a minor extension to basic, egrep and awk are minor extensions to extended. The awk extensions are conforming, however, and they're things "that most people probably assume are part of regular expressions."

Proposed resolution (N1623):
In section 7.5.1, eliminate the following syntax option types: normal, javascript, jscript, sed, perl.

7.45 Names recognized by regex_traits::lookup_classname

Submitter: Pete Becker
Status: TR

The effects clause for lookup_classname in [tr.re.traits] say, in part,

> At least the names "d", "w", "s", "alnum",
> "alpha", "blank", "cntrl", "digit", "graph",
> "lower", "print", "punct", "space",
> "upper" and "xdigit", shall be recognized.

In regex_traits<wchar_t> these names aren't valid strings. They need to be expressed as sequences of wide characters. There are two ways we can do that.

First, we can describe them as wide character strings directly. For regex_traits<wchar_t> this would be:

> At least the names L"d", L"w", L"s", L"alnum",
> L"alpha", L"blank", L"cntrl", L"digit", L"graph",

L"lower", L"print", L"punct", L"space",
L"upper" and L"xdigit", shall be recognized.

Second, we can describe them as char strings, translated at runtime:

At least the names "d", "w", "s", "alnum",
"alpha", "blank", "cntrl", "digit", "graph",
"lower", "print", "punct", "space",
"upper" and "xdigit", translated to wide
character strings by calling
use_facet<ctype<charT> >(getloc()).widen(name, name+strlen(name), tgt)
for a suitably sized array tgt, shall be recognized.

These mean two different things. The first is a compile-time translation, with an implementation-specific mapping. The second is, obviously, mapped according to the specified locale. The second is probably the one we want -- with the first it's hard for users to name those classes in their regular expressions.

The same thing applies to "For a character c" in the effects clauses for regex_traits::syntax_type and regex_traits::escape_syntax_type, to the class names and the '_' in the returns clause for regex_traits::is_class, and to the class names in the effects and postcondition clauses for regex_traits::toi.

Resolution (N1623):
In the entry for *lookup_classname* in table 7.1, remove the sentence "At least the names ... shall be recongnized."

In the Effects clause for *lookup_classname*, replace the sentence
At least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" shall be recognized.

with:
For *regex_traits<char>*, at least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" shall be recognized. For *regex_traits<wchar_t>*, at least the names L"d", L"w", L"s", L"alnum", L"alpha", L"blank", L"cntrl", L"digit", L"graph", L"lower", L"print", L"punct", L"space", L"upper" and L"xdigit" shall be recognized.

Resolution:
Basic question: how do we deal with names of character classes in the case of wide characters? Option one: specify string literals like L"alpha". Option two: specify some kind of runtime widening with facets. Option one is preferred: use wide literals. There is no evidence of a problem that can't be solved that way.

7.46 Name of error_subreg

Submitter: Pete Becker

Status: TR

error_subreg means that the regular expression had an invalid back reference. I just don't see how bad back reference turns into subreg. Should the name be changed to error_backref?

Proposed resolution:
Yes, make the change.

7.47 Interpretation of match_not_bol and match_not_eol

Submitter: Pete Becker
Status: TR

The entry for match_not_bol says "The expression "^" is not matched against the subsequence [first,first)." The entry for match_not_eol is analogous. This is somewhat unclear. One problem is that it really should refer to '^' when used in an expression, not to the expression "^" (which is a really boring regular expression). Another is that "is not matched" doesn't really convey what should happen.

Proposed resolution:
Replace the entry in the column "Effect if set" for match_not_bol, which currently reads

> The expression "^" is not matched against the subsequence [first,first)

with

> The first character in the sequence [first, last) is treated as though
> it is not at the beginning of a line, so the character '^' in the regular
> expression shall not match [first,first).

Replace the entry in the column "Effect if set" for match_not_eol, which currently reads

> The expression "^" is not matched against the subsequence [first,first)

with

> The last character in the sequence [first, last) is treated as though
> it is not at the end of a line, so the character '$' in the regular
> expression shall not match [last,last).

7.48 Changing the value type of regex_token_iterator

Submitter: Pete Becker
Status: TR

regex_token_iterator splits a text sequence into subsequences, using operator++ to move to the next subsequence. The subsequences are returned as string objects. Internally the iterator typically holds the result of the regular expression search in a match_results object, which has all

the information about the match that's needed to manage iteration and construct results. In order to support operator-> each iterator object must also hold a string object with the (internally redundant) value of the current subsequence, so that operator-> can return that string object's address.

The overhead of this string object can be removed by changing the iterator's value type from string to sub_match, which means that operator-> can return the address of a submatch object held in the match_results object., or by changing the value type to pair<BidirectionalIterator, BidirectionalIterator>, which is part of the corresponding submatch object. Users who need a string object can easily construct one from the pair of iterators.

Seems to me that the overhead of carrying around a redundant string object isn't justified by the ability to return a pointer to a string.

Resolution:
Rewrite 7.11.2 introduction as follows:

The class template regex_token_iterator is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a sub_match class template instance that represents what matched a particular sub-expression within the regular expression.

When class regex_token_iterator is used to enumerate a single sub-expression with index -1, then the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator creates and stores a value regex_iterator<BidirectionalIterator, charT, traits> position and sets the internal count N to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count N is incremented; if N exceeds or equals this->subs.size(), then the iterator increments member position and sets count N to zero.

If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1: In which case the iterator enumerates one last sub-expression that contains the iterator range from the end of the last regular expression match to the end of the input sequence being enumerated, provided this would not be an empty string.

The constructor with no arguments, regex_iterator(), always constructs an end of sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end of sequence is not defined. For any other iterator value a const sub_match<BidirectionalIterator>& is returned. The result of operator-> on an end of sequence is not defined. For any other iterator value a const sub_match<BidirectionalIterator>* is returned.

It is impossible to store things into regex_iterator's. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-

of-sequence iterators are equal when they are constructed from the same arguments.

Change:

typedef basic_string<charT>
value_type;

To:
typedef sub_match<BidirectionalIterator> value_type;

Change:

private:
match_results<BidirectionalIterator> what; // exposition only
BidirectionalIterator end; // exposition only
const regex_type* pre; // exposition only
match_flag_type flags; // exposition only
basic_string<charT> result; // exposition only
std::size_t N; // exposition only
std::vector<int> subs; // exposition only
};

To:

```
private: // data members for exposition only:
  typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator;
  position_iterator position;
  const value_type *result;
  value_type suffix;
  std::size_t N;
  std::vector<int> subs;
};
```

And add the following immediately afterwards:

A *suffix iterator* points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member result holds a pointer to the data member suffix, the value of the member suffix.match is true, suffix.first points to the beginning of the final sequence, and suffix.second points tothe end of the final sequence.

[Note – for a suffix iterator, data member suffix.first is the same as the end of the last match found, and suffix.second is the same as the end of the target sequence – end note]

The *current match* is (*position).prefix() if subs[N] == -1, or (*position)[subs[N]] for any other value of subs[N].

Then change member function definitions as follows:
regex_token_iterator constructors [tr.re.tokiter.cnstr]

regex_token_iterator();

Effects: Constructs the end-of-sequence iterator.
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, int submatch = 0, regex_constants::match_flag_type f = regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const vector<int>& submatches,
regex_constants::match_flag_type f = regex_constants::match_default);
template<std::size_t R>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const int (&submatches)[R],
regex_constants::match_flag_type f = regex_constants::match_default);

Effects: The first constructor initializes the member subs to hold the single value submatch. The second constructor initializes the member subs to hold a copy of the argument submatches. The third constructor sets the member subs to hold a copy of the sequence of integer values pointed to by the iterator range [&submatches, &submatches + R).

Each constructor then sets N to 0, and position to position_iterator(a, b, re, f). If position is not an end-of-sequence iterator the constructor sets result to the address of the current match. Otherwise if any of the values stored in subs is equal to -1 the constructor sets *this to a suffix iterator that points to the range [a,b), otherwise the constructor sets *this to an end-of-sequence iterator.
regex_token_iterator comparisons [tr.re.tokiter.comp]
bool operator==(const regex_token_iterator& right);

Returns: true if *this and right are both end-of-sequence iterators, or if *this and right are both suffix iterators and suffix == right.suffix; it returns false if *this or right is an end-of-sequence iterator or a suffix iterator. Otherwise returns true if position == right.position, N == right.N, and subs == right.subs.
bool operator!=(const regex_token_iterator& right);

Returns: !(*this == right);
regex_token_iterator dereference [tr.re.tokiter.deref]
const value_type& operator*();

Returns: *result
const value_type *operator->();

Returns: result
regex_token_iterator increment [tr.re.tokiter.incr]
regex_token_iterator& operator++();

Effects: Constructs a local variable prev of type position_iterator and initializes it with the value

of position. If *this is a suffix iterator, sets *this to an end-of-sequence iterator.

Otherwise, if N+1 < subs.size(), increments N and sets result to the address of the current match.

Otherwise, sets N to 0 and increments position. If position is not an end-of-sequence iterator the operator sets result to the address of the current match.

Otherwise if any of the values stored in subs is equal to -1 and prev.suffix().length() is not 0 the operator sets *this to a suffix iterator that points to the range [prev.suffix().first, prev.suffix().second).

Otherwise sets *this to an end-of-sequence iterator.

7.49 Descriptions of comparison operators missing

Submitter: John Maddock
Status: TR

The synopsis for the <regex> header (7.4) includes comparison operators between objects of type "specialization of sub_match" and objects of type "specialization of basic_string", for example:

```
template <class BidirectionalIterator, class traits,
          class Allocator>
bool operator == (
  const
std::basic_string<iterator_traits<BidirectionalIterator>
                                              ::value_type,
                          traits, Allocator>& lhs,
  const sub_match<BidirectionalIterator>& rhs);
```

However due to an editing error, detailed descriptions for these template comparison operators were omitted from the sub_match section (7.8.11), which is a pity, since these are the arguably more important than the comparison operators which are described in detail.

Resolution:

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator == (const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```

Returns: lhs == rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator != (const
```

```
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs != rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator < (const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs < rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator > (const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs > rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator >= (const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs >= rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator <= (const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits,
Allocator>& lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs <= rhs.str().

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() == rhs.

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() != rhs.

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() < rhs.

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() > rhs.

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() >= rhs.

```
template <class BidirectionalIterator, class traits, class
Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
const
basic_string<iterator_traits<BidirectionalIterator>::value_type,
traits, Allocator>& rhs);
```
Returns: lhs.str() <= rhs.

7.50 Convenience typedefs for regex_iterator and regex_token_iterator

Submitter: John Maddock
Status: TR

The match_results class template has the following typedefs defined for it, on the grounds that these template instances are used sufficiently frequently to make them useful, indeed these

particular template instances are used almost to the exclusion of all others (a bit like std::string and std::wstring):

```
typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

However the class templates regex_iterator and regex_token_iterator have no such typedefs defined for them, in spite of the fact that these are also almost always instantiated for the same types as match_results is. I would like to propose that the following typedefs are added to section 7.4:

After:

```
template <class BidirectionalIterator,
class charT =
iterator_traits<BidirectionalIterator>::value_type,
class traits = regex_traits<charT>,
class Allocator = allocator<charT> >
class regex_iterator;
```

add:

```
typedef regex_iterator<const char*>
cregex_iterator;typedef
regex_iterator<std::string::const_iterator>
sregex_iterator;typedef
regex_iterator<const wchar_t*>
wcregex_iterator;typedef
regex_iterator<std::wstring::const_iterator> wsregex_iterator;
```

after:

```
template <class BidirectionalIterator,
class charT =
iterator_traits<BidirectionalIterator>::value_type,
class traits = regex_traits<charT>,
class Allocator = allocator<charT> >
class regex_token_iterator;
```

add:

```
typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<std::string::const_iterator>
sregex_token_iterator;
typedef regex_token_iterator<const wchar_t*>
wcregex_token_iterator;
typedef regex_token_iterator<<std::wstring::const_iterator>
```

```
wsregex_token_iterator;
```

Finally, while we're at it, the corresponding typedefs for sub_match couldbe added as well:

```
typedef sub_match<const char*> csub_match;
typedef sub_match<const wchar_t*> wcsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;
```

7.51 Do basic_string comparison operators mandate an inefficient implementation?

Submitter: John Maddock
Status: Closed

The current text for the basic_string comparison operators has definitions such as:

```
template<class charT, class traits, class Allocator>
bool operator==(const charT* lhs, const
basic_string<charT,traits,Allocator>& rhs);
```
Returns: basic_string<charT,traits,Allocator>(lhs) == rhs.

A particularly literalist interpretation of this, would result in an unnecessarily inefficient implementation which created a temporary string object, even though a more efficient iterator-based comparison (with identical comparison semantics) is possible. I believe that a specialization of basic_string could conceivably detect which implementation technique is used. Likewise in <regex> we have proposed:

```
template <class BidirectionalIterator>
bool operator == (typename
iterator_traits<BidirectionalIterator>::value_type const* lhs,
const sub_match<BidirectionalIterator>& rhs);
```
Returns: lhs == rhs.str().

Which would result in two basic_string temporaries being created (one by the sub_match comparison operator and one by the basic_string operator to which it delegates).

In both of these cases, I think the text is clear, concise and to the point, and I don't see any better way of expressing the semantics involved, but do we need to clarify how much latitude in interpreting the "as if" rule implementers have, or am I being unnecessarily pedantic?

Rationale:
This should be filed as an issue against clause 17. It's not really an issue with the TR, and in fact the very first example in this issue applies to clause 21 of the standard, not something in the TR.

But it also appears to be a non-issue. A "Returns" clause says what value is returned, not how to compute it. Possibly clause 17 should be clarified (with a non-normative note) to say, once more, that a "returns" clause does not require side effects. Possibly the "effects" clause isn't clear enough on whether it requires side effects, but "returns" seems adequately clear.

7.52 Resolution to DR 7.1 was incomplete

Submitter: John Maddock
Status: TR

The resolution to issue TR.DR.7.1 was incomplete, in particular since the member functions str() and size() have been removed from basic_regex, then the entries for these members must be removed from the tables 7.7, 7.8, 7.9, 7.10, 7.11, 7.12 and 7.13.

7.53 Resolution to DR 7.22 was incomplete

Submitter: John Maddock
Status: TR

The changes made to solve issue 7.22 were incomplete, specifically:

p175 section 7.11.1, table 7.16: the entry for m.size() should read:

1 + e.mark_count()

This was addressed in issue 7.22, but we missed the fact that this table also has to change to match the change for match_results::size().

p177 section 7.11.2, table 7.17: the entry for m.size() should read:

1 + e.mark_count()

This was addressed in issue 7.22, but we missed the fact that this table also has to change to match the change for match_results::size().

7.54 Format_no_copy specified incorrectly

Submitter: Pete Becker
Status: TR

In [tr.re.matchflag] (7.5.2) the specification for `format_no_copy` says:

During a search and replace operation, sections of the character container sequence being searched that do match the regular expression are not copied to the output string.

This should be "... that do **not** match ... ".

7.55 typo in regex_token_iterator::operator++

Submitter: Pete Becker
Status: TR

In [tr.re.tokiter.incr] (7.12.2.4), the first paragraph of the Effects: clause describes a variable of type position_iterator named 'prev'. In the next to last paragraph of the Effects: clause we refer to 'prev.suffix()' in three places. These should be 'prev->suffix()'.

7.56 match_results stream inserter not specified

Submitter: Pete Becker
Status: TR

In [tr.re.syn] (7.2.4) the synopsis includes

```
template <class charT, class traits,
class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
operator << (basic_ostream<charT, traits>& os,
const match_results<BidirectionalIterator, Allocator>& m);
```

but there is no specification for this function. That's also the case in N1429. There's no obvious meaning for this function, so unless John disagrees I recommend we remove it from the synopsis.

Further comments from John Maddock:
It's worse than that: I appear to have been omitted all of the match_results non-members:

```
operator ==
operator!=
operator <<
and non-member swap.
```

However, having done a quick double check, it appears that containers like vector<> rely on the container/sequence requirements for the semantics of operator== and operator!=, and don't explicitly document them anywhere, so it appears that we can do the same here. The other two should be documented though, and semantics are straightforward.

Proposed resolution:
Add the following:

```
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: `m1.swap(m2);`

Remove the `match_results` insertion operator from the synopsis.

Rationale:
The omission of swap is just a goof. It's not at all clear, though, that we need an insertion operator. Inserting just m[0] seems arbitrary, and users who want to insert that component of a match_results object to a stream can just do it directly.

7.57 Imprecise Specification of match_results::size

Submitter: Pete Becker

Status: TR

TR1 currently says:

```
size_type size() const;
```
Returns: One plus the number of marked sub-expressions in the regular expression that was matched.

But match_results has a default constructor, with the postcondition that `size() == 0`. In that case no regular expression was matched. Also, in the phrase "the regular expression that was matched", the last word suggests that the search must have succeeded.

Proposed resolution:
In [tr.re.results.size] (7.10.2), change the Returns clause for the size member function from "Returns: One plus the number of marked sub-expressions in the regular expression that was matched." to "Returns: One plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0. [Note: The state of a match_results object can be modified only by passing that object to regex_match or regex_search. Sections tr.re.alg.match and tr.re.alg.search specify the effects of those algorithms on their match_results arguments.]"

In tr.re.alg.match (7.11.2) paragraph 3, change "the effect on parameter m is unspecified" to "the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true"

In tr.re.alg.search (7.11.3) paragraph 3, change "the effect on parameter m is unspecified" to "the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true"

7.58 Does match_results<>::begin() point to element 0?

Submitter: Matt Austern
Status: TR

Class template `match_results` respresents a collection of `sub_match` objects. Index 0 represents the match for the entire expression, and a nonzero index n represents the match for the n^{th} marked subexpression.

The description of `match_results<>::begin()` (7.10.3) says that it returns “A starting iterator that enumerates over all the marked sub-expression matches stored in *this..” It isn’t clear whether the object stored at index 0, a match for the entire expression, counts as a “marked sub-expression match” for the purposes of `begin()`.

Proposed resolution:
In the description of match_results<>::begin() and match_results<>::end in 7.10.3, change "enumerates over all the marked sub-expression matches stored in *this" to "enumerates over all the sub-expressions stored in *this".

7.59 “Implementation defined” in regex traits requirements

Submitter: Matt Austern

Status: TR

The regular expression traits requirements (table 15) says that the type of X::locale_type and X::char_class_type is “implementation defined.” This probably isn’t what was intended. These types aren’t defined by the standard library implementer. A traits class is supposed to be able to define those types; what we need to say in the requirements table is what the requirements on those types are.

Proposed resolution:
Change the “return type” for X::locale_type to “A copy constructible type,” and change the note to “A type that represents the local used by the traits class.” Change the “return type” for X::char_class_type to “a bitmask type” [lib.bitmask.types].” Change the note to be “A bitmask type representing a particular character classification.”

7.60 Requirements for v.isctype in regular expression traits

Submitter: Matt Austern
Status: TR

One of the expression in table 15 is v.isctype(c, v.lookup_classname(F1, F2)). This seems way too restrictive, especially since we're explicitly told that X::char_class_type is a bitmask type and that values can be or'ed together. We should just say that the second argument is a value of type char_class_type and leave it at that.

Resolution:
In [tr.re.req] (7.3) paragraph 4, add "cl is an object of type X::char_class_type". In table 15, change the expression "v.isctype(c, v.lookup_classname(F1, F2))" to "v.isctype(c, cl)" and change the note to "Returns true if character c is a member of one of the character classes designated by cl, false otherwise".

7.61 Definition of match_results::position

Submitter: John Maddock
Status: TR

In 7.10.3 the definition of position says:

Returns: std::distance(prefix().first, (*this)[sub].first).

Which is not correct when using a regex_iterator. We fixed the text for regex_iterator to account for this, but maybe we should do the same here, something like:

Returns: the distance from the start of the target sequence to the start of the match held by *this.

The current wording is wrong because the invariant given does not hold when the match_results struct comes from a regex_iterator. This is related to issue 7.9.

Proposed resolution:
In 7.10.3 change the definition of position from

Returns: std::distance(prefix().first, (*this)[sub].first).

to

Returns: The distance from the start of the target sequence to (*this)[sub].first.

7.62 Is regex_traits::size_type necessary?

Submitter: Matt Austern
Status: TR

Regex_traits defines size_type, but basic_regex doesn't use it. Is regex_traits::size_type still necessary?

Resolution:
Remove size_type from the regex trait requirements table (table 15) and from class template regex_traits (tr.re.traits, 7.7). In both places, change the return type of length() to std::size_t.

7.63 effects clause of regex algorithms and match_partial

Submitter: Eric Niebler
Status: TR

"Effects: Determines whether there is an exact match between the regular expression e, and all of the character sequence [first, last), where the parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a match exists, false otherwise."

The issue is the term "exact match". A partial match is not exact, so should regex_match return true or false for partial matches? There is a hedge in here ("the parameter flags is used to control how the expression is matched") but I'm not sure that makes it sufficiently clear.

Proposed resolution:
In the description of regex_match, change "Determines whether there is an exact match" to "Determines whether there is a match".

7.64 exception safety of assign(basic_string<>)

Submitter: Eric Niebler
Status: TR

The Effects clause does not say what happens to *this is the string does not contain a valid regular expression. Either the regex object remains unchanged or it should become a default-constructed regex. We should say.

Proposed resolution:
In the description of basic_regex::assign(string), add a note saying that if basic_regex::assign(string) throws, *this is unchanged.

7.65 We still have str() in basic_regex::assign

Submitter: Pete Becker

Status: TR

The specification of basic_regex::assign(const basic_regex&) says that it assigns the value of **str()**. There is no such member function. (There was, but it has been removed.)

Proposed resolution:
In 7.8.2/13 change "returns the result of `assign(e.str(), e.flags())`" to "returns the result of `assign(e)`".

In 7.8.4/1, change "Returns `assign(that.str(), that.flags())`" to "Copies `that` into `*this` and returns `*this`." Create a table to reflect the result of that operation, modeled on the copy constructor table.

8 Fixed-size array issues

8.1 Is "array" the right name?

Submitter: Robert Klarer
Status: NAD

The name `array` may be confusing, since `array<T>` is not in fact an array; the `is_array` type trait, for example, will return false for `array<T>`. (As it should.) Perhaps another name would make this less surprising.

Rationale:
Very few people have reason to care about the return value of `is_array`, and those people know that they're testing for builtin arrays as used in the core section of the standard. The only other suggestion that attracted any support was "block", and, by a 6-2 straw poll, the LWG preferred to keep the name "array" instead of changing it to "block".

8.2 Front() and back() for zero-sized arrays

Submitter: Alisdair Meredith
Status: TR

Zero-sized arrays are explicitly legal, and begin() and end() have well-defined semantics. However, the front() and back() operations, which return the first and last elements respectively, are meaningless if there is no element to be returned. What should their behavior be for zero-element arrays? Informally, the three choices are: mandate a compile-time error (*e.g.* remove them from the specialization), mandate an exception, and allow undefined behavior.

Proposed resolution: (see N1624)
Add a clause to section 6.2.2 [tr.array.array]: "The effect of calling front() or back() for a zero-sized array is implementation defined."

Discussion:
Sydney: We considered four options: (a) compile-time error. (b) exception. (c) implementation defined (d) undefined behavior. Results: 0-1-5-2.

8.3 Should array<>::elems be exposed as part of the interface?

Submitter: Alisdair Meredith
Status: TR

The member variable containing the array elements must be public, since array<> is required to be a POD type. However, that doesn't necessarily mean that it has to be exposed to users. In principle, an implementation could hide it by giving it a name like `_M_elems`. Should we keep it as a public part of the interface with the name `elems`, or should we requirere implementers to hide it?

Proposed resolution:
Mark the name of this member variable as "exposition only" and put in a non-normative note saying that it is not part of array<>'s interface.

8.4 Should array<> be given a tuple interface?

Submitter: Howard Hinnant
Status: TR

Resolution: (see N1624)

```
6.1.1 Header <tuple> synopsis

Add:

template <class T, size_t N > struct array;

template <class T, size_t N>        struct
tuple_size<array<T, N> >;
template <int I, class T, size_t N> struct tuple_element<I,
array<T, N> >;

template <int I, class T, size_t N>       T& get(
array<T, N>&);
template <int I, class T, size_t N> const T& get(const
array<T, N>&);


6.1.4

Add a new section 6.1.4

tuple_size<array<T, N> >::value

Type:  integral constant expression.
Value:  N
```

```
tuple_element<I, array<T, N> >::type

Requires:  0 <= I < N.   The program is ill-formed if I is
out of bounds.
Value:  The type T.

template <int I, class T, size_t N> T& get(array<T, N>& a);

Requires: 0 <= I < N. The program is ill-formed if I is out
of bounds.
Return type:  T&.
Returns: A reference to the Ith element of a, where
indexing is zero-based.

template <int I, class T, size_t N> const T& get(const
array<T, N>& a);

Requires: 0 <= I < N. The program is ill-formed if I is out
of bounds.
Return type:  const T&.
Returns: A const reference to the Ith element of a, where
indexing is zero-based.
```

8.5 Tuple interface can't access all array elements

Submitter: Alisdair Meredith
Status: New

The number of elements in an `array` is of type `std::size_t`. The number of elements in an `tuple` is of type `int`.

The array-get syntax is an overload of the tuple-get syntax, so for example on a system with 32 bit integers and 64 bit address space, it is quite possible to create an array bigger than the get syntax could reach.

I missed the discussion on using int as bound for `tuple` rather than `unsigned` or `size_t`, and clearly that is way too big a change to occur.

Not sure if a DR could change the array overloads to use size_t instead though?

```
template <size_t I, class T, size_t N>
  T& get( array<T, N>&);
template <size_t I, class T, size_t N>
  const T& get(const array<T, N>&)
```

9 Iterator concept and adapter issues

This section has been removed, because iterator concepts and adapters have been removed from the TR.

10 Function object and reference_wrapper issues

10.1 Return types of reference wrapper functions

Submitter: Alisdair Meredith
Status: TR

c++std-lib-12598:
Hopefully just picking up a couple of typos

2.1.1 function templates ref and cref do not declare return types.
2.1.2 and 2.1.2.4: member functions operator() and get() do not declare return types.

All the above require clearly defined return values in later clauses, but current drafting suggests a header that will not compile.

Resolution:
It appears that the return types were mysteriously eaten somewhere between N1436 (the original proposal, pre-Oxford) and N1453 (post-Oxford). They should be:

```
  template<typename T> reference_wrapper<T> ref(T&);
  template<typename T> reference_wrapper<const T> cref(const
T&);
  operator T&() const;
  T& get() const;
```

10.2 Swapping functions

Submitter: Alisdair Meredith
Status: TR

c++std-lib-12603:
TR 3.4.3 declares a function template to swap functions of different type, with different allocators:

```
template< typename Function1, typename Allocator1,
          typename Function2, typename Allocator2 >
void swap( function< Function1, Allocator1> &f1,
           function< Function2, Allocator2 > & f2);
```

The effects clause is that this is equivalent to f1.swap(f2);

Yet IIUC, the member function swap is only defined for functions of the same type.

```
template<...> class function
{
  ...
  void swap( function & );
  ...
};
```

Resolution:
The synopsis in 3.4.1 is correct, as is the definition in 3.4.3.5. The synopsis for swap then shows up (incorrectly, as you point out) in 3.4.3 along with the function class template definition.

10.3 Should function wrapper take allocator template argument?

Submitter: Pete Becker
Status: TR

Some time back we discussed whether function objects should have allocators. In essence, the issue is that allocators are designed for use with containers, and function objects aren't containers. On the other hand, function objects typically allocate small blocks (if they allocate anything at all), and some applications could benefit from optimizing these allocations.

Proposed resolution:
Get rid of the allocators.

10.4 Argument passing for reference_wrapper::operator()

Submitter: Doug Gregor
Status: TR

The function call operator for class template reference_wrapper is declared as follows:

```
template <typename T1, typename T2, ..., typename TN>
  typename result_of<T(T1, T2, ..., TN)>::type
  operator()(T1, T2, ..., TN) const;
```

This means that arguments are copied when they are passed through reference_wrapper, which was an unintended consequence of an editorial error introduced in N1453 (relative to N1436). Class template reference_wrapper should follow the same argument-forwarding procedures as the function object binders (TR 3.3) by accepting parameters via non-const reference.

Resolution:
Replace the above declaration in 2.1.2.3 and the summary in 2.1.2 with the following declaration:

```
template <typename T1, typename T2, ..., typename TN>
  typename result_of<T(T1, T2, ..., TN)>::type
  operator()(T1, T2, ..., TN) const;
```

Note that if the proposed resolution to issue #10.TBD is accepted, the declaration should be replaced with:

```
template <typename T1, typename T2, ..., typename TN>
  typename result_of<T(T1, T2, ..., TN)>::type
  operator()(T1&, T2&, ..., TN&) const;
```

Unclear whether changing to the reference version is good enough: don't we need const reference too, to bind to rvalues? If so, doesn't this imply 2^N versions? This is an instance of the forwarding problem.

10.5 Callable definition does not match function<> semantics

Submitter: Doug Gregor
Status: TR

In section 3.4.3, the definition of Callable uses static_cast in an unsafe manner, introducing unsafe downcasts. Example:

```
class A {};
class D : public A {};

A* f();
function<D*(void)> g;
g = f; // compiles, but with a dangerous cast from A* to D*
```

Resolution:

Replace the following paragraph in 3.4.3:

"A function object f of type F is Callable given a set of argument types T1, T2, ..., TN and a return type R, if the appropriate following function definition is well-formed:

```
// If F is not a pointer to member function
R callable(F& f, T1 t1, T2 t2, ..., TN tN)
{
  return static_cast<R>(f(t1, t2, ..., tN));
}

// If F is a pointer to member function
R callable(F f, T1 t1, T2 t2, ..., TN tN)
{
  return static_cast<R>(((*t1).*f)(t2, t3, ..., tN));
}"
```

with

"A function object f of type F is Callback given a set of argument types T1, T2, ..., TN and a return type R, if one of the following conditions holds given rvalues t1, t2, ..., tN of types T1, T2, ..., TN, respectively:

* If F is not a pointer to member function type, the expression f(t1, t2, ..., tN) is well-formed and is convertible to R.

* If F is a pointer to member function type, the expression mem_fn(f)(t1, t2, ..., tN) is well-formed and is convertible to R."

10.6 Class template function supports only unary and binary member function pointers.

Submitter: Doug Gregor
Status: TR

c++std-ext-5560
In section 3.4.3, the definition of Callable supports object pointers and smart pointers when calling a target member function pointer, via the call expression "((*t1).*f)(t2, ..., tN)." This definition, and the use of mem_fun in 3.4.3.1, limit class template function<> to supporting member functions only when:

1) the function<> instantiation is unary or binary (mem_fun only supports supports unary and binary member function pointers).

2) the first function parameter of the function<> instantiation is a pointer (mem_fun requires its first parameter to be a pointer).

This formulation is overly restrictive. For instance, this formulation does not support the following usage that has been demonstrated to be useful in the Boost.Function library on which class template 'function' was modeled:

```
struct A {
  void f(int, float, double);
};

function<void(A&, int, float, double)> g;
g = &A::f; // ill-formed due to callable requires,
```

Resolution:

* In the definition of Callable in section 3.4.3, replace the expression "((*t1).*f)(t2, ..., tN)" with "mem_fn(f)(t1, t2, ..., tN)." [Note that this change propagates to the proposed resolution of issue #10.5 as well.]

* In 3.4.3.1, replace the instance of "mem_fun" with "mem_fn".

10.7 Implementations need not define the function<> conversion operator type.

Submitter: Doug Gregor
Status: TR

c++std-ext-5558

Section 3.4.3.3 has the following "boolean-like" conversion operator:

operator implementation-defined() const;

This requires that implementors document the type of this conversion operator. However, this type should not be documentation because it should not be relied upon by users. There is precedent for calling this type "*unspecified-bool-type*" (see 2.2.3.5 [tr.util.smartptr.shared.assign]).

Resolution:
Replace the implementation-defined operator declaration in 3.4.3.3 and 3.4.3 with:

operator unspecified-bool-type() const;

The fundamental problem is that we're saying "implementation defined" when we don't really mean it. (Note to editor: put in a cross reference so we know what it means to say unspecified-bool-type. We use it elsewhere.)

10.8 Class template function should have null pointer assignment/comparison operations.

Submitter: Doug Gregor
Status: TR

Class template function should provide assignment/initialization from and comparison against the null pointer constant to achieve greater source compatibility with function pointers. For instance, the following (currently ill-formed) syntax should be legal:

```
function<void(int)> f(0); // same as default construction: no target
if (f == 0) {} // evaluates true: f has no target
f = NULL; // removes f's target. f now has no target
if (f != NULL) {} // evaluates false: f has no target
```

When this new syntax is available, the `empty` and `clear` member functions become redundant and should be removed.

Historically, the assignment/initialization from the NULL pointer constant was not supported because the author had not found a suitable implementation. The comparison syntax, although not explicitly supported, has been available in all known implementations due to the formulation of the function<> conversion operator. Assignment/initialization are now known to be implementable without any unsafe “loopholes.”

Resolution:
Introduce a new constructor into 3.4.3.1, with the following description:

`function(`*`unspecified-null-pointer-type`*`);`

Postconditions: `(bool)(*this)`;
Throws: will not throw.

Introduce a new assignment operator into 3.4.3.1, with the following description:

```
function& operator=(unspecified-null-pointer-type);
```

Effects: If (bool)(*this), deallocates current target.
Postconditions: !(*this).

Add a new subsection to 3.4.3 titled "null pointer comparison operators" containing the following:

```
template<typename R, typename T1, typename T2, …, typename TN,
         typename Allocator>
  bool operator==(const function<R(T1, T2, …, TN), Allocator>& f,
                  unspecified-null-pointer-type);
template<typename R, typename T1, typename T2, …, typename TN,
         typename Allocator>
  bool operator==(unspecified-null-pointer-type,
                  const function<R(T1, T2, …, TN), Allocator>& f);
```

Returns: `!f`
Throws: will not throw.

```
template<typename R, typename T1, typename T2, …, typename TN,
         typename Allocator>
  bool operator!=(const function<R(T1, T2, …, TN), Allocator>& f,
                  unspecified-null-pointer-type);
template<typename R, typename T1, typename T2, …, typename TN,
         typename Allocator>
  bool operator!=(unspecified-null-pointer-type,
                  const function<R(T1, T2, …, TN), Allocator>& f);
```

Returns: `(bool)f`
Throws: will not throw.

Introduce the above new declarations into the summary in section 3.4.3.

Remove the definitions of the `empty` and `clear` member functions from section 3.4.3.3 and 3.4.3.2, respectively, and from the summary in section 3.4.3.

Replace the string "`this->empty()`" with "`(bool)(*this)`" throughout section 3.4.3.

Replace the **Returns** clause for the conversion operator in 3.4.3.3 with:

Returns: if *this has a target, returns a value that will evaluate true in a boolean context; otherwise, returns a value that will evaluate false in a boolean context. The value type returned shall not be convertible to int.

Notes from Sydney:
The functionality is useful, but putting in an *unspecificed-null-pointer-type* isn't a great idea. We need better specification. Other than the specification issue, everyone agrees that this is a good idea. We will try to fix it in Redmond.

Notes from Redmond:

We couldn't think of a better solution than *unspecificed-null-pointer-type* so that's what we're doing. Note to editor: also remove the spurious `Allocator` template parameter.

10.9 result_of template type parameter unrelated to description

Submitter: Doug Gregor
Status: TR

Section 3.1.2 should relate the template parameter "FunctionCallTypes" to the types F, T1, T2, ..., TN used in the description.

Resolution:
Introduce a comment in the class definition noting the function type F(T1, T2, …, TN):

```
template<typename FunctionCallTypes> // F(T1, T2, ..., TN)
class result_of {
public :
  // types
  typedef unspecified type;
};
```

10.10 result_of should be based on rvalues, not lvalues

Submitter: Doug Gregor
Status: TR

c++std-lib-12752
In the first paragraph of section 3.1.2, the use of the word "lvalue" limits result_of's usefulness.

Resolution:
Replace each instance of "lvalue" with "rvalue", and add the phrase "reference types Ti" are treated as lvalues" to the first paragraph of section 3.1.2. The new paragraph should be:

"Given an rvalue `f` of type F and values t1, t2, ..., tN of types T1, `T2, ..., TN`, respectively, the type member type defines the result type of the expression `f(t1, t2, ...,tN)`. The values `ti` are lvalues when the corresponding type `Ti` is a reference type, and rvalues otherwise."

This may require a change in 2.1.2.3, if the resolution to 10.4 is accepted.

In section 3.3.4, replace instances of `result_of<R(T)>::type` with `result_of<R(T&)>::type` and instances of result_of<R(T1, T2, ..., T*n*)>::type with result_of<R(T1&, T2&, ..., T*n*&)>::type.

Notes from Sydney:
This looks like a good idea but it's too closely related to 10.4, which we don't think we understand well enough.

10.11 result_of can not work for function and member function types

Submitter: Doug Gregor

Status: TR

In section 3.1.2, bullet #1 starts with "If F is a function type...". F can be a function pointer or function reference, but it cannot be a function type because it is encoded as the return type of a function.

In section 3.1.2, bullet #2 starts with "If F is a member function type". F cannot be a member function type.

Resolution:
Replace the first phrase in section 3.1.2, bullet #1 with "If F is a function pointer or function reference type".

Replace the first phrase of section 3.1.2, bullet #2 with "If F is a member function pointer type".

10.12 should result_of support cv-qualified class types?

Submitter: Doug Gregor
Status: TR

In section 3.1.2, bullets #4 and #5 start with "If F is a class type...", which excludes cv-qualified class types. However, cv-qualified class types are explicitly mentioned in the rationale (see N1454).

Resolution:
Replace the phrase "If F is a class type" with "If F is a possibly cv-qualified class type" in section 3.1.2, bullets #4 and #5.

10.13 Bad_function_call should inherit from std::exception, not std::runtime_error

Submitter: Howard Hinnant
Status: TR

The exception class bad_function_call currently derives from runtime_error, but has no constructor taking a client-defined string. Deriving from runtime_error is much more expensive than deriving from exception because runtime_error must support a general client-defined string whereas exception does not. Therefore I recommend that bad_function_call derive from exception (like bad_alloc, bad_cast, bad_typeid, etc.).

10.14 Function call limits

Submitter: Pete Becker
Status: TR

We've got several templates that deal with functions that take varying numbers of arguments:

- template class 'result_of' takes a function type argument with up to N arguments

- template function 'mem_fn' supports member functions with up to n arguments (value is implementation defined)
- template function 'bind' supports functions with up to some implementation defined number of arguments
- template class 'function' supports functions with up to Nmax arguments (value is implementation defined)
- template class 'reference_wrapper' has a function call operator that supports up to N arguments

A typical implementation will share code among these templates in various ways, with the result being that the maximum number of arguments will be the same for most or all of them. It would be easier for users if we made this a requirement: that all of these templates support the same number of function arguments, so there's only one value to specify. Is there an advantage to keeping them separate?

In the same vein, we have several different styles for describing these argument lists; we really need to describe them in the same way. One way to do this would be to put the descriptive text into a subclause with a title that's something like "function call types" and refer to that text from the appropriate places.

Proposed resolution:
Addressed by N1673=04-0113, “Unifying TR1 Function Object Type Specifications,” by Pete Becker and Peter Dimov.

10.15 Missing return types for functions in 2.1.1

Submitter: Dietmar Kuehl
Status: Duplicate

The synopsis for the functions in 2.1.1 [tr.util.refwrp.synopsis]
lacks the return types.

Proposed Resolution:

Add the missing return types, ie. replace

```
template <typename T> ref(T&);
template <typename T> cref(const T&);
```

by

```
template <typename T> reference_wrapper<T>       ref(T&);
template <typename T> reference_wrapper<const T> cref(const T&);
```

10.16 Ref() should be overloaded for const types

Submitter: Dietmar Kuehl

Status: NAD

As the TR already allows binding of reference wrappers to constantobjects and supports this with the 'cref()' function, there seemsto be no reason why 'ref()' should not provide a uniform approach to creating a reference wrapper, even if the argument is itself aconstant. That is, it seems that there should be an additionaloverload for the 'ref()' function.

It is unclear whether this is an accidental omission or if theommision of the overload taking a 'T const&' is deliberate. If it is deliberate, I would be interested in the reason.

Proposed Resolution:

Add the signature
template <typename T> reference_wrapper<const T> ref(const T&);

to 2.1.1 [tr.util.refwrp.synopsis]. Also add the corresponding description to 2.1.2.4 [tr.util.refwrp.helpers], ie. add

template <typename T> reference_wrapper const T> ref(const T&);

Returns: reference_wrapper<const T>(t)
Throws: Does not throw

10.17 Unclear description of multi-argument function

Submitter: Dietmar Kuehl
Status: Closed

The class definition in 2.1.2 [tr.util.refwrp.refwrp] mentions a member function template with "N" parameters. As stated, the semantics is unclear to me: is the intention that there is *one* function with N parameters for a fixed N? This would be pretty useless because the user of this function would have to specify this unknown number of parameters. I guess, the notation is intended to mean that the function call operator is overloaded for 1, 2, ..., N parameters. This should, IMO, be stated explicitly.

Also, the function uses 'result_of<...>::type' rather than the already declared 'result_type'. Even worse the 'result_of<...>' class template is nowhere mentioned again.

Discussion from Sydney:
Two questions. First: does this notation imply zero or more arguments? Second: do we mean to include variadic functions? Leave open. We don't have a solution yet.

Rationale:
Addressed by N1673=04-0113, “Unifying TR1 Function Object Type Specifications,” by Pete Becker and Peter Dimov.

10.18 Missing return types in 2.1.2

Submitter: Dietmar Kuehl
Status: TR

In the body of the 'reference_wrapper' class template various return types are missing. Also, the type 'result_type' is not at
mentioned in the body. The latter may be intentional, however.

The missing return types were discussed on the mailing list. Although this was quite a while ago I don't see the correction being reflected. This should be a trivial change:

Proposed Resolution:
In the class synopsis for `reference_wrapper` in 2.1.2 [tr.util.refwrp.refwrp]:

- Add the declaration:
  ```
  typedef ... result_type; // not always defined (see below)
  ```
- Add result types for the member functions in the "access" section, changing that section to:
  ```
  // access
  operator T&() const;
  T& get() const;
  ```

10.19 Underspecification of reference wrapper assignment

Submitter: Dietmar Kuehl
Status: TR

The sentence right below the class template in 2.1.2 [tr.util.refwrp.refwrp] states that 'reference_wrapper<T>' is an Assignable wrapper around a reference. This seems to imply that the class template stores a reference to 'T'. This interpretation is confirmed in later sections explicitly talking of "the stored reference". Unfortunately, there are no semantics given for the assignment operator: the generated assignment operator is not applicable if the class stores a reference.

My personal guess is that the intended semantics with respect to the assignment operator are those implemented by 'boost::reference_wrapper<T>': this class does not store a reference but a pointer to the object. Assigning to a 'reference_wrapper<T>' replaces the internally stored pointer such that the wrapper actually references a different object.

The tricky issue is that the address operator (unary &) can be overloaded. The Boost implementation avoids this problem by using 'boost::addressof()' which does some ugly casts to obtain the address to the actual object. If this seems to be unfit for a standard library, there are a few possible resolutions I can think of:

- Either require the type 'T' to have a "reasonable" address operator, ie. a non-overloaded one or an overloaded one which still does the "Right Thing".
- The "Assignable" guarantee has to be dropped. This course of action may actually be a suitable alternative: at least with my limited fantasy I could not come up with a simple example of using the reference wrapper while also needing assignment.

- The semantics of the assignment operator can be spelled out explicitly, avoiding reliance on the generated semantics. Possible implementations include the 'addressof()' hack from Boost or use of explicit destruction and placement new for a nested member really storing a reference.

From a discussion on the mailing list I gather that the pointer representation seems to be the intent.

Discussion from Sydney:
We want to store a pointer, not a reference. (Or rather, we want to describe the semantics in ways that allow a pointer-based implementation; we don't want to prescribe it.) We need wording to that effect. Leave this issue open for Redmond.

Resolution:
In the reference_wrapper summary 2.1.2 [tr.util.refwrp.refwrp], replace:

```
// construct/copy/destruct
explicit reference_wrapper(T&);
```

With the following:

```
// constructors
explicit reference_wrapper(T&);
reference_wrapper(const reference_wrapper<T>& x);

// assignment
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

In 2.1.2.1 [tr.util.refwrp.const], after paragraph 2, add:

```
reference_wrapper(const reference_wrapper<T>& x);
```

Postconditions: *this references the object that x references.
Throws: Does not throw.

Add a new section 2.1.2.2 [tr.util.refwrp.assign] after section 2.1.2.1 [tr.util.refwrp.const], containing:

```
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

Postconditions: *this references the object that x references.
Throws: Does not throw.

10.20 Missing return types in 2.1.2.2

Submitter: Dietmar Kuehl
Status: TR

The signatures in this paragraph again lack the return type. This should also be a trivial correction.

Proposed resolution:

In 2.1.2.2 [tr.util.refwrp.access] replace the [broken] signature

```
  operator () const;
```
by
```
  operator T& () const;
```

and the [broken] signature

```
  get() const;
```
by
```
  T& get() const;
```

10.21 Garbled description of reference wrapper invocation

Submitter: Dietmar Kuehl
Status: Closed

This whole paragraph is entirely messed up! Here are the things I have noted:
- There is an object "f" used which is mentioned nowhere else.
- There is class tempalte "result_of<...>" used which is nowhere defined.
- Bullet 2. in 2.1.2 [tr.util.refwrp.refwrp] seems to indicate that the reference wrapper should be applicable to member function pointers.
  The notation given in 2.1.2.3 [tr.util.refwrp.invoke] is not suitable for calling member functions.
- As given, this stuff seems not to apply to functions without arugments.

I think this whole paragraph needs new wording.

There is another IMO major issue with the stuff which makes the whole approach to function forwarding somewhat questionable: the member function templates forwarding the function call to the referenced object take their parameters by value. This may yield surprising results if the called function actually takes arguments by reference:
- A function taking a reference may suddenly appear to allow calling it using a non-lvalue, for example:

```
  #include <utility>
  void f(int&);
  int main() {
    f(10);                // illegal
    std::tr1::cref(&f)(10); // legal!
  }
```

- The function arguments are copied altough the called function might accept reference and const reference arguments. Of course, this problem can be worked around using the reference wrapper...

For function pointers or member function pointers the forward problem can be solved relatively simple by accepting exactly the parameters of the given function. I don't have a solution for this problem if functors are involved because in this case the function call operator may be

overloaded or may even be a member function template. Of course, if the function call operator is overloaded, the use of a single result type may also be questionable.

I haven't provided revised wording because it is unclear to me how this should look like: the wording depends on how, if at all, the forwarding issue is resolved.

Notes from Sydney:
Part of this is a duplicate of 10.4. Part of it is related to 10.5, but it doesn't appear to be a duplicate. Leave this open for now; we can't do much with this until we get new wording.

Resolution:
Covered by N1673.

10.22 reference_wrapper invocation underspecified?

Submitter: Pete Becker
Status: Closed

2.1.2.3 [tr.util.refwrp.invoke] /1 says that the effect of operator() is f.get()(a1, a2, ..., aN). The "f." should be removed (also from /2), because it's not defined and not needed. More important, when a reference_wrapper holds a pointer to member function this code isn't valid. Looks like we need to say that for a member function we do mem_fn(get())(a1, a2, ..., aN). (Since we provide result_type for pointers to member functions, I assume the intention was to also support calling them. <g>)

Also, while we're at it, mem_fn gives us pointers to data members for free. Well, almost: if we're supporting them we need to add them to result_type.

Proposed resolution:
Covered by N1673.

10.23 Mem_fn wording not quite right

Submitter: Pete Becker
Status: Closed

The first sentence of 3.2.2 [tr.func.memfn.memfn] /2 says:

> mem_fn(&X::m), where m is a data member of X, returns an object through which a reference to &X::m can be obtained given a pointer, a smart pointer, an iterator, or a reference to X.

This is parallel to the preceding paragraph, which says that a call to &X::f can be made. However, the parallel isn't right: what you get from mem_fn(x) is a reference to the contained data object, not to &X::m.

Proposed resolution:
Covered by N1673.

10.24 Mem_fn result_type for pointer to data member

Submitter: Pete Becker

Status: TR

When mem_fn is applied to a pointer to data member &X::m, the object it returns “shall have a nested typedef result_type defined as either M or M const&, where M is the type of m.” It doesn’t give users any guidance about which of those types to expect. We should either pick one, even if the choice is arbitrary (neither choice is right in all circumstances), or else remove this nested typedef entirely.

Proposed resolution:
Remove result_type for pointer-to-data-member.

10.25 Reference_wrapper needs better standardese

Submitter: Pete Becker
Status: Duplicate

2.1.2 [tr.util.refwrp.refwrp] /1 says:
reference_wrapper<T> is a CopyConstructible and Assignable wrapper around a reference to an object of type T.

2.1.2.1 [tr.util.refwrp.const] /1 says:
Effects: Constructs a reference_wrapper object that stores a reference to t.

2.1.2.2 [tr.util.refwrp.access] /1 and /3 both say:

Returns: The stored reference.
First problem (2.1.2): we need to say "reference_wrapper<T> shall be CopyConstructible and Assignable" because that's a requirement. The rest of the sentence is descriptive, and ought to be in a separate sentence.

Second problem: we talk about the "reference" even though we know that we don't expect implementations to actually store a reference. That is, we're using "reference" in an informal sense but haven't said so. (Yes, the as-if rule says that it doesn't matter, since whether there's actually a stored reference isn't detectable, but we shouldn't have to invoke the as-if rule to say that we didn't really mean what we said). I think the way to resolve this is to add an exposition-only private data member T *ptr and to describe the ctor, operator T&, and get() in terms of their use of that data member.

(Further discussion on c++std-lib: the current wording allows for types with overloaded operator&. We should consider this for any alternate solution we come up with.)

Proposed resolution:
Duplicate of 10.19

10.26 Wrong arguments to unary_function and binary_function

Submitter: Peter Dimov
Status: TR

3.4.3 [tr.func.wrap.func] /3 says that tr1::function derives from std::unary_function<R, T1> when N == 1, and std::binary_function<R, T1, T2> when N == 2.

This is wrong, because the result type is the last argument to unary_function and binary_function.

Proposed resolution:
Put the result type argument in the right place.

10.27 Reference_wrapper should be in <functional>

Submitter: Pete Becker
Status: TR

We've got a loopy dependency among headers now:

1. <utility> provides reference_wrapper, which uses result_of
2. <functional> provides result_of, as well as bind and function,
   each of which uses reference_wrapper.

From an implementor's perspective that just means shoving some things down into a common header used by both <utility> and <functional>, but from a user's perspective, would it be easier to move reference_wrapper to <functional>?

Further discussion (c++std-lib-14107, and references therein): this issue has come up before, and it's generally agreed that putting reference_wrapper in <functional> is a good idea. That's what some people have already implemented.

Resolution:
Move reference_wrapper to <functional>.

Rationale:
It's clear that reference_wrapper and result_of should be in the same header; the only question was whether that header should be <utility> or <functional>. The LWG decided that <functional> made the most sense.

10.28 Incorrect wording for bind

Submitter: Pete Becker
Status: Closed

For template<class F> unspecified bind(F f), 3.3.4 [tr.func.bind.bind] /7 says:

> Requires: F must be CopyConstructible. lambda(f)() must be a valid expression. If f is not a simple function object, the behavior is implementation defined.

The effect of lambda(f) is to dereference a reference_wrapper and to simply pass any other type through. A "simple function object" is a pointer to function or an object of a type with a nested type "return_type."

Shouldn't the last sentence refer to lambda(f) rather than f?

Proposed resolution:
Covered by n1673.

10.29 reference_wrapper<reference_wrapper<T> >

Submitter: Pete Becker
Status: TR

What should happen if I call ref or cref with an object of type reference_wrapper<T>? According to the words we have now, I should get a reference_wrapper<reference_wrapper<T> >, which means I have to call get() twice to get the actual object. That doesn't act very much like a reference, and I assume it isn't really what we want. Seems to me that could fix this by adding:

```
template<class T>
reference_wrapper<T> ref(reference_wrapper<T> t)
{
  return ref(t.get());
}
```

and the obvious analog for cref. We probably should also say that a reference_wrapper<reference_wrapper<T> > is ill-formed, so that we don't have to deal with double dereferences anywhere.

Resolution:
Add

```
template<class T>
reference_wrapper<T> ref(reference_wrapper<T> t)
{
  return ref(t.get());
}
```

and the obvious analog for cref.

Rationale:
The LWG discussed whether it was more or less useful, and more or less uniform, to have ref(ref(x)) collapse to ref(x). The argument against: reference_wrapper<T> is treated differently from other types. The argument for: ref(x) looks more like adding a built-in C++ reference. The LWG chose not to make reference_wrapper<reference_wrapper<T> > ill-formed, because it was believed that there were some (probably rare) use cases for it.

10.30 function comparison operators unreliable

Submitter: Peter Dimov
Status: TR

tr1::function declares comparison operators of the form

template < typename Function1 , typename Function2 >
  void operator ==(const function < Function1 >&, const function < Function2 >&);

to prevent unwanted comparisons caused from the conversion to unspecified-bool-type from taking place. However these operators do not fail reliably at compile time (although they do fail at link time). In addition, an error that is caused by misspelling
  f1 = f2;
as
  f1 == f2;
now compiles cleanly (without a warning) due to the void return type of operator==.

It is better to replace the free functions with private member templates to catch these mistakes at compile time.

Proposed resolution

Remove the declarations of operator== and operator!= from [tr.func.wrap.func]/3.

Add
```
private: // undefined operators
   template<class Function2> void operator==(function<Function2> const&)
const;
   template<class Function2> void operator!=(function<Function2> const&)
const;
```
after
```
   R operator()(T1, T2, ..., TN) const;
```

in [tr.func.wrap.func]/3.

Replace the declarations in [tr.func.wrap.func.undef] as above.

10.31 [tr.func.wrap.func]/2 says rvalues, should be lvalues

Submitter: Peter Dimov
Status: Closed

[tr.func.wrap.func]/2 (3.4.3/2 in N1660) defines t1, ..., tN as rvalues of types T1, ..., TN.

However in
```
   function::operator()(T1 t1, ..., TN tN) const;
```
t1, ..., tN are lvalues of types T1, ..., TN. So the Callable requirement does not reflect the actual semantics of tr1::function.

Proposed resolution
replace "rvalues" with "lvalues" in [tr.func.wrap.func]/2.

Rationale:
Partly NAD, partly a dup of things already done in n1673.

10.32 function(reference_wrapper<F> f) targets f.get()

Submitter: Peter Dimov
Status: TR

[tr.func.wrap.func.con]/10 (3.4.3.1/10 in N1660) says that *this targets f.get(). This seems to imply that f.get() is copied and stored inside *this. This is not the intended semantics of function<> in this case; it should store f itself.

Proposed resolution
Remove [tr.func.wrap.func.con]/9-12. Remove the constructor taking reference_wrapper<F> from [tr.func.wrap.func]/3. Add "or a reference_wrapper<T> for some T" to the end of the first sentence of [tr.func.wrap.func.con]/8.

Rationale
We no longer need explicit wording for reference_wrapper<T> because it is a forwarding function object.

10.33 is reference_wrapper supposed to "call" member pointers?

Submitter: Peter Dimov
Status: Closed

The specification of reference_wrapper is inconsistent; it defines result_type when its argument is a pointer to a member function, but its operator() specification does not "call" member functions the way function, mem_fn and bind do.

Using a reference_wrapper around a pointer to member isn't common, because there are lifetime issues involved, which mem_fn avoids.

We need to decide one way or the other, and either remove result_type for member functions, or change operator() appropriately.

Proposed resolution:
Remove the second bullet of [tr.util.refwrp.refwrp]/2.

Replace "a function pointer" with "a function type or a function pointer type" in [tr.util.refwrp.refwrp]/1.

This is consistent with the proposed resolution of reference_wrapper<void()>.

Rationale:
N1673 makes this consistent (although not in the direction originally recommended).

10.34 is reference_wrapper<void()> supposed to be legal?

Submitter: Peter Dimov
Status: TR

Consider this code:

```
void f();

int main()
{
   ref(f);     // 1
   ref(f)();   // 2
}
```

Are the lines marked (1) and (2) supposed to be legal? ref(f) returns a `reference_wrapper<void()>` and the declaration of the zero-argument forwarding `operator()` tries to instantiate `result_of<void()()>::type`, which is not legal. Even worse, (2) actually tries to call this operator().

Proposed resolution:
In the proposed resolution in N1673, change the first bullet in the definition of *simple call wrapper* from:

- if T is a pointer to function, result_type shall be a synonym for the return type of T;

to the following:

- if T is a function, reference to function, or pointer to function type, result_type shall be a synonym for the return type of T;

Rationale:
This is a real bug. The refwrap in question will be callable but won't have a result type, which seems wrong. The issue is that we're passing in a function type, not a function *pointer* type. We need to change the def of weak result in 1673 so that it includes function types and function reference types.

10.35 result_of and Standard Library Function Objects

Submitter: Pete Becker
Status: TR

[tr.func.ret]/3 says:

> If the implementation cannot determine the type of the expression f(t1, t2, ..., tN), or if the expression is ill-formed, the implementation shall use the following process to determine the member type type:
> ....
> 3. If F is a function object defined by the standard library, the method of determining type is unspecified;
> ....

Since this paragraph gives the implementation license to produce incorrect results, provided the incorrect results follow the formula set out in this paragraph, it seems to say that an implementation that always determines that type is void (for function objects defined by the standard library and the TR) conforms to this requirement. There are no constraints on the result.

I assume that's not what was really intended. What is this paragraph trying to say? How can we say it better?

Resolution:
Change the beginning of the first sentence of [tr.func.ret]/3 from:

> If the implementation cannot determine the type of the expression f(t1, t2, ..., tN), or if the expression is ill-formed, the implementation shall use the following process to determine the member type type:

to:

> 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 else if the expression is ill-formed, the implementation shall use the following process to determine the member type type:

Remove bullet item 3 from [tr.func.ret]/3.

Rationale:
What we really want to say is that result_of is always required to get the right answer for stdlib types. Pete and Peter have proposed different mechanisms, each of which can be shown to get the right answer all the time. We don't need to specify which one an implementation uses, we just need to require it to do *something* that gets the right answer.

10.36 TR1 does not provide the equivalent of not1 and not2

Submitter: Scott Meyers via comp.std.c++
Status: NAD

Original posting is at the URL
http://groups.google.com/groups?selm=MPG.1babce479187c9ca98978d%40news.hevanet.com

When I posted asking about a not1/not2 replacement in TR1, I was just worried about syntax, but now I think there is a functional problem, too. TR1's mem_fn is nicely flexible:

mem_fn(&X::f), where f is a member function of X, returns an object through which &X::f can be called given a pointer, a smart pointer, an iterator, or a reference to X...

The return type of mem_fn is unspecified, but I'm imagining it's a class with a templatized operator(). That's cool until I try to pass such an object to not1/not2, as these classes are nowhere near as flexible: they want to grab a typedef identifying the parameter type(s) taken by operator() in the object they are wrapping. There's no such typedef for the objects returned by mem_fn. The result, I imagine, is that it's possible to use mem_fn with a container of smart pointers and a predicate, but it's not possible to do exactly the same thing with a negated predicate.

My test program, using boost as a tr1 proxy, yields the behavior I expect:

```
#include "boost/mem_fn.hpp"
#include "boost/smart_ptr.hpp"
#include <vector>
#include <algorithm>

class Widget {
public:
```

```
  bool isOK() const;
};


int main()
{
  using namespace std;
  namespace tr1 = boost;

  vector<tr1::shared_ptr<Widget> > v;

  find_if(v.begin(), v.end(),
          tr1::mem_fn(&Widget::isOK));        // compiles

  find_if(v.begin(), v.end(),
          not1(tr1::mem_fn(&Widget::isOK)));  // doesn't compile
}
```

This strikes me as a problem. Have I overlooked something, or should TR1 include some kind of not1/not2 functionality that's as flexible as mem_fn?

Proposed resolution:
From Peter Dimov, c++std-lib-14362:

Options:

1. Introduce an operator! operating on function objects returned by tr1::bind such that !B returns bind(logical_not<bool>(), B).

2. Introduce TR1 replacements of std::not1 and not2 such that tr1::not1(f) returns std::not1(f) when f has a nested type named "argument_type", and a new TR1 unary negator otherwise. Similarly, tr1::not2(f) should return std::not2(f) when f has nested types first_argument_type and second_argument_type, a new TR1 binary negator otherwise.

I want to stress that I am proposing that both options need to be pursued and considered independently; it's not an either/or issue.

The wording for option 1 would be:

(new paragraph somewhere in tr.func.bind.bind):
The expression !b, where b has been returned by bind shall yield bind(std::logical_not<bool>(), b).

Option 2 requires comparatively major additions:

New subcategory in 3. Function objects, 3.6 Negators [tr1.func.negators]:

The negators not1 and not2 are enhanced, backward compatible versions of std::not1 and std::not2.

```
template<class Predicate> class unary_negate2
{
public:
   typedef bool result_type;
   explicit unary_negate2(const Predicate& pred);
   template<class X> bool operator()(X const & x) const;
};
```

operator() returns !pred(x).

```
template<class Predicate> /* see below */ not1(Predicate pred);
```

Returns: std::unary_negate<Predicate>(pred) when Predicate::argument_type exists and is a type, unary_negate2<Predicate>(pred) otherwise.

```
template<class Predicate> class binary_negate2
{
public:
   typedef bool result_type;
   explicit binary_negate2(const Predicate& pred);
   template<class X, class Y>
   bool operator()(const X& x, const Y& y) const;
};
```

operator() returns !pred(x, y).

```
template<class Predicate> /* see below */ not2(Predicate pred);
```

Returns: std::binary_negate<Predicate>(pred) when Predicate::first_argument_type and Predicate::second_argument_type exist and are types, binary_negate2<Predicate>(pred) otherwise.

Rationale for option 2:

This specification allows us to concisely express that:

1. (!b)(args) returns !b(args);
2. !bind(...) has a nested result_type bool;
3. !bind(...) can appear as a nested subexpression in an outer bind;
4. is_bind_expression<>::value is true for the type of !b.

I believe that it is not overconstraining, because there is no way dor the user to inspect the actual type of the returned function object. A conforming implementation can (and in some cases, will) return

bind<bool>(__logical_not, b)

for example (to avoid compiler warnings, or for some other reason).

This new capability allows us to negate the result of invoking &X::f by using !bind(&X::f, _1) or !bind(&X::f, _1, _2), respectively. It is also of general utility.

Rationale for option 2:
These new additions present a relatively large "attack surface" for later issues and defects. However, I believe that the advantages may be worth it. This is an unique opportunity for implementers to test a change to standard library components that is 99.9% backward compatible; that is, tr1::not1 and tr1::not2 are intended to completely replace std::not1 and std::not2, not merely complement them.

Rationale for closing this as NAD:
10.37 solves most of the problem, and the remainder can be solved by combining bind with `logical_not`. We don't need to replicate the old adaptor interface.

10.37 TR1 Function Objects and Backward Compatibility

Submitter: Pete Becker
Status: TR

This is related to 10.36 but not quite identical. See c++std-lib-14247 and responses.

The underlying problem is that TR1 is a bit cavalier about interoperability with the function objects in the current standard. In general, we ought to support the type propogation mechanism that we established in the C++ standard.

For TR1 that means that result_type should be defined to give the exact type of a function call when that's possible, and not defined otherwise; argument_type should be defined for call wrappers that take a single argument whose type is determined by the type of the target object (that is, the function call operator is not a template); and first_argument_type and second_argument_type should be defined for call wrappers that take two arguments whose types are determined by the type of the target object.

The details:

1. tr1::function is derived from unary_function or binary_function when appropriate

2. tr1::mem_fn's return type can be derived from unary_function or binary_function when called with a pointer to member function that takes 0 or 1 arguments

3. tr1::reference_wrapper<T> can be derived from unary_function or binary_function, as appropriate, when T is a pointer to function, a pointer to member function, or a class type that is derived from unary_function or binary_function

4. tr1::bind returns a type with a template operator(), so can't provide this interface. It can, and does, provide result_type

1 and 4 are currently in TR1. We should also require 2 and 3.

Resolution:
In the revised version of [tr.func.memfn], add the following after the "Returns:" paragraph:

The simple call wrapper shall be derived from `std::unary_function<cv T*, R>` when pm is a pointer to member function with cv-qualifier cv and taking no arguments.

The simple call wrapper shall be derived from `std::binary_function<cv T*, T1, R>` when pm is a pointer to member function with cv-qualifier cv and taking one argument of type T1.

Change the first line of the declaration of reference_wrapper in [tr.util.refwrp.refwrp] from:

```
template <class T> class reference_wrapper {
```

to:

```
template <class T> class reference_wrapper
        : public unary_function<T1, R>          // see below
        : public binary_function<T1, T2, R>     // see below
        {
```

Add the following text immediately before [tr.util.refwrp.const]:

The template instantiation reference_wrapper<T> shall be derived from `unary_function<T1, R>` only if the type T is any of the following:

- a function type or a pointer to function type taking one argument of type T1 and returning R
- a pointer to member function type with cv-qualifier cv and no arguments; the type T1 is cv T* and R is the return type of the pointer to member function
- a class type that is derived from `unary_function<T1, R>`

The template instantiation reference_wrapper<T> shall be derived from `binary_function<T1, T2, R>` only if the type T is any of the following:

- a function type or a pointer to function type taking two arguments of types T1 and T2 and returning R
- a pointer to member function with cv-qualifier cv and taking one argument of type T2; the type T1 is cv T* and R is the return type of the pointer to member function
- a class type that is derived from `binary_function<T1, T2, R>`

10.38 Accessing the target of a TR1 function object

Submitter: Doug Gregor
Status: TR

The tr1::function class template doesn't provide any mechanism for accessing the stored function object or determining its type. This functionality is sometimes useful. See N1667, from the midterm mailing, for more details.

Proposed resolution:
Accept the proposed resolution from N1667, but renaming type to target_type, and returning the type_info by const reference instead of by value.

11 Tuple issues

11.1 Implementation limits: nonexistent Annex B

Submitter: Alisdair Meredith
Status: Editorial

In the description of tuple (TR 6.1) it refers to Annex B for the recommended minimum no. of elements.

As yet the TR has no annexes, and other libraries specify recommendations directly in their descriptions.

Tuple should either make its own recommendation (10?) or we should spawn an annex and put all such recommendations in one place (as per origanal standard)

Resolution:
We should create an annex for implementation limits

11.2 Confusing extractor language

Submitter: Pete Becker
Status: Closed

tr.tuple.io/7 says:

Notes: It is not guaranteed that a tuple written to a stream can be extracted back to a tuple of the same type.

The phrasing makes this sound like a non-normative note, but it's actually written as a normative text. Either way, I'm not clear on what it means. It sounds like it allows an implementation to always fail on an attempt to read the value of a tuple from an input stream.

Rationale:
Discussion revealed that this was more complicated than was realized at first. There are lots of types, including strings, that do not have a guarantee of symmetry between input and output. Figuring out exactly what insertion and extraction interface is most useful is tricky. But given the resolution to 10.3, this is now irrelevant.

11.3 Tuple formatting interface

Submitter: Matt Austern, Pete Becker, Howard Hinnant
Status: TR

The textual representation of a tuple is $Lt_0dt_1d...dt_nR$, where L is the opening, R is the closing, and d is the delimiter between two elements d_i and d_{i+1}. Currently L, R, and d are all single characters. They are set by three manipulators `tuple_open`, `tuple_close`, and `tuple_delimiter`, defined in [tr.tuple.form].

There are several problems with this.

- The defaults for the delimiter characters are not specified. The introductory text in the original paper says that they are left parenthesis, right parenthesis, and space, but the proposed text in the paper doesn't say this, and the text in TR1 doesn't say it.
- Single characters are insufficiently general. For example, one might reasonably want to use the string `", "` as a delimiter.
- The model of *L, d*, and *R* may be insufficiently general even if they are permitted to be general strings. For example, one might want to represent a tuple as XML:
  `<tuple><elem>1</elem> <elem>2</elem> <elem>3</elem></tuple>`
- The note in [tr.tuple.form] paragraph 4 is confusing and unnecessary. The first sentence describes an implementation-based reason for the single-character restriction, and the rest describes an alternative implementation technique. Regardless of which interface we choose, this sort of discussion doesn't belong here.

Informally, some options:

- Remove the manipulators, and just say that the textual representation of a tuple is $(t_0,t_1,\ldots,t_n)$. There's precedent for that: we don't have a mechanism for controlling `std::complex` formatting.
- Keep the existing interface, but specify the defaults and remove the confusing note.
- Keep the existing interface and generalize it slightly: from single characters to strings.
- Generalize the interface, perhaps using a local facet.

Resolution:
Remove tuple I/O.

Rationale:
The LWG would like to consider the problem of container I/O (including tuple, pair, array, vector, complex, etc.) for TR2 or C++0x. We're deferring the problem of container I/O for now because there isn't any time, not because we think container I/O is unimportant.

11.4 "ignore" unspecified

Submitter: Pete Becker
Status: TR

tr.tuple.helper/7 has:

> [Example: tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:
> `int i; std :: string s;`
> `tie(i , ignore , s ) = make_tuple (42 , 3.14 , "C++");`
> —end example]

The first sentence is more properly a non-normative note. ignore isn't mentioned anywhere else, so its use in an example (which is not normative) is problematic. Is it intended to be part of TR1? If so, it should be specified somewhere. If not, we shouldn't use it in the example.

Proposed resolution:
Add to [tr.tuple.synopsis] the following declaration:

```
const unspecified ignore;
```

Change [tr.tuple.helper]/6 from

> **Returns**: `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tn)`

to:

> **Returns**: `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tn).` When an argument ti is *ignore* assigning any value to the corresponding tuple element has no effect.

11.5 Tuple inequality tests badly specified

Submitter: Pete Becker
Status: TR

c++std-lib—13908, c++std-lib—13922
Both tr.tuple.lt/2 and tr.tuple.le/2 are missing a ')', which makes the expressions unparseable. Adding a suitable ‘)’, what we get for less-than is equivalent to: `t < u iff t0 < u0 || !(u0 < t0) && t-tail < u-tail`.

Seems to me that the logic is getting lost in the abstraction. I'd prefer to see all six of the relational operators expressed in terms of two fundamental logical operations: equality and lexically less-than. Each operator can then specify short circuit evaluation of the individual terms, without having to repeat the definitions, with variations, of the underlying comparisons. (We've already got a definition of lexicograpical comparison in the standard [lib.alg.lex.comparison/3]; that may be a suitable starting point)

The reasons for this are consistency with the rest of the TR and with the standard, and simplicity. In general we've defined == and != for containers in terms of == on individual elements, and we've defined <, <=, >, and >= in terms of < on individual elements. One big benefit from doing it that way is that we don't need to say anything more to guarantee that t < u is equivalent to u >= t; with the tuple versions of these operators we have to say a great deal more. Further, users who want to write classes that can be used in tuples in general only have to define two operators instead of all six.

Resolution:
In sections 6.1.2.5, 6.1.2.6, and 6.1.2.7, rewrite the descriptions of operator!=, operator>, operator<=, and operator>= in terms of operator== and operator<.

Rationale:
The original rationale for defining (say) tuple operator> in terms of the underlying types’ operator>, instead of in terms of their operator<, was so that clients could use operator> even for types that have no operator<. The LWG didn’t find that argument compelling, since all of the other standard and TR1 containers define all comparisons in terms of operator== and operator<.

11.6 Tuple I/O field widths

Submitter: Pete Becker
Status: Closed

c++std-lib-14311

In addition to the issues covered in 11.3, there's a problem in the handling of the stream's field width specification. As currently specified, the following code:

```
tuple<int, int> ti = make_tuple(1, 2);
os << left << setw(10) << ti;
```

results in applying the field width to the opening delimiter for the tuple (or maybe to the first element, I'm not completely sure). In the example above, the result looks like this: "(1 2)". For std::complex we had the same problem, and fixed it by specifying that the output gets written as if the insertion had taken place into a new basic_ostringstream followed by inserting the result into os. The effect is that a field width applies to the entire representation of the complex value and not to any of its elements. I think we need to do the same thing for tuple.

Rationale:
No longer relevant since tuple I/O was removed from TR1.

11.7 Tuple interface to pair and array: which header?

Submitter: Pete Becker
Status: TR

c++std-lib-14416, c++std-lib-14417, c++std-lib-14517, c++std-lib-14519, c++std-lib-14520
Why aren't tuple_size<array<T, N> >, tuple_element<I, array<T,N> >, and the two forms of get for arrays in the <array> header rather than in <tuple>. Is there a good reason for them to be in <tuple>? The current location is a bit awkward for users, who have to include both array and tuple to use get<3>(my_array).

Similarly, [tr.tuple.pairs] provides for tuple_size, tuple_element, and the two get's to apply to instances of std::pair. However, these specializations aren't mentioned in the <tuple> synopsis, [tr.tuple.synopsis], and there is no other indication of which header defines them. They're like the arrary versions, so we ought to put them in the "same" place as the array versions, either in <tuple> or in <utility>.

This issue was discussed briefly in Sydney; Bill was slightly uncomfortable with putting those specializations in <array>. This discomfort was probably because doing that made the headers more complicated. However, that complication is minor, because the array versions are just specializations of a general template. All that's needed in <array> is a declaration of the general templates and the definitions of the specializations. <array> doesn't have to provide the full definition of tuple.

Resolution:
Put the tuple interface for array in the `<array>` header, and the tuple interface for pair in `<utility>`.

11.8 Tuple_size for `pair` overspecified?

Submitter: Alisdair Meredith
Status: New

`tuple_size` is defined as having -
Return type : integral constant expression

This same definition is used for the specialization provided for `tr1::array`.

However, for the `std::pair` specialization this is restricted to static const int. This means that if vendors want to use the same value type for all `tuple_sizes`, they are limited to static const int for the general case too.

Would we not be better saying:
Return type : integral constant expression
Return value : 2