Doc No: N1597=04-0037
Date: 17 Feb 2004
Reply to: Matt Austern
austern@apple.com

Library Extension Technical Report — Issues List

Revision 2: Pre-Sydney

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: New

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)

1.3 Reference to clause 17 should be stronger

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

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.

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: Voted into the 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: New

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.

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

Submitter: Joe Gottman
Status: New

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: New

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`?

2.6 dynamic_ptr_cast and deleters

Submitter: Alisdair Meredith
Status: New

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.

2.7 weak_ptr and deleters

Submitter: Alisdair Meredith
Status: New

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();
  }
}
```

2.8 Need equivalent of shared_ptr for arrays

Submitter: Alisdair Meredith
Status: New

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;
}
```

2.9 Proposed addition: const_pointer_cast

Submitter: Peter Dimov

Status: New

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.

3 Type traits issues

3.1 Use of Language in type transformations

Submitter: Pete Becker
Status: Voted into the 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: Voted into the 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: voted into the 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: voted into the 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: voted into the 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: Open

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: Open

See N1519 for discussion of the issue.

Resolution:
Discussed at Kona. Same status as **rank**: the LWG wasn't sure whether this was useful.

3.10 New type trait: aligned_storage

Submitter: John Maddock
Status: Voted into the 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: Voted into the 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: New

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: New

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.

3.14 Type_traits specifications could be simpler

Submitter: Pete Becker
Status: New

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.

4 Random number generator issues

4.1 Confusing Text in Description of v.min()

Submitter: Pete Becker (see N1535)
Status: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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 form`U&`.

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: Voted into the TR

For `normal_distribution`, the paper says that the probability density function is`1/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/sqrt(2*pi*sigma))e`$^{-(x-mean)2/(2*sigma2)}$, 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/(sqrt(2*pi)*sigma))e`$^{-(x-mean)2/(2*sigma2)}$, 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: Open

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.

Resolution: Discussed at Kona, no decision. The status quo is awkward, but we don't have a better solution yet. Pete and Jens will work on this and will propose a solution for Sydney.

4.7 Are Global Operators Overspecified?

Submitter: Pete Becker (see N1535)
Status: Open

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: Discussed at Kona, general agreement that we don't want to require a specific signature. Pete and Jens will provide wording along these lines.

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

Submitter: Pete Becker (see N1535)
Status: Voted into the 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 named`RealType` 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: Open

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.

4.10 Unclear Complexity Requirements for variate_generator

Submitter: Pete Becker (see N1535)
Status: Voted into the 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 form`U&`. 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 of`variate_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 that`variate_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` and`distribution_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: Voted into the 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 of`UniformRandomNumberGenerator1::result_type` and`UniformRandomNumberGenerator2::result_type` that provides the most storage [basic.fundamental].

4.13 subtract_with_carry's IntType Overpecified

Submitter: Pete Becker (see N1535)
Status: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the TR

For both `subtract_with_carry::seed(In& first, In last)` and`subtract_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 `w`bits. 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: Voted into the 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 for`linear_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 means`numeric_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 is`std::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: voted into the 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: Voted into the TR

This template takes three numeric arguments, `a`, `c`, and `m`, whose type is `IntType`. `IntType`is 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: Voted into the 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: Voted into the 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: Open

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.

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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the TR

Currently, there are no restrictions on `UniformRandomNumberGenerator::result_type`, although`variate_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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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: Voted into the 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.

5 Special function issues

5.1 Clean up special function names and descriptions

Submitter: Bill Plauger, Walter Brown
Status: Voted into the 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: New

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.

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

Submitter: Bill Plauger
Status: New

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.

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

Submitter: Bill Plauger
Status: New

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.

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

Submitter: Bill Plauger
Status: New

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.

5.6 Bessel should require domain error for x < 0

Submitter: Bill Plauger
Status: New

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.

6 Unordered associative container issues

6.1 Incorrect const qualification

Submitter: Rober Klarer
Status: Voted into the 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: Voted into the 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: Voted into the 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: New

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.

6.6 Unordered assoc containers not containers

Submitter: Beman Dawes
Status: New

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: New

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
Add to the list of exception safety guarantees:

> For unordered associative containers, an exception is thrown by a reshash() 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: New

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.

6.9 Unordered_map and unordered_multimap don't have assignable value types

Submitter: Rober Klarer
Status: New

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.)

7 Regular expression issues

7.1 basic_regex should Not Keep a Copy of its Initializer

Submitter: Pete Becker (N1499)
Status: Voted into the 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 functions`begin` 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: Voted into the 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: Open

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`, and`lookup_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:
The LWG thought this seemed like a good idea, but the details need to be worked out. Note that the iterators need to be ForwardIterator, not InputIterator.

7.4 Regular expressions and internationalization

Submitter: Pete Becker (N1500)
Status: Open

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:
Discussed at Kona, the LWG was generally sympathetic to this simplification. The one Japanese representative in the room thought that this was a good idea, and that it matches the way that Japenese programmers use regular expressions. The LWG believes we should make these changes at the next meeting, pending specific wording.

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: Voted into the 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: Voted into the 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: Open

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: Open

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: Voted into the 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: Voted into the 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, voted into the 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: voted into the 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: Open

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: voted into the 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: Open

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: voted into the 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: Open

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:
Discussed at Kona. This is a real problem. However, the wording proposed in N1507 doesn't solve the problem. The extra flags Pete notices are still there, and so is the problematic sentence about seemingly inappropriate functions being able to throw exceptions. Finally, the LWG wants to know the actual type of the thrown exception object.

7.19 Unneeded basic_regex members

Submitter: John Maddock
Status: voted into the 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: voted into the 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: voted into the 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: voted into the 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: voted into the 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: voted into the 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: voted into the 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: voted into the 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: Open

See N1507 for a full description. Summary: it's not clear what kind or regex object the default constructor returns, and how that interacts with the `empty()` test.

Resolution:
Discussed at Kona. The LWG agrees that the default constructor should be equivalent to construction from an empty string. Leaving this open for now partly because we need wording expressing that, and partly because it's not clear that there's any point to having the `empty()` member function in the first place.

7.29 Regular expression constructor language

Submitter: Pete Becker (N1507)
Status: Open

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: Voted into the 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: Voted into the 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: Open

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: Open

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

> If no match is 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.

with

> If no match is found, the implementation shall return the longest sequence [from, last), where *from != last*, for which it cannot determine that there is no possible sequence of characters [from, to) of which [from, last) is a prefix, and which would result in a full match. If no such sequence exists the match fails.

7.35 Name of regex_traits::is_class

Submitter: Pete Becker (N1507)
Status: Open

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: Open

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:
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: Open

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:
We think separating the case-insensitive match from the simple case-sensitive match is probably a good idea. Pete will provide wording for a specific proposal.

7.38 Improving on traits::toi

Submitter: Pete Becker (N1507)
Status: Open

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: New
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: New

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.

7.42 iterators have incorrect definition of difference_type

Submitter: Pete Becker
Status: New

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: New

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: New

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:
Keep ECMAScript, basic, extended, and awk. Get rid of the rest.

7.45 Names recognized by regex_traits::lookup_classname

Submitter: Pete Becker
Status: New

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.

7.46 Name of error_subreg

Submitter: Pete Becker
Status: New

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: New

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: New

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: New

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);
```

Effects: 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);
```

Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: 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);
```
Effects: returns lhs.str() <= rhs.

7.50 Convenience typedefs for regex_iterator and regex_token_iterator

Submitter: John Maddock
Status: New

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: New

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?

8 Fixed-size array issues

8.1 Is "array" the right name?

Submitter: Robert Klarer
Status: New

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.

9 Iterator concept and adapter issues

9.1 iterator_access overspecified?

Submitter: Pete Becker
Status: New

The proposal includes:

enum iterator_access { readable_iterator = 1, writable_iterator = 2, swappable_iterator = 4, lvalue_iterator = 8 };

In general, the standard specifies thing like this as a bitmask type with a list of defined names, and specifies neither the exact type nor the specific values. Is there a reason for iterator_access to be more specific?

Proposed resolution:
If the proposed resolution to 9.15 is accepted, then iterator_access will be removed.

9.2 operators of iterator_facade overspecified

Submitter: Pete Becker
Status: New

In general, we've provided operational semantics for things like operator++. That is, we've said that ++iter must work, without requiring either a member function or a non-member function. iterator_facade specifies most operators as member functions. There's no inherent reason for these to be members, so we should remove this requirement. Similarly, some operations are specified as non-member functions but could be implemented as members. Again, the standard doesn't make either of these choices, and TR1 shouldn't, either. So: operator*(), operator++(), operator++(int), operator--(), operator--(int), operator+=, operator-=, operator-(difference_type), operator-(iterator_facade instance), and operator+ should be specified with operational semantics and not explicitly required to be members or non-members.

Comments from proposal authors:

The standard uses valid expressions such as ++iter in requirements tables, such as for input iterator. However, for classes, such as reverse_iterator, the standard uses function prototypes, as we have done here for iterator_facade. Further, the prototype specification does not prevent the implementor from using members or non-members, since nothing the user can do in a conforming program can detect how the function is implemented.

9.3 enable_if_interoperable needs standardese

Submitter: Pete Becker
Status: New

The only discussion of what this means is in a note, so is non-normative. Further, the note seems to be incorrect. It says that `enable_if_interoperable` only works for types that "are interoperable, by which we mean they are convertible to each other." This requirement is too strong: it should be that one of the types is convertible to the other.

Proposed resolution from Pete:

Remove the enable_if_interoperable stuff, and just write all the comparisons to return bool. Then add a blanket statement that the behavior of these functions is undefined if the two types aren't interoperable.

Proposed resolution from Dave, Jeremy, and Thomas:
Change:

[*Note:* The `enable_if_interoperable` template used above is for exposition purposes. The member operators should be only be in an overload set provided the derived types `Dr1` and `Dr2` are interoperable, by which we mean they are convertible to each other. The `enable_if_interoperable` approach uses SFINAE to take the operators out of the overload set when the types are not interoperable.]

To:

The `enable_if_interoperable` template used above is for exposition purposes. The member operators should only be in an overload set provided the derived types `Dr1` and `Dr2` are interoperable, meaning that at least one of the types is convertible to the other. The `enable_if_interoperable` approach uses SFINAE to take the operators out of the overload set when the types are not interoperable. The operators should behave *as-if* `enable_if_interoperable` were defined to be:

```
template <bool, typename> enable_if_interoperable_impl
{};

template <typename T> enable_if_interoperable_impl<true,T>
{ typedef T type; };

template<typename Dr1, typename Dr2, typename T>
struct enable_if_interoperable
  : enable_if_interoperable_impl<
        is_convertible<Dr1,Dr2>::value || is_convertible<Dr2,Dr1>::value
      , T
    >
{};
```

9.4 enable_if_convertible unspecified, conflicts with requires

Submitter: Pete Becker
Status: New

In every place where `enable_if_convertible` is used it's used like this (simplified):

```
template<class T>
struct C
{
  template<class T1>
  C(T1, enable_if_convertible<T1, T>::type* = 0);
};
```

The idea being that this constructor won't compile if T1 isn't convertible to T. As a result, the constructor won't be considered as a possible overload when constructing from an object x where the type of x isn't convertible to T. In addition, however, each of these constructors has a requires clause that requires convertibility, so the behavior of a program that attempts such a construction is undefined. Seems like the enable_if_convertible part is irrelevant, and should be removed.

There are two problems. First, `enable_if_convertible` is never specified, so we don't know what this is supposed to do. Second: we could reasonably say that this overload should be disabled in certain cases or we could reasonably say that behavior is undefined, but we can't say both.

Thomas Witt writes that the goal of putting in `enable_if_convertible` here is to make sure that a specific overload doesn't interfere with the generic case except when that overload makes sense. He agrees that what we currently have is deficient.

Dave Abrahams writes that there is no conflict with the requires cause "because the requires clause only takes effect when the function is actually called. The presence of the constructor signature
can/will be detected by is_convertible without violating the requires clause, and thus it makes a difference to disable those constructor instantiations that would be disabled by enable_if_convertible even if calling them invokes undefined behavior."

There was more discussion on the reflector: c++std-lib-12312, c++std-lib-12325, c++std-lib-12330, c++std-lib-12334, c++std-lib-12335, c++std-lib-12336, c++std-lib-12338, c++std-lib-12362.

Proposed resolution from Pete:
Specify enable_if_convertible to be as-if:

```
template <bool> enable_if_convertible_impl
{};

template <> enable_if_convertible_impl<true>
{ struct type; };

template<typename From, typename To>
struct enable_if_convertible
  : enable_if_convertible_impl<
        is_convertible<From, To>::value
{};
```

Proposed resolution from Dave, Jeremy, Thomas:
Change:
[*Note:* The `enable_if_convertible<X,Y>::type` expression used in this section is for exposition purposes. The converting constructors for specialized adaptors should be only be in an overload set provided that an object of type `X` is implicitly convertible to an object of type `Y`. The `enable_if_convertible` approach uses SFINAE to take the constructor out of the overload set when the types are not implicitly convertible.]

To:

The `enable_if_convertible<X,Y>::type` expression used in this section is for exposition purposes. The converting constructors for specialized adaptors should be only be in an overload set provided that an object of type `X` is implicitly convertible to an object of type `Y`. The signatures involving `enable_if_convertible` should behave *as-if* `enable_if_convertible` were defined to be:

```
template <bool> enable_if_convertible_impl
{};

template <> enable_if_convertible_impl<true>
{ struct type; };

template<typename From, typename To>
struct enable_if_convertible
  : enable_if_convertible_impl<is_convertible<From,To>::value>
{};
```

If an expression other than the default argument is used to supply the value of a function parameter whose type is written in terms of `enable_if_convertible`, the program is ill-formed, no diagnostic required.

[*Note:* The `enable_if_convertible` approach uses SFINAE to take the constructor out of the overload set when the types are not implicitly convertible.]

9.5 iterator_adaptor has an extraneous 'bool' at the start of the template definition

Submitter: Pete Becker
Status: New

The title says it all; this is probably just a typo.

Proposed resolution:
Get rid of it

9.6 Name of private member shouldn’t be normative

Submitter: Pete Becker
Status: New

iterator_adaptor has a private member named m_iterator. Presumably this is for exposition only, since it's an implementation detail. It needs to be marked as such.

Proposed resolution:
Change:

```
Base m_iterator;
```

to:

```
Base m_iterator; // exposition only
```

9.7 iterator_adaptor operations specifications are a bit inconsistent

Submitter: Pete Becker
Status: New

iterator_adaptor() has a Requires clause, that Base must be default constructible.

iterator_adaptor(Base) has no Requires clause, although the Returns clause says that the Base member is copy construced from the argument (this may actually be an oversight in N1550, which doesn't require iterators to be copy constructible or assignable).

Proposed resolution:
Add a requirements section for the template parameters of iterator_adaptor, and state that Base must be Copy Constructible and Assignable.

9.8 Specialized adaptors text should be normative

Submitter: Pete Becker
Status: New

similar to 9.3, "Specialized Adaptors" has a note describing enable_if_convertible. This should be normative text.

Proposed resolution:
If the proposed resolution to issue 9.4 is accepted, it will cover this as well.

9.9 Reverse_iterator text is too informal

Submitter: Pete Becker
Status: New

reverse iterator "flips the direction of the base iterator's motion". This needs to be more formal, as in the current standard. Something like: "iterates through the controlled sequence in the opposite direction"

Proposed resolution:
Change:
The reverse iterator adaptor flips the direction of a base iterator's motion. Invoking `operator++()` moves the base iterator backward and invoking `operator--()` moves the base iterator forward.

to:
The reverse iterator adaptor iterates through the adapted iterator range in the opposite direction.

9.10 "prior" is undefined

Submitter: Pete Becker
Status: New

reverse_iterator::dereference is specified as calling a function named 'prior' which has no specification.

Proposed resolution:
Change the specification to avoid using `prior` as follows.

Remove:
```
typename reverse_iterator::reference dereference() const {
return *prior(this->base()); }
```

And at the end of the operations section add:

```
reference operator*() const;
```

Effects:
```
Iterator tmp = m_iterator;
return *--tmp;
```

9.11 "In other words" is bad wording

Submitter: Pete Becker
Status: New

Transform iterator has a two-part specification: it does this, in other words, it does that. "In other words" always means "I didn't say it right, so I'll try again." We need to say it once.

Proposed resolution:
Change:
The transform iterator adapts an iterator by applying some function object to the result of dereferencing the iterator. In other words, the `operator*` of the transform iterator first dereferences the base iterator, passes the result of this to the function object, and then returns the result.

to:
The transform iterator adapts an iterator by modifying the `operator*` to apply a function object to the result of dereferencing the iterator and returning the result.

9.12 Transform_iterator shouldn't mandate private member

Submitter: Pete Becker
Status: New

transform_iterator has a private member named 'm_f' which should be marked "exposition only."

Proposed resolution:
Change:
```
UnaryFunction m_f;
```

to:
```
UnaryFunction m_f;   // exposition only
```

9.13 Unclear description of counting iterator

Submitter: Pete Becker
Status: New

The description of Counting iterator is unclear. "The counting iterator adaptor implements dereference by returning a reference to the base object. The other operations are implemented by the base m_iterator, as per the inheritance from iterator_adaptor."

Proposed resolution:
Change:
The counting iterator adaptor implements dereference by returning a reference to the base object. The other operations are implemented by the base `m_iterator`, as per the inheritance from `iterator_adaptor`.

to:
`counting_iterator` adapts an object by adding an `operator*` that returns the current value of the object. All other iterator operations are forwarded to the adapted object.

9.14 Counting_iterator's difference type

Submitter: Pete Becker
Status: New

Counting iterator has the following note:

> [Note: implementers are encouraged to provide an implementation of distance_to and a difference_type that avoids overflows in the cases when the Incrementable type is a numeric type.]

I'm not sure what this means. The user provides a template argument named Difference, but there's no difference_type. I assume this is just a glitch in the wording. But if implementors are encouraged to ignore this argument if it won't work right, why is it there?

9.15 How to detect lvalueness?

Submitter: Dave Abrahams
Status: New

Shortly after N1550 was accepted, we discovered that an iterator's lvalueness can be determined knowing only itsvalue_type. This predicate can be calculated even for old-style iterators (on whose reference type the standard places few requirements). A trait in the Boost iterator library does it by relying on the compiler's unwillingness to bind an rvalue to a T& function template parameter. Similarly, it is possible to detect an iterator's readability knowing only itsvalue_type. Thus, any interface which asks the user to explicitly describe an iterator's lvalue-ness or readability seems to introduce needless complexity.

Proposed resolution:
1. Remove the `is_writable` and `is_swappable` traits, and remove the requirements in the Writable Iterator and Swappable Iterator concepts that require their models to support these traits.
2. Change the `is_readable` specification. Remove the requirement for support of the `is_readable` trait from the Readable Iterator concept.
3. Remove the `iterator_tag` class and transplant the logic for choosing an iterator category into `iterator_facade` .
4. Change the specification of `traversal_category`.
5. Remove Access parameters from N1530

In N1550:

Remove:

Since the access concepts are not related via refinement, but instead cover orthogonal issues, we do not use tags for the access concepts, but instead use the equivalent of a bit field.

We provide an access mechanism for mapping iterator types to the new traversal tags and access bit field. Our design reuses `iterator_traits<Iter>::iterator_category` as the access mechanism. To that end, the access and traversal information is bundled into a single type using the following *iterator_tag* class.

```
enum iterator_access { readable_iterator = 1,
writable_iterator = 2,
    swappable_iterator = 4, lvalue_iterator = 8 };

template <unsigned int access_bits, class TraversalTag>
struct iterator_tag : /* appropriate old category or categories
*/ {
  static const iterator_access access =
    (iterator_access)access_bits &
      (readable_iterator | writable_iterator |
swappable_iterator);
  typedef TraversalTag traversal;
};
```

The `access_bits` argument is declared to be `unsigned int` instead of the enum `iterator_access` for convenience of use. For example, the expression `(readable_iterator | writable_iterator)` produces an unsigned int, not an `iterator_access`. The purpose of the `lvalue_iterator` part of the `iterator_access` enum is to communicate to `iterator_tag` whether the reference type is an lvalue so that the appropriate old category can be chosen for the base class. The `lvalue_iterator` bit is not recorded in the `iterator_tag::access` data member.

The `iterator_tag` class template is derived from the appropriate iterator tag or tags from the old requirements based on the access bits and traversal tag passed as template parameters. The algorithm for determining the old tag or tags picks the least refined old concepts that include all of the requirements of the access and traversal concepts (that is, the closest fit), if any such category exists. For example, the category tag for a Readable Single Pass Iterator will always be derived from `input_iterator_tag`, while the category tag for a Single Pass Iterator that is both Readable and Writable will be derived from both `input_iterator_tag` and `output_iterator_tag`.

We also provide several helper classes that make it convenient to obtain the access and traversal characteristics of an iterator. These helper classes work both for iterators whose `iterator_category` is `iterator_tag` and also for iterators using the original iterator categories.

```
template <class Iterator> struct is_readable  { typedef ...
type; };
template <class Iterator> struct is_writable { typedef ... type;
};
template <class Iterator> struct is_swappable { typedef ...
type; };
template <class Iterator> struct traversal_category { typedef
... type; };
```

After:
Like the old iterator requirements, we provide tags for purposes of dispatching based on the traversal concepts. The tags are related via inheritance so that a tag is convertible to another tag if the concept associated with the first tag is a refinement of the second tag.

Add:

Our design reuses `iterator_traits<Iter>::iterator_category` to indicate an iterator's traversal capability. To specify capabilities not captured by any old-style iterator category, an iterator designer can use an `iterator_category` type that is convertible to both the the most-derived old iterator category tag which fits, and the appropriate new iterator traversal tag.

We do not provide tags for the purposes of dispatching based on the access concepts, in part because we could not find a way to automatically infer the right access tags for old-style iterators. An iterator's writability may be dependent on the assignability of its `value_type` and there's no known way to detect whether an arbitrary type is assignable. Fortunately, the need for dispatching based on access capability is not as great as the need for dispatching based on traversal capability.

From the Readable Iterator Requirements table, remove:

```
is_readable<X>::type
true_type
```

From the Writable Iterator Requirements table, remove:

```
is_writable<X>::type
true_type
```

From the Swappable Iterator Requirements table, remove:

```
is_swappable<X>::type
true_type
```

From [lib.iterator.synopsis] replace:

```
template <class Iterator> struct is_readable;
template <class Iterator> struct is_writable;
template <class Iterator> struct is_swappable;
template <class Iterator> struct traversal_category;

enum iterator_access { readable_iterator = 1, writable_iterator
= 2,
    swappable_iterator = 4, lvalue_iterator = 8 };

template <unsigned int access_bits, class TraversalTag>
struct iterator_tag : /* appropriate old category or categories
*/ {
  static const iterator_access access =
```

```
    (iterator_access)access_bits &
      (readable_iterator | writable_iterator |
swappable_iterator);
  typedef TraversalTag traversal;
};
```

with:

```
template <class Iterator> struct is_readable_iterator;
template <class Iterator> struct iterator_traversal;
```

In [lib.iterator.traits], remove:

The `iterator_tag` class template is an iterator category tag that encodes the access enum and traversal tag in addition to being compatible with the original iterator tags. The `iterator_tag` class inherits from one of the original iterator tags according to the following pseudo-code.

```
inherit-category(access, traversal-tag) =
     if ((access & readable_iterator) && (access &
lvalue_iterator)) {
         if (traversal-tag is convertible to
random_access_traversal_tag)
             return random_access_iterator_tag;
         else if (traversal-tag is convertible to
bidirectional_traversal_tag)
             return bidirectional_iterator_tag;
         else if (traversal-tag is convertible to
forward_traversal_tag)
             return forward_iterator_tag;
         else if (traversal-tag is convertible to
single_pass_traversal_tag)
             if (access-tag is convertible to
writable_iterator_tag)
                 return input_output_iterator_tag;
             else
                 return input_iterator_tag;
         else
             return null_category_tag;
     } else if ((access & readable_iterator) and (access &
writable_iterator)
                and traversal-tag is convertible to
single_pass_iterator_tag)
         return input_output_iterator_tag;
     else if (access & readable_iterator
              and traversal-tag is convertible to
single_pass_iterator_tag)
         return input_iterator_tag;
     else if (access & writable_iterator
              and traversal-tag is convertible to
incrementable_iterator_tag)
```

```
        return output_iterator_tag;
    else
        return null_category_tag;
```

If the argument for `TraversalTag` is not convertible to `incrementable_iterator_tag` then the program is ill-formed.

Change:

The `is_readable`, `is_writable`, `is_swappable`, and `traversal_category` class templates are traits classes. For iterators whose `iterator_traits<Iter>::iterator_category` type is `iterator_tag`, these traits obtain the `access` enum and `traversal` member type from within `iterator_tag`. For iterators whose `iterator_traits<Iter>::iterator_category` type is not `iterator_tag` and instead is a tag convertible to one of the original tags, the appropriate traversal tag and access bits are deduced. The following pseudo-code describes the algorithm.

```
is-readable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & readable_iterator;
    else if (cat is convertible to input_iterator_tag)
        return true;
    else
        return false;

is-writable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & writable_iterator;
    else if (cat is convertible to output_iterator_tag)
         return true;
    else if (
         cat is convertible to forward_iterator_tag
         and iterator_traits<Iterator>::reference is a
             mutable reference)
        return true;
    else
        return false;

is-swappable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & swappable_iterator;
    else if (cat is convertible to forward_iterator_tag) {
        if (iterator_traits<Iterator>::reference is a const
reference)
            return false;
        else
            return true;
    } else
```

```
        return false;

traversal-category(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Traversal;
    else if (cat is convertible to random_access_iterator_tag)
        return random_access_traversal_tag;
    else if (cat is convertible to bidirectional_iterator_tag)
        return bidirectional_traversal_tag;
    else if (cat is convertible to forward_iterator_tag)
        return forward_traversal_tag;
    else if (cat is convertible to input_iterator_tag)
        return single_pass_iterator_tag;
    else if (cat is convertible to output_iterator_tag)
        return incrementable_iterator_tag;
    else
        return null_category_tag;
```

The following specializations provide the access and traversal category tags for pointer types.

```
template <typename T>
struct is_readable<const T*> { typedef true_type type; };
template <typename T>
struct is_writable<const T*> { typedef false_type type; };
template <typename T>
struct is_swappable<const T*> { typedef false_type type; };

template <typename T>
struct is_readable<T*> { typedef true_type type; };
template <typename T>
struct is_writable<T*> { typedef true_type type; };
template <typename T>
struct is_swappable<T*> { typedef true_type type; };

template <typename T>
struct traversal_category<T*>
{
  typedef random_access_traversal_tag type;
};
```

to:

The `is_readable_iterator` class template satisfies the UnaryTypeTrait requirements.

Given an iterator type `X`, `is_readable_iterator<X>::value` yields `true` if, for an object `a` of type `X`, `*a` is convertible to `iterator_traits<X>::value_type`, and `false` otherwise.

`iterator_traversal<X>::type` is

category-to-traversal`(iterator_traits<X>::iterator_category)`

where *category-to-traversal* is defined as follows

```
category-to-traversal(C) =
    if (C is convertible to incrementable_traversal_tag)
        return C;
    else if (C is convertible to random_access_iterator_tag)
        return random_access_traversal_tag;
    else if (C is convertible to bidirectional_iterator_tag)
        return bidirectional_traversal_tag;
    else if (C is convertible to forward_iterator_tag)
        return forward_traversal_tag;
    else if (C is convertible to input_iterator_tag)
        return single_pass_traversal_tag;
    else if (C is convertible to output_iterator_tag)
        return incrementable_traversal_tag;
    else
        the program is ill-formed
```

In N1530:

In [lib.iterator.helper.synopsis]:

Change:

```
const unsigned use_default_access = -1;

struct iterator_core_access { /* implementation detail */ };

template <
    class Derived
  , class Value
  , unsigned AccessCategory
  , class TraversalCategory
  , class Reference  = Value&
  , class Difference = ptrdiff_t
>
class iterator_facade;

template <
    class Derived
  , class Base
  , class Value      = use_default
  , unsigned Access  = use_default_access
  , class Traversal  = use_default
  , class Reference  = use_default
  , class Difference = use_default
>
class iterator_adaptor;
```

```
template <
    class Iterator
  , class Value = use_default
  , unsigned Access  = use_default_access
  , class Traversal  = use_default
  , class Reference = use_default
  , class Difference = use_default
>
class indirect_iterator;
```

To:

```
struct iterator_core_access { /* implementation detail */ };

template <
    class Derived
  , class Value
  , class CategoryOrTraversal
  , class Reference  = Value&
  , class Difference = ptrdiff_t
>
class iterator_facade;

template <
    class Derived
  , class Base
  , class Value      = use_default
  , class CategoryOrTraversal  = use_default
  , class Reference  = use_default
  , class Difference = use_default
>
class iterator_adaptor;

template <
    class Iterator
  , class Value = use_default
  , class CategoryOrTraversal = use_default
  , class Reference = use_default
  , class Difference = use_default
>
class indirect_iterator;
```

Change:

```
template <
    class Incrementable
  , unsigned Access  = use_default_access
  , class Traversal  = use_default
  , class Difference = use_default
>
```

```
class counting_iterator
```

To:

```
template <
    class Incrementable
  , class CategoryOrTraversal  = use_default
  , class Difference = use_default
>
class counting_iterator;
```

In [lib.iterator.facade]:

Change:

```
template <
    class Derived
  , class Value
  , unsigned AccessCategory
  , class TraversalCategory
  , class Reference  = /* see below */
  , class Difference = ptrdiff_t
>
class iterator_facade {
```

to:

```
template <
    class Derived
  , class Value
  , class CategoryOrTraversal
  , class Reference  = Value&
  , class Difference = ptrdiff_t
>
class iterator_facade {
```

Change:

```
typedef iterator_tag<AccessCategory, TraversalCategory>
iterator_category;
```

to:

```
typedef /* see below */ iterator_category;
```

Change:

```
// Comparison operators
template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
```

```
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type //
exposition
operator ==(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
            iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator !=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
            iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator <(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator <=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
            iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
```

```
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
            iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
            iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

// Iterator difference
template <class Dr1, class V1, class AC1, class TC1, class R1,
class D1,
          class Dr2, class V2, class AC2, class TC2, class R2,
class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator -(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const&
lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const&
rhs);

// Iterator addition
template <class Derived, class V, class AC, class TC, class R,
class D>
Derived operator+ (iterator_facade<Derived, V, AC, TC, R, D>
const&,
                   typename Derived::difference_type n)
```

to:

```
// Comparison operators
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type //
exposition
operator ==(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator !=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
```

```
             iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator <(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator <=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator >(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator >=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

// Iterator difference
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
/* see below */
operator-(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
          iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

// Iterator addition
template <class Dr, class V, class TC, class R, class D>
Derived operator+ (iterator_facade<Dr,V,TC,R,D> const&,
                   typename Derived::difference_type n);

template <class Dr, class V, class TC, class R, class D>
Derived operator+ (typename Derived::difference_type n,
                   iterator_facade<Dr,V,TC,R,D> const&);
```

After the `iterator_facade` synopsis, add:

The `iterator_category` member of `iterator_facade` is
iterator-category`(CategoryOrTraversal, value_type, reference)`

where *iterator-category* is defined as follows:

```
iterator-category(C,R,V) :=
   if (C is convertible to std::input_iterator_tag
       || C is convertible to std::output_iterator_tag
   )
       return C

   else if (C is not convertible to incrementable_traversal_tag)
       the program is ill-formed

   else return a type X satisfying the following two
constraints:

      1. X is convertible to X1, and not to any more-derived
         type, where X1 is defined by:

           if (R is a reference type
               && C is convertible to forward_traversal_tag)
           {
               if (C is convertible to
random_access_traversal_tag)
                   X1 = random_access_iterator_tag
               else if (C is convertible to
bidirectional_traversal_tag)
                   X1 = bidirectional_iterator_tag
               else
                   X1 = forward_iterator_tag
           }
           else
           {
               if (C is convertible to single_pass_traversal_tag
                   && R is convertible to V)
                   X1 = input_iterator_tag
               else
                   X1 = C
           }

      2. category-to-traversal(X) is convertible to the most
         derived traversal tag type to which X is also
         convertible, and not to any more-derived traversal tag
         type.
```

In [lib.iterator.facade] `iterator_facade` requirements:

Remove:
`AccessCategory` must be an unsigned value which uses no more bits than the greatest value of `iterator_access`.

In the **Iterator Adaptor** section, change:
Several of the template parameters of `iterator_adaptor` default to `use_default` (or `use_default_access`).

to:
Several of the template parameters of `iterator_adaptor` default to `use_default`.

In [lib.iterator.special.adaptors]:

Change:
```
template <
    class Iterator
  , class Value = use_default
  , unsigned Access  = use_default_access
  , class Traversal  = use_default
  , class Reference = use_default
  , class Difference = use_default
>
class indirect_iterator
```

to:
```
template <
    class Iterator
  , class Value = use_default
  , class CategoryOrTraversal = use_default
  , class Reference = use_default
  , class Difference = use_default
>
class indirect_iterator
```

Change:
```
template <
    class Iterator2, class Value2, unsigned Access2, class
Traversal2
  , class Reference2, class Difference2
>
indirect_iterator(
```

to:
```
template <
    class Iterator2, class Value2, class Category2
  , class Reference2, class Difference2
>
indirect_iterator(
```

Change:
```
template <
    class Incrementable
  , unsigned Access = use_default_access
  , class Traversal = use_default
```

```
  , class Difference = use_default
>
class counting_iterator
```

to:

```
template <
    class Incrementable
  , class CategoryOrTraversal = use_default
  , class Difference = use_default
>
class counting_iterator
```

Change:

```
typedef iterator_tag<
      writable_iterator
    , incrementable_traversal_tag
> iterator_category;
```

to:

typedef std::output_iterator_tag iterator_category;

In [lib.iterator.adaptor]

Change:

```
template <
    class Derived
  , class Base
  , class Value      = use_default
  , unsigned Access  = use_default_access
  , class Traversal  = use_default
  , class Reference  = use_default
  , class Difference = use_default
>
class iterator_adaptor
```

To:

```
template <
    class Derived
  , class Base
  , class Value               = use_default
  , class CategoryOrTraversal = use_default
  , class Reference           = use_default
  , class Difference = use_default
>
class iterator_adaptor
```

Rationale:

1. There are two reasons for removing `is_writable` and `is_swappable`. The first is that we do not know of a way to fix the specification so that it gives the correct answer for all iterators. Second, there was only a weak motivation for having `is_writable` and `is_swappable` there in the first place. The main motivation was simply uniformity: we have tags for the old iterator categories so we should have tags for the new iterator categories. While having tags and the capability to dispatch based on the traversal categories is often used, we see less of a need for dispatching based on writability and swappability, since typically algorithms that need these capabilities have no alternative if they are not provided.
2. We discovered that the `is_readable` trait can be implemented using only the iterator type itself and its `value_type`. Therefore we remove the requirement for `is_readable` from the Readable Iterator concept, and change the definition of `is_readable` so that it works for any iterator type.
3. The purpose of the `iterator_tag` class was to bundle the traversal and access category tags into the `iterator_category` typedef. With `is_writable` and `is_swappable` gone, and `is_readable` no longer in need of special hints, there is no reason for iterators to provide information about the access capabilities of an iterator. Thus there is no need for the `iterator_tag`. The traversal tag can be directly used for the `iterator_category`. If a new iterator is intended to be backward compatible with old iterator concepts, a tag type that is convertible to both one of the new traversal tags and also to an old iterator tag can be created and use for the `iterator_category`.
4. The changes to the specification of `traversal_category` are a direct result of the removal of `iterator_tag`.

9.16 is_writable_iterator returns false positives

Submitter: Dave Abrahams
Status: New

is_writable_iterator returns false positives for forward iterators whose value_type has a private assignment operator, or whose reference type is not a reference (currently legal).

Proposed resolution:
If the proposed resolution to 9.15 is accepted, it also covers this issue.

9.17 is_swappable_iterator returns false positives

Submitter: Dave Abrahams
Status: New

is_swappable_iterator has the same problems as is_writable_iterator. In addition, if we allow users to write their own iter_swap functions it's easy to imagine old-style iterators for which is_swappable returns false negatives.

Proposed resolution:
If the proposed resolution to 9.15 is accepted, it also covers this issue.

9.18 Are is_readable, is_writable, and is_swappable useful?

Submitter: Dave Abrahams
Status: New

I am concerned that there is little use for any of is_readable, is_writable, or is_swappable, and that not only do they unduly constrain iterator implementors but they add overhead to iterator_facade and iterator_adaptor in the form of a template parameter which would otherwise be unneeded. Since we can't implement two of them accurately for old-style iterators, I am having a hard time justifying their impact on the rest of the proposal(s).

Proposed resolution:
If the proposed resolution to 9.15 is accepted, it also covers this issue.

9.19 Non-Uniformity of the "lvalue_iterator Bit"

Submitter: Dave Abrahams
Status: New

The proposed iterator_tag class template accepts an "access bits" parameter which includes a bit to indicate the iterator's lvalueness (whether its dereference operator returns a reference to its value_type. The relevant part of N1550 says:

> The purpose of the lvalue_iterator part of the iterator_access enum is to communicate to iterator_tagwhether the reference type is an lvalue so that the appropriate old category can be chosen for the base class. The lvalue_iterator bit is not recorded in the iterator_tag::access data member.

The lvalue_iterator bit is not recorded because N1550 aims to improve orthogonality of the iterator concepts, and a new-style iterator's lvalueness is detectable by examining its reference type. This inside/outside difference is awkward and confusing.

Proposed resolution:
If the proposed resolution to 9.15 is accepted, it also covers this issue.

9.20 Traversal Concepts and Tags

Submitter: Dave Abrahams
Status: New

Howard Hinnant pointed out some inconsistencies with the naming of these tag types:

```
incrementable_iterator_tag            // ++r, r++
single_pass_iterator_tag              // adds a == b, a != b
forward_traversal_iterator_tag        // adds multi-pass
bidirectional_traversal_iterator_tag  // adds --r, r--
random_access_traversal_iterator_tag  // adds r+n,n+r,etc.
```

Howard thought that it might be better if all tag names contained the word "traversal".

It's not clear that would result in the best possible names, though. For example, incrementable iterators can only make a single pass over their input. What really distinguishes single pass iterators from incrementable iterators is not that they can make a single pass, but that they are equality comparable. Forward traversal iterators really distinguish themselves by introducing multi-pass capability. Without entering a "Parkinson's Bicycle Shed" type of discussion, it might be worth giving the names of these tags (and the associated concepts) some extra attention.

Proposed resolution:
Change the names of the traversal tags to the following names:

```
incrementable_traversal_tag
single_pass_traversal_tag
forward_traversal_tag
bidirectional_traversal_tag
random_access_traversal_tag
```

In [lib.iterator.traversal]:

Change:

`traversal_category<X>::type`
Convertible to `incrementable_iterator_tag`

to:

`iterator_traversal<X>::type`
Convertible to `incrementable_traversal_tag`

Change:

`traversal_category<X>::type`
Convertible to `single_pass_iterator_tag`

to:

`iterator_traversal<X>::type`
Convertible to `single_pass_traversal_tag`

Change:

`traversal_category<X>::type`
Convertible to `forward_traversal_iterator_tag`

to:

`iterator_traversal<X>::type`
Convertible to `forward_traversal_tag`

Change:

`traversal_category<X>::type`
Convertible to `bidirectional_traversal_iterator_tag`

to:

```
iterator_traversal<X>::type
```
Convertible to `bidirectional_traversal_tag`

Change:

```
traversal_category<X>::type
```
Convertible to `random_access_traversal_iterator_tag`

to:

```
iterator_traversal<X>::type
```
Convertible to `random_access_traversal_tag`

In [lib.iterator.synopsis], change:

```
struct incrementable_iterator_tag { };
struct single_pass_iterator_tag : incrementable_iterator_tag {
};
struct forward_traversal_tag : single_pass_iterator_tag { };
```

to:

```
struct incrementable_traversal_tag { };
struct single_pass_traversal_tag : incrementable_traversal_tag {
};
struct forward_traversal_tag : single_pass_traversal_tag { };
```

Remove:

```
struct null_category_tag { };
struct input_output_iterator_tag : input_iterator_tag,
output_iterator_tag {};
```

9.21 iterator_facade Derived template argument underspecified

Submitter: Pete Becker
Status: New

The first template argument to iterator_facade is named Derived, and the proposal says:

> The Derived template parameter must be a class derived from iterator_facade.

First, iterator_facade is a template, so cannot be derived from. Rather, the class must be derived from a specialization of iterator_facade. More important, isn't Derived required to be the class that is being defined? That is, if I understand it right, the definition of D here this is not valid:

class C : public iterator_facade<C, ... > { ... };

class D : public iterator_facade<C, ...> { ... };

In the definition of D, the Derived argument to iterator_facade is a class derived from a specialization of iterator_facade, so the requirement is met. Shouldn't the requirement be more like "when using iterator_facade to define an iterator class Iter, the class Iter must be derived from a specialization of iterator_facade whose first template argument is Iter." That's a bit awkward, but at the moment I don't see a better way of phrasing it.

Proposed resolution:
In [lib.iterator.facade]

Remove:
The `Derived` template parameter must be a class derived from `iterator_facade`.

Change:
The following table describes the other requirements on the `Derived` parameter. Depending on the resulting iterator's `iterator_category`, a subset of the expressions listed in the table are required to be valid. The operations in the first column must be accessible to member functions of class `iterator_core_access`.

to:
The following table describes the typical valid expressions on `iterator_facade`'s `Derived` parameter, depending on the iterator concept(s) it will model. The operations in the first column must be made accessible to member functions of class `iterator_core_access`. In addition, `static_cast<Derived*>(iterator_facade*)` shall be well-formed.

In [lib.iterator.adaptor]

Change:
The `iterator_adaptor` is a base class template derived from an instantiation of `iterator_facade` .

to:
Each specialization of the `iterator_adaptor` class template is derived from a specialization of `iterator_facade`.

Change:
The `Derived` template parameter must be a derived class of `iterator_adaptor`.

To:
`static_cast<Derived*>(iterator_adaptor*)` shall be well-formed.

[Note: The proposed resolution to Issue 9.37 contains related changes]

9.22 return type of Iterator difference for iterator facade

Submitter: Pete Becker
Status: New

The proposal says:

```
template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
          class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator -(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);
```

Shouldn't the return type be one of the two iterator types? Which one? The idea is that if one of the iterator types can be converted to the other type, then the subtraction is okay. Seems like the return type should then be the type that was converted to. Is that right?

Proposed resolution:
If the proposed resolution to 9.34 is accepted, it also covers this issue.

9.23 Iterator_facade: minor wording Issue

Submitter: Pete Becker
Status: New

In the table that lists the required (sort of) member functions of iterator types that are based on iterator_facade, the entry for c.equal(y) says:

> true iff c and y refer to the same position. Implements c == y and c != y.

The second sentence is inside out. c.equal(y) does not implement either of these operations. It is used to implement them. Same thing in the description of c.distance_to(z).

Proposed resolution:
remove "implements" descriptions from table. [See resolution to 9.34]

9.24 Use of undefined name in iterator_facade table

Submitter: Pete Becker
Status: New

Several of the descriptions use the name X without defining it. This seems to be a carryover from the table immediately above this section, but the text preceding that table says "In the table below, X is the derived iterator type." Looks like the X:: qualifiers aren't really needed; X::reference can simply be reference, since that's defined by the iterator_facade specialization itself.

Proposed resolution:

In [lib.iterator.facade] operations `operator->() const;`:

Change:

Returns:

If `X::reference` is a reference type, an object of type `X::pointer` equal to:
`&static_cast<Derived const*>(this)->dereference()`

Otherwise returns an object of unspecified type such that, given an object `a` of type `X`, `a->m` is equivalent to `(w = *a, w.m)` for some temporary object `w` of type `X::value_type`.

The type `X::pointer` is `Value*` if `is_writable_iterator<X>::value` is `true`, and `Value const*` otherwise.

to:

Returns:

If `reference` is a reference type, an object of type `pointer` equal to:
`&static_cast<Derived const*>(this)->dereference()`

Otherwise returns an object of unspecified type such that, `(*static_cast<Derived const*>(this))->m` is equivalent to `(w = **static_cast<Derived const*>(this), w.m)` for some temporary object `w` of type `value_type` .

9.25 Iterator_facade: wrong return type

Submitter: Pete Becker
Status: New

Several of the member functions return a Derived object or a Derived&. Their Effects clauses end with:

```
return *this;
```

This should be

```
return *static_cast<Derived*>(this);
```

Proposed resolution:
In [lib.iterator.facade], in the effects clause of the following operations:

```
Derived& operator++()
Derived& operator--()
Derived& operator+=(difference_type n)
Derived& operator-=(difference_type n)
```

Change:
`return *this`
to:
`return *static_cast<Derived*>(this);`

9.26 Iterator_facade: unclear returns clause for operator[]

Submitter: Pete Becker
Status: New

The returns clause for operator[](difference_type n) const says:

> Returns: an object convertible to X::reference and holding a copy p of a+n such that, for a constant object v of type X::value_type, X::reference(a[n] = v) is equivalent to p = v.

This needs to define 'a', but assuming it's supposed to be *this (or maybe *(Derived*)this), it still isn't clear what this says. Presumably, the idea is that you can index off of an iterator and assign to the result. But why the requirement that it hold a copy of a+n? Granted, that's probably how it's implemented, but it seems over-constrained. And the last phrase seems wrong. p is an iterator; there's no requirement that you can assign a value_type object to it. Should that be *p = v? But why the cast in reference(a[n] = v)?

Proposed resolution:
In section operator[]:

Change:
Writable iterators built with `iterator_facade` implement the semantics required by the preferred resolution to *issue 299* and adopted by proposal *n1477*: the result of `p[n]` is a proxy object containing a copy of `p+n`, and `p[n] = x` is equivalent to `*(p + n) = x`. This approach will work properly for any random-access iterator regardless of the other details of its implementation. A user who knows more about the implementation of her iterator is free to implement an `operator[]` which returns an lvalue in the derived iterator class; it will hide the one supplied by `iterator_facade` from clients of her iterator.

to:
Writable iterators built with `iterator_facade` implement the semantics required by the preferred resolution to *issue 299* and adopted by proposal *n1550*: the result of `p[n]` is an object convertible to the iterator's `value_type` , and `p[n] = x` is equivalent to `*(p + n) = x` (Note: This result object may be implemented as a proxy containing a copy of `p+n`). This approach will work properly for any random-access iterator regardless of the other details of its implementation. A user who knows more about the implementation of her iterator is free to implement an `operator[]` that returns an lvalue in the derived iterator class; it will hide the one supplied by `iterator_facade` from clients of her iterator.

In [lib.iterator.facade] operations:

Change:

Returns:
an object convertible to `X::reference` and holding a copy *p* of `a+n` such that, for a constant object `v` of type `X::value_type` , `X::reference(a[n] = v)` is equivalent to `p = v`.

to:

Returns:
an object convertible to `value_type`. For constant objects `v` of type `value_type`, and `n` of type `difference_type`, `(*this)[n] = v` is equivalent to `*(*this + n) = v`, and `static_cast<value_type const&>((*this)[n])` is equivalent to `static_cast<value_type const&>(*(*this + n))`

9.27 Iterator_facade: redundant clause

Submitter: Pete Becker
Status: New

operator- has both an effects clause and a returns clause. Looks like the returns clause should be removed.

Proposed resolution:

Remove the returns clause.

In [lib.iterator.facade] operations:
Remove:

Returns:
`static_cast<Derived const*>(this)->advance(-n);`

9.28 indirect_iterator: incorrect specification of default constructor

Submitter: Pete Becker
Status: New

The default constructor returns "An instance of indirect_iterator with a default constructed base object", but the constructor that takes an Iterator object returns "An instance of indirect_iterator with the iterator_adaptor subobject copy constructed from x." The latter is the correct form, since it does not reach inside the base class for its semantics. So the default constructor shoudl return "An instance of indirect_iterator with a default-constructed iterator_adaptor subobject."

Proposed resolution:

Change:

Returns:
An instance of `indirect_iterator` with a default constructed base object.
to:

Returns:
An instance of `indirect_iterator` with a default-constructed `m_iterator` .

Rationale:
Inheritance from iterator_adaptor has been removed, so we instead give the semantics in terms of the (exposition only) member `m_iterator`.

9.29 indirect_iterator: unclear specification of template constructor

Submitter: Pete Becker
Status: New

The templated constructor that takes an indirect_iterator with a different set of template arguments says that it returns "An instance of indirect_iterator that is a copy of [the argument]". But the type of the argument is different from the type of the object being constructed, and there is no description of what a "copy" means. The Iterator template parameter for the argument must be convertible to the Iterator template parameter for the type being constructed, which suggests that the argument's contained Iterator object should be converted to the target type's Iterator type. Is that what's meant here?

(Pete later writes: In fact, this problem is present in all of the specialized adaptors that have a constructor like this: the constructor returns "a copy" of the argument without saying what a copy is.)

Proposed resolution:

Change:

Returns:
An instance of `indirect_iterator` that is a copy of `y`.
to:

Returns:
An instance of `indirect_iterator` whose `m_iterator` subobject is constructed from `y.base()`.

Rationale:
Inheritance from iterator_adaptor has been removed, so we instead give the semantics in terms of the member `m_iterator`.

9.30 transform_iterator argument irregularity

Submitter: Pete Becker
Status: New

The specialized adaptors that take both a Value and a Reference template argument all take them in that order, i.e. Value precedes Reference in the template argument list, with the exception of transform_iterator, where Reference precedes Value. This seems like a possible source of confusion. Is there a reason why this order is preferable?

Comment from proposal authors:
This was deliberate. defaults for Value depend on Reference. A sensible Value can almost always be computed from Reference. The first parameter is UnaryFunction, so the argument order is already different from the other adapters.

9.31 function_output_iterator overconstrained

Submitter: Pete Becker
Status: New

function_output_iterator requirements says: "The UnaryFunction must be Assignable, Copy Constructible, and the expression f(x) must be valid, where f is an object of type UnaryFunction and x is an object of a type accepted by f."

Everything starting with "and," somewhat reworded, is actually a constraint on output_proxy::operator=. All that's needed to create a function_output_iterator object is that the UnaryFunction type be Assignable and CopyConstructible. That's also sufficient to dereference and to increment such an object. It's only when you try to assign through a dereferenced iterator that f(x) has to work, and then only for the particular function object that the iterator holds and for the particular value that is being assigned.

Proposed resolution:
Change:
```
output_proxy operator*();
```
to:
```
/* see below */ operator*();
```

After `function_output_iterator& operator++(int);` add:
```
private:
  UnaryFunction m_f;     // exposition only
```

Change:
The `UnaryFunction` must be Assignable, Copy Constructible, and the expression `f(x)` must be valid, where `f` is an object of type `UnaryFunction` and `x` is an object of a type accepted by `f` . The resulting `function_output_iterator` is a model of the Writable and Incrementable Iterator concepts.
to:
`UnaryFunction` must be Assignable and Copy Constructible.

After the requirements section, add:

function_output_iterator models
`function_output_iterator` is a model of the Writable and Incrementable Iterator concepts.
Change:

Returns:
An instance of `function_output_iterator` with `f` stored as a data member.
to:

Effects:
Constructs an instance of `function_output_iterator` with `m_f` constructed from `f`.
Change:

```
output_proxy operator*();
```

Returns:
An instance of `output_proxy` constructed with a copy of the unary function `f` .
to:

```
operator*();
```

Returns:
An object `r` of unspecified type such that `r = t` is equivalent to `m_f(t)` for all `t`.
Remove:

`function_output_iterator::output_proxy` operations

```
output_proxy(UnaryFunction& f);
```

Returns:
An instance of `output_proxy` with `f` stored as a data member.

```
template <class T> output_proxy& operator=(const T& value);
```

Effects:

```
m_f(value);
return *this;
```

Change:

```
explicit function_output_iterator(const UnaryFunction& f =
UnaryFunction());
```

to:

```
explicit function_output_iterator();

explicit function_output_iterator(const UnaryFunction& f);
```

9.32 Should output_proxy really be a named type?

Submitter: Pete Becker
Status: New

This means someone can store an output_proxy object for later use, whatever that means. It also constrains output_proxy to hold a copy of the function object, rather than a pointer to the iterator object. Is all this mechanism really necessary?

Proposed resolution:
If the proposed resolution to issue 9.31 is accepted, it also addresses this issue.

9.33 istreambuf_iterator isn't a Readable Iterator

Submitter: Pete Becker
Status: New

c++std-lib-12333:
N1550 requires that for a Readable Iterator a of type X, *a returns an object of type iterator_traits<X>::reference. istreambuf_iterator::operator* returns charT, but istreambuf_iterator::reference is charT&. So am I overlooking something, or is istreambuf_iterator not Readable

Proposed resolution:
Remove all constraints on `iterator_traits<X>::reference` in Readable Iterator and Lvalue Iterator. Change Lvalue Iterator to refer to `T&` instead of `iterator_traits<X>::reference`.
Change:
A class or built-in type `X` models the *Readable Iterator* concept for the value type `T` if the

following expressions are valid and respect the stated semantics. `U` is the type of any specified member of type `T` .
to:
A class or built-in type `X` models the *Readable Iterator* concept for value type `T` if, in addition to `X` being Assignable and Copy Constructible, the following expressions are valid and respect the stated semantics. `U` is the type of any specified member of type `T`.

From the Input Iterator Requirements table, remove:

`iterator_traits<X>::reference`
Convertible to `iterator_traits<X>::value_type`

Change:

`*a`
`iterator_traits<X>::reference`
pre: `a` is dereferenceable. If `a == b` then `*a` is equivalent to `*b`

to:

`*a`
Convertible to `T`
pre: `a` is dereferenceable. If `a == b` then `*a`
is equivalent to `*b`.
Change:
The *Lvalue Iterator* concept adds the requirement that the `reference` type be a reference to the value type of the iterator.
to:
The *Lvalue Iterator* concept adds the requirement that the return type of `operator*` type be a reference to the value type of the iterator.

Change:

Lvalue Iterator Requirements

Expression
Return Type
Assertion

`iterator_traits<X>::reference`
`T&`
`T` is *cv* `iterator_traits<X>::value_type` where *cv* is an optional cv-qualification

to:

Lvalue Iterator Requirements

Expression
Return Type
Note/Assertion

`*a`

T&
T is *cv* `iterator_traits<X>::value_type` where *cv* is an optional cv-qualification.
pre: `a` is dereferenceable. If `a == b` then `*a` is equivalent to `*b` .

At the end of the section reverse_iterator models, add: The type `iterator_traits<Iterator>::reference` must be the type of `*i`, where `i` is an object of type `Iterator`.

Rationale:

Ideally there should be requirements on the reference type, however, since Readable Iterator is suppose to correspond to the current standard iterator requirements (which do not place requirements on the reference type) we will leave them off for now. There is a DR in process with respect to the reference type in the stadard iterator requirements. Once that is resolved we will revisit this issue for Readable Iterator and Lvalue Iterator.

We added Assignable to the requirements for Readable Iterator. This is needed to have Readable Iterator coincide with the capabilities of Input Iterator.

9.34 iterator_facade free functions unspecified

Submitter: Pete Becker
Status: New

c++std-lib-12562:
The template functions operator==, operator!=, operator<, operator<=, operator>, operator>=, and operator- that take two arguments that are specializations of iterator_facade have no specification. The template function operator+ that takes an argument that is a specialization of iterator_facade and an argument of type difference_type has no specification.

Proposed resolution:
Add the missing specifications.

```
template <class Dr, class V, class TC, class R, class D>
Derived operator+ (iterator_facade<Dr,V,TC,R,D> const&,
                   typename Derived::difference_type n);

template <class Dr, class V, class TC, class R, class D>
Derived operator+ (typename Derived::difference_type n,
                   iterator_facade<Dr,V,TC,R,D> const&);
```

Effects:

```
Derived tmp(static_cast<Derived const*>(this));
return tmp += n;

template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator ==(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `lhs.equal(rhs)`. Otherwise, `rhs.equal(lhs)` .

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator !=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `!lhs.equal(rhs)`. Otherwise, `!rhs.equal(lhs)` .

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator <(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `lhs.distance_to(rhs) < 0`. Otherwise, `rhs.distance_to(lhs) > 0`.

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator <=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `lhs.distance_to(rhs) <= 0`. Otherwise, `rhs.distance_to(lhs) >= 0`.

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator >(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `lhs.distance_to(rhs) > 0`. Otherwise, `rhs.distance_to(lhs) < 0`.

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator >=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `lhs.distance_to(rhs) >= 0`. Otherwise, `rhs.distance_to(lhs) <= 0`.

```
template <class Dr1, class V1, class TC1, class R1, class D1,
          class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,difference>::type
operator -(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);
```

Return Type:
if `is_convertible<Dr2,Dr1>::value`, then `difference` shall be `iterator_traits<Dr1>::difference_type`. Otherwise, `difference` shall be `iterator_traits<Dr2>::difference_type`.

Returns:
if `is_convertible<Dr2,Dr1>::value`, then `-lhs.distance_to(rhs)`. Otherwise, `rhs.distance_to(lhs)`.

9.35 iterator_facade: too many equals?

Submitter: Pete Becker
Status: New

c++std-lib-12563:
The table listing the functions required for types derived from iterator_facade has two functions named equal and two named distance_to:

```
c.equal(b)
c.equal(y)

c.distance_to(b)
c.distance_to(z)
```

where b and c are const objects of the derived type, y and z are constant objects of certain iterator types that are interoperable with the derived type.

Seems like the 'b' versions are redundant: in both cases, the other version will take a 'b'. In fact, iterator_adaptor is specified to use iterator_facade, but does not provide the 'b' versions of these functions.

Are the 'b' versions needed?

Proposed resolution:
Remove the 'b' versions.

In `iterator_facade` requirements, remove:

`c.equal(b)`
convertible to bool
true iff `b` and `c` are equivalent.
Single Pass Iterator

and remove:

`c.distance_to(b)`
convertible to X::difference_type
equivalent to `distance(c, b)`
Random Access Traversal Iterator

9.36 iterator_facade function requirements

Submitter: Pete Becker
Status: New

c++std-lib-12636:
The table that lists required functions for the derived type X passed to iterator_facade lists, among others:

for a single pass iterator:
c.equal(b)
c.equal(y)

where b and c are const X objects, and y is a const object of a single pass iterator that is interoperable with X. Since X is interoperable with itself, c.equal(b) is redundant. There is a difference in their descriptions, but its meaning isn't clear. The first is "true iff b and c are equivalent", and the second is "true iff c and y refer to the same position." Is there a difference between the undefined term "equivalent" and "refer to the same position"?

Similarly, for a random access traversal iterator:
c.distance_to(b)
c.distance_to(z)

where z is a constant object of a random access traversal iterator that is interoperable with X. Again, X is interoperable with itself, so c.distance_to(b) is redundant.

Also, the specification for c.distance_to(z) isn't valid. It's written as "equivalent to distance(c, z)". The template function distance takes two arguments of the same type, so distance(c, z) isn't valid if c and z are different types. Should it be distance(c, (X)z)?

9.37 iterator_adaptor Derived argument underspecified

Submitter: Pete Becker
Status: New

The Derived argument seems to be underspecified. Same problem as described in issue 9.21 for iterator_facade.

9.38 iterator_adaptor: "Base" is a confusing name

Submitter: Pete Becker
Status: New

The name Base for the iterator that's being adapted (and in the member functions base() and base_reference()) is confusing, since it's not a base in the sense that the term is used in C++.

Proposed Resolution:
The templates indirect_iterator and reverse_iterator both name their iterator argument Iterator. We should do the same here.

9.39 iterator_adaptor should get category from iterator_facade

Submitter: Pete Becker
Status: New

The clause entitled "iterator_adaptor requirements" talks about iterator_traits<Derived>::iterator_category. The base iterator_facade defines iterator_category, so it would seem more natural to simply use that. Unless, of course, Derived is permitted to provide its own definition of iterator_category which is different from the one in the base, or that iterator_traits<Derived> can be specialized to provide a different one. That doesn't seem reasonable, since the type in the base is determined by the Access and Traversal arguments that the user passed to iterator_adaptor. Why would the user want to define it differently?

9.40 iterator_adaptor should specify arguments that are passed to iterator_facade

Submitter: Pete Becker
Status: New

The clause entitled "iterator_adaptor requirements" sets out requirements in terms of the typedefs defined in iterator_facade. It would be clearer to specify the arguments that should be passed to iterator_façade.

Proposed Resolution:

Value argument to iterator_facade:

```
if (Value != use_default)
    Value
else
    iterator_traits<Base>::value_type
```

(But note that the default here is slightly different from the default specified in the paper. The latter can't be implemented correctly with an argument to iterator_facade, since iterator_traits<Base>::value_type might be cv-qualified, and iterator_facade strips the cv-qualifier. The approach I've given strips the cv-qualifier, too. In order to implement what the paper says, iterator_adaptor would have to provide its own version of value_type.)

AccessCategory argument to iterator_facade:

```
if (Access != use_default)
    Access
```

else if (is_const<Value>)
    access_category<Base>::value & ~writable_iterator
else
    access_category<Base>::value

This assumes (as does the paper) that there is a suitable definition of access_category somewhere (N1550 doesn't specify it).

TraversalCategory argument to iterator_facade:

if (Traversal != use_default)
    Traversal
else
    traversal_category<Base>::type

This assumes (as does the paper) that there is a suitable definition of traversal_category somewhere (N1550 doesn't specify it).

iterator_category is redundant and should be removed.

Reference argument to iterator_facade:

if (Reference != use_default)
    Reference
else if (Value != use_default)
    Value&
else
    iterator_traits<Base>::reference

The Difference argument to iterator_facade isn't specified here. Needs to be added. By analogy, should it be this?

if (Difference != use_default)
    Difference
else
    iterator_traits<Base>::difference_type

9.41 Inheritance in iterator_adaptor and other adaptors is an overspecification

Submitter: Pete Becker
Status: New

The paper requires that iterator_adaptor be derived from an appropriate instance of iterator_facade, and that most of the specific forms of adaptors be derived from appropriate instances of iterator_adaptor. That seems like overspecification, and we ought to look at specifying these things in terms of what the various templates provide rather than how they're implemented.

Proposed resolution:

Remove the specfication of inheritance, and add explicit specification of all the functionality that was inherited from the specialized iterators. (But retain inheritance in iterator_adaptor.)

In n1550, after [lib.random.access.traversal.iterators], add:

Interoperable Iterators [lib.interoperable.iterators]

A class or built-in type `X` that models Single Pass Iterator is *interoperable with* a class or built-in type `Y` that also models Single Pass Iterator if the following expressions are valid and respect the stated semantics. In the tables below, `x` is an object of type `X`, `y` is an object of type `Y`, `Distance` is `iterator_traits<Y>::difference_type` , and `n` represents a constant object of type `Distance`.

Expression
Return Type
Assertion/Precondition/Postcondition

`y = x`
`Y`
post: `y == x`

`Y(x)`
`Y`
post: `Y(x) == x`

`x == y`
convertible to `bool`
`==` is an equivalence relation over its domain.

`y == x`
convertible to `bool`
`==` is an equivalence relation over its domain.

`x != y`
convertible to `bool`
`bool(a==b) != bool(a!=b)` over its domain.

`y != x`
convertible to `bool`
`bool(a==b) != bool(a!=b)` over its domain.

If `X` and `Y` both model Random Access Traversal Iterator then the following additional requirements must be met.

Expression
Return Type
Operational Semantics
Assertion/ Precondition

`x < y`
convertible to `bool`
`y - x > 0`
< is a total ordering relation

`y < x`
convertible to `bool`
`x - y > 0`
< is a total ordering relation

`x > y`
convertible to `bool`
`y < x`
> is a total ordering relation

`y > x`
convertible to `bool`
`x < y`
> is a total ordering relation

`x >= y`
convertible to `bool`
`!(x < y)`

`y >= x`
convertible to `bool`
`!(y < x)`

`x <= y`
convertible to `bool`
`!(x > y)`

`y <= x`
convertible to `bool`
`!(y > x)`

`y - x`
`Distance`
`distance(Y(x),y)`
pre: there exists a value `n` of `Distance` such that `x + n == y`. `y == x + (y - x)`.

`x - y`
`Distance`
`distance(y,Y(x))`
pre: there exists a value `n` of `Distance` such that `y + n == x`. `x == y + (x - y)`.

In N1530:

In [lib.iterator.adaptor]

Change:
`class iterator_adaptor`

```
  : public iterator_facade<Derived, /* see details ...*/>
```

To:

```
class iterator_adaptor
  : public iterator_facade<Derived, *V'*, *C'*, *R'*, *D'*> //
see details
```

Change the text from:
The `Base` type must implement the expressions involving `m_iterator` in the specifications... until the end of the **iterator_adaptor requirements** section, to:
The `Base` argument shall be Assignable and Copy Constructible.

Add:

iterator_adaptor base class parameters

The *V'*, *C'*, *R'*, and *D'* parameters of the `iterator_facade` used as a base class in the summary of `iterator_adaptor` above are defined as follows:

```
V' = if (Value is use_default)
          return iterator_traits<Base>::value_type
      else
          return Value

C' = if (CategoryOrTraversal is use_default)
          return iterator_traversal<Base>::type
      else
          return CategoryOrTraversal

R' = if (Reference is use_default)
          if (Value is use_default)
              return iterator_traits<Base>::reference
          else
              return Value&
      else
          return Reference

D' = if (Difference is use_default)
          return iterator_traits<Base>::difference_type
      else
          return Difference
```

In [lib.iterator.special.adaptors]

Change:

```
class indirect_iterator
  : public iterator_adaptor</* see discussion */>
{
    friend class iterator_core_access;
```

to:

```
class indirect_iterator
{
 public:
    typedef /* see below */ value_type;
    typedef /* see below */ reference;
    typedef /* see below */ pointer;
    typedef /* see below */ difference_type;
    typedef /* see below */ iterator_category;
```

Change:

```
private: // as-if specification
    typename indirect_iterator::reference dereference() const
    {
        return **this->base();
    }
```

to:

```
    Iterator const& base() const;
    reference operator*() const;
    indirect_iterator& operator++();
    indirect_iterator& operator--();
private:
   Iterator m_iterator; // exposition
```

After the synopsis add:

The member types of `indirect_iterator` are defined according to the following pseudo-code, where `V` is `iterator_traits<Iterator>::value_type`

```
if (Value is use_default) then
    typedef remove_const<pointee<V>::type>::type value_type;
else
    typedef remove_const<Value>::type value_type;

if (Reference is use_default) then
    if (Value is use_default) then
        typedef indirect_reference<V>::type reference;
    else
        typedef Value& reference;
else
    typedef Reference reference;

if (Value is use_default) then
    typedef pointee<V>::type* pointer;
else
    typedef Value* pointer;
```

```
if (Difference is use_default)
    typedef iterator_traits<Iterator>::difference_type
difference_type;
else
    typedef Difference difference_type;

if (CategoryOrTraversal is use_default)
    typedef iterator-category(

iterator_traversal<Iterator>::type,``reference``,``value_type``
    ) iterator_category;
else
    typedef iterator-category(
        CategoryOrTraversal,``reference``,``value_type``
    ) iterator_category;
```

[Note: See resolution to 9.44y for a description of `pointee` and `indirect_reference`]

After the requirements section, add:

indirect_iterator models

In addition to the concepts indicated by `iterator_category` and by `iterator_traversal<indirect_iterator>::type` , a specialization of `indirect_iterator` models the following concepts, Where `v` is an object of `iterator_traits<Iterator>::value_type` :

- Readable Iterator if `reference(*v)` is convertible to `value_type` .
- Writable Iterator if `reference(*v) = t` is a valid expression (where `t` is an object of type `indirect_iterator::value_type`)
- Lvalue Iterator if `reference` is a reference type.

`indirect_iterator<X,V1,C1,R1,D1>` is interoperable with `indirect_iterator<Y,V2,C2,R2,D2>` if and only if `X` is interoperable with `Y`.

Before `indirect_iterator();` add:
In addition to the operations required by the concepts described above, specializations of `indirect_iterator` provide the following operations.
Change:

Returns:
An instance of `indirect_iterator` with the `iterator_adaptor` subobject copy constructed from `x`.
to:

Returns:
An instance of `indirect_iterator` with `m_iterator` copy constructed from `x`.

At the end of the indirect_iterator operations add:

```
Iterator const& base() const;
```

Returns:
`m_iterator`

```
reference operator*() const;
```

Returns:
`**m_iterator`

```
indirect_iterator& operator++();
```

Effects:
`++m_iterator`

Returns:
`*this`

```
indirect_iterator& operator--();
```

Effects:
`--m_iterator`

Returns:
`*this`

Change:
```
template <class Iterator>
class reverse_iterator :
  public iterator_adaptor< reverse_iterator<Iterator>, Iterator
>
{
  friend class iterator_core_access;
```

to:
```
template <class Iterator>
class reverse_iterator
{
public:
  typedef iterator_traits<Iterator>::value_type value_type;
  typedef iterator_traits<Iterator>::reference reference;
  typedef iterator_traits<Iterator>::pointer pointer;
  typedef iterator_traits<Iterator>::difference_type
difference_type;
  typedef /* see below */ iterator_category;
```

Change:
```
private: // as-if specification
  typename reverse_iterator::reference dereference() const {
return *prior(this->base()); }
```

```
  void increment() { --this->base_reference(); }
  void decrement() { ++this->base_reference(); }

  void advance(typename reverse_iterator::difference_type n)
  {
      this->base_reference() += -n;
  }

  template <class OtherIterator>
  typename reverse_iterator::difference_type
  distance_to(reverse_iterator<OtherIterator> const& y) const
  {
      return this->base_reference() - y.base();
  }
```

to:

```
  Iterator const& base() const;
  reference operator*() const;
  reverse_iterator& operator++();
  reverse_iterator& operator--();
private:
  Iterator m_iterator; // exposition
```

After the synopsis for `reverse_iterator`, add:
If `Iterator` models Random Access Traversal Iterator and Readable Lvalue Iterator, then `iterator_category` is convertible to `random_access_iterator_tag`. Otherwise, if `Iterator` models Bidirectional Traversal Iterator and Readable Lvalue Iterator, then `iterator_category` is convertible to `bidirectional_iterator_tag`. Otherwise, `iterator_category` is convertible to `input_iterator_tag`.
Change:

reverse_iterator requirements

The base `Iterator` must be a model of Bidirectional Traversal Iterator. The resulting `reverse_iterator` will be a model of the most refined standard traversal and access concepts that are modeled by `Iterator`.
to:

reverse_iterator requirements

`Iterator` must be a model of Bidirectional Traversal Iterator.

reverse_iterator models

A specialization of `reverse_iterator` models the same iterator traversal and iterator access concepts modeled by its `Iterator` argument. In addition, it may model old iterator concepts specified in the following table:

If I models
then reverse_iterator<I> models

Readable Lvalue Iterator, Bidirectional Traversal Iterator
Bidirectional Iterator

Writable Lvalue Iterator, Bidirectional Traversal Iterator
Mutable Bidirectional Iterator

Readable Lvalue Iterator, Random Access Traversal Iterator
Random Access Iterator

Writable Lvalue Iterator, Random Access Traversal Iterator
Mutable Random Access Iterator

reverse_iterator<X> is interoperable with reverse_iterator<Y> if and only if X is interoperable with Y.
Change:

Returns:
An instance of reverse_iterator with a default constructed base object.
to:

Effects:
Constructs an instance of reverse_iterator with m_iterator default constructed.
Change:

Effects:
Constructs an instance of reverse_iterator with a base object copy constructed from x.
to:

Effects:
Constructs an instance of reverse_iterator with a m_iterator constructed from x.
Change:

Returns:
An instance of reverse_iterator that is a copy of r.
to:

Effects:
Constructs instance of reverse_iterator whose m_iterator subobject is constructed from y.base().
At the end of the operations for reverse_iterator, add:

```
Iterator const& base() const;
```

Returns:
m_iterator

```
reference operator*() const;
```

Effects:
```
Iterator tmp = m_iterator;
return *--tmp;
```

```
reverse_iterator& operator++();
```

Effects:
```
--m_iterator
```

Returns:
```
*this
```

```
reverse_iterator& operator--();
```

Effects:
```
++m_iterator
```

Returns:
```
*this
```

Change:
```
class transform_iterator
  : public iterator_adaptor</* see discussion */>
{
  friend class iterator_core_access;
```

to:
```
class transform_iterator
{
public:
  typedef /* see below */ value_type;
  typedef /* see below */ reference;
  typedef /* see below */ pointer;
  typedef iterator_traits<Iterator>::difference_type
difference_type;
  typedef /* see below */ iterator_category;
```

After `UnaryFunction functor() const;` add:
```
Iterator const& base() const;
reference operator*() const;
transform_iterator& operator++();
transform_iterator& operator--();
```

Change:
```
private:
  typename transform_iterator::value_type dereference() const;
  UnaryFunction m_f;
};
```

to:
```
private:
```

```
  Iterator m_iterator; // exposition only
  UnaryFunction m_f;   // exposition only
};
```

After the synopsis, add:
If `Iterator` models Readable Lvalue Iterator and if `Iterator` models Random Access Traversal Iterator, then `iterator_category` is convertible to `random_access_iterator_tag` . Otherwise, if `Iterator` models Bidirectional Traversal Iterator, then `iterator_category` is convertible to `bidirectional_iterator_tag`. Otherwise `iterator_category` is convertible to `forward_iterator_tag`. If `Iterator` does not model Readable Lvalue Iterator then `iterator_category` is convertible to `input_iterator_tag` .
In the requirements section, change:

The type `Iterator` must at least model Readable Iterator. The resulting `transform_iterator` models the most refined of the following that is also modeled by `Iterator`.

- Writable Lvalue Iterator if `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type` is a non-const reference.
- Readable Lvalue Iterator if `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type` is a const reference.
- Readable Iterator otherwise.

The `transform_iterator` models the most refined standard traversal concept that is modeled by `Iterator`.

The `reference` type of `transform_iterator` is `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type` . The `value_type` is `remove_cv<remove_reference<reference> >::type`.
to:
The argument `Iterator` shall model Readable Iterator.

After the requirements section, add:

transform_iterator models

The resulting `transform_iterator` models the most refined of the following options that is also modeled by `Iterator`.

- Writable Lvalue Iterator if `transform_iterator::reference` is a non-const reference.
- Readable Lvalue Iterator if `transform_iterator::reference` is a const reference.
- Readable Iterator otherwise.

The `transform_iterator` models the most refined standard traversal concept that is modeled by the `Iterator` argument.

If `transform_iterator` is a model of Readable Lvalue Iterator then it models the following original iterator concepts depending on what the `Iterator` argument models.

If Iterator models
then transform_iterator models

Single Pass Iterator
Input Iterator

Forward Traversal Iterator
Forward Iterator

Bidirectional Traversal Iterator
Bidirectional Iterator

Random Access Traversal Iterator
Random Access Iterator

If `transform_iterator` models Writable Lvalue Iterator then it is a mutable iterator (as defined in the old iterator requirements).

`transform_iterator<F1, X, R1, V1>` is interoperable with `transform_iterator<F2, Y, R2, V2>` if and only if `X` is interoperable with `Y`.

Remove the private operations section heading and remove:

```
``typename transform_iterator::value_type dereference() const;``

:Returns: ``m_f(transform_iterator::dereference());``
```

After the entry for `functor()`, add:

```
``Iterator const& base() const;``

:Returns: ``m_iterator``


``reference operator*() const;``

:Returns: ``m_f(*m_iterator)``


``transform_iterator& operator++();``

:Effects: ``++m_iterator``
:Returns: ``*this``


``transform_iterator& operator--();``

:Effects: ``--m_iterator``
:Returns: ``*this``
```

Change:

```
template <class Predicate, class Iterator>
```

```
class filter_iterator
   : public iterator_adaptor<
         filter_iterator<Predicate, Iterator>, Iterator
       , use_default
       , /* see details */
     >
{
 public:
```

to:

```
template <class Predicate, class Iterator>
class filter_iterator
{
 public:
   typedef iterator_traits<Iterator>::value_type value_type;
   typedef iterator_traits<Iterator>::reference reference;
   typedef iterator_traits<Iterator>::pointer pointer;
   typedef iterator_traits<Iterator>::difference_type
difference_type;
   typedef /* see below */ iterator_category;
```

Change:

```
private: // as-if specification
   void increment()
   {
       ++(this->base_reference());
       satisfy_predicate();
   }

   void satisfy_predicate()
   {
       while (this->base() != this->m_end && !this-
>m_predicate(*this->base()))
           ++(this->base_reference());
   }

   Predicate m_predicate;
   Iterator m_end;
```

to:

```
    Iterator const& base() const;
    reference operator*() const;
    filter_iterator& operator++();
private:
    Predicate m_pred; // exposition only
    Iterator m_iter;  // exposition only
    Iterator m_end;   // exposition only
```

Change:
The base `Iterator` parameter must be a model of Readable Iterator and Single Pass Iterator. The resulting `filter_iterator` will be a model of Forward Traversal Iterator if `Iterator` is, otherwise the `filter_iterator` will be a model of Single Pass Iterator. The access category of the `filter_iterator` will be the same as the access category of `Iterator`.
to:
The `Iterator` argument shall meet the requirements of Readable Iterator and Single Pass Iterator or it shall meet the requirements of Input Iterator.

After the requirements section, add:

filter_iterator models

The concepts that `filter_iterator` models are dependent on which concepts the `Iterator` argument models, as specified in the following tables.

If Iterator models	then filter_iterator models
Single Pass Iterator	Single Pass Iterator
Forward Traversal Iterator	Forward Traversal Iterator

If Iterator models	then filter_iterator models
Readable Iterator	Readable Iterator
Writable Iterator	Writable Iterator
Lvalue Iterator	Lvalue Iterator

If Iterator models	then filter_iterator models
Readable Iterator, Single Pass Iterator	Input Iterator
Readable Lvalue Iterator, Forward Traversal Iterator	Forward Iterator
Writable Lvalue Iterator, Forward Traversal Iterator	Mutable Forward Iterator

`filter_iterator<P1, X>` is interoperable with `filter_iterator<P2, Y>` if and only if `X` is interoperable with `Y`.
Change:

Returns:

a `filter_iterator` whose predicate is a default constructed `Predicate` and whose `end` is a default constructed `Iterator`.
to:

Effects:
Constructs a `filter_iterator` whose``m_pred``, `m_iter` , and `m_end` members are a default constructed.
Change:

Returns:
A `filter_iterator` at position `x` that filters according to predicate `f` and that will not increment past `end`.
to:

Effects:
Constructs a `filter_iterator` where `m_iter` is either the first position in the range `[x,end)` such that `f(*m_iter) == true` or else``m_iter == end``. The member `m_pred` is constructed from `f` and `m_end` from `end`.
Change:

Returns:
A `filter_iterator` at position `x` that filters according to a default constructed `Predicate` and that will not increment past `end`.
to:

Effects:
Constructs a `filter_iterator` where `m_iter` is either the first position in the range `[x,end)` such that `m_pred(*m_iter) == true` or else``m_iter == end``. The member `m_pred` is default constructed.
Change:

Returns:
A copy of iterator `t`.
to:

Effects:
Constructs a filter iterator whose members are copied from `t`.
Change:

Returns:
A copy of the predicate object used to construct `*this`.
to:

Returns:
`m_pred`
Change:

Returns:
The object `end` used to construct `*this`.
to:

Returns:
`m_end`

At the end of the operations section, add:

```
reference operator*() const;
```

Returns:
`*m_iter`

```
filter_iterator& operator++();
```

Effects:
Increments `m_iter` and then continues to increment `m_iter` until either `m_iter == m_end` or `m_pred(*m_iter) == true`.

Returns:
`*this`

Change:

```
class counting_iterator
  : public iterator_adaptor<
        counting_iterator<Incrementable, Access, Traversal,
Difference>
      , Incrementable
      , Incrementable
      , Access
      , /* see details for traversal category */
      , Incrementable const&
      , Incrementable const*
      , /* distance = Difference or a signed integral type */>
{
    friend class iterator_core_access;
 public:
```

to:

```
class counting_iterator
{
 public:
    typedef Incrementable value_type;
    typedef const Incrementable& reference;
    typedef const Incrementable* pointer;
    typedef /* see below */ difference_type;
    typedef /* see below */ iterator_category;
```

Change:

```
private:
    typename counting_iterator::reference dereference() const
    {
        return this->base_reference();
    }
```

to:

```
    Incrementable const& base() const;
    reference operator*() const;
    counting_iterator& operator++();
    counting_iterator& operator--();
private:
    Incrementable m_inc; // exposition
```

After the synopsis, add:

If the `Difference` argument is `use_default` then `difference_type` is an unspecified signed integral type. Otherwise `difference_type` is `Difference`.

`iterator_category` is determined according to the following algorithm:

```
if (CategoryOrTraversal is not use_default)
    return CategoryOrTraversal
else if (numeric_limits<Incrementable>::is_specialized)
    return iterator-category(
        random_access_traversal_tag, Incrementable, const
Incrementable&)
else
    return iterator-category(
         iterator_traversal<Incrementable>::type,
         Incrementable, const Incrementable&)
```

Change:
[*Note:* implementers are encouraged to provide an implementation of `distance_to` and a `difference_type` that avoids overflows in the cases when the `Incrementable` type is a numeric type.]
to:
[*Note:* implementers are encouraged to provide an implementation of `operator-` and a `difference_type` that avoid overflows in the cases where `std::numeric_limits<Incrementable>::is_specialized` is true.]
Change:

The `Incrementable` type must be Default Constructible, Copy Constructible, and Assignable. The default distance is an implementation defined signed integegral type.

The resulting `counting_iterator` models Readable Lvalue Iterator.
to:
The `Incrementable` argument shall be Copy Constructible and Assignable.
Change:
Furthermore, if you wish to create a counting iterator that is a Forward Traversal Iterator, then the following expressions must be valid:
to:
If `iterator_category` is convertible to `forward_iterator_tag` or `forward_traversal_tag` , the following must be well-formed:
Change:
If you wish to create a counting iterator that is a Bidirectional Traversal Iterator, then pre-

decrement is also required:
to:
If `iterator_category` is convertible to `bidirectional_iterator_tag` or `bidirectional_traversal_tag` , the following expression must also be well-formed:
Change:
If you wish to create a counting iterator that is a Random Access Traversal Iterator, then these additional expressions are also required:
to:
If `iterator_category` is convertible to `random_access_iterator_tag` or `random_access_traversal_tag` , the following must must also be valid:

After the requirements section, add:

counting_iterator models

Specializations of `counting_iterator` model Readable Lvalue Iterator. In addition, they model the concepts corresponding to the iterator tags to which their `iterator_category` is convertible. Also, if `CategoryOrTraversal` is not `use_default` then `counting_iterator` models the concept corresponding to the iterator tag `CategoryOrTraversal`. Otherwise, if `numeric_limits<Incrementable>::is_specialized`, then `counting_iterator` models Random Access Traversal Iterator. Otherwise, `counting_iterator` models the same iterator traversal concepts modeled by `Incrementable`.

`counting_iterator<X,C1,D1>` is interoperable with `counting_iterator<Y,C2,D2>` if and only if `X` is interoperable with `Y`.

At the begining of the operations section, add:
In addition to the operations required by the concepts modeled by `counting_iterator` , `counting_iterator` provides the following operations.
Change:

Returns:
A default constructed instance of `counting_iterator`.
to:

Requires:
`Incrementable` is Default Constructible.

Effects:
Default construct the member `m_inc`.
Change:

Returns:
An instance of `counting_iterator` that is a copy of `rhs`.
to:

Effects:
Construct member `m_inc` from `rhs.m_inc`.
Change:

Returns:

An instance of `counting_iterator` with its base object copy constructed from `x`.
to:

Effects:
Construct member `m_inc` from `x`.

At the end of the operations section, add:

```
reference operator*() const;
```

Returns:
`m_inc`

```
counting_iterator& operator++();
```

Effects:
`++m_inc`

Returns:
`*this`

```
counting_iterator& operator--();
```

Effects:
`--m_inc`

Returns:
`*this`

```
Incrementable const& base() const;
```

Returns:
`m_inc`

9.42 Problem with specification of a->m in Readable Iterator

Submitter: Howard Hinnant
Status: New

Readable Iterator Requirements says:

`a->m`
`U&`
pre: `(*a).m` is well-defined. Equivalent to `(*a).m`

Do we mean to outlaw iterators with proxy references from meeting the readable requirements?

Would it be better for the requirements to read `static_cast<T>(*a).m` instead of `(*a).m` ?

Comments from proposal authors:

NAD. If `(*a).m` is not well defined, then the iterator is not required to provide `a->m`. So a proxy iterator is not required to provide `a->m`.

9.43 counting_iterator Traversal argument unspecified

Submitter: Pete Becker
Status: New

counting_iterator takes an argument for its Traversal type, with a default value of use_default. It is derived from an instance of iterator_adaptor, where the argument passed for the Traversal type is described as "`/* see details for traversal category */`". The details for counting_iterator describe constraints on the Incrementable type imposed by various traversal categories. There is no description of what the argument to iterator_adaptor should be.

Proposed Resolution:
If all of the other proposed resolutions are accepted then this will no longer be an issue.

9.44 Indirect_iterator requirements muddled

Submitter: Pete Becker
Status: New

c++std-lib-12640:
The value_type of the Iterator template parameter should itself be dereferenceable. The return type of the `operator*` for the value_type must be the same type as the Reference template parameter.

I'd say this a bit differently, to emphasize what's required: iterator_traits<Iterator>::value_type must be dereferenceable. The Reference template parameter must be the same type as `*iterator_traits<Iterator>::value_type()` .
The Value template parameter will be the value_type for the indirect_iterator, unless Value is const. If Value is const X, then value_type will be non- const X.

Also non-volatile, right? In other words, if Value isn't use_default, it just gets passed as the Value argument for iterator_adaptor.

The default for Value is:

```
iterator_traits< iterator_traits<Iterator>::value_type
>::value_type
```

If the default is used for Value, then there must be a valid specialization of iterator_traits for the value type of the base iterator.

The earlier requirement is that `iterator_traits<Iterator>::value_type` must be dereferenceable. Now it's being treated as an iterator. Is this just a pun, or is `iterator_traits<Iterator>::value_type` required to be some form of iterator? If it's the former we need to find a different way to say it. If it's the latter we need to say so.

Proposed resolution:
Change:

The value_type of the Iterator template parameter should itself be dereferenceable. The return type of the operator* for the value_type must be the same type as the Reference template parameter. The Value template parameter will be the value_type for the indirect_iterator , unless Value is const. If Value is const X, then value_type will be *non-* const X. The default for Value is:
iterator_traits< iterator_traits<Iterator>::value_type >::value_type

If the default is used for Value, then there must be a valid specialization of iterator_traits for the value type of the base iterator.

The Reference parameter will be the reference type of the indirect_iterator. The default is Value&.

The Access and Traversal parameters are passed unchanged to the corresponding parameters of the iterator_adaptor base class, and the Iterator parameter is passed unchanged as the Base parameter to the iterator_adaptor base class.

to:
The expression *v, where v is an object of iterator_traits<Iterator>::value_type , shall be valid expression and convertible to reference. Iterator shall model the traversal concept indicated by iterator_category. Value , Reference, and Difference shall be chosen so that value_type, reference, and difference_type meet the requirements indicated by iterator_category.

[Note: there are further requirements on the iterator_traits<Iterator>::value_type if the Value parameter is not use_default, as implied by the algorithm for deducing the default for the value_type member.]

9.45 Problem with transform_iterator requirements

Submitter: Pete Becker
Status: New

c++std-lib-12641:
The reference type of transform_iterator is result_of< UnaryFunction(iterator_traits<Iterator>::reference) >::type. The value_type is remove_cv<remove_reference<reference> >::type.

These are the defaults, right? If the user supplies their own types that's what gets passed to iterator_adaptor. And again, the specification should be in terms of the specialization of iterator_adaptor, and not in terms of the result:

Reference argument to iterator_adaptor:

```
if (Reference != use_default)
    Reference
else
    result_of<
```

```
        UnaryFunction(iterator_traits<Iterator>::reference)
    >::type
```

Value argument to iterator_adaptor:

```
if (Value != use_default)
    Value
else if (Reference != use_default)
    remove_reference<reference>::type
else
    remove_reference<
        result_of<
            UnaryFunction(iterator_traits<Iterator>::reference)
        >::type
    >::type
```

There's probably a better way to specify that last alternative, but I've been at this too long, and it's all turning into a maze of twisty passages, all alike.

Proposed resolution:

Replace:
The reference type of transform_iterator is `result_of< UnaryFunction(iterator_traits<Iterator>::reference) >::type`. The `value_type` is `remove_cv<remove_reference<reference> >::type`.

with:

If `Reference` is `use_default` then the `reference` member of `transform_iterator` is `result_of< UnaryFunction(iterator_traits<Iterator>::reference) >::type`. Otherwise, `reference` is `Reference`.

If `Value` is `use_default` then the `value_type` member is `remove_cv<remove_reference<reference> >::type`. Otherwise, `value_type` is `Value`.

9.46 Filter_iterator details unspecified

Submitter: Pete Becker
Status: New

c++std-lib-12642:

The paper says:

```
template<class Predicate, class Iterator>
class filter_iterator
     : public iterator_adaptor<
         filter_iterator<Predicate, Iterator>,
         Iterator,
         use_default,
```

```
        /* see details */ >
```

That comment covers the Access, Traversal, Reference, and Difference arguments. The only specification for any of these in the details is:
The access category of the filter_iterator will be the same as the access category of Iterator.

Needs more.

Proposed resolution:

Add to the synopsis:

```
typedef iterator_traits<Iterator>::value_type value_type;
typedef iterator_traits<Iterator>::reference reference;
typedef iterator_traits<Iterator>::pointer pointer;
typedef iterator_traits<Iterator>::difference_type
difference_type;
typedef /* see below */ iterator_category;
```

and add just after the synopsis:
If `Iterator` models Readable Lvalue Iterator and Forward Traversal Iterator then `iterator_category` is convertible to `std::forward_iterator_tag`. Otherwise `iterator_category` is convertible to `std::input_iterator_tag`.

9.47 Transform_iterator interoperability too restrictive

Submitter: Jeremy Siek
Status: New

We do not need to require that the function objects have the same type, just that they be convertible.

Proposed resolution:

Change:

```
template<class OtherIterator, class R2, class V2>
transform_iterator(
      transform_iterator<UnaryFunction, OtherIterator, R2, V2>
const& t
    , typename enable_if_convertible<OtherIterator,
Iterator>::type* = 0 // exposition
);
```

to:

```
template<class F2, class I2, class R2, class V2>
transform_iterator(
      transform_iterator<F2, I2, R2, V2> const& t
    , typename enable_if_convertible<I2, Iterator>::type* = 0
// exposition only
```

```
    , typename enable_if_convertible<F2, UnaryFunction>::type* =
0 // exposition only
);
```

9.48 indirect_iterator and smart pointers

Submitter: Dave Abrahams
Status: New

`indirect_iterator` should be able to iterate over containers of smart pointers, but the mechanism that allows it was left out of the specification, even though it's present in the Boost specification

Proposed resolution:

Add `pointee` and `indirect_reference` to deal with this capability.

In [lib.iterator.helper.synopsis], add:

```
template <class Dereferenceable>
struct pointee;

template <class Dereferenceable>
struct indirect_reference;
```

After `indirect_iterator`'s abstract, add:

Class template pointee

```
template <class Dereferenceable>
struct pointee
{
    typedef /* see below */ type;
};
```

Requires:
For an object `x` of type `Dereferenceable`, `*x` is well-formed. If `++x` is ill-formed it shall neither be ambiguous nor shall it violate access control, and `Dereferenceable::element_type` shall be an accessible type. Otherwise `iterator_traits<Dereferenceable>::value_type` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `pointee`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable` :

```
if ( ++x is ill-formed )
{
    return ``Dereferenceable::element_type``
}
else if (``*x`` is a mutable reference to
         std::iterator_traits<Dereferenceable>::value_type)
```

```
{
    return iterator_traits<Dereferenceable>::value_type
}
else
{
    return iterator_traits<Dereferenceable>::value_type const
}
```

Class template indirect_reference

```
template <class Dereferenceable>
struct indirect_reference
{
    typedef /* see below */ type;
};
```

Requires:
For an object `x` of type `Dereferenceable`, `*x` is well-formed. If `++x` is ill-formed it shall neither be ambiguous nor shall it violate access control, and `pointee<Dereferenceable>::type&` shall be well-formed. Otherwise `iterator_traits<Dereferenceable>::reference` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `indirect_reference`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable` :

```
if ( ++x is ill-formed )
    return ``pointee<Dereferenceable>::type&``
else
    std::iterator_traits<Dereferenceable>::reference
```

9.49 Base() return by value is costly

Submitter: Dave Abrahams
Status: New

We've had some real-life reports that iterators that use `iterator_adaptor`'s `base()` function can be inefficient when the `Base` iterator is expensive to copy. Iterators, of all things, should be efficient.

Proposed resolution:

In [lib.iterator.adaptor]

Change:

```
Base base() const;
```

to:

```
Base const& base() const;
```

twice (once in the synopsis and once in the **public operations** section).

9.50 Default constructivel missing in Forward Traversal Iterator

Submitter: Jeremy Siek
Status: New

We want Forward Traversal Iterator plus Readable Lvalue Iterator to match the old Foward Iterator requirements, so we need Forward Traversal Iterator to include Default Constructible.

Proposed resolution:

Change:

A class or built-in type `X` models the *Forward Traversal Iterator* concept if the following expressions are valid and respect the stated semantics.

Forward Traversal Iterator Requirements (in addition to Single Pass Iterator)

to:

A class or built-in type `X` models the *Forward Traversal Iterator* concept if, in addition to `X` meeting the requirements of Default Constructible and Single Pass Iterator, the following expressions are valid and respect the stated semantics.

Forward Traversal Iterator Requirements (in addition to Default Constructible and Single Pass Iterator)

9.51 Iterator_facade requirements and type members unclear

Submitter: Dave Abrahams
Status: New

A general cleanup and simplification of the requirements and description of type members for `iterator_facade` .

The user is only allowed to add `const` as a qualifier.
Change:
```
typedef remove_cv<Value>::type value_type;
```
to:
```
typedef remove_const<Value>::type value_type;
```

We use to have an unspecified type for `pointer`, to match the return type of `operator->`, but there's no real reason to make them match, so we just use the simpler `Value*` for `pointer`.

Change:
```
typedef /* see description of operator-> */ pointer;
```
To:

`typedef Value* pointer;`
Remove:
Some of the constraints on template parameters to `iterator_facade` are expressed in terms of resulting nested types and should be viewed in the context of their impact on `iterator_traits<Derived>` .
Change:
The `Derived` template parameter must be a class derived from `iterator_facade`.
and:
The following table describes the other requirements on the `Derived` parameter. Depending on the resulting iterator's `iterator_category`, a subset of the expressions listed in the table are required to be valid. The operations in the first column must be accessible to member functions of class `iterator_core_access` .
to:
The following table describes the typical valid expressions on `iterator_facade`'s `Derived` parameter, depending on the iterator concept(s) it will model. The operations in the first column must be made accessible to member functions of class `iterator_core_access`. In addition, `static_cast<Derived*>(iterator_facade*)` shall be well-formed.
Remove:

The nested `::value_type` type will be the same as `remove_cv<Value>::type`, so the `Value` parameter must be an (optionally `const`-qualified) non-reference type.

The nested `::reference` will be the same as the `Reference` parameter; it must be a suitable reference type for the resulting iterator. The default for the `Reference` parameter is `Value&`.

Change:

In the table below, `X` is the derived iterator type, `a` is an object of type `X`, `b` and `c` are objects of type `const X`, `n` is an object of `X::difference_type`, `y` is a constant object of a single pass iterator type interoperable with X, and `z` is a constant object of a random access traversal iterator type interoperable with `X`.

Expression
Return Type
Assertion/Note
Required to implement Iterator Concept(s)

`c.dereference()`
`X::reference`

Readable Iterator, Writable Iterator

`c.equal(b)`
convertible to bool
true iff `b` and `c` are equivalent.
Single Pass Iterator

`c.equal(y)`
convertible to bool
true iff `c` and `y` refer to the same position. Implements `c == y` and `c != y`.
Single Pass Iterator

`a.advance(n)`

unused

Random Access Traversal Iterator

`a.increment()`
unused

Incrementable Iterator

`a.decrement()`
unused

Bidirectional Traversal Iterator

`c.distance_to(b)`
convertible to X::difference_type
equivalent to `distance(c, b)`
Random Access Traversal Iterator

`c.distance_to(z)`
convertible to X::difference_type
equivalent to `distance(c, z)`. Implements `c - z`, `c < z`, `c <= z`, `c > z`, and `c >= c`.
Random Access Traversal Iterator

to:

In the table below, `F` is `iterator_facade<X,V,C,R,D>`, `a` is an object of type `X`, `b` and `c` are objects of type `const X`, `n` is an object of `F::difference_type`, `y` is a constant object of a single pass iterator type interoperable with `X`, and `z` is a constant object of a random access traversal iterator type interoperable with `X`.

iterator_facade Core Operations

Expression
Return Type
Assertion/Note
Used to implement Iterator Concept(s)

`c.dereference()`
`F::reference`

Readable Iterator, Writable Iterator

`c.equal(y)`
convertible to bool
true iff `c` and `y` refer to the same position.
Single Pass Iterator

`a.increment()`
unused

Incrementable Iterator

`a.decrement()`
unused

Bidirectional Traversal Iterator

`a.advance(n)`
unused

Random Access Traversal Iterator

`c.distance_to(z)`
convertible to `F::difference_type`
equivalent to `distance(c, X(z))`.
Random Access Traversal Iterator

10 Function object and reference_wrapper issues

10.1 Return types of reference wrapper functions

Submitter: Alisdair Meredith
Status: New

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: New

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: New

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: New

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;
```

10.5 Callable definition does not match function<> semantics

Submitter: Doug Gregor
Status: New

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: New

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: New

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;
```

10.8 Class template function should have null pointer assignment/comparison operations.

Submitter: Doug Gregor
Status: New

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.

10.9 result_of template type parameter unrelated to description

Submitter: Doug Gregor
Status: New

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: New

`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, ..., Tn)>::type` with `result_of<R(T1&, T2&, ..., Tn&)>::type`.

10.11 result_of can not work for function and member function types

Submitter: Doug Gregor
Status: New

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 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: New

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 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: New

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.).

11 Tuple issues

11.1 Implementation limits: nonexistent Annex B

Submitter: Alisdair Meredith
Status: New

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:

The reference in [tr.tuple] to annex B should be changed to refer to [tr.tuple.lim].