```
Document Number: DxxxRy
Date:            2016-06-25
Reply-to:        Rein Halbersma <rhalbersma at gmail dot com>
Audience:        LEWG
```

Rough ideas for LFv3 TS / C++Next library additions

In rough order of work to be done.

1 Convenient underlying types for scoped enums

Addition to `<type_traits>`

```
template<class E>
constexpr auto to_underlying_type(E e) noexcept
{
        return static_cast<std::underlying_type_t<E>>(e);
}
```

Motivation: hide ugly `static_cast`, e.g. when accessing `tuple` members using named constansts (see Item 10 from Effective Modern C++).

Experience: I use this all over my projects. Code availabe on bitbucket.

2 `for_each` and `reverse_for_each` members for `bitset`

Addition to `<bitset>`

I am aware of the `bitset` iterator proposals, but there would still be room for an even more efficient processing of all 1-bits

```
std::bitset<N> b; /* init */
b.for_each([](auto i){ std::cout << i; }); // prints all 1-bit indices
```

Experience: I have implemented this, using bitscan intrinsics to efficiently iterate over all 1-bits. (I also have implemented bit-references and iterators, but this plain for-loop covers all my use cases and is more efficient to boot). Code available on bitbucket.

3 Three-way comparisons

For `string`

```
string a, b; /* init */
a.compare(b);
```

gives -1, 0, +1 depending on whether `a < b`, `a == b` or `a > b`, respectively.

This can be generalized to all builtin types as well as to all Standard Library types that already have `operator==` and `operator<`. Ideally, this is supplemented with a new algorithm

```
template<class Rng1, class Rng2>
bool threeway_compare(Rng1, Rng2);
```

that is more efficiently than a pair of calls to `lexicographical_compare` and `equal`.

Prior work: possibly discussed somewhere in the long list of the default comparisons proposals or in Crowl's ordering paper. Working code was presented at CppCon14 by Arthur O'Dwyer.

Experience: I have implemented this using O'Dwyer's code. Code available on BitBucket (not very robust yet, only tested for `tuple`).

Motivation: I'd like to have this for `tuple` in particular, but I think the general idea is potentially a very big proposal with lots of testing required. There are also CPU intrinsics on the way in new machines that might make this attractive for the SG14 crowd.

4 Floored and Euclidean versions of std::div

Addition to `<cstdlib>`

`std::div` does a truncated version of quotient and remainder. There are several other versions in the literature. See this Microsoft research paper for an extensive overview.

Experience: I have implemented floored and Euclidean versions of `std::div`. The latter has the especially nice property that the remainder is always non-negative (I use it to model an angle class that keeps a non-negative integer modulo 360). Code and extensive unit tests available on bitbucket.

Motivation: Since other `div` versions are in use in other programming languages, this is a never ending source of confusion and ensuing questions on StackOverflow (slight exaggeration, the differences only occur when using negative numbers).

5 A set-associative cache quasi-container

```
template<class Key, class T, size_t N, class Hash, class Replace>
 class set_associative_cache;
```

A wrapper over a `vector<array<pair<Key, T>, N>>` container, that allows hashed indexing from keys into `N`-way set-associative buckets. Typically, `sizeof(pair<Key, T>) * N` is chosen to let a bucket fit a small number (1 or 2) of cachelines.

In contrast to `std::unordered_map<Key, T, Hash>`, a cache does not reallocate but replaces elements when a bucket becomes full. Resizing on request is possible.

No iterators over elements (but a `size()` member can be supported at little cost).

Experience: I have implemented this. Code available on bitbucket. A fully associative cache is an entirely different beast, best done using Boost.MultiIndex.

Motivation: this data structure is essential to board game programs (chess, go, draughts), see any text on modern AI (Russell & Norvig e.g.). It's called the transposition table in those circles, and a fixed contiguous layout is required for performance (and also to avoid memory exhaustion). Most known open source programs have handwritten equivalents in C. I am not aware of a formalization of this data structure (not even in Boost).

6 A reboot of the bit franchise

Everything in a new namespace (`std2::` or `experimental` etc.)

- main goal: disentangle the conflation of set and vector abstractions (array of bits is both a set of ints and an array of bools).
- Rename `vector<bool, Alloc>` to `bool_vector<Alloc>`
- deprecate `vector<bool>` partial specialization, then remove
- Add similar `bool_array<N>` (also with random access proxy references/iterators)
- Rename `bitset<N>` to `flat_int_set<N>`
- Clean up interface to use `set` named members (`find`, `insert`, `erase` instead of `test`, `set` and `reset`)
- Clean up other anachronisms (implicit conversions, members vs non-members, checked vs non-checked access)
- add bidirectional proxy references/iterators (and `for_each`, see above)
- add more set algorithms (minus, lexicographical comparison)

- add `boost::dynamic_bitset<Alloc>` and rename it to `dyn_flat_int_set<Alloc>` and apply same cleanup as to the static version

Experience: I have a private version of `bitset` that has most of the `flat_int_set` features above. Code available on bitbucket. Thoroughly tested using almost identical unit tests as the official `boost::dynamic_bitset` test matrix.

Motivation: `bitset` is another essential data structure to board game programs (chess, go, draughts) because any board array of `N` squares with `2^K`-valued pieces can be conveniently expressed as a `K`-tuple of `bitset<N>` members. Typically with much higher data parallelism during pattern matching. Lack of iteration primitives (iterators or `for_each`) has led the board game programming community to ignore `bitset` in favor of raw 64-integers (chess/draughts) or handwritten classes (Go), invariably using macros for iteration.