Contributing to the C++ Standard

Starter's guide

Dan Raviv, Sound Radix
dan@soundradix.com

Nice to Meet You

- Dan Raviv, co-founder and C++ programmer at Sound Radix
- Presented my proposal “Add shift to `<algorithm>`” at last ISO meeting in Rapperswil, Switzerland - accepted into C++20! :-)
- Share the experience to help anyone thinking about contributing
- Still new to this myself

Goal

Know what you have to do to get your great idea into your favorite compiler

Background

- The C++ Standard is designed-by-committee: WG21 under ISO
- This is where the evolution path of C++ is decided!
- ~3 week-long meetings a year, at least one outside U.S. Open to everyone, but be prepared…
- 4 major groups: language/library, design/wording
- Extra domain-specific Study Groups: Concurrency, Tooling and more

Background

- Proposals are considered by the committee throughout the week by the various groups.

- For example, library additions go through LEWG, then LWG. When relevant also through Study Groups

- Final decisions about accepted/rejected proposals are made by official WG21 members: paying members, as well as representatives of “National Bodies” (countries)

- *But* most of the voting is non-formal and done in the work meetings throughout the week. Knowledgable attendees have a lot of influence on the future of C++.

Requirements

- Be an expert*
- Love C++
- Hate C++ (a little)
- Be prepared to invest time

Overview

- Idea!
- Experiment
- Gather feedback
- Write a proposal
- Submit and present your proposal
- Profit! (for the community)

Idea!

- Amazing new class/function/library feature
- Revolutionary language addition
- …or more commonly, just an incremental improvement
- Language changes = higher bar. Not recommended as a first proposal.

(A Good) Idea!

- Born of your work
- Clearly motivated and solves a real problem
    - Isn’t solved/handled already
- Be, or prepare to become a subject matter expert
- Aligned with C++ principles

(A Good) Idea!

- (Preferably) doesn’t break backwards compatibility
- Teachability
- Clearly belongs in the standard
- Not too big, unless you
    - really know what you’re doing
    - have the time to invest

Experiment

- Dollars to donuts you will need to experiment in code to gather interest / prove your idea’s merits / find issues / write your proposal
- Wandbox or similar for simple demonstrations
- Public git repo for library features
- Language changes - harder. I have no personal experience. Experimenting on paper would probably be fine for most proposals

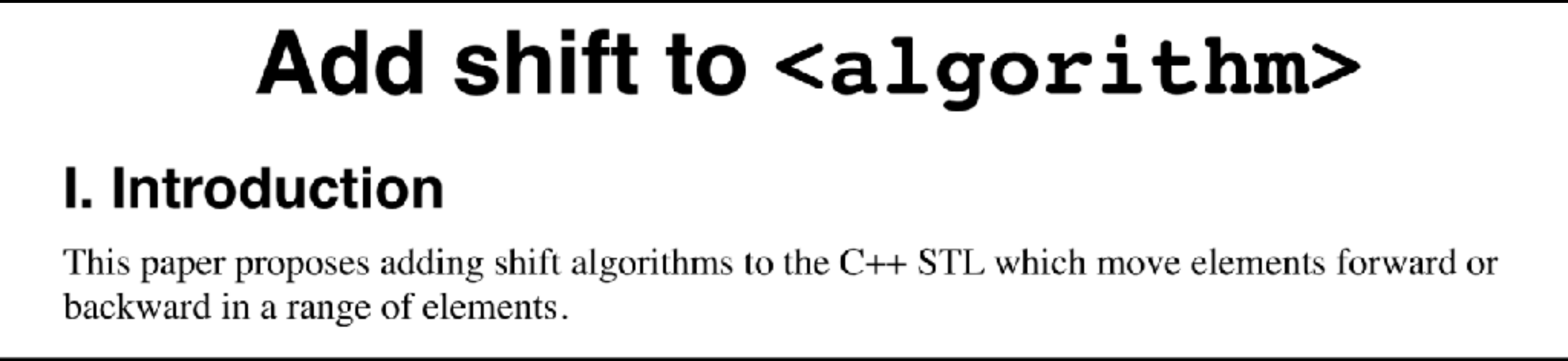
Gather Feedback

- Gather feedback at the std-proposals forum once you have something you’re comfortable presenting
- Iterative process - starting from a processed idea, followed by a first proposal draft, and followed by refinements based on feedback and your research
- Get feedback from your C++ programming friends and colleagues. People are usually curious about proposals by personal contacts

Write a Proposal

- Introduction
- Motivation and Scope
- Impact On The Standard
- Design Decisions
- Open Issues
- Proposed Wording
- Revision History
- Acknowledgements

Example



Add shift to `<algorithm>`

I. Introduction

This paper proposes adding shift algorithms to the C++ STL which move elements forward or backward in a range of elements.

II. Motivation and Scope

Shifting elements forward or backward in a range is a basic operation which the STL should allow performing easily. An important use case is time series analysis algorithms used in scientific and financial applications.

The scope of the proposal is adding the following function templates to `<algorithm>`:

```
template<class ForwardIterator>
constexpr ForwardIterator shift_left(
      ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_left(
      ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
template<class ForwardIterator>
ForwardIterator shift_right(
      ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_right(
      ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
```

A sample implementation which uses `std::move` to implement `shift_left` for forward iterators and `std::move_backward` to implement `shift_right` for bidirectional iterators can be found in https://github.com/danra/shift_proposal, though it's possible more efficient implementations could be made, since elements are guaranteed to be moved within the same range, not between two different ranges. The sample implementation also implements a non-trivial algorithm for `shift_right` of forward, non-bidirectional iterators.

IV. Impact On the Standard

The only impact on the standard is adding the proposed function templates to `<algorithm>`.

V. Design Decisions

1) `shift_left` and `shift_right` are provided as separate function templates instead of just a single `shift` function template to maximize performance and minimize compiled code size. Since shifting left and shifting right may have significantly different implementations (as is the case in the sample implementation), implementing both shift directions in a single `shift` function template would both require extra conditional logic and inline less easily than the specific direction shifts.

Given that both `shift_left` and `shift_right` are provided, it would still be possible to provide a convenience `shift` function as well, but it seems redundant.

2) `shift_left` should return an iterator to the new end of the shifted range. The beginning of the shifted range would always be equal to the beginning of the range before the shift, so there is no need to also return an iterator to the beginning of the shifted range. (This is similar to how `std::move` only returns an iterator to the end of the moved range).

Similarly, `shift_right` should return an iterator to the new beginning of the shifted range.

- In [alg.modifying.operations], after [alg.shuffle], add a new [alg.shift] section:

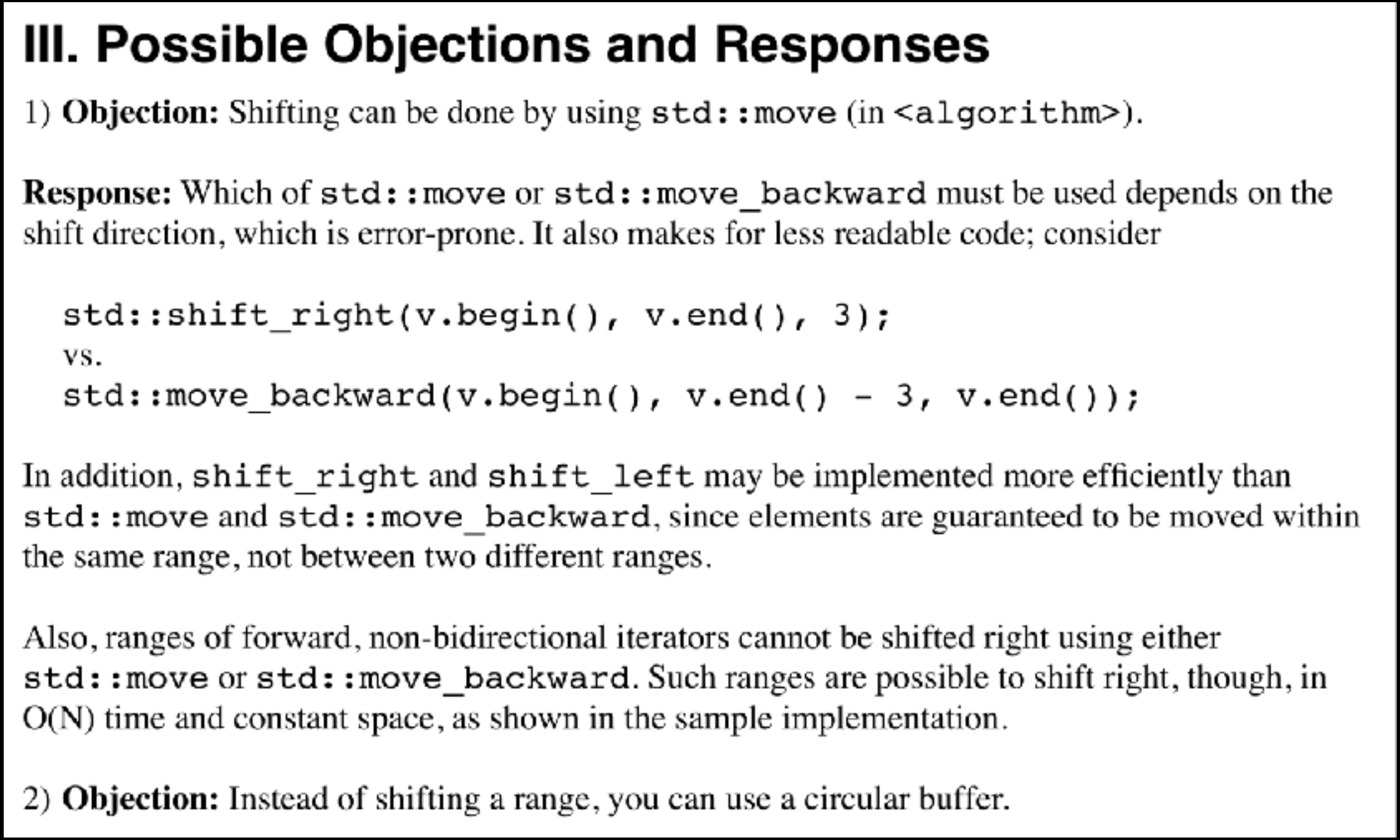
28.6.?? Shift [alg.shift]

```
template<class ForwardIterator>
constexpr ForwardIterator shift_left(
      ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_left(
      ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
```

1 *Requires:* The type of `*first` shall satisfy the `MoveAssignable` requirements.
2 *Effects:* If `n <= 0` or `n >= last - first`, does nothing. Otherwise, moves the element from position `first + n + i` into position `first + i` for each non-negative integer `i < (last - first) - n`. In the first overload case, does so in order starting from `i = 0` and proceeding to `i = (last - first) - n - 1`.

3 *Returns:* `first + (last — first) — n` if `n` is positive and `n < last — first`, otherwise `first` if `n` is positive, otherwise `last`.
4 *Complexity:* At most `(last — first) — n` assignments.

Fine to make your own



III. Possible Objections and Responses

1) **Objection:** Shifting can be done by using `std::move` (in `<algorithm>`).

Response: Which of `std::move` or `std::move_backward` must be used depends on the shift direction, which is error-prone. It also makes for less readable code; consider

```
std::shift_right(v.begin(), v.end(), 3);
```
vs.
```
std::move_backward(v.begin(), v.end() - 3, v.end());
```

In addition, `shift_right` and `shift_left` may be implemented more efficiently than `std::move` and `std::move_backward`, since elements are guaranteed to be moved within the same range, not between two different ranges.

Also, ranges of forward, non-bidirectional iterators cannot be shifted right using either `std::move` or `std::move_backward`. Such ranges are possible to shift right, though, in O(N) time and constant space, as shown in the sample implementation.

2) **Objection:** Instead of shifting a range, you can use a circular buffer.

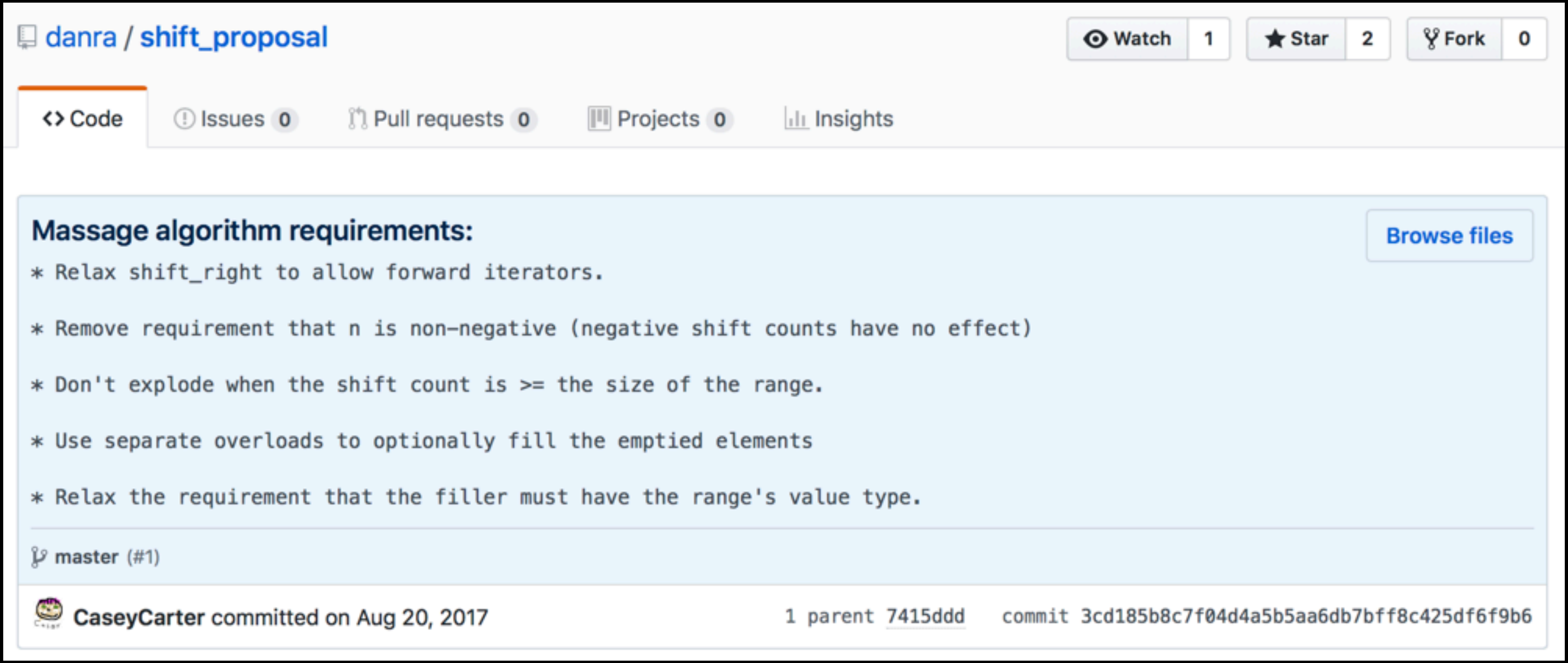
Audience

- Experts + other comments on std-proposals
- Relevant experts in the standards committee
- The most relevant people will read your proposal
- But in both cases a larger audience won't read your proposal and will expect you to describe it to them - Just as important!

Audience

- The *written* proposal is also a great tool for yourself!
  - Proof that you have explored the design space and came up with the best solution.
  - Everything that can be challenged about your proposal - will be challenged. Might as well cover the relevant issues in the proposal.
  - More easily solicit useful feedback from those who do read it
  - Allows someone else to present on your behalf.

Useful feedback and more!



Massage algorithm requirements:

```
* Relax shift_right to allow forward iterators.

* Remove requirement that n is non-negative (negative shift counts have no effect)

* Don't explode when the shift count is >= the size of the range.

* Use separate overloads to optionally fill the emptied elements

* Relax the requirement that the filler must have the range's value type.
```

master (#1)

CaseyCarter committed on Aug 20, 2017 — 1 parent 7415ddd — commit 3cd185b8c7f04d4a5b5aa6db7bff8c425df6f9b6

Submit Your Proposal

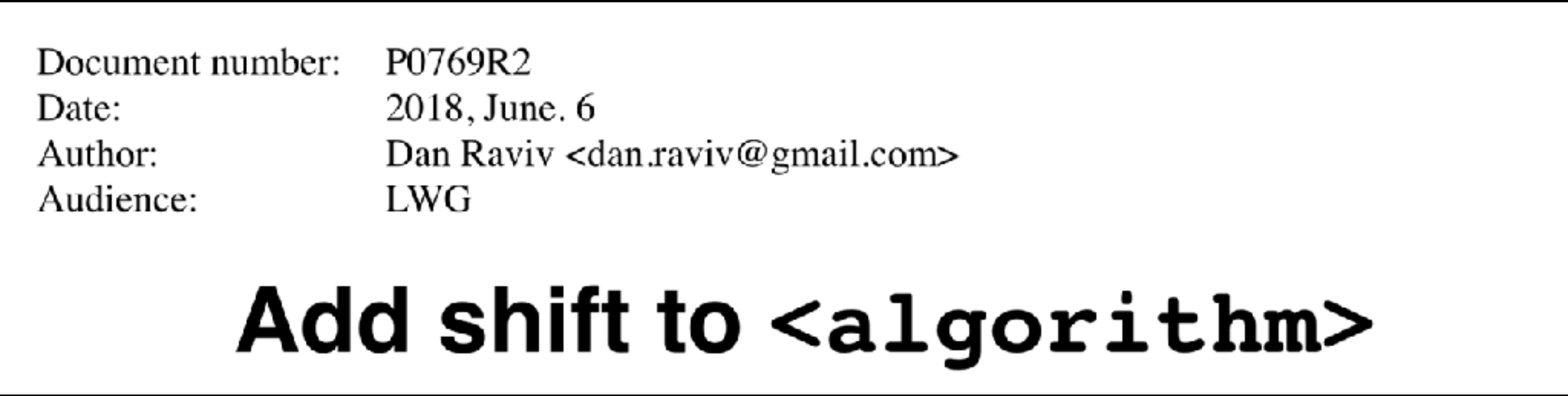
- Ask for an official number for your proposal from the WG21 vice-chair, Hal Finkel, hfinkel at anl.gov

- Use DxxxxRn for your proposal drafts on forums etc., then PxxxxRn for your final proposal which you send to the vice-chair

- List the committee groups which should receive and discuss the proposal

- Be aware of submission deadlines - currently the Monday three weeks after the start of each meeting

Example



Document number:	P0769R2
Date:	2018, June. 6
Author:	Dan Raviv <dan.raviv@gmail.com>
Audience:	LWG

Add shift to `<algorithm>`

Present Your proposal

- Two options:
  - Attend the meeting and present your own proposal
  - Get someone to present your proposal - preferably someone who has submitted feedback and thinks it’s a good one.

Present Your Proposal

- IMHO the second option, while offered by WG21, is much worse, unfortunately.
    - The pitch usually won't be as good, both content- and personal-wise. Most importantly, you <u>won't be there to respond to challenges</u> to the proposal
    - The responses during discussion, written down by note-takers, don't tell the whole story
    - Only do it if you really have no way to attend a meeting. It's better to delay your proposal for a meeting you can attend.
    - Mitigated somewhat if there are other people with invested interest in your written proposal, and more so if you co-write the proposal with someone who does attend.

Presenting to LEWG

- Present the idea, your design decisions and the considered alternatives and issues
- Your design will be challenged (“grilled”) - be prepared!
- Either consensus to advance to LWG, consensus to come back with revisions following the feedback, or a straw-poll

Presenting to LWG

- By this time the design was approved
- 95% of the time, only the Standard Wording section of the written proposal is considered.
- In few cases implementation limitations / obscure issues which appear to have been missed come up, and the proposal might be sent back to LEWG
- After presenting, you will get feedback on the wording, need to revise and then come back with the revision - if you don't attend meetings regularly, try to coordinate your presentation at the start of the week so you have time for this.

Possibly Fail…

- Some points where your proposal may fail:
    - Your idea might not gather enough interest.
    - Even if your idea is good you/your advocate may not pitch your proposal well.
    - Even if your pitch is good the committee members might not be interested, or be opposed to the proposal due to its drawbacks / impact on the existing Standard.
    - Even if an LEWG session forum supports your proposal and gives you feedback, the next session's forum where you present a revised edition might be opposed.
    - You might just give up in the middle because the entire process takes a lot of time and work.
- Did I mention you need to love C++ to do this? :)

…Or Succeed!

- Deepen your knowledge of C++ through the process
- Have discussions with and connect with top experts
- Make everyone’s life a little better… including your own!

More Information

- https://isocpp.org/std
- http://open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3370.html (a bit dated)
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/

Special Thanks

- Casey Carter
- Walter E. Brown

Questions?