stdx::interval, a library for intervals on totally ordered sets

Eric Hughes, Meadhbh Hamrick

In brief

`stdx::interval` implements the mathematical sense of an interval on a totally ordered set. The library reasons about intervals as sets, not as interval expressions.

The library is header-only and targets C++20. It is available under the MIT License.

Features

Predicates

Membership. Determine if a point is a member of the interval as a set.

Equality. Compares the intervals as sets, not as specifications.

Operations

Point comparison. A generalization of membership. Similar to `operator<=>`, but with heterogeneous operands.

Interval comparison. Also similar to `operator<=>`, but not transitive. Result is nonzero if the operands are disjoint, zero if there's a nontrivial intersection.

Intersection. Given two intervals, return their intersection. The result is the empty set if the two intervals are disjoint.

Union. Given two intervals, return the interval that is the union of the two intervals. This is a partial function, since the union of two intervals is not always an interval.

Cut. Given an interval and a point, cut the interval at the point, yielding two intervals. If the interval does not contain the point, one of the two results will be the empty set.

Why you may care

One level of abstraction higher than raw bounds. Intervals are a basic mathematical concept that appear in all kinds of algorithms, from recursive sorting to adaptive mesh refinement. Interval notation allows a more natural expression of the idea of an algorithm, replacing the clutter of variable pairs and a morass of explicit comparisons.

Generic. The library handles the differences between discrete (e.g. `int` and other integral types) and continuous (`string`, floating-point) types all under the hood. There's an internal type discriminator within each object that allows for infinite bounds without requiring that the underlying type have infinities.

Defect avoidance. There are a lot of details to get right when using intervals implicitly. It's easy to write $<$ for $\leq$, or be off-by-one on a discrete type calculation.

Nine kinds of expressions

An interval is a set of the form $\{x \mid a < x \wedge x < b\}$, where $<$ may be replaced by $\leq$ and one or both inequalities may be absent.

		Upper		
		closed	open	none
Lower	closed	$[a,b]$	$[a,b)$	$[a,+\infty)$
	open	$(a,b]$	(a,b)	$(a,+\infty)$
	none	$(-\infty,b]$	$(-\infty,b)$	$(-\infty,+\infty)$

Six groups of sets

- Empty
  - Group 1. No endpoints
    - The empty set $\emptyset$
- Finite
  - Group 2. One endpoint
    - The single point $[a,a]$
  - Group 3. Two distinct endpoints
    - Open (a,b)
    - Half-closed, half-open $[a,b)$
    - Half-open, half-closed $(a,b]$
    - Closed $[a,b]$
- Infinite
  - Group 4. Half-bounded below
    - Half-closed below $[a,+\infty)$
    - Half-open below $(a,+\infty)$
  - Group 5. Half-bounded above
    - Half-closed above $(-\infty,a]$
    - Half-open above $(-\infty,a)$
  - Group 6. Unbounded
    - Unbounded $(-\infty,+\infty)$

There are no edge cases. There are only ordinary cases.

Warning. What follows is a short rant by a known ranter [EH].

I hate the phrase "edge case". They're weasel words to avoid a commitment to quality and to shirk responsibility for achieving it. When I hear someone talking about their defective code that's because of an "edge case", I have to ask myself which kind of negative judgement of their behavior is most appropriate? Were they acting so incompetently that they didn't realize that there could even be such cases that they should account for? Did they know such cases might be there but they were too lazy to work out what they were?—or, even less charitably, they lacked the analytic skill to be able to work out what they were. Or, perhaps, were they making an appeal to the crowd that no one else gets it right, so they shouldn't be expected to either? Questions, questions, questions, but none of them have a good answer; the real question is "just how bad is it?"

In reliable software, all cases are accounted for. Edge cases do not exist, because all cases receive adequate analytical scrutiny. It's not that all cases receive equal scrutiny; some case require more analysis, some less. The actual difference between cases starts with their descriptions. Some cases have short, easy descriptions, but others only occur in certain circumstances. The more such requisite circumstances accrue, the longer the description and the greater the analysis to fully comprehend it. The proper goal is complete coverage of all cases, a full portfolio that understands everything that can happen.

So what should we call these cases instead? We can describe them along the gradient between simple and complex. The fully competent grasp the full complexity of a problem. They understand all the cases—the simple ones, the complex ones, the easy ones, the hard ones. The less-than-fully-competent grasp only an incomplete subset. They do not see the whole problem, and some cases are obscure to them.

In the case of `stdx::interval`, the complications come from degenerate cases. In the general case, there are four independent endpoints; when some coincide there are three or fewer. No individual case is difficult; what's difficult is the combinatorial expansion of cases. There are no edge cases in `stdx::interval` because the analysis is exhaustive.

Reliability: Testing the intersection operator

Eliminating edge cases means exhaustively testing all combinatorial possibilities of constructing possible intersection terms. In all of the foregoing, $a < b < c < d$. Closed bounds are shown for brevity; both open and closed bounds are tested.

- Four unique endpoints
  - Disjoint
    - $[a,b] \cap [c,d]$
    - $(-\infty,b) \cap [c,d]$
    - $[a,b] \cap [c,+\infty)$
    - $(-\infty,b] \cap [c,+\infty)$
  - Overlap
    - $[a,c] \cap [b,d]$
    - $(-\infty,c] \cap [b,d]$
    - $[a,c] \cap [b,+\infty)$
    - $(-\infty,c] \cap [b,+\infty)$
  - Surround
    - $[a,d] \cap [b,c]$
    - $(-\infty,d] \cap [b,c]$
    - $[a,+\infty) \cap [b,c]$
    - $(-\infty,+\infty) \cap [b,c]$
- Three unique endpoints
  - Disjoint
    - $[a,a] \cap [b,c]$
    - $[a,a] \cap [b,+\infty)$
    - $[a,b] \cap [c,c]$
    - $(-\infty,b] \cap [c,c]$
  - Surround
    - $[a,b] \cap [a,c]$
    - $(-\infty,b] \cap (-\infty,c]$
    - $[a,b] \cap [a,+\infty)$
    - $(+\infty,b] \cap (-\infty,+\infty)$
    - $[a,c] \cap [b,b]$
    - $(-\infty,c] \cap [b,b]$
    - $[a,+\infty) \cap [b,b]$
    - $(-\infty,+\infty) \cap [b,b]$
    - $[a,c] \cap [b,c]$
    - $(-\infty,c] \cap [b,c]$
    - $[a,+\infty) \cap [b,+\infty)$
    - $(-\infty,+\infty) \cap [b,+\infty)$
  - Adjacent
    - $[a,b] \cap [b,c]$
    - $(-\infty,b] \cap [b,c]$
    - $[a,b] \cap [b,+\infty)$
    - $(-\infty,b] \cap [b,+\infty)$
- Two unique endpoints
  - Identical
    - $[a,b] \cap [a,b]$
    - $(-\infty,b] \cap (-\infty,b]$
    - $[a,+\infty) \cap [a,+\infty)$
    - $(-\infty,+\infty) \cap (-\infty,+\infty)$
  - Disjoint
    - $[a,b] \cap \emptyset$
    - $(-\infty,b] \cap \emptyset$
    - $[a,+\infty) \cap \emptyset$
    - $(-\infty,+\infty) \cap \emptyset$
    - $[a,a] \cap [b,b]$
- One unique endpoint
  - Identical
    - $[a,a] \cap [a,a]$
  - Disjoint
    - $[a,a] \cap \emptyset$
- No endpoints
  - Disjoint
    - $\emptyset \cap \emptyset$

This is all for a single operator. The other operators have analogous decompositions that are equally exhaustive. No one ever said eliminating edge cases was simple. It's not particularly difficult, but it does require being methodical.

Example code

Bisection Search

```cpp
#include <interval>
#include <span>

using IndexI = stdx::Interval<int>;

int search(int v, std::span<int> values, IndexI interval) {

    if (interval.is_empty()) {
        return -1;
    }

    if (interval.has_single_point()) {
        if (values[interval.lower_bound()] == v) {
            return interval.lower_bound();
        } else {
            return -1;
        }
    }

    int pivot = (interval.lower_bound() + interval.upper_bound()) / 2;

    auto [lower, upper] = interval.cut(pivot);

    auto c = values[pivot] <=> v;

    if (c < 0) {
        return search(v, values, upper);
    } else if (c > 0) {
        return search(v, values, lower);
    } else {
        return pivot;
    }

}
```

Future Directions

Documentation. The code itself has adequate Doxygen comments, but is lacking exposition, examples, etc.

Better testing. The testing, while quite good, isn't perfect. The conformance tests (necessarily black-box) are commingled with some white-box tests.

Strings. Add support for `string_view`. This was always on the original plan, but fell by the wayside.

C++20. The code was originally written for C++17 and needs some updating. For example, it still uses `std::enable_if` instead of `concept`. It needs to be made available as a module for `import`.

The library needs a working accommodation with the concept `std::totally_ordered`. The unfortunate fact is that this concept does not enforce its stated semantic requirements (so why were they stated at all?), and so the concept is not reliable for input validation at construction. There will need to be some kind of workaround to use syntactic requirements as a proxy for semantic ones.

Time. Clock duration is conventionally a half-closed, half-open interval. (This is not meant to be an obvious statement.) Proper integration with `std::chrono` would take this as a default and yet also integrate correctly with other intervals.

Standard library. We'd like this library to ground a proposal for inclusion in a future standard. The first proposal has not yet been submitted.

Curiosities

The union of intervals expression $[1,2] \cup [3,4]$ depends on what set the intervals are subsets of. If they are integer intervals, then the result is the interval $[1,4]$. If they are floating-point intervals, then the result is not an interval. `stdx::interval` gets this right by using an "adjacency" predicate. Adjacency says for $a < b$ whether there is some c that separates them: $a < c < b$. Adjacent intervals can be merged.

The set of strings that begin with a particular prefix is an interval. For example, the set of strings that begin with *"abc"* is the interval $["abc", "abd")$. This set cannot be expressed with only closed bounds. Supposing the alphabet is only the lowercase letters, the sequence of strings *"abcz"*, *"abczz"*, *"abczzz"*, ... are all less that *"abd"*. Since this sequence is unbounded, there is no "last string" before the upper bound.

The code

`https://gitlab.com/stdx_interval/code`