ITERATOR HAIKU

HOW THE RANGES TS TURNED 5 ITERATOR CATEGORIES INTO 7, AND BACK INTO 5 AGAIN

CASEY@CARTER.NET

OVERVIEW

- Ranges and iterators in Standard C++
- Iterator categories in the Palo Alto TR
- Ranges and iterators in the Ranges TS
  - Concept design flaws
  - Investigation leading to inconsistency
  - Re-design

RANGES IN STANDARD C++

- Why do we need a TS if the Standard already has ranges?!?
  - I have a family to feed
- Standard has **two** kinds of ranges

RANGE-SEQUENCE: ELEMENTS "BETWEEN" TWO ITERATORS

- Often denoted [i, k)
- Implicitly requires reachability
- == is an end-of-sequence test

RANGE-OBJECT: THINGS THAT WORK IN RANGE-BASED-FOR

- “Range” is right there in the name
- begin(thing) and end(thing) return iterators
  - [begin(thing), end(thing))

THE FIVE C++14 ITERATOR CATEGORIES

- Input
  - *, ++, ==, !=
  - Single-pass
  - Readable
    - Writable?

```
auto foo = ifstream{“integers.txt”};
using I = istream_iterator<int>;
assert(is_sorted(I{foo}, I{}));
```

THE FIVE C++14 ITERATOR CATEGORIES

- Forward
  - Input
  - Multi-pass

```
auto foo = forward_list<int>{0,1,2,3};
assert(is_sorted(begin(foo), end(foo)));
```

THE FIVE C++14 ITERATOR CATEGORIES

- Bidirectional
  - Forward
  - --

```
auto foo = list<int>{0,1,2,3};
reverse(begin(foo), end(foo));
```

THE FIVE C++14 ITERATOR CATEGORIES

- RandomAccess
  - Bidirectional
  - -, +, -, +=, -=, []

```
auto foo = vector<int>{0,1,2,3};
sort(begin(foo), end(foo));
```

THE FIVE C++14 ITERATOR CATEGORIES

- Output
  - *, ++
  - Single-pass
  - Writable
    - Readable?

```
auto foo = vector<int>{0,1,2,3};
using O = ostream_iterator<int>;
copy(begin(foo), end(foo), O{cout});
```

DOMAIN OF A FUNCTION

- The set of values over which the function (operation, expression) is defined

```
int f(int i) {
    return 42 / i;
}
```

FORWARD: DOMAIN OF ==

- Iterators over the same underlying sequence (N4606 [forward.iterators]/2)

OUTPUT: DOMAIN OF ==

- There is no spoon

INPUT: DOMAIN OF ==

… the term *the domain of* == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. [*Example:* the call find(a, b, x) is defined only if the value of a has the property *p* defined as follows: b has property *p* and a value i has property *p* if (*i == x) or if (*i != x and ++i has property p). —*end example*]

PALO ALTO TR

- N3351 "A Concept Design for the STL" (http://wg21.link/n3351)
- Redefine iterators and algorithms in terms of Concepts
  - Replaces tables in the spec with syntax in the library
- Awesome. But what's a concept?

CONCEPTS

- Syntax (Concepts TS):
  - Predicates over a set of parameters
  - Requirements on types (e.g., “auto foo = bar;”)
    - Associated types that must exist
    - Expressions that must be valid
- Semantics (Ranges TS):
  - Requirements on values (e.g., “foo must equal bar”)

PALO ALTO TR

- Some algorithms take unpaired input iterators
  - “3-legged” double-range algorithms
    - equal(first1, last1, first2)
  - iterator + count algorithms
    - copy_n(first, count, out)
- Unpaired input iterators don’t need == and !=
  - New category WeakInputIterator

RANGES TS

- Range-object algorithms
  - Turn a range-object into a range-sequence with begin and end
  - Invoke the range-sequence algorithm
- Output ranges?
- Sentinels?
  - Not the X-Men hunting killer robots
  - A new way to denote range-sequences

RANGES TS: OUTPUT

- Rename OutputIterator to WeakOutputIterator
- Add a new OutputIterator with == and !=
  - WeakOutput : Output == WeakInput : Input

5 HAVE BECOME 7

- Output, Input, Forward, Bidirectional, RandomAccess, WeakOutput, WeakInput
- Ranges - now with 40% more categories!

ITERATOR VS. WEAKITERATOR??

- Factor commmonality out of WeakInput and WeakOutput into WeakIterator
- Factor commmonality out of Input and Output into Iterator
- "iterators" colloquially means....models of "WeakIterator"?!?

INTRODUCE SENTINELS INTO THE MODEL

- A distinct *value* that denotes the end of a range
  - `istream_iterator<T>()`
- Why not use a distinct *type*?

SENTINEL CONCEPT

- Relates a type I that satisfies Iterator and a type S that satisfies Regular
- Requires I and S to satisfy EqualityComparable

CROSS-TYPE EQUALITYCOMPARABLE

- Cross-type relations require relation to hold for
  - Both individual types
  - Common type
    - What’s a common type?

COMMON TYPE

- C is a common type of T and U if:
    - C(t) is valid
    - C(t1) == C(t2) iff t1 == t2
    - (symmetric requirement for u)
- e.g.:
    - int is a common type of int and short (so is long, and long long)
    - int is a common type of int and int
    - std::string is NOT a common type of char* and char const* (equality of pointers is different from equality of pointers-converted-to-strings)

CROSS-TYPE EQUALITYCOMPARABLE

- Cross-type relations require relation to hold for
  - Both individual types
  - Common type
- Cross-type EC also requires that:
  - C(t1) == C(u) && t1 == t2 implies C(t2) == C(u)
  - (symmetric requirement)

In other words, the equality is transitive across types.

SENTINEL CONCEPT

- Relates a type I that satisfies Iterator and a type S that satisfies Regular
- Requires I and S to satisfy EqualityComparable
  - I and S must be individually EC
    - No “weak” ranges
    - despite that algorithms don’t use == on sentinels or single-pass iterators

ALGORITHMS DON'T USE == FOR SINGLE-PASS ITERATORS

```
template<InputIterator I, Sentinel<I> S, /*…*/>
bool any_of(I first, S last, P predicate) {
    for (; first != last; ++first) {
        if (invoke(predicate, *first)) {
            return true;
        }
    }
    return false;
}
```

ALGORITHMS DON'T USE == FOR SINGLE-PASS ITERATORS

- Concepts require == / != anyway
- Either useless or have undefined behavior
  - Not just a footgun, but a *required* footgun!

SENTINEL CONCEPT

- Relates a type I that satisfies Iterator and a type S that satisfies Regular
- Requires I and S to satisfy EqualityComparable
  - I and S must be individually EC
    - No “weak” ranges
    - despite that algorithms don’t use == on sentinels or single-pass iterators

ITERATORS AND SENTINELS ARE BOTH “POSITIONS”

- General recommendation to treat all sentinels as equal
  - Sentinel represents the “at the end” position

AXIOMATIZATION OF ITERATORS WITH SENTINELS

- Throw out everything to do with ranges-as-iterator pairs
- Start over with only [iterator, sentinel) ranges
- Derive range properties
- Define domains of iterator and sentinel operations)
- Sadly not enough time to cover here
  - Cool math like “if [i, s) denotes a range then either i == s or [++i, s) denotes a range”
  - Everyone likes derivations
- Woops, problem with stateful sentinels

STATEFUL SENTINELS AND “ALWAYS EQUAL”

```
struct S {
    int i;

    friend bool operator==(int const* p, S const& s) {
        return *p >= s.i;
    }
    friend bool operator==(S const&, int const*);
    friend bool operator==(S const&, S const& ) {
        return true;
    }
    // define operator!= overloads appropriately
};
```

STATEFUL SENTINELS AND “ALWAYS EQUAL”

```
int a[] = {2, 1, 3};

assert(a+1 != S{2});   // !(1 >= 2)
assert(a+2 == S{2});   // (3 >= 2), so [a+1,S{2}) is {1}

assert(a+1 != S{3});   // !(1 >= 3)
assert(a+2 == S{3});   // (3 >= 3), so [a+1,S{3}) is {1}
assert(S{2} == S{3}); // Cross-type EC

assert(a+0 == S{2});   // (2 >= 2), so [a+0,S{2}) is {}
assert(a+0 == a+2);    // Cross-type EC
```

S SURE LOOKS LIKE A SENTINEL TO ME

- “Sentinels are positions” vs “Sentinels are predicates”
- How did this problem not show up in the two implementations?

START OVER WITH ALGORITHM REQUIREMENTS

- Algorithms care about:
  - i == s, s == i, i != s, s != i have the same domain
  - Symmetry: (i == s) == (s == i) and (i != s) == (s != i)
  - Complement: i != s == !(i == s)
  - i == s is well-defined when [i, s) denotes a range
- First three look like general equality comparison requirements
  - Define a weaker cross-type equality for iterators and sentinels

WEAKLYEQUALITYCOMPARABLE

- Non-transitive cross-type equality requirement for iterators/sentinels
- Prior EqualityComparable concepts refine WeaklyEqualityComparable
- Yuck: invoking a predicate with == syntax
  - Semantic abomination
    - Transitivity is absent, but preserves other EC semantics
  - Backwards compatible

WEAK RANGES ARE POSSIBLE

- Without cross-type EqualityComparable, no need for individually EqualityComparable types
- Sentinel becomes:
  - Relates a type I that satisfies WeakIterator and a type S that satisfies Semiregular
  - Requires I and S to satisfy WeaklyEqualityComparable

STATEFUL SENTINELS WORK

```
struct S {
    int i;

    friend bool operator==(int const* p, S const& s) {
        return *p >= s.i;
    }
    friend bool operator==(S const&, int const*);
    friend bool operator==(S const& x, S const& y) {
        return x.i == y.i;
    }
    // define operator!= overloads appropriately
};
```

STATEFUL SENTINELS WORK

```
int a[] = {2, 1, 3};

assert(a+1 != S{2});  // !(1 >= 2)
assert(a+2 == S{2});  // (3 >= 2), so [a+1,S{2}) is {1}

assert(a+1 != S{3});  // !(1 >= 3)
assert(a+2 == S{3});  // (3 >= 3), so [a+1,S{3}) is {1}
assert(S{2} == S{3}); // Cross-type EC
assert(S{2} != S{3});

assert(a+0 == S{2});  // (2 >= 2), so [a+0,S{2}) is {}
assert(a+0 == a+2);   // Cross-type EC
```

BUT THEN "STRONG" CONCEPTS AREN'T NEEDED

- A type satisfies (Input|Output|)Iterator if it
  - satisfies Weak(Input|Output|)Iterator
  - is a sentinel for itself (i.e., T satisfies Sentinel<T, T>())
- Algorithms don't need "strong" anyway
- If we don't have "strong" variants anymore, why use the "Weak" prefix?

7 BECOMES 5

- Augment (InputIterator|OutputIterator|Iterator) requirements with Sentinel requirements
- Strip “Weak” prefixes
- Replaces the “weak vs strong” distinction with “sentinel vs non-sentinel”

MOST IMPORTANTLY

- Writers of single-pass iterators and sentinels need not write:
  - bool operator == (..) { return true; }
- We’ve turned undefined behavior into a compile error!
  - Well, sort of.
- We’ve reduced the number of concepts
- “Iterators” actually means iterators

TAKEAWAY: CONCEPT LIBRARY DESIGN

- Concepts that don't exactly fit the usage requirements smell
  - Trade-off Minimality with Purity
- Deriving requirements from first principles can give clarity
  - I did not see the inherent contradiction in:
    - Sentinels are always equal
    - Iterators are sometimes Sentinels
    - Iterators are never always equal

HOW CAN I LEARN MORE?

- Ranges TS working paper: http://wg21.link/n4569
- Early design blogs at http://ericniebler.com/
- Implementation with Concepts: https://github.com/CaseyCarter/cmcstl2 (GCC 6+)
- Implementation with C++11 “concepts”: https://github.com/EricNiebler/range-v3 (GCC 4.8+, Clang 3.3-ish, Clang/C2)
- Implementation with extensive MSVC workarounds: https://github.com/Microsoft/Range-V3-VS2015 VS2015 Update 3

LIBRARIES ACQUISITION

- 80% of C++ projects use 2 or more 3rd party libs
  - A majority of them use open source libraries
- Acquiring and rebuilding libs on Windows can be simple
  - NuGet wasn't designed for C++ (e.g. no local rebuilding)
- Open source tool based on a port tree approach: "VCPKG"
  - Usage: VCPKG install Boost
  - Installs the .h, .lib and binaries in a "lib folder" ready to use

https://github.com/Microsoft/VCPKG

boost
cpprestsdk
curl
expat
freetype
glew
glfw3
libjpeg-turbo
libpng
libuv
libwebsockets
mpg123
openal-soft
opencv
openssl
range-v3
sdl2
sqlite3
tiff
tinyxml2
zlib