Iterators Must Go

Andrei Alexandrescu

This Talk

- The STL
- Iterators
- Range-based design
- Conclusions

What is the STL?

Yeah, what *is* the STL?

- A good library of algorithms and data structures.

Yeah, what *is* the STL?

- A $\big($ good$\big|$bad$\big)$ library of algorithms and data structures.

Yeah, what *is* the STL?

- A $\left(\text{good} \middle| \text{bad} \middle| \text{ugly} \right)$ library of algorithms and data structures.

Yeah, what *is* the STL?

- A (good|bad|ugly) library of algorithms and data structures.

- `iterators = gcd(containers, algorithms);`

Yeah, what *is* the STL?

- A (good|bad|ugly) library of algorithms and data structures.
- `iterators = gcd(containers, algorithms);`
- **S**crumptious **T**emplate **L**ore
- **S**nout to **T**ail **L**ength

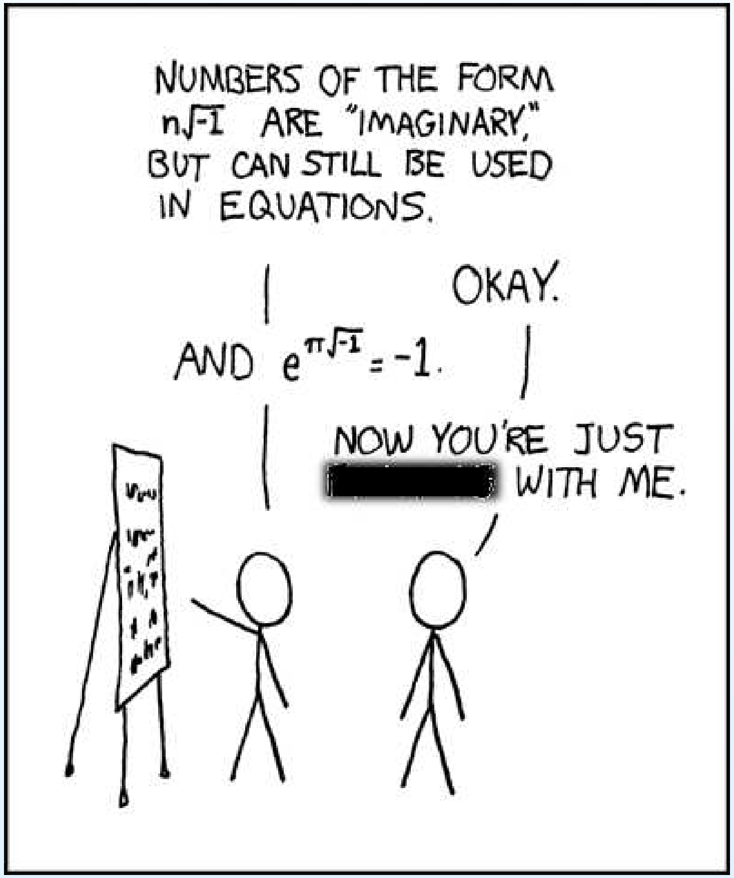
What the STL is

- More than the answer, the *question* is important in the STL
- "What would the most general implementations of fundamental containers and algorithms look like?"
- Everything else is aftermath
- Most importantly: STL is *one* answer, not *the* answer

STL is nonintuitive

STL is nonintuitive

- Same way the theory of relativity is nonintuitive
- Same way complex numbers are nonintuitive (see e.g. `xkcd.com`)



Nonintuitive

- “I want to design the most general algorithms.”
- “Sure. What you obviously need is something called iterators. Five of ’em, to be precise.”

Nonintuitive

- "I want to design the most general algorithms."
- "Sure. What you obviously need is something called iterators. Five of 'em, to be precise."
- Evidence: No language has supported the STL "by chance."
  - In spite of relentless "feature wars"
  - C++, D the only ones
  - Both were actively designed to support the STL
- Consequence: STL very hard to understand from outside C++ or D

Fundamental vs. Incidental in STL

- Algorithms defined for the narrowest interface possible
- Broad iterator categories as required by algorithms
- Choice of iterator primitives
- Syntax of iterator primitives

Fundamental vs. Incidental in STL

- **F:** Algorithms defined for the narrowest interface possible
- Broad iterator categories as required by algorithms
- Choice of iterator primitives
- Syntax of iterator primitives

Fundamental vs. Incidental in STL

- **F:** Algorithms defined for the narrowest interface possible
- **F:** Broad iterator categories as required by algorithms
- Choice of iterator primitives
- Syntax of iterator primitives

Fundamental vs. Incidental in STL

- **F:** Algorithms defined for the narrowest interface possible
- **F:** Broad iterator categories as required by algorithms
- **I:** Choice of iterator primitives
- Syntax of iterator primitives

Fundamental vs. **I**ncidental in STL

- **F:** Algorithms defined for the narrowest interface possible
- **F:** Broad iterator categories as required by algorithms
- **I:** Choice of iterator primitives
- **I:** Syntax of iterator primitives

STL: The Good

- Asked the right question
- General
- Efficient
- Reasonably extensible
- Integrated with built-in types

STL: The Bad

- Poor lambda functions support

    Not an STL problem

    High opportunity cost

- Some containers cannot be supported

    E.g. sentinel-terminated containers

    E.g. containers with distributed storage

- Some iteration methods cannot be supported

STL: The Ugly

- Attempts at `for_each` et al. didn't help
- Integration with streams is tenuous
- One word: `allocator`

STL: The Ugly

- Attempts at `for_each` et al. didn't help
- Integration with streams is tenuous
- One word: `allocator`
- Iterators suck
  - Verbose
  - Unsafe
  - Poor Interface

What's the Deal with Iterators?

Iterators Rock

- They broker interaction between containers and algorithms
- “Strength reduction:” $m + n$ implementations instead of $m \cdot n$
- Extensible: there’s been a flurry of iterators ever since STL saw the light of day

Warning Sign #1

- C++ Users Journal around 2001 ran an ad campaign

  “Submit an article to CUJ!”

  “No need to be an English major! Just start your editor!”

  “We’re interested in security, networking, C++ techniques, and more!”

Warning Sign #1

- C++ Users Journal around 2001 ran an ad campaign

  "Submit an article to CUJ!"

  "No need to be an English major! Just start your editor!"

  "We're interested in security, networking, C++ techniques, and more!"

  "Please note: Not interested in yet another iterator"

- How many of those published iterators survived?

Warning Sign #2

File copy circa 1975:

```
#include <stdio.h>
int main() {
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return errno != 0;
}
```

Warning Sign #2

Fast forward 20 years, and. . .

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

int main() {
   copy(istream_iterator<string>(cin),
      istream_iterator<string>(),
      ostream_iterator<string>(cout,"\n"));
}
```

(forgot the `try`/`catch` around `main`)

Something, somewhere, went terribly wrong.

Warning Sign #3

- Iterators are brutally hard to define
- Bulky implementations and many gotchas
- Boost includes an entire library that helps defining iterators
- The essential primitives are like... three?
  - At end
  - Access
  - Bump

Warning Sign #4

- Iterators use pointer syntax & semantics
- Integration with pointers for the win/loss
- However, this limits methods of iteration
  - Can't walk a tree in depth, need `++` with a parameter
  - Output iterators can only accept one type: `ostream_iterator` must be parameterized with each specific type to output, although they all go to the same place

Final nail in the coffin

- *All* iterator primitives are fundamentally unsafe
- For most iterator types, given an iterator
  - Can't say whether it can be compared
  - Can't say whether it can be incremented
  - Can't say whether it can be dereferenced
- Safe iterators can and have been written
  - At a high size+speed cost
  - Mostly a good argument that the design hasn't been cut quite right

Ranges

Enter Ranges

- To partly avoid these inconveniences, ranges have been defined
- A range is a pair of begin/end iterators packed together
- As such, a range has higher-level checkable invariants
- Boost and Adobe libraries defined ranges

They made an interesting step in a good direction.

Things must be taken much further.

Look, Ma, no iterators!

- How about defining ranges instead of iterators as the primitive structure for iteration?
- Ranges should define primitive operations that do not rely on iteration
- There would be no more iterators, only ranges
- What primitives should ranges support?

    Remember, `begin`/`end` are not an option

    If people squirrel away individual iterators, we're back to square one

Defining Ranges

- All of `<algorithm>` should be implementable with ranges, and other algorithms as well
- Range primitives should be checkable at low cost
- Yet, ranges should not be less efficient than iterators

Input/Forward Ranges

```
template<class T> struct InputRange {
    bool empty() const;
    void popFront();
    T& front() const;
};
```

Verifiable?

```
template<class T> struct ContigRange {
    bool empty() const { return b >= e; }
    void popFront() {
        assert(!empty());
        ++b;
    }
    T& front() const {
        assert(!empty());
        return *b;
    }
private:
    T *b, *e;
};
```

Find

```
// Original version per STL
template<class It, class T>
It find(It b, It e, T value) {
    for (; b != e; ++b)
        if (value == *b) break;
    return b;
}
...
auto i = find(v.begin(), v.end(), value);
if (i != v.end()) ...
```

Design Question

- What should `find` with ranges look like?

  1. Return a range of one element (if found) or zero elements (if not)?
  2. Return the range *before* the found element?
  3. Return the range *after* the found element?

Design Question

- What should `find` with ranges look like?

    1. Return a range of one element (if found) or zero elements (if not)?
    2. Return the range *before* the found element?
    3. Return the range *after* the found element?

- Correct answer: *return the range starting with the found element (if any), empty if not found*

Design Question

- What should `find` with ranges look like?

  1. Return a range of one element (if found) or zero elements (if not)?
  2. Return the range *before* the found element?
  3. Return the range *after* the found element?

- Correct answer: *return the range starting with the found element (if any), empty if not found*

  Why?

Find

```
// Using ranges
template<class R, class T>
R find(R r, T value) {
    for (; !r.empty(); r.popFront())
        if (value == r.front()) break;
    return r;
}
...
auto r = find(v.all(), value);
if (!r.empty()) ...
```

Elegant Specification

```
template<class R, class T>
R find(R r, T value);
```

"Reduces the range `r` from left until its front is equal with `value` or `r` is exhausted."

Bidirectional Ranges

```
template<class T> struct BidirRange {
   bool empty() const;
   void popFront();
   void popBack();
   T& front() const;
   T& back() const;
};
```

Reverse Iteration

```
template<class R> struct Retro {
   bool empty() const { return r.empty(); }
   void popFront() { return r.popBack(); }
   void popBack() { return r.popFront(); }
   E<R>::Type& front() const { return r.back(); }
   E<R>::Type& back() const { return r.front(); }
   R r;
};
template<class R> Retro<R> retro(R r) {
   return Retro<R>(r);
}
template<class R> R retro(Retro<R> r) {
   return r.r; // klever
}
```

How about `find_end`?

```
template<class R, class T>
R find_end(R r, T value) {
   return retro(find(retro(r));
}
```

- No more need for `rbegin`, `rend`
- Containers define `all` which returns a range
- To iterate backwards: `retro(cont.all())`

`find_end` with iterators sucks

```
// find_end using reverse_iterator
template<class It, class T>
It find_end(It b, It e, T value) {
    It r = find(reverse_iterator<It>(e),
        reverse_iterator<It>(b), value).base();
    return r == b ? e : --r;
}
```

- Crushing advantage of ranges: much terser code
- Easy composition because only one object needs to be composed, not two in sync

More composition opportunities

- `Chain`: chain several ranges together

  Elements are not copied!

  Category of range is the *weakest* of all ranges

- `Zip`: span several ranges in lockstep

  Needs `Tuple`

- `Stride`: span a range several steps at once

  Iterators can’t implement it!

- `Radial`: span a range in increasing distance from its middle (or any other point)

How about three-iterators functions?

```
template<class It1, It2>
void copy(It1 begin, It1 end, It2 to);
template<class It>
void partial_sort(It begin, It mid, It end);
template<class It>
void rotate(It begin, It mid, It end);
template<class It, class Pr>
It partition(It begin, It end, Pr pred);
template<class It, class Pr>
It inplace_merge(It begin, It mid, It end);
```

"Where there's hardship, there's opportunity."

– I. Meade Etop

3-legged algos ⇒ mixed-range algos

```
template<class R1, class R2>
R2 copy(R1 r1, R2 r2);
```

- Spec: Copy `r1` to `r2`, returns untouched portion of `r2`

```
vector<float> v;
list<int> s;
deque<double> d;
copy(chain(v, s), d);
```

3-legged algos ⇒ mixed-range algos

```
template<class R1, class R2>
void partial_sort(R1 r1, R2 r2);
```

- Spec: Partially sort the concatenation of `r1` and `r2` such that the smallest elements end up in `r1`
- You can take a vector and a deque and put the smallest elements of *both* in the array!

```
vector<double> v;
deque<double> d;
partial_sort(v, d);
```

But wait, there's more

```
vector<double> v1, v2;
deque<double> d;
partial_sort(v1, chain(v2, d));
sort(chain(v1, v2, d));
```

- Algorithms can now operate on any mixture of ranges seamlessly without any extra effort
- Try *that* with iterators!

But wait, there's even more

```
vector<double> vd;
vector<string> vs;
// sort the two in lockstep
sort(zip(vs, vd));
```

- Range combinators allow myriads of new uses
- Possible in theory with iterators, but the syntax would explode (again)

Output ranges

- Freedom from pointer syntax allows supporting different types

```
struct OutRange {
   typedef Typelist<int, double, string> Types;
   void put(int);
   void put(double);
   void put(string);
}
```

Back to copying `stdin` to `stdout`

```
#include <...>
int main() {
   copy(istream_range<string>(cin),
      ostream_range(cout, "\n"));
}
```

- Finally, a step forward: a one-liner that fits on one line[1]
- `ostream_range` does not need to specify `string`

[1] slide limitations notwithstanding

Infinite ranges

- Notion of infinity becomes interesting with ranges
- Generators, random numbers, series, . . . are infinite ranges
- Infinity is a trait distinct from the five classic categories; any kind of range may be infinite
- Even a random-access range may be infinite!
- Statically knowing about infinity helps algorithms

`has_size`

- Whether a range has an efficiently computed *size* is another independent trait
- (Index entry: `list.size`, endless debate on)
- Even an input range can have a known size, e.g. `take(100, rndgen)` which takes 100 random numbers
  - `take(100, r)` has length 100 if `r` is infinite
  - length `min(100, r.size())` if `r` has known length
  - unknown length if `r` is finite with unknown length

A Twist

- Can `<algorithm>` be redone with ranges?
- D's stdlib offers a superset of `<algorithm>` in modules `std.algorithm` and `std.range` (google for them)
- Ranges pervade D: algorithms, lazy evaluation, random numbers, higher-order functions, `foreach` statement. . .
- Some opportunities not yet tapped—e.g. filters (input/output ranges)
- Check "The Case for D" in Doctor Dobb's Journal, coming soon

Conclusion

- Ranges are a superior abstraction
- Better checking abilities (not perfect still)
- Easy composition
- Range-based design offers much more than a port of iterator-based functions to ranges
- Exciting development taking STL one step further

Announcement

Please note: “The D Programming Language” soon to appear on Safari’s Rough Cuts.

Andrei Alexandrescu, Ph %@! D

Please note: Andrei will soon be offering training and consulting services. Contact him for details.