D’s Gems: Ranges

Part I: Input Ranges

Mario Kröplin

SIMPLY EXPLAINED

geek & poke
PLEASE DON'T ASK ME HOW THIS WORKS AND PLEASE DON'T ASK ME WHY THIS WORKS!!

STACK OVERFLOW

References

Andrei Alexandrescu: *On Iteration* (2009)
http://erdani.com/publications/on-iteration.html

Iterators Must Go
https://accu.org/content/conf2009/
AndreiAlexandrescu_iterators-must-go.pdf

Problem

How to implement a containers and algorithms library for CDJ#++?

When *m* algorithms are coupled with *n* containers,
there must be $m \cdot n$ algorithm implementations.

When *m* algorithms and *n* containers are decoupled,
there must only be $m + n$ algorithm implementations.

STL Iterators

Alexander Stepanov: *The Standard Template Library* (1994/1995)

The question STL asked was:
What's the minimum an algorithm could ever ask
from the topology of the data it's operating on?

Building on C++’s pointers:

- brilliant strategic move helping rapid adoption
- code that used iterators looked like nicely crafted code using pointers

Example: the Essence of Linear Search

Find *val* in the range [*first*, *last*):

```
template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last, const T& val)
{
    while (first != last) {
        if (*first == val) return first;
        ++first;
    }
    return last;
}
```

Problems with STL Iterators

Ad-hoc pairing

- iterator-based code is not composable

Lack of safety

- abstractions from pointers come with pointers' specific problems

Adobe and Boost independently defined an abstraction called *range* that pairs two iterators together.

Gang of Four (GoF) Iterator

GoF: *Design Patterns: Elements of Reusable Object-Oriented Software* (1994)

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Sample Code

Iterator interface:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Ranges are Hot, Iterators are Not

Let's adopt and improve on both the STL's and the GoF iterator ideas.

Instead of building upon pointers as the fundamental abstraction,
as does the STL, it's better to start with the GoF approach.

Depending on your language of choice, you could use explicit interfaces
but also implicit interfaces and duck typing.

D’s Slices

https://tour.dlang.org/tour/en/basics/slices

A slice (aka dynamic array) consists of two members:

- a pointer to the first element
- the length of the slice

Using the *[begin .. end]* syntax a sub-slice is constructed from an existing slice, which contains all elements from *begin* to the element before *end*.

Example: the Essence of Linear Search

Find a *needle* in a *haystack* slice:

```
auto find(T)(T[] haystack, T needle)
{
    while (haystack.length != 0 && haystack[0] != needle)
    {
        haystack = haystack[1 .. $];
    }
    return haystack;
}
```

Example: the Essence of Linear Search

Find a *needle* in a *haystack* range:

```
auto find(Range, T)(Range haystack, T needle)
{
    while (!haystack.empty && haystack.front != needle)
    {
        haystack.popFront;
    }
    return haystack;
}
```

D’s Uniform Function Call Syntax (UFCS)

https://tour.dlang.org/tour/en/gems/uniform-function-call-syntax-ufcs

The following expressions are equivalent:

```
foo(bar(a))
a.bar().foo()  // member function syntax
a.bar.foo      // Optional Parentheses
```

Examples: *fluent interface, internal DSL*

```
"42".to!int
42.seconds
```

Adapting Slices

Use free functions to retrofit the missing member functions:

```
void popFront(T)(ref T[] a) { a = a[1 .. $]; }

bool empty(T)(T[] a) { return !a.length; }

T front(T)(T[] a) { return a[0]; }
```

Higher-Order Ranges

Pragmatic definition: the range equivalent of higher-order functions.

Examples found in many functional programming languages:

- *map*
- *filter*
- *fold / reduce*

Building ranges that decorate other ranges is easy, useful, and fun.

map Function

```
template map(alias fun)
{
    auto map(Range)(Range range)
    {
        return MapResult!(fun, Range)(range);
    }
}
```

map Range

```
struct MapResult(alias fun, Range)
{
    Range _range;

    this(Range input) { _range = input; }

    void popFront() { _range.popFront(); }

    bool empty() { return _range.empty; }

    auto front() { return fun(_range.front); }
}
```

filter Function

```
template filter(alias predicate)
{
    auto filter(Range)(Range range)
    {
        return FilterResult!(predicate, Range)(range);
    }
}
```

filter Range

```
struct FilterResult(alias pred, Range)
{
    Range _range;

    this(Range range)  { _range = range;
        while (!_range.empty && !pred(_range.front)) { _range.popFront(); }
    }

    void popFront()  {
        do { _range.popFront(); } while (!_range.empty && !pred(_range.front));
    }

    ...
}
```

Laziness to Infinity and Beyond

What if *empty* is never *true*?

Fibonacci Numbers:
http://tour.dlang.org/tour/en/basics/ranges

Where is the container?

... in Education

```
struct FibonacciRange
{
    int prev = 0;
    int next = 1;

    enum empty = false;

    void popFront() {
        int next = prev + current;
        prev = current; current = next;
    }

    int front() { return current; }
}
```

... in Practice

```d
import std.range : recurrence;
```

```d
auto fib = recurrence!((a, n) => a[n - 2] + a[n - 1])(1, 1);
```

or

```d
auto fib = recurrence!"a[n - 2] + a[n - 1]"(1, 1);
```

Conclusions

This article describes ranges — an iteration device that combines the safety, ease of definition, and ease of use of GoF iterators on one hand, with the unparalleled expressive power of STL iterators on the other.

Ranges offer simple definition and use, foster lazy computation without contortions, and offer interesting new opportunities.

Effective D

Scott Meyers: *Effective STL* (2001)

Item 43:
Prefer algorithm calls to hand-written loops.