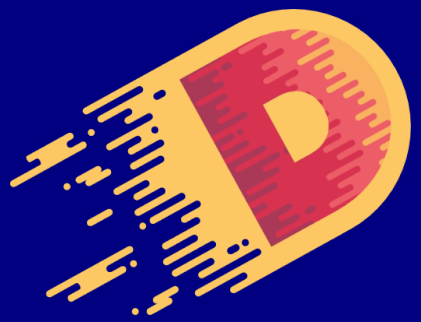
Reworking the Range API for Phobos v3

by Jonathan M Davis



Ranges in Phobos v2

- Ranges have been a great success story with Phobos v2.
- We want to continue that with Phobos v3.
- Over the years, we have found a variety of issues with ranges which we would like to take the opportunity to fix.
- The planned changes are iterative, not drastic.

What Are Ranges?

- D's answer to C++'s iterators.
- Sequences / lists / ranges of elements.
- An abstraction for dynamic arrays / slices.

Categories of Ranges

- random-access ranges
- bidirectional ranges
- forward ranges
- basic input ranges

Basic Range API

```
bool empty();
T front();
void popFront();
```

Basic Range API

```
bool empty();
T front();
void popFront();
```

```
typeof(this) save();
```

Basic Range API

```
bool empty();
T front();
void popFront();
```

```
typeof(this) save();
```

```
T back();
void popBack();
```

Basic Range API

```
bool empty();
T front();
void popFront();
```

```
typeof(this) save();
```

```
T back();
void popBack();
```

```
size_t length;
T opIndex(size_t);
```

Static Checks vs Behavioral Requirements

- A range must have front, popFront, and empty with the correct signatures.
- front and back must return the same type.
- Random-access ranges must have length which evaluates to size_t.

Static Checks vs Behavioral Requirements

- front must return the same value every time until popFront is called.
- Two independent copies of a range must contain the same elements in the same order.
- All range API functions must be O(1).

Problems with the Current API

- Auto-decoding
- save
- Underspecified / unspecifiable behavioral requirements

Auto-decoding

- Auto-decoding is an attempt to ensure Unicode-correctness by default in D.
- Arrays of char and wchar are ranges of dchar.
- UTF-8 (char) and UTF-16 (wchar) are variable-length encodings.
- Accessing individual indices of char[] and wchar[] risks getting garbage.

Why Auto-decoding Has Failed

1. You do not get Unicode correctness by default; you still need to understand Unicode to get correct results.

2. If you do understand Unicode, the way that auto-decoding solves the problem just gets in the way and makes it harder to write correct code that is performant.

3. It complicates Phobos considerably because of all the code that tries to work around it.

save

- Provides a way to get an independent copy of a forward range.
- Required for types where copying them does not result in an independent copy.
- Frequently forgotten.

Copy Semantics

1. Value types
2. Reference types
3. Pseudo-reference types whose iteration state has value semantics
4. Pseudo-reference types whose iteration state does *not* have value semantics

Copy Semantics

```d
auto copy = orig;
orig.popFront();
// What is the state of copy.front here?
```

```d
auto copy = orig;
copy.popFront();
// What is the state of orig.front here?
```

```d
auto copy = orig.save;
orig.popFront();
```

Copy Semantics

```
foreach(e; range)
{
    // ...
    if(foo)
        break;
}
// range must be considered invalid.
```

Copy Semantics

```d
foreach(e; range)
{
    // ...
    if(foo)
        break;
}
// range must be considered invalid.
```

```d
for(auto __c = range; !__c.empty; __c.popFront())
{
    auto e = __c.front;
    // ...
    if(foo)
        break;
}
// range must be considered invalid.
```

Assignment Semantics

```d
auto copy = orig.save;
copy.popFront();
orig = copy;
// What relation do orig and copy now have?
// copy must no longer be used.
```

init Poorly Defined

There is no guarantee that init is valid, let alone empty.

```
struct Range
{
    int front() { return 42; }
    void popFront() {}
    bool empty() { return true; }
}

void main()
{
    import std.range;
    assert(chain(Range.init,
                 Range.init).init.empty);
}
```

Empty Ranges

There is no way to get an empty range from a range in O(1) except via slicing.

```
void foo(R)(ref R range)
{
    // ...
    if(range.front == 42)
    {
        range = R.init;
        return;
    }
    // ...
}
```

Transient Front

The range API does not specify what happens if you call popFront after copying front.

```
auto f = range.front;
range.popFront();
// Is f unchanged?
```

A Prime example that causes problems with this would be buffer reuse.

Non-Copyable Types

Range-based functions tend to ignore that non-copyable types are a thing.

```
auto range = [NonCopyable.init, NonCopyable.init];
// Error
auto f = range.front;
```

```
// Error
auto range = only(NonCopyable.init,
                  NonCopyable.init);
```

Random-Access Ranges and Slicing

- Random-access ranges do not require slicing, and forward ranges are allowed to have slicing.
- $ is not required for either indexing or slicing.
- $ cannot be used for either indexing or slicing.

const and Ranges Don't Mix

- const elements are fine.
- popFront doesn't work on a const range.
- There is no way to get a mutable range of const elements from a const range.
- Slicing arrays does give you a tail-const slice, but this is special to arrays, and user-defined types cannot emulate it.

Import Required for Arrays

- In order to use arrays as ranges, you must import std.range.primitives (or std.range).

Proposed Changes: No Autodecoding

- All arrays will be treated as ranges of their actual element type - i.e. no auto-decoding.
- decode / decodeFront can be called on arrays and ranges to explicitly decode code points.
- byUTF, byChar, and etc. will allow you to get ranges of each character type.
- Choice between the replacement character and UTFExceptions.
- Phobos v3 will mostly stick to ranges of char and not support ranges of wchar or dchar.

init Must Be Valid

- If a range can be default-initialized, its init value must be valid.
- If a range is infinite, it's allowed to disable default initialization.
- We may or may not allow finite ranges which disable default initialization.
- If a finite range can be default-initialized, its init value must be empty.
- The current plan is to allow finite ranges which disable default initialization but require them to define emptyRange to provide an empty range.

Dynamic Arrays or Structs

- All ranges must be dynamic arrays or structs.
- No pointers, no classes.
- Pointers and classes must be wrapped by structs.
- This allows for consistent copy and assignment semantics.

No save

- save will no longer be part of the range API.
- All forward ranges must have copy semantics which make each copy independent.
- Copying a forward range does not need to be implemented in the same way as save (e.g. it could use ref-counting), but the semantics are effectively the same.

Basic Input Ranges are Non-Copyable

- Forward ranges are defined by their ability to be copied.
- In order to differentiate between forward ranges and basic input ranges, basic input ranges must be non-copyable.
- Bugs related to partial copies will be eliminated.
- Functions designed for basic input ranges must either use ref or require move.

Assignment Semantics

- The assignment semantics must be the obvious semantics which match the copy semantics.
- Assigning to forward ranges results in replacing the lhs with an independent copy of the rhs.
- Assigning to a basic input range only works in cases where it's a move; otherwise, an explicit move will be required.

Transient front Is Not Allowed

- It is a requirement that if front can be copied, calling popFront does not affect the copy.
- In some cases, opApply may need to be used instead.
- In some cases, a solution may involve using non-copyable elements.
- In some cases, reference-counting may be a good solution.

Non-Copyable Elements Will Be Supported

- We will have the necessary traits for non-copyable elements.
- Algorithms should test for non-copyable elements where appropriate.

```
__traits(isCopyable, ElementType!R)
```

```
hasCopyableElements!R
```

Tail-const Ranges

- The range API will not directly solve the problem.
- Walter has a DIP that tries to solve the problem more generically.

Range API Function Names

- We cannot add the range functions to object.d with their current names because of import conflicts.
- The new range API needs to not have the old basic input ranges look like the new forward ranges.
- A DIP is required.

Range API Function Names

- front -> first
- popFront -> popFirst
- empty -> isEmpty
- back -> last
- popBack -> popLast

Implementations in Phobos

- Traits and basic helper functions.
- Wrappers for classes as well as updated implementations for std.range.interfaces.
- Wrappers for ranges to convert between the old and new API.
- Test helpers - both to help ensure that a range type has the correct behavior and that a range-based function works with various combinations of ranges.

Documentation

- Basic introduction to ranges for the average user.
- Clear documentation on the behavioral requirements that come with the range API.
- Clear documentation on the assumptions that can be made with code that correctly implents the range API.

Questions?