# Introduction to Ranges

by Jonathan M Davis

# Dynamic Arrays

```
T[]


struct DynamicArray(T)
{
    size_t length;
    T* ptr;
}
```

# Dynamic Arrays

```
void foo(int[] arr) // Original array is sliced
{
    arr[0] = 2; // Mutates original array
    arr ~= 19;  // Does _not_ mutate original array
    arr = new int[](5);  // Does _not_ mutate original array
}

void bar(ref int[] arr) // arr _is_ the original array
{
    arr[0] = 2;  // Mutates original array
    arr ~= 19;   // Mutates original array
    arr = new int[](5);  // Mutates original array
}
```

# Dynamic Arrays

- Dynamic arrays do not know or care who owns or manages their memory.
- The runtime does not keep track of when arrays are sliced or when they are assigned new values.
- The _only_ time that the runtime keeps track of dynamic arrays is when it does a garbage collection.
- The only dynamic array operations which involve the GC involve capacity, concatenation, and/or appending.
- Dynamic arrays are not containers.

# Dynamic Arrays

- Do _not_ let a dynamic array which is a slice of a static array escape the scope of the static array.
- Do _not_ free malloc-ed memory when a dynamic array exists which refers to that memory.
- Beware of implicit slicing of static arrays.

# Iterators

```
auto iter = find(list.begin(), list.end(), value);
auto wasFound == iter != list.end();

if(!wasFound)
    return;

auto element = *iter;
++iter;
auto iter2 = find(iter, list.end(), value2);
auto wasFound2 = iter2 != list.end();
```

# Iterators

- Abstraction for a pointer.
- Refer to a single element.
- Require two iterators for algorithms.
- Unwieldy for chaining algorithms.

# Ranges

```
auto range = find(list[], value);
auto wasFound == !range.empty;

if(!wasFound)
    return;

auto element = range.front;
range.popFront();
auto range2 = find(range, value2);
auto wasFound2 = !range2.empty;
```

# Ranges

- Abstraction for a dynamic array (sort of).
- Refer to several elements - a range of elements.
- Require a single range for algorithms.
- Chain very well for algorithms.

# Ranges

- Input Range
- Forward Range
- Bidirectional Range
- Random-Access Range

# Input Ranges

```
template isInputRange(R)
{
    enum bool isInputRange = is(typeof(
    (inout int = 0)
    {
        R r = R.init;     // can define a range object
        if (r.empty) {}   // can test for empty
        r.popFront();     // can invoke popFront()
        auto h = r.front; // can get the front of the range
    }));
}
```

# Input Ranges

```
R r;                  // can define a range object

if (r.empty) {}       // can test for empty

r.popFront();         // can invoke popFront()

auto h = r.front; // can get the front of the range of
                  // non-void type
```

# Input Ranges

```
struct IterWrapRange(Iter)
{
public:
    @property auto front() { return *_iter; }
    @property bool empty() { return _iter == _end; }
    void popFront() { ++_iter; }
private:
    Iter _iter;
    Iter _end;
}
```

# Input Ranges

```
struct CountTo10
{
public:
    @property int front() { return _currVal; }
    @property bool empty() { return _currVal == 10; }
    void popFront() { ++_currVal; }
private:
    int _currVal = 0;
}
```

# Input Ranges

```
auto count(int max)
{
    static struct Result
    {
    public:
        @property int front() { return _currVal; }
        @property bool empty() { return _currVal == _max; }
        void popFront() { ++_currVal; }
    private:
        int _currVal;
        int _max;
    }
    return Result(0, max);
}
```

# Input Ranges

```
foreach(e; range)
{
    // do stuff
}

for(auto __c = range; !__c.empty; __c.popFront())
{
    auto e = __c.front;
    // do stuff
}
```

# Input Ranges

```
auto range = list[];


foreach(e; list[])
{
    // do stuff
}


foreach(e; list)
{
    // do stuff
}
```

# Dynamic Arrays as Ranges

- Dynamic arrays are ranges.
- The range functions that dynamic arrays do not natively define are in std.array.
- Dynamic arrays are _not_ containers.
- Static arrays are not ranges.
- opSlice with no arguments is _not_ a range function. It is a container function for supporting ranges.
- Concatenation and appending are _not_ range operations.

# Ranges

```
auto arr = array(take(
           map!(a => cast(int)(a % 100))(rndGen()), 10));

auto arr = rndGen().map!(a => cast(int)(a % 100))().
           take(10).array();

auto arr = rndGen().
           map!(a => cast(int)(a % 100))().
           take(10).
           array();
```

# Ranges

```
auto file = File("myFile.txt");
foreach(line; file.byLine())
{
    auto s = line.splitter(", ");
    if(s.empty)
        writeln();
    else
    {
        static auto parenMe(char[] a) { return format("(%s)", a); }
        writefln("%s: %s", s.front, map!parenMe(drop(s, 1)));
    }
}
```

# Ranges

- There is no way to “un”-pop an element.
- Semantics of copying ranges are undefined.

# Ranges

```
auto secondRef = refRange;
auto secondValue = valueRange;
auto secondPseudo = pseudoRange;

refRange = secondRef;
valueRange = secondValue;
pseudoRange = secondPseudo;

foo(refRange);
foo(valueRange);
foo(pseudoRange);
```

# Forward Ranges

```
template isForwardRange(R)
{
    enum bool isForwardRange = isInputRange!R &&
is(typeof(
    (inout int = 0)
    {
        R r1 = R.init;
        static assert (is(typeof(r1.save) == R));
    }));
}
```

# Forward Ranges

```
static assert(isInputRange!R);
R r1;
static assert(is(typeof(r1.save) == R));
```

# Forward Ranges

```
struct CountTo10
{
public:
    @property int front() { return _currVal; }
    @property bool empty() { return _currVal == 10; }
    void popFront() { ++_currVal; }
    @property auto save() { return *this; }
private:
    int _currVal = 0;
}
```

# Forward Ranges

```
class CountTo10
{
public:
    @property int front() { return _currVal; }
    @property bool empty() { return _currVal == 10; }
    void popFront() { ++_currVal; }
    @property auto save()
    { return new CountTo10(_currVal); }
private:
    int _currVal = 0;
}
```

# Forward Ranges

```
auto orig = range.save;
assert(equal(range.save, [5, 7, 42, 99]));
assert(equal(orig.save, [5, 7, 42, 99]));

range.popFront();
range.popFront();
assert(equal(range.save, [42, 99]));
assert(equal(orig.save, [5, 7, 42, 99]));

range = orig.save;
```

# Forward Ranges

- save is like slicing a whole array.
- Calls to save are frequently forgotten.
- Most ranges are either dynamic arrays or structs.
- Make sure that you test your range-based functions with a variety of range types - including reference types.

# Bidirectional Ranges

```
template isBidirectionalRange(R)
{
    enum bool isBidirectionalRange = isForwardRange!R && is(typeof(
    (inout int = 0)
    {
        R r = R.init;
        r.popBack();
        auto t = r.back;
        auto w = r.front;
        static assert(is(typeof(t) == typeof(w)));
    }));
}
```

# Bidirectional Ranges

```
static assert(isForwardRange!R);
R r1;
r.popBack();
auto t = r.back;
auto w = r.front;
static assert(is(typeof(t) == typeof(w)));
```

# Range Traits

- isInputRange
- isForwardRange
- isBidirectionalRange
- isRandomAccessRange

# Range Traits

```
R find(alias pred = "a == b", R, E)
      (R haystack, E needle)
if (isInputRange!R &&
    is (typeof(binaryFun!pred(haystack.front, needle)) : bool))
{
    ...
}
```

# Range Traits

- ElementType
- hasLength
- hasSlicing
- isInfinite
- hasSwappableElements
- hasAssignableElements
- hasMobileElements // moveFront, moveBack, moveAt
- hasLvalueElements

# Range Traits

```
int[] arr;
alias R = typeof(arr);
static assert(ElementType!R == int);
static assert(hasLength!R);
static assert(hasSlicing!R);
static assert(!isInfinite!R);
```

# Range Traits

```
int[] arr;
auto range = filter!(a => a < 42)(arr);
alias R = typeof(range);
static assert(ElementType!R == int);
static assert(!hasLength!R);
static assert(!hasSlicing!R);
static assert(!isInfinite!R);

auto len = walkLength(range.save);
auto listLen = walkLength(list[]);
```

# Random-Access Ranges

```
template isRandomAccessRange(R)
{
    enum bool isRandomAccessRange = is(typeof(
    (inout int = 0)
    {
        static assert(isBidirectionalRange!R ||
                      isForwardRange!R && isInfinite!R);
        R r = R.init;
        auto e = r[1];
        static assert(is(typeof(e) == typeof(r.front)));
        static assert(!isNarrowString!R);
        static assert(hasLength!R || isInfinite!R);

        static if(is(typeof(r[$])))
        {
            static assert(is(typeof(r.front) == typeof(r[$])));

            static if(!isInfinite!R)
                static assert(is(typeof(r.front) == typeof(r[$ - 1])));
        }
    }));
}
```

# Random-Access Ranges

```
static assert(isBidirectionalRange!R ||
              isForwardRange!R && isInfinite!R);
R r = R.init;
auto e = r[1];
static assert(is(typeof(e) == typeof(r.front)));
static assert(!isNarrowString!R);
static assert(hasLength!R || isInfinite!R);
```

# Strings

- Code Units
  - char (UTF-8), wchar (UTF-16), dchar (UTF-32)
  - pieces of characters
- Code Points
  - dchar
  - mostly whole characters
- Graphemes
  - Whole characters

# Strings

```
assert(`Ma Chérie`.length == 10);
assert(`Ma Chérie`w.length == 9);
assert(`Ma Chérie`d.length == 9);

assert(`さいごの果実 / ミツバチと科学者`.length == 45);
assert(`さいごの果実 / ミツバチと科学者`w.length == 17);
assert(`さいごの果実 / ミツバチと科学者`d.length == 17);

assert("\U00010143\u0100\U00010143 hello".length == 16);
assert("\U00010143\u0100\U00010143 hello"w.length == 11);
assert("\U00010143\u0100\U00010143 hello"d.length == 9);
```

# Strings

```
assert(walkLength(`Ma Chérie`) == 9);
assert(walkLength(`Ma Chérie`w) == 9);
assert(walkLength(`Ma Chérie`d) == 9);

assert(walkLength(`さいごの果実 / ミツバチと科学者`) == 17);
assert(walkLength(`さいごの果実 / ミツバチと科学者`w) == 17);
assert(walkLength(`さいごの果実 / ミツバチと科学者`d) == 17);

assert(walkLength("\U00010143\u0100\U00010143 hello") == 9);
assert(walkLength("\U00010143\u0100\U00010143 hello"w) == 9);
assert(walkLength("\U00010143\u0100\U00010143 hello"d) == 9);
```

# Narrow Strings

```
foreach(S; TypeTuple!(string, wstring))
{
    static assert(!hasLength!S);
    static assert(!hasSlicing!S);
    static assert(!isRandomAccessRange!S);
    static assert(is(ElementType!S == dchar));
}
assert(is(ElementEncodingType!string == immutable char));
assert(is(ElementEncodingType!wstring == immutable wchar));
```

# Narrow Strings

```
auto s = `さいごの果実 / ミツバチと科学者`;

assert(walkLength(s.save) == 17);

size_t count = 0;
foreach(c; s.save)
    ++count;
assert(count == 45);
```

# Strings

- Code Units
  - std.utf.byCodeUnit
  - std.utf.byChar / std.utf.byWchar
- Code Points
  - std.uni.byCodePoint
  - std.utf.byDchar
- Graphemes
  - std.uni.byGrapheme

# std.range

- chain
- drop / dropExactly
- iota
- lockstep / zip
- retro
- stride
- take / takeExactly / takeOne / takeNone
- popFrontN

# std.array

- array
- join
- replace
- replicate
- split

# std.algorithm

- equal
- filter
- joiner
- map
- reduce
- splitter
- remove
- strip, stripLeft, stripRight

# std.algorithm

- all / any
- count / countUntil
- find / canFind
- findSplit / findSplitBefore / findSplitAfter
- startsWith / endsWith
- until
- sort

# String Algorithms

- std.algorithm
- std.array
- std.range
- std.string

# Removing from Containers

```
auto range = list[];
auto found = range.find(42);
list.remove(found); // Remove everything starting with 42

list.remove(found.take(5));
```

# Output Ranges

```
template isOutputRange(R, E)
{
    enum bool isOutputRange = is(typeof(
    (inout int = 0)
    {
        R r = R.init;
        E e = E.init;
        put(r, e);
    }));
};
```

# Output Ranges

- Use lazy input ranges.

# Questions