CSE-491
range-v3 examples

cgnitash@msu.edu

April 1, 2019

1 Introduction

Hopefully, y'all have already watched the ranges talk by Eric Niebler, and are as excited as I am about how cool it is. However, like most new paradigms, it takes practise to get used to it. And I have a hunch that ranges are going to be the most radical change ever to how we code C++, and so below are some problem sets that give some practise with ranges, and also gives us some concrete examples of usage that we can discuss in class. The problems are listed in increasing order of difficulty. Actually, this may not be the case for all of you, so feedback would be great.

2 Simple algorithms

To start off, let's implement the roulette-wheel selection approach that I posted last week. There are obviously many different approaches, but here's a fairly straightforward one.

```
struct Agent {
  double score = 0;
  double get_score() { return score; }
};

std::vector<Agent> roulette_wheel(std::vector<Agent> &agents,
                                  int new_population_size) {

  std::vector<double> wheel;
  // get all the scores
  std::transform(std::begin(agents), std::end(agents),
                 std::back_inserter(wheel),
                 [](auto agent) { return agent.get_score(); });
  // create the roulette wheel
  std::partial_sum(std::begin(wheel), std::end(wheel),
                   std::begin(wheel));
```

```
  // 'wheel' has the nice property that a randomly generated
  // double will fall on an index of wheel, with a probability
  // proportional to the fitness of an agent at that index.
  // (Sketch an example to see that).
  // O(n) complexity

  // create the random spinner, and uniformly distributed tip
  std::mt19937 spinner;
  std::uniform_real_distribution<double> tip(0, wheel.back());

  // croupier spins the wheel and selects an agent
  // every call to this function will select an agent
  // fitness-proportionately
  // O(log(n)) complexity
  auto croupier = [&] {
    return agents[std::distance(
        std::begin(wheel),
        std::lower_bound(std::begin(wheel), std::end(wheel),
                         tip(spinner)))];
  };

  // croupier spins the wheel N times to generate next population
  // O(n.log(n)) complexity
  std::vector<Agent> next_agents;
  std::generate_n(std::back_inserter(next_agents),
                  new_population_size, croupier);

  return next_agents;
}
```

Notice, no explicit loops, or conditionals. Convert this implementation to one that uses ranges. There are several advantages that can be had

1. improved legibility with not having to say .end(), and .begin()
2. projection functions to reduce lambda boilerplate
3. piping to improve legibility
4. not needing to name variables unnecessarily.
5. Also, wheel should be const, which is not possible without ranges (as far as I can tell).

Try to take advantage of all these points.

3 1D Matrix

Here's a classic; writing a 2D-Matrix class that stores its data in a 1D data structure. We're going to push this further, and simply store a view to the data, i.e. the matrix will be a non-mutable view on data!

```
#include <range/v3/all.hpp>
#include // all necessary headers
using namespace ranges;

struct Matrix {

  size_t rows_=0, cols_=0;
  // just any view on ints
  any_view<int> v_;

  // default constructor just stores a view to counted ints
  Matrix(size_t rows, size_t cols, size_t init)
    : rows_(rows) , cols_(cols) ,
      v_(view::iota(init, init + rows_*cols_)) {}

  // useful constructor
  Matrix(size_t rows, size_t cols, any_view<int> av)
     : rows_(rows) , cols_(cols) , v_(av) {}

  // I'll wrtite the easy one :)
  Matrix operator+(Matrix m) {
    assert(rows_ == m.rows_);
    assert(cols_ == m.cols_);
    return {rows_, cols_,
            view::zip_with(std::plus<int>{}, v_, m.v_)};
 }

  // return a transposed matrix
  // the ranges talk uses interleave(), which is not a part of
  // the range-v3 library, but this is not difficult to write
  // for the case when we know both dimensions
  // hint: check out view::stride
  Matrix tr() {
      // ...
  }

  // this might take some work!
  Matrix operator*(Matrix m) {
```

```
    assert(cols_ == m.rows_);
    // ...
};

int main() {
  auto m1 = Matrix(3,4,1);
  std::cout << m1.v_;

  any_view<int> v = view::iota(100,112);
  auto m2 = Matrix(3,4,v);
  std::cout << m2.v_;

  auto m3 = m1 + m2;
  std::cout << m3.v_;

 // ...
 // Try other matrix operations
 // Try describing some common matrices,
 // e.g. use view::slide to create the identity matrix
}
```

Try implementing the functions mentioned above. This is just a taste of how much of a change ranges is going to be.

4 Patterns

When I first learned programming, a large component of the first semester was drawing patterns, e.g.

```
> ./a.out 4
*
**
***
****

> ./a.out 4
dcba
 cba
  ba
   a
```

,etc. I'm sure you get the idea. A lot of us disliked this, because we didn't see the point of it (who needs to draw so many patterns?). Also, we disliked it even more, because we had to write it on paper, not a computer!. However, the purpose of that was to develop familiarity with loop structures. In that regard at least, it served its purpose; I can write a for-loop

without really thinking too hard about the loop conditions. In the same vein, here's some pattern practise with ranges. Write a program that generates the following pattern

```
> ./a.out 4
                    a
            a       aba       a
      a     aba     abcba     aba     a
  a   aba   abcba   abcdcba   abcba   aba   a
      a     aba     abcba     aba     a
            a       aba       a
                    a

> ./a.out 5
                              a
                    a         aba         a
            a       aba       abcba       aba       a
      a     aba     abcba     abcdcba     abcba     aba     a
  a   aba   abcba   abcdcba   abcdedcba   abcdcba   abcba   aba   a
      a     aba     abcba     abcdcba     abcba     aba     a
            a       aba       abcba       aba       a
                    a         aba         a
                              a
```

The pattern should be clear. I assume everyone can write the loops necessary to draw it, and can also see that it would be quite fiddly. Implement this program without using any explicit loops, or conditionals. In fact, if you try hard, you should be able to do this without any ranges::actions or ranges algorithms, but only using ranges::views. If you try really hard, you could do this without strings at all (except when printing to the screen; I couldn't find a way around it). Obviously, don't try solving the whole problem at once; build it up piece by piece (correct by construction, as Eric puts it in his talk).