DNA Matching

The lesson explains the use and working of searcher functions using the DNA structure.

To demonstrate the range of uses for `std::search`, let’s have a look at a simple DNA matching demo. The example will match custom types rather than regular characters.

For instance, we’d like to search a DNA sequence to see whether `GCTGC` occurs in the sequence `CTGATGTTAAGTCAACGCTGC`.

The code block uses a simple data structure for Nucleotides:

```
struct Nucleotide {
    enum class Type : uint8_t {
        A = 0,
        C = 1,
        G = 3,
        T = 2
    };

    Type mType;

    friend bool operator==(Nucleotide a, Nucleotide b) noexcept {
        return a.mType == b.mType;
    }

    static char ToChar(Nucleotide t);
    static Nucleotide FromChar(char ch);
};
```

With the two converting static methods:

```
char Nucleotide::ToChar(Nucleotide t) {
    switch (t.mType) {
    case Nucleotide::Type::A: return 'A';
    case Nucleotide::Type::C: return 'C';
    case Nucleotide::Type::G: return 'G';
    case Nucleotide::Type::T: return 'T';
    }
    return 0;
}

Nucleotide Nucleotide::FromChar(char ch) {
    return Nucleotide { static_cast<Nucleotide::Type>((ch >> 1) & 0x03) };
}
```

And the two functions that work on a whole string:

```
std::vector<Nucleotide> FromString(const std::string& s) {
    std::vector<Nucleotide> out;
    out.reserve(s.length());
    std::transform(std::cbegin(s), std::cend(s),
                   std::back_inserter(out), Nucleotide::FromChar);
    return out;
}

std::string ToString(const std::vector<Nucleotide>& vec) {
    std::stringstream ss;
    std::ostream_iterator<char> out_it(ss);
    std::transform(std::cbegin(vec), std::cend(vec), out_it, Nucleotide::ToChar);
    return ss.str();
}
```

The demo uses `boyer_moore_horspool_searcher` which requires hashing support. So we have to define it as follows:

```
namespace std {
    template<> struct hash<Nucleotide> {
        size_t operator()(Nucleotide n) const noexcept {
            return std::hash<Nucleotide::Type>{}(n.mType);
        }
    };
}
```

`std::hash` has support for enums, so we just have to “redirect” it from the whole class.

And then the test code:

```
const std::vector<Nucleotide> dna = FromString("CTGATGTTAAGTCAACGCTGC");
std::cout << ToString(dna) << '\n';
const std::vector<Nucleotide> s = FromString("GCTGC");
std::cout << ToString(s) << '\n';

std::boyer_moore_horspool_searcher searcher(std::cbegin(s), std::cend(s));
const auto it = std::search(std::cbegin(dna), std::cend(dna), searcher);

if (it == std::cend(dna))
    std::cout << "The pattern " << ToString(s) << " not found\n";
else {
    std::cout << "DNA matched at position: "
              << std::distance(std::cbegin(dna), it) << '\n';
}
```

Now that you know all the bits and pieces, here is the entire code for you to execute:

```
#include <iostream>
#include <string>
#include <algorithm>  // std::search
#include <vector>
#include <functional> // searchers
#include <iterator>
#include <sstream>

// A 65 01000 | 00 | 1  0
// C 67 01000 | 01 | 1  1
// G 71 01000 | 11 | 1  3
// T 84 01010 | 10 | 0  2
struct Nucleotide {
    enum class Type : uint8_t {
        A = 0,
        C = 1,
        G = 3,
        T = 2
    };

    Type mType;

    friend bool operator==(Nucleotide a, Nucleotide b) noexcept {
        return a.mType == b.mType;
    }

    static char ToChar(Nucleotide t);
    static Nucleotide FromChar(char ch);
};

namespace std {
    template<> struct hash<Nucleotide> {
        size_t operator()(Nucleotide n) const noexcept {
            return std::hash<Nucleotide::Type>{}(n.mType);
        }
    };
}

char Nucleotide::ToChar(Nucleotide t) {
    switch (t.mType) {
    case Nucleotide::Type::A: return 'A';
    case Nucleotide::Type::C: return 'C';
    case Nucleotide::Type::G: return 'G';
    case Nucleotide::Type::T: return 'T';
    }
    return 0;
}

Nucleotide Nucleotide::FromChar(char ch) {
    return Nucleotide { static_cast<Nucleotide::Type>((ch >> 1) & 0x03) };
}

std::vector<Nucleotide> FromString(const std::string& s) {
    std::vector<Nucleotide> out;
    out.reserve(s.length());
    std::transform(std::cbegin(s), std::cend(s),
                   std::back_inserter(out), Nucleotide::FromChar);
    return out;
}

std::string ToString(const std::vector<Nucleotide>& vec) {

    std::stringstream ss;
    std::ostream_iterator<char> out_it(ss);
    std::transform(std::cbegin(vec), std::cend(vec), out_it, Nucleotide::ToChar);
    return ss.str();
}

int main() {
    const std::vector<Nucleotide> dna = FromString("CTGATGTTAAGTCAACGCTGC");
    std::cout << ToString(dna) << '\n';
    const std::vector<Nucleotide> s = FromString("GCTGC");
    std::cout << ToString(s) << '\n';

    std::boyer_moore_horspool_searcher searcher(std::cbegin(s), std::cend(s));
    const auto it = std::search(std::cbegin(dna), std::cend(dna), searcher);

    if (it == std::cend(dna))
        std::cout << "The pattern " << ToString(s) << " not found\n";
    else {
        std::cout << "DNA matched at position: "
                  << std::distance(std::cbegin(dna), it) << '\n';
    }
}
```

As you can see, the example builds a vector of custom types - Nucleotides. To satisfy the searcher, a custom type needs to support `std::hash` interface and also define `operator==`.

The Nucleotide type wastes a bit of space - as we use the full byte just to store four options - C T G A. We could use only 2 bits, though the implementation would be more complicated. Another option is to represent the triplets of Nucleotides - Codons. Each codon can be expressed in 6 bits, so that way we'd use the full byte more efficiently.

And that's a wrap! Let's move to a quick summary of whatever we have covered in this module.