

Project information: README.md

CISC-310: Lab 2

- Author: Nils Olsson
- Semester: SDSU Fall 2019

About the program

- Uses [cxxopts], an open-source command-line option parser that facilitates building robust command-line behavior easily. (I wouldn't include this if I thought it impacted any part of the project significantly, it just makes the end-user experience richer.)
- For help and usage, invoke with the `-h` optional flag.

[cxxopts]: <https://github.com/jarro2783/cxxopts>

About the project structure

This project structure was originally adapted from [cmake-project-template][cpt] for a project from last year, [acm-cpp]. The structure is split into two parts, the `src` directory which contains all source-code for the project proper, and the `test` directory which includes the [googletest][gt] library and unit-test source files.

(Although I never really got around to implementing any unit-tests for the decoder.)

There's also a folder, `samples`, of the sample code and sequence files which the professor provided.

[cpt]: <https://github.com/kigster/cmake-project-template>

[acm-cpp]: <https://github.com/nilsso/acm-cpp>

[cmake]: <https://cmake.org/>

[gt]: <https://github.com/google/googletest>

Acquiring and building

****(Note: cloning wont work since the repository is currently private)****

To clone this project along with the [googletest][gt] submodule:

```
```bash
git clone --recursive git://github.com/nilsso/cisc310-lab2
```
```

Then from the command-line, make a directory called `build` in the root of the project, move into it, invoke `cmake` with the previous directory as its argument, and invoke `make` with the generated Makefiles.

```
```bash
mkdir build
cd build
cmake ..
make
```
```

(Additionally, running `make install` will move the built binaries, built libraries, and header

files into these folders in the project root respectively: ``bin`, `lib`, `include``. Resetting the project structure to a clean state after having "installed" is just a matter of deleting these directories: ``rm -rf {bin,lib,include}``.)

Running

The built binary will be located at ``build/src/lab2``.

Invoke with the ``-h`` flag for a full list of options to the program.

By default, attempts to read MILES code patterns from a file named ``miles.csv`` and sequences from a file named ``seqs.csv`` (these files can be changed through optional flags). Additional optional arguments are ``-n`` for normalizing input sequences, and ``-o <FILE>`` for specifying an output file into which found MILES codes will be written.

MILES code/sequence generator

Additional to the project, I wrote a quick Python script to help generate MILES code and sequences files for use in testing the main program. Similar to the main program, invoking ``./miles_generator.py -h`` will print a full list of options. But for example:

```
```bash
> ./miles_generator.py m 0 1 7

0,20,100,140,160,200,0
0,20,60,120,180,200,1
0,20,60,80,120,140,7

> ./miles_generator.py s 0 1 7

0,20,100,140,160,200,220,240,280,340,400,420,440,460,500,520,560,580
...`
```

Note that the values of the output sequence will always be sequential.

---

## Lab 1 driver program: main.cpp

---

```
// CISC-310: Lab 2 -- Nils Olsson
//
// Driver program in charge of:
// - Streaming data from a MILES CSV file and constructing a list of MILES code objects
// - Streaming data from a sequence CSV file into a list of sequences (queues)
// - Constructing a decoder from the list of MILES code objects
// - Using the decoder to decode the sequences and collecting found IDs
// - Outputting the found MILES code IDs in CSV format
#include <iostream>
#include <fstream>
#include <vector>

// Note for cxxopts (https://github.com/jarro2783/cxxopts):
// This is an open-source commandline option parser that facilitates
// building robust commandline behavior easily.
#include "cxxopts.hpp"

#include "util.hpp"
#include "queue.hpp"
#include "decoder.hpp"
```

```

// Facet extension for use in CSV file stream extraction
struct CSV_facet: std::ctype<char> {
 static const mask *make_table() {
 static std::vector<mask> v(
 classic_table(),
 classic_table() + table_size);
 v[' ', ''] = space;
 return &v[0];
 }

 CSV_facet(std::size_t refs = 0):
 ctype{make_table(), false, refs}
 {}
};

// Validate std::vector as code pattern
// @param pattern Vector to validate
bool validate_pattern(const std::vector<int> &pattern);

// Default file names (paths)
const std::string MILES_FILE_DEFAULT = "miles.csv";
const std::string SEQ_FILE_DEFAULT = "seq.csv";

// Entry point
int main(int argc, char *argv[]) {
 // Construct command-line arguments parser
 cxxopts::Options options = { argv[0], "CS 310: Lab 2" };
 options
 .add_options()
 ("h,help", "Print this help message")
 ("m,miles", "MILES pattern file",
 cxxopts::value<std::string>()>default_value(MILES_FILE_DEFAULT))
 ("s,sequence", "Value sequence file",
 cxxopts::value<std::string>()>default_value(SEQ_FILE_DEFAULT))
 ("n,normalize", "Normalize input sequences")
 ("o,output", "Output file for found MILES codes in CSV format",
 cxxopts::value<std::string>());
 auto result = options.parse(argc, argv);
 if (result.count("help")) {
 std::cout << options.help() << std::endl;
 return EXIT_SUCCESS;
 }

 // Get file paths
 std::string path_m = result["miles"].as<std::string>();
 std::string path_s = result["sequence"].as<std::string>();
 if (path_m == path_s) {
 std::cerr
 << "Error: MILES and sequence files cannot be the same."
 << std::endl;
 return EXIT_FAILURE;
 }

 // Open files
 std::ifstream ifs_m = std::ifstream(path_m, std::ios::in);
 std::ifstream ifs_s = std::ifstream(path_s, std::ios::in);
 if (!(ifs_m.is_open() && ifs_s.is_open())) {
 if (!ifs_m.is_open())

```

```

 std::cerr
 << "Error: failed to open MILES file \""
 << path_m << "\" to be read." << std::endl;
 if (!ifs_s.is_open())
 std::cerr
 << "Error: failed to open sequence file \""
 << path_s << "\" to be read." << std::endl;
 ifs_m.close();
 ifs_s.close();
 return EXIT_FAILURE;
}

// Variables for CSV reading
int temp;
std::string line;
std::istringstream iss;
iss.imbue(std::locale(iss.getloc(), new CSV_facet));

// Collect codes
std::vector<MILES::Code> codes;
while (ifs_m >> line) {
 iss.clear();
 iss.str(std::move(line));
 std::vector<int> pattern;
 while (iss >> temp)
 pattern.push_back(temp);
 if (!validate_pattern(pattern)) {
 std::cout
 << "Error: Malformed MILES code \"" << util::join(pattern) << "\"." << std::endl
 << " (MILES codes consist of 6 sequential numbers and an ID number.)" << std::endl;
 return EXIT_FAILURE;
 }
 int id = pattern.back();
 pattern.pop_back();
 codes.emplace_back(std::move(pattern), id);
}
ifs_m.close();

// Collect sequences
std::vector<queue<int>> sequences;
while (ifs_s >> line) {
 iss.clear();
 iss.str(std::move(line));
 queue<int> sequence;
 while (iss >> temp) {
 sequence.enqueue(temp);
 }
 if (result.count("normalize"))
 sequence.normalize();
 sequences.push_back(sequence);
}
ifs_s.close();

// Set output stream (standard out, or to file)
std::ostream *out;
if (result.count("output")) {
 std::string out_path = result["output"].as<std::string>();
 auto file = new std::ofstream(out_path);
 if (!file->is_open()) {

```

```

 std::cerr
 << "Error: failed to open output file \""
 << out_path << "\"." << std::endl;
 }
 out = file;
} else {
 out = &std::cout;
}

// Process sequences and output found code ID's
MILES::Decoder decoder(std::move(codes));
for (auto sequence: sequences) {
 auto found_ids = decoder.decode(std::move(sequence));
 std::cout << util::join(found_ids.peek(), ",", "", "") << std::endl;
}

return EXIT_SUCCESS;
}

bool validate_pattern(const std::vector<int> &pattern) {
 if (pattern.empty() || pattern[0] != 0 || pattern.size() != 7)
 return false;
 for (int i = 1; i < pattern.size(); i++) {
 if (pattern[i-1] >= pattern[i]) {
 return false;
 }
 }
 return true;
}

```

---

## Queue implementation: queue.hpp

---

```

#pragma once

#include <list>
#include <initializer_list>

//! Generic queue
// Generalized implementation of the queue data structure which
// enforces FIFO (first in, first out) insertion/deletion order.
template <class T>
class queue {
private:
 //! Underlying buffer
 std::list<T> m_buffer;

public:
 //! Default constructor
 queue() = default;

 //! Copy constructor
 queue(const queue<T> &other) = default;

 //! Initializer list constructor
 queue(std::initializer_list<T> list):

```

```

 m_buffer{ list.begin(), list.end() }
 {}

 /// Iterators constructor
 template <class InputIt>
 queue(InputIt first, InputIt last):
 m_buffer{ first, last }
 {}

 /// Is empty
 bool empty() const {
 return m_buffer.empty();
 }

 /// Enqueue value
 // @param val Queued (added) value.
 void enqueue(T val) {
 m_buffer.push_back(val);
 }

 /// Dequeue value
 // @return Dequeued (removed) value, or -1 if empty.
 T dequeue() {
 uint16_t val = -1;
 if (!m_buffer.empty()) {
 val = m_buffer.front();
 m_buffer.pop_front();
 }
 return val;
 }

 /// Erase elements
 // @param n Number of elements to erase.
 void erase(int n) {
 m_buffer.erase(m_buffer.begin(), std::next(m_buffer.begin(), n));
 }

 /// Clear queue of all elements
 void clear() {
 m_buffer.clear();
 }

 /// Is normal (first element is zero)
 bool normal() const {
 return m_buffer.front() == 0;
 }

 /// Normalize
 // If not already normal, subtracts the first element from itself and all elements.
 void normalize() {
 if (!normal()) {
 uint16_t front = m_buffer.front();
 for (auto &v: m_buffer)
 v -= front;
 }
 }

 /// Peek some
 // @return Vector of first n values in queue.

```

```

std::vector<T> peek(size_t n) const {
 return {
 m_buffer.cbegin(),
 std::next(m_buffer.cbegin(), std::min(n, m_buffer.size()))
 };
};

//! Peek all
// @return Vector of all values in queue.
std::vector<T> peek() const {
 return peek(m_buffer.size());
}

//! Size
size_t size() const {
 return m_buffer.size();
}
};

```

---

## Decoder declaration: decoder.hpp

---

```

#pragma once

#include <vector>
#include <iostream>

#include "queue.hpp"
#include "util.hpp"

namespace MILES {
 //! MILES code abstraction
 // Encapsulates a MILES code. Keeps track of how much of the pattern has been previously matched
 // by maintaining an iterator over the pattern. Using advance attempts to advance this iterator,
 // and using reset resets the state of the object (to the beginning of the pattern).
 class Code {
 private:
 //! Code ID
 int m_id;

 //! Code pattern
 std::vector<int> m_pattern;

 //! Position in code pattern
 std::vector<int>::iterator m_pattern_itr;

 //! Starting value
 int m_start;

 //! Bad flag
 bool m_bad;

 public:
 //! Default constructor (deleted)
 Code() = delete;
 };
}

```

```

 ///! Copy constructor
 explicit Code(const Code &other);

 ///! Parameterized constructor
 explicit Code(std::vector<int> &&pattern, int id);

 ///! ID
 int id() const
 { return m_id; }

 ///! Pattern
 const std::vector<int>& pattern() const
 { return m_pattern; }

 ///! Size
 size_t size() const
 { return m_pattern.size(); }

 ///! Is bad
 // The bad flag is set when advance encounters a value who's difference from the starting
 // value is greater than the pattern value that's currently being looked for.
 bool bad() const
 { return m_bad; }

 ///! Reset
 // Unset the bad flag and set the new starting value.
 // @param pulse New starting value
 void reset(int pulse);

 ///! Advance
 // Advances the code object by a pulse value. If the difference between the value and the
 // starting value is less than the pattern value that is currently being looked for, nothing
 // happens. If equal, advances the pattern value being looked for to the next value in the
 // pattern. If greater, sets the bad flag.
 // @ return True if the last pattern value has been found, false if otherwise or if the bad
 // flag is set.
 bool advance(int pulse);

 ///! String conversion operator
 explicit operator std::string() const;
};

///! MILES code sequence decoder abstraction
// Encapsulates a list of MILES code pattern objects which are used in decoding MILES sequences.
class Decoder {
private:
 ///! List of codes
 std::vector<Code> m_codes;

 ///! Reset all codes
 void reset_all(int pulse);

 ///! If all codes are bad
 bool all_bad() const;

public:
 ///! Default constructor (deleted)
 Decoder() = delete;

```



```

 //! Parameterized constructor (universal)
 explicit Decoder(std::vector<Code> &&codes);

 //! Decode sequence
 // Decodes the given sequence by checking for the codes the decoder was constructed with.
 // In decoding, the code objects are advanced per value of the sequence until either a
 // pattern is found and the found ID is added to a queue of found IDs (and the values of the
 // sequence in which the code pattern was found are removed), or until all codes become bad
 // in which case the process begins again by resetting all codes to the new first value of
 // the sequences after having dequeued a single value from the sequence (the old first
 // value). Once the sequence queue is empty (or the size is less than the length of all of
 // the code patterns), returns the queue of found IDs.
 // @param seq Input sequence
 // @return Queue of found code IDs
 queue<int> decode(queue<int> &&seq);
};

}

//! ostream insertion overload
inline std::ostream& operator<<(std::ostream& os, const MILES::Code &code) {
 return os << std::string(code);
}

```

---

## Decoder implementation: decoder.hpp

---

```

#include <cstdlib>
#include <vector>
#include <string>

#include "util.hpp"
#include "queue.hpp"
#include "decoder.hpp"

```

```

namespace MILES {
 // -----
 // Class: Code
 // -----
 Code::Code(const Code &other) = default;

 Code::Code(std::vector<int> &&pattern, int id):
 m_id { id },
 m_pattern { std::move(pattern) },
 m_pattern_ittr { m_pattern.begin() },
 m_start { 0 },
 m_bad { false }
 {}

 void Code::reset(int pulse) {
 m_pattern_ittr = m_pattern.begin();
 m_start = pulse;
 m_bad = false;
 }

 bool Code::advance(int pulse) {
 if (m_bad)

```

```

 return false;
 int delta = pulse - m_start;
 if (delta == *m_pattern_ittr) {
 m_pattern_ittr += 1;
 } else if (delta > *m_pattern_ittr) {
 m_bad = true;
 }
 return m_pattern_ittr == m_pattern.end();
}

```

```

Code::operator std::string() const {
 std::ostringstream oss;
 oss << "id:" << m_id
 << " pattern:" << util::join(m_pattern)
 << " curr:" << *m_pattern_ittr
 << " start:" << m_start
 << " bad:" << std::boolalpha << m_bad;
 return oss.str();
}

```

```

// -----
// Class: Decoder
// -----

```

```

Decoder::Decoder(std::vector<Code> &&codes):
 m_codes { std::move(codes) }
{}

```

```

void Decoder::reset_all(int pulse) {
 for (auto &code: m_codes)
 code.reset(pulse);
}

```

```

bool Decoder::all_bad() const {
 return std::all_of(m_codes.begin(), m_codes.end(),
 [](auto &code) { return code.bad(); });
}

```

```

queue<int> Decoder::decode(queue<int> &&seq) {
 queue<int> found_codes;
 while (!seq.empty()) {
 // break if sequence smaller than all codes
 if (std::all_of(m_codes.cbegin(), m_codes.cend(),
 [size = seq.size()](auto &code) { return size < code.size(); }))
 break;
 // peek entire queue, reset codes to first value, and look for a code
 auto peeked = seq.peek();
 auto ittr = peeked.begin();
 reset_all(*ittr);
 for (size_t i = 0; ittr != peeked.end(); ++ittr, ++i) {
 auto found = std::find_if(m_codes.begin(), m_codes.end(),
 [pulse = *ittr](auto &code) { return code.advance(pulse); });
 if (all_bad()) break;
 if (found != m_codes.end()) {
 found_codes.enqueue(found->id());
 seq.erase(i);
 break;
 }
 }
 }
 seq.erase(1);
}

```

```

 }
 return found_codes;
}
}

```

---

## Utility code: util.hpp

---

```

#pragma once

#include <string>
#include <sstream>

namespace util {
 const std::string DELIM = ",";
 const std::string LCAP = "[";
 const std::string RCAP = "]";

 //! Join C-array
 // Similar to the Python join function.
 // @param arr Input C-array
 // @param len Length of the array
 // @param delim Delimiter string
 // @param lcap Left surrounding character
 // @param rcap Right surrounding character
 // @return String of joined values
 template <class T>
 const std::string join(
 const T arr[],
 size_t len,
 const std::string &delim=DELIM,
 const std::string &lcap=LCAP,
 const std::string &rcap=RCAP)
 {
 std::ostringstream oss;
 oss << lcap;
 if (len > 0) {
 for (size_t i = 0; i < len - 1; ++i)
 oss << arr[i] << delim;
 oss << arr[len - 1];
 }
 oss << rcap;
 return oss.str();
 }

 //! Join generic STL container
 // Similar to the Python join function.
 // @param c Input container
 // @param delim Delimiter string
 // @param lcap Left surrounding character
 // @param rcap Right surrounding character
 // @return String of joined values
 template <template<typename ...> class Tc, typename T>
 const std::string join(
 const Tc<T> &c,
 const std::string &delim=DELIM,

```

```

 const std::string &lcap=LCAP,
 const std::string &rcap=RCAP)
 {
 std::stringstream oss;
 oss << lcap;
 if (c.size() > 0) {
 auto it = c.cbegin();
 for (size_t i = 0; i < c.size() - 1; ++i, ++it)
 oss << *it << delim;
 oss << *it;
 }
 oss << rcap;
 return oss.str();
 };
}

```

---