Project 2: Post Machine

Realization of Emil Post's abstract machine as a C++ class

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Describe the Turing / Von Neuman / Post concept of "data = instructions = data"
- Describe the architecture and processing algorithm of a Post machine
- Use data structures Queue and List to implement a Post machine as a C++ class
- Write client programs for List, Queue, and other data structures

Operational Objectives: Create a C++ implementation of a Post machine that conforms to the given driver program.

Deliverables: Three files `post.h`, `post.cpp`, `log.txt`.

Post Machines

A Post machine is an abstract computer. It is primitive only in the sense of usability: A theorem (proved originally by Emil Post [1897-1954]) states that a Post machine can compute anything that any modern computer can compute. See [Chapter 25 of Daniel I.A. Cohen's Introduction to Computer Theory, Wiley, 1991]. Post was a contemporary of Alan Turing, and Post machines are similar (and computationally equivalent) to Turing machines.

Architecture. A *Post machine* consists of

1. A *program memory*, which stores a program (as a list of instructions)
2. A *processing tape*
3. An *internal state*
4. A *processor*

A Post machine *instruction* consists of a character string of the form

```
<current_state><ch><new_state><word>
```

where <current_state>, <ch>, and <new_state> are single characters and <word> is a string of (zero or more) characters. A *program* consists of a list of instructions. (A program is stored on disk as a text file, one instruction per line. A Post machine stores program internally as a list of instructions.) The internal state is a value of the alphabet, a single character.

Processing Algorithm. A Post machine runs by first reading a program, which it keeps as a list of strings, and then receiving user input in the form of another string (which could be empty). The characters of the user input string are written to the front of the processing tape, followed by the symbol '#'. Execution begins with the internal state initially set to S (for Start). The machine executes by cycling through its program repeatedly and processing the tape according to the following rules:

If <current_state> matches the current (internal) state of the machine and <ch> matches the front of the tape, then <ch> is erased from the tape, the machine (internal) state is changed to <new_state>, and all of the characters in <string> are written to the tape. If there is no match for any instruction, the machine is said to crash. Note that if the tape is blank, there is no match.

After a cycle through the program, execution is terminated if (1) the current internal state is H (for Halt) - in which case the input string is ``accepted'', (2) a crash has occurred (the input string is ``rejected''), or (3) a maximum number of cycles has been reached (this prevents infinite loops). The contents of the processing tape is interpreted as the result of the computation in case (1), and as a core dump in cases (2) and (3).

A program is stored externally as a text file, one instruction per line, ending with '*' on a line by itself. Any line beginning with '*' is ignored (these are comment lines for the program).

Terminology Translator

The behavior of Post's "tape" is that of our ADT queue, including "writing to the tape" manifesting as the Queue Push operation and "erasing from the tape" manifesting as the Queue Pop operation.

Procedural Requirements

1. The official development/testing/assessment environment is specified in the Course Organizer.

2. Copy `LIB/scripts/submit.sh` and all of the files in `LIB/proj2/` into your `proj2` directory. You should now have these files:

   ```
   main.cpp         # driver program
   makefile         # project builder
   *.pp             # various post machine programs
   submit.sh        # submit script
   deliverables.sh # submission configuration file
   ```

   The files suffixed "`.pp`" are post machine programs ("pp" = "post machine program").

3. Maintain a work log `log.txt` containing work by topic/time as well as any experimental or testing observations you make during the development process.

4. Turn in three files `post.h`, `post.cpp`, and `log.txt` by entering the command `submit.sh`.

   ***Warning:** Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive two confirmations, the second with the contents of your project, there has been a malfunction.*

Code Requirements and Specifications

1. Class PostMachine should have the following simple API:

   ```
   public:
     bool Load     (bool batch = 0);
   ```

```
        void Run      (bool batch = 0);
```

as well as "announcing" constructor and destructor (run the distributed executable to see what these announcements should say).

2. Class PostMachine should use a private List `program_` to store programs, a private Queue `tape_` to serve as a tape, and a private char `state_` to store internal state:

```
    private:
      fsu::List  < fsu::String >                        program_;
      fsu::Queue < char , fsu::Deque < char> >          tape_;
      char                                              state_;
```

3. PostMachine::Load should query for a filename, open the file, and read the contents into its program memory (the List object), one instruction per list element. (Recall that an instruction is a line in the program file.)

4. PostMachine::Run() should execute according to the processing algorithm given above and report results. When in doubt, use `LIB/area51/post_i.x` as a guide.

5. In "batch" mode, user input is duplexed to the screen, so that when files of commands are used the user input is visible on the screen as if it were typed. This is accomplished as follows:

```
bool PostMachine::Load(bool batch)
{
  // I/O variables
  std::ifstream in1;
  fsu::String filename;

  // open program file
  std::cout << "  Name of instruction file: ";
  std::cin >> filename;
  if (batch) std::cout << filename << '\n';
  ...
}
void PostMachine::Run(bool batch)
{
  char ch;

  // simulation variables
  bool finished, halt, crash, maxits_reached, match;  // booleans
  size_t its = 0;

  // now loop until '*' is received as (first character of) input
  while(1)
```

```
    {
      // empty the tape
      tape_.Clear();

      // read input and place on tape, followed by '#'
      // if first char is '*', purge input buffer and return - Run is over
      std::cout << "   Input string (* to end): ";
      ch = std::cin.get();
      if (batch) std::cout.put(ch);
      if (ch == '*')
      {
        ClearBuffer(std::cin);
        if (batch) std::cout.put('\n');
        return;
      }
      while (ch != '\n')
      {
        tape_.Push(ch);
        ch = std::cin.get();
        if (batch) std::cout.put(ch);
      }
      if (batch) std::cout.put('\n');
      tape_.Push('#');

      // run Post machine
      state_ = 'S';
      finished = halt = crash = maxits_reached = 0;
      its = 0;
      do
      {
        match = 0;
        for (typename fsu::List<fsu::String>::ConstIterator i = program_.Begin(); i != program_.End(); ++i)
        {
          // check for match and manipulate state and tape accordingly
        }
        ++its;
        halt           = (state_ == 'H');
        crash          = !match;
        maxits_reached = (its == maxIterations);
        finished       = halt || crash || maxits_reached;
      }
      while (!finished);
      // report results
      if (halt)
      {
        ...
      }
      else if (crash)
      {
        ...
      }
      else if (maxits_reached)
      {
        ...
      }
      else // presumed unreachable branch
      {
        std::cerr << "** PostMachine error: bad processing termination\n";
      }
    } // end while(1)
  } // end Run()
```

   Note the suggested organization and logic in this example code.

6. Batch mode is invoked by supplying a command line argument ("batch" - or any other non-empty string), after which redirect the command file:

   ```
   linprog2:~/cop4530/proj2>post.x batch < test2.com
   ```

7. Test your post machine using the distributed post programs and comparing results from the distributed `LIB/area51/post_i.x`.

Hints

- Several Post machine programs are distributed - as files with ".pp" suffix. Each of these is documented in its header and named appropriately. For example, `bin2hex2bin.pp` programs a Post machine to convert binary to hexadecimal and vice versa.
- We will not get carried away with Post machine programs ... this is after all a C++-based class in data structures and algorithms. The main things to take away are (1) what a post machine is, (2) use of our List and Queue in implementing other classes, and (3) a whiff of the history underpinning modern computing (and facilitating a modern world).
- Substantial help can be obtained by reading carefully the example code above.
- Knowledge of the fsu::String API can be very helpful in this and future assignments. In particular, note the useful const method Element which returns by value and is a safe alternative to the string bracket operator. (See `LIB/cpp/xstring.*`.)

- The following stand-alone helper function may be useful in clearing unused keyboard input or skipping documentation in a program file:

```
void ClearBuffer(std::istream& is)
{
  char ch;
  do
  {
    is.get(ch);
  }
  while ((!is.eof()) && (ch != '\n'));
}
```

- Sample executable `post_i.x` distributed in `LIB/area51`.