University of Liverpool
Department of Computer Science

COMP282 – Advanced Object-Oriented C Languages

Coursework 1 – C++

Deadline: Tuesday 2nd of May at 17:00
Feedback will be released Tuesday the 9th of May and you can therefore not hand in later than that, even with an ELP (you will need to submit an EC in that case instead).

Weighting: 50%

What to submit: You are meant to submit 10 files (ordered by which task asks you to start implementing them) – in each case, you are meant to submit a .h file and the corresponding .cpp file for each of the below:

- TuringMachineState
- DenseTuringMachine
- TuringTape
- MenuSystem
- SparseTuringMachine

Note, submitting the .h file is mainly meant to be a help for you, since it will let you add in other methods and so on into the classes if you so wish and a baseline .h (except for TuringMachineState, where part of the task is to write one) is provided that contains everything required, but again, you may wish to add more. You should in general not remove anything from the .h files however, since, as mentioned, they just contain the code required for each task. You may add in any number of additional method/constructor/similar you wish, but if it is not required to have a constructor and you add one, also add in an empty constructor, because otherwise the way it will be marked might fail and you will lose points. Besides the mentioned files you submit, your program will be compiled with two additional files, namely TuringMachine.h and Main.cpp (the latter only intuitively – it will really be compiled with a subset, depending on which task you completed – this is to ensure that you can get points for having done e.g. only the first task, even though the Main file provided will require all parts to be done). Penalties for late work will be applied following the Code of Practice on Assessment. Your grade and penalty are based on your **last** submission.

Work alone: You must work alone on the assignment. Plagiarism checks will be run!

Tests and grading: Each task will have associated some public tests and some private tests (besides the final task, task 9, which will only have private tests). The public tests are described in secondary files, one for each task, but you will be graded based on how you do on the private tests (mainly to ensure you are not hardcoding the answer, except for the last task, task 9, where figuring out what to look for is also

important). Each task also gives some % of the mark for this assignment, mentioned in the title of the task.

You will be told (by CodeGrade) which of the public tests you passed whenever you submit – it will take a few minutes to run, depending on the number of people trying to submit at the same time. **You are therefore heavily suggested to submit a number of times – before the deadline - and change your submission based on how it went!** Your grade will be based on the outcome of the tests on your last submission.

The compilation of your code will be done using clang++ with the options: `-Wall -std=c++11`

The former gives more warnings, and the latter ensures you can only do C++11 as is the case by the baseline setup of Visual Studio 2019 you were meant to install/which is installed in the labs.

Tests cannot be done without some assumed behaviour. Some of the tests for different tasks (both public and private) assume that you have done the earlier tasks. If you have not, it is unlikely that you can succeed on that test/task. This was kept to a minimum, but still necessary at times. Each task will tell you which previous task (if any) you need to have done.

Project Overview

You will create a small interactive program to input a Turing Machine and run it, while being able to display the tape and current state). The project consists of 9 tasks, see below. It is likely easier to do the tasks in order (and besides that, as mentioned, some of the tests for some of the tasks assume this has been done). Each task builds on the previous ones. Do as much as you can and then package your project for submission. Begin by downloading the Visual Studio project template for this assessment.

Read through this entire document before you start coding, so you're aware of all parts and the overall structure of the program. Your solution should demonstrate your knowledge of C++. Important: Each task requires you to add extra code and each part is expected to work also at the end.

Task 1 – TuringMachineState Class Definition – 5%

Create a TuringMachineState class (incl. both a .h file and a .cpp file) that stores:

- a current state (a non-negative integer, implemented as an integer)
- a current content (a non-negative integer, implemented as an integer)
- a next state (a non-negative integer, implemented as an integer)
- a next content (a non-negative integer, implemented as an integer)
- a move direction (either the string "->" or the string "<-")

Note, while the parameters describe some requirements for the different parameters, you do not need to check that, but may simply assume that the input is such.

Declare and define a public constructor TuringMachineState that takes each parameter, in that order, and stores them in the object.

Create also public methods: getCurrentState(), getCurrentContent(), getNextState(), getNextContent(), getMoveDirection(). They should return the value of the corresponding parameter.

Task 2 – Turing machine state I/O - 5%

Implement the << and >> operators so you can output and input a Turing Machine state object with the following string format:

```
2 5 4 3 ->
```

In other words, the Turing machine states are output/input by outputting/inputting the parameters in order. For this task, you do not need to do any input validation or error handling. Assume the input has been validated elsewhere. For obvious reasons, you should do task 1 before this one.

Task 3 – Comparison Operators – 5%

Implement comparison operators (<, >, and ==) for the TuringMachineState class. These should work lexicographically based on current state and current content: I.e. if the current state differs, then if first current state > second current state, then the first > second. Otherwise, if the current states are equal, if the first current content > second current content, then first > second (similar for the others – equality requires just that they have equal current state and current content). For obvious reasons, you should do task 1 before this one.

About Turing machines

In the remaining parts of the assessment, we are looking at Turing machines.

A Turing machine is a set of Turing machine states, where, for each pair of numbers x, y, there is at most one TuringMachineState with current state x and current content y (i.e. getCurrentState()=x and getCurrentContent()=y). You then also have a tape (for this assessment, initially filled with 0s) and a current position of the tape head and a current state of the Turing machine (for this assessment, both are initially 0 – you may assume that both can be stored in ints). It then repeatedly does a step as follows, until termination, (or indefinitely if that does not happen):

1. Check that the current position of the tape head is at least 0 and below the size of the tape (the latter only if the tape is supposed to be finite). If not, you are done and should terminate with an error message.
2. Find the TuringMachineState with current state (i.e. the output of getCurrentState()) equally to the current state of the Turing machine (i.e. the first time you should find state 0) and the current content equal to the content of the tape at the position of the tape head. If there is no such state, you are done and can terminate with an error message.
3. Update the current state of the Turing machine to be the next state of the TuringMachineState (i.e. getNextState()), the content at the current position of the tape head to be the next content of the TuringMachineState (i.e. getNextContent()), and afterwards increment the position of the tape head if the

move direction of the TuringMachineState is -> otherwise, if it is <-, decrement it.

The remaining tasks are about implementing the storage of the TuringMachineStates so that we can do item 2 fast and implementing the finite/unbounded tape so that we can do item 3 fast (item 1 should be quite straightforward).

Task 4 – DenseTuringMachine – 10%+5%=15%

In this task, we are trying to implement item 2 in the case where, if you put TuringMachineState

```
2 5 4 3 ->
```

into entry (2,5) – i.e. the entry of the same current state and current content - of a matrix and similar for the other TuringMachineStates, you would get a dense matrix (meaning that a large fraction of entries contains a TuringMachineState).

The implementation should be done in the files for DenseTuringMachine.cpp and a corresponding .h file is provided (that you may wish to modify), should implement DenseTuringMachine, which should be a child of the abstract class TuringMachine. In particular, it should implement the following constructor and functions:

- DenseTuringMachine(int x, int y), creates a new DenseTuringMachine in which each TuringMachineState has current state <= x and current content <= y (when x and y are non-negative).
- TuringMachineState* find(int x, int y), which should return the TuringMachineState that has current state x and current content y or NULL if no such TuringMachineState exist at the time the command is run
- void add(TuringMachineState s), which should add s to the states that could be found.
- vector<TuringMachineState>* getAll(), which should return a pointer to the vector of all TuringMachineStates, in some order.

You are meant to implement these so that the total time to run the constructor followed by any z functions runs in total time **O(xy+z)**. This will be verified by hand and gives 10%. You won't get the 10% if the functions aren't implemented basically correctly (i.e. if nearly correct, then fine, but if it is not close to correct then no).

Besides that, you also need to implement the functions correctly and will receive the last 5% for that. For obvious reasons, you should do task 1 before this one.

Task 5 – TuringTape – 10%+5%=15%

In this task, we implement a finite Turing Tape (we will do the infinite one later in task 8).

The implementation should be done in the file TuringTape.cpp (a corresponding .h file is provided, but you may wish to add more code to it), which should implement TuringTape and the following constructor and virtual functions:

- TuringTape(int n), which should construct a tape for a Turing machine with n spaces, initially all 0 and initially with current position 0. n should be positive (or, later, -1).
- bool moveRight(), which should increment the current position and output false iff the current position afterwards is >n (or <0)
- bool moveLeft(), which should decrement the current position and output false iff the current position is <0 (or >n)
- int getContent() which should return the content at the current position. If the current position is outside the tape, return 0.
- void setContent(int c), which should set the current content to c. If outside the tape, this should do nothing.
- int getPosition(), which should return the current position.
- operator<<(std::ostream& out,const TuringTape& T), which should output the content of the tape from 0 until the highest position that has been/is the current position (you will likely need a variable for this – we do this instead of always outputting the full tape to make the change to infinite tape easier). Note that this should be a friend and not actually a part of TuringTape.

It is expected to be based on an iterator that can move left and right in O(1) time (output does not necessarily need to use the same iterator though). You will get 10% for doing so and this will be checked by hand. Like in Task 4, you won't get the 10% if the functions aren't implemented basically correctly.

You will get the remaining 5% for implementing the functions correctly.

Task 6 – Menu – 25%

In this task, we are implementing a command line program for creating and executing Turing machines and is expected to be based on the earlier classes. You should do the implementation in main.cpp and a corresponding .h file is provided, but you may wish to add more code, and the program should run when void menu() is called.

Specifically, you are meant to create a program that initially displays:

```
How long should the tape be?
```

And then waits for input of an int. You are then expected to use that to create the tape. You would also be expected to create a variable for the current state, which should initially be 1.

Afterwards, it should show the following menu:

```
1. Create dense Turing machine
2. Create sparse Turing machine
3. Add state to Turing machine
4. Compact Turing machine
5. Execute Turing machine
6. Output current information
Write q or Q to quit
Enter Option
```

Note that there are no requirements on whitespace, except that there is some between each and you could even write it all on one line (but it would be ugly).

You are then expected to wait until the user inputs a string and if it is a:

1. If the user enters 1, you should display:

   ```
   What is the maximum state and what is the maximum content?
   ```

   and then wait for 2 ints, which is expected to be used to create a DenseTuringMachine.

2. We will do 2 in a later task.
3. If the user enters 3, you should display:

   ```
   What state do you wish to add?
   ```

   And then wait for them to enter a TuringMachineState, which should be added to the current TuringMachine.
4. We will do 4 in a later task.
5. If the user enters 5, you should display:

   ```
   How many steps do you wish to execute?
   ```

   And then wait for the user to write an int.
   You then do the following repeatedly until you have done it that many times (or until an error occurs, in which case you just go back to the menu):
   - If the current position is either below 0 or at least the length of the tape, you should display the error message:

     ```
     In step x, the position is y, but that is outside the tape.
     ```

     The number x in the message should be the number of steps done so far (both in the current execution of 4 and from previous executions) and the number y should be the current position in the tape.
   - Find the current TuringMachineState to use. I.e., the one which matches the current state and matches the content in the current position of the tape. If no such state exists, you should display an error:

     ```
     In step x, there is no Turing machine state with state y and content z
     ```

     The number x in the message should be the number of steps done so far, like above. The number y should be the current state and the number z the content of the tape at the current position. After having displayed that, you should go back to the main menu (skipping the rest of the executions).
   - Given the current TuringMachineState from the previous bullet, you should update the state to be the next state of that TuringMachineState, the content at the current position of the tape to be the next content and then increment the current position if the next move is -> and otherwise, if the next move is <-, decrement it.
6. If the user enters 6, you should display:

   ```
   The current state is x. The current position is y.
   The content of the tape is z.
   The states of the Turing machine is <a> <b> …
   ```

Where x is the current state of the Turing machine (i.e. not TuringMachineState), y the current position in the tape, z is the content of the tape (i.e., output in TuringTape) and a, b, … are the states of the Turing machine

Q. If the user enters Q (or q), you should terminate the method.

Whenever a command ends (besides q and Q), you are expected to go back and display the menu and again wait for a new choice to be made. This is also the case if the user selected a wrong option.

If you do this correctly, you get 25%, since this felt like a large part of the project. There are a few private runs and each gives partial credit because it is so large a part of the project.

While not strictly required, this task is likely easier if you have done each previous task, except not necessarily task 3.

Task 7 – The last two menu items – 10%+5%=15%

In this task, we implement the last 2 menu items, items 2 and 4.

For item 2, you should implement the class SparseTuringMachine in the file SparseTuringMachine.cpp (a corresponding .h file is provided, but you may wish to add more), which is much like DenseTuringMachine. The difference is that the constructor takes no input and that you **can't** assume that the matrix described in DenseTuringMachine is dense. This means that very likely, if you just tried to use your dense implementation (at least if you satisfy the time constraints), it will take a very long time to run the example below.

To be precise, you should implement:

- TuringMachineState* find(int x, int y), which should return the TuringMachineState such that has current state x and current content y or NULL if no such TuringMachineState exist at the time the command is run
- void add(TuringMachineState s), which should add s to the states that could be found.
- vector<TuringMachineState>* getAll(), which should return a pointer to the vector of all TuringMachineStates, in some order.

We want the time to run the constructor and any z commands to be O(z log z) (O(z log log z) can be done in cases like this, but you are not expected to do so).

When selecting option 2 in the menu, you are then expected to create a new SparseTuringMachine.

We will describe what the effect of item 4 should be: We want to create a dense version of the current Turing machine (or at least as dense a version as we can). To do so, take each distinct state (NOT TuringMachineState, but getCurrentState() /getNextState()) and sort them in increasing order. Then, rename state x to be the

position of x in the sorted order (i.e. the smallest state number used becomes state 0, the second smallest state 1 and so on). Then do similarly for the content (the smallest content becomes 0, the next smallest 1 and so on). Then, create a dense Turing machine with those TuringMachineStates after renaming. Note, while the point is mainly to use this for when you have created a sparse Turing machine, it should also work correctly even if the currently created Turing machine is already dense. The run time for this should be O(n log n) if there are n TuringMachineStates.

The runtimes will be verified by hand and will give you 10% of your grade. Note, like before, if your implementation is not close to correct, you will not get points for getting the time right. The remaining 5% is for getting things correct.

You need to have done Task 6 before you can get points for this one, but having done each of the previous tasks will likely be helpful.

Task 8 – -1 is infinite –5%

We want to be able to enter infinite (encoded as -1) for the length of the tape (i.e. at the beginning of the menu), the number of steps to execute (meaning it runs until an error occurs) and the maximum state and content for dense Turing machines (i.e. the input to the constructor) – in the latter case, the run time should then be O(xy+z) to run z commands, where x and y are the largest used state and content respectively (you do not get to let it run forever…). It is acceptable to just use the largest int value instead of infinite (because we are assuming you can use ints for each of these anyway), except it might mess with the time complexity... This will be verified by hand.

To get points for this, you need to have done Task 6.

Task 9 – Error correcting/input validation –10%

In this task, you are supposed to ensure that no crashes or similar can occur (the behaviour should be clear in each case, but you can of course ask if you are not sure) and validate input for the menu system. For simplicity, you may assume that the datatype is correct each time and only the value is wrong. To get points for this item, you are expected to have done all other tasks. If an input is not valid, display the last prompt, which is:

```
Enter Option
```

for the menu and otherwise, the string outputted just before you waited for input. Just to be clear, it makes no sense to run some of the commands if no Turing machine has been made yet and in that case, you are supposed to show the above Enter Option prompt again…

Also, in each case, if we wait 1 or 2 integers (and nothing else), then an input of 0 (or 0 0) makes no sense and you should ask the user for input again, showing the right prompt as above.

There are no public tests since a large part is to figure out where the errors might be. Instead, there are several private tests, each giving a part of the final percentage, so if you find something and miss something else, you will get partial credit.

Marking Descriptors

We draw your attention to the standard Department Grade Descriptors, which are listed in the Student Handbook.