Practical 9

COS132

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Official Deadline: 31/05/2024 at 17:30

Extended Deadline: 02/06/2024 at 23:59

Marks: 95

1 General instructions:

- This assignment should be completed individually; no group effort is allowed.
- Be ready to upload your practical well before the deadline, as no extension will be granted.
- **The extended deadline has been put in place in case of loadshedding or other unforeseen circumstances. No further extension will be granted.**
- **Note that no tutor or lecturer will be available after the official deadline**
- You may not import any libraries that have not been imported for you, except `<iostream>`.
- You may `use namespace std;`
- If your code does not compile, you will be awarded a zero mark. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire practical before you start coding.
- You will be afforded 20 upload opportunities per task.

- **You should no longer be using the online compiler. You should compile, run and test your code locally on your machine using terminal commands or makefiles**.
- **You have to use C++98 in order to get marks**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm. **If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.**

3 Outcomes

Upon successful completion of this practical, students will be able to:

- Manipulate and iterate through strings using pointers.
- Declare and initialize arrays.
- Use pointers or indexes to access and modify array elements.
- Implement functions that utilise pointers and references to manipulate data.
- Understand and use double pointers for pointer manipulation.
- Utilise pointers for dynamic memory allocation and deallocation.
- Develop logical strategies for solving complex problems using arrays and pointers.

4 Structure

This practical consists of two tasks. Each task is self-contained and all of the code you will require to complete it will be provided in the appropriate Task folder. Each task will require you to submit a separate archive to an individual upload slot. That is, each separate task will require its own answer archive upload. You will upload Task 1 to Practical 9 Task 1 and so on. You can assume that all input will be valid (i.e. of the correct data type and within range)

5 Mark Distribution

Activity	Mark
Task 1 - Turing Machine	31
Task 2 - CodeBrew collapse	64
Total	**95**

Table 1: Mark Distribution

6 Task 1

Your task is to implement the functions in the TuringMachine.h and TuringMachine.cpp files. You will write a program that simulates a Turing Machine based on rules give.

6.0.1 Introduction to Turing Machines

Turing Machines are a fundamental model of computation developed by Alan Turing in 1936. They provide a simple yet powerful mechanism to simulate any algorithm. A Turing Machine consists of:

- **Tape**: An infinitely long tape which acts as the memory of the machine, divided into cells. Each cell can contain a symbol from a finite alphabet. In our implementation we will simulate this with a large char array (while not actually infinite we will ensure that it is sufficient for the given input and rules). We will treat '_' as the blank symbol.
- **Head**: A head that reads and writes symbols on the tape and can move left or right. In our implementation you will use a 'head' variable as an index into the tape array.
- **State Register**: Maintains the current state of the Turing Machine. The machine starts in an initial state and changes states according to rules based on the current state and the symbol it reads. In our implementation this will be a string variable.
- **Transition Function**: Defines what the machine does (write, move, change state) based on the current state and the symbol under the head. In our implementation we will simplify this:
  - We will always 'write' even if we are actually copying the same symbol.
  - We will always 'update the state' even if we are updating it back to the same state
  - We will move the head left ('L') or right ('R') or stay in the same place (any other character)
- **Accept or Reject States**: Special states that stop the machine and signify whether the input was accepted ("accept") or rejected ("reject"). In our implementation we also have a special "start" state.

Our rules will be formatted as a 2D array of strings, eg.

```
const int numRules = 6;
string rules[numRules][5] = {
    {"start", "0", "start", "0", "R"},
    {"start", "1", "start", "1", "R"},
```

```
        {"start", "_", "check", "_", "L"},
        {"check", "0", "accept", "0", "R"},
        {"check", "1", "reject", "1", "R"},
        {"check", "_", "reject", "_", "R"}
    };
```

Each rule consists of the following:

1. current state
2. read symbol (truncated to the first character)
3. next state
4. write symbol (truncated to the first character)
5. direction 'L' means left, 'R' means right. Anything else means no move

For example, consider the rule:

```
{"start", "0", "start", "0", "R"}
```

Interpretation:

- Current State: The machine is currently in the "start" state.
- Read Symbol: The machine reads a '0' at the current position of the head on the tape.
- New State: After applying this rule, the machine remains in the "start" state.
- Write Symbol: A '0' will be written to the tape at the current position of the head.
- Move Direction: The head moves one position to the right ("R"). If the rule specified "L", it would move one position to the left.

6.0.2 Turing Machine Header

The `TuringMachine.h` header file includes the declarations for functions that simulate a Turing Machine

```
#ifndef TURING_MACHINE_H
#define TURING_MACHINE_H

#include <iostream>
#include <string>
using namespace std;

void initializeMachine(char tape[], int& head, const string& currentState, int tapeSize);

bool matchRule(const string& currentState, const char& currentSymbol, string rules[][5],
               int numRules, string& newState, char& newSymbol, string& move);

void applyRule(char tape[], int& head, const string& newState, string& currentState,
            const char& newSymbol, const string& move, int tapeSize);

bool checkFinalState(const string& currentState);

void printOutput(const char tape[], const string& currentState, int tapeSize);

void simulateTuringMachine(char tape[], string rules[][5], int numRules, int tapeSize);
#endif // TURING_MACHINE_H
```

Create `TuringMachine.cpp` and include the `TuringMachine.h` header.

```
#include "TuringMachine.h"

// Function implementations will go here
```

You will implement the functions in `TuringMachine.cpp`

6.1 Task: Simulate a Turing Machine

Your task is to implement several functions that collectively simulate a Turing Machine. Below are the specifications for each function involved in the simulation.

6.1.1 void initializeMachine(char tape[], int& head, string& currentState, int tapeSize)

- **Purpose:** Sets up the tape and positions the head at the first non-blank symbol (or at the start if all are blank). Remember that '_' is the "blank symbol".

- **Parameters:**
  - **char tape[]:** The tape array containing symbols.
  - **int& head:** Reference to the head position.
  - **string& currentState:** Reference to the current state, initially set to "start".
  - **int tapeSize:** Total number of cells on the tape.

6.1.2 bool matchRule(const string& currentState, const char& currentSymbol, string rules[][5], int numRules, string& newState, char& newSymbol, string& move)

- **Purpose:** Identifies and selects the applicable rule based on the current state and the tape symbol at the head.
- **Parameters:**
  - **const string& currentState:** Current state of the machine.
  - **const char& currentSymbol:** Symbol currently under the head.
  - **string rules[][5]:** Transition rules.
  - **int numRules:** Number of available rules.
  - **string& newState:** Update `newState` when appropriate rule found.
  - **char& newSymbol:** Update `newSymbol` when appropriate rule found.
  - **string& move:** Update `move` for the head movement direction.
- **Return:** True if a matching rule is found, otherwise false.

6.1.3 void applyRule(char tape[], int& head, const string& newState, string& currentState, const char& newSymbol, const string& move, int tapeSize)

- **Purpose:** Applies the rule by writing a symbol, moving the head, and updating the state.
- **Parameters:**
  - **char tape[]:** The tape array.
  - **int& head:** Reference to the head position, to be updated based on movement.
  - **const string& newState:** New state, this should be used to update the current state in the function.
  - **string& currentState:** Current state, this should be updated using newState.
  - **const char& newSymbol:** New symbol to write at the head's current position. This should be done before moving the head.
  - **const string& move:** Direction to move the head ('L' or 'R'), if neither then don't move the head.
  - **int tapeSize:** Total number of cells on the tape, to prevent moving out of bounds.

6.1.4 bool checkFinalState(const string& currentState)

- **Purpose:** Determines if the current state is a stopping state ('accept' or 'reject').
- **Parameters:**
  - **const string& currentState:** The current state of the machine.
- **Return:** True if the state is a final state, otherwise false.

6.1.5 void printOutput(const char tape[], const string& currentState, int tapeSize)

- **Purpose:** Outputs the final state of the tape and the Turing machine's current state.
- **Parameters:**
  - **const char tape[]:** The tape array to be printed.
  - **const string& currentState:** The final state of the Turing machine.
  - **int tapeSize:** Size of the tape.
- **Print Statements:**

```
Final tape: <the tape printed by looping through characters>
Machine ended in state: < currentState >
```

  with newlines after each line.

6.1.6 void simulateTuringMachine(char tape[], string rules[][5], int numRules, int tapeSize)

- **Purpose:** Simulates the operation of a Turing Machine using given rules and tape.
- **Parameters:**
  - **char tape[]:** An array representing the tape, where each cell contains a single character from the tape's alphabet.
  - **string rules[][5]:** A 2D array of strings representing the transition rules. Each rule is an array of five strings: current state, read symbol, new state, write symbol, and move direction ('L' for left, 'R' for right).
  - **int numRules:** The number of transition rules provided.
  - **int tapeSize:** The number of cells on the tape.
- **Process:**
  1. **Initialization:** The machine's tape and initial head position are set up using the appropriate function. The machine starts in the "start" state.

2. **Simulate Turing Machine:** The function should continue to loop until the machine is no longer running. In other words, until the machine is in a final state, or no rule can be applied (then the machine hangs). Use the appropriate functions to match and apply rules.
3. **Output:** After exiting the loop, the function outputs the final state of the tape and the ending state of the machine using the `printOutput` function.

7 Task 2

You defused the bomb, but then the figure appeared again, this time with a grin that felt familiar. They looked at you and said it was just a distraction. Then, in a base64-encoded message without padding (so no ==), they said, `eW91IGNyZWF0ZWQgdGhpcw`. As you looked at them more closely, you realized it was the CodeBrew founder, your old boss... and then they disappeared again. You must decrypt the message to understand what must be done next because they are planning something else evil for the summit.
This is your final mission, your final task. You must finish this, with all the skills you have learnt, you believe you can end Code Brew.

7.1 Header and Source File Creation Guide

This guide outlines the structure and responsibilities of the header ('.h') and source ('.cpp') files for the monitoring system. Each header file contains the declarations of functions related to the environmental controls for temperature, pressure, and humidity.

7.1.1 Message Decoder Header

The `MessageDecoder.h` header file includes the declarations for function that will use substring analysis to decode a Base64 encoded string, you will follow the real world application of decoding a Base64 message.
NB!! All messages tested will not have any padding issues, so do not cater for padding when decoding!!

```
#ifndef MESSAGE_DECODER_H
#define MESSAGE_DECODER_H

int findIndex(char c);
void decodeBase64(const char encoded[], char decoded[], int encodedLength);

#endif
```

Create `MessageDecoder.cpp` and include the `MessageDecoder.h` header.

```
#include "MessageDecoder.h"

// Implement the functions to decode Base64 message here
```

7.1.2 Treasure Map Header

The `TreasureMap.h` header file contains the function declarations related to finding the string and X of the treasure.

```
#ifndef TREASURE_MAP_H
#define TREASURE_MAP_H

bool findTreasure(char map[ROWS][COLS], int jumps[], int jumpsLength, char *result);

#endif
```

Create `TreasureMap.cpp` and include the `TreasureMap.h` header at the top:

```
#include "TreasureMap.h"

// Implement the functions to find the treasure
```

7.1.3 Poison Dilute Header

The `PoisonDilute.h` header file contains the function declarations related to finding the nth largest amount needed to dilute the poison.

```
#ifndef POISON_DILUTE_H
#define POISON_DILUTE_H

int* findNthLargest(int *arr, int size, int n, int *largest)

#endif
```

Create `PoisonDilute.cpp` and include the `PoisonDilute.h` header at the top:

```
#include "PoisonDilute.h"

// Implement the functions to find the nth largest amount needed to dilute the poison
```

Each '.cpp' file should implement the functions declared in its respective '.h' file. Pay attention to ensure that each '.cpp' file includes its corresponding header file to provide function prototypes, enabling proper compilation.

7.2 Detailed Function Implementation Specifications for Message Decoder

Base64 Character Array

Create a const array that holds all the Base64 characters in the correct order: `A-Z`, `a-z`, `0-9`, and `+/` characters.

Finding Index of a Base64 Character

Implement `int findIndex(char c)` that returns the index of a given Base64 character from the Base64 character array. If the character is not found, return -1.

Decoding Base64

Implement `void decodeBase64(const char encoded[], char decoded[], int encodedLength)` that:

- Iterates over the encoded string in blocks of 4 characters.
- Uses `findIndex` to convert each character to its corresponding index.
- Uses these indices to calculate the original 3 bytes from each block of 4 characters.
- Stores these bytes in the `decoded` array.
- Properly handles the null-termination of the `decoded` array.

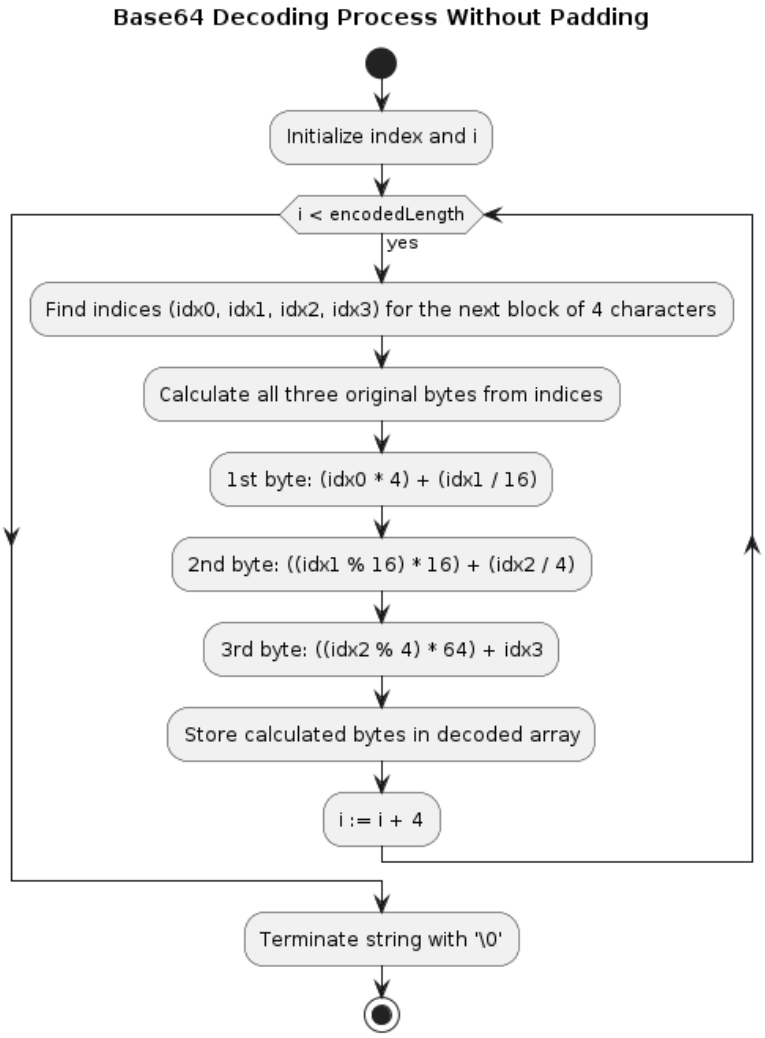

Figure 1: Base64 Decoding Process Without Padding

7.3 Detailed Function Implementation Specifications for Treasure Map

Constants `ROWS` and `COLS`

Define constants to specify that ROWS is 4 and COLS is 5 of for the 2D array in the correct file.

Function `findTreasure`

Implement the function `bool findTreasure(char map[ROWS][COLS], int jumps[], int jumpsLength, char *result)` that:

- Initialize an index for the `result` array.
- Iterate through the `jumps` array.
- Convert each jump to a row and column index in the 2D array.
- Check if the current position in the map contains the treasure ('X'). If found, null-terminate the `result` array and returns true.
- If the current position contains a character other than '-', collect the character into the `result` array.
- Continue until all jumps are processed, then null-terminate the `result` array and returns false if the treasure is not found.

7.4 Detailed Function Implementation Specifications for Poison Dilute

Function `findNthLargest`

Implement the function `int* findNthLargest(int *arr, int size, int n, int *largest)` that:

- Iterates to find the first `n` largest elements in the array.
- Stores these elements in the `largest` array and sets the found elements in the original array to zero.
- Sort the `largest` array in ascending order.
- Returns sorted `largest` array.

8 Submission checklist

For Task 1:

- Archive (zip) all the files used for Task 1 and rename the archive uXXXXXXXX.zip where XXXXXXX is your student number. The zip should include:
  - TuringMachine.h
  - TuringMachine.cpp

- Upload the archive to FitchFork Practical 9 Task 1 before the deadline

For Task 2:

- Archive (zip) all files used for Task 2 and rename the archive uXXXXXXXX.zip where XXXXXXX is your student number. The zip should include:
  - `MessageDecoder.h`
  - `MessageDecoder.cpp`
  - `TreasureMap.h`
  - `TreasureMap.cpp`
  - `PoisonDilute.h`
  - `PoisonDilute.cpp`
- Upload the archive to FitchFork Practical 9 Task 2 before the deadline