Technical University of Denmark

Written examination, December 5, 2016 Page 1 of 13 pages

Course name: Programming in C++

Course number: 02393

Aids allowed: All aids are permitted

Exam duration: 4 hours

Weighting: pass/fail

Submission details:

1. You can hand-in your solutions manually (on paper). However, we strongly recommend you to submit them electronically.

2. For electronic submission, you **must** upload your solutions on CampusNet. Each assignment must be uploaded as one separate `.cpp` file, using the names specified in the exercises, namely `Z-library.cpp`, where `Z` ranges from `1` to `6`. The files must be handed in separately (not as a zip-file) and must have these exact filenames. Feel free to add comments to your code.

3. For your convenience, we have also set up the exam in CodeJudge, under `Exam` at `https://dtu.codejudge.net/02393-e16`. You can use this platform to test your programs before submitting them on CampusNet. Please be aware that CodeJudge **is not** the submission platform for this exam, but CampusNet! Also, the use of CodeJudge in an exam is new, and there may be bugs and technical problems — so please do not worry if anything goes wrong, just submit your best answer finally on CampusNet.

1. **Euler's Number (15 pt)**

Euler's number $e \approx 2.718$ can be defined by the following infinite sum:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \dots$$

Task: Write a program that takes as input a positive integer k and outputs the sum of the first k summands of this infinite series, i.e.,

$$\sum_{n=0}^{k-1} \frac{1}{n!} .$$

(For input $k \leq 0$, your program may behave as you wish.)

To get full points, please ensure that the program has **linear time**, i.e. avoid to compute a factorial for each summand.

The given code is shown below...

Public test cases:

Input	Output
1	1
2	2
4	2.66667
10	2.71828

File `1-library.cpp`

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

double approx_e(int k){
  // To be written
}

int main(){
  int k;
  cin >> k;
  cout << approx_e(k) << endl;
  return 0;
}
```

2. **Making Categories (15 pt)**

   Given a sequence of test results between 0 and 100 points, we want to classify the results as *poor*, *good*, and *excellent* according to how the results are distributed, where *poor* should be the bottom 25%, *excellent* the top 25%, and *good* everything in between. For instance, given numbers $92, 48, 71, 52$, the classification would be:

$$\underbrace{\boxed{48}}_{\textit{poor}} \quad \underbrace{\boxed{52}\boxed{71}}_{\textit{good}} \quad \underbrace{\boxed{92}}_{\textit{excellent}}$$

   We also want to ensure that equal results always belong to the same category, and when necessary, elements should rather belong to the *good* category. Thus, the *good* category should be at least 50%, and the other two at most 25%. For instance for the numbers $52, 52, 48, 48, 53$, we would have:

$$\underbrace{\boxed{48}\boxed{48}\boxed{52}\boxed{52}}_{\textit{good}} \quad \underbrace{\boxed{53}}_{\textit{excellent}}$$

   Task: Write a C++ program that takes as input an arbitrarily long sequence of integers between 0 and 100, computes the categories and outputs smallest number in the *good* and *excellent* categories. You may assume that there are at least four numbers on input.

   The given code follows on the next page...

   Public test cases:

Input	Output
`92 48 71 52`	`good>=52 excellent>=92`
`52 52 48 48 53`	`good>=48 excellent>=53`

File `2-library.cpp`

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

int main(){
  // Todo: Declare datastructure to hold the elements

  int i;
  while(true){
    cin >> i;
    if(cin.fail()){
      break;
    }

    // Todo: insert i into the datastructure

  }

  int good_min, excellent_min;

  // write main code here

  cout << "good>=" << good_min << "␣excellent>=" << excellent_min << endl;
}
```

3. **Bag (10pt)**

   Scenario: a customer is shopping a number of items (represented by strings). At the end, we want to print out a receipt where the items are alphabetically ordered. Each item appears only once on the receipt with the quantity (how many times it was bought).

   Task: Write such a program in C++ with the output format in the test case below.

   The given code is shown below...

   Public test case:

Input	`Apple Pear Banana Pear Apple Kiwi Kiwi Pear Coconut`
Output	`2x Apple`
	`1x Banana`
	`1x Coconut`
	`2x Kiwi`
	`3x Pear`

   File `3-library.cpp`

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

// Format for outputting elements:
void print(string s, int n){
  cout << n << "x␣" << s << endl;
}

int main(){
  // Todo: Declare datastructure to hold the elements

  string s;
  while(true){
    cin >> s;
    if(cin.fail()){
      break;
    }

    // Todo: insert s into the datastructure

  }

  // call the print function accordingly to present the shopping list
}
```

4. **Queue as Array (25 pt)**

   In this task, we redefine the Queue class from the lecture, using an array of a fixed size, which we also call the *capacity* of the queue. This Queue shall terminate with an error (or throw an exception) when more elements than this capacity are in the queue at any time.

   A header file is given: `4-library.h` (displayed on the next page) where the class `queue` has the following private member variables:

   - `int capacity;` – the mentioned capacity
   - `int * elements;` – an integer array of size capacity to hold the queued-up elements
   - `int first,last;` – indices into the array to point to the first and last element of the queue. (According to standard C++ convention you may choose to set index of the last element *plus 1*.)

   Task Implement the member functions of the class `queue` (as defined in the header-file `4-library.h` below) that are as follows:

   - `queue(int capacity);` – the main constructor that gets as an argument the capacity the queue shall have.
   - `queue(const queue & q);` – the copy constructor that initializes this queue as a copy of the given queue q, in particular making a copy of its array of elements.
   - `~queue();` – the destructor to deallocate the array.
   - `void enqueue(int i);` – insert number i at the end of the queue. This shall give an error when the capacity is exceeded.
   - `int dequeue();` – remove and return the first item of the queue. This shall give an error when the queue is empty.
   - `int top();` – return the first item of the queue. This shall give an error when the queue is empty.
   - `bool isEmpty();` – return whether the queue is empty.

   The given code follows on the next page...

File `4-library.h` (Do not modify)

```
#ifndef __MYQUEUE
#define __MYQUEUE

class queue{
private:
  int * elements;
  int first,last;
  int capacity;

public:
  queue(int capacity=10);
  queue(const queue & q);
  ~queue();
  void enqueue(int i);
  int dequeue();
  int top();
  bool isEmpty();
};


#endif
```

File `4-main.cpp` (Do not modify)

```
#include "4-library.h"
#include <iostream>
using namespace std;

int main(){
  queue q(3);
  q.enqueue(17);
  q.enqueue(42);
  q.dequeue();
  q.enqueue(33);
  q.enqueue(12);
  while (!q.isEmpty())
    cout << q.dequeue() << "␣";
  cout << endl;
}
```

File `4-library.cpp` (Hand in only this file)

```
#include "4-library.h"

// To be written
```

Public Test Case:

The test case is given by the file `4-main.cpp` that uses the queue class. This resulting program should produce the output: `42 33 12`

Note that the total number of `enqueue` operations is greater than the capacity, but the capacity is never exceeded since we have also `dequeue` operations in between.

Hint: one can implement this without shifting of elements in the array by simply allowing that first and last, when reaching the end of the array, start at the beginning again, so we have in the above example the following array, and first and last point to the respective indices:

12	42	33

5. **Tic Tac Toe (20 Points)**

   We implement now an automatic procedure for playing the well-known Tic Tac Toe game. (In fact, you can complete this task without even understanding the game, because all game specifics are already part of the given code.)

   Tic Tac Toe game has a 3-by-3 matrix where two players, denoted as `X` and `O` alternatingly can mark a field. The winner is who first has an entire row, column, or diagonal marked. If we have no winner when all fields are filled, it's a draw (nobody wins).

   We represent this (the given code in `ttt.cpp`) as follows: for the matrix we use a vector of size 9,

0	1	2	3	4	5	6	7	8

   representing the fields of 3-by-3 matrix in the following order:

0	1	2
3	4	5
6	7	8

   A field is defined by the datatype

   ```
   enum field {O,X,draw};
   ```

   where `O` and `X` represent fields marked by the respective player and `draw` means an empty field. A *state* of the matrix is then a vector (of size 9) of fields:

   ```
   typedef vector<field> state;
   ```

   Already given (in `5-library.cpp`) is a function

   ```
   field winner(const state & s);
   ```

   that checks for a given state `s` whether one of the players has already won (i.e. marked a full column, row, or diagonal) and then returns the name of that winner. Otherwise it returns `draw` to indicate that nobody has won the game yet.

Given (in `5-library.cpp`) is also a function

```
vector<state> next(const state & s, field turn);
```

that takes as arguments a state and which player's turn it is, i.e. the variable `turn` must be either `X` or `O`. It then computes all the possible next states, i.e., where the player whose turn it is has marked one more field. If there are no more unmarked fields (the game is finished), then the function `next` will thus return a vector of size 0.

Task: Write a function

```
field best(const state & s, field turn);
```

that gets as arguments a state of the game and whose turn it is – where again `turn` must be either `X` or `O`. The result is the outcome if both players play perfect, i.e., each player at each step marks the field that gives the best outcome for that player.

This is best computed recursively:

- Check the base cases (that somebody has won or no more moves are possible).
- Otherwise, compute all next states for the current player and recursively compute `best(...)` for all next states when it is the opponents turn. The final result is the best of these outcomes for the current player (i.e. a draw is preferred over losing, winning is preferred over a draw).

The given code follows on the next page...

Public test cases:

For the initial empty matrix (all fields set to `draw`), the best outcome (for either player) is `draw`, i.e., there is no winning strategy for this game.

In the state (where the unmarked fields are here shown empty instead of writing "draw")

X		O

the player whose turn it is will win (when playing perfect).

File `5-library.h` (Do not modify)

```
#ifndef __TTT
#define __TTT
#include<iostream>
#include<vector>
using namespace std;

enum field {O,X,draw};

typedef vector<field> state;
// 0 1 2
// 3 4 5
// 6 7 8

// given a state, check if any has already won, and return draw otherwise.
field winner(const state & s);

// given a state and whose turn it is (turn!=draw!), compute the set of next states.
vector<state> next(const state & s, field turn);

// given a state and whose turn it is (turn!=draw), compute the best
// possible outcome for that player
field best(const state & s, field turn);

string show(const field p);
#endif
```

File `5-library.cpp` (Hand in only this file)

```
#include<iostream>
#include "5-library.h"
#include<vector>
using namespace std;

#define check(a,b,c) if(s[a]==s[b] && s[b]==s[c] && s[c]!=draw) return s[c]
#define inv(a) a==O?X:O

field winner(const state & s){
  check(0,1,2);
  check(3,4,5);
  check(6,7,8);
  check(0,3,6);
  check(1,4,7);
  check(2,5,8);
  check(0,4,8);
  check(2,4,6);
  return draw;
}

vector<state> next(const state & s, field turn){
  vector<state> choices;
  for(int i=0; i<9; i++){
    if(s[i]==draw){
      state s2(s);
      s2[i]=turn;
      choices.push_back(s2);
    }
  }
  return choices;
}
```

```
field best(const state & s, field turn){
  // to program
}

string show(const field p){
  if (p==X) return "X";
  if (p==O) return "O";
  if (p==draw) return "-";
  return "?";
}
```

File `5-main.cpp` (Do not modify)

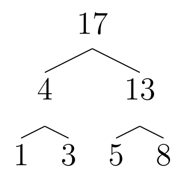
```
#include "5-library.h"
#include <iostream>
using namespace std;

int main(){
  state s(9);
  for(int i=0; i<9; i++) s[i]=draw;

  cout << "projected␣winner:␣" << show(best(s,X)) << endl;

  s[0]=X;
  cout << "projected␣winner:␣" << show(best(s,O)) << endl;

  s[2]=O;
  cout << "projected␣winner:␣" << show(best(s,X)) << endl;
}
```

The `5-main.cpp` also represents a test, the output of that program should be:

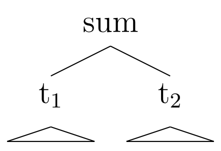
```
projected winner: -
projected winner: -
projected winner: X
```

6. **Huffman Trees (15 Points)**

   Huffman Trees are trees where each node is labeled with a number and each inner node is the sum of the value of its two children. For instance:



- Write a class `Huffman` to represent a node of such a tree, so it should contain an integer value and two pointers for the subtrees as private members.
- There should be two constructors:
  - one constructor that takes an integer as argument and constructs a single leaf from it.
  - another constructor takes two (pointers to) trees t_1 and t_2 and and builds a tree



    where *sum* is the sum of the values of the two trees. This constructor should not make a copy of the given trees.

  Thus, the Huffman tree depicted above can be constructed by the following code:

```
Huffman *t1=new Huffman(1); Huffman *t2=new Huffman(3);
Huffman *t3=new Huffman(t1,t2);
Huffman *t4=new Huffman(5); Huffman *t5=new Huffman(8);
Huffman *t6=new Huffman(t4,t5);
Huffman *t =new Huffman(t3,t6);
```

- Write a method for printing the tree so that the above example tree looks as follows:

```
[17[4[1][3]][13[5][8]]]
```

- Write a destructor that recursively deletes the entire tree.

The given code follows on the next page...

File `6-library.h` (Hand in only this file)

```
#ifndef __HUFFMAN
#define __HUFFMAN

class Huffman{
  // declare some private variables;

 public:
  Huffman(int w){
    // To be written
  }
  Huffman(Huffman *t1, Huffman *t2){
    // To be written
  }
  ~Huffman(){
    // To be written
  }
  void display(){
    // To be written
  }

};

#endif
```

File `6-main.cpp` (Do not modify)

```
#include <iostream>
using namespace std;
#include "6-library.h"

int main(){
  Huffman *t1=new Huffman(1);
  Huffman *t2=new Huffman(3);
  Huffman *t3=new Huffman(t1,t2);
  Huffman *t4=new Huffman(5);
  Huffman *t5=new Huffman(8);
  Huffman *t6=new Huffman(t4,t5);
  Huffman *t =new Huffman(t3,t6);

  t->display();

}
```

Note: For this exercise, please implement the solutions directly in the file `6-library.h` and hand in this file on CampusNet (there is **no** `6-library.cpp` in this case).