Technical University of Denmark

Written examination date: December 08, 2019

Course title: Programming in C++

Course number: 02393

Aids allowed: All aids allowed

Exam duration: 4 hours

Weighting: pass/fail

Exercises: 4 exercises of 2.5 points each for a total of 10 points.

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 and you can do it only once: resubmission is not possible, so submit only when you have finished all exercises. Each exercise must be uploaded as one separate `.cpp` file, using the names specified in the exercises, namely `exZZ-library.cpp`, where `ZZ` ranges from `01` to `04`. **Exercise 4 requires also the upload of file** `ex04-library.h`. 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. You can also upload your solutions on CodeJudge under `Exam December 2019` at `https://dtu.codejudge.net/02393-e19/exercises`. This is not an official submission, and will not be considered for evaluation. When you hand in a solution on CodeJudge, some tests will be run on it. Additional tests may be run on your solutions after the exam. You can upload to CodeJudge as many times as you like.

Exercise 1. Checkers/Draughts (2.5 points)

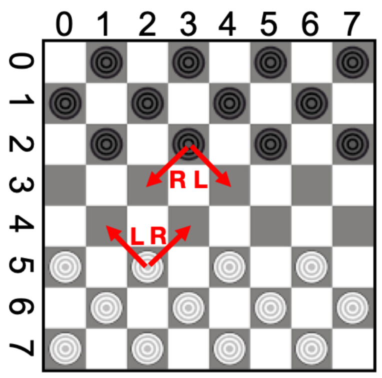

Figure 1: Initial status

Alice is implementing a library to play checkers (also known as draughts), a strategy board game for two players. Each player has pieces of a color, either black or white. The pieces are placed on a square chessboard consisting of cells of alternating bright and dark color. Several variants of the game exist, requiring chessboards of different size (e.g., 8×8 or 10×10 cells). Figure 1 shows an initial configuration for the so called "English draughts", using an 8×8 chessboard. The game proceeds in turns. In each turn, one player can move one of its pieces by one cell. Pieces can be moved only in diagonal direction, and cannot be moved backward. With respect to Figure 1: black pieces can be moved downwards, while white ones upwards. Therefore, the black piece in row 2 and column 3 (in short, (2,3)) can be moved to its left (`L`) getting in (3,4) or to its right (`R`) getting in (3,2). Instead, the white piece in position (5,2) can be moved to its left (`L`) getting in (4,1), or to its right (`R`) getting in (4,3). Pieces cannot move outside the chessboard, and cannot move in a cell already occupied by another piece.

Alice is only interested in representing the chessboard, the pieces, and their movements as described above. Alice has already implemented part of the code but some parts are still missing. Her first test program is in file `ex01-main.cpp` and the (incomplete) code with some functions she needs is in files `ex01-library.h` and `ex01-library.cpp`. All files are available on CampusNet and in the next pages. Cells of the chessboard are represented using a `struct cell` containing information on its color (either `bright` or `dark`), and its status (either `emptyC`, or occupied by a `whitePiece` or by a `blackPiece`). The chessboard is represented using a two-dimensional array of `cells`. This array has to be allocated dynamically, so to support allow for using chessboards of different size.

Help Alice by solving the following tasks:

(a) Implement function

```
cell ** createAndInitBoard(int n);
```

It should create and return a two dimensional array of size `n`×`n` of `cell`, i.e. `cell **`. The function should allocate the required memory, and initialize each cell:

- Each cell should get a color, computed using the function `computeColor(r,c)`, where the parameters denote the row and column of the cell;
- The dark cells in the top 3 rows should get status `blackPiece`;
- The dark cells in the bottom 3 rows should get status `whitePiece`;
- All the other cells should get status `emptyC`.

Note: by invoking the function with `n=8` we should get the configuration in Figure 1.

Exercise follows in next page...

(b) Implement function

```
cell ** duplicateBoard(cell ** A, int n);
```

The function should create and return a copy of the chessboard `A` of size `n`×`n`.

(c) Implement function

```
bool moveRight(cell ** A,int n,int r, int c);
```

This function should move the piece in row `r` and column `c` (if any) by one cell in diagonal, to its right (`R`). Note that, as shown in Figure 1, *moving to its right* has a different meaning whether the piece is black or white. The movement should be performed only if it is admissible, in which case `true` should be returned. Otherwise the value `false` should be returned, without modifying the chessboard (i.e. without applying any change).

Notes: A movement is **not** admissible if:

- The source cell (r,c) does not contain a piece, or
- The target cell is outside the chessboard, or
- The target cell is occupied by another piece.

(d) Implement function

```
bool moveLeft(cell ** A,int n,int r, int c);
```

This function is similar to `moveRight`, with the only difference being that the movement should be to the left (`L`) of the considered piece. Note that, as shown in Figure 1, *moving to its left* has a different meaning whether the piece is black or white. The movement should be performed only if it is admissible, in which case `true` should be returned. Otherwise the value `false` should be returned, without modifying the chessboard (i.e. without applying any change).

Exercise follows in next page...

File `ex01-main.cpp`

```cpp
#include <iostream>
#include "ex01-library.h"

using namespace std;

void testMoveRight(cell ** b,int n,int r,int c){
  bool moved=moveRight(b,n,r,c);
  cout<<"(" <<r<<','<<c<<")␣can";
  cout<<(moved?"":"'t") <<"␣move␣right"<<endl;
  printBoard(b,n);
}
void testMoveLeft(cell ** b,int n,int r,int c){
  bool moved=moveLeft(b,n,r,c);
  cout<<"(" <<r<<','<<c<<")␣can";
  cout<<(moved?"":"'t") <<"␣move␣left"<<endl;
  printBoard(b,n);
}

int main() {
  int n=8;
  cell ** b = createAndInitBoard(n);
  printBoard(b,n);

  cell ** copy = duplicateBoard(b,n);

  testMoveRight(b,n,2,1);
  testMoveRight(b,n,3,0);
  testMoveLeft(b,n,3,0);

  testMoveRight(b,n,5,6);
  testMoveLeft(b,n,4,7);

  cout <<"The␣copy:"<<endl;
  printBoard(copy,n);
  return 0;
}
```

File `ex01-library.h`

```cpp
#ifndef EX01_LIBRARY_H_
#define EX01_LIBRARY_H_

#include <iostream>

enum cellColor {bright, dark};
enum cellStatus {emptyC, whitePiece, blackPiece};
struct cell{
  cellColor color;
  cellStatus status;
};

cell ** createAndInitBoard(int n);
cell ** duplicateBoard(cell ** A, int n);
bool moveRight(cell ** A,int n,int r, int c);
bool moveLeft(cell ** A,int n,int r, int c);

cellColor computeColor(int r,int c);
void printBoard(cell ** A, int n);

#endif /* EX01_LIBRARY_H_ */
```

File `ex01-library.cpp`

```cpp
#include <iostream>
#include "ex01-library.h"

using namespace std;

//Exercise 1 (a) Implement this function
cell ** createAndInitBoard(int n){
  //put your code here
}

//Exercise 1 (b) Implement this function
cell ** duplicateBoard(cell ** A, int n){
  //put your code here
}

//Exercise 1 (c) Implement this function
bool moveRight(cell ** A,int n,int r, int c){
  //put your code here
}

//Exercise 1 (d) Implement this function
bool moveLeft(cell ** A,int n,int r, int c){
  //put your code here
}

//Do not modify
cellColor computeColor(int r,int c){
  if((r%2 == 0 && c%2 == 0) || (r%2 != 0 && c%2 != 0)){
    return bright;
  }
  return dark;
}

//Do not modify
void printBoard(cell ** A, int n){
  cout << '␣';
  for(int c=0;c<n;c++){
    cout << c;
  }
  cout << endl;
  for(int r=0;r<n;r++){
    cout << r;
    for(int c=0;c<n;c++){
      //Print ' ' or '#' if emptyC, or the piece color (W or B)
      switch (A[r][c].status) {
      case emptyC:
        cout << ((A[r][c].color==bright)?'␣':'#');
        break;
      case whitePiece:
        cout << "W";
        break;
      case blackPiece:
        cout << "B";
        break;
      }
    }
    cout << endl;
  }
  cout << endl;
}
```

Exercise 2. Palindromes (2.5 points)

Bob wants to build a library for performing operations on words. He wants to compute the reverse of a word (e.g., from `"abc"` to `"cba"`), and check if a word is palindrome. A word is palindrome if by reading it backward starting from the last letter we get the very same word, that is, it is equal to its reverse. For example `"abba"` is palindrome, while `"abca"` is not, because its reverse is `"acba"`. The word `"abca"` becomes palindrome by changing `'c'` in `'b'`, getting `"abba"`. The reverse of `""` is `""`, which is therefore palindrome.

Bob has already written some code. His first test program is in file `ex02-main.cpp` and the (incomplete) code with some functions he needs is in files `ex02-library.h` and `ex02-library.cpp`. All files are available on CampusNet and in the next pages.

Bob has decided to represent a word using a string. Bob knows that this kind of problems can be solved using recursion. Help Bob by solving the following tasks:

(a) Implement the recursive function

```
string reverse(string s, int n)
```

which computes and returns the reverse of the string `s`. The function is initially invoked with `n` equal to `s.length()-1`. Each invocation produces a new invocation with `n-1`, until a base case if found.

(b) Implement the recursive function

```
bool isPalindrome(string s, int n1, int n2)
```

which returns `true` if `s` is palindrome, and `false` otherwise. This function is initially invoked with `n1` equal to `0` and `n2` to `s.length()-1`. Each invocation produces the invocation of the function with `n1+1` and `n2-1`, until a base case is found.

(c) Implement the recursive function

```
int distancePalindrome(string s, int n1, int n2)
```

which returns the minimum number of characters to be changed to make `s` palindrome (`0` if `s` is palindrome). This function is initially invoked with `n1` equal to `0` and `n2` equal to `s.length()-1`. Each invocation produces the invocation of the function with `n1+1` and `n2-1`, until a base case is found.

For example, invoking `distancePalindrome` for

- `"ab"` gives `1`. Replacing `'b'` with `'a'` we get `"aa"`.
- `"abc"` gives `1`. Replacing `'c'` with `'a'` we get `"aba"`.
- `"abcdba"` gives `1`. Replacing `'d'` with `'c'` we get `"abccba"`.
- `"abcdea"` gives `2`. Replacing `'d'` with `'c'` and `'e'` with `'b'` we get `"abccba"`.
- `"abcdef"` gives `3`. We need to change `'d'` in `'c'`, `'e'` in `'b'`, and `'f'` in `'a'`.

Exercise follows in next page...

File ex02-main.cpp

```cpp
#include <iostream>

#include "ex02-library.h"
using namespace std;

void testReverse(string s){
  string rev = reverse(s,s.length()-1);
  cout << s << "␣|␣" << rev << endl;
}

void testPalindrome(string s){
  bool pal = isPalindrome(s,0,s.length()-1);
  cout << s << ((pal)?"␣is":"␣is␣not");
  cout <<"␣palindrome" << endl;
}

void testDistancePalindrome(string s){
  int k = distancePalindrome(s,0,s.length()-1);
  cout << s << "␣requires␣" << k;
  cout << "␣changes␣to␣be␣palindrome"<< endl;
}

int main() {
  string words[] = {"","a","aa","ab","aba","abc",
    "abccba","abcdba","abcdea","abcdef","abcdefg"};
  int length=11;

  cout << "REVERSE" << endl;
  for(int i=0;i<length;i++){
    testReverse(words[i]);
  }

  cout << endl << "PALINDROME" << endl;
  for(int i=0;i<length;i++){
    testPalindrome(words[i]);
  }

  cout << endl << "DISTANCE-PALINDROME" << endl;
  for(int i=0;i<length;i++){
    testDistancePalindrome(words[i]);
  }

  return 0;
}
```

File ex02-library.h

```cpp
#ifndef EX02_LIBRARY_H_
#define EX02_LIBRARY_H_

#include <string>

using namespace std;

string reverse(string s, int n);
bool isPalindrome(string s, int n1, int n2);
int distancePalindrome(string s, int n1, int n2);

#endif /* EX02_LIBRARY_H_ */
```

File ex02-library.cpp

```cpp
#include "ex02-library.h"
#include <iostream>

//Exercise 2 (a) Implement this function
string reverse(string s, int n){
  //put your code here
}

//Exercise 2 (b) Implement this function
bool isPalindrome(string s, int n1, int n2){
  //put your code here
}

//Exercise 2 (c) Implement this function
int distancePalindrome(string s, int n1, int n2){
  //put your code here
}
```

Exercise 3. Palindromes 2 (2.5 points)

Similarly to Bob in Exercise 2, Claire wants to build a library for computing the reverse of a word (e.g., from `"abc"` to `"cba"`), and for checking if a word is palindrome. A word is palindrome if by reading it backward starting from the last letter we get the very same word (it is equal to its reverse). E.g., `"abba"` is palindrome, while `"abca"` is not, because its reverse is `"acba"`. The word `"abca"` can be made palindrome by changing one letter only, `'c'` in `'b'`, obtaining `"abba"`. The reverse of `""` is `""`, which is therefore palindrome.

Claire has already written some code. Her first test program is in file `ex03-main.cpp` and the (incomplete) code with some functions she needs is in files `ex03-library.h` and `ex03-library.cpp`. All files are available on CampusNet and in the next pages. Help Claire by implementing the class `WordsList` in file `ex03-library.cpp`.

Differently from Bob, Claire decided to use object-oriented programming, creating a class containing the list of strings of interest. Also, she is not interested in recursive programming, therefore she wants to implement all the methods following the iterative approach. Claire decided to use the `vector` and `map` containers of the standard library. She has decided to use the following internal (`private`) representation for the library:

- `vector<string> allWords`: A vector containing the words of interest.
- `map<string,int> wordsToPalindromeDistance`: A map from each word in `allWords` to the minimum number of characters to make the considered word a palindrome.

The general idea is that whenever a new word is added, both containers `allWords` and `wordsToPalindromeDistance` should be updated accordingly.

Claire already implemented the default constructor of `WordsList`, which adds two default words: `"121"` and `"122"`. Help Claire by performing the following tasks:

(a) Implement method

```
void WordsList::print()
```

The method should correctly print information on the added words in this form:

```
n words: word1 (d1) word2 (d2) ... wordn (dn)
```

where `n` is the number of words in `allWords`, `wordi` is the i_th word, and `di` is the minimum number of characters to make `wordi` palindrome, as stored in the `map`. Note that each word is preceded by a blank space.

Exercise follows in next page...

(b) Implement method

```
int WordsList::distancePalindrome(string s)
```

which should compute and return the minimum number of characters of `s` to be changed to make it palindrome. This method should not use recursion.

(c) Implement method

```
void WordsList::addWord(string word)
```

which should add the word `word` to `allWords`, and update `wordsToPalindromeDistance` accordingly.

(d) Implement method

```
WordsList::palindromeWords()
```

which should compute and return the number of palindrome words in the object.

Exercise follows in next page...

File `ex03-main.cpp`

```
#include <iostream>

#include "ex03-library.h"
using namespace std;

int main() {
  string words[] = {"","a","aa","ab","aba","abc",
    "abccba", "abcdba","abcdea","abcdef","abcdefg"};
  int length=11;

  WordsList wl;
  cout << "The␣initial␣configuration" << endl;
  wl.print();

  cout << endl;
  cout << "Are␣the␣strings␣in␣'words'␣palindromes?";
  cout << endl;
  for(int i=0;i<length;i++){
    int dist = wl.distancePalindrome(words[i]);
    cout << words[i] << "␣requires␣" << dist ;
    cout << "␣changes␣to␣be␣palindrome" << endl;
  }

  cout << endl ;
  int pWords = wl.palindromeWords();
  cout << "The␣initial␣configuration␣contains␣" ;
  cout << pWords << "␣palindromes" << endl;

  for(int i=0;i<length;i++){
    wl.addWord(words[i]);
  }
  cout << endl << "The␣final␣list" << endl;
  wl.print();

  cout << endl;
  pWords = wl.palindromeWords();
  cout << "The␣final␣list␣contains␣";
  cout << pWords << "␣palindromes" << endl;

  return 0;
}
```

File `ex03-library.h`

```
#ifndef EX03_LIBRARY_H_
#define EX03_LIBRARY_H_

#include <string>
#include <vector>
#include <map>

using namespace std;

class WordsList {
private:
  vector<string> allWords;
  map<string,int> wordsToPalindromeDistance;
public:
  WordsList();
  void print();
  int distancePalindrome(string s);
  void addWord(string word);
  int palindromeWords();
};

#endif /* EX03_LIBRARY_H_ */
```

File `ex03-library.cpp`

```
#include "ex03-library.h"
#include <iostream>

//Do not modify
WordsList::WordsList(){
  allWords.push_back("121");
  wordsToPalindromeDistance["121"]=0;
  allWords.push_back("122");
  wordsToPalindromeDistance["122"]=1;
}

//Exercise 3 (a) implement this method
void WordsList::print(){
  //put your code here
}

//Exercise 3 (b) implement this method
int WordsList::distancePalindrome(string s){
  //put your code here
}

//Exercise 3 (c) implement this method
void WordsList::addWord(string word){
  //put your code here
}

//Exercise 3 (d) implement this method
int WordsList::palindromeWords(){
  //put your code here
}
```

Exercise 4. Checkers/Draughts 2 (2.5 points)

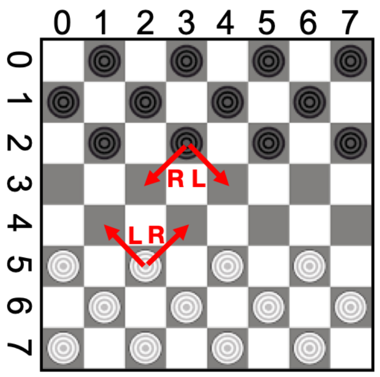

Figure 2: Initial status

Similarly to Alice in Exercise 1, Daisy wants to implement a library to play checkers (also known as draughts), a strategy board game for two players. Each player has pieces of a color, either black or white. Figure 2 shows an initial configuration for the so called “English draughts”, using an 8×8 chessboard. The game proceeds in turns. In each turn, one player can move one of its pieces by one cell. Pieces can be moved only in diagonal direction, and cannot be moved backward. With respect to Figure 2: black pieces can be moved downwards, while white ones upwards. Therefore, e.g., the black piece in row 2 and column 3 (in short, (2,3)) can be moved to its left (`L`) getting in (3,4) or to its right (`R`) getting in (3,2). Instead, the white piece in position (5,2) can be moved to its left (`L`) getting in (4,1), or to its right (`R`) getting in (4,3).

Daisy is only interested in representing the pieces and their movements as described above. The chessboard is ignored, and moves are always admissible in this exercise. Daisy has already implemented part of the code but some parts are still missing. Her first test program is in file `ex04-main.cpp` and the (incomplete) code with some functions she needs is in files `ex04-library.h` and `ex04-library.cpp`. All files are available on CampusNet and in the next pages. Daisy decided to declare an abstract class `Piece` which declares all the methods required by white and black pieces. First of all, every piece has two protected members `r` and `c` to store the position in terms of row and column. This position is printed by method `void Piece::printPosition()` independently on the color of the piece. Furthermore, `Piece` has the following color-specific pure virtual functions:

- `string getColor()` should return `"white"` if invoked on an white piece, and `"black"` if invoked on a black piece
- `string printCode()` should return `"W"` (for white) if invoked on a white piece, and `"B"` (for black) if invoked on a black piece
- `void moveRight()` should update the current row (`r`) and column (`c`) due to a movement to the right of the piece as explained above. Note that white and black pieces have a different notion of *move right*. Moves are always admissible.
- `void moveLeft()` should update the current row (`r`) and column (`c`) due to a movement to the left of the piece as explained above. Note that white and black pieces have a different notion of *move left*. Moves are always admissible.

Help Daisy by solving the following tasks:

(a) Declare in `ex04-library.h` and implement in `ex04-library.cpp` class WhitePiece, extending class Piece. It should implement a constructor with parameters the current row and column, as well as all *pure virtual functions* of Piece (i.e., those with =0).

Exercise follows in next page...

(b) Declare in `ex04-library.h` and implement in `ex04-library.cpp` class BlackPiece, extending class Piece. It should implement a constructor with parameters the current row and column, as well as all *pure virtual functions* of Piece (i.e., those with =0).

NOTE: For this exercises you are required to submit both files `ex04-library.h` and `ex04-library.cpp`

File `ex04-main.cpp`

```cpp
#include <iostream>
#include <vector>
#include "ex04-library.h"
using namespace std;

int main() {
  vector<Piece*> pieces;
  pieces.push_back(new BlackPiece(0,3));
  pieces.push_back(new BlackPiece(0,5));

  pieces.push_back(new WhitePiece(7,2));
  pieces.push_back(new WhitePiece(7,4));

  for(int i=0;i<pieces.size();i++){
    cout << "Piece␣" << i << "␣is␣";
    cout << pieces[i]->getColor();
    cout << "␣(" << pieces[i]->printCode();
    cout << ")" << endl;

    cout << "It␣is␣in␣position␣";
    pieces[i]->printPosition();

    pieces[i]->moveRight();
    cout << "After␣moving␣right:␣";
    pieces[i]->printPosition();

    pieces[i]->moveLeft();
    cout << "After␣moving␣left:␣";
    pieces[i]->printPosition();
    cout << endl;
  }
}
```

File `ex04-library.h`

```cpp
#ifndef EX04_LIBRARY_H_
#define EX04_LIBRARY_H_

#include <string>
using namespace std;

//Do not modify
class Piece {
protected:
  int r;
  int c;
public:
  virtual ~Piece();
  void printPosition();
  virtual string getColor()=0;
  virtual string printCode()=0;
  virtual void moveRight()=0;
  virtual void moveLeft()=0;
};

//Exercise 4 (a) declare WhitePiece
//put your code here

//Exercise 4 (b) declare BlackPiece
//put your code here

#endif
```

File `ex04-library.cpp`

```cpp
#include "ex04-library.h"
#include <iostream>

//Do not modify
Piece::~Piece(){}

//Do not modify
void Piece::printPosition(){
  cout << "(" << r << "," << c << ")" << endl;
}

//Exercise 4 (a) implement methods of WhitePiece
//put your code here

//Exercise 4 (b) implement methods of BlackPiece
//put your code here
```