Technical University of Denmark

Written examination date: 7 December 2020

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 **must** submit your solution on DTU Inside, under the course "Assignments:"

   `https://cn.inside.dtu.dk/cnnet/Assignments/student/623043`

   You can do it only once, so submit only when you have completed your work.

2. Each exercise must be submitted 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 also requires to submit the file** `ex04-library.h`. The files must be submitted separately (not as a Zip archive) and must have these exact filenames.

3. You can also test your solutions by uploading them on CodeJudge, under "`Exam December 2020`" at:

   `https://dtu.codejudge.net/02393-e20/exercises`

4. You can test your solutions on CodeJudge as many times as you like. *Uploads on CodeJudge are not official submissions* and will not affect your grade.

5. Additional tests may be run on your submissions after the exam.

6. Feel free to add comments to your code.

7. **Suggestion:** read all exercises before starting your work, and begin with the tasks that look easier.

Exercise 1. Complex Matrices (2.5 points)

Alice needs to perform computations on *complex matrices*, i.e., matrices having *complex numbers* as elements. A complex number has the form $a + bi$, where:

- a is a real number, and is called the "real part" of the complex number;
- b is also a real number, and is called the "imaginary part;"
- i is the *imaginary unit*: a value such that $i^2 = -1$.

Alice has already written some code. 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 DTU Inside and in the next pages.

Structure of the code. A complex number is represented as a `struct Complex` with two fields, named `re` and `im`: they are, respectively, the `re`al and `im`aginary part of the number. Alice's code already includes these functions:

```
Complex add(Complex c, Complex d)
Complex mult(Complex c, Complex d)
void deleteMatrix(Complex **A, unsigned int nRows)
```

The first two methods respectively add or multiply `c` and `d`, and return the result. The last function deallocates a matrix allocated with `createMatrix()` (see task **(a)** below).

Tasks. Help Alice by completing the following tasks.

(a) Implement the function:

```
Complex **createMatrix(unsigned int m, unsigned int n, Complex c)
```

The function must return an array of `m` × `n` `Complex` numbers, i.e., `Complex **`. It must allocate the required memory, and initialise each array element as argument `c`.

(b) Implement the function:

```
void displayMatrix(Complex **A, unsigned int m, unsigned int n)
```

The function must print the contents of matrix `A` on screen:

- each complex element must be printed *without* spaces between its real and imaginary parts;
- each imaginary part must be preceded by "+" if positive, or by "–" if negative;
- elements on a same row must be separated by one space;
- there must be no space after the last element of each row.

For example, a 2 × 4 matrix should look like:

```
1+2i 2+5i 4-4i 1+2i
1-2i 0-2i 0+2i 2+6i
```

This exercise continues on the next page...

(c) Implement the function:

```
Complex **createIdentityMatrix(unsigned int n)
```

The function must return a matrix of `n` × `n` `Complex` elements, where:

- each element on the diagonal is the complex number $1 + 0i$;
- each other element is the complex number $0 + 0i$.

The method must allocate the required memory. *Suggestion: the solution could be simplified by using `createMatrix()` from task **(a)**.*

(d) Implement the function:

```
void multMatrix(Complex **A, Complex **B, Complex **C,
                    unsigned int m, unsigned int n, unsigned int p)
```

Where:

- argument `A` is a complex matrix of size `m` × `n`;
- argument `B` is a complex matrix of size `n` × `p`;
- argument `C` is a complex matrix of size `m` × `p`.

The function must multiply `A` by `B`, storing the result in `C`. Therefore, as in standard matrix multiplication, the element at row i (for $0 \leq i < m$) and column j (for $0 \leq j < p$) of `C` is computed as:

$$\mathtt{C}_{i,j} \;=\; \sum_{0 \leq k < n} \mathtt{A}_{i,k} \cdot \mathtt{B}_{k,j}$$

Recall that the addition and multiplication of two complex numbers is already implemented in functions `add()` and `mult()` (see "Structure of the code" above). Therefore, such functions can be used to implement the operations "$\sum$" (complex summation) and "$\cdot$" (complex multiplication) in the formula above.

File `ex01-main.cpp`

```
#include <iostream>
#include "ex01-library.h"
using namespace std;

int main() {
    Complex c = {3, 1};
    Complex d = {2, -2};

    Complex **A = createMatrix(3, 3, c);
    cout << "Complex␣matrix␣A:" << endl;
    displayMatrix(A, 3, 3);
    cout << endl;

    Complex **I = createIdentityMatrix(3);
    cout << "Identity␣matrix␣I:" << endl;
    displayMatrix(I, 3, 3);
    cout << endl;

    Complex **R = createMatrix(3, 3, {0,0});
    cout << "Result␣of␣A␣*␣I:" << endl;
    multMatrix(I, A, R, 3, 3, 3);
    displayMatrix(R, 3, 3);
    cout << endl;

    Complex **B = createMatrix(3, 2, c);
    cout << "Complex␣matrix␣B:" << endl;
    displayMatrix(B, 3, 2);
    cout << endl;

    Complex **C = createMatrix(2, 3, d);
    cout << "Complex␣matrix␣C:" << endl;
    displayMatrix(C, 2, 3);
    cout << endl;

    cout << "Result␣of␣B␣*␣C:" << endl;
    multMatrix(B, C, R, 3, 2, 3);
    displayMatrix(R, 3, 3);

    deleteMatrix(A, 3); deleteMatrix(B, 3);
    deleteMatrix(C, 2);
    deleteMatrix(R, 3); deleteMatrix(I, 3);
    return 0;
}
```

File `ex01-library.h`

```
#ifndef EX01_LIBRARY_H_
#define EX01_LIBRARY_H_

struct Complex {
    double re; // Real part
    double im; // Imaginary part
};

Complex add(Complex c, Complex d);
Complex mult(Complex c, Complex d);
Complex **createMatrix(unsigned int m, unsigned int n, Complex c);
void displayMatrix(Complex **A, unsigned int m, unsigned int n);
Complex **createIdentityMatrix(unsigned int n);
void multMatrix(Complex **A, Complex **B, Complex **C,
                unsigned int m, unsigned int n, unsigned int p);
void deleteMatrix(Complex **A, unsigned int nRows);

#endif /* EX01_LIBRARY_H_ */
```

File `ex01-library.cpp`

```
#include <iostream>
#include "ex01-library.h"
using namespace std;

// Task 1(a). Implement this function
Complex **createMatrix(unsigned int m, unsigned int n, Complex c) {
    // Write your code here
}

// Task 1(b). Implement this function
void displayMatrix(Complex **A, unsigned int m, unsigned int n) {
    // Write your code here
}

// Task 1(c). Implement this function
Complex **createIdentityMatrix(unsigned int n) {
    // Write your code here
}

// Task 1(d). Implement this function
void multMatrix(Complex **A, Complex **B, Complex **C,
                unsigned int m, unsigned int n, unsigned int p) {
    // Write your code here
}

// Do not modify
Complex add(Complex c, Complex d) {
    Complex result = { c.re + d.re, c.im + d.im };
    return result;
}

// Do not modify
Complex mult(Complex c, Complex d) {
    Complex result;
    result.re = (c.re * d.re) - (c.im * d.im);
    result.im = (c.re * d.im) + (c.im * d.re);
    return result;
}

// Do not modify
void deleteMatrix(Complex **A, unsigned int nRows) {
    for (unsigned int i = 0; i < nRows; i++) { delete[] A[i]; }
    delete[] A;
}
```

Exercise 2. RLE Linked List (2.5 points)

Bob wants to build a linked list with a compression technique called *Run-Length Encoding (RLE)*: each element of the list records on how many times its value is repeated. For instance, the following sequence of values

```
1 1 25 3 3 3 3 3 42 42 5 5 5 5 5 5 5 5 5 5 42 42 42 42 42 42 42 42
```

is compressed with RLE as a sequence of values followed by the number of repetitions:

$1_{(\times 2)}$ $25_{(\times 1)}$ $3_{(\times 5)}$ $42_{(\times 2)}$ $5_{(\times 10)}$ $42_{(\times 8)}$

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 DTU Inside and in the next pages.

Structure of the code. An RLE list element is represented as a `struct Elem` with three fields, named `value`, `times`, and `next`: they are, respectively, the value of the list element, the number of times that value is repeated, and the pointer to the next list element (or `nullptr` when there are no more elements). An empty list is represented as an `Elem*` pointer equal to `nullptr`. Bob's code already includes the function:

```
void displayRLEList(Elem *list)
```

which prints an RLE list on screen, in the compressed form shown above.

Tasks. Help Bob by completing the following tasks.

(a) Implement the function:

```
unsigned int length(Elem *list)
```

which computes and returns the length of the RLE `list` — that is, the number of *values* contained in the list, taking into account their repetitions. For example: the RLE list $7_{(\times 25)}$ $9_{(\times 90)}$ only has two elements, but its length is 25 + 90 = 115.

Note: this function can be implemented recursively; it can invoke itself on the next element of the list, until the base case (i.e., the list pointer `nullptr`) is reached.

This exercise continues on the next page...

(b) Implement the function:

```
Elem* append(Elem *list, int v)
```

which appends value `v` at the end of `list`, and returns a pointer to the first `Element` of the updated list. The function must compress the repetitions of the new value: e.g., if the given RLE list is $\texttt{7}_{(\times 2)}$ $\texttt{6}_{(\times 1)}$ $\texttt{9}_{(\times 2)}$, then appending value `9` must return the RLE list $\texttt{7}_{(\times 2)}$ $\texttt{6}_{(\times 1)}$ $\texttt{9}_{(\times 3)}$.

(c) Implement the function:

```
Elem* buildRLEList(int *data, unsigned int n)
```

which builds a new RLE list using `n` integer values read from the array `data`, and returns a pointer to the first element of the new list. The function must compress the repetitions: e.g., if `data` contains the 5 values `7 7 6 9 9`, then the resulting RLE list must have 3 elements: $\texttt{7}_{(\times 2)}$ $\texttt{6}_{(\times 1)}$ $\texttt{9}_{(\times 2)}$.

*Suggestion: the solution can be simplified by using **`append()`** from task **(b)**.*

File ex02-main.cpp

```
#include <iostream>
#include "ex02-library.h"
using namespace std;

int main() {
    Elem e0 = {10, 5, nullptr};
    Elem e1 = {12, 6, &e0};
    Elem e2 = {4, 10, &e1};

    cout << "The␣RLE␣list␣is:␣" << endl;
    displayRLEList(&e2);
    cout << endl;

    cout << "Its␣lenght␣is:␣" << length(&e2) << endl;

    cout << endl;

    int data[] = {1, 2, 3, 3, 3, 4, 5};
    Elem *list = buildRLEList(data, 7);
    cout << "The␣new␣RLE␣list␣is:␣" << endl;
    displayRLEList(list);
    cout << endl;

    cout << "Its␣lenght␣is:␣" << length(list) << endl;
    Elem *list2 = append(list, 5);
    cout << "After␣we␣append␣5,␣the␣resulting␣RLE␣list␣is:␣" << endl;
    displayRLEList(list2);
    cout << endl;

    return 0;
}
```

File ex02-library.h

```
#ifndef EX02_LIBRARY_H_
#define EX02_LIBRARY_H_

struct Elem {
    int value;
    unsigned int times; // Number of repetitions
    Elem *next;
};

void displayRLEList(Elem *list);

unsigned int length(Elem *list);
Elem* append(Elem *list, int v);
Elem* buildRLEList(int *data, unsigned int n);

#endif /* EX02_LIBRARY_H_ */
```

File ex02-library.cpp

```
#include <iostream>
#include "ex02-library.h"
using namespace std;

// Task 2(a). Implement this function
unsigned int length(Elem *list) {
    // Write your code here
}

// Task 2(b). Implement this function
Elem* append(Elem *list, int v) {
    // Write your code here
}

// Task 2(c). Implement this function
Elem* buildRLEList(int *data, unsigned int n) {
    // Write your code here
}

// Do not modify
void displayRLEList(Elem *list) {
    if (list == nullptr) {
        return;
    }
    cout << "␣" << list->value << "␣(x" << list->times << ")";
    displayRLEList(list->next);
}
```

Exercise 3. Song Evaluations (2.5 points)

Claire wants to implement a class `SongDatabase` to store information about some of the songs she knows. She has already written some code: her first test program is in file `ex03-main.cpp` and the (incomplete) code of the class is in files `ex03-library.h` and `ex03-library.cpp`. All files are available on DTU Inside and in the next pages.

Structure of the code. Claire has represented song information using a `struct Info`, with two fields:

- `url`: an address to listen to the song;
- `score`: how much she likes the song, from `0` to `10`.

Claire knows that the `map` and `vector` containers of the C++ standard library provide many functionalities she needs. *(See hints on page 9.)* Therefore, she has decided to use the following internal (`private`) representation for the library:

- `vector<string> songs` — the titles of the songs in the database;
- `map<string,Info> songsInfo` — a mapping from `strings` (song titles) to instances of `Info` (the information about the song).

Claire has already implemented the default constructor of `SongDatabase`, which creates a database with some songs she knows.

Tasks. Help Alice by completing the following tasks.

(a) Implement the following method:

```
void SongDatabase::display()
```

The method must print the database contents on screen, by following the order in which song titles appear in the `songs` vector. For each song, the method must print a line of the form:

title=*songTitle*; url=*songUrl*; score=*songScore*

where *songTitle*, *songUrl* and *songScore* are, respectively, the title, URL, and score of a song in the database (notice that *songUrl* and *songScore* are stored in the `Info` of each song).

This exercise continues on the next page...

(b) Implement the following method to add songs to the database:

```
bool SongDatabase::addSong(string title, string url, unsigned int score)
```

The method must work as follows:

(a) if `title` is already in the database, do *not* change anything and return `false`;

(b) if `score` is greater than 10, do *not* change the database and return `false`;

(c) otherwise, when neither *(a)* nor *(b)* above apply: add the given `title` at the end of the `songs` vector, map it to the given `url` and `score` (by updating `songsInfo`), and return `true`.

(c) This is a follow-up to point **(b)**. Implement the method:

```
void SongDatabase::searchSongs(string howGood)
```

This method must print a list of songs with the same format of `display()` (see point **(a)** above) — but only show songs that satisfy the argument `howGood`, which can be one of the following:

- `"abysmal"`: only show songs with $0 \leq$ `score` < 3;
- `"lousy"`: only show songs with $3 \leq$ `score` < 5;
- `"meh"`: only show songs with $5 \leq$ `score` < 7;
- `"cool"`: only show songs with $7 \leq$ `score` < 9;
- `"OMG"`: only show songs with `score` ≥ 9.

Special case: if `howGood` is none of the above, the method must not print anything.

The song titles and their information must be printed by following the order in which they appear in the `songs` vector.

Hints on using `maps`

- A key `k` in a map `m` can be mapped to `v` with: `m[k] = v;` with this operation, the entry for `k` in `m` is created (if not already present) or updated (if already present).
- To check if key `k` is present in map `m`, you can check: `m.find(k) != m.end()`.
- The value mapped to a key `k` in a map `m` is obtained with: `m[k]`.

File `ex03-main.cpp`

```cpp
#include <iostream>
#include "ex03-library.h"
using namespace std;

int main() {
    SongDatabase db = SongDatabase();

    db.addSong("Ob-La-Di,␣Ob-La-Da", "https://youtu.be/_J9NpHKrKMw", 4);
    db.addSong("I␣am␣the␣Walrus", "https://youtu.be/t1Jm5epJr10", 8);
    db.addSong("Leave␣My␣Kitten␣Alone", "https://youtu.be/t1Jm5epJr10", 8);

    cout << "The␣database␣contains␣the␣songs:" << endl;
    db.display();

    cout << endl << "Abysmal␣songs:" << endl;
    db.searchSongs("abysmal");

    cout << endl << "Lousy␣songs:" << endl;
    db.searchSongs("lousy");

    cout << endl << "Meh␣songs:" << endl;
    db.searchSongs("meh");

    cout << endl << "Cool␣songs:" << endl;
    db.searchSongs("cool");

    cout << endl << "OMG␣songs:" << endl;
    db.searchSongs("OMG");

    return 0;
}
```

File `ex03-library.h`

```cpp
#ifndef EX03_LIBRARY_H_
#define EX03_LIBRARY_H_

#include <string>
#include <vector>
#include <map>
using namespace std;

struct Info {
    string url;
    unsigned int score;
};

class SongDatabase {
private:
    vector<string> songs;
    map<string,Info> songsInfo;
public:
    SongDatabase();
    void display();
    bool addSong(string title, string url, unsigned int score);
    void searchSongs(string howGood);
};

#endif /* EX03_LIBRARY_H_ */
```

File `ex03-library.cpp`

```
#include <iostream>
#include "ex03-library.h"
using namespace std;

// Do not modify
SongDatabase::SongDatabase() {
    this->songs.push_back("Penny␣Lane");
    this->songsInfo["Penny␣Lane"] = {"https://youtu.be/S-rB0pHI9fU", 8};

    this->songs.push_back("Trololo");
    this->songsInfo["Trololo"] = {"https://youtu.be/oavMtUWDBTM", 10};

    this->songs.push_back("Ob-La-Di,␣Ob-La-Da");
    this->songsInfo["Ob-La-Di,␣Ob-La-Da"] = {"https://youtu.be/_J9NpHKrKMw", 2};

    this->songs.push_back("Don’t␣Worry,␣Be␣Happy");
    this->songsInfo["Don’t␣Worry,␣Be␣Happy"] = {"https://youtu.be/d-diB65scQU", 3};

    this->songs.push_back("Leave␣My␣Kitten␣Alone");
    this->songsInfo["Leave␣My␣Kitten␣Alone"] = {"https://youtu.be/7BKsy9-Bvok", 5};
}

// Task 3(a). Implement this method
void SongDatabase::display() {
    // Write your code here
}

// Task 3(b). Implement this method
bool SongDatabase::addSong(string title, string url, unsigned int score) {
    // Write your code here
}

// Task 3(a). Implement this method
void SongDatabase::searchSongs(string howGood) {
    // Write your code here
}
```

Exercise 4. Limited Buffer (2.5 points)

Daisy needs to develop a buffer class to store and retrieve integer values. She plans an interface consisting of 3 methods:

- `write(v)` — appends value `v` to the buffer;
- `read()` — removes the oldest value from the buffer and returns it;
- `occupancy()` — returns the number of elements in the buffer. Calling `write()` increases the occupancy by 1, while calling `read()` decreases the occupancy by 1.

Therefore, the buffer works in FIFO (First-In-First-Out) order: e.g., if `write()` is invoked to append `1`, and then invoked again to append `2`, then a subsequent call to `read()` must return `1`, and a further call must return `2`.

For her application, Alice needs to implement a *limited* buffer that accumulates values up-to a maximum capacity; when the maximum capacity is reached, then further calls to `write()` have no effect, until some value is removed using `read()`.

Her first test program is in file `ex04-main.cpp` and the (incomplete) code of the class is in files `ex04-library.h` and `ex04-library.cpp`. All files are available on DTU Inside and in the next pages.

Structure of the code. Daisy has defined a high-level abstract class `Buffer` with the pure virtual methods `write()`, `read()`, and `occupancy()`.

Tasks. Help Alice by completing the following tasks. **IMPORTANT:** for these tasks you are required to submit ***both*** files `ex04-library.h` and `ex04-library.cpp`.

(a) Declare in `ex04-library.h` and sketch in `ex04-library.cpp` a class `LimitedBuffer` that extends `Buffer`. This task is completed (and passes CodeJudge tests) when `ex04-main.cpp` compiles without errors. To achieve this, you will need to:

1. define a constructor for `LimitedBuffer` that takes 2 parameters:

   (i) an `unsigned int`eger representing the maximum buffer capacity; and

   (ii) a value of type `int` representing a default (it is used in point **(c)** below);

2. in `LimitedBuffer`, override the *pure virtual methods* of `Buffer` (i.e., those with "`=0`"), and write (possibly non-working) placeholder implementations.

This exercise continues on the next page...

(b) This is a follow-up to point **(a)** above. In `ex04-library.cpp`, write a working implementation of the methods:

```
void LimitedBuffer::write()
unsigned int LimitedBuffer::occupancy()
```

Their intended behaviour is that each time `write()` is invoked, the value returned by `occupancy()` increases by 1, until the maximum capacity is reached (such maximum capacity is specified in the constructor — see point **(a)**1(i) above). While the occupancy is at its maximum, calls to `write()` have no effect.

(c) This is a follow-up to points **(a)** and **(b)** above. In `ex04-library.cpp`, write a working implementation of the method:

```
int LimitedBuffer::read()
```

When `read()` is invoked, it removes the oldest value previously added by `write()`, and returns it; correspondingly, the value returned by `occupancy()` decreases by 1.

Special case: if the buffer is empty, then `read()` must return the default value specified in the constructor (see point **(a)**1(ii) above).

NOTE: you are free to define the `private` members of `LimitedBuffer` however you see fit. For instance, you might choose to store the values in a `vector<int>`, or in a linked list. The tests will only consider the behaviour of the public methods `write()`, `read()`, and `occupancy()`.

File `ex04-main.cpp`

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

int main() {
    Buffer *b = new LimitedBuffer(5, -999);

    cout << "Current␣buffer␣occupancy:␣" << b->occupancy() << endl;
    cout << "Reading␣from␣the␣buffer␣returns:␣" << b->read() << endl;

    for (unsigned int i = 0; i < 10; i++) {
        b->write(i * 10);
    }
    cout << "Current␣buffer␣occupancy:␣" << b->occupancy() << endl;

    for (unsigned int i = 0; i < 3; i++) {
      cout << "Reading␣from␣the␣buffer␣returns:␣" << b->read() << endl;
    }
    cout << "Current␣buffer␣occupancy:␣" << b->occupancy() << endl;

    for (unsigned int i = 0; i < 10; i++) {
        b->write((i+1) * 100);
    }
    cout << "Current␣buffer␣occupancy:␣" << b->occupancy() << endl;
    for (unsigned int i = 0; i < 6; i++) {
      cout << "Reading␣from␣the␣buffer␣returns:␣" << b->read() << endl;
    }

    delete b;
    return 0;
}
```

File `ex04-library.h`

```
#ifndef EX04_LIBRARY_H_
#define EX04_LIBRARY_H_

#include <vector>
using namespace std;

class Buffer {
public:
    virtual void write(int v) = 0;
    virtual int read() = 0;
    virtual unsigned int occupancy() = 0;
    virtual ~Buffer();
};

// Task 4(a). Declare the class LimitedBuffer, by extending Buffer
// Write your code here

#endif /* EX04_LIBRARY_H_ */
```

File `ex04-library.cpp`

```
#include "ex04-library.h"

// Task 4(a). Write a placeholder implementation of LimitedBuffer's
// constructor and methods

// Task 4(b). Write a working implementation of write() and occupancy()

// Task 4(c). Write a working implementation of read()

// Do not modify
Buffer::~Buffer() {
    // Empty destructor
}
```