

Technical University of Denmark

Written examination date: 17 May 2021



**Course title:** Programming in C++

Page 1 of 15 pages

**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 Digital Eksamen**. You can do it **only once**, so submit only when you have completed your work.
2. You must submit your solution as **one ZIP archive** containing the following files, with these exact names:
  - `exZZ-library.cpp`, where ZZ ranges from 01 to 04 (i.e., one per exercise);
  - `ex04-library.h` (additionally required for exercise 4).
3. You can test your solutions by uploading them on CodeJudge, under “Reexam May 2021” 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. VECTOR FIELDS (2.5 POINTS)

Alice needs to perform computations on *vector fields*, i.e., matrices having bidimensional geometric vectors as elements. 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`. Such files are available with this exam paper (in a separate ZIP archive), and they are also reported in the next pages.

**Structure of the code.** A geometric vector is represented as a `struct Vector` with two fields, named `x` and `y`: they are, respectively, the  $x$  and  $y$  component of the vector. Alice's code already includes the function:

```
void deleteField(Vector **A, unsigned int nRows)
```

which deallocates a vector field allocated with `createField()` (see task (a) below).

**Tasks.** Help Alice by completing the following tasks. You need to edit and submit the file `ex01-library.cpp`.

(a) Implement the function:

```
Complex **createField(unsigned int m, unsigned int n, Vector v)
```

The function must return an array of  $m \times n$  Vectors, i.e., `Vector **`. It must allocate the required memory, and initialise each array element as argument `v`.

(b) Implement the function:

```
void displayField(Vector **A, unsigned int m, unsigned int n)
```

The function must print on screen the contents of the vector field `A` of size  $m \times n$ :

- each vector must be printed as  $(x,y)$  *without* spaces between the  $x,y$  field values;
- 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  $2 \times 4$  vector field should look like:

```
(1,2) (2,5) (4,4) (1,2)
(1,2) (0,2) (0,2) (2,6)
```

*This exercise continues on the next page...*

(c) Implement the function:

```
void addFields(Vector **A, Vector **B, Vector **C,
               unsigned int m, unsigned int n)
```

Where:

- argument **A** is a vector field of size  $m \times n$ ;
- argument **B** is a vector field of size  $m \times n$ ;
- argument **C** is a vector field of size  $m \times n$ .

The function must add the corresponding elements of **A** by **B**, storing the result in **C**. Therefore, as in standard matrix addition, the element at row  $i$  and column  $j$  of **C** is computed as:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

where “+” is the standard vector addition: the addition of two **Vectors** **u** and **v** is a **Vector** whose fields have values  $u.x + v.x$  and  $u.y + v.y$ .

(d) Implement the function:

```
void scaleField(Vector **A, double c, unsigned int m, unsigned int n)
```

Where:

- argument **A** is a vector field of size  $m \times n$ ;
- argument **c** is a scalar value.

The function must multiply each element of **A** by **c**, storing the result in **A** itself. More precisely, the element at row  $i$  and column  $j$  of **A** must be updated as follows:

$$A_{i,j} = A_{i,j} \times c$$

where “ $\times$ ” is the standard vector scalar multiplication: to multiply a **Vector** **v** by a scalar **c**, we multiply both  $v.x$  and  $v.y$  by **c**.

## 02393 Programming in C++

### File ex01-main.cpp

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

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

    Vector **A = createField(3, 3, c);
    A[1][1] = {2, 2};
    cout << "Vector_field_A:" << endl;
    displayField(A, 3, 3);
    cout << endl;

    Vector **B = createField(3, 3, d);
    B[0][0] = B[2][2] = {9, 8};
    cout << "Vector_field_B:" << endl;
    displayField(B, 3, 3);
    cout << endl;

    Vector **R = createField(3, 3, {0,0});
    cout << "Result_of_A+B:" << endl;
    addFields(A, B, R, 3, 3);
    displayField(R, 3, 3);
    cout << endl;

    cout << "Result_of_scaling_A_by_2:" << endl;
    scaleField(A, 2, 3, 3);
    displayField(A, 3, 3);

    deleteField(A, 3); deleteField(B, 3);
    deleteField(R, 3);
    return 0;
}
```

### File ex01-library.cpp

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

using namespace std;

// Task 1(a). Implement this function
Vector **createField(unsigned int m, unsigned int n, Vector v) {
    // Write your code here
}

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

// Task 1(c). Implement this function
void addFields(Vector **A, Vector **B, Vector **C,
               unsigned int m, unsigned int n) {
    // Write your code here
}

// Task 1(d). Implement this function
void scaleField(Vector **A, double c,
                unsigned int m, unsigned int n) {
    // Write your code here
}

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

### File ex01-library.h

```
#ifndef EX01_LIBRARY_H_
#define EX01_LIBRARY_H_

struct Vector {
    double x;
    double y;
};

Vector **createField(unsigned int m, unsigned int n, Vector v);
void displayField(Vector **A, unsigned int m, unsigned int n);
void addFields(Vector **A, Vector **B, Vector **C,
               unsigned int m, unsigned int n);
void scaleField(Vector **A, double c, unsigned int m, unsigned int n);
void deleteField(Vector **A, unsigned int nRows);

#endif /* EX01_LIBRARY_H_ */
```

## 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 with their respective 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`. Such files are available with this exam paper (in a separate ZIP archive), and they are also reported 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. You need to edit and submit the file `ex02-library.cpp`.

(a) Implement the function:

```
Elem* reverse(Elem *list);
```

which reverses the RLE list `list` *in place*, that is, by updating the pointers of its elements. The function returns a pointer to the first element of the reversed list (which corresponds to the last element of the original `list`). For example: if the RLE list  $7_{(\times 25)} 9_{(\times 90)}$  is reversed, the result is  $9_{(\times 90)} 7_{(\times 25)}$ .

*This exercise continues on the next page...*

(b) Implement the function:

```
Elem* concatenate(Elem *list1, Elem *list2)
```

which concatenates the lists `list1` and `list2`, and returns a pointer to the first `Element` of the resulting list. The function must compress the repetitions resulting from the concatenation. For example, if the arguments of the function are:

- `list1 = 7(×2) 6(×1) 9(×2)`
- `list2 = 9(×3) 10(×3)`

then the resulting list must be:

$$7_{(\times 2)} \ 6_{(\times 1)} \ 9_{(\times 5)} \ 10_{(\times 3)}$$

Notice that the last element of `list1` and the first element of `list2` have been compressed into one.

*Important:* the function must *not* use `delete` on any element of `list1` nor `list2`. Besides this, you can choose to implement the function by either creating and returning a new list, or modifying `list1` and `list2`.

(c) Implement the function:

```
int sum(Elem *list)
```

which returns the sum of the elements of `list`, taking into account their repetitions. For example, if `list` is `7(×2) 6(×1) 9(×2)`, then the function must return 38.

### 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);

Elem* reverse(Elem *list);
Elem* concatenate(Elem *list1, Elem *list2);
int sum(Elem *list);

#endif /* EX02_LIBRARY_H_ */
```

### 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};

    Elem e4 = {100, 7, nullptr};
    Elem e5 = {4, 3, &e4};
    Elem e6 = {101, 9, &e5};

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

    cout << "The reversed list is:" << endl;
    Elem *r = reverse(&e2);
    displayRLEList(r);

    cout << endl;

    cout << "After concatenation, the list is:" << endl;
    Elem *l = concatenate(r, &e6);
    displayRLEList(l);
    cout << endl;

    cout << "The sum of its elements is:" << sum(l) << endl;

    return 0;
}
```

### File ex02-library.cpp

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

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

// Task 2(b). Implement this function
Elem* concatenate(Elem *list1, Elem *list2) {
    // Write your code here
}

// Task 2(c). Implement this function
int sum(Elem *list) {
    // 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. GROCERY LIST (2.5 POINTS)

Claire wants to implement a class `GroceryList` to store and update her grocery list. 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`. Such files are available with this exam paper (in a separate ZIP archive), and they are also reported in the next pages.

**Structure of the code.** Claire has represented the information about each entry in the grocery list using a `struct Info`, with two fields:

- **quantity**: how much to buy of a certain item;
- **notes**: any remark about the item.

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> items` — the names of the items to buy;
- `map<string, Info> itemsInfo` — a mapping from `strings` (item names) to instances of `Info` (the information about the item to buy).

Claire has already implemented the default constructor of `GroceryList`, which creates a database with some needed items. She has also implemented the method `display()`, which shows the contents of the grocery list.

**Tasks.** Help Claire by completing the following tasks. You need to edit and submit the file `ex03-library.cpp`.

(a) Implement the following method to add an entry to the grocery list:

```
void GroceryList::add(string name, unsigned int quantity, string notes)
```

The method must work as follows:

- if `name` is *not* in the grocery list, add the given `name` at the end of the `items` vector, and map it to the given `quantity` and `notes` (by updating `itemsInfo`);
- if `name` is already in the grocery list, update its information in `itemsInfo` as follows:
  - increase the original quantity by the given `quantity`. For example: if the original quantity is 100 and the method is invoked with `quantity=200`, the updated quantity must be 300;
  - extend the original notes by adding  `";"` and the given `notes`. For example: if the original notes are `"A"` and the method is invoked with `notes="B"`, the updated notes must be `"A;B"`.

*This exercise continues on the next page...*



(b) Implement the method:

```
bool GroceryList::remove(string name, unsigned int quantity)
```

This method tries to remove the given `quantity` from the grocery list item with the given `name`; it returns `true` if the operation succeeds, and `false` otherwise. The method must work as follows:

- (a) if the grocery list does *not* contain an item with the given `name`, then the method returns `false` without changing the grocery list;
- (b) if the grocery list *does* contain an item with the given `name`, then:
  - if the item's quantity is lower than the given `quantity`, then the method must return `false` without changing the grocery list.
  - otherwise, the method must reduce the item's quantity by subtracting the given `quantity`; then, if the updated item quantity becomes 0, then the method must remove the item from the shopping list. (*See hints below.*) In either case, the method must return `true`.

(c) Implement the method:

```
bool GroceryList::copyEntry(string name, string newName)
```

This method creates a new grocery list entry named `newName`, by copying the information of the item called `name`; it returns `true` if the operation succeeds, and `false` otherwise. The method must work as follows:

- (a) if the grocery list does *not* contain an item with the given `name`, *or* it already contains an item called `newName`, then the method returns `false` without changing the grocery list;
- (b) otherwise, the method must add `newName` at the end of the `items` vector, and update `itemsInfo` to map `newName` to the same information of `name`.

### Hints on using maps and vectors

- 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]`.
- To remove an element from a map or a vector, you can use their `erase(...)` methods.

## 02393 Programming in C++

### File ex03-main.cpp

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

int main() {
    GroceryList gl = GroceryList();

    cout << "Initial grocery list:" << endl;
    gl.display();

    cout << endl << "After adding cheddar:" << endl;
    gl.add("Cheddar", 500, "Not too mature");
    gl.display();

    cout << endl << "After removing some spinach:" << endl;
    if (gl.remove("Spinach", 200)) {
        gl.display();
    } else {
        cout << "FAILED! (this should not happen)" << endl;
    }

    cout << endl << "After copying salmon into haddock:" << endl;
    if (gl.copyEntry("Salmon", "Haddock")) {
        gl.display();
    } else {
        cout << "FAILED! (this should not happen)" << endl;
    }

    return 0;
}
```

### File ex03-library.h

```
#ifndef EX03_LIBRARY_H_
#define EX03_LIBRARY_H_

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

struct Info {
    unsigned int quantity;
    string notes;
};

class GroceryList {
private:
    vector<string> items;
    map<string, Info> itemsInfo;
public:
    GroceryList();
    void add(string name, unsigned int quantity, string notes);
    bool remove(string name, unsigned int quantity);
    bool copyEntry(string name, string newName);
    void display();
};

#endif /* EX03_LIBRARY_H_ */
```

File ex03-library.cpp

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

// Do not modify
GroceryList::GroceryList() {
    this->items.push_back("Lasagne");
    this->itemsInfo["Lasagne"] = {1, "With_eggs_if_available"};

    this->items.push_back("Salmon");
    this->itemsInfo["Salmon"] = {500, "Smoked_if_available"};

    this->items.push_back("Spinach");
    this->itemsInfo["Spinach"] = {300, "Fresh"};

    this->items.push_back("Dessert");
    this->itemsInfo["Dessert"] = {8, "Maybe_lagkage?"};
}

// Task 3(a). Implement this method
void GroceryList::add(string name, unsigned int quantity, string notes) {
    // Write your code here
}

// Task 3(b). Implement this method
bool GroceryList::remove(string name, unsigned int quantity) {
    // Write your code here
}

// Task 3(c). Implement this method
bool GroceryList::copyEntry(string name, string newName) {
    // Write your code here
}

// Do not modify
void GroceryList::display() {
    // Write your code here
    for (auto it = this->items.begin(); it != this->items.end(); it++) {
        Info &item = this->itemsInfo[*it];
        cout << "name=" << *it << " ";
        cout << "quantity=" << item.quantity << " ";
        cout << "notes=" << item.notes << " " << endl;
    }
}
```

### EXERCISE 4. FILTERING BUFFER (2.5 POINTS)

Daisy needs to develop a buffer class to store and retrieve `integer` values. She plans an interface consisting of 4 methods:

- `write(v)` — appends value `v` to the buffer;
- `read()` — removes the oldest value from the buffer and returns it;
- `occupancy()` — returns the number buffered values;
- `reset()` — empties the buffer.

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 *filtering* buffer that accumulates *unique* values, by remembering which values it has contained during its lifecycle. For example, assume that a `FilteringBuffer b` has never contained the value 42:

- the first time `b.write(42)` is called, the value 42 is appended to the buffer contents. From now on, `b` remembers that it has contained 42 — even after 42 is removed by `b.read()`. If `b.write(42)` is executed again, the operation has no effect;
- if `b.reset()` is called, then the buffer `b` is emptied, and it also “forgets” which values it has contained in the past. Therefore, the first call `b.write(42)` after the reset will append 42 to the buffer contents.

Daisy’s first test program is in the file `ex04-main.cpp` and the (incomplete) code of the class is in files `ex04-library.h` and `ex04-library.cpp`. Such files are available with this exam paper (in a separate ZIP archive), and they are also reported in the next pages.

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

**Tasks.** Help Daisy by completing the following tasks. You need to edit and submit **two files**: `ex04-library.h` and `ex04-library.cpp`.

- (a) Declare in `ex04-library.h` and sketch in `ex04-library.cpp` a class `FilteringBuffer` 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 `FilteringBuffer` that takes one parameter: a value of type `int` representing a default (it is used in point (c) below);
  2. in `FilteringBuffer`, 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 FilteringBuffer::write(int v)
unsigned int FilteringBuffer::occupancy()
```

The method `occupancy()` returns the number of values currently stored in the buffer. The intended behaviour of `write(v)` is to check the value `v`, and:

- if the buffer has already contained `v` in the past, then the method has no effect;
- otherwise, the method appends `v` to the buffer contents, and remembers that it has contained `v` (hence, invoking `write(v)` again will have no effect). Correspondingly, the buffer occupancy increases by 1.

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

```
int FilteringBuffer::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. Crucially, `read()` must *not* cause the buffer to “forget” which values it has contained in the past: for example, if `b.read()` returns 42, then invoking `b.write(42)` afterwards must have no effect — because the buffer `b` must remember that it has contained the value 42 (although it might not *currently* contain 42).

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

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

```
void FilteringBuffer::reset()
```

When `reset()` is invoked, the buffer becomes empty (hence, its occupancy becomes 0), and it also forgets which values it has contained in the past.

For example: if buffer `b` contains (or has contained) the value 42, then invoking `b.write(42)` has no effect; however, invoking `b.reset()` and then `b.write(42)` causes 42 to be appended to the (empty) buffer.

**NOTE:** you are free to define the `private` members of `FilteringBuffer` however you see fit. For instance, you might choose to store the values in a `vector<int>`, or in a linked list. Similarly, you are free to choose how to remember which values have been already contained in the buffer. The tests will only consider the behaviour of the public methods `write()`, `read()`, `occupancy()`, and `reset()`.

## 02393 Programming in C++

### File ex04-main.cpp

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

int main() {
    Buffer *b = new FilteringBuffer(-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 < 10; i++) {
        b->write(20);
    }
    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;

    b->reset();
    cout << "Current_buffer_occupancy:" << b->occupancy() << endl;
    cout << "Reading_from_the_buffer_returns:" << b->read() << endl;

    delete b;
    return 0;
}
```

## 02393 Programming in C++

### File ex04-library.h

```
#ifndef EX04_LIBRARY_H_
#define EX04_LIBRARY_H_

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

// Task 4(a). Declare the class FilteringBuffer, 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 FilteringBuffer's
// constructor and methods

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

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

// Task 4(d). Write a working implementation of reset()

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