Technical University of Denmark

Written examination date: 31 May 2022

Course title: Programming in C++

Course number: 02393

Aids allowed: All aids allowed

Exam duration: 4 hours

Weighting: pass/fail

Exercises: 4 exercises with 3 or 4 tasks each, for a total of 14 tasks

Submission details:

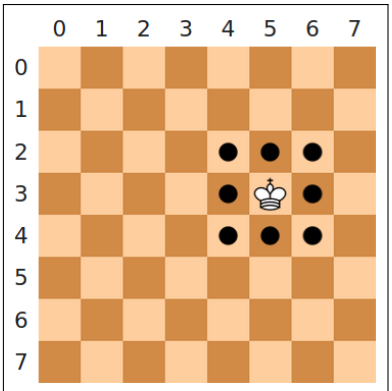
1. You must **submit your solution on DTU Digital Eksamen**. You can do it **only once**, so submit only when you have completed your work.
2. You must submit your solution as **one ZIP archive** containing the following files, with **these exact names**:
   - `ex`*`ZZ`*`-library.cpp`, where *ZZ* ranges 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 2022`" at:

   `https://dtu.codejudge.net/02393-f22/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; start by solving the tasks that look easier, even if they belong to different exercises.

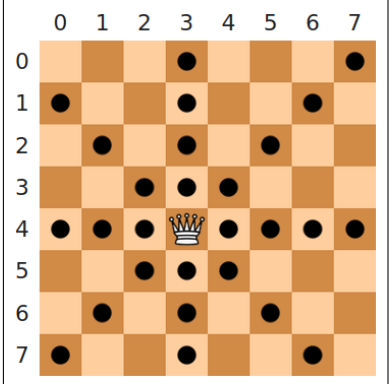
Exercise 1. Mini Chess

Alice wants to implement a variant of the game of chess:

- there are two opposing teams: black and white;
- the chessboard can have any size of $m \times n$ squares (with $m \geq 2$ and $n \geq 2$);
- each team has two kinds of pieces on the chessboard:
  - kings, who move horizontally or diagonally by one square (see figure on the right);
  - queens, who move horizontally or diagonally by any number of squares (see figure on the right);
- pieces can traverse already-occupied squares while moving (unlike standard chess rules);
- a piece can only end its movement on a square that is empty, or occupied by an opponent's piece. In the second case, the opponent's piece is *captured* (i.e. removed).



Valid moves for a king.



Valid moves for a queen.

Source: https://en.wikipedia.org/wiki/Chess

Alice has written some code: her first test program is the file `ex01-main.cpp`, and the (incomplete) code with some functions she needs is in `ex01-library.h` and `ex01-library.cpp`. Such files are available with this exam paper (in a separate ZIP archive), and in the next pages.

Structure of the code. A square on the board is represented as a `struct Square` with two fields named `piece` and `team`, which are two `enum`s representing, respectively:

- which piece (if any) is occupying the square: `king`, `queen`, or `none`;
- which team (if any) owns the piece: `black`, `white`, or `nobody` (if the piece is `none`).

Alice's code already includes the function:

```
void deleteChessboard(Square **c, unsigned int m)
```

which deallocates a chessboard `c` with `m` rows created with `createChessboard()` (task **(a)**).

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

(a) Implement the function:

```
Square** createChessboard(unsigned int m, unsigned int n)
```

The function must return an array of `m` × `n` `Square`s, i.e., `Square**`. It must allocate the required memory, and initialise each square to be empty (i.e. having `none` as piece and `nobody` as team).

This exercise continues on the next page...

(b) Implement the function:

```
void displayChessboard(Square **c, unsigned int m, unsigned int n)
```

The function must print on screen the contents of the chessboard `c` of size `m` × `n`:

- each empty square must be displayed as `_` (underscore);
- if a square is occupied by a piece, it must be displayed as either:
  - `K` (if occupied by a black king) or `k` (if occupied by a white king);
  - `Q` (if occupied by a black queen) or `q` (if occupied by a white queen);
- adjacent squares on a same row must be separated by one space.

Example. A 3×4 chessboard might look as follows: (position (0,0) on the top-left)

```
Q _ _ K
_ _ q _
k _ _ _
```

(c) Implement the function:

```
bool move(Square **c, unsigned int m, unsigned int n,
          int r1, int c1, int r2, int c2)
```

- Argument `c` is a chessboard of size `m` × `n`;
- Arguments `r1` and `c1` are a row and column position on the chessboard;
- Arguments `r2` and `c2` are a row and column position on the chessboard.

The function attempts to move a piece on the chessboard `c` from position (`r1`,`c1`) into position (`r2`,`c2`). The function must check whether the move is valid, i.e.:

- there is a piece at position (`r1`,`c1`);
- positions (`r1`,`c1`) and (`r2`,`c2`) are different and not occupied by the same team;
- the move respects the game rules (see beginning of the exercise).

If the move is valid, the function must update the chessboard `c` and return `true`; otherwise, it must return `false` without altering the chessboard.

You can assume that positions (`r1`,`c1`) and (`r2`,`c2`) are within the chessboard bounds.

Example. Assume that `c` is the chessboard shown in Task **(b)** above:

- `move(c, 3, 4, 0, 0, 0, 3)` must return `false` (same team on both positions);
- `move(c, 3, 4, 0, 0, 0, 2)` must return `true` (the queen move is valid);
- `move(c, 3, 4, 2, 0, 2, 2)` must return `false` (the king move is invalid).
- `move(c, 3, 4, 1, 2, 0, 3)` must return `true` (white queen captures black king);

Hints. Positions (`r1`,`c1`) and (`r2`,`c2`) are on the same diagonal if they differ by equal (absolute) numbers of rows and columns. For instance, positions (4,5) and (6,3) are on the same diagonal, because $|4-6| = |5-3| = 2$. This check (and others) are also needed in Task **(d)** below: with a bit of planning, you can avoid code duplication...

This exercise continues on the next page...

(d) Implement the function:

```
bool threatened(Square **c, unsigned int m, unsigned int n, int row, int col)
```

Where:

- argument `c` is a chessboard of size `m` × `n`;
- arguments `row` and `col` are a row and column position on the chessboard.

The function must return `true` if at position (`row`, `col`) of chessboard `c` there is a piece that can be captured by another piece on the same chessboard (see game rules at the beginning of the exercise). Otherwise, the function must return `false`. In both cases, the function must *not* change the chessboard.

You can assume that the position (`row`, `col`) is within the chessboard bounds.

Example. Assume that `c` is the chessboard shown in Task **(b)** above:

- `threatened(c, 3, 4, 0, 0)` must return `false` (no piece can capture at (0,0));
- `threatened(c, 3, 4, 2, 0)` must return `true` (the white piece at (2,0) can be captured by the black queen at (0,0));
- `threatened(c, 3, 4, 1, 1)` must return `false` (there is no piece at (1,1)).

Hint: this function must perform several checks that are also needed in Task **(c)** above (see also its hints). With a bit of planning, you can avoid code duplication...

File `ex01-main.cpp`

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

int main() {
    Square **c = createChessboard(3, 4);
    c[0][0] = {queen, black};
    c[0][3] = {king, black};
    c[2][0] = {king, white};
    c[1][2] = {queen, white};
    cout << "Chessboard:" << endl;
    displayChessboard(c, 3, 4);

    cout << "Is␣the␣piece␣in␣(0,0)␣threatened?␣";
    if (threatened(c, 3, 4, 0, 0)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << "Can␣we␣move␣from␣(0,0)␣to␣(0,3)?␣";
    if (move(c, 3, 4, 0, 0, 0, 3)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << "Can␣we␣move␣from␣(0,0)␣to␣(0,2)?␣";
    if (move(c, 3, 4, 0, 0, 0, 2)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << "Can␣we␣move␣from␣(2,0)␣to␣(2,2)?␣";
    if (move(c, 3, 4, 2, 0, 2, 2)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << "Can␣we␣move␣from␣(1,2)␣to␣(0,3)?␣";
    if (move(c, 3, 4, 1, 2, 0, 3)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << endl << "The␣chessboard␣is␣now:" << endl;
    displayChessboard(c, 3, 4);

    cout << "Is␣the␣piece␣in␣(2,0)␣threatened?␣";
    if (threatened(c, 3, 4, 2, 0)) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    deleteChessboard(c, 3);
    return 0;
}
```

File `ex01-library.h`

```
#ifndef EX01_LIBRARY_H_
#define EX01_LIBRARY_H_

enum Piece { king, queen, none };
enum Team { black, white, nobody };

struct Square {
    Piece piece;
    Team team;
};

Square **createChessboard(unsigned int m, unsigned int n);
void displayChessboard(Square **c, unsigned int m, unsigned int n);
bool move(Square **c, unsigned int m, unsigned int n,
          int r1, int c1, int r2, int c2);
bool threatened(Square **c, unsigned int m, unsigned int n, int row, int col);
void deleteChessboard(Square **c, unsigned int m);

#endif /* EX01_LIBRARY_H_ */
```

File `ex01-library.cpp`

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

using namespace std;

// Task 1(a). Implement this function
Square **createChessboard(unsigned int m, unsigned int n)
    // Replace the following with your code
    return nullptr;
}

// Task 1(b). Implement this function
void displayChessboard(Square **c,
                       unsigned int m, unsigned int n) {
    // Write your code here
}

// Task 1(c). Implement this function
bool move(Square **c, unsigned int m, unsigned int n,
          int r1, int c1, int r2, int c2) {
    // Replace the following with your code
    return false;
}

// Task 1(d). Implement this function
bool threatened(Square **c, unsigned int m, unsigned int n
                int row, int col) {
    // Replace the following with your code
    return false;
}

// Do not modify
void deleteChessboard(Square **c, unsigned int m) {
    for (unsigned int i = 0; i < m; i++) {
        delete[] c[i];
    }
    delete[] c;
}
```

Exercise 2. Airline Passengers Queue

Bob is writing a program to manage airline passengers boarding a flight. Passengers queue in order of arrival, but they may board the flight in a different order, depending on where they sit. Bob decides to represent the queue as a linked list.

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. A queue element is represented as a `struct Passenger` with 4 fields: `name`, `ticket`, `row`, `seat`, and `next`. Such fields represent, respectively, the passenger name, the ticket number, the row number and the seat (a letter between `'A'` and `'F'`) where the passenger sits, and a pointer to the next passenger in the queue (or `nullptr` when there are no more passengers). An empty queue is represented as a `Passenger*` pointer equal to `nullptr`. Bob's code already includes a function to print the passengers queue on screen:

```
void displayQueue(Passenger *q)
```

Tasks. Help Bob by completing the following tasks. You need to edit and submit the file `ex02-library.cpp`. **NOTE:** some tasks may be easier to solve using recursion, but you can use iteration if you prefer.

(a) Implement the function:

```
void shift(Passenger *q, unsigned int n)
```

which modifies the seat of each passenger in the queue `q`, by adding `n` rows to it. For example: if `q` contains a passenger sitting in row `3`, a call to `shift(q, 2)` will update that passenger's seat to row `3 + 2 = 5`.

Important: the function must modify the elements of `q` in-place.

(b) Implement the function:

```
Passenger* find(Passenger *q, unsigned int rowMin, unsigned int rowMax)
```

which returns a new queue containing all `Passenger`s in the queue `q` that sit between rows `rowMin` and `rowMax` (included), preserving the order in which they appear in `q`. **Important:** the function must return a new queue, where each `Passenger` is a dynamically-allocated copy of its original from `q`; the function must *not* modify `q`.

This exercise continues on the next page...

(c) Implement the function:

```
bool occupied(Passenger *q, unsigned int row, char seat)
```

which checks all passengers in the queue `q` and returns `true` if there is someone sitting in the given `row` and `seat`. If no passenger in the queue is sitting in that position, the function returns `false`.

File ex02-library.h

```
#ifndef EX02_LIBRARY_H_
#define EX02_LIBRARY_H_

#include <string>

struct Passenger {
    std::string name;
    unsigned int ticket;
    unsigned int row;
    char seat;
    Passenger *next;
};

void displayQueue(Passenger *q);

void shift(Passenger *q, unsigned int n);
Passenger* find(Passenger *q, unsigned int rowMin,
                unsigned int rowMax);
bool occupied(Passenger *q, unsigned int row, char seat);

#endif /* EX02_LIBRARY_H_ */
```

File ex02-main.cpp

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

int main() {
    Passenger p0 = {"Alfred␣A.", 123, 5, 'A', nullptr};
    Passenger p1 = {"Barbara␣B.", 321, 1, 'B', &p0};
    Passenger p2 = {"Charlie␣C.", 456, 10, 'D', &p1};
    Passenger p3 = {"Daria␣D.", 654, 22, 'C', &p2};
    Passenger p4 = {"Emil␣E.", 789, 10, 'E', &p3};
    Passenger p5 = {"Fiona␣F.", 987, 21, 'F', &p4};

    Passenger *q = &p5;

    cout << "The␣passengers␣queue␣is:␣" << endl;
    displayQueue(q);
    cout << endl;

    shift(q, 2);
    cout << "After␣shifting␣the␣passengers␣by␣2␣rows␣we␣have:␣" << endl;
    displayQueue(q);
    cout << endl;

    cout << "The␣passengers␣sitting␣between␣rows␣5␣and␣12␣are:␣" << endl;
    Passenger *q2 = find(q, 5, 12);
    if (q2 == nullptr) { cout << "nobody!" << endl; }
    else { displayQueue(q2); }
    cout << endl;

    cout << "Is␣seat␣10D␣occupied?␣";
    if (occupied(q, 10, 'D')) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    cout << "Is␣seat␣12E␣occupied?␣";
    if (occupied(q, 12, 'E')) { cout << "Yes!" << endl; }
    else { cout << "No!" << endl; }

    return 0;
}
```

File ex02-library.cpp

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

// Task 2(a). Implement this function
void shift(Passenger *q, unsigned int n) {
    // Write your code here
}

// Task 2(b). Implement this function
Passenger* find(Passenger *q, unsigned int rowMin,
                unsigned int rowMax) {
    // Replace the following with your code
    return nullptr;
}

// Task 2(c). Implement this function
bool occupied(Passenger *q, unsigned int row, char seat) {
    // Replace the following with your code
    return false;
}

// Do not modify
void displayQueue(Passenger *q) {
    if (q == nullptr) {
        return;
    }
    cout << q->name << "␣-␣ticket:␣" << q->ticket;
    cout << ",␣seat␣" << q->row << q->seat << endl;
    displayQueue(q->next);
}
```

Exercise 3. Hotel Management

Claire owns a fancy hotel where every room has the name of a flower. She is writing a class `Hotel` to manage the information about the rooms and guests. 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 a guest using a `struct Guest`, with two fields:

- `name`: the full name of the guest;
- `id`: the document id provided by the guest.

Claire knows that the `map` and `vector` containers of the C++ standard library provide many functionalities she needs. *(See hints on page 10.)* Therefore, she has decided to use the following internal (`private`) representation for the library:

- `vector<string> roomNames` — the names of the fancy hotel rooms;
- `map<string,Guest> roomOccupancy` — a mapping from `string`s (room names) to instances of `Guest` (info about the guest occupying the room, if any). When a room name does not appear in this mapping, it means that the room is empty and available.

Claire has already implemented the default constructor of `Hotel`, which creates an internal database with all rooms. She has also implemented the method `display()`, which shows which guest occupies which room.

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 rename a room:

```
void Hotel::renameRoom(string oldName, string newName)
```

The method must work as follows:

(a) if `oldName` is *not* in `roomNames`, do nothing;

(b) if `newName` is already in `roomNames`, do nothing;

(c) otherwise, update `roomNames` by replacing the element `oldName` with `newName`; also, update `roomOccupancy` so that any guest occupying `oldName` is moved to `newName`.

This exercise continues on the next page...

(b) Implement the following method to remove a guest:

```
void Hotel::removeGuest(string roomName, string guestName, string guestId)
```

The method must check whether `roomName` is occupied by a guest with the given `guestName` and `guestId`; if so, remove the occupancy (so the room is available again); otherwise, do nothing.

(c) Implement the method:

```
void Hotel::findRoomByGuestId(vector<string> guestIds)
```

This method displays the name(s) of the room(s) with an occupant whose id is contained in the given collection `guestIds`.

The room names must be displayed one-per-line, by following their order in `roomNames`.

For example, suppose that we have a vector `v` containing the strings `"123"` and `"abc"`. Then, `hotel.findRoomByGuestId(v)` will display the names of all rooms whose guest has id equal to either `"123"` or `"abc"`.

Hints on using `maps` and `vectors` (See also: `https://www.cplusplus.com/reference/map/map/` and `https://www.cplusplus.com/reference/vector/vector/`)

- To remove an element from a map or a vector, you can use their `erase(...)` methods:
  - `https://www.cplusplus.com/reference/map/map/erase/`
  - `https://www.cplusplus.com/reference/vector/vector/erase/`
- 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 loop on all (key, value) pairs in a map `m`, you can use: `for (auto p: m) { ... }`. The loop variable `p` is a `pair` with the map key as `p.first`, and the corresponding value as `p.second` (see `https://www.cplusplus.com/reference/utility/pair/`)

File `ex03-main.cpp`

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

int main() {
    Hotel hotel = Hotel();

    cout << "Initial␣hotel␣occupancy:" << endl;
    hotel.display();

    hotel.renameRoom("Lotus", "Waterlily");
    cout << endl << "After␣renaming␣room␣'Lotus'␣to␣'Waterlily':" << endl;
    hotel.display();

    hotel.removeGuest("Orchid", "Alan␣Smithee", "abc123");
    cout << endl << "After␣removing␣a␣guest:" << endl;
    hotel.display();

    cout << endl << "Room(s)␣with␣guests␣with␣id␣'123xyz'␣or␣'456abc':" << endl;
    vector<string> v;
    v.push_back("123xyz");
    v.push_back("456abc");
    hotel.findRoomByGuestId(v);

    return 0;
}
```

File `ex03-library.h`

```
#ifndef EX03_LIBRARY_H_
#define EX03_LIBRARY_H_

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

struct Guest {
    string name;
    string id;
};

class Hotel {
private:
    vector<string> roomNames;
    map<string,Guest> roomOccupancy;
public:
    Hotel();
    void renameRoom(string oldName, string newName);
    void removeGuest(string roomName, string guestName, string guestId);
    void findRoomByGuestId(vector<string> guestIds);
    void display();
};

#endif /* EX03_LIBRARY_H_ */
```

File `ex03-library.cpp`

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

// Do not modify
Hotel::Hotel() {
    this->roomNames.push_back("Daisy");
    this->roomOccupancy["Daisy"] = {"Alan␣Smithee", "xyz890"};

    this->roomNames.push_back("Geranium");

    this->roomNames.push_back("Lotus");
    this->roomOccupancy["Lotus"] = {"Kathryn␣Bigelow", "456abc"};

    this->roomNames.push_back("Orchid");
    this->roomOccupancy["Orchid"] = {"Alan␣Smithee", "abc123"};

    this->roomNames.push_back("Tulip");
    this->roomOccupancy["Tulip"] = {"Denis␣Villeneuve", "123xyz"};
}

// Task 3(a). Implement this method
void Hotel::renameRoom(string oldName, string newName) {
    // Write your code here
}

// Task 3(b). Implement this method
void Hotel::removeGuest(string roomName, string guestName, string guestId) {
    // Write your code here
}

// Task 3(c). Implement this method
void Hotel::findRoomByGuestId(vector<string> guestIds) {
    // Write your code here
}

// Do not modify
void Hotel::display() {
    for (auto it = this->roomNames.begin(); it != this->roomNames.end(); it++) {
        cout << "Room␣'" << *it << "'␣is␣";
        if (this->roomOccupancy.find(*it) == this->roomOccupancy.end()) {
            cout << "empty" << endl;
        } else {
            cout << "occupied␣by␣" << this->roomOccupancy[*it].name;
            cout << "␣(id:␣" << this->roomOccupancy[*it].id << ")" << endl;
        }
    }
}
```

Exercise 4. Sensor Data Buffer

Daisy is writing a program that reads integer values from a sensor; the sensor may sometimes yield values that are outside a certain allowed range, so her program must account for this possibility. Therefore, she plans a `SensorBuffer` class with the following interface:

- `write(v)`: appends value `v` (obtained from the sensor) into the buffer;
- `read()`: returns the oldest value written in the buffer, and removes it from the buffer;
- `faults()`: returns the number of time the method `write(v)` has been invoked with a value `v` outside a range specified with the `SensorBuffer` constructor (see below).
- `clear()`: empties the buffer, and resets the number of faults to 0.

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()` and `read()`. She wants to implement `SensorBuffer` as a subclass of `Buffer`, with the additional methods `faults()` and `clear()`.

Example. Once completed, the class `SensorBuffer` must work as follows:

- suppose that we create `buf = SensorBuffer(-1, 5, 10)` (i.e. `buf` has a default value `-1`, and expects that written values are between `5` and `10` included);
- suppose that `buf.write(7)` is invoked, followed by `buf.write(9)`. Then, a call to `buf.read()` must return `7`, and a further call to `buf.read()` must return `9` (therefore, `read()` removes the returned value from the buffer). A further call to `buf.read()` must return the default value `-1` (since `buf` is now empty);
- then, suppose that `buf.write(3)` is invoked. Since the given value `3` is below the minimum value `5`, it must be corrected as `5`, so a subsequent call to `buf.read()` must return `5`. Moreover, `buf.faults()` must return `1`;
- finally, suppose that `buf.clear()` is invoked. Then, `buf.read()` must return the default value `-1`, and `buf.faults()` must return `0`.

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

NOTE: you are free to define the `private` members of `SensorBuffer` 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()`, `faults()`, and `clear()`.

This exercise continues on the next page...

(a) Declare in `ex04-library.h` and sketch in `ex04-library.cpp` a class `SensorBuffer` 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 `SensorBuffer` constructor that takes 3 parameters: 3 `int` values representing, respectively, the default value (returned by `read()` when the buffer is empty), and the minimum and maximum values (checked when using `write()`);
2. in `SensorBuffer`, override the *pure virtual methods* of `Buffer` (i.e., those with "`=0`"), and add the following `public` methods to the class interface:
   - `unsigned int faults()`
   - `void clear()`
3. finally, write a placeholder implementation of all the `SensorBuffer` methods above (e.g. they may do nothing and/or just return `0` when invoked).

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

```
void SensorBuffer::write(int v)
unsigned int SensorBuffer::faults()
```

The intended behaviour of `write(v)` is to check whether `v` is between the minimum and maximum values specified in the `SensorBuffer` constructor:

i) if `v` is within such values, it is stored (so it can be retrieved by `read()`);

ii) otherwise, if `v` is below the minimum value, the minimum value is stored;

iii) otherwise (i.e. if `v` is above the maximum value), the maximum value is stored.

In both cases *(ii)* and *(iii)*, the method `write()` should increment an internal counter used by `faults()`. The method `faults()` returns how many times the method `write(v)` has been invoked with an out-of-range value `v`.

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

```
int SensorBuffer::read()
```

When invoked, `read()` returns the oldest value stored in the buffer using `write()`, removing the value from the buffer.

Special case: if the buffer is empty, then `read()` must return the default value specified in the `SensorBuffer` constructor.

This exercise continues on the next page...

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

```
void SensorBuffer::clear()
```

When invoked, `clear()` empties the buffer and resets the faults counter. Consequently, invoking `clear()` and then `read()` returns the default buffer value (specified in the `SensorBuffer` constructor); also, invoking `clear()` and then `faults()` returns 0.

File ex04-main.cpp

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

int main() {
    SensorBuffer *sb = new SensorBuffer(-1, 5, 10);
    Buffer *b = sb; // Just an alias for 'sb' above, but using the superclass

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

    b->write(7); b->write(9);
    cout << "Wrote␣7␣and␣9.␣␣Current␣faults:␣" << sb->faults() << endl;
    cout << "Reading␣from␣the␣buffer␣now␣returns:␣" << b->read() << endl;
    cout << "Reading␣from␣the␣buffer␣now␣returns:␣" << b->read() << endl;

    b->write(3); b->write(10);
    cout << "Wrote␣3␣and␣10.␣␣Current␣faults:␣" << sb->faults() << endl;
    cout << "Reading␣from␣the␣buffer␣now␣returns:␣" << b->read() << endl;
    cout << "Reading␣from␣the␣buffer␣now␣returns:␣" << b->read() << endl;

    sb->clear();
    cout << "Buffer␣cleared.␣␣Current␣faults:␣" << sb->faults() << endl;
    cout << "Reading␣from␣the␣buffer␣now␣returns:␣" << b->read() << endl;

    delete sb;
    return 0;
}
```

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 ~Buffer();
};

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

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

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

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

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