Technical University of Denmark

Written examination date: 21 December 2021

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:
   - `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 "`Exam December 2021`" at:

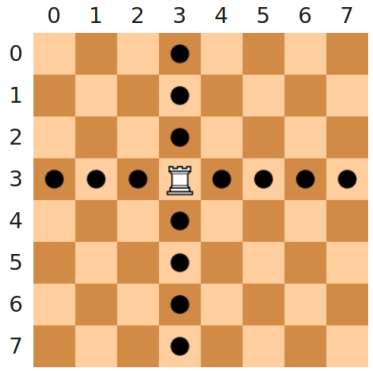
   `https://dtu.codejudge.net/02393-e21/exercises`
4. You can test your solutions on CodeJudge as many times as you like. *Uploads on CodeJudge are not official submissions* and will not affect your grade.
5. Additional tests may be run on your submissions after the exam.
6. Feel free to add comments to 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

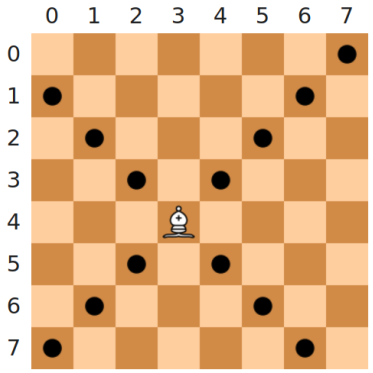
Alice wants to implement a variant of the game of chess:

- there are two opposing teams: black and white;
- the chessboard can have any size of $n \times n$ squares ($n \geq 2$);
- each team has two kinds of pieces on the chessboard:
  - rooks, who move horizontally and vertically by any number of squares (see figure on the right);
  - bishops, who move 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).

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.



Valid moves for a rook.



Valid moves for a bishop.

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

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: `rook`, `bishop`, 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 n)
```

which deallocates a chessboard `c` allocated with `createChessboard()` (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:

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

The function must return an array of `n` × `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 n)
```

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

- each empty square must be displayed as `_` (underscore);
- if a square is occupied by a piece, it must be displayed as either:
  - `R` (if occupied by a black rook) or `r` (if occupied by a white rook);
  - `B` (if occupied by a black bishop) or `b` (if occupied by a white bishop);
- adjacent squares on a same row must be separated by one space.

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

```
b _ r
_ _ _
R B _
```

(c) Implement the function:

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

- Argument `c` is a chessboard of size `n` × `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, 0, 0, 1, 0)` must return `false` (bishops only move diagonally);
- `move(c, 3, 2, 0, 2, 1)` must return `false` (same team on both positions);
- `move(c, 3, 2, 0, 0, 0)` must return `true` (black rook captures white bishop);
- `move(c, 3, 2, 1, 1, 2)` must return `true` (the bishop move is valid).

Hints. Positions (`r1`,`c1`) and (`r2`,`c2`) are on the same diagonal if they differ by equal (absolute) numbers of rows and columns. For example: 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 n, int row, int col)
```

Where:

- argument `c` is a chessboard of size `n` × `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, 0, 0)` must return `true` (can be captured by rook at (2,0));
- `threatened(c, 3, 0, 2)` must return `false` (no piece can capture at (0,2));
- `threatened(c, 3, 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);
    c[0][0] = {bishop, white};
    c[0][2] = {rook, white};
    c[2][0] = {rook, black};
    c[2][1] = {bishop, black};
    cout << "Chessboard:" << endl;
    displayChessboard(c, 3);

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

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

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

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

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

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

    cout << "Is␣the␣piece␣in␣(0,2)␣threatened?␣";
    if (threatened(c, 3, 0, 2)) { 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 { rook, bishop, none };
enum Team { black, white, nobody };

struct Square {
    Piece piece;
    Team team;
};

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

#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 n) {
    // Replace the following with your code
    return nullptr;
}

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

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

// Do not modify
void deleteChessboard(Square **c, unsigned int n) {
    for (unsigned int i = 0; i < n; 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 whether they bought a Priority add-on. 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`, `priority`, and `next`. Such fields represent, respectively, the passenger name, the ticket number, whether the passenger has Priority (`true` or `false`), 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:

```
Passenger* find(Passenger *q, unsigned int ticket)
```

which returns a pointer to the `Passenger` in the queue `q` that has the given `ticket`. You can assume that at most one passenger has that `ticket`; if no passenger has it, the function must return `nullptr`. **Important:** the function must *not* modify `q`.

(b) Implement the function:

```
Passenger* remove(Passenger *q, unsigned int ticket)
```

which returns a passenger queue including all passengers in `q` (in the same order) *except* the passenger with the given `ticket`. You can assume that at most one passenger has that `ticket`. If no passenger owns that ticket, the returned list must have the same passengers of `q`, in the same order.

Important: the function must *not* use `delete` on any element of `q`. Besides this, you can choose to implement the function by either creating and returning a new list of passengers, or modifying `q`.

This exercise continues on the next page...

(c) Implement the function:

```
Passenger* priority(Passenger *q)
```

which returns a *new* passenger queue containing a *copy* of all passengers in the queue `q` who have bought Priority, and in the same order they have in `q`.

Important: the function must *not* modify `q`.

File ex02-library.h

```
#ifndef EX02_LIBRARY_H_
#define EX02_LIBRARY_H_

#include <string>

struct Passenger {
    std::string name;
    unsigned int ticket;
    bool priority;
    Passenger *next;
};

void displayQueue(Passenger *q);

Passenger* find(Passenger *q, unsigned int ticket);
Passenger* remove(Passenger *q, unsigned int ticket);
Passenger* priority(Passenger *q);

#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, false, nullptr};
    Passenger p1 = {"Barbara B.", 321, true, &p0};
    Passenger p2 = {"Charlie C.", 456, true, &p1};
    Passenger p3 = {"Daria D.", 654, false, &p2};
    Passenger p4 = {"Emil E.", 789, false, &p3};
    Passenger p5 = {"Fiona F.", 987, true, &p4};

    Passenger *q = &p5;

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

    cout << "The passenger with ticket 654 is: ";
    Passenger *pp = find(q, 654);
    if (pp == nullptr) {
        cout << "nobody!" << endl;
    } else {
        cout << pp->name << endl;
    }

    Passenger* q2 = remove(q, 654);
    cout << "After removing the passenger with ticket 654, the queue is:" << endl;
    displayQueue(q2);
    cout << endl;

    Passenger *qp = priority(q2);
    cout << "The queue of priority passengers is:" << endl;
    displayQueue(qp);

    return 0;
}
```

File ex02-library.cpp

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

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

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

// Task 2(c). Implement this function
Passenger* priority(Passenger *q) {
    // Replace the following with your code
    return nullptr;
}

// Do not modify
void displayQueue(Passenger *q) {
    if (q == nullptr) {
        return;
    }
    cout << q->name << " - ticket: " << q->ticket;
    if (q->priority) {
        cout << " (priority)";
    }
    cout << 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 a 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 add a room name:

```
void Hotel::addRoom(string name)
```

The method must work as follows:

(a) if `name` is *not* in `roomNames`, add the given `name` at the end of `roomNames`;

(b) if `name` is already in `roomNames`, do nothing.

This exercise continues on the next page...

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

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

The method must work as follows:

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

(b) otherwise:

- if the room `roomName` is already occupied by some guest, do nothing;
- if the room `roomName` is available, then:
  - if there is already a room with a guest having the given `guestId`, do nothing (document ids cannot be duplicated!);
  - otherwise, update the map `roomOccupancy` to assign the given room to the given guest.

(c) Implement the method:

```
void Hotel::findRoomByGuest(string guestName, string guestId)
```

This method displays the name(s) of the room(s) with an occupant who has the given guest name and id.

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

The string `"*"` can be passed as `guestName` or `guestId` to match anything. Therefore:

- `hotel.findRoomByGuest("*", "*")` must display all hotel rooms having a guest;
- `hotel.findRoomByGuest("Ed␣Wood", "*")` must display all hotel rooms occupied by a guest called Ed Wood;
- `hotel.findRoomByGuest("Ed␣Wood", "123abc")` must display all hotel rooms occupied by a guest called Ed Wood with id `"123abc"`.

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

- 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.addRoom("Waterlily");
    cout << endl << "After␣adding␣room␣'Waterlily':" << endl;
    hotel.display();

    hotel.addGuest("Waterlily", "Taika␣Waititi", "pqr567");
    cout << endl << "After␣adding␣a␣guest:" << endl;
    hotel.display();

    cout << endl << "Room(s)␣occupied␣by␣someone␣called␣Alan␣Smithee:" << endl;
    hotel.findRoomByGuest("Alan␣Smithee", "*");

    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 addRoom(string name);
    void addGuest(string roomName, string guestName, string guestId);
    void findRoomByGuest(string guestName, string guestId);
    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::addRoom(string name) {
    // Write your code here
}

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

// Task 3(c). Implement this method
void Hotel::findRoomByGuest(string guestName, string guestId) {
    // 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 `int`eger values from a sensor. Her program may need to read either the most recent value obtained from the sensor, or the average of the most recent values. 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 most recent value written in the buffer;
- `readAvg()`: returns the average of the most recent values written in the buffer. The average is computed using up to n values, where n is a parameter given to the `SensorBuffer` constructor (see below);
- `writeCount()`: returns how many times the method `write()` has been invoked since the buffer was created.

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 `readAvg()` and `writeCount()`.

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

- suppose that we create `buf = SensorBuffer(4)` (i.e. `buf` uses up to 4 values to compute an average);
- suppose that `buf.write(1)` is invoked, followed by `buf.write(2)`. Then, a call to `buf.read()` must return `2`, and a further call to `buf.read()` must still return `2`; (therefore, `read()` does *not* remove the returned value from the buffer)
- then, suppose that `buf.write(4)` is invoked, followed by `write(5)`. Then, a subsequent call to `buf.readAvg()` must return `3` (i.e. $(1+2+4+5) \div 4$);
- finally, suppose that `buf.write(5)` is invoked. Then, `buf.read()` must return `5`, and `buf.readAvg()` must return `4` (i.e. $(2+4+5+5) \div 4$).

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()`, `readAvg()`, and `writeCount()`.

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 one parameter: an `unsigned int` representing the number of values used to compute averages (used in Task **(d)**);
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:
   - `int readAvg()`
   - `unsigned int writeCount()`
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::writeCount()
```

The intended behaviour of `write(v)` is to store value `v` (so it can be used by `read()` and `readAvg()`) and increment an internal counter used by `writeCount()`. The method `writeCount()` returns how many times the method `write()` has been invoked since the buffer was created.

(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 latest value written in the buffer using `write()`.

Special case: if `write()` was never used before, then `read()` must return `0`.

This exercise continues on the next page...

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

```
int SensorBuffer::readAvg()
```

When invoked, `readAvg()` returns the average of latest n values written in the buffer using `write()` — where n is the constructor parameter described in task **(a)**1 above.

Special cases:

- if `write()` was never used before, then `readAvg()` must return `0`;
- if `write()` has only been called m times with $0 < m < n$, then the returned average must be computed using the latest m values.

File ex04-main.cpp

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

int main() {
    SensorBuffer *sb = new SensorBuffer(4);
    Buffer *b = sb; // Just an alias for 'sb' above, but using the superclass

    cout << "Current write count: " << sb->writeCount() << endl;
    cout << "Reading from the buffer returns: " << b->read() << endl;

    b->write(1); b->write(2);
    cout << "Current write count: " << sb->writeCount() << endl;
    cout << "Reading from the buffer now returns: " << b->read() << endl;

    b->write(4); b->write(5);
    cout << "Current write count: " << sb->writeCount() << endl;
    cout << "Reading from the buffer now returns: " << b->read() << endl;
    cout << "The buffer average is now: " << sb->readAvg() << endl;

    b->write(5);
    cout << "Current write count: " << sb->writeCount() << endl;
    cout << "Reading from the buffer now returns: " << b->read() << endl;
    cout << "The buffer average is now: " << sb->readAvg() << 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 writeCount()

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

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

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