Technical University of Denmark

Written examination date: 21 December 2023

Course title: Programming in C++ Page 1 of 12 pages
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 15 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 file per exercise**, with **these exact names**:
   - For exercises 1 to 3, `ex`*`ZZ`*`-library.cpp`, where *`ZZ`* ranges from `01` to `03` (i.e., one per exercise);
   - For exercise 4, `ex04-library.zip`, which must contain `ex04-library.cpp` and `ex04-library.h`.
3. You can test your solutions by uploading them on Autolab, under "`Exam December 2023`" at:

   `https://autolab.compute.dtu.dk/courses/02393-E23/assessments`
4. You can test your solutions on Autolab as many times as you like. *Uploads on Autolab are not official submissions* and will not affect your grade.
5. In case Autolab is malfunctioning, you can test your solution by comparing your output with `ex`*`ZZ`*`-main-output.txt`, which is enclosed in the ZIP archive for each handout.
6. Additional tests may be run on your submissions after the exam.
7. Feel free to add comments to your code.
8. **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. Battleship

Alice is implementing a variant of the old game *Battleship* (also known as Sea Battle): the first player places up to 9 ships in a sea with a map made of tiles. Each ship can occupy up to 4 consecutive tiles, and it can be placed either horizontally or vertically. Each ship must be numbered differently from the other ships. Ships must not overlap. Then, the second player has to sink all the ships by launching missiles targeting a specific tile. A map of size 6 × 16 might look like **Fig. 1** on the right:

- ' ' is an empty tile;
- '`3`' is a tile occupied by ship number 3, which has not been hit by a missile;
- '`X`' is a tile occupied by a ship, which has been hit by a missile.

```
1  22
       3  4X44
5      X
   6   3
   X         77
          X
```

Fig. 1. 6 × 16 sea with all tiles visible.

After the first player has placed all the ships, they become hidden. The sea with hidden ships is drawn as shown in **Fig. 2** on the right:

- tiles that are not hit by a missile are shown as '`?`';
- if a tile was hit by a missile, then:
  - if the tile is occupied by a ship, it is shown as '`X`';
  - otherwise, it is shown as ' '.

```
??????????? ??
???????????X????
?????? X????????
?????? ?????????
?? X ???????????
??   ?????X  ???
```

Fig. 2. The sea above with hidden ships, some of which have been hit.

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

Structure of the code. A tile of the sea is represented as a `struct Tile` with two fields named `ship` and `hit`, which represent, respectively:

- the number associated to the ship occupying that tile; 0 indicates that no ship occupies that tile;
- whether the tile has been hit by a missile.

Alice's code already includes a function (that you must not modify):

- `void deleteSea(Tile **sea, unsigned int m)`

This function deallocates the sea.

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

This exercise continues on the next page. . .

(a) Implement the function:

```
Tile** createSea(unsigned int m, unsigned int n)
```

It must return an array of `m`×`n` `Tile`s, i.e., `Tile**`. It must allocate the required memory, and initialise each tile with `ship` set to 0 and `hit` set to `false`.

(b) Implement the function:

```
void displaySea(Tile **sea, unsigned int m, unsigned int n,
  bool reveal)
```

The function must print on screen the contents of all the tiles of the sea `sea` of size `m`×`n`, according to the explanation at the beginning of this exercise: when `reveal` is set to `true`, the sea must be displayed as shown in **Fig. 1** of page 2; when `reveal` is set to `false`, the sea must be displayed as shown in **Fig. 2** of page 2.

(c) Implement the function:

```
bool placeShip(Tile **sea, int m, int n, int r, int c, int number,
  int size, bool vertical)
```

where `sea` is a sea of size `m`×`n`, `r` and `c` are a row and column position, `number` is the number of the ship, `size` the number of tiles it occupies, and `vertical` indicates whether the ship should be placed vertically or horizontally in the sea.

The function tries to place a ship in the sea, starting from the tile at position (`r`,`c`). The function must check whether:

- No other ship in the sea has the same `number`;
- the size of the ship is between 1 and 4 (included);
- the position (`r`,`c`) is within the boundaries of the sea;
- if the ship is placed horizontally:
  - The position (`r`,`c+size`) is within the boundaries of the sea;
  - Tiles at position (`r`,`c+i`), where `i` is between 0 and `size` (included), are not occupied by another ship.
- if the ship is placed vertically:
  - The position (`r+size`,`c`) is within the boundaries of the sea;
  - Tiles at position (`r+i`,`c`), where `i` is between 0 and `size` (included), are not occupied by another ship.

This exercise continues on the next page...

If any of these conditions is *not* satisfied, the function must return `false` without altering the sea. If all conditions are satisfied, the function must update the sea `sea` and return `true`:

- if the ship is placed horizontally, tiles at position (`r`,`c+i`), where `i` is between 0 and `size` (included), will be associated to the ship `number`;
- if the ship is placed vertically, tiles at position (`r+i`,`c`), where `i` is between 0 and `size` (included), will be associated to the ship `number`.

Example. Assume that `sea` is the sea shown on page 2, **Fig. 1**: it has size 6×16.

- `placeShip(sea, 6, 16, 5, 0, 1, 2, false)` must return `false` (another ship numbered 1 exists);
- `placeShip(sea, 6, 16, 5, 0, 9, 5, false)` must return `false` (ship size exceeds limit);
- `placeShip(sea, 6, 16, 6, 7, 9, 2, false)` must return `false` (position out of boundary);
- `placeShip(sea, 6, 16, 5, 15, 9, 2, false)` must return `false` (position out of boundary);
- `placeShip(sea, 6, 16, 5, 9, 9, 3, false)` must return `false` (tile already occupied);
- `placeShip(sea, 6, 16, 5, 0, 9, 2, false)` must return `true` (after updating the sea);

(d) Implement the function:

```
bool launchMissile(Tile **sea, int m, int n, int r, int c)
```

where `sea` is a sea of size `m` × `n`, and `r` and `c` are a row and column position of the tile targeted by the missile.

The function tries to launch a missile targeting the tile at position (`r`,`c`). The function must check whether:

- The position (`r`,`c`) is within the boundaries of the sea;
- the tile has already been hit.

If any of these conditions is *not* satisfied, the function must return `false` without altering the sea. If all conditions are satisfied, the function must update the sea `sea` by marking the tile as hit, and return `true`.

Exercise 2. Playlist

Bob is writing a program to manage the songs aired by a radio station. In particular, songs queue in a playlist, where they are removed once they are played. Normally, the playlist works in FIFO (first-in-first-out) order: the oldest song being added to the playlist will be the first one to be played. However, one can decide to reorder the playlist by shifting a listed song before or after the others. Bob decides to represent the playlist as a linked list, where the head of the list represents the song that will be played first, and the tail the song that will be played last.

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).

Structure of the code. A playlist element is represented as a `struct Song` with 4 fields: `title`, `artist`, `genre`, `duration`, and `next`. Such fields represent, respectively, the title of the song, the artist, the genre and the duration in seconds, and a pointer to the next song in the playlist (which will be played after the current one) (or `nullptr` when there are no more songs). An empty playlist is represented as a `Song*` pointer equal to `nullptr`. Bob's code already includes a function to print the content of the playlist on screen:

```
void displayPlaylist(Song *s)
```

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:

```
unsigned int totalDuration(Song *s)
```

which computes the total duration of all songs in the playlist `s` and returns it.

This exercise continues on the next page...

(b) Implement the function:

```
Song* find(Song *s, string genre)
```

which returns a new playlist containing all `Song`s in the playlist `s` that belong to the same `genre`, preserving the order in which they appear in `s`. **Important:** the function must return a new playlist, where each `Song` is a dynamically-allocated copy of its original from `s`; the function must *not* modify `s`.

(c) Implement the function:

```
bool shift(Song *&s, unsigned int pos, unsigned int n)
```

which tries to modify the playlist `s` by moving the song located at position `pos` after the `n` subsequent songs. Note that the position of songs start from 1.

The function must check whether:

- A song exists at position `pos`.
- A song exists at position `pos` + `n`.

If any of these conditions is *not* satisfied, the function must return `false` without altering `s`. If all conditions are satisfied, the function must update `s` accordingly and return `true`.

Example. Assume that `s` contains the following items (from the head to the tail):

- `Billie Jean, Micheal Jackson, Pop, 232`
- `Sandstorm, Darude, Dance, 225`
- `Never Gonna Give You Up, Rick Hashley, Pop, 215`
- `Barbie Girl, Aqua, Dance, 196`

Then, `shift(*s, 3, 2)` must return `false` since the element at position 3 does not have 2 successors. Conversely, `shift(*s, 2, 2)` must return `true` and change `s` to reflect the following order:

- `Billie Jean, Micheal Jackson, Pop, 232`
- `Never Gonna Give You Up, Rick Hashley, Pop, 215`
- `Barbie Girl, Aqua, Dance, 196`
- `Sandstorm, Darude, Dance, 225`

Exercise 3. Parking area

Claire works in a municipality, which has recently installed several cameras to monitor a public parking area. She is writing a class `ParkingArea` to manage the information about the parking spaces in that car park, and the vehicles parked in there. 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).

Structure of the code. Claire has represented the information about a vehicle occupying a parking space using a `struct Vehicle`, with 3 fields:

- `plate`: the license plate of the vehicle;
- `owner`: the name of the owner of the vehicle;
- `category`: the category of vehicle: car, motorbike, or truck.

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:

- `enum Category { CAR, MOTORBIKE, TRUCK }` — an enumeration to specify the category of a vehicle.
- `map<string,Category> parkingSpaces` — a mapping from `string`s (parking space IDs) to `Category` (info about the category of vehicle that should occupy the parking space);
- `map<string,Vehicle> parkingOccupancy` — a mapping from `string`s (parking space IDs) to instances of `Vehicle` (info about the vehicle occupying the parking space, if any). When a parking space ID does not appear in this mapping, it means that the parking space is empty.

Claire has already implemented the default constructor of `ParkingArea`, which creates an internal database with all the parking spaces installed. She has also implemented the methods `display()`, which shows whether a parking space is occupied, and what vehicle occupies it, and `categoryToString()`, which converts a `Category` into a `string`.

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

This exercise continues on the next page...

(a) Implement the following method to register a vehicle occupying a parking space:

```
void ParkingArea::park(string parkingSpaceID, string plate,
  string owner, Category category)
```

The method must work as follows:

(a) if `parkingSpaceID` is *not* in `parkingSpaces`, do nothing;

(b) otherwise:

- if the parking space `parkingSpaceID` is already occupied by a vehicle, do nothing;
- if the parking space `parkingSpaceID` is available, then update the map `parkingOccupancy` to register the vehicle occupying that parking space.

(b) Implement the following method to register a vehicle leaving a parking space:

```
void ParkingArea::leave(string plate)
```

The method must work as follows:

(a) if no parking space is occupied by the vehicle having `plate` as license plate, do nothing; otherwise

(b) update the map `parkingOccupancy` to remove information on the vehicle having `plate` as license plate.

(c) Implement the following method to identify misplaced vehicles:

```
void ParkingArea::findMisplacedVehicles()
```

The method displays the license plates of the vehicles that occupy a parking space of a different `category` than the one of that vehicle.

The license plates must be displayed one-per-line, and parking spaces must be explored by following their order in `parkingSpaces`.

For example, if parking space “LYNGBY04” should be occupied by a car, but it is occupied by a motorbike, the license plate of the motorbike must be displayed.

This exercise continues on the next page...

(d) Implement the method:

```
void ParkingArea::findVehicles(vector<string> plates)
```

This method displays the ID(s) of the parking space(s) with a vehicle whose license plate is contained in the given collection `plates`.

The parking space ID must be displayed one-per-line, by following their order in `parkingSpaces`.

For example, suppose that we have a vector `v` containing the strings `"AB123XY"` and `"CD987QW"`. Then, `parkingSpaces.findVehicles(v)` will display the IDs of all parking spaces whose vehicle has a license plate equal to either `"AB123XY"` or `"CD987QW"`.

Hints on using `maps` and `vectors` (See also: `https://en.cppreference.com/w/cpp/container/map` and `https://en.cppreference.com/w/cpp/container/vector`)

- To remove an element from a map or a vector, you can use their `erase(...)` methods:
  - `https://en.cppreference.com/w/cpp/container/map/erase`
  - `https://en.cppreference.com/w/cpp/container/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://en.cppreference.com/w/cpp/utility/pair`).

Exercise 4. Auto-reducing Buffer

Daisy is writing a program that stores and retrieves `int`eger values from a sensor.

Her program needs to implement an *auto-reducing* buffer that accumulates values up to a maximum capacity n; when n is exceeded, the oldest value within the buffer capacity is replaced by the average of the oldest value within the buffer capacity and the oldest values beyond the buffer capacity.

Therefore, she plans a `AutoReducingBuffer` class with the following interface:

- `write(v)`: appends value `v` to the buffer; if, after appending `v`, the buffer exceeds the maximum capacity, the oldest value within the buffer capacity is replaced by the average of the oldest value within the buffer capacity and the oldest values beyond the buffer capacity.
- `read()`: removes the oldest value from the buffer and returns it; if no value is in the buffer, it returns the default value specified in the `AutoReducingBuffer` constructor (see below).
- `occupancy()`: returns the number of elements in the buffer; calling `write()` increases the occupancy by 1 until the buffer capacity is reached, while calling `read()` decreases the occupancy by 1.
- `contains(v)`: `true` if the buffer contains value `v`; otherwise, it returns `false`.

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 with this exam paper (in a separate ZIP archive).

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

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

- suppose that we create `buf = AutoReducingBuffer(3,-1)` (i.e. `buf` has a capacity of `3` elements and a default value `-1`);
- suppose that `buf.write(9)` is invoked, followed by `buf.write(1)`. Then, a call to `buf.contains(9)` must return `true`. Also, a call to `buf.contains(1)` must return `true`, and a call to `buf.occupancy()` must return 2;
- now, suppose that `buf.read()` is invoked. Then, a call to `buf.contains(9)` must now return `false`, and a call to `buf.occupancy()` must return `1`, since `9` was the oldest value in the buffer;

This exercise continues on the next page...

- now, suppose that `buf.write(3)` is invoked, followed by `buf.write(8)`, `buf.write(12)` and `buf.write(7)`. Then, a call to `buf.occupancy()` must return `3` since the buffer capacity was reached. A call to `buf.contains(1)` must now return `false`, since `1` is beyond the buffer capacity. A call to `buf.contains(8)` must return `false`, and a call to `buf.contains(4)` must return `true`, since `8` has been replaced by `4` (i.e., the average between `1`, `3`, and `8`);
- now, suppose that `buf.write(5)` is invoked. Then, a call to `buf.contains(4)` must now return `false`, and a call to `buf.contains(6)` must return `true`, since the oldest value within the buffer capacity is no longer `4`, but `6` (i.e., the average between `1`, `3`, `8` and `12`);
- finally, suppose that `buf.read()` is invoked. Then, a call to `buf.contains(6)` must now return `false`, and a call to `buf.occupancy()` must return `2`, since `6` was removed from the buffer when `buf.read()` was invoked.

Tasks. Help Daisy by completing the following tasks. **IMPORTANT:** for these tasks you are required to submit ***both*** files `ex04-library.h` and `ex04-library.cpp` enclosed in a ZIP archive.

NOTE: you are free to define the `private` members of `AutoReducingBuffer` 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(v)`, `read()`, `occupancy()` and `contains(v)`. When the occupancy of the buffer is at its maximum, you are also free to compute the average of the oldest value within the buffer capacity and the oldest values beyond the buffer capacity whenever `write()` is invoked, or to do so when `read()` is invoked.

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

1. define a constructor for `AutoReducingBuffer` 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 `AutoReducingBuffer`, override the *pure virtual methods* of `Buffer` (i.e., those with "=0"), and add the following `public` method to the class interface:
   - `boolean contains(int v)`
3. finally, write a placeholder implementation of all the `AutoReducingBuffer` methods above (e.g. they may do nothing and/or just return `0` when invoked).

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 AutoReducingBuffer::write(int v)
unsigned int AutoReducingBuffer::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).

Special case: if, after invoking `write(v)`, the maximum capacity is exceeded, the oldest value within the buffer capacity must replaced by the average of that value and the oldest values beyond the buffer capacity.

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

```
int AutoReducingBuffer::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).

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

```
boolean AutoReducingBuffer::contains(int v)
```

When `contains(v)` is invoked, it returns `true` if the buffer contains value `v`. Otherwise, it returns `false`.