Software Development with C++ Templates

Lab Submission 2

Exercises should be solved in groups of two. However, with approval from the lecturer, exercises may also be solved alone or in groups of three. Groups of three students will face more stringent requirements on code readability, for instance, the code has to be perfectly structured and robust (i.e., error handling, etc).

Each exercise should be implemented together with a small program testing the individual functions implemented for the exercise. Those test cases should check whether the code correctly handles errors, such as division by zero, etc.

Several exercises build on top of each other. Before continuing to implement the successor, please create a copy of the previous example, so that each step can be shown during the exercise submission.

If the code is not self-explanatory, it should be commented. For instance, a single variable assignment in the constructor or the beginning of a function should not be commented with “initialization.” On the other hand, if the structure of a function is not clear and cannot be split up into self-speaking sub-functions, a short comment should be inserted indicating the individual phases.

Use a consistent coding style at least within a given exercise, preferably across all of your exercises.

If you did not submit the exercises from the 5th lecture, submit them together with this lab submission. If you did not implement connect 4 from the first lab submission, implement it now and you will be rated 80% of the points from the first lab submission.

Exercise 6.1: Inheritance-Based Connect 4

Implement an inheritance-based version of the Connect 4 game. Connect 4 builds on a playing field composed out of 7 columns each having 6 rows. When a player puts a stone into a column, gravity pulls the stone towards the lowest unoccupied row. The player who first has 4 stones in a row (horizontally, vertically, or diagonally) wins.

After each turn display the game field using simple ASCII graphics. Implement the game in such a way that players can be exchanged easily. To facilitate this, implement a playfield class that is based on the following playfield skeleton. You may extend these classes but you are not allowed to modify the class itself.

Implement a human player implementing the player interface as well as a computer player. For the computer-player, implement it in a way that you can use the same code for the template-based version and the inheritance-based version with, if necessary, a small façade or adapter.

This game will later-on be extended to allow exchanging players. To facilitate this, implement a playfield class that is based on the playfield interface. Do not change this interface.

```c++
#ifndef PLAYFIELD_H_
#define PLAYFIELD_H_
```
class playfield {
public:
    // the size of the field
    const static int width=7;
    const static int height=6;

    // the elements stored at individual field positions
    const static int none=0;
    const static int player1=1;
    const static int player2=2;

    // returns the stone (none/player1/player2) at the position
    // 0 <= x <= width
    // 0 <= y <= height
    // stoneat(0,0) ............... top left
    // stoneat(width-1,height-1) ... bottom right
    // if we insert a stone in a new game in column i,
    // it lands at (i,height-1)
    virtual int stoneat(int x, int y) const = 0;
    virtual ~playfield() {};
}

#endif /* PLAYFIELD_H_ */

Implement a computer player. The computer player of this version does not have to be intelligent. At a minimum, however, the computer player should be able to identify whether the opponent can win the game by placing a stone and block this.

The computer player shall inherit from the player interface. It can access the playfield through the playfield interface. Your player should not rely on any extra functionality provided by the playfield class. That is, no down-casts from the playfield interface should be necessary. Do not modify the player interface.

#ifndef PLAYER_H_
#define PLAYER_H_

#include <iostream>
#include "playfield.h"
#include "my_playfield.h"

class player {
public:
    virtual int play(const playfield &field) = 0;
    virtual ~player() {};
}

#endif /* PLAYER_H_ */

Exercise 6.2: Demonstrate the use of std::mem_fun

Write a small program that demonstrates the use of std::mem_fun and explain its purpose.
Exercise 6.3: Connect 4 Interoperability

Improve the computer player of your “Connect 4” implementation and make sure you can accommodate computer players from your colleagues. Get at least 2 computer players from other colleagues and see who is better. In order to be able to do this you need to do the following (see the web for details):

- You need to implement a class religiously implementing the playfield interface that gives the computer player access to the playfield through the stoneat(x,y) function.
- Make sure that your program works in such a way that two computer players can play against each other.
- Do not change the playfield or player interfaces or otherwise your implementation will not work with your colleagues’ implementations.

Ideally, interoperability applies to both the template-based and the inheritance-based version of your computer player.

Exercise 6.4: Ensure your RPN calculator works with \texttt{std::complex}

Ensure that your RPN calculator works with the complex numbers provided by the C++ Standard Library. How do you deal with the minimum function? Make several suggestions, implement one of them.

Exercise 7.1: Implement an Iterator that Performs Range Checking

Implement an iterator that acts like an STL iterator and that performs range checking. Ideally, your iterator is initialized in the constructor with the two iterators indicating the range (first to one beyond the last element in the sequence) within which your iterator may be moved.

If the iterator points to the first element and is decreased OR if the iterator points to the last element and is increased signal an error – choose an appropriate form of signaling the error.

Exercise 7.2: Stack with Inheritance

Implement the stack class with the stack limited to 256 elements and virtual members from lecture 6. Create a subclass called unlimited stack. How do you implement the class? How well can the original stack be replaced with your unlimited version of the stack? What happens in case of slicing?

Exercise 7.3: Merge containers with \texttt{std::back_inserter}

Revisit your implementation of the function to merge containers. Reimplement by making use of the \texttt{std::back_inserter} helper class. In how far does the new implementation differ from your original implementation?

Exercise 7.4: Locking class

Implement a class that handles the locking of files similar to the DBLocker class discussed in the lecture. Make use of that class in the persistent version of your RPN calculator.
Exercise 8.1: Connect 4 with Factory

You obtained already computer players from your colleagues. To make it easier to choose between the different players at program start, create those computer players with a factory modelled after the one from lecture 8. In the hopefully unlikely case you did not manage to get players from your colleagues, ensure that at least a human player that asks a human for its turn and a computer player are registered at the factory.

Note: Those students who create a library from all the object files and link the library, link the object files directly with the main program, or otherwise the objects will be ignored as no linker-visible dependencies exist.

Exercise 8.2: Spell Checker with Regular Expressions

Reimplement your spell checker. Use an sregex_iterator with "(\w+)" as regular expression for parsing full words from the input, i.e., omitting punctuation characters. Your dictionary file, this time contains regular expressions, for each word you read check each regular expression, whether it matches the word, if none matches, display the incorrectly spelled word on the terminal.

This time the spell checker only needs to check words, and display the unknown words on the terminal, no need for replacing incorrect words, etc.

Exercise 8.3: Connect 4 with Threads

At the end of a computer player’s move, create one or more threads that computes the computer player’s next move (i.e., while the other player is thinking).

A trivial implementation would

- Create 7 threads (one for each possibility of the opponent)
- When it’s the computer players turn again wait for the 7 threads to finish
- And pick the move suggested by the thread that simulated the other player’s move

You can also follow a different approach as long as you have a thread running providing some computation for your computer player’s next move.

Exercise 8.4: Constructor, Destructor Execution

In what order are constructors and destructors being executed when an object is constructed or destructed respectively. Show the same in case of multiple inheritance.

Justify your answer with a program that illustrates your answer.

Exercise 9.1: Change format for single output operation

It would be nice if we could change format for only a single value to be output such as shown by the following snippet:

```cpp
Form gen4(4);
void f(double d) {
    Form sci8=gen4;
    sci8.scientific().setprecision(8);
}  
```
Exercise 9.2: Output formatting through a custom streambuf

Write a streambuf that nicely indents the output

- When a ( or { or [ is encountered the following lines are indented by n spaces
- When a ) or } or ] is encountered the following lines are outdented by n spaces
- When a string is encountered enclosed by “ or ‘, characters until the string terminates with the same characters “ or ‘ are ignored
- Whitespaces at the beginning of a line are ignored (since you are handling the indenting yourself)
- To make this exercise easier, you do not need to maintain your own buffer (setp); if you do not specify a buffer, for every character the overflow function will be called

You may slightly adapt the above rules if you prefer to format a stream of a different type as long as you your streambuf needs to maintain an internal state in order to correctly format the output.

Write a program that allows to use your streambuf on some sample C, C++, Java, JSON, … code.

Exercise 10.1: Distance function with enable_if

Reimplement the distance function that used dynamic algorithm selection (lecture 4) with the enable_if feature we have seen in lecture 10.

Hint: If you use the using directives from slides 8 and 9, life gets easier and the code more readable.

Exercise 10.2: RPN Calculator with optional min function (optional)

The BOOST concept classes and enable_if don’t work well together. As a result making this exercise work will require some tinkering and the solution may not be perfect anyway. Hence, this exercise is optional.

Extend the RPN calculator such that the min function is available for ordered types but unavailable for unordered types (i.e., complex numbers) using the metaprogramming techniques from lecture 10. They may be combined with techniques from other lectures though.