# Exercises Day 5

October 02

***Problem 1: Fun with iterators***

In this exercise we will explore the use of STL iterators and the algorithms library. If you are uncertain what a function does or how it works, look it up at your favourite C++ resource, e.g. `cppreference.com`

- Create an `std::ostream_iterator`, and use that together with the `std::copy` algorithm to print a vector to the terminal
- Use the same algorithm, but the `reverse_iterator` of the vector to print it in reverse
- Make use of the iterator constructor of the `std::set` and `std::vector` to sort and delete the copies of the elements of a vector
- Create an `std::back_inserter` iterator and use this together with `std::copy_if` and a unary function to filter one vector into another

***Problem 2: Locking `std::cout`***

Say you have written a piece of code that prints everything to `std::cout`, but sometimes when running the software you don't want to send output to the screen. You could of course solve this by going through you program, recode it and make it accept an `std::ostream` object, but there is another possible solution.

It is possible to save and change any stream's buffer using the `.rdbuf()` function. So without arguments you access the stream's buffer, and with arguments you change the stream's buffer. Use this information to create a `CoutLock` class that changes `std::cout`'s buffer to print to a file when constructed, and reinstates `std::cout`'s functionality when going out of scope:

```
{
  CoutLock lock {"file"};
  annoyingPrintingFunction();
}

std::cout << "Still works" << std::endl;
```

***Problem 3: Blackjack***

The final game in this course to implement is blackjack. I leave it up to you do decide how feature complete you want your game to be, but the basic game should be available. Once more we need a couple of natural classes, such as:

- A `Card` class that represents a single playing card. How you want this implemented is of course up to you
- A `Deck` class which contain a bunch of cards. It should of course be able to deal cards and be shuffled and maybe restocked. Think about which STL container is the most suitable, and remember that a normal deck should contain 52 distinct cards and when one has been dealt it can't be dealt again. You might want a `DeckHandler` class or something along those lines that can fill and shuffle the `Deck`. Should also be possible to fill the deck with more than one normal "deck of cards" so that you can beat those pesky card counters
- A `Blackjack` class that manages the game interface, such as prompting the player on what to do, checking win/loose conditions and so forth

Once again you might of course want to add more helper classes as you find it natural.

*Optional:* If you want to put some more time into it you can add a gambling aspect to it so that the player has credits it can bet. In this case you might want to implement a `Player` class to keep track of such things.

***Problem 4: Linking with boost***

Finally you will try to make use of an external library and get it to link properly. Many of boosts libraries are header only libraries, which is great, because it means that you can use them without thinking about linking. They can even be installed without building anything, just place the header files in the appropriate directories and you are good to go.

In this exercise we will use a non-header only library from boost, namely the regex library. The documentation for this library can be found at:

`http://www.boost.org/doc/libs/1_59_0/libs/regex/doc/html/index.html`

Go back to the `AddressBook` class you created back on day three, and implement a search with regular expressions using this boost library. There should be an article on the course homepage about linking external libraries, and the library file for this library should be called `libboost_regex.a`