5.1 String Algorithms Library

Summary and Goals

From Boost documentation:
"The String Algorithm Library provides a generic implementation of string-related algorithms which are missing in STL. It is an extension to the algorithms library of STL and it includes trimming, case conversion, predicates and find/replace functions. All of them come in different variants so it is easier to choose the best fit for a particular need. The implementation is not restricted to work with a particular container (like `std::basic_string`), rather it is as generic as possible. This generalisation is not compromising the performance since algorithms are using container specific features when it means a performance gain."

We have included this library in the course because it addresses topics that you may sometimes need in your life as developer, specifically performing operations on strings. It is easy to use and it offers more functionality than STL in this regard and it is easier to learn and to use than C++11 regex in my opinion. A possible application is to use *String Algorithm* for the basic string handling chores and to use C++11 regex for the more advanced pattern-matching activities.

The exercises in this section test your knowledge of the library. We conclude the exercises with a mini-project on extracting financial commodity information from files containing *comma-separated* (CSV) data.

1. (String Algorithm: Preprocessing Functions)

In this exercise we use functions to prune and spruce up a string in different ways before performing other operations on it. Create a number of string containing a combination of digits, alphanumeric characters and special characters.

Answer the following questions:

a) Trim leading and trailing blanks. Use two versions in which the input string is modified and in which a copy of the input string is created. Use the default character separator (blank space).
b) Perform the operations in part a) by trimming the string using different predicates, for example trimming leading and trailing characters such as "AAA", for example. You can use Boolean operator such as OR and AND.
c) Test if a string starts or ends with a given string. Consider both case-sensitive and case-insensitive variants.
d) Test if a string contains another string. Consider both case-sensitive and case-insensitive variants. Test if two strings are equal.

2. (Classification)

The library has support for predicates such as: does the string contain alphanumeric characters, digit, HEX characters, upper/case, control characters and so on. For example, here is an example to test if a string is lower case:

```
std::string str2("abdz");
std::cout << all(str2, is_lower()) << '\n'; // true
```

It is possible to combine predicates (NOT, OR, AND) to test if a string contains (or does not contain) combinations of different kinds of characteristics. It is possible to use de Morgan's Laws to emulate XOR:

```
// Can use de Morgan's Laws
// !(A || B) == !A && !B
// !(A && B) == !A || !B
```

Answer the following questions:

a) Consider the string `std::string str1(" abd1 234\*")`. Which predicates will you use to test the following and what is the outcome (`true` or `false`) (use function `all()`)
   - P1 Recognise digits or letters.
   - P2 Recognise digits and not letters.
   - P3 Recognise characters from the range [‘a’,’z’].

b) Create a simple password checker function that accepts a target string as input and result type of (`true` or `false`) and a string stating the cause of possible error or success. Requirements are:
   - It must contain at least 8 characters.
   - It must contain a combination of digits and characters.
   - At least one character must be upper case.
   - No control characters allowed.
   - No spaces allowed.

   Test the function with strings such as:

```
//std::string pwd("DanielDuffy1952");
//std::string pwd("DanielDuffy");
//std::string pwd("U19520829");
std::string pwd("19520829U");
```

   What is the return type in these cases?

3. (Splitting and Joining of Strings)

Split functions extract information from a string while join functions merge information from several sources (for example, from an STL container) to produce a single string. In this exercise in particular, we split a delimited string into a vector of strings or of doubles while we join a vector of strings into a delimited string.

Answer the following questions:

a) Consider the string `std::string sA("1,2,3,4/5/9*56")`. It has delimiters / and *. Split this string into a vector of strings and then join them to form the string `std::string sA("1/2/3/4/5/9/56")`.
b) Consider a string that models dates, for example “2015-12-31”. Write a function that converts strings having this format into a Boost date. You will need to use Boost `lexical_cast` to convert strings to integers.
c) Consider a string consisting of string pairs of the form key = value (for example, port = 23, pin = 87, value = 34.4). Write a function to convert the string to an instance of `std::map<std::string, double>`. In other words, we create *name-value maps* .

4. (Time Series Data, mini-application)

In this exercise we process the data in an ASCII file in CSV format. The regex format is:

```
//
// File := Header | (Row)*
// Row  := Date | (data)*
//
```

The test input file is:

– Date,Open,High,Low,Close,Volume,Adj Close

– 2013-02-01,54.87,55.20,54.67,54.92,2347600,54.92

– 2013-01-31,54.74,54.97,53.99,54.29,3343300,54.29

– 2013-01-30,54.84,55.35,54.68,54.68,2472800,54.68
– 2013-01-29,53.68,54.49,53.65,54.37,1739600,54.37

– 2013-01-28,53.48,53.49,53.12,53.28,1110900,53.28
– 2013-01-25,53.66,53.81,53.36,53.51,1006100,53.51
– 2013-01-24,52.69,53.25,52.61,52.97,821800,52.97
– 2013-01-23,52.43,52.76,52.21,52.76,864000,52.76
– 2013-01-22,52.59,52.96,52.41,52.79,923700,52.79
– 2013-01-18,52.79,52.85,52.22,52.69,1744400,52.69
– 2013-01-17,53.25,53.27,52.94,53.12,786100,53.12
– 2013-01-16,52.86,53.20,52.78,52.95,773000,52.95
– 2013-01-15,52.79,53.25,52.75,53.20,872100,53.20
– 2013-01-14,52.89,53.02,52.68,52.91,1107100,52.91
– 2013-01-11,52.59,52.92,52.32,52.88,1160800,52.88
– 2013-01-10,52.33,52.50,52.11,52.38,1185200,52.38
– 2013-01-09,51.59,51.87,51.54,51.55,822200,51.55
– 2013-01-08,51.58,51.66,51.32,51.58,1092500,51.58

The objective of this exercise is to design a C++ program to process this file and produce output that is an instance of the following data structure:

```
using Data =
std::tuple<boost::gregorian::date, std::vector<double>>;
using ResultSet = std::list<Data>;
```

You will need to know the following:
- Boost date and time.
- Boost lexical_cast.
- Relevant functionality in String Algorithms library.
- Working with text files.

Remark: The exercises in this section do not discuss the following functionality in *String Algorithms* library:
- Erase/replace algorithms.
- Find algorithms.
- Finders and formatters.
- Iterators.

5.2 5.3 Regex

Summary and Goals

In this section we give exercises on Boost Regex and C++11 Regex. The latter library seems to be based on the former library and their interfaces are almost identical. In general, you should try to use C++11 syntax in your code; failing that, you can use Boost Regex which has a bit more functionality than C++11 regex. There are a number of Boost libraries that complement C++11 regex:

- *Tokenizer:* break a string or character sequence into a sequence of tokens.
- *String Algorithms* (already discussed).
- *Xpressive*: writing regular expressions as strings or as expression templates.
- *Spirit*: parser framework modelling parser as EBNF grammars using inline C++.

Please feel free to use them if you prefer them to the sometimes what stodgy (IMO) regex functionality. In the following exercises you are expected to provide extra examples to those that we have proposed here. It is important to get hands-on nitty-gritty experience by **actually coding up lots of examples**.

1. (Regex 101, `regex_match()`)

We present a number of test cases of regular expressions and target strings. Which evaluate to `true` and which evaluate to `false`?

a)
```
regex myReg("[a-z]*");
regex myReg2("[a-z]+");

std::string s1("aza");
std::string s2("1");
std::string s3("b");
std::string s4("ZZ TOP");
```

b)
```
// Digits
regex myNumericReg("\\d{2}");
std::string f("34");
std::string s("345");
```

c)
```
// Alternatives
regex myAltReg("(new)|(delete)");
std::string s4A("new");
std::string s5("delete");
std::string s6("malloc");
```

d)
```
// Use of Kleene star syntax
regex myReg3("A*");
regex myReg4("A+");
regex myReg5("(\\d{2})");
//regex myReg6("\\d{2, 4}");
regex myReg7("\\d{1,}");

std::string testA("1");
```

e)
```
// Alphanumeric characters
std::cout << std::endl;
regex rC("(\\w)*");              // Alphanumeric (word) A-Za-z0-9
regex rC1("(\\W)*");             // Not a word
regex rC2("[A-Za-z0-9_]*");      // Alphanumeric (word) A-Za-z0-9

std::string sC1("abc");
std::string sC2("A12678Z909za");
```

Experiment with other kinds of regular expressions (for example, ECMA and Perl) and target strings.

2. (Searching, `regex_search()`)

We wish to find subsequences of the input string that match a regular expression. Consider the regular expression:

```
regex mySearchReg("(rain)|(Spain)");
```

and the target string:

```
std::string myText("The rain in Spain stays mainly on the plain");
```

Search in this string and determine if the regular expression is matched (use class `smatch`).

3. (Regex Iterators, `sregex_iterator` 101)

In this example we create a `std::set<std::string> result` from a target string such as `std::string S2 = "1,1,2,3,5,8,13,21"`. The corresponding regular expression is `std::regex myReg("(\\d+),?")`.

Answer the following questions:

a) Create an instance `it` of `sregex_iterator` based on `S2` and `myReg`.
b) Iterate over , converting its dereferenced value and inserting it into the output result.
c) Print `it` and check your output.
d) Modify the code so that the output is of type `std::set<int>`.

4. (Token Iterator, std::sregex_token_iterator)

This class allows us to process all the contents between matched expressions, for example when splitting a string into separate tokens separated by something. This class supports STL strings, C strings, wide strings and wide C strings. We can choose:

- Find all subsequences between matched regular expressions (token separators).
- Find all the matched regular expressions.

These options are activated as it were by giving the values -1 and 0, respectively in the constructor of `std::sregex_token_iterator`.

Answer the following questions:

a) Consider the regular expression and the target string:

```
std::regex myReg10("/");
std::string S3 = "2016/3/15";
```

   Extract the data 2016, 3 and 15 from this string.
b) Extract '/' from this string.
c) Choose appropriate data structures to contain the extracted data in both cases.

5. (Extracting name-value pairs, using Regex and String Algo)

Many applications process ASCII data in the form of key/value pairs. The objective of this exercise is to process this kind of regular expression and to output a map.

Answer the following questions:

a) Consider string of the form `std::string sA("a = 1, b = 2, c = 3")`. Define a regular expression defined by the delimiter ",". Use `std::sregex_token_iterator` to extract strings of the form "a = 1" etc.
b) For each generated string from part a) use Boost Algo `split()` to extract its left and right parts, which will become the map's key and the value, respectively.

6. (ECMAScript Grammar Example)

Consider the regular expression:

```
std::regex ecmaReg
("((\\+|-)?[[:digit:]]+)(\\.(([[:digit:]]+)?))?((e|E)((\\+|-
)?)[[:digit:]]+)?");
```

Answer the following questions:

a) What kinds of numbers does this regular expression subsume?
b) Create a program to test valid and invalid numbers.
c) For valid numbers, convert them to `double`.

7. (Replacing Regular Expressions)

Consider the target string and regular expression:

```
std::string text("Quick brown fox");
std::regex vowels("a|e|i|o|u");
```

Use `std::regex_replace` to produce the following output:

```
"Q**ck br*wn f*x"
```

8. (Time Series Data, mini-application)

The objective of this exercise is to code it up using C++11 Regex and/or Boost Regex. We (you!) have already solved this problem using the String Algorithms library.

In this exercise we process the data in an ASCII file in CSV format. The regex format is:

```
//
// File := Header | (Row)*
// Row  := Date | (data)*
//
```

The test input file is:

- Date,Open,High,Low,Close,Volume,Adj Close

- 2013-02-01,54.87,55.20,54.67,54.92,2347600,54.92

- 2013-01-31,54.74,54.97,53.99,54.29,3343300,54.29

- 2013-01-30,54.84,55.35,54.68,54.68,2472800,54.68

- 2013-01-29,53.68,54.49,53.65,54.37,1739600,54.37

- 2013-01-28,53.48,53.49,53.12,53.28,1110900,53.28

- 2013-01-25,53.66,53.81,53.36,53.51,1006100,53.51

- 2013-01-24,52.69,53.25,52.61,52.97,821800,52.97

- 2013-01-23,52.43,52.76,52.21,52.76,864000,52.76

- 2013-01-22,52.59,52.96,52.41,52.79,923700,52.79

- 2013-01-18,52.79,52.85,52.22,52.69,1744400,52.69

- 2013-01-17,53.25,53.27,52.94,53.12,786100,53.12

- 2013-01-16,52.86,53.20,52.78,52.95,773000,52.95
- 2013-01-15,52.79,53.25,52.75,53.20,872100,53.20
- 2013-01-14,52.89,53.02,52.68,52.91,1107100,52.91
- 2013-01-11,52.59,52.92,52.32,52.88,1160800,52.88
- 2013-01-10,52.33,52.50,52.11,52.38,1185200,52.38
- 2013-01-09,51.59,51.87,51.54,51.55,822200,51.55
- 2013-01-08,51.58,51.66,51.32,51.58,1092500,51.58

The objective of this exercise is to design a C++ program to process this file and produce output that is an instance of the following data structure:

```
using Data =
std::tuple<boost::gregorian::date, std::vector<double>>;
using ResultSet = std::list<Data>;
```

You will need to know the following:

- Boost date and time.
- Boost `lexical_cast`.
- Relevant functionality in Boost/C++11 *Regex* libraries.
- Working with text files.

5.4 5.5 5.6 Boost Hash and Boost Unordered

Summary and Goals

The exercises in this section centre around Boost *Functional.Hash* and *Boost Unordered*. Much of the functionality is already in C++11. To this end, we try to do justice to both libraries.
The template parameters of an unordered map (for example) are:

```
template <
    class Key, class Mapped,
    class Hash = boost::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<Key const,Mapped>> >
                 class unordered_map;
```

Similar definitions exist for unordered multimaps, sets and multisets.

1. (Hash 101)

A hash function is a mapping from some data type to `std::size_t`. Said differently, it transforms keys into numbers.

Answer the following questions:

a) Create two generic functions to hash arbitrary data types using Boost and C++11.
b) Test the functions using integers, strings, pointers, `std::numeric_limits<long>::max()`.
c) Create two hashes h1 and h2 of two strings and then compute `h1 ^ (h2 << 1)`.

2. (Hash for User-defined Types and User-defined Hash)

We consider creating hashes for a `Point` (consisting of x and y values) and `LineSegment` (a *composition* of two points).

Answer the following questions:

a) Create hashes for `Point` and `LineSegment` using `boost::hash_combine`. The functionality in `LineSegment` should delegate to the functionality in `Point`.
b) Create a list of `Point` instances and compute its hash using `boost::hash_range()`.
c) Create a function object that represents a *custom hash function* for the `Point` class. Its specification is:

```
struct PointHash :
          std::unary_function < Point, std::size_t>
{
    std::size_t operator () (const Point& pt) const
    {

          // TBD by student
          std::size_t seed = 0;
          boost::hash_combine(seed, pt.X());
          boost::hash_combine(seed, pt.Y());

          return seed;
    }
};
```

3. (Unordered Collections)

In this exercise we create STL multisets and Boost/C++11 unordered multisets of `Point` instances. The objective are showing how to use them and measuring efficiency.

Answer the following questions:

a) Create an unordered multiset of Point instances using *default hash*. Add many points to this multiset using `insert()` and `emplace()`; remove elements from the multiset using `erase()` and `clear()`. Measure processing time using `std::chrono`.

b) Now create your own hash functions and repeat part a) using multisets with these hashes as parameters. Measure processing time using `std::chrono`.
c) Repeat part a) using 'standard' STL multisets. Measure processing time using `std::chrono`. (Remark: unordered containers have O(1) complexity while STL containers have logarithmic complexity).

4. (Variadic Hash)

It is possible to create a single generic function to hash a variable number of arguments of different types (you may need to review your *variadics*).

Answer the following questions:
a) Create a hash function using a seed:

```
template <typename T>
    void hash_value(std::size_t& seed, const T& val)
{
            // TBD
}

template <typename T, typename... Types>
    void hash_value(std::size_t& seed, const T& val, const Types&... args)
{
            // TBD
}
```

Create a test case.
b) Create a generic auxiliary function to create a hash value from a heterogeneous list of arguments (in this case the seed is initialised to zero):

```
template <typename... Types>
    std::size_t hash_value(const Types&... args)
{
    std::size_t seed = 0;
    hash_value(seed, args...);
    return seed;
}
```

Create some test cases, in particular objects that are compositions of other (heterogeneous) objects.

5. (The Bucket Interface)

Create a wrapper function to examine the bucket structure of an unordered set. The interface is:

```
template <typename Key, typename Hash, typename EqPred>
    void BucketInformation(const std::unordered_set<Key, Hash,EqPred>& c)
{
    // TBD
}
```

Answer the following questions:
a) Show the number of buckets and the maximum number of possible buckets.
b) Show current load factor and the current maximum load factor.
c) Show the size of each bucket.
d) Rehash the container so that it has a bucket size of at least some given size.

6. (The full Monte)

We create an unordered set of instances of a given class with user-defined hash function and equality predicate:

```
        std::unordered_set<S, SHash, SEqual> mySet;
```

where

```
   struct S
   {
       std::string f; std::string s;

       S(const std::string& s1, const std::string& s2)
                    : f(s1), s(s2) {}

   };


   class SHash // Hash for class S
   {
   public:
                    std::size_t operator() (const S& s) const
                    {
                            return hash_value(s.f, s.s);
                    }
   };

   class SEqual // Equality for S
   {
   public:
       bool operator ()(const S& lhs, const S& rhs) const
       {
               return (rhs.f == lhs.f && rhs.s == lhs.s);
       }
   };
```

Answer the following questions:

a) Add elements to this unordered set.
b) Examine the bucket information for this unordered set.

5.7 Boost Bimap

Summary, Review and Goals

From Boost documentation:

"*Boost.Bimap* is a bidirectional map library for C++. With *Boost.Bimap* you can create associative containers in which both types can be used as key. A `bimap<X,Y>` can be thought of as a combination of a `std::map<X,Y>` and a `std::map<Y,X>`. The learning curve of *bimap* is almost flat if you know how to use standard containers. A great deal of effort has been put into mapping the naming scheme of the STL in *Boost.Bimap*. The library is designed to match the common STL containers."

In order to complete the exercises in this section you should be familiar with the following concepts:

- UML associations, association class; 1:1, 1:N, N:N relationships.
- The three views of a bimap.
- Customising a bimap: hash function, key comparison and equal function.
- STL container types.

1. (Bimap 101)

We create a bimap representing people/persons (a pair consisting of a string (name) and an integer (age)).

Answer the following questions:

a) Populate the bimap with some names using `insert()`. Does C++11 `emplace()` work with Bimap?
b) Print the *left* and *right* maps of the bimap (first and second views).
c) Search for age based on name and search for name based on age. Use function `at()`. Take exception handling into consideration.
d) Iterate in the bimap using the third view (that is, a *set of relations*).

2. (User-friendly Bimap and Tags)

We create a data structure that emulated DNS (*Domain Name System*) and IP addressing. Some requirements and features of your solution using Bimap will be:

- The mapping is 1:1 (use unordered sets).
- IP addresses (128 bits) using Boost UUID.
- Using tags to improve readability of the left and right maps of a bimap.

Answer the following questions:

a) Create a bimap (call it `DNS`, why not?) with tags `IpAddress` and `DomainName` whose left and right containers are unordered sets. Model domain names as strings and IP addresses as Boost UUID instances.
b) Create some instances of `DNS`. Find a domain name for a given IP address and find an IP address for a given domain name.
c) Create a function to print the contents of the `DNS` 'database'.

3. (Implementing UML *Association Class*)

An *association class* is one that arises when two other classes come together. The association class can have its own functions and data members (attributes). As example we models books written by authors. The price of a book is an *association attribute*.

Answer the following questions:

a) Create a 1 N association between book title and author (choose the appropriate STL data structures). Model book price in Bimap.
b) Print the price of a book from a given author.
c) Now create an association attribute containing a tuple consisting of book abstract and its price.

4. (Collection Types with Bimap)

In general, Bimap implements a mapping between a *domain* D and a *range* (or *co-domain*) R. For convenience the containers contain strings and integers. The main focus is on whether the keys are unique (or not) in D or whether the data are unique (or not) in R.

Answer the following questions:

a) Create bimaps where the domain D is:
   - Set
   - Multiset
   - Unordered set

   and the range R is:
   - List
   - Set
   - Unordered set

b) Create instances of the bimaps in part a) and print their values.

c) Modify the code in part b) to support different kinds of *comparitors* (for example, *std::greater*<>).

5.8 Heap

Summary and Goals

We recall that a *max heap* is a special kind of binary tree with the properties:

- The value of each node is not less than the values stored in each of its children.
- The tree is *perfectly balanced* and the leaves in the last level are all in the leftmost positions.

It is possible to define a *min heap* similarly. Max heaps and heap algorithms are supported in both Boost and C++. As a special case, we discuss *priority queues* in STL.

1. (Heap 101)

In this exercise we create heaps from STL containers.

Answer the following questions:

a) Create a vector with the following elements:

```
vector<int> vec2{ 3,4,5,6,7,5,6,7,8,9,1,2,3,4 };
```

b) Call `std::make_heap()` on this vector and print the result. Draw the corresponding binary tree by hand.
c) Pop the root of the heap by calling `pop_heap()`. What are the effects of calling this function (or not calling it) if the underlying container is not a heap? And why do you need to call `pop_back()` after calling `pop_heap()`?
d) Push a value onto the heap by calling `push_heap()`. What are the effects of calling this function (or not calling it) if the underlying container is not a heap? And why do you need to call `push_back()` before calling `push_heap()`?
e) Sort the heap into a sorted collection.
f) Find the largest element in a heap.

2. (Priority Queue)

In this exercise we create a *priority queue* that is supported in STL:

```
template<

    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>

> class priority_queue;
```

In this case elements are read according to their priority.

Answer the following questions:

a) Create a default priority queue of integers with elements {10,5,20,30,25,7,40}. Print the queue.
b) Modify the queue in part a) using `std::greater<int>` as comparator function. Print the queue.
c) Create a priority queue using an underlying data container and the following lambda function as comparator:

```
auto cmp = [](int left, int right)->bool { return (left > right); };
```

5.9 Signals and Slots

Summary and Goals

This section is a set of exercises on the (thread-safe) Signals2 and Slots library. It is a signature-based approach to the *Observer* pattern. We shall use signals and slots in later modules.

1. (Signals 101)

This first exercise is to get you thinking in terms of *function signatures* and less in terms of traditional object-oriented polymorphism. In particular, a signature can be implemented by:

- Free functions (function pointers).
- Stored lambda functions.
- Static member functions.
- Instance-level member functions.
- Binded member functions (to resolve *function argument mismatch*).

For the moment, we focus on a simple signature describing signals (and slots!):

```
// DEPRECATED boost::signal<void ()> signal;
boost::signals2::signal<void ()> signal;
```

The objective is to show that client code uses a signal in some way, while the signal can have several attached slots. The slots can be any of the above signal implementations.

Answer the following questions:

a) Create a signal and create slots using lambda functions, free functions and function objects. Connect the slots to the signal and emit the signal. Examine the output.

b) Create the following class:

```
struct MyStruct
{
    double val;

    MyStruct(double v) { val = v; }

    void modify(double newValue)
    {
           val = newValue;
           std::cout << "A third slot " << val << std::endl;
    }
};
```

Create an instance of this class and create a slot based on `modify()` whose argument is equal to some value. Add this slot to the signal and then emit the signal.

c) Experiment with disconnecting slots in your code.

d) The example in parts a) to c) was based on one signal and several slots connected to it. An interesting observation is that a signal can also be connected to another signal! This allows us to create trees of signals and slots. As an exercise, create the signals A,B,C and D as well as the slots S1, S2, S3 and S4.

```
// Define potential emitters
boost::signals2::signal<void()> signalA;
boost::signals2::signal<void()> signalB;
boost::signals2::signal<void()> signalC;
boost::signals2::signal<void()> signalD;

// Define slots
auto slotB = []() {std::cout << "Slot B called by B\n " << std::endl; };
auto slotC = []() {std::cout << "Slot C called by C\n " << std::endl; };
auto slotD1 = []() {std::cout << "Slot D1 called by D\n " << std::endl; };
auto slotD2 = []() {std::cout << "Sloat D2 called by D\n " << std::endl; };
```

Make the following connections:

- Signal B connected to signal A.
- Signal C connected to signal B.
- Signal D connected to signal C.
- Slot B connected to signal B.
- Slot C connected to signal C.
- Slot D1 connected to signal D.
- Slot D2 connected to signal D.

Emit the signal. Examine what happens when you try to disconnect signal C from signal B. Is it possible or do you have to first disconnect the terminal slot?

2. (Slot Groups)

Slots are free to have side effects. And there maybe dependencies between the order of invocation of slots. It is possible to place slots into *groups* that are ordered in some way. The default group value is `int` and this is what we use here.

Answer the following questions:

a) Modify the code in exercise 1 by placing each slot in its own group.
b) Run the program and examine the output.

3. (Combining returned Values from Slots)

The slots in exercises 1 and 2 had `void` as return type. For non-`void` functions we can create user-defined combiners; a *combiner* is a function object that iterates in the slots of a signal and saves their return types in some way. Some examples of where combiners can be used are:

- Stopping the slot chain execution when a certain condition/threshold has been reached.
- Combining all the returned values in some way (for example, sum/average, minimum/maximum of the return types of a sequence of slot functions).
- Returning all computed values in a container.

In general, a combiner is a function object and is a template parameter in `signal`.

Answer the following questions:

a) Consider the signature of a new signal type:

```
boost::signals2::signal<bool (), BootstrapCheck> sig;
```

where `BootstrapCheck` is a combiner:

```
struct BootstrapCheck
{ // Iterate in slots and return first 'false'
  // value; otherwise, 'true'

    typedef bool result_type;

    template <typename InputIterator>
          bool operator()(InputIterator first,
                                  InputIterator last) const
    {

          while(first != last)
          {
                if (!*first)
                {
                      return false;
                }
                ++first;
          }
```

```
            return true;
        }
    };
```

Implement this signal and combiner.

b) Create three slots each returning `true` with the exception of the second slot.
c) Connect the slots to the signal and then call the signal.

This problem can be extended to other problems.

4. (Numeric Combiners)

We use the design from exercise 3 to add the return types of several slots into a single *accumulated number*. The signature in this case is:

```
// Create a signal + custom combiner
boost::signals2::signal<double (double x, double y),
SumCombiner<double>> sig;
```

Answer the following questions:

a) Create a number of slots, each of which takes two numeric arguments as input and combines them in some way to produce another numeric value as output.
b) Create the template combiner that sums the values returned by the slots. The code is a variation of the code in part a) of exercise 3.
c) Test your code and check that the output is correct.

5. (Reengineering The (infamous) GOF *Observer* Pattern)

The *Observer* pattern is well-known and it defines a *one-to-many dependency* between objects so that when one object changes state all it dependents are updated and notified automatically. It can be implemented in a number of ways, for example:

A. Using the object-oriented approach (inheritance and subtype polymorphism).
B. Using collections/lists of function pointers (sample code below).
C. Using Boost signals2 (this exercise).

The objective of this exercise is to port a solution based on B to a solution based on C.

Answer the following questions:

a) Solution B code for the subject/publisher is:

```
// Using boost Function library
typedef boost::function<void (double)> FunctionType;

class Subject
{ // The notifier (Observable)in Publisher-Subscriber pattern
private:


    std::list<FunctionType> attentionList;

public:
    Subject() { attentionList = list<FunctionType>(); }

    void AddObserver(const FunctionType& ft)
                { attentionList.push_back(ft); }

    void ChangeEvent(double x)
    {
            for (auto it = attentionList.begin();
                    it!= attentionList.end(); ++it)
            {
                    (*it)(x);
                }
        }
};
```

Reprogram this code using Boost signals.

b) Create slots:

```
void CPrint(double x)
{
    cout << "C function: " << x << endl;
}

struct Print
        {
    void operator()  (double x)
    {
            cout << "I am a printer " << x << endl;
    }
};
```

c) Connect the slots to a signal:

```
// Create the notifier
Subject mySubject;

// Create the attention list
Print myPrint;
MathsWhiz myMaths(10.0);
Database myDatabase;

mySubject.AddObserver(myPrint);
mySubject.AddObserver(myMaths);
mySubject.AddObserver(myDatabase);
mySubject.AddObserver(&CPrint);


// Trigger the event
cout << "Give the value: "; double val; cin >> val;

mySubject.ChangeEvent(val);
```

d) Test the code.

6. (Layered Communication with Signals)

We discuss how to implement a simpler version of the GOF *Layers* pattern (see Module 5). In this case we discuss a data push model in which data is pushed from one layer to the layer 'above' it. The layers and their responsibilities are:

- *Exterior* (model as main()): where the data is initialised.
- *Hardware*: data is checked to be in the range [2,4]; if not in the range it is set to the value 3.
- *Data*: the data is modified by an algorithm.
- *Communication*: the data is formatted and printed.

The signature in this exercise is:

```
boost::signals2::signal<void (double& d)>
```

Answer the following questions:

a) Create four signals based on the above layering regime and for each signal create a slot that modifies its input data in some way.

b) Connect the signals to each other, starting from the exterior layer where the data is initialised:

```
boost::signals2::signal<void(double& d)> signalExterior;
```

```
double value = -3.7; // in range [2.0, 5.0]
signalExterior(value);
```

See exercise 1 d) for more motivation.

c) Run and test the program.

Remark: Due to the scope, we have not included the *connection* class, *scoped connection* or *automatic connection management*.

5.10 Boost *uBLAS* Matrix Library

Summary and Goals

The purpose of the exercises in this section is to introduce the Boost *uBLAS* matrix library. C++ does not have a native matrix library and for a large class of applications there is a need for various kinds of numerical vectors and matrices for operations such as data storage, mathematical properties and numerical linear algebra.

Boost *uBLAS* vectors are STL-compatible which means that STL algorithms can be used on them.

Boost *uBLAS* has online documentation at www.boost.org and two chapters in "Introduction to the Boost C++ Libraries" by Robert Demming and Daniel J. Duffy, Datasim Press 2012.

In the exercises you should use `double` and `std::complex<double>` as data types.

1. (Dense Vectors 101)

A *dense vector* of length n is one all of whose elements are explicitly given a value. In other words, there are no 'gaps' in such vectors.

Answer the following questions:

a) Create a vector v1 of a given size. Use the indexing operators `()` and [] to access the elements of the vector.
b) Create a vector v2 having the same size as v1. Compute v1 += v2, v1 *= 2.5, v1 -= v3.
c) Use `std::transform` and `std::multiply<>` to multiply v1 and v2 to produce a vector v3.
d) Produce a *scalar vector* of size 100 all of whose values are equal to 5.0.

2. (Dense Matrices 101)

A *dense matrix* with n rows and m columns is one whose n X m elements are stored in memory. In other words, there are no 'gaps' in such matrices. The template parameters are:

- The underlying data type.
- The function object for storage organization (`row-major`, `column-major`).
- Allocation: bounded (fixed-size) or unbounded (dynamic) storage.

Answer the following questions:

a) Create matrices with given row and column using variations of the above template parameters.
b) Use the operator `()` to access and modify the elements of matrices.
c) Add and subtract matrices.
d) Resize a matrix (modify the number of rows or columns).
e) Create the *zero* (all zeroes) and *identity* (all zeroes except on main diagonal where the value is equal to 1) matrices. The identity matrix is square (number of rows == number of columns) but the zero matrix may be rectangular.

3. (Sparse Matrices, Triangular Matrices 101)

These kinds of matrices are used in many kinds of applications. A *sparse matrix* in which a small percentage of elements have values that we are interested in. Most of the elements have zero or undefined values so that we do not need to store them in memory. A *lower-triangular matrix* is one whose non-zero values are on or below the main diagonal while an *upper-triangular matrix* is one whose non-zero are on or above the main diagonal.

Triangular matrices have the following template parameters:

- The underlying data type.
- The function object that determines if the triangular matrix is `upper` or `lower`.
- The function object for storage organization (`row-major`, `column-major`).
- Allocation: bounded (fixed-size) or unbounded (dynamic) storage.

Answer the following questions:

a) Create a sparse matrix whose elements are complex numbers. The interface is similar to that of a dense matrix. Experiment with insertion, erasure and clearing a sparse matrix.
b) Create a number of upper and lower triangular matrices with row and column-major storage organization.
c) Boost uBLAS supports matrix adapters for dense matrices. For example, it is possible to define upper and lower triangular views of a dense matrix. Create these views by taking some examples. (Boost uBLAS also has support for symmetric, Hermitian and banded matrices (as well as the corresponding adapters) but a discussion is outside the scope of this course.

4. (Boost uBLAS and STL Compatibility)

The objective of this exercise is to port the code from exercise 1 in exercise set 3.7 from STL to Boost uBLAS. In that exercise we searched for certain values in STL vectors. Now we do the same for Boost vectors. We recall some of the code:

```
using Range = std::tuple<std::size_t, std::size_t>;
using Result = std::tuple<Range, bool>;

Result find_sequential_greater(const Vector& v, value_type x)
{
    for (std::size_t j = 0; j < v.size(); ++j)
    {
           if (v[j] <= x && v[j+1] > x)
           {
                  return std::make_tuple(std::make_tuple(j, j+1), true);
           }
    }

    return std::make_tuple(std::make_tuple(999, 999), false);
}
```

Answer the following questions (NB: `Vector` is now a Boost uBLAS vector).

a) Now implement the same functionality using `std::find_if`. Remember that the output is in index space and you will need `std::distance` to convert iterators to indexes. Furthermore, create a wrapper for `std::find` to find the index corresponding to a given value in the vector.
b) Implement the *O (log n)* algorithms `std::lower_bound` and `std::upper_bound` to affect the same functionality as in part a). Again, you need (as always) to create a wrapper function.
c) Implement the *O (log n)* algorithm `std::equal_range` to effect the same functionality as in part b). Again, you need (as always) to create a wrapper function.
d) Test the algorithm `std::find_if_not()` on an example of your choice.

5. (Unified Vector Interface using *Template Template Parameters*)

An interesting issue now arises: the Boost code in exercise 4 is essentially a copy of the original STL code. In this exercise we add an extra level of indirection so that we only need to write the generic code **once** and then specialise it to STL vectors, Boost vectors and any vectors that are STL-compatible. To this end, we use the *template template parameter* trick. We give an example of a generic print function for motivational purposes:

```
template <typename T, template <typename S, typename Alloc >
                                      class Container, typename TAlloc>
void print(const std::string& comment, const Container<T, TAlloc>& container)
{  // A generic print function for general containers

    std::cout << comment << ": ";

    // We use lambda for readability
    // a. iterators
    auto f = [](const T& t) { std::cout << t << ", "; };
```

```
        std::for_each(container.begin(), container.end(), f);
        std::cout << std::endl;

        // b. indexing
        for (std::size_t i = 0; i < container.size(); ++i)
        {
                std::cout << container[i] << ",";
        }
        std::cout << std::endl;
    }
```

Some examples of use are:

```
std::vector<double> v1{ 2.0, 3.0, 4.0, -6.0 };
print(std::string("STL"), v1);

boost::numeric::ublas::vector<double> v2(4);
v2[0] = 2.0; v2[1] = 3.0; v2[2] = 4.0; v2[3] = -6.0;
print(std::string("Boost"), v2);
```

The objective now is to use this idea to create searching functionality that is highly generic in the above sense.

6. (Vector and Matrix Proxies)

In general, a *proxy* is an object that provides an interface to another object. For example, we can define the following proxies to matrices (vectors are similar):

- `matrix_row`: a given row of a matrix (one-dimensional).
- `matrix_column`: a given column of a matrix (one-dimensional).
- `matrix_range`: a submatrix of a matrix (two-dimensional).
- `matrix_slice`: allows addressing a sliced sub matrix of a matrix.

These structures have many applications in Numerical Analysis.

Answer the following questions:

a) Create examples of each of the above containers and print their values on the console.
b) We now consider slices; a *slice* defines a set of indices having three components:
   - Its start index.
   - The number of elements (size).
   - The distance between element (*stride*).

For example, `boost::numeric::ublas::slice(2,4,3)` is a slice with four elements, starting with index 2 and distance 3. In other words, the expression specifies the following set of indices: 2, 5, 8, 11. Create the following matrix slice and print its elements:

```
matrix<double> m(3, 3);
matrix_slice<matrix<double> > ms(m, slice(0, 1, 3), slice(0, 1, 3));
```

7. (Vector and Matrix Expressions)

Investigate how to perform the following operations and give some examples:

- Inner and outer products of two vectors.
- Adding and subtracting two vectors.
- The Euclidean, L1 and max norm of a vector.
- L1 and max norms of a matrix.
- The product of a matrix and a vector.