Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	4, Templates, Indexer & Lambdas
Lecturer:	Dr. Markus Lumpe

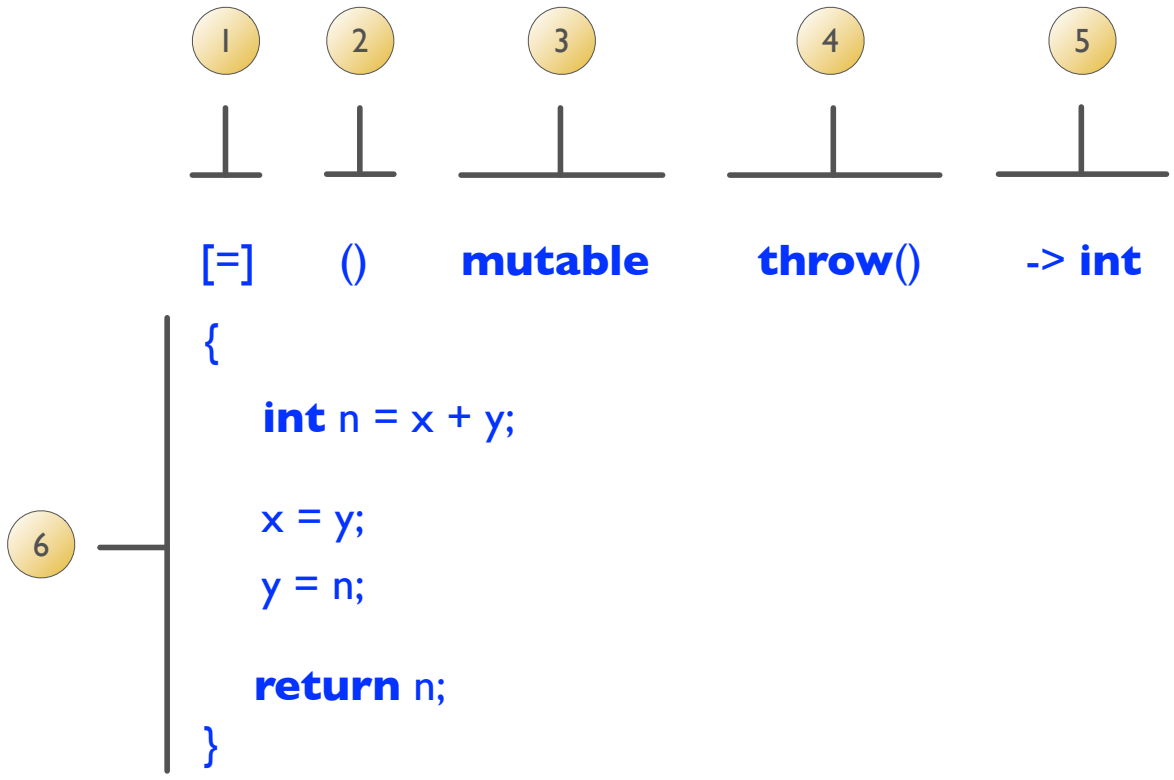

Figure 1: Lambda Expression in C++.

Preliminaries

This tutorial is concerned with the definition of templates, indexers, and lambda expressions. For this purpose, we define a small auxiliary data type which allows us to read data from an input file. Once we have loaded the data into memory, we can use an indexer to provide random access to the data, or use a lambda, to systematically traverse/access the data.

The input data has been randomized. The actual information entailed in the data is hidden. When we use the indexer, we only obtain the raw information at a given index. This is how the task is set up. When using a lambda, we will employ an additional process that performs *sorting on-the-fly*. The process that we target, in combination with the lambda, is similar to *Bubble Sort*, a naïve sorting algorithm with quadratic running time complexity (i.e., $O(n^2)$ which means that we check every element in the input against all other elements in two for-loops). The actual sorting is split in two parts: an outer loop that runs through all possible indices, and an inner loop (implemented in a lambda expression) that performs linear search to map the out index to a corresponding datum.

Format of Input File

The input data is a sequence of decimal numbers stored in a text file. The first number represents the number of value pairs following. Every value pair consists of an index and a datum separated by a whitespace character. Only one value pair occurs per line. The name of the input file is `Data.txt`:

```
1050
738 46
667 96
545 32
549 10
793 32
…
663 32
565 46
630 32
```

The file `Data.txt` contains `1050` key-value pairs, each on a separate line (every line ends with a newline character). The indices in the first column range between 0 and 1049. The values in the second column range between 0 and 255.

In this task, we use a generic map to store the individual key-value pairs. This generic map defines a basic storage facility for key-value pairs. In addition, it also defines I/O operators and a generic type conversion operator. The latter allows us to type cast a key-value pair to a compatible type of our choosing. The details of specification of these operators are somewhat subtle, even though they follow logic principles.

The solution also requires an array to be dynamically allocated at runtime using a **`new`** expression. Technically, we have to allocate an array of objects. The base type has to provide a default constructor to properly initialize the newly created array. If the array goes out of scope, that is, the containing object reaches the end of its lifetime, we have to explicitly free the memory using a **`delete[]`** expression. We define the corresponding behavior in a destructor.

Problem 1

We start with the auxiliary generic class `Map`, which is a template class over the types `Key` and `Value`:

```
#pragma once

#include <istream>
#include <ostream>

template<typename Key, typename Value>
class Map
{
private:
  Key fKey;
  Value fValue;

public:
  Map( const Key& aKey = Key{}, const Value& aValue = Value{}) noexcept;

  const Key& key() const noexcept;
  const Value& value() const noexcept;

  template<typename U>
    operator U() const noexcept;

  friend std::istream& operator>>( std::istream& aIStream, Map& aMap );

  friend std::ostream& operator<<( std::ostream& aOStream, const Map& aMap );
};
```

Template class `Map` is a simple container for key-value pairs. It defines one constructor, two getters, and a generic type conversion operator. In addition, it declares the stream-based input and output operators as friends.

Class `Map` is a template. Consequently, there is no source file (i.e., `Map.cpp`). All member functions have to be implemented in the header file `Map.h`. When the compiler instantiates the template class `Map` for actual type arguments it requires the complete code, not just the signatures of the methods.

The member functions of `Map` are defined as follows:

- The constructor initializes the member variables `fKey` and `fValue` using a member initializer list. Please note, that we give the parameters default values. These default values are obtained by calling the default constructor of the types instantiated for the template type parameters `Key` and `Value`. Hence, we require the actual types to define a default constructor. This is standard for built-in types like **`int`** or **`char`**.
- The getters `key()` and `value()` just return the corresponding attribute value. The getters return constant references, that is, we avoid copying the values and provide read-only access only.
- The type conversion operator `U()` converts a `Map` object to a value of type `U`. The type conversion operator is a template function whose type argument is independent from the template types `Key` and `Value`. The conversion operator performs a static cast of the value component, which has type `Value`, to an object of type `U`. The compiler verifies that such a type conversion is possible.
- The I/O operators provide the corresponding stream-based logic to read and write `Map` objects from an input stream and to an output stream, respectively. The definitions create non-template operators for the types `Key` and `Value`. The input operator has to fetch two values in sequence from the input stream. The output operator sends the key and value enclosed in braces to the output stream.

When implemented correctly, the test code

```
#include <iostream>

#include "Map.h"

using DataMap = Map<int,int>;

DataMap lArray1[] = {{1,32}, {2,68}, {3,83}, {4,80},
                     {5,87}, {6,69}, {7,75}, {8, 52}};
const size_t lSize1 = sizeof(lArray1)/sizeof(DataMap);
int lArray2[] { 1, 2, 3, 0, 4, 5, 5, 6, 0, 7};
const size_t lSize2 = sizeof(lArray2)/sizeof(int);

for ( size_t i = 0; i < lSize1; i++ )
{
    std::cout << "Key-value pair " << i << ": " << lArray1[i] << std::endl;
}

std::cout << "Test type conversion: ";

for ( size_t i = 0; i < lSize2; i++ )
{
    std::cout << static_cast<char>(lArray1[lArray2[i]]);
}

std::cout << std::endl;
```

should produce the following output

```
Key-value pair 0: {1, 32}
Key-value pair 1: {2, 68}
Key-value pair 2: {3, 83}
Key-value pair 3: {4, 80}
Key-value pair 4: {5, 87}
Key-value pair 5: {6, 69}
Key-value pair 6: {7, 75}
Key-value pair 7: {8, 52}
Test type conversion: DSP WEEK 4
```

Problem 2

We use the template class `Map` to define the payload type for class `DataWrapper`. Class `DataWrapper` encapsulates an array of key-value pairs and provides an indexer to fetch the pair that corresponds to a given index:

```
#pragma once

#include "Map.h"

#include <string>

using DataMap = Map<size_t, size_t>;

class DataWrapper
{
private:
  size_t fSize;
  DataMap* fData;

public:
  DataWrapper();
  ~DataWrapper();

  // DataWrapper object are not copyable
  DataWrapper( const DataWrapper& ) = delete;
  DataWrapper& operator=( const DataWrapper& ) = delete;

  bool load( const std::string& aFileName );

  size_t size() const noexcept;

  const DataMap& operator[]( size_t aIndex ) const;
};
```

Class `DataWrapper` is not copyable. The corresponding features are deleted.

- The constructor has to initialize both values to `0` and `nullptr`, respectively. We do not allocate any memory initially.
- The destructor frees allocated memory using the **`delete[]`** operation on `fData`. The delete must work even if no memory has been allocated.
- The `load()` function takes a file name string reference and returns true, if `load()` succeeds. Within function `load()`, we need to create a file input stream, read the size, and fetch all value pairs. The array elements are Map objects that can be read using the corresponding input operator.
- The `size()` function returns the size of the data array.
- Finally, the indexer returns a constant reference to the corresponding `Map` object in the array `fData`.

When implemented correctly, the function `basicIndexer()` in `main()` should produce the following output

```
Using basic indexer:
.`
 `
   '\/ `     \ ,.
    o  - /    d \  _    -| _ '     \;  _ :
   / .            -    |o-      \    ' -',  "  | |      )        ` ' ,8               `-  ,
-  \ '    . ,         |  .`    `   ,_       .,'
     .         ._   `    ` .
…
```

The indexer test code prints some characters, but it is not legible. The program works. Our data is just not properly sorted.

Problem 3

Inspect the function `lambdaIndexer()` in `main.cpp`. It contains the definition of the lambda expression `lOrderedMapper`.

```
auto lOrderedMapper = [&aWrapper] (size_t aIndex) -> char
    {

      ...

    };
```

The lambda expression `lOrderedMapper` captures object `aWrapper` by reference. As a result, we have access to it within the body of the lambda expression. This lambda expression has to implement a linear search that maps `aIndex` to the corresponding payload datum in the container `aWrapper`. This linear search in combination with a for-each loop over the iterator achieves sorting on-the-fly in a manner similar to *Bubble Sort*. For example, if `aIndex` is `738`, then the payload index is `0` (see Format of Input File). The lambda `lOrderedMapper` would have to return the character that corresponds to the unsigned value `46`.

The for-loop

```
for ( size_t i = 0; i < aWrapper.size(); i++ )
{
    std::cout << lOrderedMapper( i );;
}

std::cout << std::endl;
```

should produce legible output when the lambda `lOrderedMapper` is correctly implemented. The output is an "Easter Egg".

You may need to **develop a plan**, that is, analyze the problem in depth, identify the unknowns, check the C++ reference and DSP lecture notes for suitable solution scenarios. You must not write a single line of code prior finishing the problem analysis.

Sketch out a plan/solution on **paper**. There might be hidden issues.

Once we understand all the requirements and possible issues of the project, we can start building the solution.

This is a rather complex tutorial. The solutions require approx. 120-130 lines of low density C++ code. Use this tutorial to practice the idioms of C++ and the proper coding of them.