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, Basic Template Operations
Lecturer:	Dr. Markus Lumpe

```
int max(const int a, const int b)
{
    return a > b ? a : b;
}
```

```
double max(const double a, const double b)
{
    return a > b ? a : b;
}
```

```
const AClass& max(const AClass& a, const AClass& b)
{
    return a > b ? a : b;
}
```

Figure 1: The max function for int, double, and AClass.

Problem 0

Before we start, we must understand the need for templates in C++. In essence, templates help us avoid writing repetitive code that only differs in the data types used. For instance, consider a `max` function that returns the maximum of two values. For type **`int`**, we can easily implement it as follows:

```
int max(const int a, const int b)
{
    return a > b ? a : b;
}
```

The implementation utilizes the conditional operator *E1 ? E2 : E3* where expression *E1* is evaluated and contextually converted to a Boolean. If the value is **`true`** the result of the conditional operator is the value of *E2*; otherwise, the result is the value of *E3*.

What if we need the same function but with arguments of type **`double`**? We can overload the function `max` for type **`double`**:

```
double max(const double a, const double b)
{
    return a > b ? a : b;
}
```

This works, but there are other numeric types like **`char`**, **`short`**, **`unsigned int`**, **`float`**, `size_t` that can be compared via less equal, and any user-defined data type that overloads the operator for less equal (i.e., **`operator>`**). We cannot overload the `max` function for all.

Instead, we define a template function `max` that uses a type template parameter using the syntax **`template`**<**`typename`** `T`>. `T` is the type parameter. It can have any name. We write

```
template<typename T>
T max(const T a, const T b)
{
    return a > b ? a : b;
}
```

The template function `max` is a blueprint. When instantiated with a type for parameter `T`, the compiler generates a new function overload for each type the template is used with.

Care is required though when using templates. Any type may be applicable for instantiating a template, including complex used- or library-defined class types. Copying values of these types can become prohibitively expensive when using call-by-value parameter passing. To avoid copies, function parameters involving type template arguments should employ the call-by-reference parameter passing mode.

```
template<typename T>
const T& max(const T& a, const T& b)
{
    return a > b ? a : b;
}
```

Using constant references works for all types including built-in types such as **`int`**, **`float`**, and `size_t`.

Enable test `P0` in Main.cpp. Run the test and use the debugger to step through the calls of max applied to **`int`**, **`float`**, at `AStruct` arguments.

Why do we need to explicitly specify the instantiation type **`double`** in the call of the function `max<`**`double`**`>( 200.0f, 300.0 )`?

Problem 1

Classes can also be defined as templates with the **`template`** keyword and the template parameters list preceding the class declaration.

Consider template `Map`, which is a class template 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 );
};
```

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

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

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 these parameters have a default value, expression `Key{}` and `Value{}`, respectively. The expressions `Key{}` and `Value{}` denote calls to the default constructors for the type template parameters `Key` and `Value`. Consequently, any type used to instantiate `Key` and `Value` must define a default constructor. For built-in types like **`int`** or **`char`**, the compiler synthesizes the default construction and yields the value 0 as a result.
- 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 template argument is independent of the type template arguments `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 type template arguments `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
```

Enable test `P1` in Main.cpp to test template class `Map`.

Problem 2

We use the template class `Map` to define the payload type for the 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 objects 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 member variables 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 constant reference to a file name string and returns **`true`**, if `load()` succeeds. Within function `load()`, we need to create an input file 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 input data is a sequence of decimal numbers stored in a text file. The first number specifies the number of key-value pairs that follow. Every key-value pair consists of an index and a datum separated by a whitespace character. Only one key-value pair occurs per line. The name of the input file is `Data.txt`:

  ```
  1050
  738 46
  667 96
  …
  565 46
  630
  ```

  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.

- 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`.

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.

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

```
Data loaded.
Accessing data sequentially:
.`
 `
  '\/ `    \ ,.
   o  - /    d \  _   -| _ '    \;  _ :
  / .           -   |o-    \   ' -',  "  | |     )       ` ' ,8              `-  ,
-  \ '   . ,       |  .`   `  ,_      .,'
    .        ._  `    ` .
…
```

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

Problem 3

Enable `P3` and inspect the function `printSorted()` in `Main.cpp`. It outputs the data sorted.

We use a linear search that maps the loop variable `i` to the corresponding payload datum in the container `aWrapper`. This linear search in combination with the **`for`**-loop over key-value indices achieves "sorting on the fly" like *Bubble Sort*. For example, if the value of `i` is `0` then the matching value of `j` is `453`. At index `j = 453`, we find a key-value pair `{0, 32}`. The value `32` has to be printed as a character first.

The call to function `printSorted()` should produce legible output. The output is an "Easter Egg".