Seminar 07-08
Templates

1. What is a template?

- In C++ templates are much like a plan that tells the compiler how to create a function/class using any given data type. Once the function/class is used with a specific data type, the compiler will compile a function/class with that type.

2. Syntax and meaning.

- Marking a function or a class as a template is done by adding the following line before the function's/class' definition:

  `template <typename [name], ...>`

  Note: `class` can be used instead of `typename` almost always.

  Note: more than one typename can be defined.

  Example of a template function:

```
template <typename T>
const T& min(const T& firstElem, const T& secondElem) {
    return (firstElem < secondElem) ? firstElem : secondElem;
}
```

- This means that when the template function is used with a certain data type the compiler will see that and compile a new function replacing the `[name]`, or in this case the T, with the used data type. Which effectively means that this function can work with any* data type.

  * any data type that has operator < in this case.

- How can we make the `min` function accept two different types?

- Template constants.

  Example:

```
template <typename Type, size_t SIZE>
struct Array {
    Type elements[SIZE];
    ...
};
```

3. Template classes and structs.

- Writing `template <typename T>` right before a class declaration will make it a template class.
  - The implementation of the template class must **NOT** be written in a .cpp file! Instead, it should be written in the header file or alternatively in a **.ipp, .imp, .tci** or a similar file that's included at the end of the header file.
  - When writing the implementation of the class, each method must include `template <typename T>` right before its implementation.
    Each method must refer to the class as `ClassName<T>`.

Example:

-- *Pair.hpp* --

```
#pragma once

template <typename Type1, typename Type2>
class Pair {
public:
    Pair(const Type1& first, const Type2& second);
    inline const Type1& getFirst() const { return m_first; }
    inline const Type2& getSecond() const { return m_second; }
    inline void setFirst(const Type1& first) { m_first = first; }
    inline void setSecond(const Type2& second) { m_second = second; }

private:
    Type1 m_first;
    Type2 m_second;
};

#include “Pair.ipp”
```

-- *Pair.ipp* --

```
#include “Pair.hpp”

template <typename Type1, typename Type2>
Pair<Type1, Type2>::Pair(const Type1& first, const Type2& second)
    : m_first(first), m_second(second)
{}
```

-- *Usage* --

```
#include <iostream>
#include “Pair.hpp”

int main()
{
    Pair<bool, int> p1 = { true, 5 };
    std::cout << p1.getSecond() << std::endl;

    Pair<double, double> coord = { 0, 0 };
    coord.setFirst(5.3);
    std::cout << coord.getFirst() << std::endl;

    return 0;
}
```

4. Template specializations.

- We can create a specialization of a template function/class, meaning we can define a different behavior of the function/class for a specific type.
  *Note: When creating a class specialization **ALL** of the class' methods and fields must be defined for that type specialization. Partial class specialization is **NOT** allowed.*

  Example:

```
template <typename T>
void fun(T elem)
{
    cout << “The template function printing: ” << elem << endl;
}

template <>
void fun(int elem)
{
    cout << “The specialized function for int: ” << elem << endl;
}
```

  Running the following code:

```
    fun(“FMI”);
    fun(56);
    fun(56.5);
```

  Will produce the following output:

```
    The template function printing: FMI
    The specialized function for int: 56
    The template function printing: 56.5
```