Object Oriented Programming

Genericity

Templates

1. Templates - Intro

- What if there is **the same** algorithm for processing several **different types** of data? What is the approach?

Code **manually** several different functions?

```
void myswap(int& x, int& y) {
      int temp = x;
      x = y;
      y = temp;
}
void myswap(double& x, double& y) {
      double temp = x;
      x = y;
      y = temp;
}
void myswap(char& x, char& y) {
      char temp = x;
      x = y;
      y = temp;
}
```

- We need to **avoid** code **duplication** (one of the "*code smells* " - a sign for poor quality)
- Therefore we'd like to write the function code **once** replacing some of the specific type(s) with **parameters** (i.e. instead of `int` or `double` or `char` write `T`)   `// T like Type`
- We need to **avoid** code **duplication** (one of the "*code smells* " - a sign for poor quality)
- Provide `T` "*argument*" where the template is used, somewhat similar to function arguments. Just like regular function

arguments can be used to pass values to a function, template arguments allow to pass also types to a function.

- This is yet another way to achieve better *reusability* and *maintainability*.

2. Templates - Function templates

- The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

- To create a *template function* that swaps two variables independently of their type:

```
#include <iostream>

template <typename T>     // template <class T> is the same, but class would be
//a little misleading here
void myswap(T& x, T& y) {
	T temp = x;
	x = y;
	y = temp;
}

int main() {
	int x = 5, y = 6;
	double z = 8.7, t = 1.2;
	int p = 18, q = 3;

	myswap<int>(x, y);    // Pass int as template argument. The compiler
//generates a function myswap(int&, int&) for you!
	myswap<double>(z, t);    // Pass double as template argument. The
//compiler generates myswap(double&, double&)!
	myswap(p, q);    // The compiler deduces the template argument (here
//int). Thus myswap(int&, int&) is used.

	std::cout << x << ", " << y << std:: endl;
	std:: cout << z << ", " << t << std::endl;
	std:: cout << p << ", " << q << std::endl;

	return 0;
}
```

- Note that the *template argument* is only one (T) and that means it is not possible to call *myswap* with two arguments of different types, for example:

```
int x = 5;
double z = 8.7;
myswap(p, q);
```

- This would not be correct, since myswap expects two arguments of the same type.
- We can also define function templates that accept more than one *type argument,* simply by specifying more than one template argument between the angle brackets:

```
template <typename T, typename U>
```

3. Templates - Class templates

- We also have the possibility to write ***class templates,*** so that a class can have members that use template parameters as types.

```
// In a header file (i.e. MyPair.hpp)
template <typename T>    // Again, template <class T> works the same
class MyPair {
public:
	MyPair(const T& first, const T& second)
	: m_first(first), m_second(second) {}
	T& max();	// Can be defined outside the class...
private:
	T m_first;
	T m_second;
};

// Definition of a method outside the template class, but still in the header //(MyPair.hpp)!
template<typename T>
T& MyPair<T>::max() {
	return m_first > m_second ? m_first : m_second;
}
```

- Up to this moment we defined classes in separate .hpp and .cpp files, following the *separate compilation* principle. But for the *template classes* there is **no compilation of the class at all**. Instead a new class is generated from **template** every time some new type comes as *template argument* to the class. With that simple difference to the classes we wrote until now we face with the need **not to separate the implementation in .cpp file.** This need is referred only to the cases where template classes are used!

```
// How to use the template class MyPair - i.e. in main.cpp
#include <iostream>
#include "MyPair.hpp"

int main() {
      MyPair<int> intPair(5, 3);         // The compiler generates a
//class MyPair<int> for you!
      MyPair<double> doublePair(5.4, 11.9);     // The compiler
//generates a class MyPair<double> for you!
      MyPair<char> charPair('q', 'a');         // The compiler generates
//a class MyPair<char> for you!

      std::cout << intPair.max() << std::endl;
      std::cout << doublePair.max() << std::endl;
      std::cout << charPair.max() << std::endl;

      return 0;
}
```