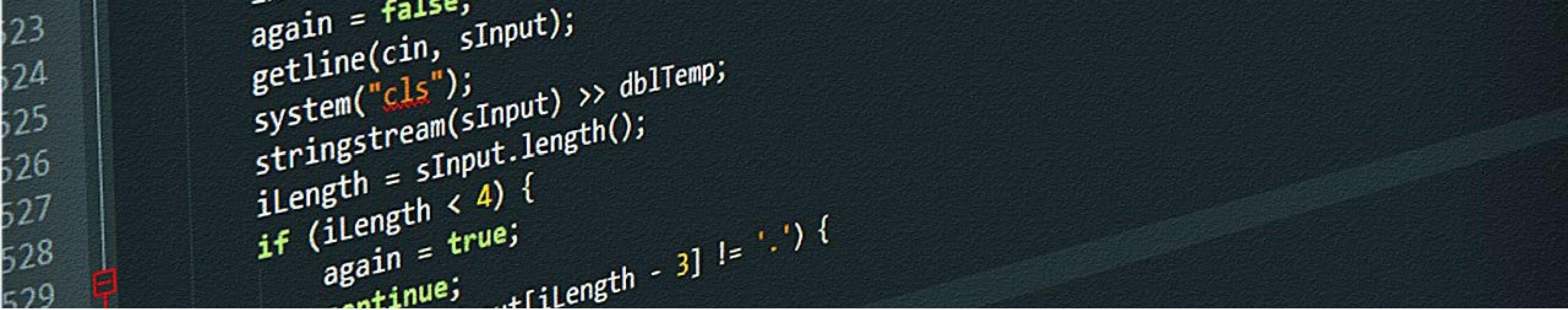

Thomas

C23-10.2 Templates

C23 - Advanced Algorithms and Programming

v5

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Slides created using material from R. Peters and P. Puschmann / HTW Berlin

Templates
Concept

- Templates are a concept of generic programming (see programming paradigms)

> ***Generic programming:*** Writing code that works with a variety of types presented as arguments, as long as those argument types meet specific syntactic and semantic requirements.

- Templates are functions or classes that allow placeholders for data types
- The placeholders for data types are used in the code,
  and will be replaced by the compiler with real types during compilation

Templates
Syntax

- Template placeholders are defined directly above a function or class definition
  These parameters can then be used in the function or class instead of data types

```
template <class T>
```

A placeholder named T

```
template <typename T>
```

Same result. `class` and typename have the same meaning here.
The programmer can choose which keyword he uses.

```
template <typename T1, typename T2, typename T3>
```

Multiple placeholders.

Templates

Function templates – example 1


```cpp
#include <iostream>

// typically, 'T' is used as placeholder for general types
template <typename T>
T min(T a, T b) // a and b and the return value are of the same type 'T'.
{
    // '<' must be defined for each template instantiation used
    // else the template will not yield the expected result
    return a < b ? a : b;
}
```

- Any type can be used for `T`
- `a` and `b` and the return type are of the same type `T`
- `<` must be defined for each type T used!

Templates

Function templates – example 1

- The compiler can derive the type automatically:

```cpp
int main()
{
    int j = 4, k = 6, i;
    float x = 1.5f, y = 3.7f, z;

    // the compiler automatically detects that int is used and generates the function
    // const int min (int a, int b) { ... }
    i = min(j, k);
    std::cout << "i = " << i << std::endl;

    // const float min (float a, float b) { ... }
    z = min(x, y);
    std::cout << "z = " << z << std::endl;
}
```

```
i = 4
z = 1.5
```

Templates

Function templates – example 2

- In some cases, the template type must be specified explicitly

```cpp
int main()
{
    int j = 4, k = 6, i;
    float x = 1.5f, y = 3.7f, z;

    // z = min(j, x); // compiler error
    // const int min (int a, float b) { ... } < not defined (both params must be of the
same type

    // explicit: const float min (float a, float b) { ... }
    z = min< float>(j, x); // j is then converted to float.
    std::cout << "z = " << z << std::endl;
}
```

```
z = 1.5
```

Templates
Function templates – example 3

- Attention with data types for which the template code does not work (e.g. because used operators are not defined)

```cpp
#include <iostream>

// typically, 'T' is used as placeholder for general types
template <typename T>
T min(T a, T b) // a and b and the return value are of the same type 'T'.
{
    // '<' must be defined for each template instantiation used
    // else the template will not yield the expected result
    return a < b ? a : b;
}

class MyClass { };

int main()
{
    MyClass m1, m2;
    MyClass m3 = min(m1, m2); // compiler error
}
```

Templates

Function templates – example 4

- Attention with data types for which the template code causes an incorrect behavior

```
#include <iostream>

// typically, 'T' is used as placeholder for general types
template <typename T>
T min(T a, T b) // a and b and the return value are of the same type 'T'.
{
    // '<' must be defined for each template instantiation used
    // else the template will not yield the expected result
    return a < b ? a : b;
}

int main()
{
    // const char* min(const char* a, const char* b)
    const char* minString = min("Hello ", "World!");

    std::cout << minString << "\n";
}
```

```
World!
```

Templates

Function templates – example 4

- Templates for special types can get their own implementation
- The compiler recognizes if there is a template specialization for a particular type and instantiates the correct function

```cpp
#include <iostream>
#include <cstring>

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

template <> // do not forget this, else this is a function overload
const char* min(const char* a, const char* b) // const char* instead of T
{
    return strcmp(a, b) < 0 ? a : b;
}

int main()
{
    // Now it works, because the string specialization is called!
    const char* minString = min("Hello ", "World!");
    std::cout << "Min String: " << minString << std::endl;
}
```

```
Min String: Hello
```

Templates
Function templates – example 5

- Template specialization vs. function overload: The example would also work with a function overload, but be careful …

```cpp
#include <iostream>

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

// no specialization, but an overload
const char* min(const char* a, const char* b)
{
    return strcmp(a, b) < 0 ? a : b;
}

int main()
{
    const char* minString1 = min("Hello ", "World!");
    const char* minString2 = min<const char*>("Hello ", "World!");

    std::cout << "Min String (Overload): " << minString1 << std::endl;
    std::cout << "Min String (Template): " << minString2 << std::endl;
}
```

If the template is used explicitly, the wrong function is used!

```
Min String (Overload): Hello
Min String (Template): World!
```

Templates

Function templates – inline example

Function templates must be implemented in the header!

Min.h

```
#pragma once
#include <iostream>

// Templates must be implemented in the header, because they are 'inline'.
template<typename T>
T min(T a, T b) {
    return a < b ? a : b;
}

// Attention, because in this template specialization no placeholders are used,
// it is no longer a template in the true sense of the word -> not automatically 'inline'
template <>
inline const char* min(const char* a, const char* b)
{
    return strcmp(a, b) < 0 ? a : b;
}
```

Templates

Class templates

- Classes can also be defined as templates
  Types are defined above the class, which are then used within the class.
- The defined placeholders can be used anywhere in the class.
- Within classes, member functions can be defined as function templates.

Templates

Class templates – example

- `T` can be used anywhere in the class

```
#pragma once

template<typename T>
class Array
{
public:

    Array(unsigned int size) : m_size(size)
    {
        m_pArray = new T[size];
    }

    ~Array() { delete[] m_pArray; }

    T& operator[](int index) { return m_pArray[index]; }

    unsigned int getSize() const { return m_size; }

private:
    T* m_pArray;
    unsigned int m_size;
};
```

Templates
Class templates – example

- Helper (template) function for printing an array

```
#include <iostream>
#include <string>
#include "Array.h"

// Function template, which expects a class template as parameter:
template<typename T>
void printArray(Array<T>& rArray)
{
    std::cout << "Size: " << rArray.getSize() << ", content: [ ";

    for (unsigned int i = 0; i < rArray.getSize(); i++)
    {
        if (i != 0) std::cout << ", ";
        std::cout << rArray[i];
    }
    std::cout << " ]" << std::endl;
}
```

Templates

Class templates – example


```cpp
int main()
{
    Array<int> numbers(3);
    numbers[0] = 1; numbers[1] = 2; numbers[2] = 5;
    std::cout << "Numbers-Array: " << std::endl;
    printArray(numbers);
    std::cout << std::endl;

    Array< std::string> strings(2);
    strings[0] = "Hello ";
    strings[1] = "World!";
    std::cout << "Strings-Array: " << std::endl;
    printArray(strings);
}
```

```
Numbers-Array:
Size: 3, content: [ 1, 2, 5 ]

Strings-Array:
Size: 2, content: [ Hello , World! ]
```

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de