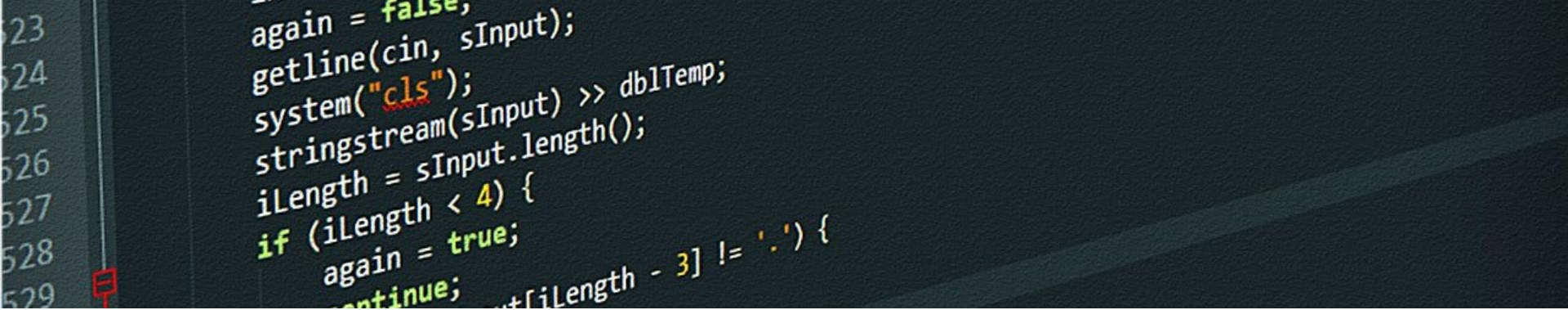

Thomas

C23-06.3 Namespaces, friends

Advanced algorithms and programming

v5

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Namespaces
Definition and access

- Namespaces are ranges of validity in the code.
  - All elements that have a name can be assigned to a namespace:
    e.g. variables, objects, classes, data types, functions, ...
  - From 'outside' their namespace, these elements can only be used by specifying the namespace

Motivation

- Prevention of name conflicts within programs or between libraries
- Rough structuring of the program code for better overview
  E.g., *GUI, DATA, TESTING ...*

Namespaces
Definition and access

Definition

- Definitions must be in the global area of the application:

```
namespace NAME { /* content */ }
```

```
namespace NamespaceA
{
    namespace NamespaceB
    {
        // nested namespaces
    }
}
```

Nesting is possible

Access to name spaces

- Using the scope operator

```
NamespaceA::ELEMENT
```

```
NamespaceA::NamespaceB::ELEMENT
```

For nested name spaces

Namespaces

Simple example

```cpp
#include <iostream>

namespace MyNameSpace
{
    // Everything defined here belongs to the namespace 'MyNameSpace'
    void printHelloWorld()
    {
        std::cout << "Hello World!" << std::endl;
    }
}

int main()
{
    MyNameSpace::printHelloWorld(); // Access via the Scope operator
}
```

Namespaces
Definition and access

- Namespaces can be reopened & extended at any time:

```
namespace NamespaceA { /* Content */ }
namespace NamespaceA { /* Other content */ }
```

- Alias names are possible:
  - Access to 'namespaceA' now also via 'NA'
  - For aliases, the same scope of validity applies as for variables!

```
namespace NA = namespaceA;
```

Namespaces
Definition and access

- Using the `using directive`
  - Elements in the namespace can then be used without a scope operator
  - For `using the` same range of validity applies as for variables

```
using namespace NamespaceA;
…
```

Namespaces

Nested namespaces – example

```
#pragma once
#include <iostream>

namespace HTW
{
    int a = 0; // global variable in namespace HTW
    namespace GUI
    {
        void printA() { std::cout << a << std::endl; }
    }
}

namespace HTW {
    void incrementA() { a += 1; }
}
```

```
// This does not work:
//      namespace HTW::GUI { void printHello() { ... } }
// But this:
namespace HTW
{
    namespace GUI
    {
        void printHello()
        {
            std::cout << "Hello Namespace!" << std::endl;
        }
    }
}

void f()
{
    using namespace HTW; // 'HTW::' can be omitted from here ...
    incrementA();
    GUI::printA();
}

void g()
{
    namespace CE = HTW; // CE is now alias name for HTW
    CE::incrementA();
    CE::GUI::printA();
}
```

Namespaces

Nested namespaces – example

```
int main()
{
    HTW::GUI::printHello();
    // printA();    -> DOES NOT GO (because other namespace)

    std::cout << HTW::a << std::endl;
    HTW::incrementA();
    HTW::GUI::printA(); // this is how it works!

    f();
    g();

    using namespace HTW;
    using namespace GUI;
    // using namespace HTW::GUI;    -> works too!

    // now direct access is possible:
    incrementA();
    printA();
}
```

```
Hello Namespace!
0
1
2
3
4
```

Namespaces

Standard library namespace – example

The C++ standard library uses the namespace `std`

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string text = "Hello World";

    cout << text << endl;
}
```

```
Hello World
```

Classes and friends

Friend functions

- Classes can allow certain functions direct access to their `private` or `protected` area, even if they are not member functions of the class

- This is actually contrary to object orientation, but helpful for cross-object tasks:
  - Functions that work directly with several different objects
  - Operators

Syntax

```
friend FUNCTIONS_PROTOTYPE;
```

- It does not matter in which access area (`public`, `private`) the `friend` classifier is used.

Classes and friends

Friend functions – example

Fraction.h

```
#pragma once

class Fraction // math. fraction
{
public:
Fraction(int numerator, int denominator)
: m_numerator(numerator), m_denominator(denominator) {}

private:
int m_numerator;
int m_denominator;

// It does not matter if it is under private or public:
friend double compare(const Fraction& rFrac1,
    const Fraction& rFrac2);
};
```

Classes and friends

Friend functions – example

Main.cpp

```cpp
#include < iostream>
#include "fraction.h"

double compare(const Fraction& rFrac1, const Fraction& rFrac2)
{
    // Access to member variables, although it is not a member function!
    double f1 = rFrac1.m_numerator / (double)rFrac1.m_denominator;
    double f2 = rFrac2.m_numerator / (double)rFrac2.m_denominator;
    return f1 - f2;
}

int main()
{
    Fraction frac1(10, 12);
    Fraction frac2(11, 12);

    if (compare(frac1, frac2) == 0)
        std::cout << "Both fractions are equal in size!" << std::endl;
    else if (compare(frac1, frac2) > 0)
        std::cout << "Fraction 1 is larger!" << std::endl;
    else
        std::cout << "Fraction 1 is smaller!" << std::endl;
}
```

```
Fraction 1 is smaller!
```

Classes and friends

Friend classes

- A class can also allow other classes access to its protected data and functions

Syntax

```
friend class class_name;
```

```
class Disk
{
    friend class Computer; // carte blanche for computers
    ...
};

class Computer
{
    Disk disk1; // Computer has direct access to all member variables of Disk
    ...
};
```

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de