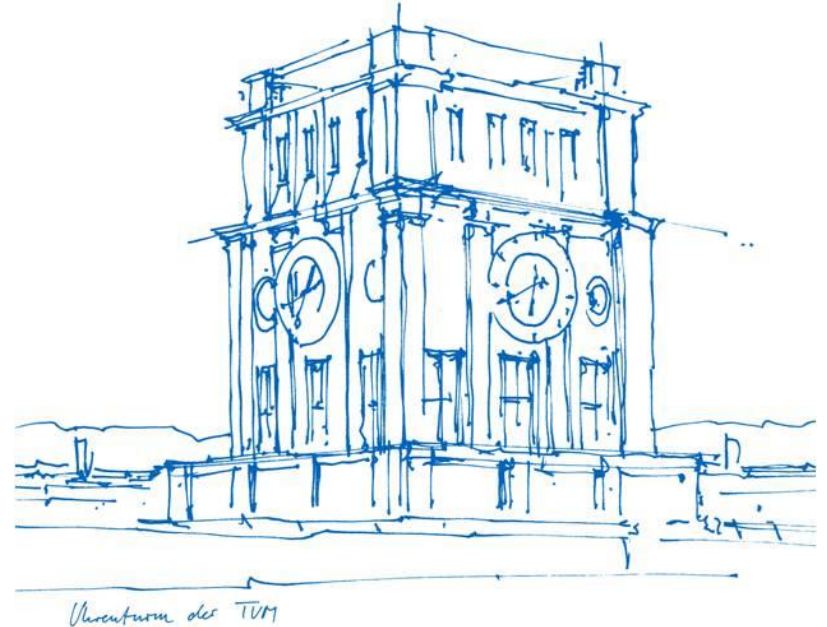


# Grundlagenpraktikum: Rechnerarchitektur

SoSe 2024

~ *Danial Arbabi*

[danial.arbabi@tum.de](mailto:danial.arbabi@tum.de)



# Zulip-Gruppen

Gruppe 29

FR 12:00



<https://zulip.in.tum.de/#narrow/stream/2276-GRA-Tutorium---Gruppe-29>

Gruppe 32

FR 15:00



<https://zulip.in.tum.de/#narrow/stream/2279-GRA-Tutorium---Gruppe-32>

# Tutoriums-Website



<https://home.in.tum.de/~arb/>

Disclaimer:

*Dies sind keine offiziellen  
Materialien, somit besteht keine  
Garantie auf Korrektheit und  
Vollständigkeit.*

*Falls euch Fehler auffallen, bitte  
gerne melden.*

## C++ vs. C vs. Java

	C++	C	Java
Kompiliert	✓	✓	✓
Objektorientiert	✓	×	✓
Direkter Zugriff auf Hardware	✓	✓	×
Garbage Collected	×	×	✓

# Simples C++ Programm

- `#include <iostream>` für Standardbibliothek (Ein- und Ausgabe)
- `std` ist der Namespace für die Standardbibliothek
- `::` ist der scope resolution-Operator
- `std::cout` bezieht sich auf den Namen `cout` im Namespace `std`
- `<<` ist der Ausgabeoperator

```
1  #include <iostream>
2
3  int main (int argc , char ** argv ) {
4      std::cout << "Hello , World !" << std::endl;
5      // "Hello , World !\n"
6      std::cout << " Answer : " << 42 << "\n";
7      // " Answer : 42\ n"
8      return 0;
9  }
```

# Objektorientierung

- Klasse: Standardmäßig Private

## C++

```
1 class Shape {  
2     // default private  
3     float area;  
4  
5     public:  
6         Shape(float a): area(a)  
7         {}  
8  
9         float getArea() {  
10             return area;  
11         }  
12 }; // <- Semicolon notwendig!
```

## Java

```
1 public class Shape {  
2     private float area;  
3  
4     public Shape(float a) {  
5         area = a;  
6     }  
7  
8     public float getArea() {  
9         return area;  
10    }  
11 }
```

# Klassendeklaration

```
1 class Shape {  
2     float area; // area ist private  
3  
4     float getArea() { return area; }  
5 };  
6  
7 struct Shape {  
8     float area; // area ist public  
9  
10    float getArea() { return area; }  
11 };
```

# Konstruktoren

```
1 class Shape {  
2     public:  
3         // Konstruktor  
4         Shape(float a) {}  
5         // Default-Konstruktor  
6         Shape() {}  
7         // Kein Konstruktor  
8         void Shape() {}  
9  
10 };
```



# Konstruktor

```
class Person {  
private:  
    string name;  
    int age;  
  
public:  
    // Konstruktor  
    Person(string n, int a) {  
        name = n;  
        age = a;  
    }  
  
    // Getter-Funktionen  
    string getName() const {  
        return name;  
    }  
  
    int getAge() const {  
        return age;  
    }  
};
```

```
struct Shape {  
    int width;  
    int height;  
    int area;  
  
    // Konstruktor mit Parametern  
    Shape(int w, int h) : width(w), height(h), area(w * h) {}  
  
    // Standardkonstruktor ohne Parameter  
    Shape() : width(1), height(1), area(1) {}  
};
```

Shape shape(10, 20); // auf dem Stack

Shape\* shape = new Shape(10, 20); // auf dem Heap

object.member für konkrete Objekte

object->member für Pointer

## Aufrufen von Feldern und Methoden

```
1 ...  
2  
3 Shape shape1(10, 20);  
4 std::cout << shape1.area << std::endl;  
5 // shape1 ist kein Pointer und benötigt den '.' Operator.  
6  
7 Shape* shape2 = new Shape(10, 20);  
8 std::cout << shape2->area << std::endl;  
9 // shape2 ist ein Pointer und benötigt den '->' Operator.
```

Dasselbe gilt auch für Methoden

# Erstellen einer Liste aus ints/strings

```
1 #include<vector>
2 ...
3
4 std::vector<int> vec;
5 // Eine Liste aus ints.
6 vec.push_back(10);
7 vec.push_back(25);
8 // Integers zur Liste hinzufügen.
9 int myInt = vec.at(0);
10 // Auf Integers aus der Liste zugreifen.
```

```
1 #include<vector>
2 ...
3
4 std::vector<const char*> vec;
5 // Eine Liste aus String-Literals.
6 vec.push_back("Hello");
7 vec.push_back("World");
8 // String Literal zur Liste hinzufügen.
9 const char* myString = vec[0];
10 // Auf String Literal aus der Liste zugreifen.
```

# SystemC

- `#include <systemc>` (Hauptfunktionalitäten)
- `#include "systemc.h"` (Nebenfunktionalitäten)
- `using namespace sc_core;` Einfacherer Zugriff auf SystemC Konzepte

# SystemC

## Main-Funktion

- Main-Funktion für SystemC: `int sc_main (int argc, char** argv)`

```
1 #include <iostream>
2 #include <systemc>
3 using namespace sc_core;
4
5 int sc_main(int argc, char** argv) {
6     std::cout << "Hello World!" << std::endl;
7 }
```

# SystemC

## sc\_signal<T>

- Stellt „Signal“ oder „Kabel“ dar
- Generischer Typ T ermöglicht anlegen beliebiger Datentypen

```
1 ...  
2  
3 sc_signal<bool> boolSignal;  
4 boolSignal = true;  
5 // Signal, das TRUE/FALSE, 1/0 überträgt.  
6  
7 sc_signal<int> intSignal;  
8 intSignal = 42;  
9 // Signal, das Integerwerte überträgt.
```

# SystemC

## Lesen / Schreiben von Signalen

- Zwei Möglichkeiten
  - write(), read()
  - Lesen und Schreiben mit Zuweisung

```
1  sc_signal<bool> mySignal;  
2  mySignal = true;  
3  //schreiben mit Zuweisung.  
4  
5  mySignal.write(true);  
6  //schreiben mit sc_signal.write(value).  
7  
8  bool myBool = mySignal;  
9  //lesen mit Zuweisung.  
10  
11 myBool = mySignal.read();  
12 //lesen mit sc_signal.read();
```

# SystemC

## Ausführen eines SystemC Programms

- `sc_start()` für starten (bzw. `sc_pause()` für Pause und `sc_stop()` für stoppen)

```
int sc_main(int argc, char* argv[]) {  
    sc_signal<bool> mySignal ;  
  
    mySignal = true;  
    // Oder  
    mySignal.write(true);  
  
    // Starten der Simulation  
    sc_start();  
  
    std::cout << "Signal " << mySignal << std::endl;  
    // Oder  
    std::cout << "Signal " << mySignal.read() << std::endl;  
  
    return 0;  
}
```

Output:

Signal 1  
Signal 1