

## **9. C++ vertieft (II): Templates**

# Motivation

Ziel: generische Vektor-Klasse und Funktionalität.

## Beispiele

```
vector<double> vd(10);
```

```
vector<int> vi(10);
```

```
vector<char> vi(20);
```

```
auto nd = vd * vd; // norm (vector of double)
```

```
auto ni = vi * vi; // norm (vector of int)
```

# Typen als Template Parameter

- 1 Ersetze in der konkreten Implementation einer Klasse den Typ, der generisch werden soll (beim Vektor: `double`) durch einen Stellvertreter, z.B. `T`.
- 2 Stelle der Klasse das Konstrukt `template<typename T>`<sup>9</sup> voran (ersetze `T` ggfs. durch den Stellvertreter)..

Das Konstrukt `template<typename T>` kann gelesen werden als “für alle Typen `T`”.

---

<sup>9</sup>gleichbedeutend: `template<class T>`

# Typen als Template Parameter

```
template <typename ElementType>
class vector{
    size_t size;
    ElementType* elem;
public:
    ...
    vector(size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[](size_t pos){
        return elem[pos];
    }
    ...
}
```

# Template Instanziierung

`vector<typeName>` erzeugt Typinstanz von `vector` mit `ElementType=typeName`.

Bezeichnung: **Instanziierung**.

## Beispiele

```
vector<double> x;           // vector of double
vector<int> y;             // vector of int
vector<vector<double>> x;  // vector of vector of double
```

# Type-checking

Templates sind weitgehend Ersetzungsregeln zur Instanzierungszeit und während der Kompilation. Es wird immer so wenig geprüft wie nötig und so viel wie möglich.

# Beispiel

```
template <typename T>
class vector{
...
    // pre: vector contains at least one element, elements comparable
    // post: return minimum of contained elements
    T min() const{
        auto min = elem[0];
        for (auto x=elem+1; x<elem+size; ++x){
            if (*x<min) min = *x;
        }
        return min;
    }
...
}
```

```
vector<int> a(10); // ok
auto m = a.min(); // ok
vector<vector<int>> b(10); // ok;
auto n = b.min(); no match for operator< !
```

# Generische Programmierung

Generische Komponenten sollten eher als **Generalisierung eines oder mehrerer Beispiele** entwickelt werden als durch Ableitung von Grundprinzipien.

```
using size_t=std::size_t;
template <typename T>
class vector{
public:
    vector ();
    vector(size_t s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    vector (vector&& v);
    vector& operator=(vector&& v);
    T operator[] (size_t pos) const;
    T& operator[] (size_t pos);
    int length() const;
    T* begin();
    T* end();
    const T* begin() const;
    const T* end() const;
}
```



# Funktionentemplates

- 1 Ersetze in der konkreten Implementation einer Funktion den Typ, der generisch werden soll durch einen Stellvertreter, z.B. `T`,
- 2 Stelle der Funktion das Konstrukt `template<typename T>`<sup>10</sup> voran (ersetze `T` ggfs. durch den Stellvertreter).

---

<sup>10</sup>gleichbedeutend: `template<class T>`

# Funktionentemplates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Typen der Aufrufparameter determinieren die Version der Funktion, welche (kompiliert und) verwendet wird:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```

# Grenzen der Magie

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Eine unverträgliche Version der Funktion wird nicht erzeugt:

```
int x=5;
double y=6;
swap(x,y); // error: no matching function for ...
```

# Praktisch!

```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (auto x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

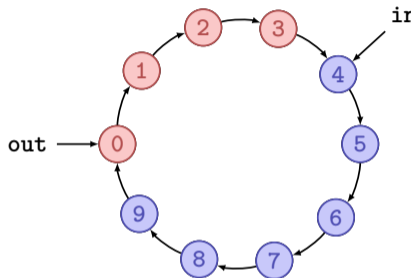
int main(){
    std::vector<int> v={1,2,3};
    output(v); // 1 2 3
}
```

# Mächtig!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x ← f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
    std::vector<int> v={1,2,3};
    apply(v,sq<int>);
    output(v); // 1 4 9
}
```

# Templateparametrisierung mit Werten

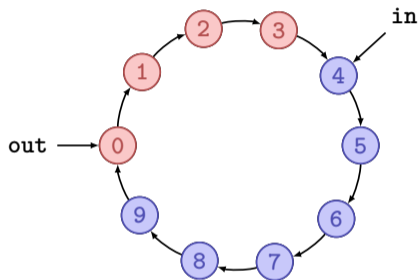
```
template <typename T, int size>
class CircularBuffer{
    T buf[size] ;
    int in; int out;
public:
    CircularBuffer():in{0},out{0}{};
    bool empty(){
        return in == out;
    }
    bool full(){
        return (in + 1) % size == out;
    }
    void put(T x); // declaration
    T get();      // declaration
};
```



# Templateparametrisierung mit Werten

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```

```
template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```



← Optimierungspotential, wenn  $size = 2^k$ .