

X Generische Programmierung

x.1 Klassen-Templates

X.2 Funktions-Templates (*)

X.3 Besondere Programmier-Techniken (*)
(Smart Pointer)

(*) Auch in dieser Datei.

Lehrbuch: 5.1

Kompendium, 3. Auflage: 8.10, 9.7

Kompendium, 4. Auflage: 10.1

Know-how-Abfrage

Ihre Selbsteinschätzung

1. Habe schon Templates in C++ selber erstellt.
2. Habe schon Generics/Templates in Java selber erstellt
3. Habe schon Templates in C++ genutzt (z.B. Klasse `vector<T>`)
4. Keine Ahnung von Templates

Generik ohne Templates (1)

```
typedef int T;
```

Funktionen

```
void swap(T& x, T& y) {  
    T temp = x;  
    x = y; y = temp;  
}
```

```
int main() {  
    T a = 123, b = 456;  
    swap(a, b);  
    cout << a << " " << b << endl;  
}
```

```
typedef int T;
```

Klassen

```
class X {  
    T value;  
public:  
    X(T init) { value = init; }  
    void Print() { cout << value; }  
};
```

```
int main() {  
    X x(27);  
    x.Print();  
}
```

Nachteil: Die Funktion swap und die Klasse X können jeweils nur für einen Typ verwendet werden!

Funktions-Templates

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

int main() {
    int a = 123, b = 456;
    double x1 = 1.23, x2 = 4.56;
    swap(a, b);
    swap(x1, x2);
    cout << a << " " << b << endl;
    cout << x1 << " " << x2 << endl;
}
```

Dieses Beispiel zeigt das Prinzip: Es wird eine Schablone (template) definiert, und T ist ein Typparameter.

Durch die Angabe $swap(a,b)$; wird der Compiler angewiesen, aus der Schablone heraus eine Funktion $swap$ für den Typ *int* zu generieren.

Klassen-Templates, Grundlagen (1)

```
template <typename T>
class Stack {
    T Data[100];
    int n;
public:
    ...
    void Push(const T& t) {
        Data[n++] = t;
    }
    ...
};

...
double x;
Stack<double> s;
s.Push(x);
...
```

Dieses Beispiel zeigt das Prinzip:
Es wird eine Schablone (template) definiert, und T ist ein Typparameter.

Durch die Angabe *Stack<float> s;* wird der Compiler angewiesen, aus der Schablone heraus eine Klasse *Stack* für den Elementetyp *float* zu generieren.

Klassen-Templates, Grundlagen (2)

```
template <typename T>
class Stack {
    T Data[100];
    int n;
public:
    ...
    void Push(const T& t);
    ...
};

...
template<typename T>
void Stack<T> ::Push(const T& t) {
    Data[n++] = t;
}
...
```

Hier wird gezeigt, wie die Definitionen der Methoden für das Klassen-Template aussehen, die dann in der Regel in der entsprechenden cpp-Datei stehen.

Klassen-Templates, Grundlagen (3)

```
template <typename T, int Size>
class Stack {
    T Data[Size];
    int n;
public:
    ...
};

...
Stack<double, 100> s;
...
```

Neben Typen, wie hier T , können auch *normale Werte* wie hier $Size$ als Parameter verwendet werden.

Klassen-Templates, Instanziierung

```
#ifndef _Stack_h
#define _Stack_h

template <typename T>
class Stack {
    ...
};

#include "Stack.cpp"

#endif
```

H-Datei

```
Stack<int> s1;
Stack<double> s2;
Stack<Somewhat> s3;
Stack<Stack<Stack<X> > > s4;
...
typedef Stack<int> intStack;
...
typedef Stack<X> XStack;
typedef
    Stack<XStack> StackOfStack;
```

Anwendung

Häufig gibt es Probleme bei der Instanziierung (Codegenerierung).
Empfehlung für kleine Projekte: cpp-Datei per „include“ in h-Datei einfügen,
die cpp-Datei dann aber nicht mehr dem Projekt hinzufügen.

Die Anwendung demonstriert einige Möglichkeiten, Stack-Objekte zu erzeugen.

Gemeinsame Übung für Vektor

Problem bei globalen Operatoren als friend

```
template <typename T>
class Vektor{
public:
    Vektor(int dim);
    ...
    friend const Vektor<T>& operator+(const
        Vektor<T>& v1, const Vektor<T>&
        v2);
    friend inline ostream&
        operator<<(ostream& str, const
            Vektor<T>& v);
};
```

Geht nicht.

```
template <typename T>
class Vektor{
public:
    Vektor(int dim);
    ...
    template <typename U>
    friend Vektor<U> operator+(
        const Vektor<U> & v1, const Vektor<U>& v2);

    template <typename V>
    friend inline ostream& operator<<(ostream & str,
        const Vektor<V>& v);
};
```

Geht.

Problem bei globalen Operatoren als friend

```
template <typename T>
class Vektor{
public:
    Vektor(int dim);
    ...
    friend const Vektor<T>& operator+(const
        Vektor<T>& v1, const Vektor<T>&
        v2);
};

const Vektor<T>& operator+(const
    Vektor<T>& v1, const
    Vektor<T>& v2);
```

Geht nicht.

```
template <typename T>
class Vektor{
public:
    Vektor(int dim);
    ...
    template <typename U>
    friend Vektor<U> operator+(
        const Vektor<U> & v1, const Vektor<U>& v2);

    template <typename V>
    friend inline ostream& operator<<(ostream & str,
        const Vektor<V>& v);
};

template <typename U>
Vektor<U> operator+(
    const Vektor<U> & v1, const Vektor<U>& v2);
```

Geht.

Problem bei globalen Operatoren als friend

```
template <typename T>
class Vektor{
public:
    Vektor(int dim);
    ...
};
```

```
template <typename U>
Vektor<U> operator+(
const Vektor<U> & v1, const Vektor<U>& v2);

template <typename V>
friend inline ostream& operator<<(ostream & str, const Vektor<V>& v);
```