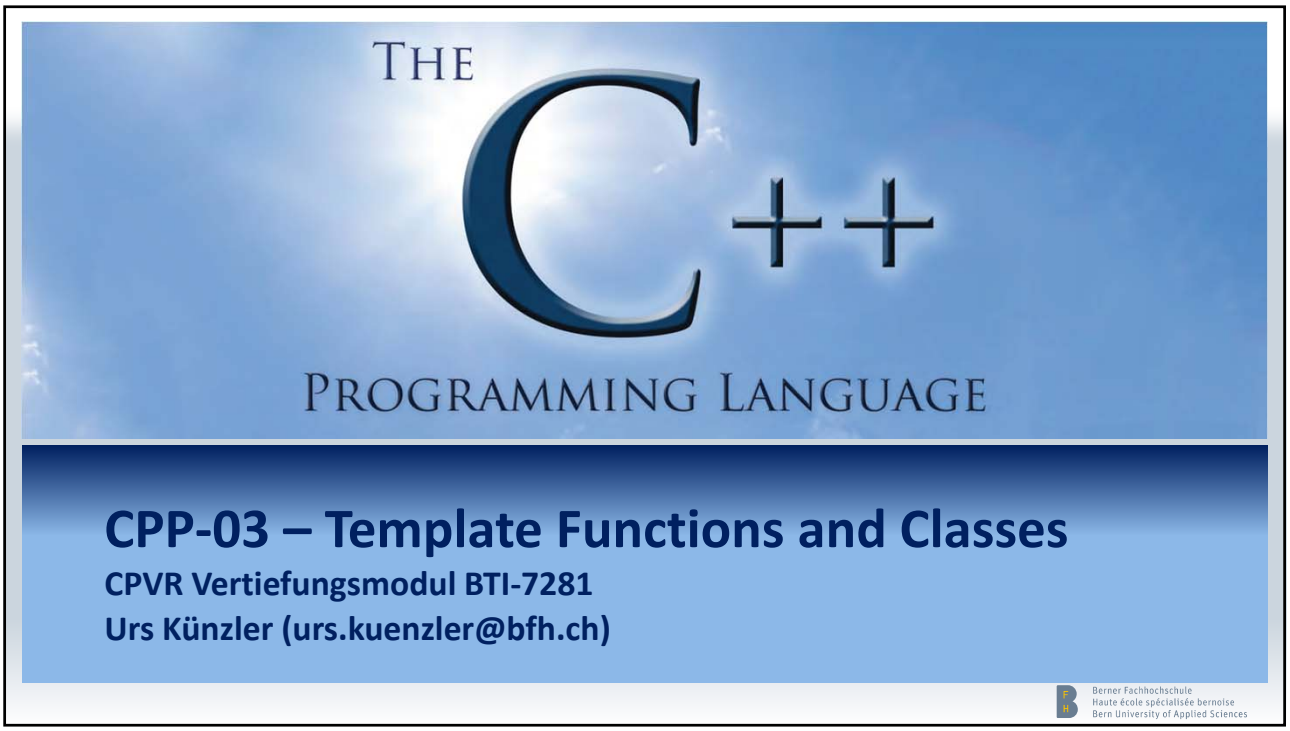

CPP-03 – Template Functions and Classes

CPVR Vertiefungsmodul BTI-7281

Urs Künzler (urs.kuenzler@bfh.ch)

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CPP-03 Table of Contents – Course Introduction

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

03.01 Templates Overview

- **Templates Übersicht**
  - C++ - Templates (Vorlage, Schablone) ermöglichen die generische (allgemeingültige) Programmierung von Funktionen und Klassen die mit einem oder mehreren Datentypen parametrisiert werden können.
  - Der typenspezifische Code für Funktionen und Klassen wird vom Compiler für konkret definierte Datentypen automatisch, entsprechend der Template-Deklaration, generiert (instanziert).
  - Templates werden hauptsächlich für die Realisierung von typen-sicheren, allgemeinen Hilfsfunktionen (z.B. min, max) und für Container-Klassen (z.B. Stacks, verkettete Listen, etc.) verwendet.
  - Richtig eingesetzt, sind Templates zeitsparend und fehler-reduzierend weil die Implementation von redundantem Code vermieden werden kann.

03.02 Template Functions

- **Die Deklaration von Template Funktionen und Klassen erfolgt mit dem Keyword `template` sowie einer Typenparameterliste.**
- **Beispiel - Generische Max() Funktion:**

```
// Deklaration der Template-Funktion Max()
template<class T> T Max(T, T); // äquivalent: 'typename' für 'class'

// Implementation der Template-Funktion Max()
template<class T> T Max (T a, T b)
{
   return ( a > b ? a : b );
}

// Verwendung der Template-Funktion Max()
int i1=10, i2=20, i3=0;
double d1=10.1, d2=20.2, d3=0.0;
i3 = Max(i1,i2); // generiert: int Max(int,int);
d3 = Max(d1,d2); // generiert: double Max(double, double);
```

Übergebener Datentyp muss entsprechende Operatoren implementiert haben

Template Funktionen II

- **Regeln für die Implementation von Template-Funktionen:**
  - Beim Aufruf einer Template-Funktionen erfolgt keine implizite Typenumwandlung der Funktionsparameter durch den Compiler.
  - Template-Funktionen können inline implementiert werden.
  - Die Implementation einer Template-Funktion kann intern beliebige lokale Variablen des Template-Datentyps besitzen. Lokale Variablen können zudem statisch deklariert werden, wobei für jeden Datentyp für den die Funktion generiert wird, eine separate statische Variable definiert wird.
  - Eine Template-Funktion kann durch mehrere Datentypen parametrisiert werden:

```
template <class T1, class T2>
void func(T1 a, T2 b, int size) {
      cout << "Argument 1: " << a << endl;
      cout << "Argument 2: " << b << endl;
      cout << "Argument 3: " << size << endl;
}
```


Template Funktionen III

- Default-Argumente für Typ-Parameter sind bei Template-Funktionen nicht zulässig, sondern sind nur für Template-Klassen zulässig.
- Jeder Datentyp der Template-Parameterliste muss *Funktionsparameter* sein oder der Aufruf der Template-Funktion muss explizit qualifiziert werden:

```
// Deklaration
template<class X, class Y> X func(Y y) {...};
...
double d = 123.456;
int i = func<int,double>(d); // expliziter Aufruf bzw. Instanzierung
```

Template Funktionen IV

- Durch Überladen einer Template-Funktion kann die Funktion für beliebige Kombinationen von Typenparametern verwendet oder die Implementation für spezifische Datentypen optimiert werden.
- Beispiel - Standard Max() Template-Funktion für C-Strings:

```
char *s1 = "AAAA", *s2 = "BBBB";
char* maxstr = Max(s1, s2); // Vergleicht nur String-Adressen!
```

- Beispiel - Überladen der Max() Template-Funktion für C-Strings:

```
char* Max(char* a, char* b) {
   return ((strcmp(a, b) > 0) ? a : b);
}
```

CPP-03 | 7 | Demo: CPP-03-D.01 - Template Functions

03.03 Template Classes

- Eine Template-Klasse ist eine generische Klasse, die mit einem oder mehreren Datentypen parametrisiert ist.
- Ein Objekt einer Template-Klasse wird vom Compiler erzeugt (instanziert), indem die Typenparameter durch die in der Definition spezifizierten Datentypen ersetzt werden.
  - Beispiel: `CStack<int> IntStack;  // generiert Stack vom Typ int`
- Damit der Compiler die typenspezifischen Klassen instanzieren kann, muss der Template-Code im Header-File implementiert werden (Widerspruch zum Konzept der Kapselung).
- Durch Verwendung von Templates ergibt sich ein Performance-Vorteil durch die Compile-Time Auflösung von typspezifischem Code.

CPP-03 | 8

Template Klassen II

- **Regeln für die Implementation von Template-Klassen:**
  - Klassen können grundsätzlich als Template (allgemeingültig) implementiert werden.
  - Für jeden unterschiedlichen Typparameter eines Templates wird durch den Compiler implizit Code generiert.
  - Innerhalb einer Template-Klasse kann der Datentyp des Template-Parameters wie jeder andere Datentyp zur Deklaration von Klassen-Membern verwendet werden.
  - Bei der Instanzierung einer Template-Klasse müssen die verwendeten Datentypen explizit deklariert werden.
  - Template-Klassen können in Vererbungen verwendet werden.
  - In Template-Klassen ist das Überladen von Member-Funktionen erlaubt.
  - Inline-Implementationen von Member-Funktionen sind zulässig.
  - Template-Klassen können verschachtelt werden, d.h. eine Template-Klasse kann als Typenparameter einer anderen Template-Klasse verwendet werden.
  - Statische Variablen können verwendet werden, wobei separate statische Member für jede instanzierte Klasse angelegt werden.

Template Klassen - Beispiel

- **Beispiel - Deklaration einer generischen Stack-Klasse:**

```
template <class T>
class CStack {
private:
       int _iSize;
       int _iCount;
       T*  _Elements;
public:
       ~CStack();                                  // destructor
       CStack(int size = 10);                      // default constructor
       CStack(const CStack&);                      // copy constructor
       CStack& operator=(const CStack&);           // assignment operator

       void Push(T);                               // push element on stack
       T Pop(void);                                // pop element from stack
};
```

Template Klassen - Beispiel II

- **Beispiel - Implementation einer generischen Stack-Klasse:**

```
// stack constructor
template <class T> CStack<T>::CStack(int size)
:_iSize(size), _iCount(0), _Elements(new T[_iSize]){}

// stack destruktor
template <class T> CStack<T>::~CStack() {
      delete[] _Elements;
}

// push stack element
template <class T> void CStack<T>::Push(T x) {
      _Elements[_iCount++] = x;
}

// pop stack element
template <class T> T CStack<T>::Pop(void) {
      return _Elements[--_iCount];
}
```


Template Klassen - Beispiel III

- **Beispiel - Verwendung einer generischen Stack-Klasse:**

```
// Statische Definition
CStack<int> IntStack(20);                  // stack of integers
CStack<double> DoubleStack;                // stack of doubles
CStack<CPoint> PointStack;                 // stack of points

// Dynamische Definition
CStack<int>* pIntStack = new CStack<int>;  // pointer to int stack

// Typedef Definition
typedef CStack<int> IStack;
IStack aIntStack[10];                      // array of int stacks

// Verschachtelte Definition
CStack< CStack<int> > IntStackStack;       // stack of int stacks
CStack< CStack<CPoint> > PointStackStack;  // stack of point stacks
```