

Kurseinheit 8: Templates

1. Motivation: Generische Programmierung
2. Klassentemplates
3. Beispiel Stackklasse

1. Motivation: Generische Programmierung

BubbleSort aus C-Vorlesung

```

void DoBubbleSortWithEquivalencePointerArray(int* paiA,
                                              unsigned int uiSize)
{
    unsigned uiY;
    unsigned uiX;
    int iDummy;

    for (uiY = 0U; uiY < (uiSize - 1U); uiY++)
    {
        for (uiX = 0U; uiX < (uiSize - 1U - uiY); uiX++)
        {
            if (paiA[uiX] > paiA[uiX + 1])
            {
                iDummy = paiA[uiX];
                paiA[uiX] = paiA[uiX + 1];
                paiA[uiX + 1] = iDummy;
            }
        }
    }
}

```

Algorithmus ist datentypabhängig.
Nicht generisch!

1. Motivation: Generische Programmierung

3

QuickSort aus stdlib

```
/* Always compile this module for speed, not size */  
#pragma optimize("t", on)
```

```
#if defined (_M_CEE)  
#define __fileDECL __cdecl  
#else /* defined (_M_CEE) */  
#define __fileDECL __stdcall  
#endif /* defined (_M_CEE) */
```

qsort.c
Microsoft
Implementation
__cdecl und __stdcall
sind Calling
Conventions.
Wer räumt den Stack
wieder auf?

```
void __fileDECL qsort (  
    void *base, ← Generischer Zeiger auf Arrayanfang  
    size_t num, ← Anzahl Elemente  
    size_t width, ← Größe eines Elements  
    int (__fileDECL *comp)(const void *, const void *)  
)  
    ↑  
    Funktionszeiger auf Vergleichsfunktion: Gibt -1, 0 oder 1 zurück
```

1. Motivation: Generische Programmierung

4

QuickSort aus stdlib

```
char *lo, *hi;          /* ends of sub-array currently sorting */  
char *mid;              /* points to middle of subarray */
```

lo, hi und mid sind char-Pointer, die innerhalb von *base auf die Elemente zeigen.

```
lo = (char *)base;  
hi = (char *)base + width * (num-1);    /* initialize limits */
```

```
mid = lo + (size / 2) * width;    /* find middle element */
```

Der rekursive Sortieralgorithmus QuickSort partitioniert das Array anhand des Pivot-Elements. In dieser Implementierung wird das mittlere Element verwendet.

```
if (__COMPARE(context, lo, mid) > 0)  
{  
    swap(lo, mid, width);  
}
```

__COMPARE ist Makro für Funktionszeiger comp.

Der Algorithmus enthält drei obige Vergleiche.

Ohne Coding Styleguide recht schlecht lesbar!

1. Motivation: Generische Programmierung

5

QuickSort aus stdlib

```
static void __fileDECL swap (
    char *a,
    char *b,
    size_t width
)
{
    char tmp;

    if ( a != b )
        /* Do the swap one character at a time to avoid potential alignment
           problems. */
        while ( width-- ) {
            tmp = *a;
            *a++ = *b;
            *b++ = tmp;
        }
}
```

Die Umsetzung generischer Funktionalität ist in C sehr komplex und fehleranfällig.

2. Klassentemplates

6

Allgemeines zu Templates

Templates (englisch für **Schablonen** oder **Vorlagen**) sind ein Mittel zur Typparametrierung in C++. Templates ermöglichen generische Programmierung und typsichere Container. In der C++-Standardbibliothek werden Templates zur Bereitstellung typsicherer Container, wie z. B. Listen, und zur Implementierung von generischen Algorithmen, wie z. B. Sortierverfahren, verwendet.
www.wikipedia.de

Templates werden auch als **Polymorphismus zur Compilezeit** beschrieben.

Prinzipielle Idee:

Programmcode für unterschiedliche Datentypen wird nur einmal geschrieben (Sourcecode), der Compiler generiert bei Bedarf dann mehrfachen Maschinencode.
Unterstützt das **DRY**-Prinzip: Don't Repeat Yourself!

Arten von Templates in C++:

- Funktionstemplates (Kurseinheit 1)
- Klassentemplates

2. Klassentemplates

7

Einfaches Klassentemplate (ohne Exceptions)

```
template <typename T, uint32_t u32S>
class SimpleArray
{
public:
    SimpleArray() //Constructor
    {
        pElements_ = new T[u32Size_ = u32S];
    }
    ~SimpleArray() //Destructor
    {
        delete[] pElements_;
    }
    T& operator[](int32_t s32Index)
    {
        return (pElements_[s32Index]);
    }
private:
    T* pElements_;
    uint32_t u32Size_;
};
```

Statt dem Schlüsselwort *typename* kann auch *class* verwendet werden.

Bei **T** handelt es sich um einen sogenannten **Typparameter**.

Weitere Typparameter sind möglich (üblicherweise nimmt man dann U, V, W, Z).

Alle Features einer Klasse wie:

- Operatorüberladung
- Exception
- Move/Copy-Semantik
- ...

können in einem Template verwendet werden.

2. Klassentemplates

8

Einfaches Klassentemplate – Erweiterung <<-Operator

```
template<class T, uint32_t u32S> // Forward declaration template class
class SimpleArray;
template <class T, uint32_t u32S> // Declaration friend template Methode
std::ostream& operator<<(std::ostream& ostr, const SimpleArray<T, u32S>& crT);

template <typename T, uint32_t u32S>
class SimpleArray
{
//more Code
friend std::ostream& operator<<<<T, u32S>(std::ostream& rostr,
    ↑
    const SimpleArray<T, u32S>& crT);
};

template <class T, uint32_t u32S>
std::ostream& operator<<<<T, u32S>(std::ostream& ostr, const SimpleArray<T, u32S>&
crT)
{
    for (uint32_t u32I = 0; u32I < crT.u32Size_; u32I++)
    {
        ostr << crT.pElements_[u32I] << " " << std::endl;
    }
    return ostr;
}
```

Templateparameter sind hier notwendig

2. Klassentemplates

9

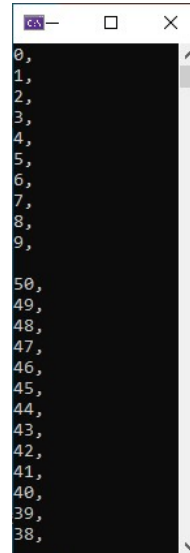
Einfaches Klassentemplate – Anwendung

```
int main(void)
{
    SimpleArray<int32_t, 10U> s32TempArray;
    SimpleArray<f64_t, 50U> f64TempArray;

    for (uint32_t u32I = 0U; u32I < 10U; u32I++)
    {
        s32TempArray[u32I] = (int32_t)u32I;
    }
    std::cout << s32TempArray << std::endl;

    for (uint32_t u32I = 0U; u32I < 50U; u32I++)
    {
        f64TempArray[u32I] = (f64_t)(50U - u32I);
    }
    std::cout << f64TempArray << std::endl;

    return 0;
}
```



2. Klassentemplates

10

Einfaches Klassentemplate – Polymorphismus zur Compiletime

```
template <typename T, uint32_t u32S>
class SimpleArray
```

```
SimpleArray<int32_t, 10U> s32TempArray;
```

```
SimpleArray<f64_t, 50U> f64TempArray;
```

Während der Compilezeit wird aus dem Template eine Klasse generiert und übersetzt. Die Template-Parameter werden dabei durch die in < > aufgeführten Datentypen ersetzt.

Während der Compilezeit wird aus dem Template eine Klasse generiert und übersetzt. Die Template-Parameter werden dabei durch die in < > aufgeführten Datentypen ersetzt.

Es gibt jetzt zwei Klassen SimpleArray. Mehr Speicher wird im Code Segment benötigt.

Es können theoretisch eine unbegrenzte Anzahl von Varianten aus einer Templateklasse erzeugt werden.

2. Klassentemplates

11

Einfaches Klassentemplate – Ersetzen der Templateparameter

template <typename T, uint32_t u32S>
class SimpleArray
{
public:
 SimpleArray() //Constructor
 {
 pElements_ = new T[u32Size_ = u32S];
 }
 ~SimpleArray() //Destructor
 {
 delete[] pElements_;
 }
 T& operator[](int32_t s32Index)
 {
 return (pElements_[s32Index]);
 }
private:
 T* pElements_;
 uint32_t u32Size_;
};

⇒ SimpleArray<int32_t, 10U> s32TempArray;

Compiler generiert eine Klasse SimpleArrayX

```
class SimpleArrayX
{
public:
    SimpleArrayX() //Constructor
    {
        pElements_ = new int32_t[u32Size_ = 10U];
    }
    ~SimpleArrayX() //Destructor
    {
        delete[] pElements_;
    }
    int32_t& operator[](int32_t s32Index)
    {
        return (pElements_[s32Index]);
    }
private:
    int32_t* pElements_;
    uint32_t u32Size_;
};
```

Template-Parameter werden ersetzt

2. Klassentemplates

12

Template-Parameter

← Typparameter ← Gewöhnliche Parameter →

```
template <typename T, typename U, int32_t s32V, f64_t f64V>
class X
{ ... };
// ...
X<int32_t, Car, 5, 3.0> MyX;
```

Bedingungen

- Mindestens ein Typparameter (generischer Datentyp)
- Optionale gewöhnliche Parameter

Es sind aber auch **Default-Argumente** möglich:

```
template <typename T=int32_t, uint32_t u32V = 10U>
class Y;
{ ... };
// ...
Y<> MyY;
```

Sobald ein Template-Parameter ein Default-Argument hat, müssen alle folgenden Template-Parameter auch ein Default-Argument haben (ähnlich wie bei Default-Parametern).

2. Klassentemplates

13

Möglichkeiten der Realisierung von Templates

Der C++ Compiler muss zum Zeitpunkt des Anlegens der speziellen Klasse (basierend auf dem Template) das gesamte Template „sehen“ (also Deklaration und Definition).

Der **einfachste Weg** ist, dass man alles nur in eine Header-Datei packt. Die Objektmethode werden somit alle „inline“. Dies sieht zwar unschön aus, wird auch oft prinzipiell abgelehnt. Dies wurde gerade auf den vorherigen Folien gezeigt.

Die zweite **Möglichkeit** ist, dass man Deklaration (Header-Datei) und Definition (Cpp-Datei) trennt und man aber die **Cpp-Datei am Ende im Header inkludiert**.

Hinweis: In der C/C++-Programmierung sollte dies sollte grundsätzlich vermieden werden (Mehrfachinkludierung in mehreren C/C++ Dateien), da sonst Funktionen und globale Variablen mehrfach vorkommen können. Bei reinen Templates ist dies allerdings erlaubt/möglich.

In beiden Fällen wird nur die Header-Datei im jeweiligen Cpp-File der Anwendung hinzugebunden.

```
#include "TemplateSimpleArray.h"
```

2. Klassentemplates

14

Funktionsdefinition außerhalb der Templatedeklaration

Zweite **Möglichkeit**: Deklaration (Header-Datei) und Definition (cpp-Datei) getrennt.

Header-Datei

```
template <typename T>
class X
{
public:
    T foo(T);
    //...
};
```

Bei der Implementierung
müssen die Template-
Parameter nochmals
aufgeführt werden.

cpp-Datei

```
template <typename T>
T X<T>::foo(T t)
{
    //Do something with t
    return t;
}
```

Im vorliegenden Beispiel von SimpleArray musste die Operator-Funktion << auch außerhalb des Templates implementiert werden. Hier müssen auch nochmals die Template-Parameter aufgeführt werden: Sowohl in der Friend-Deklaration als auch bei der Implementierung.

Bei der Forward-Deklaration allerdings nicht.

```
template <class T, uint32_t u32S>
std::ostream& operator<<<T, u32S>(std::ostream& ostr,
                                const SimpleArray<T, u32S>& crT)
```

3. Beispiel Stackklasse

15

Prinzip der Datenstruktur Stack

In der Informatik gibt es eine Datenstruktur Stack, welches ein **LIFO** (Unterschied dazu: **FIFO**) darstellt. Der Stackspeicher arbeitet nach dem gleichen Prinzip. Das neueste (jüngste) Element wird gelesen. Die Lese- und Schreibfunktionen heißen meist im Zusammenhang mit einem Stack: **Push** und **Pop**.

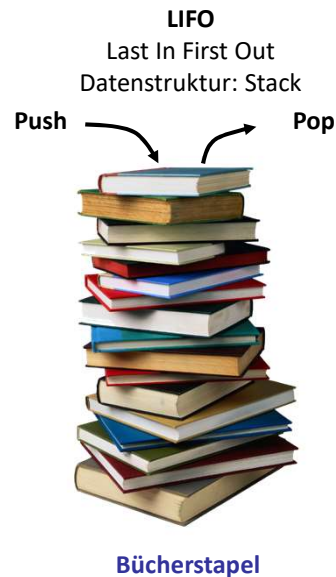
Push: Auf dem Stack ablegen

Pop: Vom Stack wieder entnehmen

In einer Stackklasse befindet sich dann wiederum ein Array (oder eine Collection), welches die Element speichert.

Im vorliegenden Beispiel soll die Anzahl der Elemente begrenzt sein.

Um Stacks für beliebige Elemente (int32_t, int8_t*, Complex, ...) anlegen zu können, soll dafür ein Template realisiert werden.



3. Beispiel Stackklasse

16

Implementierung in getrennter .h- und .cpp-Datei

```
namespace MyDataStructures
{
    template<typename T, uint32_t u32Size>
    class Stack
    {
    public:
        bool bIsEmpty(void) const noexcept(true);
        bool bIsFull(void) const noexcept(true);
        uint32_t u32GetNumberElements(void) const noexcept(true);
        void vClear(void) noexcept(true);
        const T& xPop(void) noexcept(false); //x own Datatype
        void vPush(const T& crT) noexcept(false);
    private:
        T Elements_[u32Size]{ 0 };
        uint32_t u32ElementsInStack_{ 0 }; //Brace yourself
    };
}
#include "TemplateSimpleStack.cpp"
```

Reihenfolge const und
noexcept ist wichtig.

Die cpp-Datei darf **nicht** ins Projekt (Header Files oder Resource Files) hinzugefügt werden.

3. Beispiel Stackklasse

17

Methoden in TemplateSimpleStack.cpp

Erspart MyDataStructures:: vor Stack
`using namespace MyDataStructures;`

```
#include <stdint>
#include <exception>
```

```
template<typename T, uint32_t u32Size>
bool Stack<T, u32Size>::bIsEmpty(void) const noexcept(true)
{
    return (0U == u32ElementsInStack_);
}

template<typename T, uint32_t u32Size>
bool Stack<T, u32Size>::bIsFull(void) const noexcept(true)
{
    return (u32Size == u32ElementsInStack_);
}

template<typename T, uint32_t u32Size>
uint32_t Stack<T, u32Size>::u32GetNumberElements(void) const noexcept(true)
{
    return (u32ElementsInStack_);
}
```

Alle Methoden verändern nicht die Attribute und werfen auch keine Exceptions.

3. Beispiel Stackklasse

18

Methoden in TemplateSimpleStack.cpp

```
template<typename T, uint32_t u32Size>
void Stack<T, u32Size>::vClear(void) noexcept(true)
{
    u32ElementsInStack_ = 0U;
}

template<typename T, uint32_t u32Size>
const T& Stack<T, u32Size>::xPop(void) noexcept(false)
{
    if (bIsEmpty() == true)
    {
        throw std::runtime_error("Stack is Empty");
    }
    else
    {
        u32ElementsInStack_--;
        return (Elements_[u32ElementsInStack_]);
    }
}
```

Beide Methoden verändern die Attribute. xPop wirft Exception (noexcept(false))

3. Beispiel Stackklasse

19

Methoden in TemplateSimpleStack.cpp

```
template<typename T, uint32_t u32Size>
void Stack<T, u32Size>::vPush(const T& crT) noexcept(false)
{
    if (bIsFull() == true)
    {
        throw std::runtime_error("Stack is Full");
    }
    else
    {
        Elements_[u32ElementsInStack_] = crT;
        u32ElementsInStack_++;
    }
}
```

vPush verändert die
Attribut und wirft
Exception (noexcept(false))

Was für eine Methode könnte noch sinnvoll fürs Debugging sein?

3. Beispiel Stackklasse

20

Ausgabe des Stacks auf dem Bildschirm

```
Microsoft ... - x
-Top of Stack-
xxxxxx
13
12
11
10
-Bottom of Stack-
Read from Stack: 13 and 12
-Top of Stack-
xxxxxx
xxxxxx
xxxxxx
11
10
-Bottom of Stack-
```

xxxxxx steht für freien Platz
im Stack.

```
MyDataStructures::Stack<int32_t, 5U> MyStack;

try
{
    MyStack.vPush(10);
    MyStack.vPush(11);
    MyStack.vPush(12);
    MyStack.vPush(13);
    std::cout << MyStack;

    auto a1 = MyStack.xPop();
    auto a2 = MyStack.xPop();
    std::cout << "Read from Stack: " << a1 <<
        " and " << a2 << std::endl;
    std::cout << MyStack << std::endl;
}
catch (std::exception ex)
{
    std::cout << ex.what() << std::endl;
}
```

3. Beispiel Stackklasse

21

Ausgabe des Stacks auf dem Bildschirm - Implementierung

```
namespace MyDataStructures
{
    template<class T, uint32_t u32Size>
    class Stack;

    template<class T, uint32_t u32Size>
    std::ostream& operator<<(std::ostream& ostr, const Stack<T, u32Size>& crT);

    template<typename T, uint32_t u32Size>
    class Stack
    {
    //more Code
    friend std::ostream& operator<<<T, u32Size>
        (std::ostream& rostr, const Stack<T, u32Size>& crT);

    };
    template<class T, uint32_t u32Size>
    std::ostream& operator<<<T, u32Size>
        (std::ostream& ostr, const MyDataStructures::Stack<T, u32Size>& crT)
    {
    //more Code
    }
}
```

Forward-Deklaration Template Stack

Forward-Deklaration
friend Operatorfunktion.

friend-Beziehung

Implementierung

3. Beispiel Stackklasse

22

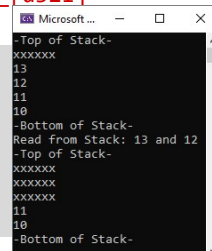
Ausgabe des Stacks auf dem Bildschirm - Implementierung

```
template<class T, uint32_t u32Size>
std::ostream& operator<<<T, u32Size>(std::ostream& ostr,
    const MyDataStructures::Stack<T, u32Size>& crT)
{
    std::cout << "-Top of Stack-" << std::endl;
    for (uint32_t u32I = u32Size; u32I > 0; u32I--)
    {
        if (u32I > crT.u32ElementsInStack_)
        {
            ostr << "xxxxxx" << std::endl;
        }
        else
        {
            ostr << crT.Elements_[u32I - 1U] << std::endl;
        }
    }
    std::cout << "-Bottom of Stack-" << std::endl;
    return ostr;
}
```

Alternative?

```
for (uint32_t u32I = u32Size-1;
    u32I >= 0;
    u32I--)
    und crT.Elements_[u32I]
```

Stack muss noch []-Operator haben.



```
Microsoft... - - - X
-Top of Stack-
xxxxxx
13
12
11
10
-Bottom of Stack-
Read from Stack: 13 and 12
-Top of Stack-
xxxxxx
xxxxxx
11
10
-Bottom of Stack-
```

3. Beispiel Stackklasse

23

Nutzung für eigene Datentypen?

Könnte die Stackklasse mit einem eigenen Datentyp verwendet werden? Was müsste am Stack-Template geändert werden, um die eigene Complex-Klasse zu nutzen?

```
template<typename T, uint32_t u32Size>
class Stack
{
public:
    bool bIsEmpty(void) const noexcept(true);
    bool bIsFull(void) const noexcept(true);
    uint32_t u32GetNumberElements(void) const noexcept(true);
    void vClear(void) noexcept(true);
    const T& xPop(void) noexcept(false); //x own Datatype
    void vPush(const T& crT) noexcept(false);
    T& operator[] (int32_t s32Index) noexcept (false);

private:
    T Elements_[u32Size];
    uint32_t u32ElementsInStack_{ 0 }; //Brace yourself
    friend std::ostream& operator<<<T, u32Size>(std::ostream& rostr,
                                                Stack<T, u32Size>& crT);
};
```

1. T muss leeren Konstruktor haben

Für die Ausgabe von des Stacks auf der Konsole benötigt Stack noch []

2. T muss <<-Operator haben – der <<-Operator von Stack verwendet diesen.

3. Beispiel Stackklasse

24

Erweiterte Complex-Klasse

```
class Complex
{
public:
    Complex()
    {
        f64Real_ = 0.;
        f64Img_ = 0.;
    }
    //more Code
    friend std::ostream& operator<<<(std::ostream& rostr, Complex& rC);
};

std::ostream& operator<<<(std::ostream& rostr, Complex& rC)
{
    rostr << rC.f64Real_ << " +j " << rC.f64Img_;
    return rostr;
}
```

1. T muss leeren Konstruktor haben. Der leere Default-Konstruktor wurde durch einen Konstruktor mit zwei f64_t Parametern gelöscht.

2. T muss <<-Operator haben – der <<-Operator von Stack verwendet diesen.

3. Beispiel Stackklasse

25

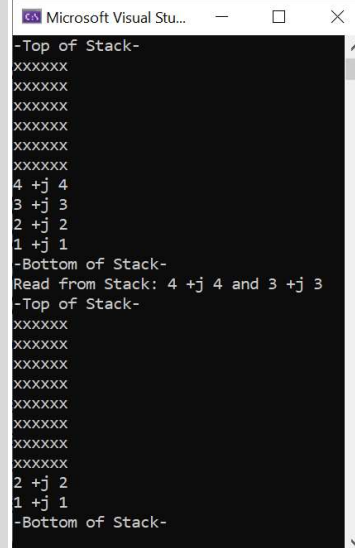
Beispiel mit Ausgabe

```
MyDataStructures::Stack<Complex, 10U> MyStack2;
```

Stack für Complex, 10 Elemente

```
try
{
    MyStack2.vPush(Complex(1.,1.));
    MyStack2.vPush(Complex(2.,2.));
    MyStack2.vPush(Complex(3.,3.));
    MyStack2.vPush(Complex(4.,4.));
    std::cout << MyStack2;

    auto a1 = MyStack2.xPop();
    auto a2 = MyStack2.xPop();
    std::cout << "Read from Stack: " << a1 <<
        " and " << a2 << std::endl;
    std::cout << MyStack2 << std::endl;
}
catch (std::exception ex)
{
    std::cout << ex.what() << std::endl;
}
```



```
Microsoft Visual Stu...
-Top of Stack-
xxxxxx
xxxxxx
xxxxxx
xxxxxx
xxxxxx
4 +j 4
3 +j 3
2 +j 2
1 +j 1
-Bottom of Stack-
Read from Stack: 4 +j 4 and 3 +j 3
-Top of Stack-
xxxxxx
xxxxxx
xxxxxx
xxxxxx
xxxxxx
2 +j 2
1 +j 1
-Bottom of Stack-
```

Templates

26

Zusammenfassung

- DRY-Prinzip: **Don't repeat yourself!** Code muss nur einmal für mehrere Datentypen geschrieben werden.
- Syntax gerade im Zusammenhang mit friend-Funktionen ist schwierig.
- Code für konkreten Typparameter und/oder gewöhnliche Parameter wird zur Compilezeit erzeugt (eigene Klasse).
- Ein Sourcecode (Templateklasse) ergibt je nach Nutzung Maschinencode für die unterschiedlichen Klassen.
- Auf den ersten Blick ist es nicht immer sofort ersichtlich, welche Operatoren/Funktionen eine Klasse mitbringen muss, die als Templateparameter verwendet wird.
- Templates sind ein sehr wichtiges Features ... würde genügend Stoff für eine Master-Vorlesung bieten.