Templates

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Template-Konzepte in C++`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/03_Templates.md
Autoren	Sebastian Zug & Georg Jäger

Zielstellung der heutigen Veranstaltung

- Einführung in die Konzepte der generischen Programmierung unter C++
- Beschreibung der aktuellen Entwicklungen in C++17 und C++20
- Abgrenzung zu den unter C# bekannten Ansätzen (*Templates* vs. *Generics*)

Fragen aus der vergangenen Woche

Innerhalb des .NET Frameworks definieren Generics Typparameter, wodurch Sie Klassen und Methoden entwerfen können, die die Angabe eines oder mehrerer Typen verzögern können, bis die Klasse oder Methode vom Clientcode deklariert und instantiiert wird. Ein Platzhalter, der, generischen Typparameter der häufig mit `T` bezeichnet wird, definiert die Art einer Variablen, die von anderem Clientcode verwendet werden kann, ohne dass die Kosten und Risiken von Umwandlungen zur Laufzeit oder Boxingvorgängen anfallen.

Das folgende Beispiel zeigt eine Anwendung in Kombination mit einer Einschränkung des Typs, die Sicherstellt, dass in jedem Fall die angeforderte Vergleichsoperation besteht.

Example.csp

```
1  using System;
2
3  public class Student
4  {
5    public string name;
6    // ... and some other information
7
8    public Student(string name){
9      this.name = name;
10   }
11 }
12
13 public class Program{
14
15   static void SwapIfGreater<T>(ref T lhs, ref T rhs)
16                 where T : System.IComparable<T>
17   {
18       T temp;
19       if (lhs.CompareTo(rhs) > 0)
20       {
21           temp = lhs;
22           lhs = rhs;
23           rhs = temp;
24       }
25   }
26
27   public static void Main(string[] args)
28   {
29       int a = 5;
30       int b = 7;
31       SwapIfGreater<int>(ref a, ref b);
32       System.Console.WriteLine("a=" + a + ", b=" + b);
33   }
34 }
```

project.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

> Templates ermöglichen die Realsierung eines typunabhängigen Verhaltens und damit die Konzentration von Implementierungsaufwand.

Arten von Templates

Templates 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. Damit gibt es einen einzige Definition für jeden Container, wie z.B. Vektor, aber wir können viele verschiedene Arten von Vektoren definieren, z.B. `std::vector<int>` oder `std::vector<string>`.

Dabei unterschieden wir zwei grundsätzliche Anwendungsfälle:

- Funktionstemplates
- Klassentemplates

Funktionstemplates

Motivation

Welche Möglichkeiten haben wir unter C++ schon kennengelernt, die einen variablen Umgang von identischen Funktionsaufrufen mit unterschiedlichen Typen realisieren? Dabei unterstützen uns Cast-Operatoren und das Überladen von Funktionen.

FunctionOverloading.cpp

```cpp
#include <iostream>

void print (int value){
  std::cout << "Der Wert ist " << value << std::endl;
}

int main()
{
  print(5);
  print(10.234);
  //print("TU Freiberg");
  return EXIT_SUCCESS;
}
```

Ein Funktions-Template (nicht Template-Funktion!) verhält sich wie eine Funktion, die Argumente verschiedener Typen akzeptiert oder unterschiedliche Rückgabetypen liefert. Die C++-Standardbibliothek enthält eine Vielzahl von Funktionstemplates, die folgendem Muster einer selbstdefinierten Funktion entsprechen.

FunctionTemplate.cpp

```cpp
#include <iostream>

template<typename T>          // Definition des Typalias T
void print (T value){         // Innerhalb von print wirkt T als Platzhalter
  std::cout << value << std::endl;
}

int main()
{
  print(5);
  print(10.234);
  print("TU Freiberg");
  return EXIT_SUCCESS;
}
```

Umsetzung

Was macht der Compiler daraus? Lassen Sie uns prüfen, welche Symbole erzeugt wurden.

Das Linux-Tool `nm` wird verwendet, um Binärdateien (einschließlich Bibliotheken, kompilierter Objektmodule, gemeinsam genutzter Objektdateien und eigenständiger ausführbarer Dateien) zu untersuchen und den Inhalt dieser Dateien oder die in ihnen gespeicherten Metainformationen, insbesondere die Symboltabelle, anzuzeigen.

Um die gleichnamigen Funktionen unterscheiden zu können, kodieren C++ sie in einen Low-Level-Assemblernamen, der jede unterschiedliche Version eindeutig identifiziert. Dieser Vorgang wird als *mangling* bezeichnet. Das Programm c++filt führt die umgekehrte Abbildung durch: es dekodiert (*demangled*) Low-Level-Namen in User-Level-Namen, so dass sie gelesen werden können.

Offenbar wurde aus unserem Funktionstemplate entsprechend der Referenzierung 3 unterschiedliche Varianten generiert. Mit `c++filt` kann der Klarname rekonstruiert werden.

```
>g++ Templates.cpp
>nm a.out | grep print
0000000000000a26 W _Z5printIdEvT_
00000000000009f2 W _Z5printIiEvT_
0000000000000a64 W _Z5printIPKcEvT
>c++filt _Z5printIdEvT_
void print<double>(double)
```

Begriffe

Damit sollten folgende Begriffe festgehalten werden:

- ein *Funktionstemplate* definiert ein Muster oder eine Schablone für die eigentliche Funktionalität
- das *Funktionstemplate* wird initialisiert, in dem die Templateparameter festgelegt werden. Es entsteht eine *Templatefunktion*.
- der Compiler übprüft ob die angegebenen Datentypen zu einer validen Funktionsdefinition führen und generiert diese bei einem positiven Match.
- der Compiler versucht solange alle Funktionstemplates mit den angegebenen Datentypen zu initialisieren, bis diese erfolgreich war, oder kein weiteres Funktionstemplate zur Verfügung steht.

Warum funktioniert das Konzept bisher so gut? Alle bisher verwendeten Typen bedienen den Operator `<<` für die Stream-Ausgabe. Eine Datenstruktur, die dieses Kriterium nicht erfüllt würde einen Compilerfehler generieren.

OperatorOverloading.cpp

```cpp
#include <iostream>

struct Student{
  std::string name;
  Student(std::string n): name{n} { }
};

template<typename T>          // Definition des Typalias T
void print (T value){         // Innerhalb von print wirkt T als Platzhalter
  std::cout << value << std::endl;
}

int main()
{
  print(5);
  Student Humboldt{"Alexander"};
  print(Humboldt);
  return EXIT_SUCCESS;
}
```

Templateparamter und Templatespezialisierungen

Zwei Fragen bleiben noch offen:

Frage	Antwort
Muss der Templateparameter zwingend angegeben werden?	Nein, wenn Sie im nachfolgenden Codebeispiel die Funktion `print(int value)` entfernen, funktioniert die Codegenerierung noch immer. Der Compiler erkennt den Typen anhand des übergebenen Wertes.
Ist ein Nebeneinander von Funktionstemplates und allgemeinen Funktionen möglich?	Ja, denn der Compiler versucht immer die *spezifischste* Funktion zu nutzen. Das heißt, zunächst werden alle nicht-templatisierten Funktionen in Betracht gezogen. Im zweiten Schritt werden teilweise-spezialisierte Funktionstemplates herangezogen und erst zuletzt werden vollständig templatisierte Funktionen genutzt. Untersuchen Sie das Beispiel mit `nm` und `c++filter`!

FunctionTemplate.cpp

```cpp
#include <iostream>

void print (int value){
  std::cout << "Die Funktion wird aufgerufen" << std::endl;
  std::cout << value << std::endl;
}

template<typename T>
void print (T value){
  std::cout << "Die Templatefunktion wird aufgerufen" << std::endl;
  std::cout << value << std::endl;
}

template<>
void print<float> (float value){
  std::cout << "Die spezialisierte Templatefunktion wird aufgerufen"
    ::endl;
  std::cout << value << std::endl;
}

int main()
{
  print(5);         // funktioniert auch ohne explizite print(int valu
                    // der Kompiler deduziert den Typ aus unserem Aufr
  print<int>(5);

  double v = 5.0;
  print<float>(v);
  print(v);
  return EXIT_SUCCESS;
}
```

Insbesondere teilweise und vollständige Templatespezialisierungen ermöglichen es Ausnahmen von generellen Abbildungsregeln darzustellen.

TemplateSpecialization.cpp

```cpp
#include <iostream>

template<typename T>
void print (T value){
  std::cout << value << std::endl;
}

template <> void print<bool> (bool value){
  std::cout << (value ? "true" : "false") << std::endl;
}

int main()
{
  print(5);
  print<bool>(true);
  return EXIT_SUCCESS;
}
```

Damit lassen sich dann insbesondere für Templates mit mehr als einem Typparameter komplexe Regelsets aufstellen:

```cpp
template<class T, class U>  // Generische Funktion
void f(T a, U b) {}

template<class T>           // Teilweise spezialisiertes Funktions-Template
void f(T a, int b) {}

template<>                  // Vollständig spezialisiert; immer noch Templa
void f(int a, int b) {}
```

Daraus ergibt sich folgende Aufrufstruktur:

Aufruf	Adressierte Funktion
`f<int, int> (3,7);`	Generische Funktion
`f<double> (3.5, 5);`	Überladenes Funktionstemplate
`f(3.5, 5);`	Überladenes Funktionstemplate
`f(3, 5);`	Vollständig spezialisiertes Funktionstemplate

Methoden-Templates sind auch nur Funktionstemplates, dass heißt die gerade vorstellten Mechanismen lassen sich im Kontext Funktion, die einer Klasse zugeordnet ist analog umsetzen.

Zahlen als Templateparameter

Es ist üblich, dass wir Typen als Templateparameter verwenden. Der C++ Standard lässt aber explizit auch die Übergabe von Zahlenwerten als Konfigurationsparameter zu. Wo brauche ich so was?

TemplateSpecialization.cpp

```cpp
#include <iostream>
#include <array>

template<typename T, size_t SIZE>
std::array<T, SIZE> createArray() {
    std::array<T, SIZE> result{};
    return result;
}

int main()
{
  auto data = createArray<std::string, 5>();
  data.fill("___");
  data[1] = "Tralla";
  for(auto e: data) std::cout << e << " ";
  return EXIT_SUCCESS;
}
```

Klassentemplates

Grundsätzliche Idee

Ein Klassen-Template geht einen Schritt weiter und wendet die Template-Konzepte einheitlich auf eine Klasse an. Dies können selbstgeschriebene Klassen sein oder Klassen, die zum Beispiel in der C++ Standardbibliothek (STL) eingebettet sind und Container für Daten definieren.

Das folgende Bespiel definiert ein eigenes Klassentemplate, dass die Verwaltung einer Variablen übernimmt. Sofern ein gültiger Wert hinterlegt wurde, ist die entsprechende Kontrollvariable gesetzt:

ClassTemplate.cpp

```cpp
#include <iostream>
#include <list>
#include <string>

template <typename T>
class OptionalVariable{
    T variable;
    bool valid;
  public:
    OptionalVariable() : valid(false) {}
    OptionalVariable(T variable): variable(variable), valid(true) {}
    T getVariable(){
      if (!valid){
        std::cerr << "Variable not valid! ";
      }
      return variable;
    }
    void clear(){
      valid = false;
    }
};

int main(){
  OptionalVariable<int> Para1 = OptionalVariable<int>(5);
  std::cout << Para1.getVariable() << std::endl;

  OptionalVariable<std::string> Para2 = OptionalVariable<std::string
    ("Das ist ein Test");
  Para2.clear();
  std::cout << Para2.getVariable() << std::endl;
}
```

Auch in diesem Kontext kann eine Spezialisierung von Templates für bestimmte Typen erfolgen. Spezialisieren Sie beispielsweise das Template für den Typ `std::string` und setzen Sie in der Methode `clear` die zugehörige Variable auf einen leeren Ausdruck `""`.

Klassentemplates der STL

Anwendungsseitig spielen Templates im Zusammenhang mit den Containern der STL eine große Bedeutung. Die Datenstrukturen sind (analog zu C#) als Klassentemplates realisiert.

TemplatesInSTL.cpp

```cpp
#include <iostream>
#include <list>
#include <string>

class MyClass{
   private:
     std::string name;
   public:
     MyClass(std::string name) : name(name) {}
     std::string getName() const {return this->name;}
};

int main(){
    // Beispiel 1
    std::list< int > array = { 3, 5, 7, 11 };
    for(auto i = std::begin(array);
        i != std::end(array);
        ++i
    ){
        std::cout << *i << ", ";
    }
      std::cout << std::endl;

    // Beispiel 2
    std::list<MyClass> objects;

    objects.emplace_back("Hello World!");
    for (auto it = objects.begin(); it!=objects.end(); it++) {
      std::cout << it->getName() << " ";
    }
}
```

Die Containerklassen und deren Methoden werden in der kommenden Vorlesung vorgestellt.

Mehrfache Template-Parameter

Darüber hinaus können (wie auch bei den Funktionstemplates) mehrere Datentypen in Klassentemplates angegeben werden. Der vorliegende Code illustriert dies am Beispiel einer Klassifikation, die zum Beispiel mit einem neuronalen Netz erfolgte. Ein Set von Features wird auf eine Klasse abgebildet.

Container.cpp

```cpp
#include <iostream>
#include <list>
#include <string>
#include <iterator>

template<typename T, typename U>
class Classification{
    private:
      std::list<T> input;
      U category;
    public:
      template<typename ContainerType>
      Classification(ContainerType t, U u) : input(t.begin(), t.end()),
        category(u) {}
      void print(std::ostream& os) const{
         os << "Category: " << this->category << std::endl;
         for(const auto& v : this->input)
         {
            os << v << std::endl;
         }
      }
};

int main()
{
    std::list<int> samples {23, 2, 19, -12};
    std::string category  {"Person"};
    Classification<int, std::string> datasample {samples, category};
    datasample.print(std::cout);
}
```

Ausgehend davon folgt dann sofort die Frage, ob sich die Zuordnung von mehr als einem formalen Datentypen auch bei der Spezialisierung berücksichtigen lässt. Ja, in vollem Umfang lassen sich die Kombinationen der Typparameter partiell UND vollständig spezialisieren.

> Merke: Funktionstemplates können nicht partiell spezialisiert werden!

...aber überladen werden:

FunctionTemplateOverloading.cpp

```cpp
#include <iostream>

template<typename T1, typename T2>
void print2 (T1 value, T2 value2){
  std::cout << "Die Templatefunktion wird aufgerufen" << std::endl;
  std::cout << value << std::endl;
  std::cout << value2 << std::endl;
}

template<typename T>
void print2 (std::string value, T value2){
  std::cout << "Die überladene Templatefunktion wird aufgerufen" << s
    ::endl;
  std::cout << value << std::endl;
  std::cout << value2 << std::endl;
}

int main()
{
  print2(1, 3.14);
  print2(std::string("Value: "), 1.0);
  return EXIT_SUCCESS;
}
```

Templates und Vererbung

Zwischen Klassentemplates können Vererbungsrelationen bestehen, wie zwischen konkreten Klassen. Dabei sind verschiedene Konfigurationen möglich:

- die erbende Klasse nutzt den gleichen formalen Datentypen wie die Basisklasse (vgl. nachfolgendes Beispiel 1)
- die erbende Klasse erweitert das Set der Templates um zusätzliche Parameter
- die erbende Klasse konkretisiert einen oder alle Parameter
- die Vererbungsrelation besteht zu einem formalen Datentyp, der dann aber eine Klasse sein muss. (Beispiel 2)

Inheritance.cpp

```cpp
#include <iostream>
#include <list>
#include <string>

template<typename T>
class Base{
   private:
     T data;
   public:
     void set (const T& value){
        data = value;
     }
};

template<typename U>
class Derived: public Base<U>{
  public:
    void set (const U& value){
      Base<U>::set(value);
      // some additional operations happen here
    }
};

int main(){
    Derived<int> A;
    A.set(5);
}
```

Im Beispiel 2 erbt die abgeleitete Klasse unmittelbar vom Templateparameter.

InheritanceFromFormelType.cpp

```cpp
#include <iostream>
#include <list>
#include <string>
#include <chrono>

class Data{
  private:
    double value;
  public:
    Data(double data): value(data) {}
    double get() const{
      return value;
    }
};

template<typename T>
class TimeStampedData: public T{
  private:
    std::chrono::time_point<std::chrono::system_clock> time_point;
  public:
    TimeStampedData(double value): T(value) {
      time_point = std::chrono::system_clock::now();
    }
    void print(std::ostream& os) const{
      std::time_t ttp = std::chrono::system_clock::to_time_t(time_poi
      os << "time: " << std::ctime(&ttp) << "value: " << this->get();
    }
};

int main(){
    TimeStampedData<Data> A {5};
    A.print(std::cout);
}
```

Was ist kritisch an dieser Implementierung?

- Die formelle Festlegung auf `double` in der Klasse `Data` schränkt die Wiederverwendbarkeit drastisch ein!
- Das Klassentemplate setzt ein entsprechendes Interface voraus, dass einen Konstruktor, eine set-Funktion und ein Member data vom Typ double erwartet.

Typprüfungen - SFINAE

SFINAE - "Substitution Failure Is Not An Error" - sind ein zentrales Element der Verwendung von Templates in C++. Über den Generationen des Standards haben sich hier deutliche Vereinfachungen ergeben.

> *... the point of SFINAE is to deactivate a piece of template code for certain types.* Jonathan Boccara

Dieser Teil der Vorlesung wurde in starkem Maße durch den Blogbeitrag von Bartlomiej Filipek motiviert Link. Beginnen wir zunächst mit einem Motivationsbeispiel, dass das Problem beschreiben soll. Wie können wir vermeiden, dass eine

SubstitutionError.cpp

```cpp
#include <iostream>

struct Bar {
    typedef double internalType;
};

template <typename T>
typename T::internalType foo(const T& t) {
    std::cout << "foo<T>\n";
    return 0;
}

//int foo(int i) {
//  cout << "foo(int)\n"; return 0;
//}

int main() {
    foo(Bar());
    foo(0); // << error!
}

```

Offenbar findet sich nach der Auflösung der Templateparameter T keine überladene Funktion, die für eine Integer-Variable gültig ist. Die Ersetzung scheitert am Aufruf des Rückgabewertes `T::interalType`, der für `int` nicht implementiert ist. Der Compiler realisiert also die fehlende Verfügbarkeit der Membervariable.

> **Achtung:** Die folgenden Beispiele hängen von den jeweiligen Standards ab, die der Comiler abdeckt. Beim g++ zum Beispiel kann über `-std=c++17` angegeben werden, dass dieser den C++17 Standard und damit den Funktionsumfang der Standardbibliothek umfasst.

C++11 Methoden

Achtung: Um es noch mal in aller Deutlichkeit zu sagen ... wir prüfen hier typbezogene Bedingungen zur Compilezeit ab!

Ausgangspunkt ist die Methode `enable_if`, die das Abprüfen von Bedingungen erlaubt. Die Implementierung besteht aus zwei Funktionstemplates - eine generischen und einer spezifizierten Variante.

```
template<bool Condition, typename T = void>
struct enable_if
{
};

template<typename T>
struct enable_if<true, T>
{
    using type = T;
};
```

`enable_if` wurde in C++ in C++11 standardisiert. Das folgende Codebeispiel zeigt die Verwendung:

enable_if.cpp

```
1  #include <iostream>
2  #include <list>
3  #include <string>
4  #include <chrono>
5
6  struct Bar {
7      int x;
8  };
9
10 template <class T>
11 typename std::enable_if<std::is_arithmetic<T>::value, T>::type
12 foo(T t) {
13   std::cout << "foo<arithmetic T>\n";
14   return t;
15 }
16
17 int main() {
18     foo(5);
19     foo(10.0);
20     foo(Bar());
21 }
```

```
                                                    Im Fall einer gültigen
                                                    Typrückgabe durch enable_if
```

```
                                                    |
std::enable_if<std::is_arithmetic<T>::value, T>::type
                  |                                 |
              Bedingung                         Resultat
```

```
std::is_abstract<>, std::is_base_of<>, std::is_const<>, std::is_object<>,
std::is_same<> ...
```

http://www.cplusplus.com/reference/type_traits/is_arithmetic/

Und mehrere Bedingungen?

enable_and.cpp

```
1  #include <iostream>
2  #include <string>
3
4  struct Bar {
5      int x;
6  };
7
8  template <typename T>
9  typename
10 std::enable_if<(std::is_floating_point<T>::value || std::is_integral<
     >::value ), T>::type
11 foo(T t) {
12   std::cout << "foo<arithmetic T>\n";
13   return t;
14 }
15
16 int main() {
17     foo(5);
18     //foo(Bar());
19 }
```

Achtung! ROS2 basiert auf C++11! Entsprechend werden Sie dort nur Constraints in dem hier gezeigten Format finden.

C++14/17 Methoden

C++14 fügt eine Variation von `std::enable_if` - `std::enable_if_t` hinzu. Dies ist nur ein Alias für den Zugriff auf den `::type` innerhalb von `std::enable_if`. In der selben Art wurden auch Aliase für die Zugriffe auf die Werte `_v` eingefügt. Damit wurde `std::is_floating_point<T>::value`

zu `std::is_floating_point_v<T>`. Das oben gezigte Beispiel in C++11 Syntax vereinfacht sich damit zu:

```
std::enable_if_t<std::is_arithmetic_v<T>, T>
                          |                  |
                      Bedingung          Resultat
```

enable_and.cpp

```cpp
#include <iostream>
#include <string>

template <typename T>
typename std::enable_if_t<std::is_arithmetic_v<T>, T>
foo(T t) {
  std::cout << "foo<arithmetic T>\n";
  return t;
}

int main() {
    foo(5);
    foo(15.6);
}
```

Irgendwelche Nachteile hat der SFINAE-Ansatz?

SFINAE und `enable_if` sind überzeugende Merkmale, aber auch schwer in realen Anwendungen einzusetzen:

- die Lesbarkeit des Codes und
- die Lesbarkeit der Fehlermeldungen leiden (dramatisch).

Zur Erinnerung an unsere Erfahrungen aus C# ... zumindestens die Angabe von verbindlichen Klassen und Interfaces ist deutlich besser lesbar gelöst.

Generics

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>
{
    // ...
}
```

C++20 Methoden

Variante 1 - Explizite Benennung von Requirements

ecplicite.cpp

```cpp
#include <iostream>
#include <string>
#include <typeinfo>

template <typename T>
auto calc(const T a, const T b) requires std::is_arithmetic_v<T>
{
  std::cout << "calc für " << typeid(a).name() << std::endl;
  return a + b;
}

int main() {
    calc(5, 5);
    calc(15.6, 234.345);
    //calc(15.6, 234);
}
```

Variante 2 - Concepts

concepts.cpp

```cpp
#include <iostream>
#include <string>
#include <typeinfo>

template<typename T>
concept number = std::is_arithmetic_v<T> ;

template <number T>
auto calc(const T a, const T b)
{
  std::cout << "calc für " << typeid(a).name() << std::endl;
  return a + b;
}

int main() {
    calc(5, 5);
    calc(15.6, 234.345);
    //calc(15.6, 234);
}
```

Template Parameter

Bei der Definition eines Templates kann entweder `class` oder aber `typename` verwendet werden. Beide Formate sind in den meisten Fällen austauschbar.

```
template<class T>
class Foo
{
};

template<typename T>
class Foo
{
};
```

Allerdings gibt es Unterschiede bei der Verwendung in Bezug auf Template-Templates (bis C++17) und bei abhängigen Typen.

1. Datentypen

... waren Gegenstand des vorangegangenen Kapitels.

2. Nichttyp-Parameter

Nichttyp-Parameter sind Konstanten, mit denen Größen, Verfahren oder Prädikate als Template-Parameter übergeben werden können. Als Nichttyp-Template-Parameter sind erlaubt:

- ganzzahlige Konstanten (inklusive Zeichenkonstanten),
- Zeigerkonstanten (Daten- und Funktionszeiger, inklusive Zeiger auf Member-Variablen und -Funktionen) und
- Zeichenkettenkonstanten

Verwendung finden Nichttyp-Parametern z. B. als Größenangabe bei std::array oder als Sortier- und Suchkriterium bei vielen Algorithmen der Standardbibliothek.

ConstantsAsTemplateParameters.cpp

```cpp
#include <iostream>
#include <cstddef>          // Für den Typ size_t

template <std::size_t N, typename T>
void array_init(T (&array)[N], T const &startwert){
    for(std::size_t i=0; i<N; ++i)
        array[i]=startwert + i;
}

int main(){
    const std::size_t size {10};
    int A[size];
    array_init<size, int>(A, 6);
    for (unsigned int i=0; i < size; i++){
      std::cout << A[i] << " ";
    }
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Template-Templates

Als Template-Parameter können aber auch wiederum Templates genutzt werden.

```cpp
template <template <typename, typename> class Container,
          typename Type>
class Example {
    Container<Type, std::allocator <Type> > myContainer;
    //...
};

// Anwendung
Example <std::deque, int> example;
```

Anwendung findet dies zum Beispiel in der Implementierung der `std::stack` Klasse, die ohne weitere Parameter eine `std::deque` als Datenstruktur verwendet. Diese wird in der Klassendeklaration als Standardparameter übergeben. Der zugrunde liegende Container kann eine der Standard-Container-Klassenvorlagen sein, der aber folgenden Vorgänge unterstützen muss:

- empty
- size
- back
- push_back
- pop_back

```
template<
    class T,
    class Container = std::deque<T>    // <- Warum wird hier auf die erneut
>                                      // Templatisierung verzichtet?
class stack;
```

Für andere Container gibt es ähnliche Realisierungen. Der Beitrag von Herb Sutter gibt dazu eine fundierte Diskussion der Performanceunterschiede von `std::list`, `std::vector` und `std::deque` Link.

Parameter-Packs

Der Vollständigkeit halber soll noch darauf hingewiesen werden, dass auch eine variable Zahl von Templateparametern realisiert werden kann. Diese Flexibilität wird dann durch `...` als Platzhalter ausgedrückt.

Vergleich generischer Konzepte unter C# und C++

https://www.artima.com/intv/generics.html#part2

Kriterium	C# Generics	C++ Templates
Template-Ziele	Klassen	Klassen und Funktionen
Typisierung	Stark	Schwach
Instanziierung	zur Laufzeit	zur Compilezeit
Default-Werte	nein	ja
Vollständige Spezialisierung	nein	ja
Partielle Spezialisierung	nein	ja
Nicht-Typen als Template-Parameter	nein	ja
Typ-Parameter als Basisklassen	nein	ja
Template-Templates	nein	ja

Und wie geht es weiter unter C++? Die Template-Metaprogrammierung greift noch weiter und definiert eine eigene Spracheebene in C++. Eine sehr anschauliche erste Einführung dazu bietet der Blogbeitrag von *-AB-* auf `coding::board` unter folgendem Link

Aufgabe der Woche

1. Recherchieren Sie die Möglichkeit *default* Werte bei der Angabe der Templateparameter zu berücksichtigen.
2. Implementieren Sie ein Beispiel, dass partielle und vollständige Spezialisierung einer Templateklasse realisiert.
3. Implementieren Sie eine Funktionstemplate, dass die Übergabe von Ganzzahlen und Floatingpoint-Zahlen in gemischter Form akzeptiert und verschiedene Casts für das Ergebnis realisiert.