

## Templates §10

### Motivation

- [Haase22, §10.1]: Jeweils 3 Funktionen mit identischem Funktionskörper, aber unterschiedlichen Datentypen bei den Input-Parametern.  
Die 3 Funktionen haben jeweils unterschiedliche Parameterlisten und damit unterschiedliche *Signatures*.
- Auch mit höheren Datentypen wie `vector<int>`, `vector<float>`, ...  
Zeigen in Beispiel `v_8a/`
- $\implies$  wie eine Schablone (engl. Template)

### Die erste Templatefunktion §10.1

Mit Fkt. `float max_elem(const vector<float>& x)` im Bsp. `v_8a/`:

- Ersetze alle (passenden) Typdeklarationen `float` durch den *Templateparameter T* (oder einen anderen Bezeichner).
- Deklariere unmittelbar **vor** der Funktion den Templateparameter:  
`template <class T>`
- Das ganze im Templatefile `tfkt.tcc` speichern (auch `tfkt.tpp`, `tfkt.hpp`, `tfkt.h` üblich) und als **Headerfile inkludieren**.
- In `main.cpp` die 6 Funktionsdefinitionen kommentieren und Zeile 1 aktivieren.
- Erläuterung implizite und explizite Templateargumente demonstrieren und an Tafel erläutern [Haase22, §10.1.2].
- Spezialisierung von Templates (Extrawurst für bestimmte Datentypen) [Haase22, §10.1.3].

Eine Template-Funktion ist eigentlich eine Familie von Funktionen.

### Größte Fibonaccizahl im Zahlbereich `v_8b/`

Grundidee: Was ist die größte Fibonaccizahl  $f_n := f_{n-1} + f_{n-2}$  für den übergebenen Datentyp?

Lösungsidee:

- Alle  $f_n$  sind nichtnegativ.
- Damit muß gelten:  $f_n \geq f_{n-1}$ .
  - Falls diese Bedingung nicht mehr gilt, dann wurde offenbar der (Integer-) Zahlbereich bei der Addition  $f_{n-1} + f_{n-2}$  überschritten (wir fangen wieder links am Zahlenstrahl an).
  - Somit ist  $f_{n-1}$  die größte im Datentyp noch darstellbare F-Zahl.

- Template-Funktion `T fibo(T& n)` realisiert dies und (Loop in Funktion als Struktogramm erläutern)  
Template-Parameter `T` wird implizit beim Aufruf aus dem [Datentyp von n](#) der Parameterliste ermittelt.
- Die Template-Funktion `void check_fibo()` kann den Template-Parameter nicht aus der Parameterliste ableiten, daher muß der Template-Parameter explizit beim Aufruf angegeben werden: `check_fibo<short int>()`;
- Spezialisierung für `check_fibo<signed char>()`; nötig, da `char` nicht als Zahl ausgegeben wird:  
`template <> void check_fibo<unsigned char>()`  
Demonstrieren, es gibt auch kürzere Lösungen.
- T-Fkt.: `void check_fibo2()`
  - Überprüft auf `char`-Datentypen über Anzahl der Bytes für Speicherung  
`const bool is_char = sizeof(T)<2;`
  - Flexibler Datentyp für `tmp1` in  
`auto tmp1 = is_char?static_cast<short>(f1):f1;`
  - Nutzt `is_integral<T>::value` um zur Compilezeit auf einen Integer-Datentyp für den Template-Parameter `T` zu überprüfen.  
Demo mit `check_fibo2<double>()`; und Abbruch der Compilierung.  
Erfordert: `#include <type_traits>`

Funktionen (und Klassen) können mehr als einen Templateparameter besitzen, z.B.:

```
template <class T, class S>
S myfunc(T const a, list<S> const &b);
```

Vorteile von Templatefunktionen:

- Algorithmische Verbesserungen müssen nur einmal implementiert werden.
- Vermeidet Copy-Paste Fehler im Vergleich zu mehreren Funktionen.
- Kleine Funktionen werden automatisch *inline* in aufrufender Funktion eingebaut.

## Templateklasse §10.2

Analog zu Templatefunktionen wird eine Familie von Klassen erzeugt.

```
1 template<class T>
  class X
3 {
  ...           // Definition der Klasse X<T>
5 };
```

Erst mit der Belegung des Templateparameters wird eine Klasse erzeugt und ein Objekt dieser Klasse deklariert.

```
1 {
  X<int> ai;
3  X<float> fi;
  vector< X<char> > vc;
5 };
```

Umformulierung der Klasse `Komplex` in eine Template-Klasse `Komplex<T>`: *v\_8b/*

1. `template <class T>` vor die Klassendeklaration schreiben. *(\*h)*
2. Den zu verallgemeinernden Datentyp durch den Template-Parameter `T` ersetzen. *(\*h, \*.tcc)*  
Achtung, vielleicht wollen Sie nicht an allen Stellen diesen Datentyp ersetzen.
3. Ersetze in allen Parameterlisten, Returntypen und Variablendeklarationen den Klassentyp `Komplex` durch die Template-Klasse `Komplex<T>`. *(\*h, \*.tcc)*
4. Vor jede Methodenimplementierung `template <class T>` schreiben. *(\*tpp)*  
Desgleichen vor Funktionen, welche die Klasse `Komplex::` als Parameter benutzen. *(\*tpp)*  
Bei `friend`-Funktionen muß man `template <class T>` auch im Deklarationsteil vor der Funktion angeben. *(\*h)*
5. In der Methodenimplementierung `Komplex::` durch `Komplex<T>::` ersetzen. *(\*tpp)*  
Achtung: `Komplex::Komplex(...)`  $\rightarrow$  `Komplex<T>::Komplex(...)`, dasselbe beim Destruktor.
6. Im Headerfile das Sourcefile includieren, also `#include "komplex.tcc"`, oder gleich alles in das Headerfile schreiben (nicht empfohlen). *(\*h)*

Weitere Topics:

- Mehrere Template-Parameter [Haase22, §10.2.2].
- `friend`-Funktionen und Template-Klasse [Haase22, §10.2.4].

## Einschränkungen an Template-Datentyp §10.3

- Überprüfung via Type Traits [Haase22, §10.3.1]. *v\_8c\_cpp17/*
- C++20: *Concepts* für Typüberprüfung [Haase22, §10.3.2]. *v\_8c\_cpp20/ komplex2.h*

# STL §11

Standard Template Library: Bsp. aus Listing 11.1 des Skriptes.

- Container
- Algorithmen
- Iteratoren
- Container und Algorithmen wissen nichts voneinander  $\implies$  Iteratoren als Bindeglied.

Analog zu §11.2 des Skriptes

- Algorithmen und welche Funktionen von den Containererelementen bereitgestellt werden müssen.
- Algorithmen mit  $\lambda$ -Funktion als binäre/unäre Funktion/Operator.
- unäre Boolean-Fkt: `bool bool_op(T const &a)`
- binäre Boolean-Fkt: `bool bool_op(T const &a, T const &b)`
- unäre arithm. Operation: `T arith_op(T const &a)`
- binäre arithm. Operation: `S arith_op(S const &a, T const &b)`

Example *v\_9a/* analog zu §11.3 demonstrieren.

- zuerst mit `vector<double>`
- dann mit `vector<Komplex>`
- und `list<Komplex>`

Anwendungen:

- Beispiel *sort\_index/*
- *graph\_1/* : Klasse einfacher Graph mit durchgehender Nummerierung
- *graph\_2/* : Klasse Graph ohne durchgehende Nummerierung (`set` für Vertexindizes)
- *graph\_3/* : Wie 3, aber eigene Datenstruktur `Edge` statt `array<int,2>`  
Mehr Methoden: `Delete`, `Append`, `max_vertex`, ...

---

Fun with ASCII<sup>1</sup>: *char\_sum.cpp*

- How to achieve 103% performance?
- { `HARDWORK`, `KNOWLEDGE`, `ATTITUDE`, `BULLSHIT` };
- `magic=96`  $\longrightarrow$  `magic=0` : `BILLGATES` (the third!)

---

<sup>1</sup><http://www.torsten-horn.de/techdocs/ascii.htm>

## Literatur

- [Haase22] Gundolf Haase: Einführung in die Programmierung mit C++ (2022), *www*<sup>2</sup>.
- [Stroustrup10] Bjarne Stroustrup: Einführung in die Programmierung mit C++. Pearson Studium, München (2010).

---

<sup>2</sup>[http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script\\_programmieren.pdf](http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf)