Notizen AIK ProgTech 3: Templates

Klaus Kusche

Sinn und Zweck:

Mit Templates kann man Code schreiben, der *für mehrere verschiedene Typen verwendet* werden kann: Ein Stück Template-Code hat einen *Typ als Parameter*, der dann bei der Verwendung des Templates durch einen bestimmten Typ *ersetzt* wird.

Beispiele:

- Eine *Sortier-Funktion* kann Arrays von **`int`**'s, **`double`**'s, Strings oder sogar von eigenen Objekten (wenn sie einen **<**-Operator haben) sortieren.
- Eine Klasse für *Listen* kann für *beliebige Element-Typen* verwendet werden.

Wie schreibt man Templates?

Unmittelbar *vor* dem Prototyp und dem Code der Funktion bzw. *vor* dem **`class`**:
`template <typename T>` oder **`template <class T>`** (beides ist genau dasselbe)

- Das Template gilt *für genau **ein** nachfolgendes Konstrukt*:
  Will man *zwei* Template-Funktionen schreiben, braucht man *zwei Mal* **`template`**.
- Der Parameter muss nicht **`T`** heißen, man darf ihn beliebig nennen (z.B. **`ElemType`**).
- Im nachfolgenden Konstrukt kann **`T`** ganz normal *als Typ verwendet* werden (für Parameter, Returntypen, Variablen- und Array-Deklarationen usw.).
- Ein Template kann auch *mehrere* Typ-Parameter enthalten:
  `template <class T1, class T2>`
- Ein Template kann *auch* **`int`***-Parameter* (z.B. für Arraygrößen) haben:
  `template <class T, int size>`
- **Achtung**:
  Wenn man Methoden einer Template-Klasse *außerhalb* des **`class`** definiert, muss man vor *jede* solche Definition *wieder* **`template`** schreiben und den *Template-Parameter zum Klassennamen* hinzufügen:

  ```
  template <class T>
  class myClass {
       int myMeth(int n);
  };

  template <class T>
  int myClass<T>::myMeth(int n) { ...
  ```

  Dasselbe gilt für die Definition *statischer Member* außerhalb von **`class`**.

Verwendung des Templates:

- *Template-Funktionen* können meistens *"ganz normal" aufgerufen* werden: Der Compiler findet automatisch heraus, welchen Typ er im Template einsetzen muss, damit das Template zu den Argumentwerten im Aufruf passt.
- Wenn das nicht klappt (z.B. weil die Typ-Parameter des Templates nicht aus den Argument-Typen im Aufruf berechnet werden können oder weil mehrere Varianten möglich wären), muss man die Typen *im Aufruf angeben*:

  ```
  x = myFunc<double>(i, 100);
  ```

- *Bei* **class**-*Templates* werden bei jeder Verwendung die einzusetzenden *Typen und Werte angegeben*:

  ```
  myArray<int, 100> arr;
  ```

  Für **int**-Parameter eines Templates können *nur Konstanten* eingesetzt werden, keine Variablen oder Rechnungen.

Anmerkungen:

- Bei Aufteilung in Header und Code-File gehören Templates in den *Header*!
- Um Fehlermeldungen im Zusammenhang mit Templates besser zu verstehen:
  Der Template-Code wird an der Stelle der Verwendung *eingesetzt*,
  die Template-Parameter werden erst dann durch die tatsächlichen Typen *ersetzt*,
  und *erst dann* wird das so entstandene Codestück *compiliert*
  (folglich liefert der Compiler auch erst an der *Stelle der Verwendung* des Templates *Meldungen für Fehler* im Template-Code, und das meist gleich mehrmals, wenn das Template mehrmals verwendet wird).
- **friend**-*Deklarationen* in **template**-Klassen sind aufwändig und bei den Schülern nicht Stoff (Details im Studenten-Skript).
- Templates *können viel mehr*, als wir hier besprechen:
  Spezial-Code für bestimmte Typen bei Typ-Parametern,
  Defaults für Template-Parameter,
  Function Pointer als Template-Parameter,
  variabel viele Template-Parameter,
  Templates als Template-Parameter, …
- In der Praxis sind vor allem im C++-Standard *vordefinierte Template-Klassen* wie **vector**, **list** oder **map** (die sogenannten "*Container*") von großer Bedeutung, die aber bei uns nicht Stoff sind.

  Deshalb heißt die im C++ festgelegte Sammlung von Funktionen, Klassen und Methoden **STL** (*"Standard Templates Library"*).