Database Systems II B: DBMS-Implementation

Anhang C: Mehr C++ für Java-Programmierer

Prof. Dr. Stefan Brass
Martin-Luther-Universität Halle-Wittenberg
Wintersemester 2021/22

http://www.informatik.uni-halle.de/~brass/dbi21/

Inhalt

Definition von Templates (1)

- Definition des Templates für einfache Listenknoten:

```
template<class T> class List {
    private:
        T elem;
        List* next;
    public:
        List(T e) { elem = e; next = 0; }
        void set_next(List* l) { next = l; }
        T get_elem() { return elem; }
        List* get_next() { return next; }
};
```

Syntaktisch besteht ein Klassentemplate aus einer normalen Klassendeklaration, der das Schlüsselwort "`template`" und dann in spitzen Klammern die Parameter vorangestellt sind.

Definition von Templates (2)

- Typparameter werden in der Parameterliste durch das Schlüsselwort "`class`" gekennzeichnet.

  Man kann dafür aber später beliebige Typen einsetzen, nicht nur Klassen. Z.B. auch den Typ `int`, was in Java nicht möglich wäre. Insofern passt das Schlüsselwort nicht ganz. Inzwischen kann man alternativ "`typename`" schreiben (kein semantischer Unterschied, eventuell Probleme bei älteren Compilern).

- In der Klassendeklaration kann der Typparameter `T` wie ein normaler Typ verwendet werden.

- Im Innern der Klassendeklaration ist es egal, ob man "`List`" oder "`List<T>`" schreibt.

  Einzige Ausnahme ist die Definition der Template-Klasse selbst.
  In "`template<class T> class List {`" darf man nur "`List`" schreiben.
  Dort definiert man ja gerade das Template.

Definition von Templates (3)

- Falls man in der Klassendeklaration den Rumpf für eine Funktion nicht angegeben hat, muss man die Definition über ein weiteres Template nachholen:

```
template<class T>
    void List<T>::set_next(List* l)
        { next = l; }
```

  In der Festlegung der Klasse (links vom `::`) bei der nachträglichen Definition einer Funktion ist der Parameter nötig (man muss `List<T>` schreiben).

- Der Compiler muss allerdings wissen, für welche konkreten Element-Typen `T` Code erzeugt werden soll. Daher steht häufig die ganze Template-Definition in einer `.h`-Datei.

  Wenn man (wie im obigen Beispiel) einen Teil der Template-Definition in eine cpp-Datei schreibt, muss man hier explizit angeben, welche Instanzen man braucht, z.B. "`template class List<int>;`".

Template-Instanziierung (1)

- Man instanziiert das Template, indem man einen konkreten Typ für den Typ-Parameter einsetzt:

```
int main()
{   List<int>* tmp = new List<int>(3);
    List<int>* first = tmp;
    List<int>* last = tmp;
    tmp = new List<int>(5); // Element anhängen
    last->set_next(tmp);
    last = tmp;
    for(List<int>*p = first; p; p=p->get_next())
        cout << p->get_elem() << "\n";
    return 0;
}
```

Wenn man für einen Template-Parameter einen Typ einsetzt, der selbst durch Instanziierung eines Templates entsteht, muss man "> >" schreiben, nicht >>.

Template-Instanziierung (2)

- In der Template-Deklaration können für Werte des Typ-Parameters zunächst beliebige Operatoren, Methoden, Funktionen benutzt werden.
- Bei der Instanziierung kann es dann zu einem Fehler kommen, wenn ein Typ eingesetzt wird, der diese Funktionen etc. nicht hat.

  Es ist eine Schwäche von C++, dass man Anforderungen an Template-Parameter in der Template-Deklaration nicht explizit angeben kann.
- Man kann sich den Template-Mechanismus also wie eine Art Macro vorstellen.

  Bei der Template-Definition wird im wesentlichen nur der Text abgespeichert. Bei der Instanziierung werden die Parameter ausgefüllt und erst dann findet die eigentliche Compilierung und Prüfung statt.

Template-Instanziierung (3)

- Da es etwas mühsam ist, den Parameter immer explizit anzugeben, kann man mit `typedef` einen Namen für diese Instanz der Templateklasse einführen:

```
typedef List<int> intlist;
```

`typedef` erzeugt in C++ keinen neuen Typ (nur Abkürzung).

- Wenn man Deklaration des Templates `list.h` und Definition längerer Methoden `list.cpp` trennen will, kann man für jede nötige Instanziierung, z.B. `T = int`,
  - eine Datei "`intlist.h`" anlegen mit zwei Zeilen:

```
#include "list.h"
typedef List<int> intlist;
```

  - außerdem eine Datei "`intlist.cpp`":

```
#include "list.cpp"
template class List<int>;
```

Templates: Ausblick

- Es sind nicht nur Typ-Parameter möglich, sondern z.B. auch ganze Zahlen (Konstanten) und Adressen von globalen Objekten und Funktionen.

```
template<class T, int Max> class Stack {
        T elems[Max];
        int num_elems;
        ...
```

- Templates sind nicht nur für Klassen möglich, sondern auch für Funktionen:

```
template<class T> T maximum(T n, T m)
    { if(n > m) return n; else return m; }
```

Während man bei der Verwendung von Klassen-Templates Werte für die Parameter explizit angeben muss, bestimmt der Compiler bei Funktionstemplates wie `max` den Parameter `T` aus dem Aufruf (soweit als möglich, ggf. angeben).