3.1 Funktionstemplates

3.1.1 Syntax

```
template <class T1, ..., class Tk> //formale Typ-Parameterliste (alternativ typename)
T FUNC(“Parameterliste”, hier müssen T1,..., Tk vorkommen!)
{

    T1, ..., Tk kann vorkommen

}
```

Ergebnistyp darf T1, ..., Tk sein oder beliebiger Typ.

3.1.2 Beispiele (Kandidaten: vordefinierten Makros in C)

Min/Max

1. Versuch: Definition

```
template <class T>

    T max (T a, T b)
    { return (a < b) ? b: a; }
```

2. Versuch

```
template <class T>

    const T & max(const T & a, const T & b)
```

Verwendung

```
    int x = 17;
    int y = 7;
    int z = max(x,y)
```

Compiler instanziiert das max-Template für T = int falls es keine normale max-Funktion für int gibt.

Restriktionen

1. hier x,y müssen gleichen Typ haben (Parameterliste).
2. Im Rumpf der Funktion müssen die für T verwendeten Operationen für den aktuellen Typ (int) definiert sein.

template <class T>

```
void swap (T & a, T & b)
    {
        T tmp = a;
        a=b;
        b=tmp;
    } //im Sortieralgorithmus verwendet
```

Algorithmen als Funktionstemplates

```
template <classT>
    void quicksort (T*A, int l, int r)
    {//sortiert A[l...r] bzgl ‘‘<=’’-Ordnung
        ...
        swap (A[l], A[r]); //Aufruf anderer Template-Funktion
        ...
        quicksort (A, l, j-1);
        quicksort(A,j+1, r); //Rekursion
    }
```

Anwendung:

```
double A[100];
...
quicksort(A,0,99); //An dieser Stelle sieht der
//Compiler, dass A vom Typ double * ist
```

Alternativ:

```
template <class T>
void quicksort(T* begin, T* stop)
{
    T* i = begin;
    T* j = stop;
    swap (*begin, j);
}
```

3.2 Klassentemplates

Mechanismus zur Implementierung parametrisierter Datentype, z.B. Stack mit Elementen vom Typ T (T beliebiger Typ).

3.2.1 Definition

```
template <class T1, ..., class Tk>
class A {

    //T1, ..., Tk dürfen als Typ vom
    //Membervariablen (Daten) und in
    //Parameterliste & Rückgabewert
    //von Methoden verwendet werden

}
```

3.2.2 Verwendung

aktueller Typname

```
A <int, double, string, B<int> a>; //Klassentypen
```

Compiler instanziiert die Klasse mit den aktuellen Typ-Parametern →Fehlermöglichkeiten!

3.2.3 Konkrete Beispiele

```
template <class T>
class stack
{

    T* A;  //Feld
    int sz;
    int t;
    public:
    stack (int n)
    {
        A 0 new T[n];
        sz = n;
        t = -1;
    };
    void push (const T & x)
    {
        A[++t] = x;
    }

}
```

3.2.4 Bemerkungen

Argumente in Parameterlisten vom gleichen Klassentyp: mit Template-Klammern

```
stack::stack(const stack<T> & S) ...

    stack <T>& operator = (const stack <T>& S) ...
```

Methoden von Klassen template Methoden von Klassen template, die formale Parameter des Template verwenden, werden wie Funktionstemplates behandelt, insbesondere werden sie wie Funktionstemplates definiert, (falls nicht inline)

```
template <class T>

    class stack s
    ...
    void push (const T&);
    ...
    };
    template <class T>
    void stack<T>::push(const T& x)
    { A[++t] = x; }
    template <class T>
    stack <T> stack <T>::operator = (const stack <T> &);
```

Klassentemplates können Integer-Konstanten als Parameter verwenden.

```
template <int n>

    class int_stack
    {
        int A[n];
        int t;
    template <class T, int n>
    {
        T A[n];
        stack <100> s; //muss eine Konstante sein!
        //(zur Compilezeit bekannt)
        //max. 100 Integer
        stack <int, 100> s;
```

29.06.09

3.3 Klassentemplates → Parametrisierte Datentypen

```
stack.h
template <class T>
class stack
{

    wie int_stack
    Werte vom Typ T

};
```

3.3.1 Verwendung

```
#include <stack.h>
stack <int> s;
stack <double> ...
stack <point> ...
```

Für jeden aktuellen Parameter wird eine Stack-Klasse instanziiert

Andere Beispiele

```
template <class T>
class list
{

    list_elem <T> * first;
    list_elem <T> * last;
    void push (const T & x)
    { ... }

};
```

verwendet Template

```
template <class T>
class list_elem
{

    T value;
    list_elem<T> *succ;
    list_elem<T> *pred;

}
```

3.3.2 Benutzung

```
#include <list.h>
list<int> L;
list<point> ...
```

3.3.3 Fortgeschrittene Datenstrukturen

balancierte Suchbäume, Warteschlangen, Graphen, Hash...

- aufwendige Templateklassen
- Instanziierung erzeugt viel Code
  Aber nur kleiner Teil ist vom Typ-Parameter T abhängig

2.Nachteil Kompletter Quellcode öffentlich

Lösung Kombination von Templates + Ableitung

Beispiel Suchbäume →dictionary<K,I> (Schlüsseltyp, Info-Typ)

Wichtig Verschiedene Instanten von dictionary<K,I> unterscheiden sich nur im Vergleich von 2 Schlüsseln. Der Rest der Programm-Codes ist identisch.

Idee Verwende Ableitung bin_tree_node (point, polygon, circle) Beliebiger Typ →Übung

```
template <class K, class I>
class dictionary ::public bin_tree
{
    //definiere lineare Ordnung
    int cmp_node (bin_tree_node * p, bin_tree_node * q)
    {
        K* a = (K*)p;
        K* b = (K*)p;
        return compare (*a, *b);
    }; //compare muss für K definiert sein
    public
    void insert (K* p, const I& y)
    {
        bin_tree::insert(p, y);
    }
```

3.3.4 Verwendung

```
dictionary <point, int> D;
```

2.Schritt: (Details in der Übung)

beliebige Schlüssel & Informationstypen (ohne explizite Ableitung vom Knotentyp)

```
base_node
template <class K, class I>
class bin_tree_node : public base_node
{

    //K key;
    //I int;

}
```

bin_tree wie vorher: arbeitet mit Klassen vom Typ base_node

```
bin_tree
template <class K, class I>
class dictionary: public bin_tree

    cmp_nodes

void insert (const K & X, const I & y)
{

    new bin_tree_node (x,y);
    bin_tree::insert(p);

}
```

- Schlüsseltyp ist hier beliebiger Typ
- muss nicht von einem Klassentyp abgeleitet sein.

Details in der Übung. Analog für Listen usw.

Dies ermöglicht die Realisierung von Software-Bibliotheken mit parametrisierten Datentypen.

3.3.5 2 Beispiele

Standard-Template library STL Reine Template-Bibliothek von einfachen Datenstrunkturen & Algorithmen.

Verwendet für die Algorithmen das sogenannte Iterationsinterface.