Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

main.c

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
    int i;
    list_t *li, *ld;

    li = create(2, sizeof(int));    // list of int-values !!!
    for (i = 1; i <= 10; i++)
        append(li, &i);

    for (i = 0; i < 20 && !getError(li); i++) {
        void *val = getValueAt(li, i);

        if (getError(li) == 0)
            printf("%d: %d\n", i, *(int *)val);
    }
    destroy(li);
```

```
    double f;

    ld = create(2, sizeof(double)); // list of double-values !!
    for (i = 1, f = 1.25; i <= 10; i++, f += 0.25)
        append(ld, &f);

    for (i = 0; i < 20 && !getError(ld); i++) {
        void *val = getValueAt(ld, i);
        double fval = *(double *)val;

        if (getError(ld) == 0)
            printf("%d: %f\n", i, fval);
    }
    destroy(ld);

    return 0;
}
```

Damit wir Werte eines beliebigen Datentyps in der Liste speichern können, definieren wir ein dynamisches Array, dass an jedem Index einen Zeiger auf `void` speichert.

liste.h

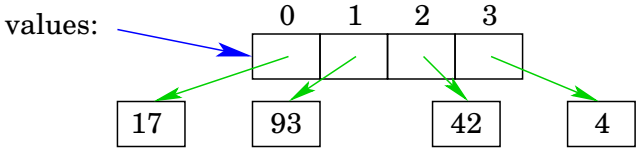
```
/*
 * incomplete data type
 */
typedef struct list list_t;

/*
 * interface
 */
list_t * create(int nmemb, int esize);
void append(list_t *l, void *val);
void * getValueAt(list_t *l, int pos);
char getError(list_t *l);
void destroy(list_t *l);
```

Anmerkungen:

- Der Parameter `val` der Funktion `append` ist nun vom Typ `Zeiger auf void`.
- Passend dazu ist der Typ der Funktion `getValueAt` nun ebenfalls `Zeiger auf void`.
- Der Funktion `create` muss nun nicht nur die initiale Größe des Arrays übergeben werden, es muss auch die Größe der einzelnen, zu speichernden Elemente übergeben werden.

  Dies ist notwendig, damit die Funktion `append` entsprechend viel Speicherplatz für die Kopie des zu speichernden Wertes allokieren kann.

Der Typ des Attributes `values` ist nun Zeiger auf Zeiger auf `void`, denn es soll ein dynamisches Array (erster Zeiger) erzeugt werden, dass an jedem Index einen Zeiger auf `void` (zweiter Zeiger) speichert.



Ein Zeiger auf `void` hat auf jedem Rechner eine feste Größe, daher kann die Größe des Speichers, die das Array belegt, berechnet werden.

liste.c

```c
#include <stdlib.h>
#include <string.h>
#include "liste.h"

struct list {
    void **value;
    int nmemb;
    int next;
    char error;
    int esize;
};

static void increase(list_t *l) {   // private
    l->nmemb *= 2;
    l->value = (void **) realloc(l->value,
            l->nmemb * sizeof(void *));
}
```

```
list_t * create(int nmemb, int esize) {
    list_t *l;

    l = (list_t *) malloc(sizeof(list_t));
    l->next = 0;
    l->nmemb = nmemb;
    l->esize = esize;
    l->error = 0;
    l->value = (void **) calloc(nmemb, sizeof(void *));

    return l;
}

char getError(list_t *l) {
    return l->error;
}
```

```
static char isFull(list_t *l) {      // private
    return l->next == l->nmemb;
}

void append(list_t *l, void *val) {
    void *elem;

    if (isFull(l))
        increase(l);

    elem = malloc(l->esize);
    memcpy(elem, val, l->esize);

    l->value[l->next] = elem;
    l->next += 1;
}
```

Die Funktion `append` prüft zunächst, ob noch Platz im Array vorhanden ist und allokiert ggf. mehr Speicher durch aufrufen der Funktion `increase`.

Danach wird Speicher allokiert, um eine Kopie des zu speichernden Wertes ablegen zu können. Die Kopie wird mittels der Funktion `memcpy` aus der Standard-Bibliothek erzeugt, die byte-weise den Speicher, auf den `val` zeigt, an die Stelle kopiert, auf die `elem` zeigt.

Übung: Warum wird nicht einfach der Zeiger `val`, also die Speicheradresse, auf die `val` zeigt, gespeichert? Warum wird Speicherplatz allokiert und eine Kopie erzeugt?

Die Antwort kommt später.

```
void * getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->next) {
        l->error = 1;
        return NULL;
    }

    return l->value[pos]; // oder Kopie liefern?
}

void destroy(list_t *l) {
    int i;

    for (i = 0; i < l->next; i++)
        free(l->value[i]);

    free(l->value);
    free(l);
}
```

Wenn wir aus der Funktion append (wie oben) *keine* Kopie des Wertes zurück geben, wird das Prinzip der Datenkapselung verletzt:

```
.....
int main(void) {
    int i;
    list_t *l = create(2, sizeof(int));

    .....
    for (i = 0; i < 20 && !getError(l); i++) {
        void *val = getValueAt(l, i);

        if (getError(l) == 0)
            *(int *)val = 42;    // !!!!!!
    }
    .....
}
```

Wie kann eine Kopie zurück gegeben werden?

Im einfachsten Fall ändern wir die Funktion `getValueAt` unserer Liste wie folgt:

```
void * getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->next) {
        l->error = 1;
        return NULL;
    }

    // Kopie erzeugen !!!!!
    void *result = malloc(esize);
    memcpy(result, values[i], esize);

    return result;
}
```

Problem: Wer gibt den Speicher wieder frei, der mit `malloc` allokiert wurde?

Um das Problem der Speicherfreigabe zu vermeiden, legen wir in unserer Struktur ein zusätzliches Attribut `result` an und allokieren in der Funktion `create` Speicher für die Kopie:

```
struct list {
   void **value;
   .....
   int esize;
   void *result;                // neu !!!!!
};

list_t * create(int nmemb, int esize) {
   list_t *l= (list_t *) malloc(sizeof(list_t));
   .....  // wie bisher
   l->result = malloc(esize);  // neu !!!!!
   return l;
}
```

Wir belegen den Speicher in der Funktion append und geben den Speicher in der Funktion destroy wieder frei:

```
void *getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->next) {
        .....    // Fehlerbehandlung
    }
    memcpy(l->result, l->values[pos], l->esize);

    return l->result;
}

void destroy(list_t *l) {
    for (int i = 0; i < l->next; i++)
        free(l->value[i]);
    free(l->value);
    free(l->result);    // neu !!!!!
    free(l);
}
```

Alternativ können wir der Funktion einen Zeiger auf einen Speicherbereich übergeben, in dem die Kopie abgelegt werden kann.

```
char getValueAt(list_t *list, int pos, void *res) {
    if (pos < 0 || pos >= list->insPos) {
        list->error = OUT_OF_RANGE;
        return 1;
    }

    list->error = 0;
    memcpy(res, list->values[pos], list->esize);

    return 0;
}
```

Der Aufruf der Funktion `getValueAt` ist auf der folgenden Folie dargestellt.

generische Liste in C

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
    int r, j = 0;
    list_t *l;

    l = create(2, sizeof(int));
    for (int i = 1; i <= 10; i++)
        append(l, &i);

    for (int i = 0; i < 20 && !getError(l); i++) {
        if (0 == getValueAt(l, i, &r))
            printf("%d: %d\n", i, r);
    }
    destroy(l);
    return 0;
}
```

Nur am Rande: Eine echte Datenkapselung lässt sich nicht sicherstellen, man kann die Liste immer mutwillig zerstören, weil wir mit Zeigern arbeiten.

```
#include <stdio.h>
#include "liste.h"

int main(void) {
    int r, j = 0;
    list_t *l = create(2, sizeof(int));

    int *ll = (int *)l;
    for (int *ip = ll - 100; ip < ll + 100; ip += 1)
        *ip = 0;
    .....  // hier ist die Liste kaputt
}
```

In C ist Zeigerarithmetik möglich, dadurch kann der Zeiger beliebig verändert werden. In Sprachen wir Java oder C# können Zeiger nur auf andere Objekte zeigen, nicht beliebig verbogen werden.

Nachteile einer solch generischen Lösung:

- Keine Typsicherheit, da Zeiger vom Typ `void *` mit allen Zeigertypen kompatibel sind. Der Vorteil, dass der Compiler für uns Überprüfungen auf Datentyp-Verträglichkeit vornehmen kann, geht verloren.
- Komplizierte Syntax durch explizite Typumwandlungen (type cast) beim Auslesen der Daten aus der Datenstruktur.

In C++ geht das alles viel eleganter. Bevor wir uns das ansehen, wollen wir kurz etwas aus PE1 wiederholen.

Parametrisierte Makros bieten Funktionalität unabhängig vom konkreten Datentyp.

```
1  #define SWAP(A, B, T)      \
2  {                          \
3      T t = A;               \
4      A = B;                 \
5      B = t;                 \
6  }
7
8  void sort(long *a, int n) {
9      for (int i = 0; i < n; i++)
10         for (int j = i + 1; j < n; j++)
11             if (a[i] > a[j])
12                 SWAP(a[i], a[j], long)
13 }
```

In Zeile 12 ist keine Übergabe per Referenz nötig, weil keine Funktion aufgerufen wird, sondern nur eine textuelle Ersetzung stattfindet. Zeile 12 wird vom Prä-Compiler durch die Zeilen 2 bis 6 ersetzt, deshalb ist auch die Klammerung des Blocks wichtig.

Templateklasse Liste

liste.h

```
#include "exception.h"

template <typename T>       // T ist ein Platzhalter
class Liste {
    T *_values;             // Array für Elemente vom Typ T
    int _next, _size;
    bool isFull();
    int find(T val);        // Wert vom Typ T als Parameter
    void increase();
    void decrease();
public:
    Liste(int size);
    ~Liste();
    void append(T val);     // Wert vom Typ T als Parameter
    T getValueAt(int pos); // Rückgabe-Typ T
    void erase(T val);      // Wert vom Typ T als Parameter
    void toScreen();
};
```

Templateklasse Liste

```
template <typename T>
Liste<T>::Liste(int size) {
    _size = size;
    _next = 0;
    _values = new T[size];
}

template <typename T>
Liste<T>::~Liste() {
    delete[] _values;
}

template <typename T>
T Liste<T>::getValueAt(int pos) {
    if (pos < 0 || pos >= _next)
        throw Exception("out of bounds");
    return _values[pos];
}
```

Templateklasse Liste

```
template <typename T>
void Liste<T>::append(T val) {
    if (isFull())
        increase();

    _values[_next] = val;
    _next += 1;
}

template <typename T>
void Liste<T>::toScreen() {
    for (int i = 0; i < _next; i++) {
        if (i > 0)
            cout << ", ";
        cout << _values[i];
    }
    cout << endl;
}
```

Templateklasse Liste

```
template <typename T>
bool Liste<T>::isFull() {
    return _next == _size;
}

template <typename T>
void Liste<T>::increase() {
    _size *= 2;
    T *tmp = new T[_size];

    for (int i = 0; i < _next; i++)
        tmp[i] = _values[i];

    delete[] _values;
    _values = tmp;
}
```

Templateklasse Liste

```
template <typename T>
int Liste<T>::find(T val) {
    for (int pos = 0; pos < _next; pos++)
        if (_values[pos] == val)
            return pos;
    return -1;
}

template <typename T>
void Liste<T>::decrease() {
    _size /= 2;
    T *tmp = new T[_size];

    for (int i = 0; i < _next; i++)
        tmp[i] = _values[i];
    delete[] _values;
    _values = tmp;
}
```

Templateklasse Liste

```
template <typename T>
void Liste<T>::erase(T val) {
    int pos = find(val);

    if (pos == -1)
        throw Exception("value not found");

    for (; pos < _next -1; pos++)
         _values[pos] = _values[pos + 1];
    _next -= 1;

    if (_next < _size / 4)
        decrease();
}
```

main.cpp

```
#include .....

int main(void) {
    Liste<int> l(4);      // Typ angeben !!!

    for (int i = 1; i <= 20; i++)
        l.append(i);
    l.toScreen();

    try {
        for (int i = 1; i <= 20; i += 2)
            l.erase(i);
        l.toScreen()
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
```

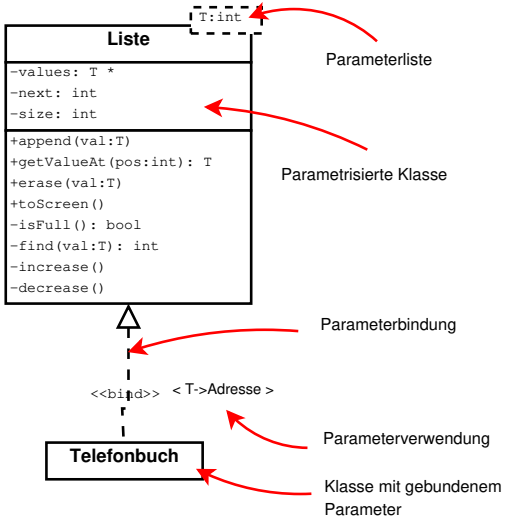
```
    Liste<double> ll(4);      // Typ angeben !!!

    for (double f = 1.25; f <= 5.5; f += 0.25)
        ll.append(f);
    ll.toScreen();

    try {
        for (double f = 1.5; f <= 5.5; f += 0.5)
            ll.erase(f);

        for (int i = 0; i <= 20; i++)
            cout << ll.getValueAt(i) << endl;
    } catch (Exception e) {
        cout << e.toString() << endl;
    }

    return 0;
}
```

Templateklasse Liste

Anmerkungen zur Liste:

- Die Datei `liste.cpp` entfällt.
- Die Listen-Implementierung muss in die Header-Datei! Warum??
- Eine konkrete Liste speichert immer nur einen einzigen Datentyp!

→ *Homogene Datenstruktur*



Templates:

- Parametrisierung von Funktionen und Klassen.
- Beschreibung von Datenstrukturen und Algorithmen unabhängig von bestimmtem Typ. (in C: `void *`)
- Durch Instanziierung konkrete Formulierung erzeugen: An die Stelle des unbestimmten Typs tritt ein konkreter Typ.

Vorteile:

- Instanziierung durch den Compiler.
- Typprüfung zur Compile-Zeit $\rightarrow$ große Typsicherheit
- Die verwendeten Typen müssen nicht auf einer allgemeinen Basisklasse beruhen. (siehe Kapitel Vererbung)

Nachteile:

- Größe des Codes: Jede Schablonen-Instanziierung führt zu weiterem Objekt-Code, anders als bspw. bei Java Generics.

Anwendung:

- Überall dort, wo der Algorithmus nicht von den Daten abhängt.

Anstelle des Schlüsselwortes `typename` kann auch das Schlüsselwort `class` bei der Definition eines Templates verwendet werden. Nur am Rande:

- Stroustrup nutzte `class` um Typen in Templates zu spezifizieren, um kein neues Schlüsselwort einführen zu müssen.
- Um Verwechselungen durch die doppelte Verwendung von `class` zu vermeiden, wurde das neue Schlüsselwort `typename` eingeführt.
- `class` blieb erlaubt, um alte Programme weiterhin kompilieren zu können.
- manchmal: Verwende `class` um auszudrücken, dass hier beliebige Klassen genutzt werden können. Wenn nur Typen wie `int`, `double` oder `char *` genutzt werden sollen, verwende `typename`.

Template der Klassendeklaration

```
template <typename T>
class Liste {
private:
    T *values;
    ...
public:
    ...
    void append(T val);
    T getValueAt(int pos);
};
```

- Der Name `T` für den Platzhalter hat sich eingebürgert.
- Eine Instanziierung erfolgt erst, wenn Objekte vom Typ `Liste<T>` für einen konkreten Typ `T` deklariert werden:

```
Liste<int> iList;
```

Template der Methodenimplementierung

```
template <typename T>
int Liste<T>::getSize() {
    return size;
}

template <typename T>
T Liste<T>::getValueAt(int pos) {
    return values[pos];
}
```

Wichtig: Die Methoden beziehen sich auf die parametrisierte Liste, daher muss `Liste<T>::` vor jeder Methode angegeben werden.

Werden die Methoden direkt bei deren Deklaration implementiert, entfällt die Angabe `template <typename T>` vor jeder Methode.

Funktions-Templates (keiner Klasse zugehörig)

```
template <typename T>
T summe(T *array, int len) {
    T sum = array[0];
    for (int i = 1; i < len; i++)
        sum = sum + array[i];
    return sum;
}
...
int a[] = {1, 1, 2, -3, 2};
int b[] = {11, 2, 3, 2, 7, 5};
double c[] = {1.1, 1.001, -12.8};

cout << summe<int>(a,5) << endl;
cout << summe<int>(b,6) << endl;
cout << summe<double>(c,3) << endl;
```

Die Angabe des Template-Typen bei der Objekterzeugung oder beim Funktionsaufruf kann entfallen, wenn der Typ vom Compiler eindeutig bestimmt werden kann.

```
#include <iostream>
using namespace std;

template <typename T>
T fkt(T a, T b) {
    return a + b;
}
int main(void) {
    int x = 4, y = 5;
    double e = 4.5, f = 6.125;

    cout << x << "+" << y << "=" << fkt(x, y) << endl;
    cout << e << "+" << f << "=" << fkt(e, f) << endl;
  //cout << x << "+" << f << "=" << fkt(x, f) << endl;  // !!
    cout << x << "+" << f << "=" << fkt<int>(x, f) << endl;
    cout << x << "+" << f << "=" << fkt<double>(x, f) << endl;
}
```

Funktions-Templates:

- mehrere Parameter: spezifiziere Template-Argumente in der Reihenfolge, in der die Parameter deklariert sind.

```
template<typename S, typename T>
void func(S x, T y, S* z) { ... }
```

- Spezifizierer wie `inline` oder `friend` stehen hinter `template<...>`
- Template-Funktionen können überladen werden:

```
template<typename T> T methode(T);
template<typename T> T methode(T, int);
```

Die Syntax hat aber Tücken: Folgender Code wird nicht kompiliert!

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4
5  template <typename T, typename S>
6  S summe(map<T, S> &toAdd) {
7      map<T, S>::iterator it = toAdd.begin();
8
9      if (it == toAdd.end())
10         throw "empty map exception";
11
12     S sum = it->second;
13     for ( ; it != toAdd.end(); it++)
14         sum += it->second;
15     return sum;
16 }
```

```
int main(void) {
    map<string, double> aMap;
    aMap["eins"] = 1.0;
    aMap["zwei"] = 2.0;
    aMap["drei"] = 3.0;

    cout << "Summe: " << summe(aMap) << endl;
}
```

Bei der Deklaration des Iterators in Zeile 7 fehlt die Angabe `typename`. Es muss heißen:

```
typename map<T, S>::iterator it = toAdd.begin();
```

Aber warum?

Die folgende Funktion möchte eine innere Klasse `iterator` der Klasse `T` nutzen, um irgendwelche sinnvollen Dinge zu tun:

```
template <typename T>
void foo() {
    T::iterator *iter;
    .....
}
```

Wir gehen also davon aus, dass es eine Klasse wie `Blubb` gibt, die eine innere Klasse `iterator` hat und wir `foo<T>()` mit dieser Klasse parametrisieren möchten:

```
class Blubb {
    class iterator { ..... };
    .....
};
foo<Blubb>();
```

Was wir eher nicht erwarten, ist, dass `iterator` eine Variable der Klasse ist:

```
class BlaBlubb {
    static int iterator;
    .....
};
```

Parametrisieren wir die Funktion `foo<T>()` mit dieser Klasse, dann gibt es ein Problem: Die Zeile

```
T::iterator *iter;
```

wird zu

```
BlaBlubb::iterator *iter;
```

und der Compiler interpretiert `iter` nicht als Zeiger auf eine innere Klasse, sondern die Zeile als Multiplikation, was zu einem Fehler führt!

Was gemeint ist, kann also erst bei der Exemplifizierung (auch Instanziierung genannt) entschieden werden.

Anstatt die Interpretation bis zur Exemplifizierung aufzuschieben, wurde festgelegt:

> Fehlt das Schlüsselwort `typename`, dann werden qualifizierte, abhängige Namen (qualified dependent names) nicht als Typen interpretiert, auch wenn das zu einem Fehler führt.

Ein abhängiger Name ist ein Name, der von einer Parametrisierung (also einem Template) abhängig ist.

Ein qualifizierter Name ist z.B. `std::cout`, wenn wir allerdings mit einer `using`-Direktive arbeiten, ist `cout` ein nicht-qualifizierter Name.

Unsere parametrisierte Liste können wir nutzen, um einen String-Tokenizer zu erstellen.

main.cpp

```
#include <iostream>
#include "tokenizer.h"
using namespace std;

int main(void) {
    string str = "Hans Meier:Gabi Fischer:"
                 "Franz Schulz:Anne Mayer:";
    Tokenizer tok(str, ":;,");

    while (tok.hasMoreTokens()) {
        string mitarb = tok.nextToken();
        cout << mitarb << endl;
        .....
    }
    return 0;
}
```

tokenizer.h

```
#include <string>
#include "liste.h"

class Tokenizer {
private:
    unsigned int _pos;
    Liste<std::string> _tokens;

public:
    Tokenizer(std::string data, std::string separators);

    unsigned int countTokens();
    std::string nextToken();
    bool hasMoreTokens();
};
```

tokenizer.cpp

```
#include "tokenizer.h"
using namespace std;

Tokenizer::Tokenizer(string data, string sep) {
    _pos = 0;

    string::size_type beg, end;
    beg = data.find_first_not_of(sep, 0);
    end = data.find_first_of(sep, beg);

    while (string::npos != beg || string::npos != end) {
        string s = data.substr(beg, end - beg);

        _tokens.append(s);
        beg = data.find_first_not_of(sep, end);
        end = data.find_first_of(sep, beg);
    }
}
```

```
unsigned int Tokenizer::countTokens() {
    return _tokens.size();
}

string Tokenizer::nextToken() {
    return _tokens.getValueAt(_pos++);
}

bool Tokenizer::hasMoreTokens() {
    return _pos < _tokens.size();
}
```

Reicht der Standard-Destruktor? Werden damit auch die Einträge in der Liste gelöscht?

Übung

Damit der obige String-Tokenizer funktioniert, müssen wir unsere Klasse `Liste` um eine Methode erweitern:

- `unsigned int size()` liefert die Anzahl der gespeicherten Elemente

Überlegen Sie, in welchen Fällen der obige String-Tokenizer nicht korrekt funktioniert bzw. sich nicht so verhält, wie man es vielleicht erwartet.