Objektorientierte Anwendungsentwicklung

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Davids

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2018

Inhalt

- *C/C++*
  - aktuelle Programmiersprache für Betriebssysteme, eingebettete Systeme, virtuelle Maschinen, Treiber und Signalprozessoren
  - gute Grundlage für C#, Java, PHP oder Perl
  - aktuell C++11: Unterstützung von Nebenläufigkeit (Threads), Erweiterung der Programmbibliothek um z.B. reguläre Ausdrücke, intelligente Zeiger (smart pointer), ungeordnete assoziative Container, eine Zufallszahlenbibliothek, numerische und mathematische Bibliotheken
- *UML* – Unified Modeling Language
  $\rightarrow$ graphische Darstellung der Systemkomponenten
- *Entwurfsmuster*
  $\rightarrow$ irgendwer hat Ihr (Entwurfs-)Problem schon gelöst
- *Refactoring*
  $\rightarrow$ Design bestehender Software verbessern

Inhalt

Auszug aus *The C++ programming language* von Bjarne Stroustrup:

- You don't have to know every detail of C++ to write good programs.
- Focus on programming techniques, not on language features.
- Don't reinvent the wheel, use libraries.
- Don't believe in magic: understand what your libraries do, how they do it, and at what cost they do it.

Literatur: C

- Brian W. Kernighan, Dennis M. Ritchie:
  Programmieren in C.
  Carl Hanser Verlag.
- Karlheinz Zeiner:
  Programmieren lernen mit C.
  Carl Hanser Verlag.
- Jürgen Wolf:
  C von A bis Z.
  Galileo Computing.

Literatur: C++

- Bjarne Stroustrup:
  The C++ Programming Language.
  Addison-Wesley.
- Martin Schader, Stefan Kuhlins:
  Programmieren in C++.
  Springer Verlag.
- Jürgen Wolf:
  C++ von A bis Z.
  Galileo Computing.
- Stefan Kuhlins, Martin Schader:
  Die C++ Standardbibliothek.
  Springer Verlag.

Literatur: OOA/OOD

- Bernd Oestereich:
  Objektorientierte Software-Entwicklung.
  Oldenbourg Verlag.
- Heide Balzert:
  Lehrbuch der Objektmodellierung.
  Spektrum Akademischer Verlag.
- Bernhard Rumpe:
  Modellierung mit UML.
  Springer Verlag.
- Scott W. Ambler:
  Process Patterns.
  Cambridge University Press.

Literatur: Software-Entwicklung

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
  Entwurfsmuster.
  Addison-Wesley.
- Eric Freeman und Elisabeth Freeman mit Kathy Sierra und Bert Bates:
  Entwurfsmuster von Kopf bis Fuß.
  O'Reilly.
- Martin Fowler:
  Refactoring.
  Addison-Wesley.

Aktuelle Informationen, Sprechzeiten, Folien unter

`http://lionel.kr.hsnr.de/~rethmann/index.html`

Anmerkungen, Korrekturen oder Verbesserungsvorschläge sind immer willkommen! Sprechen Sie mich an oder schicken Sie mir eine E-Mail.

Büro: F 202
E-Mail: `jochen.rethmann@hs-niederrhein.de`

Büro: B 327
E-Mail: `peter.davids@hs-niederrhein.de`

Stellen Sie Fragen! Nur so kann ich beurteilen, ob Sie etwas verstanden haben oder noch im Trüben fischen.

Lernerfolg

Konfuzius:

> Wer fragt, ist ein Narr für eine Minute.
> Wer nicht fragt, ist ein Narr sein Leben lang.

aus `www.lernen-als-weg.de`:

- Entspannen Sie sich. Richten Sie Ihre volle Aufmerksamkeit auf die Veranstaltung.
- Setzen Sie sich Ziele. Was wollen Sie in dieser Veranstaltung lernen?
- Hören Sie aktiv zu. Denken Sie mit und sorgen Sie dafür, dass alle Unklarheiten ausgeräumt werden.
- Notieren Sie Wichtiges. Machen Sie sich Notizen zur Veranstaltung und markieren Sie die wichtigsten Aspekte.
- Formulieren Sie Fragen. Notieren Sie Fragen und bringen Sie diese ein.

Lernerfolg

aus `www.lernen-als-weg.de`:

- Beteiligen Sie sich. Bringen Sie Ihre Anliegen und Ideen ein.
- Haben Sie Geduld. Lernen Sie, andere Ansichten zu akzeptieren. Helfen Sie, andere besser zu verstehen.
- Denken Sie positiv. Werden Sie sich darüber klar, wie die Veranstaltung zu Ihrem Lernerfolg beiträgt.
- Setzen Sie sich weitere Ziele. Entscheiden Sie, was Sie nach der Veranstaltung tun und wie Sie diese vertiefen.
- Handeln Sie schnell. Setzen Sie diese Ziele bald um. Verzögerung ist der erste Schritt zum Vergessen.

Lernerfolg

Der Lernerfolg wird am Ende durch eine Klausur geprüft:

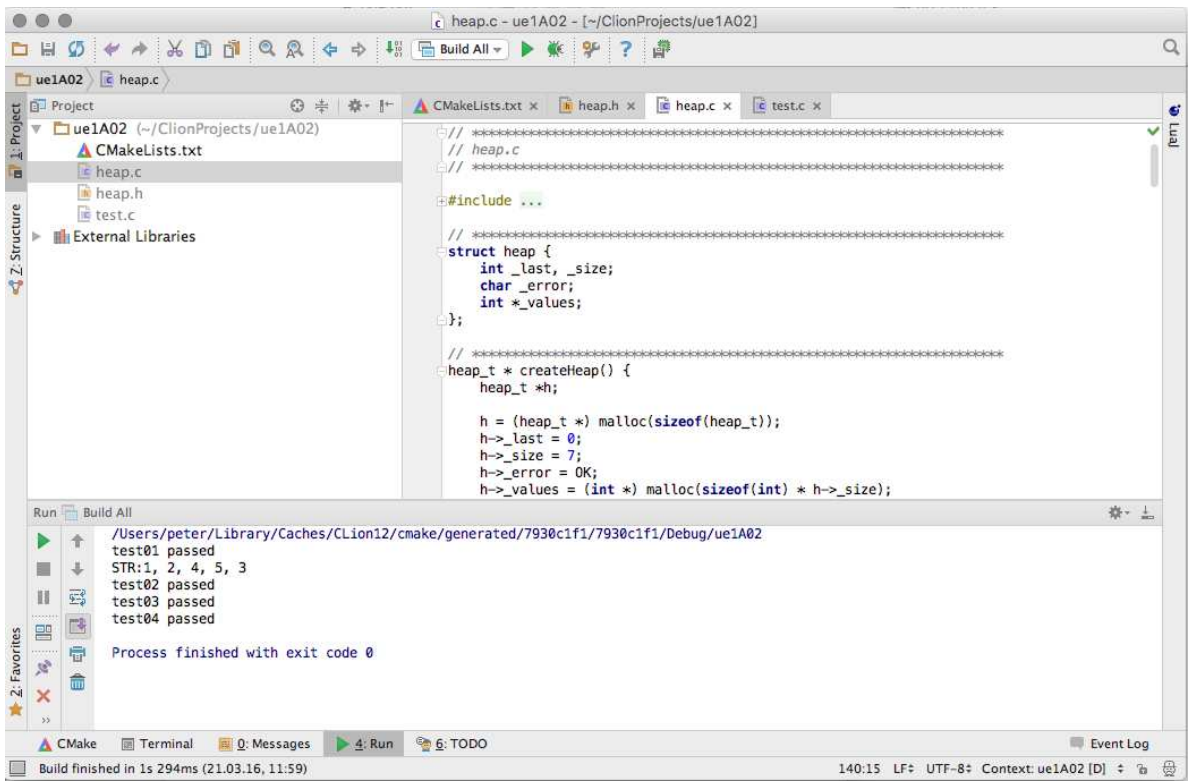
- In der Klausur steht Ihnen kein Computer, keine Online-Hilfe, kein Debugger und kein Compiler zur Verfügung.
- Die Klausursituation ist daher extrem anders als die Situation in der Übung oder dem Praktikum und muss geübt werden.

Bereiten Sie sich auf die Klausur vor, indem Sie Programme zunächst auf einem Blatt Papier entwickeln.

- Gehen Sie die einzelnen Programmschritte durch und vollziehen Sie dabei nach, ob das Programm korrekt ist.
- Implementieren Sie dann das Programm genau so, wie es auf dem Papier steht und kompilieren Sie es.
- Syntaxfehler beim nächsten Programm möglichst vermeiden!
- Nach dem Beseitigen der Syntaxfehler: Programm testen.
- Logische Fehler beim nächsten Programm vermeiden!

CLion

- Plattformunabhängige Programmierumgebung für C/C++
- Installiert unter Linux & Windows: B312, B315 und B322
- Kostenlose Studierenden-Lizenz verfügbar (..@stud.hn.de)
- URL: https://www.jetbrains.com/student



Übersicht

Evolution

- *Strukturierte Programmierung*
- Modulare Programmierung
- Objektorientierte Programmierung

Strukturierte Programmierung

Bei der strukturierten Programmierung werden

- Funktionen und Prozeduren dazu benutzt, Programme zu organisieren, z.B. `sqrt`, `sin`, `printf`, `toString`, ...
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken zur Verfügung gestellt: `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`, ...

Funktionen reduzieren Copy-and-Paste von Programmteilen enorm. Anstelle von

```
if ((d1.jahr > d2.jahr)
        || (d1.jahr == d2.jahr
            && d1.monat > d2.monat)
        || (d1.jahr == d2.jahr
            && d1.monat == d2.monat
            && d1.tag > d2.tag)) {
    ...
}
```

würden wir die Logik eines Datumvergleichs in einer Funktion bereitstellen

```
bool isGreater(date_t a, date_t b) {
    return (a.jahr > b.jahr)
           || (a.jahr == b.jahr
               && a.monat > b.monat)
           || (a.jahr == b.jahr
               && a.monat == b.monat
               && a.tag > b.tag);
}
```

und an den jeweiligen Programmstellen die Funktion aufrufen:

```
if (isGreater(d1, d2)) {
    ...
}
```

→ das Programm wird lesbar: literarisches Programmieren

Strukturierte Programmierung

Literarisches Programmieren bezeichnet das Schreiben von Computerprogrammen in einer Form, sodass sie vor allem für Menschen lesbar sind.

Dies steht im Gegensatz zur konventionellen Ansicht, dass Programme hauptsächlich effizient sein sollen und dann oft nur noch für den Computer lesbar sind.

Jon Bentley fragte in *Communications of the ACM*: „When was the last time you spent a pleasant evening in a comfortable chair, reading a good program?“

aus: `http://de.wikipedia.org/wiki/Literate_programming`

Stellen wir die Funktion dann noch in einer Bibliothek bereit, kann die Funktion sogar projektübergreifend verwendet werden.

In C++ können wir das Ganze durch geeignete Operatorüberladung noch lesbarer schreiben: `if (d1 > d2) ...`

Ziele der strukturierten Programmierung:

- Verständlicher und übersichtlicher Code.
- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code durch allgemeingültige Funktionen oder Makros anstelle von Copy-and-Paste.

Problem: Typisierung

```
int ival[30];
int icmp(const void *, const void *);
...
void qsort(ival, 30, sizeof(int), icmp);
...
int icmp(const void *a, const void *b) {
    int x = *(int *) a;
    int y = *(int *) b;
    return x - y;
}
```

Lösung in C: Zeiger auf `void` bzw. Makros
besser in C++: Templates, Vererbung, Polymorphismus

Wiederholung: C-Makros

`#define name ersatztext` bewirkt, dass im Quelltext die Zeichenfolge `name` durch `ersatztext` ersetzt wird. Der Ersatztext ist der Rest der Zeile.

Makros mit Parametern erlauben, dass der Ersatztext bei verschiedenen Aufrufen verschieden sein kann:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Ein Makroaufruf ist **kein** Funktionsaufruf! Ein Aufruf von `MAX` wird **direkt im Programmtext expandiert**. Die Zeile

```
x = MAX(p + q, r + s);
```

wird ersetzt durch die Zeile

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Textersetzung! Keine Auswertung von Ausdrücken!

Wiederholung: C-Makros

Anders als bei Funktionen genügt eine einzige Definition von `MAX` für verschiedene Datentypen.

Vorsicht: Der Ersatztext muss sorgfältig geklammert sein, damit die Reihenfolge von Bewertungen erhalten bleibt. Überlegen Sie, was beim Aufruf `SQR(z + 1)` des nachfolgenden Makros passiert:

```
#define SQR(x) x * x
```

Makros können bereits definierte Makros enthalten:

```
#define SQR(x) (x) * (x)
#define CUBE(x) SQR(x) * (x)
```

Die Textersetzung erfolgt durch den Prä-Prozessor, also vor dem eigentlichen Übersetzen des Programms. Unter Linux kann mittels `gcc -E test.c` das Übersetzen nach der Prä-Prozessorphase gestoppt werden. Die Ausgabe ist der durch den Prä-Prozessor bearbeitete Quelltext.

Problem: globale Variablen oder lange Parameterlisten

- Programme sind einfacher zu verstehen, wenn sie aus kleinen, in sich geschlossenen, unabhängigen Teilen bestehen.
  - Globale Variablen führen zu voneinander abhängigen Funktionen. Das Ändern einer Funktion kann dazu führen, dass andere Funktionen nicht mehr korrekt funktionieren. Nach jeder Änderung muss man erneut das ganze Programm testen.
  - Übergeben wir alle benötigten Variablen als Parameter an die Funktionen, ergeben sich lange, unklare Parameterlisten.
- Keine Zugriffskontrolle: Bei den heutigen nebenläufigen Programmen ist es wichtig, den gleichzeitigen Zugriff mehrerer Threads auf gemeinsame Variablen zu synchronisieren.
- Namenskonflikte: In umfangreichen Programmen wird oft derselbe Variablenname zweimal verwendet.

Lösung in C: Module, incomplete data type
besser in C++: Klassen, private/protected, Namensräume

Problem: Lesbarkeit und Wartbarkeit

- Vergleichsoperatoren bei allgemeinen Datentypen
  Lösung in C: Funktionen
  besser in C++: Operatorüberladung
- Fehlerbehandlung
  Lösung in C: Fehlerflags als Rückgabewert einer Funktion, globale Fehlervariable `errno`, Signal-Handler
  besser in C++: Exceptions

Gehen wir die Probleme an! Lernen wir mit C++ eine tolle Programmiersprache kennen.

Liste

Oft müssen wir eine Liste von Elementen verwalten:

- Bücherliste in der Bibliothek
- Studentenliste im Prüfungsbüro
- Mitarbeiterliste in der Verwaltung
- KFZ-Liste im Straßenverkehrsamt
- ...

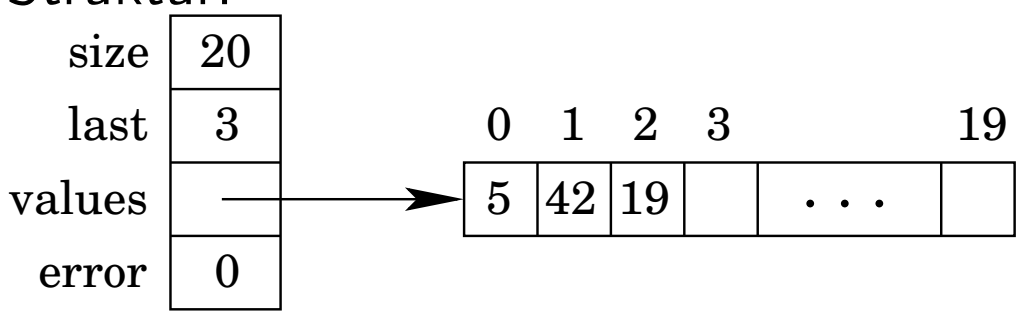
Die Anforderungen an solche Listen sind immer gleich:

- hinzufügen von Werten
- löschen von Werten
- suchen (z.B. Halter des Fahrzeugs KR-AB 123)
- ausdrucken oder anzeigen der Liste
- ...

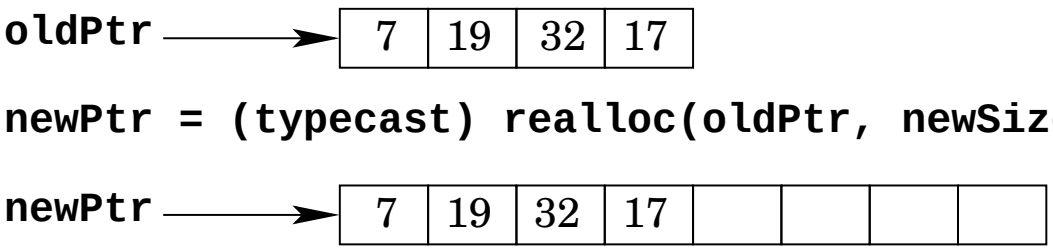
Liste: erster Versuch

Einige Details zu unserer Implementierung:

- Die Liste beruht auf einem Array.
  - Das Array wird bei Bedarf automatisch vergrößert und
  - wird automatisch verkleinert, wenn so viele Elemente aus der Liste entfernt wurden, dass das Array nur noch zu einem Viertel gefüllt ist.
- Damit der Code wiederverwendet werden kann,
  - wurden alle wichtigen Variablen in einer Struktur zusammengefasst
  - und alle Operationen sind als Funktionen ausgeführt.
- Struktur:



Wiederholen wir zunächst kurz, wie ein dynamisch angelegtes Array vergrößert werden kann:



Konnte der alte Speicherbereich nicht vergrößert werden, dann wird neuer Speicherbereich allokiert, die alten Werte in den neuen Speicher kopiert und der alte Speicherbereich frei gegeben. In diesem Fall ist `oldPtr` nicht mehr gültig.

Oft soll unter dem gleichen Namen wie zuvor das Array weiterhin benutzt werden, dann ist `newPtr = oldPtr`:

```
int *dArr = (int *) calloc(sizeof(int), 4);
    ...
dArr = (int *) realloc(dArr, sizeof(int) * 8);
```

Wichtig: Unterscheide Variablen und Strukturattribute!

```
typedef struct {
  int wert;      // Strukturattribut
  char *name;    // Strukturattribut
} foo_t;         // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t a;       // Variable vom Typ foo_t

  a.wert = 15; // Strukturattribut der Variablen
  a.name = (char *) malloc(strlen(h) + 1);
  strcpy(a.name, h);
  ...
}
```

Über eine Struktur kann ein Array gebildet werden!

```
typedef struct {
  int wert;      // Strukturattribut
  char *name;    // Strukturattribut
} foo_t;         // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t a[5];  // Array mit 5x Typ foo_t

  a[0].wert = 15;
  a[0].name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0].name, h);
  ...
}
```

Ein Array kann Zeiger auf Strukturen enthalten!

```
typedef struct {
  int wert;          // Strukturattribut
  char *name;        // Strukturattribut
} foo_t;             // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t *a[5];    // Array: 5x Zeiger auf foo_t

  a[0] = (foo_t *) malloc(sizeof(foo_t));
  a[0]->wert = 15;
  a[0]->name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0]->name, h);
  ...
}
```

Arrays von Strukturen können auch dynamisch angelegt werden!

```
typedef struct {
  int wert;          // Strukturattribut
  char *name;        // Strukturattribut
} foo_t;             // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t *a;          // dynamisches Array

  a = (foo_t *) calloc(sizeof(foo_t), 5);

  a[0].wert = 15;
  a[0].name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0].name, h);
  ...
}
```

Dynamische Arrays können Zeiger auf Strukturen enthalten!

```
typedef struct {
  int wert;    // Strukturattribut
  char *name; // Strukturattribut
} foo_t;       // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t **a;  // dyn. Array mit Zeiger auf foo_t

  a = (foo_t **) calloc(sizeof(foo_t *), 5);

  a[0] = (foo_t *) malloc(sizeof(foo_t));
  a[0]->wert = 15;
  a[0]->name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0]->name, h);
  ...
}
```

Liste: erster Versuch

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int size, last, *values;
    char error;
} list_t;

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;
    return l;
}
```

Liste: erster Versuch

```
char isFull(list_t *l) {
    return l->size == l->last;
}

void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->last] = val;
    l->last += 1;
}
```

Liste: erster Versuch

```
int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}

void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}
```

Liste: erster Versuch

```
void erase(list_t *l, int val) {
    int pos = find(l, val);

    if (pos == -1)
        return;

    for (; pos < l->last -1; pos++)
        l->values[pos] = l->values[pos + 1];
    l->last -= 1;

    if (l->last < l->size / 4)
        decrease(l);
}
```

Liste: erster Versuch

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 1;
        return -1;
    }
    return l->values[pos];
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

Liste: erster Versuch

```
void toScreen(list_t *l) {
    int i;

    for (i = 0; i < l->last; i++)
        printf("%d\n", l->values[i]);
}

int main(void) {
   int i;
   list_t *l;

   l = create();
   for (i = 1; i < 30; i++)
       append(l, i);
   toScreen(l);
```

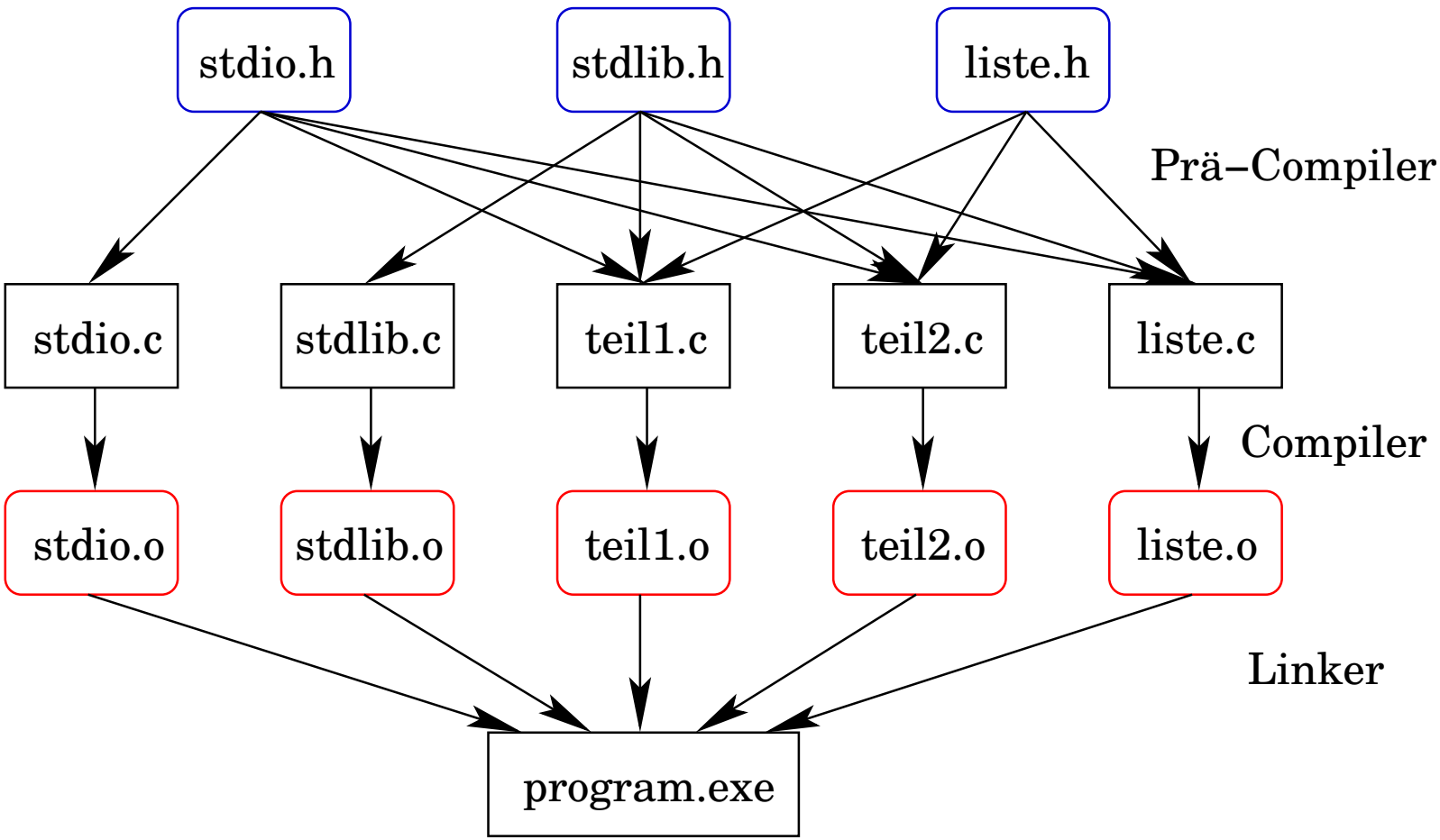
Liste: erster Versuch

```
    for (i = 1; i < 30; i += 2)
        erase(l, i);
    toScreen(l);

    i = getValueAt(l, 20);
    if (l->error == 0)
        printf("value[%2d] = %2d\n", 20, i);
    else printf("20 out of range\n");

    destroy(l);
    return 0;
}
```

Frage: Was halten Sie von der Implementierung?

Aufteilen in Header- und Code-Datei

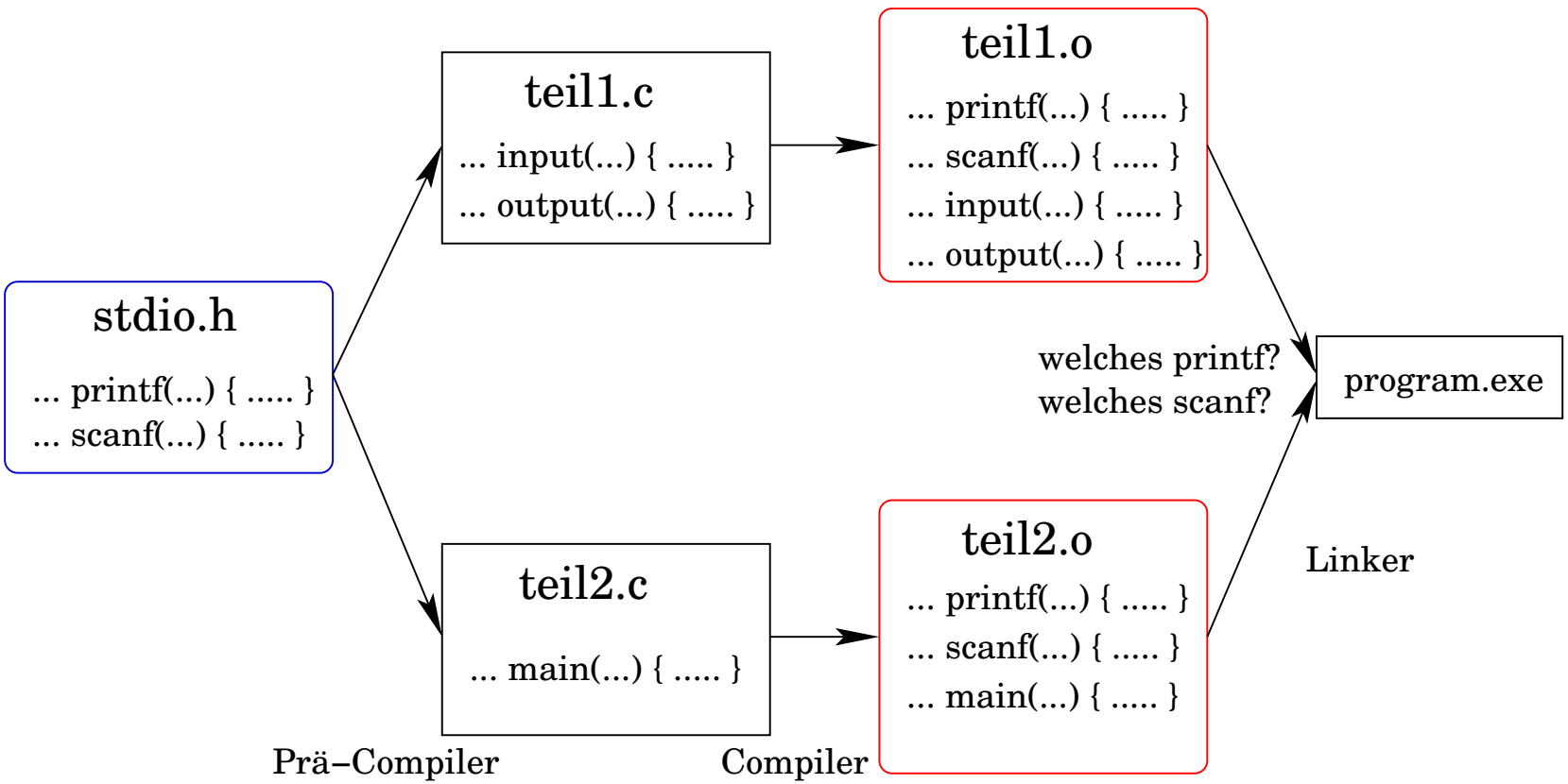
Wiederverwendung ist nur möglich, wenn Deklaration und Implementierung getrennt sind.



Aufteilen in Header- und Code-Datei

Die Implementierung muss von der Deklaration getrennt werden, da sonst beim Linken Probleme auftreten!

Wären bspw. die Prozeduren und Funktionen der `stdio.h` auch innerhalb der `stdio.h` implementiert, gäbe es Probleme:



Aufteilen in Header- und Code-Datei

Daher sind in den Header-Dateien nur Typdefinitionen und Funktionsprototypen enthalten.

Programme würden zum Teil auch ohne Header-Dateien und ohne explizite Typumwandlungen funktionieren, wie an folgendem Beispiel ersichtlich ist:

```
void main(void) {
    int *p = malloc(sizeof(int));
    *p = 42;

    printf("Hallo, Welt!\nint: %d\n", *p);
    free(p);
}
```

Allerdings kann der Compiler dann nicht überprüfen, ob die Parameter und Rückgaben von Funktionen vom richtigen Typ sind.

Aufteilen in Header- und Code-Datei

Wir teilen den Code unserer Liste wie folgt auf:

- `liste.h`: Funktionsprototypen und Datentypen
  Die Header-Datei beschreibt die Schnittstelle der Liste, also den Funktionsumfang, und steht allen interessierten Programmteilen zur Verfügung.
- `liste.c`: Implementierung der Funktionen
  Die Code-Datei implementiert die Listenfunktionalität, enthält also die Funktionsdefinitionen.
- `main.c`: Programm, dass die Liste verwendet
  Die Code-Datei wir nur einmal kompiliert und in einer Bibliothek oder als Objekt-Datei bereitgestellt.
  Der Linker fügt dann die Funktionalität dem eigentlichen Programm hinzu.

Liste: zweiter Versuch

liste.h

```
typedef struct {
    int size, last, *values;
    char error;
} list_t;

list_t *create();
char isFull(list_t *l);
void increase(list_t *l);
void decrease(list_t *l);
void append(list_t *l, int val);
int find(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
void destroy(list_t *l);
```

Liste: zweiter Versuch

liste.c

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

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}

.....
```

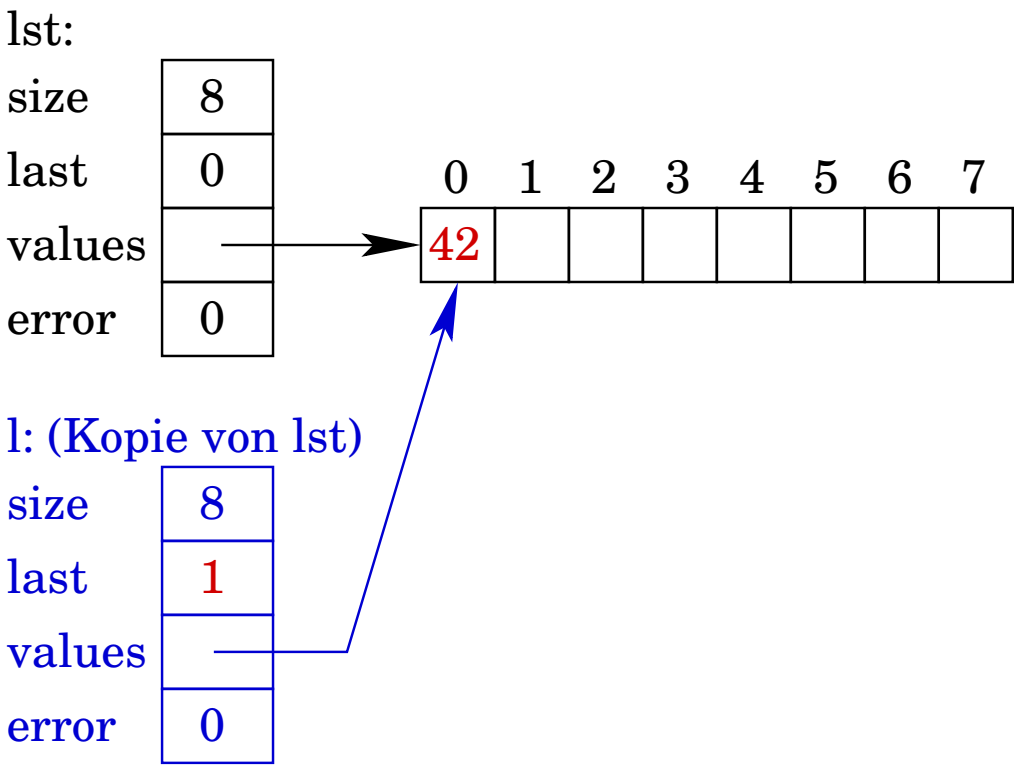
Liste: zweiter Versuch

main.c

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

void main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (l->error != 0)
        printf("value[%2d] = %2d\n", 30, i);
    destroy(l);
}
```

Wiederholung: Call-by-Reference

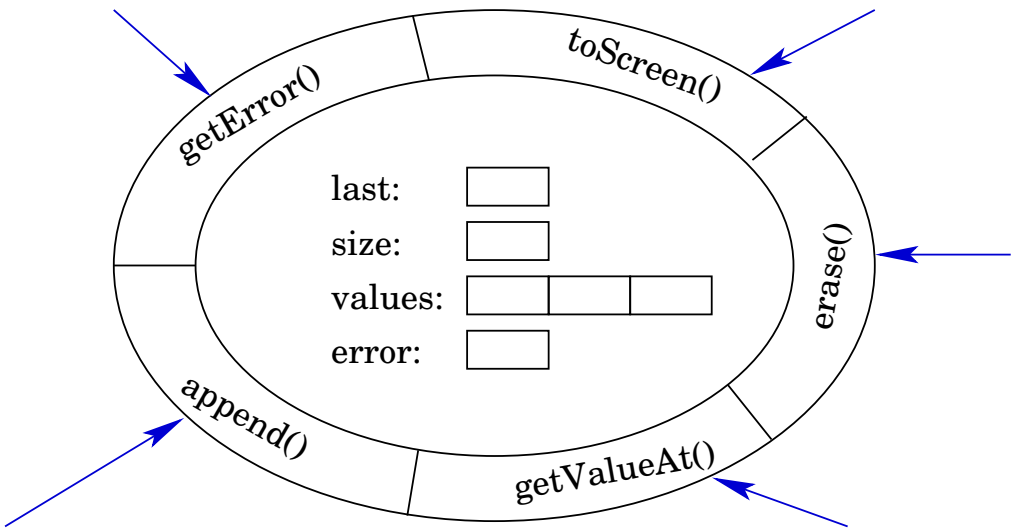
Warum wird die Liste als Zeiger übergeben? Weil sonst eine Kopie erzeugt würde, mit der die Funktionen wie `append(lst, 42)` dann arbeiten, und die Liste beim Aufrufer nicht geändert würde:



Liste: Verbesserungsmöglichkeiten

- Die Funktionen `increase()` und `decrease()` sind öffentlich bekannt, obwohl sie nur innerhalb der Liste verwendet werden und nicht von außen aufrufbar sein sollten.
- Der innere Aufbau der Liste ist durch das `typedef` in der Header-Datei nach außen hin bekannt.

$\Rightarrow$ Keine Datenkapselung! Im Hauptprogramm kann die Funktionalität der Liste durch einen schreibenden Zugriff wie `l->size = 0;` zerstört werden.



Übersicht

Evolution

- Strukturierte Programmierung
- *Modulare Programmierung*
- Objektorientierte Programmierung

Modulare Programmierung

Modulare Programmierung versucht der wachsenden Größe von Softwareprojekten Herr zu werden. Module können einzeln geplant, programmiert und getestet werden.

Universelle Module müssen nur einmal programmiert und können wiederverwendet werden. Je öfter ein Modul wiederverwendet wurde, desto sicherer kann man sein, dass es fehlerfrei ist.

Wenn alle Module erfolgreich getestet sind, können diese Einzelteile logisch miteinander verknüpft und zu einer größeren Anwendung zusammengesetzt werden.

Die modulare Programmierung erweitert den prozeduralen Ansatz, indem *Prozeduren zusammen mit Daten* in logischen Einheiten zusammengefasst werden.

aus: `http://de.wikipedia.org/wiki/Modulare_Programmierung`

In unserem Beispiel haben wir folgendes zu tun, um eine Liste als wiederverwendbares Modul zur Verfügung stellen zu können:

- In `liste.h` die Strukturvereinbarung entfernen und durch einen unvollständigen Typen (Vorwärtsdeklaration) ersetzen.
- In `liste.c` die aus `liste.h` entfernte Strukturvereinbarung aufnehmen.
- Wir müssen die Schnittstelle erweitern: Methoden zum Zugriff auf interne Variablen definieren, die im Hauptprogramm benötigt werden: `getError()`
- Datenkapselung sicherstellen, indem wir mit `static` die Funktionen „verstecken“, die nach außen nicht sichtbar sein sollen: `isFull()`, `increase()`, usw.

Liste: dritter Versuch

liste.h

```
// ================================
// incomplete data type
// (forward declaration)
// ================================
typedef struct list_s list_t;

// ================================
// interface
// ================================
list_t *create();
void append(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
char getError(list_t *l);     // neu !!!
void destroy(list_t *l);
```

Liste: dritter Versuch

liste.c

```
#include ....

struct list_s {
    int size, last, *values;
    char error;
};

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}
```

Liste: dritter Versuch

```
// private !
static void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static char isFull(list_t *l) {
    return l->size == l->last;
}
```

Liste: dritter Versuch

```
void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->last] = val;
    l->last += 1;
}

// private !
static int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
```

Liste: dritter Versuch

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 2;
        return -1;
    }
    return l->values[pos];
}

.....

char getError(list_t *l) {      // neu !
    return l->error;
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

Liste: dritter Versuch

main.c

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

int main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (getError(l) == 0)
        printf("value[%2d] = %2d\n", 30, i);

    destroy(l);
    return 0;
}
```

Kommt Ihnen diese Art der Programmierung fremd vor? Finden Sie diese Art der Programmierung seltsam und zu kompliziert?

Wir kennen diese Art der Programmierung bereits aus C von den Dateioperationen. In der Header-Datei `stdio.h` ist definiert:

```
typedef  struct _IO_FILE  FILE;
```

In unseren Programmen konnten wir Funktionen darauf nutzen:

```
FILE *f;
f = fopen("dat.txt", "rw"); // vgl. create()
fprintf(f, ...);            // vgl. append()
fscanf(f, ...);             // vgl. getValueAt()
fgets(..., f);
fclose(f);                  // vgl. destroy()
```

Liste: dritter Versuch

Unsere Liste sieht doch schon ganz gut aus!

In C++ werden Module als Klassen realisiert. Klassen sind Grundelemente in der objektorientierten Programmierung.

Schauen wir es uns an!

Übersicht

Evolution

- Strukturierte Programmierung
- Modulare Programmierung
- *Objektorientierte Programmierung*

Liste in C++

liste.h

```
class Liste {
private:     // nicht sichtbar
     int _size, _last, *_values;
     char _error;
     bool isFull();    // neuer Datentyp
     int find(int value);
     void increase();
     void decrease();

public:      // sichtbar
     Liste(int size); // Konstruktor statt create
     ~Liste();        // Destruktor statt destroy
     void append(int val);
     int getValueAt(int pos);
     void erase(int val);
     void toScreen();
     char getError();
};
```

Liste in C++

liste.cpp

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

Liste::Liste(int size) {
    _size = size;
    _last = 0;
    _error = 0;
    _values = new int[size];    // statt malloc
}

Liste::~Liste() {
    delete[] _values;           // statt free
}
```

Liste in C++

```
void Liste::increase() {
    int *tmp = new int[_size * 2];

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

    delete[] _values;
    _values = tmp;
    _size *= 2;
}

void Liste::append(int val) {
    if (isFull())
        increase();

    _values[_last] = val;
    _last += 1;
}
```

Liste in C++

```
int Liste::getValueAt(int pos) {
    if (pos < 0 || pos >= _last) {
        _error = 1;
        return -1;
    }
    return _values[pos];
}

int Liste::find(int val) {
    int pos;

    for (pos = 0; pos < _last; pos++)
        if (_values[pos] == val)
            return pos;
    return -1;
}
```

Liste in C++

```
bool Liste::isFull() {
    return _last == _size;
}

void Liste::decrease() {
    _size /= 2;
    int *tmp = new int[_size];

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

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

Liste in C++

```
void Liste::erase(int val) {
    int pos = find(val);

    if (pos == -1)
        return;

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

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

Liste in C++

```
void Liste::toScreen() {
    for (int i = 0; i < _last; i++)
        cout << i << ": " << _values[i] << endl;
}

char Liste::getError() {
    return _error;
}
```

Anmerkung zu der Notation:

- Die Variablen, die mit einem Unterstrich beginnen, sind Attribute der Klasse, also Klassenvariablen.
- Im Unterschied dazu beginnen die lokalen Variablen und Parameter von Methoden immer mit einem Buchstaben.
- Dies ist eine eigene Notation und weder normiert noch in irgendwelchen Richtlinien empfohlen.

Liste in C++

main.cpp

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

int main(void) {
    Liste l(10);

    for (int i = 1; i < 60; i++)
        l.append(i);
    l.toScreen();

    cout << endl;
    for (int i = 10; i < 60; i++)
        l.erase(i);
    l.toScreen();

    return 0;
}
```