Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Nachdem wir unsere Liste getestet haben, können wir feststellen: Unsere Liste sieht schon ganz gut aus!

In C++ werden Module als Klassen realisiert.

Klassen sind Grundelemente in der objektorientierten Programmierung.

Anstelle von `static` nutzen wir `private`, anstelle von `extern` nutzen wir `public`, um Attribute und Methoden einer Klasse nach außen sichtbar oder unsichtbar zu machen.

Es gibt neue Bibliotheken.

Schauen wir es uns an!

liste.h

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

public:     // von außen 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.cpp

```
#include <iostream>        // neue Bibliothek
#include "liste.h"
using namespace std;       // nutze Namensraum

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

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

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

Liste in C++

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

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

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

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

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

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

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

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

    if (pos == -1)
        return;

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

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

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

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

Anmerkung zu der Notation:

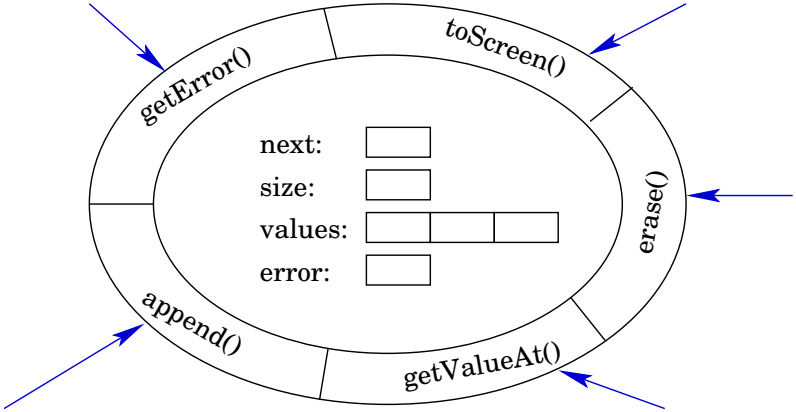
- Die Variablen, die mit einem Unterstrich beginnen, sind Attribute der Klasse, also Klassenvariablen.
- Im Unterschied dazu beginnen die lokalen Variablen und Parameter von Methoden immer mit einem Buchstaben.
- Dies ist eine eigene Notation, die weder normiert ist noch in irgendwelchen Richtlinien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

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;
}  // Blockende: Destruktor für l wird automatisch aufgerufen
```

- Klassen definieren einen neuen Datentyp und enthalten:
  - Datenelemente (auch Variablen oder Attribute genannt)
  - Elementfunktionen (auch Methoden genannt)
  - eingebettete Typen (z.B. andere Klassen)
  - Elementkonstanten (`enum`)
- Eine Klasse ist ein Bauplan für gleichartige Objekte.
- Im Allgemeinen beginnen Klassennamen mit einem Großbuchstaben und sind Substantive. Microsoft Windows: Klassennamen beginnen mit einem C für Class, Attribute beginnen mit `m_` für Member.
- Der Klassenname kann wie jeder andere vordefinierte Datentyp benutzt werden.
- Die Klassendeklaration alleine belegt keinen Speicherplatz. Sie legt nur die Struktur fest, nach der der Compiler ein Objekt der Klasse erzeugt.
- Jedes Objekt (auch Exemplar, fälschlicherweise oft auch Instanz genannt) hat seine eigenen Variablen. Ausnahme: `static` deklarierte Attribute
- Initialisierung von Variablen ist innerhalb einer Klassendeklaration nicht möglich, dies erfolgt im Konstruktor. (anders ab C++17)

Wichtiges Prinzip bei der Programmierung allgemein und bei objektorientierter Programmierung im Speziellen: Datenkapselung!



Die Attribute sind vor direkten Zugriffen von außen geschützt, ein Zugriff kann nur über die öffentlichen Methoden erfolgen.

Angabe von Zugriffsrechten: `public` und `private`

- Auf `public`-Membervariablen und -funktionen darf von jeder Stelle eines Programms zugegriffen werden, also insbesondere von anderen Klassen.
- Auf `private`-Elemente darf nur von den Methoden einer Klasse selber zugegriffen werden. Nach außen hin sind diese Variablen und Funktionen unsichtbar. Ausnahme: Freunde dürfen auch auf private Elemente zugreifen.
- Das Sichtbarkeitsattribut `protected` wird im Abschnitt *Vererbung* behandelt.

A private member of a class can only be accessed by the members (and friends) of that class, regardless of whether the members are on the same or different instances:

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

class C {
    int _x;     // implizit private !!!

public:
    C(int x) {
        _x = x;
    }
    void output(C c) {
        cout << "c.x: " << c._x << ", x: " << _x << endl;
    }
};

int main(void) {
    C a(10), b(15);
    a.output(b);
}
```

Geltungsbereichsoperator (scope resolution operator):

- Trennung von Deklaration und Implementation mittels `::`.
- Zugriff auf Membervariable, die den gleichen Namen wie lokale Variable oder Parameter besitzt, oder mittels `this`.
- Innnerhalb einer Methode statt auf ein Member auf eine globale Variable/Funktion gleichen Namens zugreifen.

```
void klassenname::methode(int x, int y) {
    var1 = 10;      // Zugriff auf Membervariable
    x = 20;         // Zugriff auf Parameter x
    ::var1 = 30;    // Zugriff auf globale Variable
    ::x = 40;       // Zugriff auf globale Variable
    func();         // Zugriff auf Memberfunktion
    ::func();       // Zugriff auf globale Funktion
}
```

Unified Modeling Language

- Heutiger Standard der Darstellung der objektorientierten Sichtweise eines realen Problems.
- Basis sind die Klassen (bzw. Objekte) mit Attributen und Methoden eines Systems.
- Das statische Modell beschreibt die Zusammenhänge zwischen den Klassen.
- Das dynamische Modell beschreibt die Zustände und Abläufe innerhalb des Systems.
- Grafische Darstellung eines komplexen Systems zur Reduzierung der Komplexität.

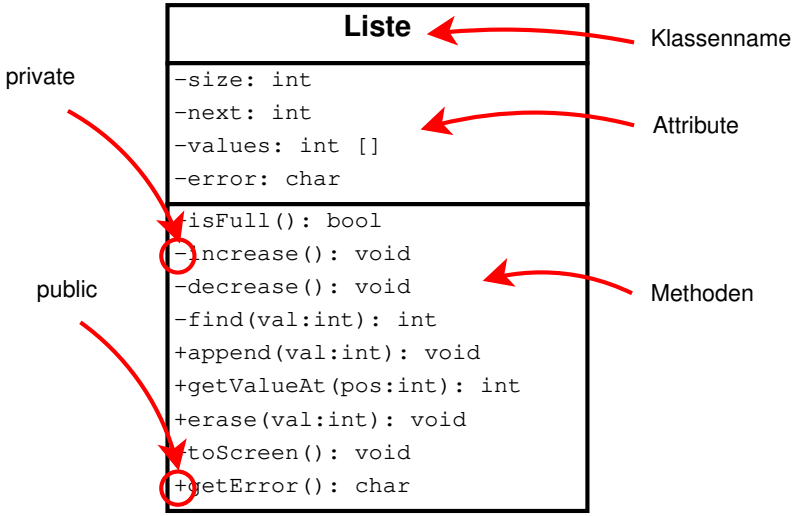
  „The art of programming is the art of organizing complexity.“ (Edsgar W. Dijkstra)

Eine Klasse wird in UML als Rechteck dargestellt. Das Rechteck enthält jeweils einen Bereich für

- den Namen der Klasse,
- die Attribute der Klasse und
- die Methoden der Klasse.

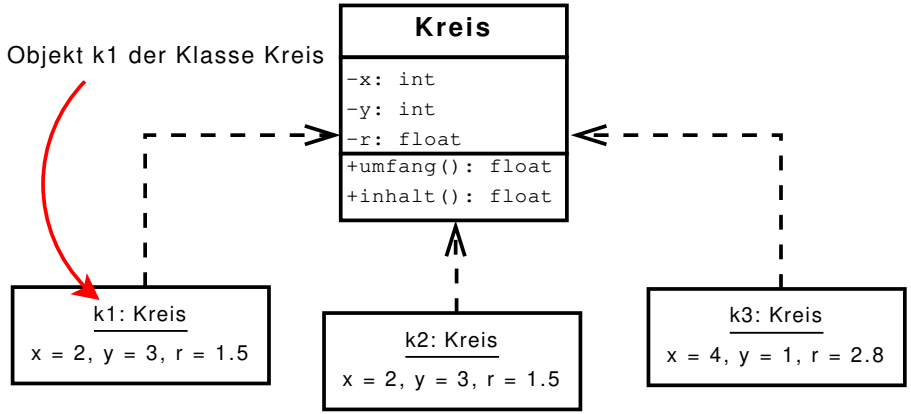
Die Sichtbarkeit der Attribute wird gekennzeichnet mit

- `-` für `private`
- `+` für `public`
- `#` für `protected` (später)
- `~` für `package` (nicht in C++)


Liste
Klassenname
private
-size: int
-next: int
-values: int []
-error: char
Attribute
+isFull(): bool
-increase(): void
-decrease(): void
-find(val:int): int
+append(val:int): void
+getValueAt(pos:int): int
+erase(val:int): void
+toScreen(): void
+getError(): char
public
Methoden

Variablen eines Objekts

Eine Klasse ist ein Bauplan für gleichartige Objekte. Jedes anhand eines solchen Bauplans erstelltes Objekt hat seinen eigenen Satz von Variablen.



Die Objekte k1 und k2 sind gleich, aber nicht identisch! Oder anders gesagt: k1 und k2 sind die gleichen Objekte, aber nicht dieselben.

Variablen eines Objekts

Wann zwei Objekte als gleich angesehen werden, bestimmt der Programmierer.

- `Konto:` gleiche Kontonummer und Bankleitzahl
- `Student:` gleiche Hochschule und Matrikelnummer

In C++ kann dazu der Operator `==` überladen werden, in Java würde dazu die `equals`-Methode überschrieben werden.

Variablen eines Objekts

Der C++-Compiler generiert bei jedem Methodenaufruf die Übergabe eines Zeigers auf das Objekt und setzt in der Methode vor jede Zustandsvariable diesen Zeiger:

```
Liste l(10);                    Liste l(10);
l.append(i);          ⟶         Liste::append(&l, i);
```

Unsere Methode append entspricht also:

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

    l->_values[l->_next] = val;
    l->_next += 1;
}
```

Beachte: So setzt es der Compiler um, programmiert wird anders!

Variablen eines Objekts

Der Zeiger ist eine implizite Objektreferenz. Explizit ist diese Referenz durch das Schlüsselwort `this` ansprechbar.

Unsere Methode `append` können wir auch so schreiben:

```
void Liste::append(int val) {
    if (this->isFull())
        this->increase();

    this->_values[this->_next] = val;
    this->_next += 1;
}
```

Das Schlüsselwort `this` wird überall dort automatisch vom Compiler eingefügt, wo es eindeutig ist. Im obigen Beispiel kann `this` entfallen.

Variablen eines Objekts

Im folgenden Beispiel ist es nicht eindeutig und muss angegeben werden:

```
class C {
private:
    int var;

public:
    C(int var) {
        this->var = var;
    }

    void fkt(int var) {
        cout << "Attribut: " << this->var;
        cout << ", Parameter: " << var << endl;
    }
};
```

Konstruktor

- Der Konstruktor hat den gleichen Namen wie die Klasse.
- Der Konstruktor ist eine typlose – also auch nicht void! – Methode.
- *default-Konstruktor:* Konstruktor ohne Argumente

  Wurde kein Konstruktor programmiert, wird ein default-Konstruktor vom Compiler hinzugefügt.
- Wenn eine Klasse einen Konstruktor besitzt, wird jedes Klassenobjekt vor seiner ersten Verwendung durch den Konstruktoraufruf initialisiert.
- Innerhalb eines Konstruktors können auch Methoden der Klasse aufgerufen werden.
- Der Vollständigkeit halber: Ein Konstruktor darf nicht als `const`, `static` oder `virtual` spezifiziert werden. Was das heißt, sehen wir später.

- bisher: Initialisieren der Datenelemente durch Zuweisung im Rumpf des Konstruktors.

```
Liste::Liste(int size) {
    _size = size;
    _next = 0;
    _values = new int[size];
}
```

- jetzt: Konstruktor kann zwischen Parameterliste und Funktionsrumpf eine *Initialisiererliste* enthalten.

```
Liste::Liste(int size): _next(0), _size(size) {
    _values = new int[size];
}
```

Sollen mehrere Attribute in der Initialisiererliste initialisiert werden, so werden die Einträge mittels Komma getrennt.

Auch die eingebauten Datentypen wie `int` oder `double` haben einen Konstruktor. Diesem Konstruktor kann ein Wert übergeben werden, der initial zugewiesen wird:

```
int *ip = new int(17);
int *dynArr = new int[13];
```

- Die erste Anweisung erstellt einen Zeiger `ip` auf einen einzigen `int`-Wert, der mit 17 initialisiert wird.
- Der Zeiger `dynArr` zeigt auf einen Speicherbereich, in dem 13 Werte vom Typ `int` gespeichert werden können.

  Für jeden der 13 Werte wird der Standard-Konstruktor aufgerufen, der jeden einzelnen Wert initialisiert. Der Standard-Konstruktor muss also existieren!

Wird ein Array von Objekten dynamisch erzeugt, so wird für jeden Wert im Array der Standard-Konstruktor der entsprechenden Klasse aufgerufen.

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

class Foo {
   int _a, _b;          // implizit private !!!
   int *_x, *_y;
public:
   Foo() {
      _a = 1; _b = 2;
      _x = nullptr; _y = nullptr;
   }
   void toScreen() {
      cout << "a:" << _a << " b:" << _b << endl;
      cout << "x:" << _x << " y:" << _y << endl;
   }
};
```

```
int main(void) {
    Foo *p = new Foo[3];

    for (int i = 0; i < 3; i++)
        p[i].toScreen();
    return 0;
}
```

Als Ausgabe wird erzeugt:

```
a:1 b:2
x:0 y:0
a:1 b:2
x:0 y:0
a:1 b:2
x:0 y:0
```

Default Parameterwerte: Der Konstruktor kann, wie jede andere Methode auch, Default-Werte für die formalen Parameter haben.

- In der Header-Datei:

```
Liste(int size = 18);
```

- In der Anwendung:

```
.....
Liste l;          // Liste mit size = 18;
Liste ll(5);      // Liste mit size = 5;
.....
```

- Wird der Konstruktor mit Parameter aufgerufen, erhält die entsprechende Variable den Wert des Parameters, ansonsten wird die Variable mit dem vordefinierten Wert belegt.

Destruktor

Löschen von Objekten

- globale Objekte implizit bei Programmende
- lokale Objekte implizit bei Prozedur- oder Blockende
- dynamisch erzeugte Objekte mit `delete`-Operator

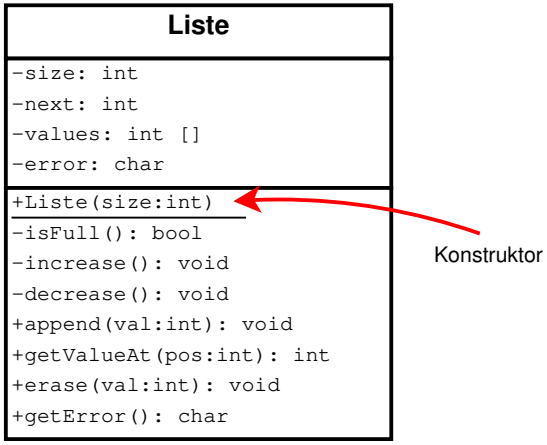
Destruktoren

- Zweck: Aufräumarbeiten beim Löschen eines Objektes.
- Der Name des Destruktors ist *Tilde* + *Klassenname*. In unserem Beispiel der Klasse `Liste` also `~Liste()`.
- Automatischer Aufruf, wenn ein Objekt zerstört wird.
- Ergebnistyp ist ungenannt, also auch nicht `void`, die Parameterliste ist leer.
- Ein default-Destruktor wird vom Compiler hinzugefügt, falls kein Destruktor programmiert wurde.

Welche Ausgabe erzeugt das Programm?

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

class B {
public:
    B() {
        cout << "Jetzt geht's los!" << endl;
    }
    ~B() {
        cout << "Jetzt ist Schluss!" << endl;
    }
} b;    // Definition einer globalen Variablen!

int main(void) {
    cout << "Hello, World!" << endl;
}
```

In dieser Vorlesung: Konstruktoren werden in der Form `Klassenname(Argumente)` dargestellt und unterstrichen.



Achtung: Entspricht nicht dem UML-Standard!

Strukturen in C++ sind spezielle Klassen:

- Eine Struktur kann Methoden enthalten.
- Es sind Zugriffsbeschränkungen mittels `public`, `private` und `protected` möglich.
- `this`: Zeiger auf Strukturobjekt selbst.
- Die Memberfunktionen können überladen werden und default-Parameter besitzen.
- Es können Konstruktoren und Destruktoren definiert werden.

  Standardmäßig wird sowohl ein default-Konstruktor als auch ein default-Destruktor bereitgestellt.
- Sind wie Klassen parametrisierbar, siehe Abschnitt Templates.

Im Gegensatz zu einer C-Struktur ist kein `typedef` notwendig.

Unterschied zu Klassen: Alle Daten und Methoden einer Struktur sind per default `public`, in Klassen `private`.

C:

```
struct elem {
    int value;
    struct elem *next;
};

int main(void) {
    struct elem a, b;

    a.value = 1;
    a.next = &b;
    b.value = 2;
    b.next = NULL;
}
```

C++:

```
struct elem_t {
    int value;
    elem_t *next;
};

int main(void) {
    elem_t a, b;

    a.value = 1;
    a.next = &b;
    b.value = 2;
    b.next = nullptr;
}
```

Wenn man in C nicht `struct elem` schreiben möchte, kann man mittels `typedef` einen neuen Typen definieren.

C ohne typedef:

```
struct elem {
    int value;
    struct elem *next;
};

int main(void) {
    struct elem a, b;

    a.value = 1;
    a.next = &b;
    b.value = 2;
    b.next = NULL;
}
```

C mit typedef:

```
typedef struct elem {
    int value;
    struct elem *next;
} elem_t;  // Name des Typs

int main(void) {
    elem_t a, b;

    a.value = 1;
    a.next = &b;
    b.value = 2;
    b.next = NULL;
}
```

Statische Attribute

bisher:

- Attribute sind die Zustandsvariablen der Objekte einer Klasse.

→ Jede Zustandsvariable ist an ein Objekt gebunden und daher nur mit dem Objekt existent.

statische Attribute:

- Durch `static` sind die Attribute nicht objektbezogen.
- Die Werte sind für alle Objekte einer Klasse gleich.
- `static`-Attribute müssen (außer `const static`) außerhalb der Klassendeklaration mit einem Anfangswert definiert werden.
- Sie haben eine durchgehende Lebensdauer.

Statische Attribute

Anwendung: Um globale, objektunabhängige Daten zu definieren.

- Grundgebühr bei Telefonanschlüssen
- fortlaufende Nummern (vgl. Sequenz in SQL)
- Entwurfsmuster Singleton (später)

bisher: Wir würden dem Konstruktor einer Klasse `Konto` den Wert des Attributs `nr` als Parameter übergeben. Das Konto erhält damit die entsprechende Kontonummer von außen. Irgendwo im Programm muss die fortlaufende Nummer verwaltet werden.

```
class Konto {
    .....
    Konto::Konto(string inhaber, int nr, int pin) {
        stand = 0;
        this->inhaber = inhaber;
        this->nr = nr;            // !!!!!!!!!!!!!!!
        this->pin = pin;
    }
};
```

jetzt: Die Klasse `Konto` ist für die fortlaufende Vergabe der Kontonummern verantwortlich.

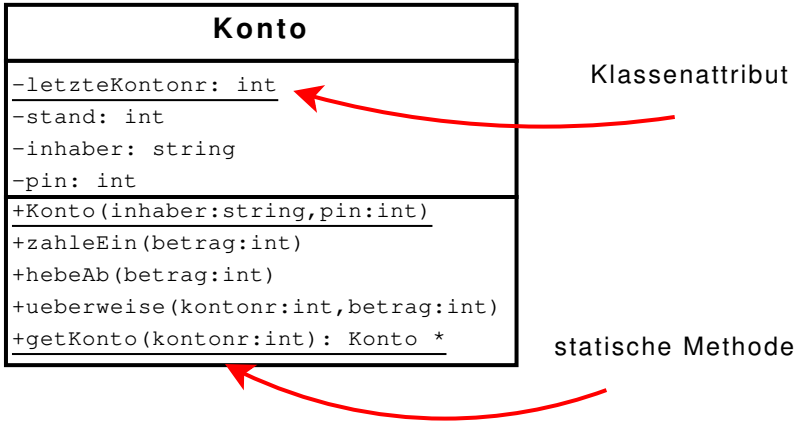
konto.h

```
class Konto {
private:
    static int neueKontonr;
    ....
public:
    Konto(string inhaber, int pin) : stand(0) {
        this->inhaber = inhaber;
        nr = Konto::neueKontonr++;    // !!!!!!!
        this->pin = pin;
    }
    ...
};
```

konto.cpp

```
int Konto::neueKontonr = 1;
```

Statische Attribute und Methoden werden unterstrichen.



Anmerkung: Die Methode `getKonto` soll zu einer Kontonummer das entsprechende Konto-Objekt liefern. Daher kann die Methode nicht mit einem Objekt der Klasse `Konto` aufgerufen werden und ist deshalb als statisch zu deklarieren.

Statische Methoden werden oft anstelle von Konstruktoren verwendet:

```
#include <ctime>
.....
class Date {
    int _day, _month, _year;
public:
    Date(int d, int m, int y) : _day(d), _month(m), _year(y) {}

    static Date getCurrentDate() {   // !!!!!
        time_t now = time(0);
        tm *today = localtime(&now);
        return Date(today->tm_mday, today->tm_mon + 1,
                    today->tm_year + 1900);
    }
    .....
};
```

In statischen Methoden steht implizite Objektreferenz `this` nicht zur Verfügung, da die Methode anhand des Klassennamens, nicht anhand eines Objekts aufgerufen wird.