Objektorientierte Anwendungsentwicklung

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Davids

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2018

Übersicht

Grundlagen C++

- *Klassen und Objekte*
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Klassenkonzept in C++

- Klassen definieren einen neuen Datentyp.
- Klassen 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.

  Klassennamen bei Microsoft Windows beginnen meist mit einem C für Class. Die Attribute beginnen mit `m_` für Member und unterscheiden sich damit von lokalen Variablen und Parametern von Methoden.

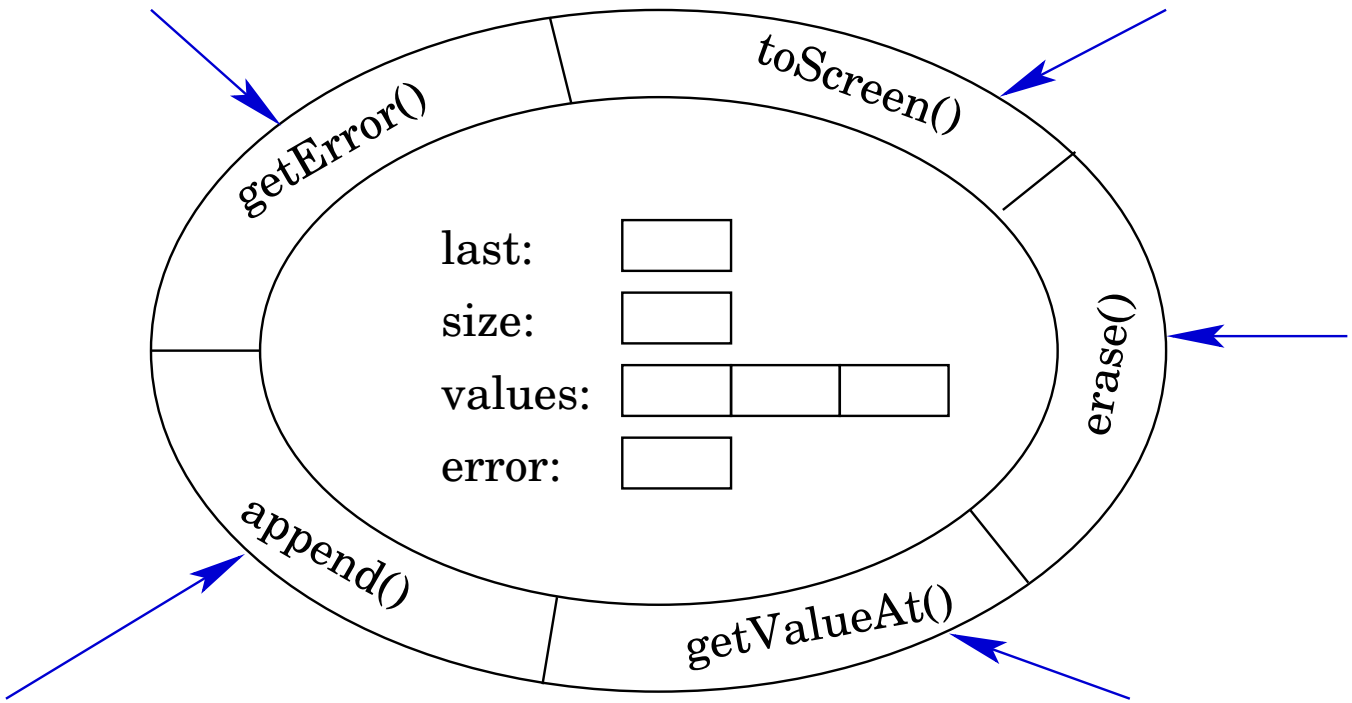
Klassenkonzept in C++

- 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.

Klassenkonzept in C++

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.

Klassenkonzept in C++

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.
- Das Sichtbarkeitsattribut `protected` wird im Abschnitt *Vererbung* behandelt.

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)

Modellierung mit UML

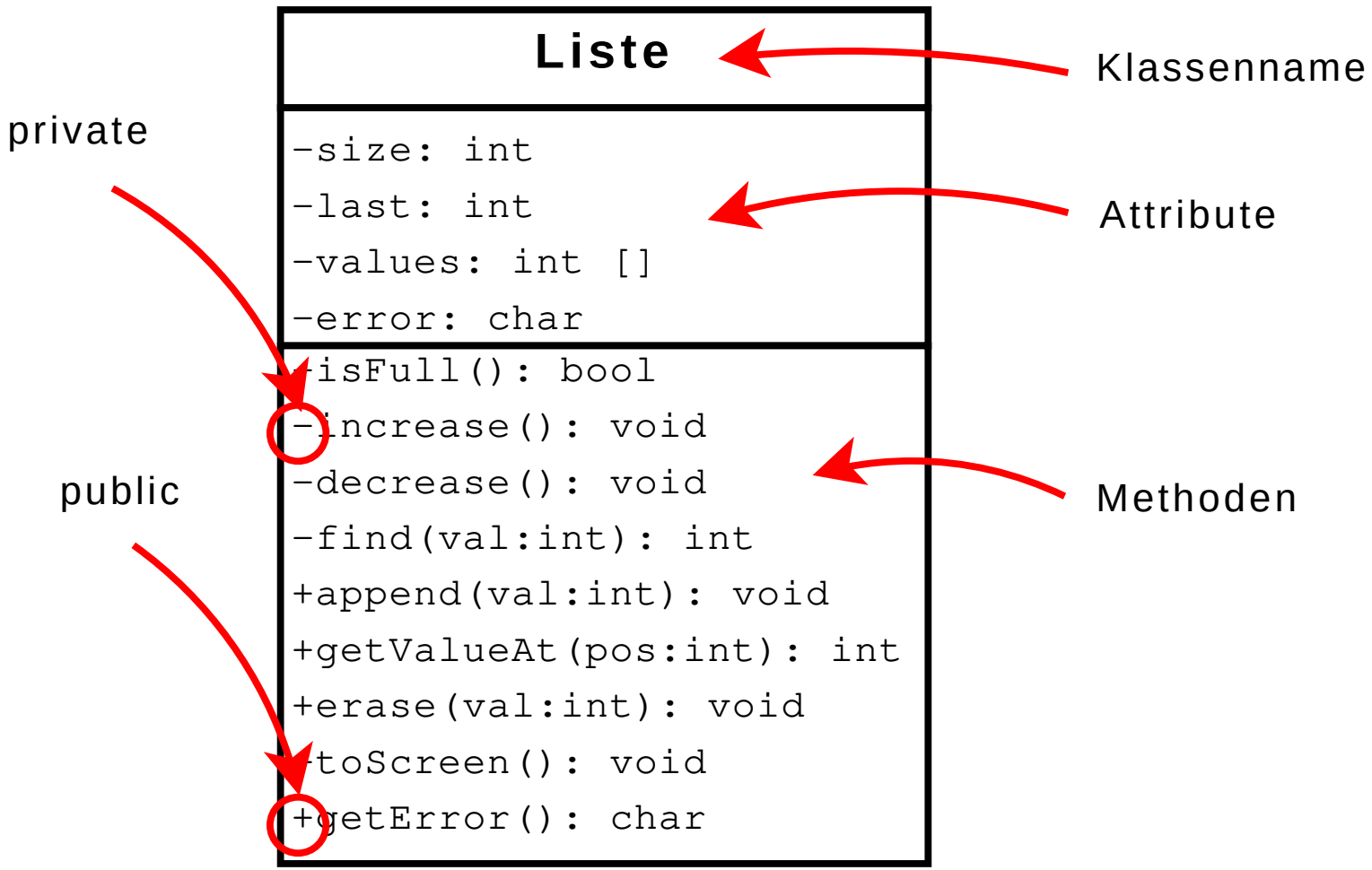
Eine Klasse wird in UML als Rechteck dargestellt. Das Rechteck enthält jeweils einen Bereich für

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

Die Sichtbarkeit der Attribute wird gekennzeichnet mit

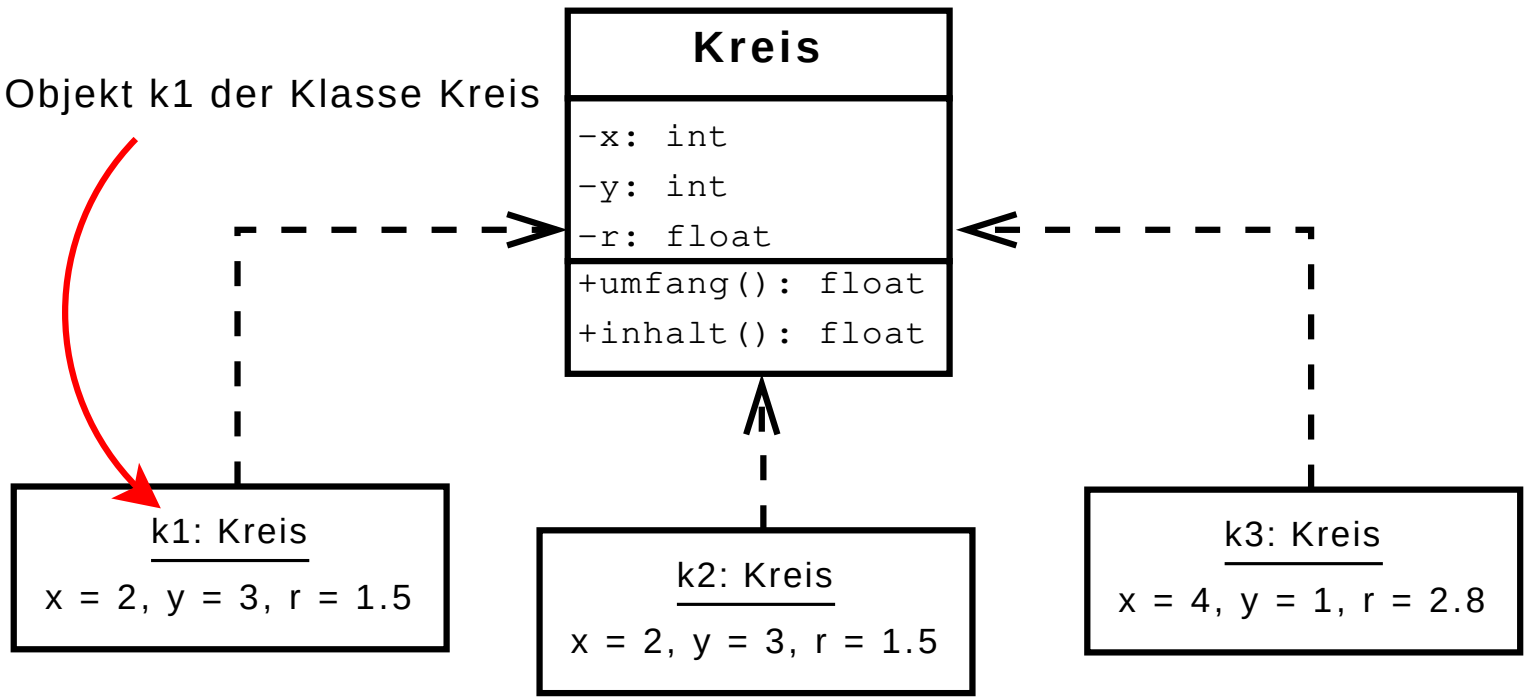
- `-` für `private`
- `+` für `public`
- `#` für `protected` (später)
- `~` für `package` (nicht in C++)

Modellierung mit UML



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 Java wird dazu die Methode `equals` überschrieben, in C++ kann der Operator `==` überladen 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 (isFull(l))
        increase(l);

    l->_values[l->_last] = val;
    l->_last += 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->_last] = val;
    this->_last += 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;
    }
};
```

Übersicht

Grundlagen C++

- Klassen und Objekte
- *Konstruktor und Destruktor*
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

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.

Konstruktor

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

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

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

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

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

Konstruktor

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.

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

```
class Foo {
    int _a, _b;
    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;
    }
};
```

Konstruktor

```
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
```

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;
}
```

Default Parameterwerte: Der Konstruktor kann wie jede andere Methode 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

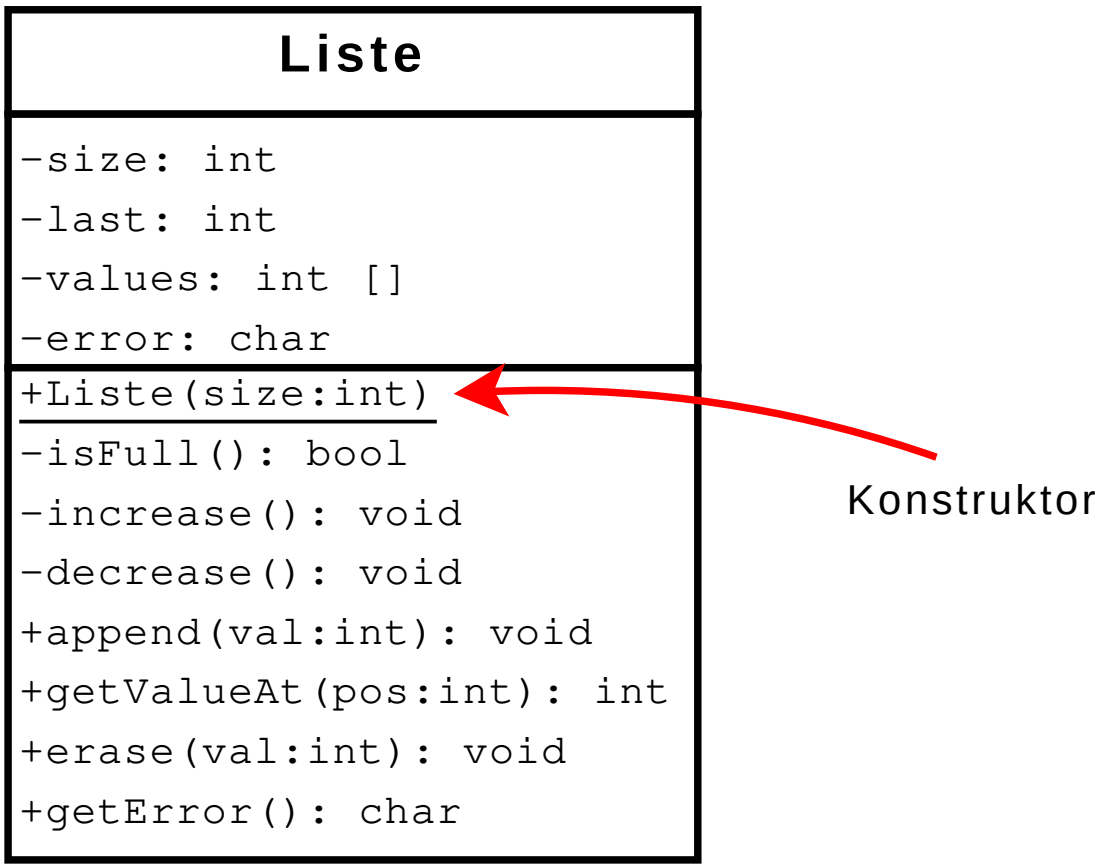
- globale Objekte implizit bei Programmende
- lokale Objekte implizit bei Prozedur- oder Blockende
- dynamisch erzeugte Objekte mit `delete`-Operator

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, die Parameterliste ist leer.
- Ein default-Destruktor wird vom Compiler hinzugefügt, falls kein Destruktor programmiert wurde.

Konstruktor und Destruktor in UML

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



Achtung: Entspricht nicht dem UML-Standard!

Wiederholung: Lokale Variablen und Zeiger

Betrachten Sie folgende C-Funktion. Was berechnet die Funktion und welcher Programmierfehler versteckt sich hier?

```
char *itoa(unsigned int val) {
    char res[12];
    int pot = 1;
    int i = 0;

    // Spezialfall val = 0 behandeln
    if (val == 0) {
        res[0] = '0';
        res[1] = '\0';
        return res;
    }
    .....
```

Wiederholung: Lokale Variablen und Zeiger

```
    while (pot <= val)
        pot *= 10;
    pot /= 10;

    while (pot > 0) {
        int ziffer = val / pot;

        res[i] = ziffer + '0';
        val -= ziffer * pot;
        pot /= 10;
        i += 1;
    }
    res[i] = '\0';

    return res;
}
```

Abgesehen vom Fehler: Ist die Funktion verständlich geschrieben?

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- *Strukturen*
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

struct in C++

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.

struct in C++

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

alt:

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

neu:

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

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- *Wichtige Klassen*
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Include-Dateien von C nutzen

Alte Bibliotheken aus C können in C++ weiterhin verwendet werden, allerdings wird beim Einbinden die Endung `.h` weggelassen, und stattdessen vor den Namen der Bibliothek der Buchstabe `c` voran gestellt.

Beispiele:

- aus `#include <math.h>` wird `#include <cmath>`
- aus `#include <stdlib.h>` wird `#include <cstdlib>`

Übersicht

Wichtige Klassen:

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

die Klasse iostream

Input-Streams ermöglichen Eingaben über die Tastatur,
Output-Streams ermöglichen Ausgaben auf dem Bildschirm.

Sie werden über die Operatoren << zur Ausgabe auf dem Bildschirm und >> zum Einlesen von der Tastatur angesprochen:

```
#include <iostream>      // keine Endung .h !

int main(void) {
    int a;

    std::cout << "Bitte int-Wert eingeben: ";
    std::cin >> a;       // kein Adressoperator !

    char c = 'a';
    std::cout << c << " = " << a << std::endl;
    return 0;
}
```

die Klasse iostream

Damit wir nicht überall `std::` schreiben müssen, können wir die `using namespace`-Anweisung benutzen:

```
#include <iostream>
using namespace std;          // damit kein "std::"

int main(void) {
    int a;

    cout << "Bitte int-Wert eingeben: ";
    cin >> a;

    char c = 'a';
    cout << c << " = " << a << endl;
    return 0;
}
```

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

Formatierte Ausgabe: `iomanip`

- `setw`: konstante Breite festlegen
- `setfill(char)`: Zeichen zum Auffüllen festlegen
- `setbase`: Basis der Zahl festlegen (oktal, dezimal oder hexadezimal)
- `setprecision`: Anzahl der Nachkommastellen festlegen
- `fixed`: Fließkommadarstellung
- `scientific`: Exponentialdarstellung

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setbase(8) << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << i << endl;
}
```

Ausgabe:

```
1
12
144
1750
23420
303240
3641100
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setfill('_') << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << setw(10) << i << endl;
}
```

Ausgabe:

```
_________1
________10
_______100
______1000
_____10000
____100000
___1000000
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setprecision(5) << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.1
0.01
0.001
0.0001
1e-05
1e-06
1e-07
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setprecision(5) << fixed << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.10000
0.01000
0.00100
0.00010
0.00001
0.00000
0.00000
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << scientific << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
1.000000e-01
1.000000e-02
1.000000e-03
1.000000e-04
1.000000e-05
1.000000e-06
1.000000e-07
```

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

Strings

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

int main() {
    string t("Ene mene muh und raus bist du!");

    cout << t << endl;
    cout << "size: " << t.size() << endl;
    cout << "substr: " << t.substr(13, 8)
         << endl;

    size_t pos = t.find("muh");
    cout << "substr: " << t.substr(pos) << endl;

    return 0;
}
```

Strings

```
#include <string>
#include <iostream>

int main() {
    std::string s("Ene mene muh");
    std::string t(" und raus bist du!");
    std::string u(" noch lange nicht");

    s.append(t);                // oder:  s += t;
    std::cout << s << std::endl;

    size_t pos = s.find("!");
    s.insert(pos, u);
    std::cout << s << std::endl;

    s.replace(pos, u.size(), " jetzt doch");
    std::cout << s << std::endl;
}
```

Wo ist der Fehler in folgendem Programm?

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char *s = "Hallo Welt";
    char *t = strtok(s, " ");

    printf("%s\n", t);
    t = strtok(NULL, " ");
    printf("%s\n", t);

    return 0;
}
```

Syntaktisch ist das Programm korrekt:

- Die Funktion `strtok` ändert die übergebene Zeichenkette so, dass an den Trennstellen ein `\0` eingefügt wird, um die Teilzeichenkette zu erzeugen.
- Aber `"Hallo Welt"` ist eine Programmkonstante, also ein konstante Zeichenkette, die in einem vom Betriebssystem besonders geschützten Speicherbereich liegt.

  Das Programm kann zwar übersetzt werden, aber bei der Ausführung kommt es zu einem Speicherzugriffsfehler.

Wie kann das obige Problem ganz einfach gelöst werden?

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

int main(void) {
    char s[] = "Hallo Welt";    // !!!!!
    char *t = strtok(s, " ");

    printf("%s\n", t);
    t = strtok(NULL, " ");
    printf("%s\n", t);

    return 0;
}
```

Wenn wir statt des Zeigers ein Array verwenden, können wir die spezielle Art der Initialisierung von Zeichenketten nutzen. Da `s` als Array auf dem Stack liegt, wird die Programmkonstante kopiert. Der Code dafür wird vom Compiler erzeugt.

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

File I/O

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

int main() {
    fstream f;

    f.open("test1.dat", ios::app | ios::out);
    if (f.is_open())
        f.write("Hallo, Welt!\n", 13);
    else cerr << "failed to open test1.dat\n";
    f.close();

    return 0;
}
```

File I/O

Beim Öffnen verschiedene Modi mittels ODER verknüpfen:

- `ios::app` append output
- `ios::ate` seek to EOF when opened
- `ios::binary` open the file in binary mode
- `ios::in` open the file for reading
- `ios::out` open the file for writing
- `ios::trunc` overwrite the existing file

Online-Tutorials:

- `http://www.cplusplus.com/`
- `http://www.cppreference.com/wiki/`

Vereinfachung: Im Konstruktor kann die zu öffnende Datei angegeben werden, außerdem sind Operator-Überladungen definiert.

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

int main() {
    fstream f("test1.dat", ios::app | ios::out);

    if (!f)
        cerr << "failed to open test1.dat\n";
    else f << "Hallo, Welt!\n";
    f.close();
}
```

File I/O

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

int main() {
    fstream f("test1.dat", ios::in);
    char line[256];        // kein string !!!!!

    if (! f.is_open()) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }
    while (! f.eof()) {
        f.getline(line, 256);
        cout << line << endl;
    }
    f.close();
}
```

Dasselbe mit Strings:

```
#include .....

int main() {
    fstream f("test1.dat", ios::in);
    string line;          // string statt char *

    if (! f.is_open()) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }
    while (! f.eof()) {
        getline(f, line);  // globale Funktion
        cout << line << endl;
    }
    f.close();
}
```

Auch beim Lesen sind Operatorüberladungen definiert.
Positionieren in Dateien ist mittels `tellg` und `seekg` möglich.

```
int main() {
    char word[256];
    long start, end;
    fstream f("test1.dat", ios::in);

    start = f.tellg();
    f.seekg(0, ios::end);   // offset, direction
    end = f.tellg();
    cout << "size is: " << (end-start)
         << " bytes.\n";

    f.seekg(0, ios::beg);   // back to beginning
    f >> word;
    cout << word << endl;
}
```

Die Streams `cin`, `cout` und `cerr` kennen wir schon aus C, dort hießen sie allerdings anders:

```
fprintf(stdout, "Hallo!");   // printf("Hallo!");
fscanf(stdin, "%d", &i);     // scanf("%d", &i);
fprintf(stderr, "Hallo!");   // ????
```

Der Stream `cerr` oder `stderr` ist nicht gebuffert, so dass die Nachrichten direkt angezeigt werden, und nicht erst nach einem `endl`.

Was wird bei dem folgenden Programm in die Datei geschrieben?

```
#include <fstream>
.....
class Klasse {
private:
    ofstream file;
public:
    Klasse(char *name = "default.txt") {
        file.open(name);
        if (!file) {
            cout << "could not open file" << endl;
            return;
        } else file << "Eins" << endl;
    }
    void function() {
        file << "Zwei" << endl;
    }
```

```
    ~Klasse() {
        file << "Drei" << endl;
        file.close();
    }
};

int main(void) {
    Klasse *k = new Klasse();

    k->function();
    return 0;
}
```

Was wird in die Datei geschrieben?

Lösung: 'Eins' und 'Zwei' werden in die Datei geschrieben, da der Destruktor nicht aufgerufen wird, die Datei vom Betriebssystem geschlossen wird und die Puffer durch die 'endl' geleert wurden!

Anmerkung: Da wir die Variable `file` vom Typ `ofstream` definiert haben, also vom Typ Output-File-Stream, kann bei der Operation `open` die Angabe `ios::out` entfallen.

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

```
class Datum {
private:
    int _tag, _monat, _jahr;

public:
    Datum(int t = 1, int m = 1, int j = 2000);
    Datum(string dat);     // Implementierung ?

    bool istSchaltjahr();
    int kalenderwoche();
    int tagImJahr();
    string toString();     // Implementierung ?
};

int main() {
    Datum d(7, 4, 2018);
    cout << d.toString() << endl;
}
```

So funktioniert es leider nicht:

```
string toString() {
    string str;

    str += _tag;
    str += ".";
    str += _monat;
    str += ".";
    str += _jahr;

    return str;
}
```

Die Attribute `_tag`, `_monat` und `_jahr` sind vom Typ `int`, für den weder eine Operatorüberladung noch eine Methode `append` in der Klasse `string` definiert ist!

So funktioniert es:

```
string toString() {
    ostringstream os;

    os << setw(2) << _tag << ".";
    os << setw(2) << _monat << ".";
    os << setw(4) << _jahr;

    return os.str();
}
```

Ein Objekt der Klasse `ostringstream` verhält sich wie ein Objekt der Klasse `ostream`, nur das die Werte nicht auf dem Bildschirm ausgegeben werden sondern in einen `stringstream` geschrieben werden. Der Inhalt des Streams wird dann mittels der Methode `str` in einen String gewandelt. Hier müssen die Header `sstream` und `iomanip` eingebunden werden.

Wir können auch einen Konstruktor definieren, der die Daten aus einem `string` ausliest:

```
Datum(string dat) {
    istringstream is(dat);
    char t;

    is >> _tag;
    is >> t;
    is >> _monat;
    is >> t;
    is >> _jahr;
}
```

Der Aufruf `Datum d("27.3.2017");` würde jeweils den Punkt in der Variablen `t` speichern und verwerfen, Tag, Monat und Jahr werden in die entsprechenden Klassenattribute eingetragen.

Würde das obige Beispiel auch ohne die Variable `t` funktionieren?

```
Datum(string dat) {
    istringstream is(dat);

    is >> _tag >> _monat >> _jahr;
}
```

Nein! Sonst würde der Monat nicht gelesen werden, da der Punkt „.“ im Datum nicht als Wert vom Typ `int` eingelesen werden kann.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- *Statische Attribute*
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

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.

```
Konto::Konto(string inhaber, int nr, int pin) {
    _stand = 0;
    _inhaber = inhaber;
    _nr = nr;                  // !!!!!!!!!!!!!!!
    _pin = pin;
}
```

Statische Attribute

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

konto.h

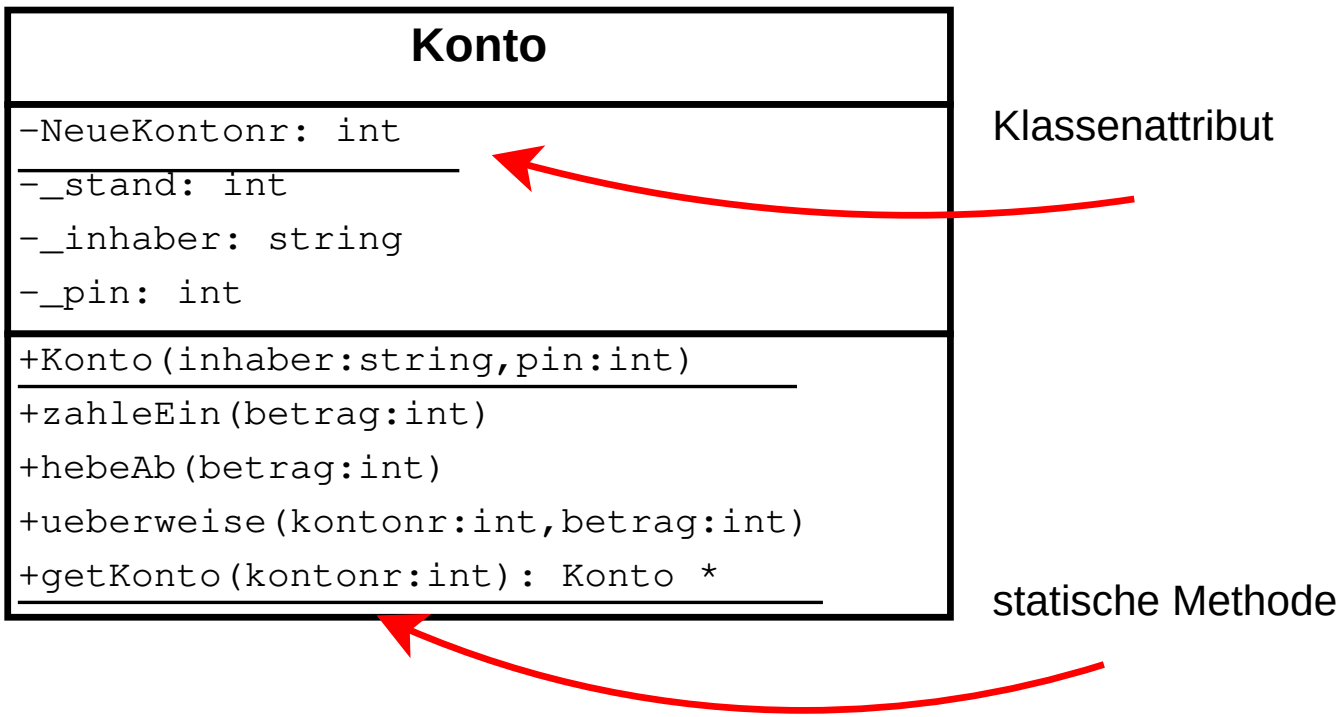
```
class Konto {
private:
    static int NeueKontonr;
    ....
public:
    Konto(string inhaber, int pin) : _stand(0) {
        _inhaber = inhaber;
        _nr = Konto::NeueKontonr++;    // !!!!!!!
        _pin = pin;
    }
    ...
};
```

konto.cpp

```
int Konto::NeueKontonr = 1;
```

Statische Attribute in UML

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

Statische Methoden werden oft anstelle von Konstruktoren verwendet:

```
class Date {
private:
    int _day, _month, _year;
public:
    Date(int d, int m, int y) {
        _day = d;
        _month = m;
        _year = y;
    }
    static Date getCurrentDate();    // !!!!!
    .....
};
```

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

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- *Ausnahmebehandlung*
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Exceptions – Ausnahmebehandlung

Zur Laufzeit eines Programms können Fehlersituationen auftreten, die eine weitere Programmausführung nur bedingt oder gar nicht mehr erlauben:

- Division durch 0
- Überlauf (zu kleiner/großer Wert für einen Datentypen)
- nicht genügend Speicherplatz vorhanden (`malloc`)
- fehlerhafte Eingaben durch Benutzer
- Zugriff auf ungültige Adressen im Hauptspeicher usw.

Frage: Wie erfolgte in C eine Ausnahmebehandlung? Wie konnte in C festgestellt werden, ob eine Funktion korrekt beendet wurde oder nicht?

Exceptions – Ausnahmebehandlung

Die Funktion kann einen speziellen Fehlerwert zurückliefern, der im Programm mittels `if` abgefragt wird.

```
.....
int find(list_t *l, int val) {
    for (int pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
.....
pos = find(myList, 10);
if (pos < 0)
    printf("Value not found\n");
else .....
```

Leider gibt es Funktionen, bei denen jeder Rückgabewert gültig ist, z.B. Funktionen aus `math.h` wie `pow` oder `ldexp`.

Exceptions – Ausnahmebehandlung

Bei einigen Funktionen in C wird daher nachträglich abgefragt, ob die zuletzt ausgeführte Operation erfolgreich war.

```
....
int main(void) {
    list_t *l;
    ....

    for (i = 0; i < 20; i++)
        append(l, i);

    i = getValueAt(l, 30);    // Fehlerwert ?????
    if (getError(l) == 0)     // Fehlerbehandlung
        printf("value[%2d] = %2d\n", 30, i);
    ....
}
```

Leider wird das oft vergessen und führt so zu Programmabbrüchen.

In C können Signal-Handler mittels `signal` eingerichtet werden.

```
#include <signal.h>
#include .....

void nullDiv(int sig) {
    printf("0-Division\n");
    exit(1);
}

void main(void) {
    int z, erg;

    signal(SIGFPE, nullDiv);

    scanf("%d", &z);
    erg = 123 / z;
    printf("123 / %d = %d\n", z, erg);
}
```

Exceptions – Ausnahmebehandlung

In C++ gibt es eine spezielle Fehlerbehandlung:

- Wir klammern die Anweisungen, in denen Exceptions, also Ausnahmen, evtl. Fehler auftreten können, mit einem `try`-Block.
- In den `catch`-Block, der dem `try`-Block unmittelbar folgt, schreiben wir die Anweisungen, die beim Auftreten von Exceptions ausgeführt werden sollen.
- Die `throw`-Anweisung löst eine Exception aus:
  - `const char *`-Exception: `throw "Division durch 0";`
  - `int`-Exception: `throw 4711;`

→ Der Code zur „normalen“ Programmausführung ist vom Code zur Fehlerbehandlung getrennt. Man verspricht sich davon eine bessere Lesbarkeit des Codes!

Liste mit Exceptions

exception.h

```
#ifndef _EXCEPTION_H
#define _EXCEPTION_H

#include <string>

class Exception {
private:
    std::string _error;

public:
    Exception(std::string error);
    std::string toString();
};

#endif
```

Liste mit Exceptions

exception.cpp

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

// Konstruktor
Exception::Exception(string error) {
    _error = error;
}

// zur Ausgabe
string Exception::toString() {
    return _error;
}
```

Liste mit Exceptions

liste.cpp

```
#include "liste.h"
#include "exception.h"

.....

int Liste::getValueAt(int idx) {
    if (idx < 0 || idx >= _last)
        throw Exception("out of bounds");
    return _values[idx];
}
```

Die Abarbeitung der Methode `getValueAt` wird durch das `throw` abgebrochen, falls ein unzulässiger Index angeben wird. Es erfolgt dann unmittelbar ein Rücksprung an die aufrufende Methode.

Wie im aufrufenden Programmteil eine solche Ausnahme abgefangen werden kann, zeigt die nächste Folie:

Liste mit Exceptions

main.cpp

```
#include .....
using namespace std;

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

    for (int value = 3; value < 8; value++)
        l.append(value);

    try {
        for (int i = 0; i < 8; i++)
            cout << i + 1 << ":  "
                 << l.getValueAt(i) << endl;
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
}
```

Exceptions – Ausnahmebehandlung

Eine Schachtelung von `try`-Blöcken ist erlaubt.

Wenn eine Exception außerhalb eines `try`-Blocks auftritt, gilt:

- Es wird die Funktion `terminate` aufgerufen, die standardmäßig die Funktion `abort` aufruft, die das Programm beendet.
- Wenn unser Programm vor dem Halten noch etwas anderes tun soll, können wir eine andere Funktion als `abort` hinterlegen.

  Dazu rufen wir die Funktion `set_terminate` auf, der wir einen Zeiger auf eine Funktion übergeben.

Exceptions – Ausnahmebehandlung

Für jeden möglichen Typ, der geworfen werden kann, müssen wir einen entsprechenden `catch`-Block angeben:

```
try {
    // some statements
} catch (OutOfBoundsException e) {
    // other statements
} catch (OverflowError e) {
    // more statements
} catch (BadAllocException e) {
    // even more statements
} catch (...) {  // Syntax korrekt!!!!!
    // handle any other exception!!!!!!
}
```

- Die Syntax ist analog zu einer Funktionsdeklaration.
- Der Parameter `e` kann weggelassen werden, falls der Wert innerhalb der geschweiften Klammern nicht benötigt wird.

In C++ sind einige Klassen zur Ausnahmebehandlung bereits vorhanden:

- `#include <exception>` stellt die Basisklasse `exception` bereit. (Basisklasse: siehe Kapitel Vererbung)
- `#include <new>` stellt die Klasse `bad_alloc` bereit, die geworfen wird, wenn kein Speicher bereit gestellt werden konnte.
- `#include <typeinfo>` stellt die Klassen `bad_cast` und `bad_typeid` bereit.
- `#include <stdexcept>` stellt die Klassen `domain_error`, `invalid_argument`, `length_error` und `out_of_range` bereit.
- Jede dieser Klassen implementiert die Methode `what`, die ein `const char*` liefert, also eine C-Zeichenkette, die den Fehler näher beschreibt.

Wie geht es weiter?

Anmerkung zur Liste:

- Unsere Liste kann aufgrund der Typisierung nur `int`-Werte speichern.

→ Eine Wiederverwendung ist daher nur durch Copy-and-Paste möglich: Erstelle für jeden Datentypen eine eigene Liste.

Copy-and-Paste ist natürlich keine Lösung. Alternativen

- in C:
  - Datentyp `int` durch `void *` ersetzen oder
  - Funktionen durch parametrisierte Makros ersetzen.
- in C++: Generische Programmierung mittels Templates.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- *Generische Programmierung*
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

main.c

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

int main(void) {
    int i;
    list_t *li, *lf;
    // list of int-values !!!!!
    li = create(2, sizeof(int));
    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);
```

Liste in C

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

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

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

    return 0;
}
```

Liste in C

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_s 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);
```

Liste in C

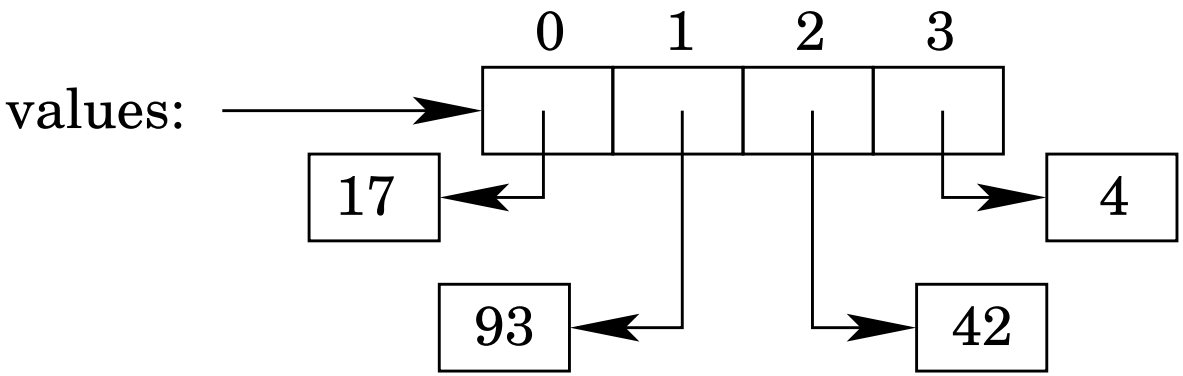
Anmerkungen:

- 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.

Liste in C

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 in C

liste.c

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

struct list_s {
    void **value;
    int nmemb;
    int last;
    char error;
    int esize;
};

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

Liste in C

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

    l = (list_t *) malloc(sizeof(list_t));
    l->last = 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;
}
```

Liste in C

```
static char isFull(list_t *l) {      // private
    return l->last == 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->last] = elem;
    l->last += 1;
}
```

Liste in C

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.

Liste in C

```
void * getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 1;
        return NULL;
    }
    return l->value[pos]; // oder Kopie liefern?
}

void destroy(list_t *l) {
    int i;

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

Liste in C

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->last) {
        l->error = 1;
        return NULL;
    }

    // Kopie erzeugen !!!!!
    void *result = malloc(esize);
    memcpy(result, values[pos], esize);
    return result;
}
```

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

Liste in C

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

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

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

Liste in C

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->last) {
        .....     // Fehlerbehandlung
    }
    memcpy(l->result, l->values[pos], l->esize);
    return l->result;
}

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

Eine andere Möglichkeit: Die Funktion `getValueAt` bekommt einen Out-Parameter, über den der Wert geliefert wird.

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

Dann ist der Nutzer für den Speicherbereich verantwortlich.

```
.....
int main(void) {
    int v;
    .....
    getValueAt(l, 20, &v);
    .....
}
```

Liste in C

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.

Templateklasse Liste

liste.h

```
#include "exception.h"

template <typename T>
class Liste {
    T *_values;
    int _last, _size;
    bool isFull();
    int find(T val);
    void increase();
    void decrease();
public:
    Liste(int size);
    ~Liste();
    void append(T val);
    T getValueAt(int pos);
    void erase(T val);
    void toScreen();
};
```

Templateklasse Liste

```
template <typename T>
Liste<T>::Liste(int size) {
    _size = size;
    _last = 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 >= _last)
        throw Exception("out of bounds");
    return _values[pos];
}
```

Templateklasse Liste

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

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

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

Templateklasse Liste

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

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

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

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

Templateklasse Liste

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

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

main.cpp

```
#include .....

int main(void) {
    Liste<int> l(4);

    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;
    }
```

Templateklasse Liste

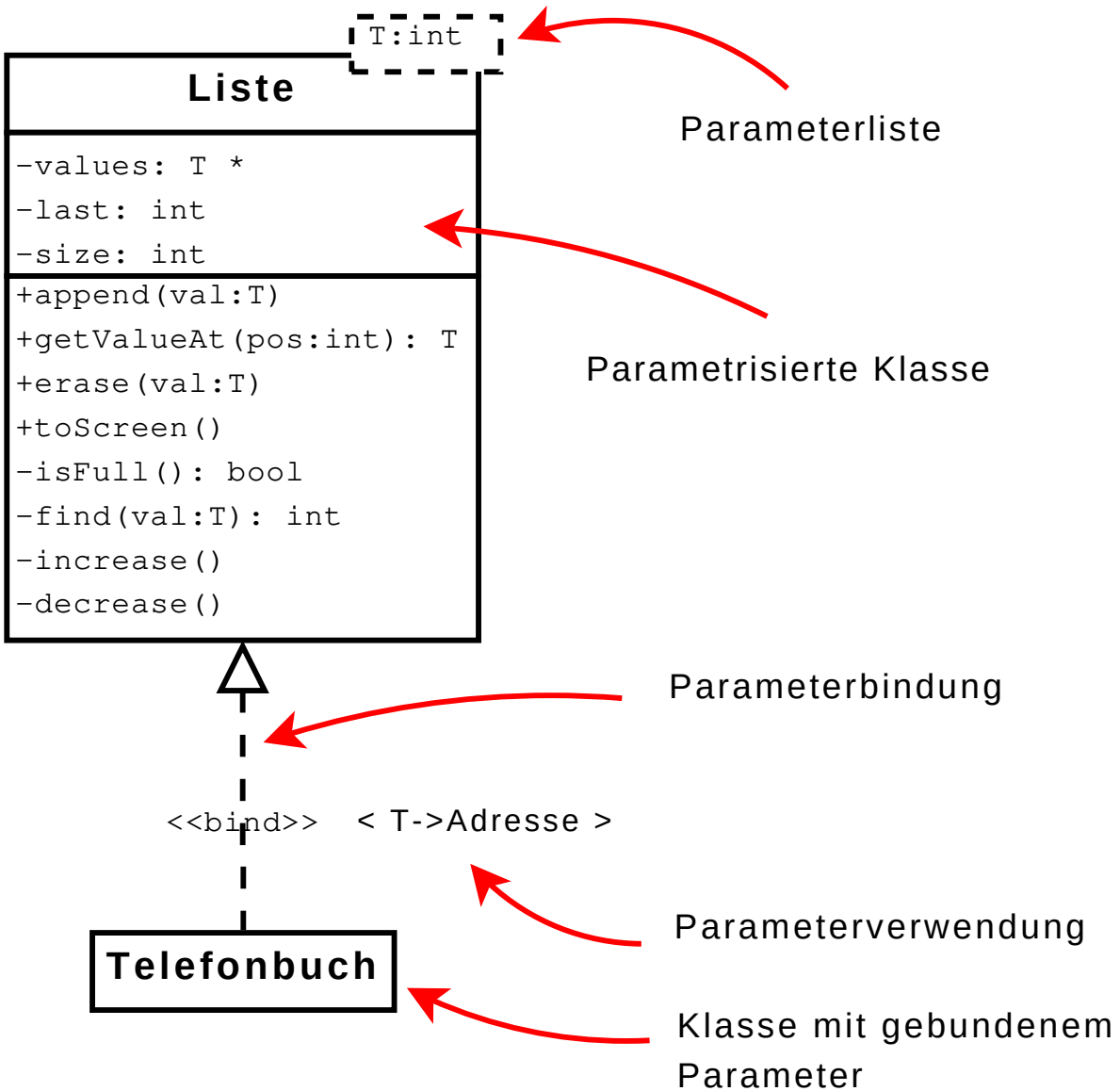
```
    Liste<float> ll(4);
    float f;

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

    try {
        for (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;
    }
}
```

Templateklasse Liste

Anmerkungen zur Liste:

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

Templateklassen in UML



Generische Programmierung

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.

Verwendet wird heute das Schlüsselwort `typename`, veraltet ist dagegen `class`.

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 n) {
    T sum = array[0];
    for (int i = 1; i < n; 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;
```

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!

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

template <typename T, typename S>
S summe(map<T, S> &toAdd) {
    map<T, S>::iterator it = toAdd.begin();

    if (it == toAdd.end())
        throw "empty map exception";

    S sum = it->second;
    for ( ; it != toAdd.end(); it++)
        sum += it->second;
    return sum;
}
```

```
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 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.

String-Tokenizer

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();
        .....
    }
    return 0;
}
```

tokenizer.h

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

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

public:
    Tokenizer(string data, string separators);

    int countTokens();
    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);
    }
}
```

String-Tokenizer

```
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:

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

Überlegen Sie, in welchen Fällen der obige String-Tokenizer nicht korrekt funktioniert.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- *Referenzen*
- Kopieren von Objekten
- Operatorüberladung

Referenzen

Referenzen erübrigen die Übergabe von Zeigern, wenn Werte innerhalb einer Funktion geändert werden sollen:

```
void swap(int &x, int &y) {
    int t = x;

    x = y;
    y = t;
}

int main(void) {
    int a, b;
    ...
    swap(a, b);
    ...
}
```

Referenzen

Pointer sind klarer und sollten vorgezogen werden: Bei Verwendung von Pointern ist für den *Anwender* einer Funktion ersichtlich, dass die Werte in der Funktion verändert werden können, vergleiche dazu die Aufrufe `swap(a, b)` und `swap(&a, &b)`.

Bjarne Stroustrup, the designer and original implementor of C++: Consequently “plain“ reference arguments should be used only when the name of the function gives a strong hint that the reference argument is modified.

Referenzen sind implementiert über die Verwendung von konstanten Zeigern!

Übung: Voriges `swap(int &x, int &y)` entspricht welcher der folgenden Implementierungen?

```
void swap1(const int *x, const int *y) {
    ...              // Implementierung siehe swap3

void swap2(int const *x, int const *y) {
    ...              // Implementierung siehe swap3

void swap3(int * const x, int * const y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

Referenzen

Eine Referenz ist ein alternativer Name für eine Variable, Konstante oder Funktion.

```
// typische Deklaration:
T & Bezeichner = Ausdruck;
// Beispiel:
const int &x = 4711;
```

Einige Aspekte von Referenzen:

- Referenzen müssen bei ihrer Deklaration initialisiert werden.
- Referenzen können nicht durch `new` erzeugt werden.
- Zeiger auf Referenzen sind nicht möglich.
- Felder von Referenzen sind nicht möglich.
- Es gibt keine Referenzen auf Referenzen.
- Es gibt keine Referenzen auf `void`.

Lernkontrolle

Ist das folgende Programm korrekt? Falls ja, welche Ausgabe erzeugt es?

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

int summe(int &a, int &b) {
    return a + b;
}

int main() {
    cout << summe(1, 2) << endl;
}
```

Das vorige Programm ist nicht korrekt: 1 und 2 sind Konstanten und könnten unter dem Alias `a` bzw. `b` verändert werden!

besser:

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

int summe(const int &a, const int &b) {
    return a + b;
}

int main() {
    cout << summe(1, 2) << endl;
}
```

Gegeben seien folgende Funktionen:

```
int  g(int);
int& h(const int &);
```

Welche der folgenden Aufrufe der obigen Funktionen sind richtig?

		richtig	falsch
(a)	`int i = ++g(42);`	☐	☐
(b)	`int i = ++h(42);`	☐	☐
(c)	`int *p = &g(42);`	☐	☐
(d)	`int *p = &h(42);`	☐	☐
(e)	`g(42) = 42;`	☐	☐
(f)	`h(42) = 42;`	☐	☐
(g)	`int i = g(h(42));`	☐	☐
(h)	`int i = h(g(42));`	☐	☐

Gegeben seien folgende Funktionen:

```
void f(int);
void g(int &);
void h(const int &);
```

Welche Aufrufe der Funktionen sind richtig, falls `i` eine Variable vom Typ `int` ist?

		richtig	falsch
(a)	`f(7);`	☐	☐
(b)	`f(i);`	☐	☐
(c)	`g(7);`	☐	☐
(d)	`g(i);`	☐	☐
(e)	`h(7);`	☐	☐
(f)	`h(i);`	☐	☐

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- *Kopieren von Objekten*
- Operatorüberladung

Kopieren von Objekten

In folgenden Fällen werden die Attributwerte eines Objekts in ein anderes Objekt derselben Klasse kopiert:

- Initialisieren eines Objekts mit einem anderen Objekt derselben Klasse: Zuweisung bei Deklaration.
- Übergabe eines Objekts als Argument bei einem Funktionsaufruf, nicht bei Zeigern oder Referenzen!
- Rückgabe eines Objekts als Funktionswert.

Dabei wird nicht der Konstruktor aufgerufen, sondern der *Copy-Konstruktor*!

Syntax:

```
Klasse(const Klasse &zuKopierendesObjekt)
```

Kopieren von Objekten

Wird kein Copy-Konstruktor programmiert, dann erzeugt der Compiler einen *default copy constructor*:

- Alle Werte des Objekts werden elementweise kopiert, unabhängig davon, ob es Adressen sind.

→ Verweise im Original und in der Kopie sind identisch!

Eine solche Kopie wird als *flache Kopie* bezeichnet.

Soll eine echte Kopie unserer Liste erzeugt werden, muss das Array, das die eigentlichen Werte enthält, kopiert werden.

Kopieren von Objekten

```
.....
template <typename T>
Liste<T>::Liste(const Liste& l) {
    _size = l._size;
    _last = l._last;
    _values = new T[_size];

    for (int i = 0; i < _last; i++)
        _values[i] = l._values[i];
}
```

Welche Ausgabe erzeugt das Programm?

```
#include <iostream>

class CTest {
public:
    CTest() { std::cout << "K\n"; }
    ~CTest() { std::cout << "D\n"; }
    CTest(const CTest &c) { std::cout << "C\n"; }
};

void down(CTest t) { return; }

int main(void) {
    CTest u;
    CTest v = u;
    down(u);
}
```

Kopieren von Objekten

Anmerkungen:

- Ein Copy-Konstruktor kann auch explizit aufgerufen werden:

```
CTest *t = new CTest(orig);
```

- Weitere Parameter sind möglich, sie müssen aber alle default-Werte haben.
- Der Copy-Konstruktor kann wie andere Konstruktoren eine Initialisierungsliste haben.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- *Operatorüberladung*

Wie war es bisher?

Rationale Zahlen in C mit modularer Programmierung:

rational.h

```
// incomplete data type
typedef struct rat_s rat_t;

// interface
rat_t *createRat(int zaehler, int nenner);
void destroyRat(rat_t *r);

rat_t *addRat(rat_t *a, rat_t *b);   // +
rat_t *subRat(rat_t *a, rat_t *b);   // -
rat_t *mulRat(rat_t *a, rat_t *b);   // *
rat_t *divRat(rat_t *a, rat_t *b);   // /
.....
void printRat(rat_t *a);
```

Motivation

Aufruf mit:

```
rat_t *a, *b, *c; *d;
a = createRat(1, 2);
b = createRat(3, 7);

c = addRat(a, b);
d = mulRat(a, b);
printRat(c);
if (isGreater(c, d))
    ...
else ...
.....
```

Schöner wäre:

```
Rational a(1, 2);
Rational b(3, 7);
Rational c, d;

c = a + b;
d = a * b;
cout << c << endl;
if (c > d)
    ...
else ...
.....
```

rational.h

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

class Rational {
    friend ostream& operator<<(ostream& os,
                const Rational& x) {
        os << x.zaehler << "/" << x.nenner;
        return os;
    }
private:
    int zaehler, nenner;
    Rational add(Rational x);   // Addition
    Rational sub(Rational x);   // Subtraktion
    Rational mul(Rational x);   // Multiplikation
    Rational div(Rational x);   // Division
    void kehrwert();
    void kuerzen();
```

Rationale Zahlen in C++

```
public:
    Rational(int z = 1, int n = 1);

    Rational operator-();     // Vorzeichen
    Rational operator+(const Rational& x);
    Rational operator-(const Rational& x);
    Rational operator*(const Rational& x);
    Rational operator/(const Rational& x);

    bool operator<(const Rational& x);
    bool operator>(const Rational& x);
    bool operator==(const Rational& x);

    int getZaehler();
    int getNenner();
};
```

rational.cpp

```
#include "rational.h"

// Konstruktor
Rational::Rational(int z, int n) {
    zaehler = z;
    nenner = n;
    kuerzen();
}

void Rational::kehrwert() {
    int t = zaehler;

    zaehler = nenner;
    nenner = t;
}
```

Rationale Zahlen in C++

```
void Rational::kuerzen() {
    // Algorithmus nach Euklid
    int r;
    int p = zaehler;
    int q = nenner;

    do {
        r = p % q;
        p = q;
        q = r;
    } while (r != 0);

    zaehler /= p;
    nenner /= p;
}
```

Rationale Zahlen in C++

```
Rational Rational::add(Rational x) {
    Rational r;

    r.zaehler = zaehler * x.nenner
              + nenner * x.zaehler;
    r.nenner = nenner * x.nenner;
    r.kuerzen();

    return r;
}

Rational Rational::sub(Rational x) {
    return add(-x);
}
```

Rationale Zahlen in C++

```
Rational Rational::mul(Rational x) {
    return Rational(zaehler * x.zaehler,
                    nenner * x.nenner);
}

Rational Rational::div(Rational x) {
    x.kehrwert();
    return mul(x);
}

int Rational::getZaehler() {
    return zaehler;
}

int Rational::getNenner() {
    return nenner;
}
```

Rationale Zahlen in C++

```
Rational Rational::operator-() {    // Vorzeichen
    return Rational(-zaehler, nenner);
}

Rational Rational::operator+(const Rational& x){
    return add(x);
}
Rational Rational::operator-(const Rational& x){
    return sub(x);
}
Rational Rational::operator*(const Rational& x){
    return mul(x);
}
Rational Rational::operator/(const Rational& x){
    return div(x);
}
```

Rationale Zahlen in C++

```
bool Rational::operator<(const Rational& x) {
    return zaehler * x.nenner
           < nenner * x.zaehler;
}

bool Rational::operator>(const Rational& x) {
    return zaehler * x.nenner
           > nenner * x.zaehler;
}

bool Rational::operator==(const Rational& x) {
    return zaehler * x.nenner
           == nenner * x.zaehler;
}
```

main.cpp

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

int main(void) {
    Rational a(2), b(1, 3), c;

    c = a + b;
    cout << "a + b = " << c << endl;
    c = a - b;
    cout << "a - b = " << c << endl;
    c = a * b;
    cout << "a * b = " << c << endl;
    c = a / b;
    cout << "a / b = " << c << endl;
```

```
    if (a < b)
        cout << "a < b" << endl;

    if (a > b)
        cout << "a > b" << endl;

    if (a == b)
        cout << "a == b" << endl;

    cout << "c.zaehler = " << c.getZaehler();
    cout << "c.nenner  = " << c.getNenner();

    return 0;
}
```

Überladen von Operatoren

Der Operator << in der Klasse ostream ist vielfach überladen, um beliebige Datentypen in ein ostream-Objekt schreiben zu können:

```
class ostream {
public:
    ostream& operator<< (bool& wert);
    ostream& operator<< (short& wert);
    ostream& operator<< (unsigned short& wert);
    ostream& operator<< (int& wert);
    ostream& operator<< (unsigned int& wert);
    ostream& operator<< (long& wert);
    ostream& operator<< (unsigned long& wert);
    .....
};
```

Überladen von Operatoren

in C++: `cout` ist ein vordefiniertes Objekt der Klasse `ostream`
in C: `stdout` ist ein vordefinierter Zeiger vom Typ `FILE *`

```
cout << 1;
```

wird vom Compiler übersetzt nach

```
cout.operator<<(1);
```

Alle `<<`-Operatorfunktionen liefern eine Referenz auf das aktuelle Stream-Objekt als Rückgabewert, wodurch eine Verkettung mehrerer Ausgaben möglich wird:

```
cout << 1 << ", " << 2 << endl;
```

Überladen von Operatoren

Der Operator << wird von links nach rechts abgearbeitet:

```
cout << 1 << ", " << 2 << endl;
```

entspricht also

```
(((cout << 1) << ", ") << 2) << endl;
```

und daher

```
cout.operator<<(1).operator<<(", ")
    .operator<<(2).operator<<(endl);
```

Auch die Priorität kann nicht geändert werden:

```
cout << a + b;        // Klammern unnoetig
cout << (a & b);      // Klammern notwendig
```

Überladen von Operatoren

Wenn Sie den Operator `<<` auf eigene Klassen erweitern wollen, müssen Sie die globale Operatorfunktion `operator<<` überladen:

```
ostream& operator<< (ostream& os,
                     const Rational& r) {
    os << r.getZaehler();
    os << "/";
    os << r.getNenner();
    return os;
}
```

Die Getter-Methoden `getZaehler` und `getNenner` sind notwendig, da der Zugriff auf die privaten Variablen `zaehler` und `nenner` nicht möglich ist.

Es sei denn ....

Überladen von Operatoren

.... die Operatorfunktion `operator<<` wird als Freund der Klasse `Rational` definiert. Freunde dürfen private Variablen sehen!

```
class Rational {
    friend ostream& operator<< (ostream& os,
                const Rational& r) {
        os << r.zaehler << "/" << r.nenner;
        return os;
    }
.....
};
```

Anmerkungen:

- Zur besseren Lesbarkeit stehen `friend`-Deklarationen am Anfang der Klasse.
- Es können ganze Klassen als Freund definiert werden.
- Das Konzept der Freunde gibt es nicht in allen objektorientierten Programmiersprachen.

Überladen von Operatoren

Operatoren können wie Funktionen überladen werden.

Jede Operatoranwendung kann aufgefasst werden als Aufruf einer Operatorfunktion der Form:

```
<Typ> operator<Zeichen>(<Formalparameter>)
Rational operator+(Rational a)
```

Zum Überladen muss die Operatorfunktion entweder

- als Klassenfunktion definiert werden, dann besitzt sie einen impliziten Objektparameter der Form `T &` oder `const T &` oder
- als Funktion mit mindestens einem Parameter vom Typ Klasse, oder Referenz auf Klasse definiert werden.

Überladen von Operatoren

Anmerkungen:

- Operatoren für Standardtypen können nicht überladen werden.
- Operatorfunktionen können explizit aufgerufen werden.
- Priorität und Assoziativität eines Operators wird nicht verändert.
- Operandenzahl eines Operators kann nicht verändert werden.
- Es können keine neuen Operatorzeichen definiert werden.

  Zum Beispiel ist ** für Potenzieren nicht möglich.

Übungen

Überladen Sie den Index-Operator für unsere Liste, sodass anstelle von `val = l.getValueAt(10)` einfacher `val = l[10]` zum Zugriff auf das zehnte Element geschrieben werden kann.

Warum ist beim Einlesen eines Wertes von der Tastatur kein Adress-Operator notwendig, wenn doch das Einlesen mittels `>>` in einen Funktionsaufruf umgewandelt wird?

```
int a;
cin >> a;    // entspricht: cin.operator>>(a);
```

Lösungen

Um den Indexoperator für unsere Liste zu implementieren, nutzen wir die bereits implementierte Methode `getValueAt`, da diese prüft, ob der angegebene Index innerhalb des gültigen Bereichs liegt, und genau die gewünschte Funktionalität bereit stellt:

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

Beim Einlesen eines Wertes von der Tastatur ist kein Adress-Operator nötig, weil der entsprechende `operator>>`-Aufruf eine Referenz als Parameter hat und daher bereits ein Call-By-Reference ausgeführt wird:

```
istream& operator>> (int& wert);
```

Überladen von Operatoren

Wir hatten bereits festgestellt: Die Programmiererin legt fest, wann zwei Objekte als gleich angesehen werden. In C++ wird dazu der ==-Operator überladen.

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

Wunscheigenschaften an den ==-Operator:

- *Reflexivität:* Ein Objekt ist gleich zu sich selbst. Es sollte also immer `obj == obj` gelten.
- *Symmetrie:* Wenn `obj1 == obj2` gilt, dann sollte natürlich auch `obj2 == obj1` gelten.
- *Transitivität:* Gleichheit kann verkettet werden und gilt für mehrere Objekte: Wenn `ob1 == obj2` und `obj2 == obj3` gilt, dann sollte auch `obj1 == obj3` gelten.

Überladen von Operatoren

Frage: Gibt es Probleme, wenn wir folgendes schreiben?

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

int main(void) {
    int x = 5;
    Rational b(1, 3), c;

    c = x + b;
    cout << "a + b = " << c << endl;

    c = b - x;
    cout << "a - b = " << c << endl;
}
```

Überladen von Operatoren

```
...
int main(void) {
    int x = 5;
    Rational b(1, 3), c;

    // explizite Typumwandlung erforderlich
    c = (Rational) x + b;
    cout << "a + b = " << c << endl;

    // nichts zu tun
    c = b - x;    // c = b.operator+(x)
    cout << "a - b = " << c << endl;
}
```

In beiden Fällen wird `x` in den Typ `Rational` umgewandelt, indem der passende Konstruktor aufgerufen wird.

Frage: Was wäre zu tun, damit eine solche Schreibweise auch für Werte vom Typ `double` funktioniert?

Antwort: Wir müssten einen Konstruktor angeben, der einen `double`-Wert in einen Wert vom Typ `Rational` umwandelt.

Frage: Wie wandelt man einen `double`-Wert wie 2.0815 in einen Bruch um?

Antwort: Den Wert solange mit 10 multiplizieren, bis kein Nachkommateil mehr vorhanden ist, und anschließend den so entstandenen Bruch kürzen.

$$2.0815 = \frac{2.0815}{1} = \frac{20.815}{10} = \frac{208.15}{100} = \frac{2081.5}{1000} = \frac{20815}{10000} = \frac{4163}{2000}$$

Übung: Formulieren Sie einen Konstruktor der Klasse `Rational`, der einen Wert vom Typ `double` als Bruch darstellt.

Überladen von Operatoren

```
Rational::Rational(float val) {
    double z = val;
    int n = 1;

    double d = z - (int) z;

    while (d >= 1e-4) {  // oder d != 0.0 ???
        z *= 2;          // nicht z *= 10; !!!
        n *= 2;          // nicht n *= 10; !!!
        d = z - (int) z;
    }
    zaehler = z;
    nenner = n;
    kuerzen();
}
```

Überladen von Operatoren

Frage: Warum multiplizieren wir Zähler und Nenner mit 2, anstatt mit 10?

`float`- und `double`-Werte sind im Format IEEE 754 gespeichert. Eine Multiplikation mit 2 verschiebt daher das Komma um genau eine Stelle.

Würde mit 10 multipliziert werden, entstehen eventuell weitere Nachkommastellen und die dargestellte Zahl würde durch Rundungsfehler noch ungenauer.

Übung: Formulieren Sie einen Konstruktor der Klasse `Rational`, der eine Zahl, gegeben als `string`, als Bruch darstellt.

Überladen von Operatoren

```
Rational::Rational(string v) {
    int i, z = 0, n = 1, size = (int) v.size();

    // Vorkommateil (ohne Fehlerbehandlung !!!)
    for (i = 0; i < size && isdigit(v[i]); i++) {
        z *= 10;
        z += v[i] - '0';
    }
    // Nachkommateil (ohne Fehlerbehandlung !!!)
    for (++i; i < size && isdigit(v[i]); i++) {
        z *= 10;
        z += v[i] - '0';
        n *= 10;
    }
    zaehler = z;
    nenner = n;
    kuerzen();
}
```