

1.1 Ein- und Ausgabe

Erstellen Sie in C++ folgendes Programm: Lesen sie aus der Datei `eingabe.txt` zeilenweise die Werte aus und schreiben diese formatiert in die Datei `ausgabe.txt`. Die Formatierung soll über Manipulatoren erfolgen.

eingabe.txt-Zeile: 1343 4.23423434 HalloWelt

Die einzelnen Zeilen sollen nummeriert werden. Die Zahl soll als hexadezimale Zahl dargestellt werden. Die hexadezimale Zahl soll eine minimale Breite von vier haben und durch Nullen aufgefüllt werden. Die double-Zahl soll mit sechs Zeichen ausgegeben werden. Der String soll eine minimale Länge von 15 haben. Die fehlenden Ziffern sollen mit Minuszeichen aufgefüllt werden. Die einzelnen Spalten sollen durch | getrennt werden

ausgabe.txt-Zeile: 1. | 053f | 4.23423 | -----HalloWelt

`eingabe.txt:`

```
1343 4.23423434 HalloWelt
34234 4323.1234243 C++machtSpaß
65455 4.30658456 Testing
3412 1.234567 abcderfghijklmn
65535 5.3620909482 Hexadezimal
32342 3423.343423 Referenzen
9834 09.934343 yxzuerf
003234 9883.32414 ABCDERFGHIJKLM
213 3.141593239853 Beispiel
```

`ausgabe.txt`

```
1. | 053f | 4.23423 | -----HalloWelt
2. | 85ba | 4323.12 | ---C++machtSpaß
3. | ffaf | 4.30658 | -----Testing
4. | 0d54 | 1.23457 | abcderfghijklmn
5. | ffff | 5.36209 | ---Hexadezimal
6. | 7e56 | 3423.34 | -----Referenzen
7. | 266a | 9.93434 | -----yxzuerf
8. | 0ca2 | 9883.32 | -ABCDERFGHIJKLM
9. | 00d5 | 3.14159 | -----Beispiel
```

1.2 Funktionen

Implementieren Sie die folgenden Funktionen:

1. Schreiben Sie eine Funktion `abschneiden`, die als ersten Parameter eine float-Variable `x` (zwischen 0 und 1000) und als zweiten Parameter einen positiven int-Wert `n` (zwischen 1 und 5) übergeben bekommt. Die Funktion soll den ersten Parameter auf `n` Nachkommastellen begrenzen (durch Abschneiden der restlichen Nachkommastellen). Das Ergebnis soll in `x` stehen. Die Funktion soll ohne Nutzung von Funktionen der mathematischen Bibliothek implementiert werden.
2. Schreiben Sie eine Funktion `enthaeltGenauEinmal7`, die als Parameter einen int-Wert `zahl` übergeben bekommt und überprüft, ob die Ziffer 7 in `zahl` genau einmal vorkommt und die Dezimalstelle der Ziffer 7 in `zahl` zurückgibt (Beispiel: `enthaelt7(-2578) == 2`).

Kommt die Ziffer 7 gar nicht vor, soll ein Ausnahmeobjekt vom Typ `invalid_argument` geworfen und vom Aufrufer der Funktion behandelt werden (Ausgabe einer Fehlermeldung). Kommt die Ziffer 7 in `zahl` mehrmals vor, soll eine Ausnahme geworfen und innerhalb der Funktion dahingehend behandelt werden, dass der Nutzer spezifiziert, welche der gefundenen 7 zählt.

1.4 Listenbearbeitung

Schreiben Sie mindestens zwei der folgenden Funktionen, die für je zwei übergebene integer-Listen

- überprüft, ob die Listen gemeinsame Werte besitzen
(Rückgabe: Liste dieser Werte (Die Liste soll keine doppelten Elemente enthalten))
- überprüft, ob die Listen keine gemeinsamen Werte besitzen
(Rückgabe: true/false)
- genau dann true zurückliefert, wenn alle Werte der ersten Liste genauso häufig in der zweiten Liste vorkommen und umgekehrt
- die erste Liste mit denjenigen Elementen der zweiten Liste auffüllt, die nur in der zweiten Liste vorkommen

Diese Aufgabe können Sie entweder „von Hand“ programmieren oder durch geschickten Einsatz von Containermethoden kurze Lösungen finden.

Schreiben Sie ein Hauptprogramm, das Ihre Funktionen testet. Denken Sie auch an die leere Liste.

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <iomanip>
5 #include <vector>
6 #include <stdexcept>
7 #include <exception>
8
9 void EinUndAusgabe(){
10    std::ifstream input("eingabe.txt",std::fstream::in);
11    std::fstream output("ausgabe.txt",std::fstream::out|std::fstream::app);
12    int spalte1 {};
13    double spalte2 {};
14    std::string spalte3 {};
15    int i{1};
16
17    while(input >> spalte1 >> spalte2 >> spalte3){
18        output << i << ". | " << std::setfill('0') << std::uppercase << std::hex << std::setw(4)
19        ) << spalte1 << " | "
20        << spalte2 << " | " << std::setw(15) << std::setfill('-') << spalte3 << '\n';
21        std::cout << i++ << ". | " << std::setfill('0') << std::uppercase << std::hex << std::setw(4)
22        ) << spalte1 << " | " << spalte2 << " | "
23        << std::setw(15) << std::setfill('-') << spalte3 << '\n';
24    }
25
26 float abschneiden(float x, int n){
27
28    for (int i = 0; i < n; i++){
29        x = x * 10;
30    }
31    x = (int) x * 1;
32    for (int a = 0; a < n; a++){
33        x = x / 10;
34    }
35    return x;
36 }
37
38 int enthaeltGenauEinmal7(int zahl){
39    std::vector<int> dieSieben {};
40    int count {0};
41    while (zahl != 0){
42        count++;
43        int help = zahl % 10;
44        if (help == 7 || help == -7){
45            dieSieben.push_back(count);
46        }
47        zahl = zahl / 10;
48    }
49
50    if(dieSieben.size() == 0){
51        throw std::invalid_argument("geht nicht");
52    }else if(dieSieben.size() > 1){
53        for(int i = 0; i<dieSieben.size(); i++){
54            std::cout << dieSieben.at(i) << " ";
55        }
56        throw std::exception();
57    }else if(dieSieben.size() == 1){
58        return dieSieben.at(0);
59    }
60
61    return 0;
62 }

```

```

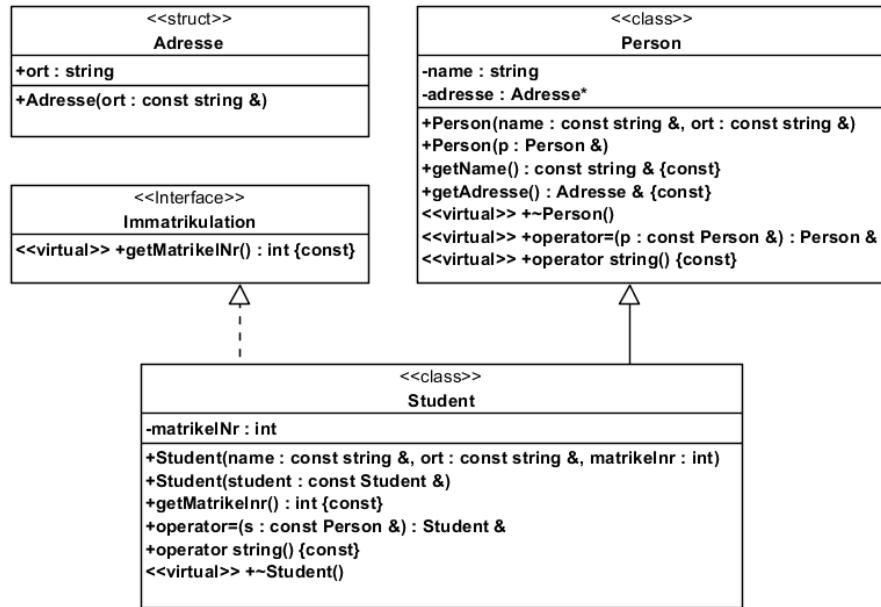
63
64
65
66 std::vector<int> gemeinsameWerte(std::vector<int> a, std::vector<int> b){
67     std::vector<int> erg {};
68     for (int i = 0; i < a.size(); i++){
69         for (int j = 0; j < b.size(); j++){
70             if (a.at(i)==b.at(j)){
71                 erg.push_back(a.at(i));
72             }
73         }
74         for (int k = 0; k < erg.size()-1; k++){
75             if(erg.at(k) == erg.back()){
76                 erg.pop_back();
77             }
78         }
79     }
80     return erg;
81 }
82
83 bool keineGemeinsamenWerte(std::vector<int> a, std::vector<int> b){
84     for (int i = 0; i < a.size(); i++){
85         for (int j = 0; j < b.size(); j++){
86             if (a.at(i)==b.at(j)){
87                 return true;
88             }
89         }
90     }
91     return false;
92 }
93
94 int main() {
95     //Aufgabe 1.1
96     //EinUndAusgabe();
97
98     // Aufgabe 1.2.1
99 /*
100     float x {};
101     std::cout << "Zahl x zwischen 0 - 1000 wählen" << std::endl;
102     std::cin >> x;
103
104     while (x < 0 || x > 1000){
105         std::cout << "Die Zahl liegt nicht im Zahlenbereich 0 - 1000" << std::endl;
106         std::cin >> x;
107     }
108
109     int n {};
110     std::cout << "Nachkommastelle eingeben" << std::endl;
111     std::cin >> n;
112     while (n < 1 || n > 5){
113         std::cout << "Richtig eingeben" << std::endl;
114         std::cin >> n;
115     }
116     x = abschneiden(x,n);
117     std::cout << x << std::endl;
118 */
119
120     //Aufgabe 1.2.2 Funktionen
121
122     int zahl {};
123     std::cout << "Zahl x eingeben" << std::endl;
124     std::cin >> zahl;
125     try{
126         std::cout << enthaeltGenauEinmal7(zahl) << std::endl;

```

```
127     } catch (const std::invalid_argument){  
128         std::cout << "Eine Zahl mit 7 wird erwartet. :)" << std::endl;  
129     } catch (const std::exception){  
130         std::cout << std::endl << "Zu viele 7! Geben Sie die Stelle an, die zählen soll" << std::endl;  
131         int x {};  
132         std::cin >> x;  
133         std::cout << "Sie wählten: " << x;  
134     }  
135  
136  
137 //Aufgabe 1.4 Listenbearbeitung  
138 /*int zahl1 {0};  
139 std::vector<int> liste1 {};  
140 std::cout << "Zahl eingeben für Liste 1: " << std::endl;  
141 std::cin >> zahl1;  
142 while (zahl1 != -1) {  
143     liste1.push_back(zahl1);  
144     std::cout << "Zahl eingeben für Liste 1: " << std::endl;  
145     std::cin >> zahl1;  
146 }  
147  
148 int zahl2 {0};  
149 std::vector<int> liste2 {};  
150 std::cout << "Zahl eingeben für Liste 2: " << std::endl;  
151 std::cin >> zahl2;  
152 while (zahl2 != -1) {  
153     liste2.push_back(zahl2);  
154     std::cout << "Zahl eingeben für Liste 2: " << std::endl;  
155     std::cin >> zahl2;  
156 }  
157 std::vector<int> gemeinsam = gemeinsameWerte(liste1,liste2);  
158 for (int i = 0; i < gemeinsam.size(); i++){  
159     std::cout << gemeinsam.at(i) << std::endl;  
160 }  
161 std::cout << std::boolalpha << keineGemeinsamenWerte(liste1,liste2) << std::endl;*/  
162 return 0;  
163 }  
164
```

5. Vererbung (15 Punkte)

Umgesetzt werden sollen die Klassen des folgenden UML-Klassendiagramms mit der Programmiersprache C++. Der Strukturdatentyp `Adresse` und die Klasse `Person` werden in Aufgabe 4 definiert und hier weiter verwendet.



Aufgabe:

- 5.a Definieren Sie das Interface `Immatrikulation` als abstrakte Klasse mit der rein virtuellen Methode `getMatrikelnummer()`. Setzen Sie die Vorgaben des UML-Klassendiagramms um.

Zu beantwortende Frage:

- Überlegen Sie, ob die Einführung eines virtuellen Destruktors für diese abstrakte Klasse sinnvoll ist. Begründen Sie Ihre Entscheidung und erklären Sie kurz und stichpunktartig die Auswirkungen des Schlüsselworts `virtual` im Zusammenhang mit Destruktoren in Vererbungsbeziehungen.

- 5.b Definieren Sie die Klasse `Student` als öffentliche Ableitung der Klassen `Person` und `Immatrikulation` mit der Objektvariablen `matrikelnr` vom Typ `int`.

Anforderung: Die Methoden sollen in dieser Aufgabe im Klassenblock deklariert und in Aufgabe 5.c außerhalb des Klassenblocks definiert werden.

- 5.c Definieren Sie folgende Methoden der Klasse `Student` außerhalb des Klassenblocks:

- Überladener Konstruktor: Übergeben werden die Parameter `name` und `ort` als konstante Referenzen auf Objekte vom Typ `string` sowie `matrikelNr` vom Typ `int`. Zu beachten: die Objektvariablen der Klasse `Person` sind `private` und es existieren keine setter-Methoden.
- Zuweisungsoperator: Bedenken Sie, dass es Regeln beim Überschreiben des Zuweisungsoperators gibt. Halten Sie diese ein und setzen Sie zur Typkonvertierung `dynamic_cast<const Student&>(...)` ein (\Rightarrow Down-Cast).
- Konvertierungsoperator (nach string): rufen Sie den Konvertierungsoperator der Elternklasse `Person` auf und nutzen Sie die Klasse `stringstream`, u.a. um eine Konvertierung von `int` nach `string` durchzuführen.

```
1 #include <iostream>
2 #include "Student.cpp"
3
4
5 int main() {
6
7
8     Person* dummy = new Person("dummy", "Schwanzstadt");
9     Student* berkan = new Student(1234556);
10    Student* johannes = new Student(42311);
11    johannes->setName("uganda");
12    johannes->getAdresse()->setOrt("ugandacastle");
13
14
15    cout<< johannes->toString();
16
17    berkan = johannes;
18    berkan->setName("uffuff");
19    // cout << berkan->toString();
20
21    dummy = berkan;
22
23    cout << berkan->toString();
24    cout << dummy->toString();
25
26    return 0;
27 }
28
```

```
1 //  
2 // Created by boery on 27.06.2021.  
3 //  
4 #include "Adresse.cpp"  
5  
6  
7 class Person{  
8 private:  
9     string m_name;  
10    Adresse* m_adresse;  
11  
12 public:  
13     Person(){  
14         Adresse* neueAdresse = new Adresse();  
15         m_adresse=neueAdresse;  
16     };  
17  
18     Person(string name, string adrName){  
19         Adresse* neueAdresse = new Adresse(adrName);  
20         m_adresse= neueAdresse;  
21         m_name=name;  
22     };  
23  
24     // Standardmethoden Copykonstruktor  
25     Person(const Person& copy){  
26         m_name = copy.m_name;  
27         m_adresse = copy.m_adresse;  
28     }  
29  
30  
31     // Clone  
32     virtual Person* clone(){  
33         return new Person(*this);  
34     }  
35  
36  
37  
38 // Zwingend Notwendig fuer Polymorphie  
39     Person& operator=(const Person& rhs){  
40         /*  
41             m_name = copy.m_name;  
42             m_adresse=copy.m_adresse;  
43             return *this;  
44             */  
45             if(this == &rhs) return *this;  
46             return assign(rhs);  
47     }  
48  
49  
50     // Setter  
51     void setName(string name){  
52         m_name = name;  
53     }  
54  
55     // Assing  
56     virtual Person& assign(const Person& rhs){  
57         m_name = rhs.m_name;  
58         m_adresse = rhs.m_adresse;  
59         return *this;  
60     }  
61  
62     // Getter  
63     string getName(){  
64         return m_name;
```

```
65     }
66
67     Adresse* getAdresse(){
68         return m_adresse;
69     }
70
71     //toStringPerson
72     string toString(){
73         stringstream strom;
74         strom << "Name: " << m_name << ". Adresse: " << m_adresse->toString() << endl;
75
76         return strom.str();
77     }
78 };
```

```
1 //  
2 // Created by boery on 27.06.2021.  
3 //  
4 #include <iostream>  
5 #include <sstream>  
6 using namespace std;  
7  
8  
9 struct Adresse{  
10 public:  
11     string m_ort;  
12  
13     // Konstruktor  
14     Adresse():{};  
15  
16     Adresse(string &ort):m_ort(ort){};  
17  
18  
19     // Setter  
20     void setOrt(string ort){  
21         m_ort = ort;  
22     }  
23  
24  
25     // Getter  
26  
27     string getOrt(){  
28         return m_ort;  
29     }  
30  
31     //toString Adresse  
32     string toString(){  
33         stringstream strom;  
34         strom<< m_ort;  
35  
36         return strom.str();  
37     }  
38  
39 };  
40
```

```
1 //  
2 // Created by boery on 27.06.2021.  
3 //  
4 #include "Person.cpp"  
5 #include "Immatrikulation.cpp"  
6  
7 class Student: public Person, Immatrikulation{  
8 private:  
9     int m_matrikelNr;  
10  
11 public:  
12     // StandardMethoden  
13     Student(int matrikelNr):m_matrikelNr(matrikelNr){};  
14  
15     //Kopierkonstruktor  
16     Student(const Student& huso):m_matrikelNr(huso.m_matrikelNr){};  
17  
18  
19     //OPERATORUEBERLADUNG - Beachte Implementierung in Mutterklasse!  
20  
21     Student& operator=(const Student& copy){  
22         if(this==&copy) return *this;  
23         return assign(copy);  
24     }  
25  
26     // Clone  
27     virtual Student* clone() const{  
28         return new Student(*this);  
29     }  
30  
31     // Assign  
32     virtual Student& assign(const Person &rhs){  
33         const Student* neuerStudent = dynamic_cast<const Student*>(&rhs);  
34         if(!neuerStudent) return *this;  
35         Person::assign((rhs));  
36         m_matrikelNr = neuerStudent->m_matrikelNr;  
37         return *this;  
38     }  
39  
40     // Getter  
41     int getMatrikelNr(){  
42         return m_matrikelNr;  
43     }  
44  
45     // Setter  
46     void setMatrikelNr(int neueNr){  
47         m_matrikelNr=neueNr;  
48     }  
49  
50     // Hilfsmethoden  
51     string toString(){  
52         stringstream strom;  
53         strom << "Name: " << getName() << ". Adresse: " << getAdresse()->toString() << ". MatrikelNr  
: " << m_matrikelNr << ". " << endl;  
54  
55         return strom.str();  
56     }  
57 };
```

```
1 cmake_minimum_required(VERSION 3.19)
2 project(VerebungKlausurVB)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(VerebungKlausurVB main.cpp Immatrikulation.cpp Adresse.cpp Person.cpp Student.cpp)
```

```
1 //  
2 // Created by boery on 27.06.2021.  
3 //  
4 class Immatrikulation{  
5 public:  
6     virtual int getMatrikelNr()=0;  
7 };  
8  
9
```

1.3 Operatormethoden [wichtig]

Operatormethoden

Eine Ringliste ist ein logisch ringförmig angelegter Speicherbereich (Anwendungsfälle z.B. Ein-/Ausgabepuffer) fester Größe (Kapazität). Es gibt je einen Marker für die aktuelle Lese- und Schreibposition. Diese stellen den logischen Beginn und das logische Ende der in der Ringliste gespeicherten Datensequenz dar. Beim Beschreiben wird an der Schreibposition ein neuer Wert eingetragen und der Positionsmarker rückt eine Position vor. Beim lesenden Zugriff wird der Wert an der Leseposition ausgelesen und der Marker rückt eine Position vor (damit gilt das ausgelesene Element als aus der Ringliste „entfernt“). Das Vorrücken geschieht zyklisch, d.h. wird von der letzten Position gelesen oder geschrieben, rückt der zugehörige Marker auf die Anfangsposition. Werden mehr Schreiboperationen durchgeführt als die Kapazität des Speichers beträgt, so werden die ältesten eingetragenen Werte mit den neuen überschrieben.

Gegeben sei ein Verwendungsbeispiel einer zu schreibenden Klasse Ringliste (Datei: `RinglisteSpielerei`), die eine Ringliste für integer-Werte realisiert. Die Kapazität wird dabei im Konstruktor gesetzt und besitzt den Default-Wert 10. Die integer-Daten sollen intern in einem `vector`-Container gespeichert werden. Lese- und Schreibposition sollen anfangs mit 0 initialisiert sein. Die Klasse soll alle Standardmethoden (Konstruktoren, Destruktoren, Zuweisungsoperator) ausprogrammiert enthalten, sowie diejenigen Methoden, sodass das Verwendungsbeispiel die Ausgabe

```
0/5 |
3/5 | 1 2 3
5/5 | 2 3 4 5 6
true
3/10 | 9 9 9
5/5 | 5 6 9 9 9
3/10 | 9 9 9
```

produziert.

Hinweise:

- Beachten Sie die zyklische Struktur, bzw. die möglichen Fälle für die Positionen der Lese- und Schreibmarken. Es ist möglich, dass

der Schreibmarker den Lesemarker „überholt“ oder zyklisch an den Anfang springt und dann „vor“ der Lesemarke sitzt. Es ist ebenfalls möglich, dass die beiden Marker an derselben Position stehen: Direkt nach der Objektkonstruktion (oder nach dem Auslesen sämtlicher Elemente) sind solche Ringlisten leer, nach Auffüllen mit der Maximalzahl von Werten aber nicht.

- Der Vergleich zweier Ringlisten ist rein inhaltlich und hängt nur von den in den Ringlisten gespeicherten Werten und ihrer Anordnung ab, nicht von ihrer tatsächlichen Position im `vector`-Container.
- Beachten Sie außerdem, dass die Operation „`+=`“ in diesem Beispiel kein `Ringliste`-Objekt ist, die Operatormethode `operator+=` also von den Konventionen der Standardimplementation abweicht

Prädikate

Schreiben Sie Prädikate (mindestens je ein unäres und ein binäres)

- `istTeilerVon_n`: prüft, ob die übergebene ganze Zahl ein Teiler der im Prädikat hinterlegten Zahl n ist
- `istNahe`: prüft, ob die übergebene double-Zahl sich von der hinterlegten double Zahl um höchstens `double tolerance` unterscheidet. `tolerance` besitzt als Default-Wert 10^{-4} und kann im Konstruktor verändert werden.
- `istKuerzerAls`: prüft für die beiden übergebenen `string`-Objekte, ob der erste übergebene String eine kürzere Länge als der zweite hat.
- `besitztMehrWorteAls`: prüft ob der erste der übergebenen Texte (Typ: `string`) aus mehr Worten besteht als der zweite. Ein Wort ist ein nicht durch Whitespace getrennte Sequenz von mindestens 2 Zeichen.

```
1 #include <iostream>
2 #include "Praedikate.h"
3
4
5 int main() {
6     istKuerzerAls istKuerzerAlstest;
7     istTeilerVon_n teilerVonN(4);
8     istNahe istNahe0001(3.123);
9     istNahe istNahe4125(3.123, 4.1250);
10    besitztMehrWorteAls mehrWorte;
11
12    cout << "istKuerzerAls Test -> " << "Hallo : Tag | " << boolalpha << istKuerzerAlstest("Hallo",
13        "Tag") << endl;
14    cout << "istKuerzerAls Test -> " << "Tag : Hallo | " << istKuerzerAlstest("Tag", "Hallo") <<
15        endl;
16    cout << "istTeilerVon_n Test -> " << "Ist 63 ein Teiler von 4? | " << teilerVonN(63) << endl;
17    cout << "istTeilerVon_n Test -> " << "Ist 40 ein Teiler von 4? | " << teilerVonN(40) << endl;
18    cout << "istNahe Test -> " << "3.123, 2.1234, Tolerance: 0.0001 | " << istNahe0001(2.1234) <<
19        endl;
20    cout << "istNahe Test -> " << "3.123, 2.1234, Tolerance: 4.1250 | " << istNahe4125(2.1234) <<
21        endl;
22    cout << "besitztMehrWorterAls Test -> " << "Hallo was geht? vs Moin | "
23        << mehrWorte("Hallo was geht?", "Moin") << endl;
24    cout << "besitztMehrWorterAls Test -> " << "Moin vs. Hallo was geht? | "
25        << mehrWorte("Moin", "Hallo was geht?") << endl;
26
27    return 0;
28 }
```

```
1 //  
2 // Created by aazat on 28.04.2021.  
3 //  
4 #ifndef INC_1_3_PRAEDIKATE_H  
5 #define INC_1_3_PRAEDIKATE_H  
6  
7  
8 #include <string>  
9 #include <vector>  
10  
11 using namespace std;  
12  
13 class istKuerzerAls {  
14 public:  
15     bool operator()(const string& a, const string& b);  
16 };  
17  
18 class istTeilerVon_n {  
19     int teiler;  
20 public:  
21     explicit istTeilerVon_n(int n) : teiler(n) {}  
22     bool operator()(int zahl);  
23 };  
24  
25 class istNahe {  
26     double tolerance;  
27     double toTest;  
28 public:  
29     explicit istNahe(double test, double tol) : tolerance(tol), toTest(test) {}  
30  
31     explicit istNahe(double test) : toTest(test), tolerance(0.0001) {}  
32  
33     bool operator()(double b);  
34  
35 };  
36  
37 class besitztMehrWorteAls {  
38 public:  
39     static vector<string> cutter(string a);  
40  
41     static int actualWords(vector<string> words);  
42  
43     bool operator()(string a, string b);  
44 };  
45  
46  
47 #endif //UNTITLED2_PRAEDIKATE_H  
48
```

```
1 cmake_minimum_required(VERSION 3.17)
2 project(untilled2)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(untilled2 main.cpp Praedikate.h Praedikate.cpp)
```

```
1 //  
2 // Created by aazat on 28.04.2021.  
3 //  
4  
5 #include "Praedikate.h"  
6  
7 #include <string>  
8 #include <vector>  
9 #include <iostream>  
10 #include <iterator>  
11  
12 using namespace std;  
13  
14 bool istKuerzerAls::operator()(const string &a, const string &b) {  
15     return a.size() < b.size();  
16 }  
17  
18 bool istTeilerVon_n::operator()(int zahl) {  
19     return !(zahl % teiler);  
20 }  
21  
22 bool istNahe::operator()(double b) {  
23     if (tolerance < 0) {  
24         tolerance *= -1;  
25     }  
26     return toTest - tolerance <= b && b <= toTest + tolerance;  
27 }  
28  
29 vector<string> besitztMehrWorteAls::cutter(string str) {  
30     istringstream iss(str);  
31     vector<string> result(istream_iterator<string>{iss}, istream_iterator<string>());  
32     return result;  
33 }  
34  
35 int besitztMehrWorteAls::actualWords(vector<string> words) {  
36     int result = 0;  
37     for (string s : words) {  
38         if (s.size() >= 2)  
39             result++;  
40     }  
41     return result;  
42 }  
43  
44 bool besitztMehrWorteAls::operator()(string a, string b) {  
45     {  
46         vector<string> cutFirst = cutter(a);  
47         vector<string> cutSecond = cutter(b);  
48         return actualWords(cutFirst) > actualWords(cutSecond);  
49     }  
50 };  
51  
52
```

```
1 /*
2 * -----
3 *
4 * Beispielprogramm: RinglisteSpielerei.cpp
5 *
6 * Verwendungsbeispiel der Klasse Ringliste
7 *
8 * Praktikum Programmierung 2, O. Henkel, HS Osnabrueck
9 * -----
10 */
11
12 #include "Ringliste.h"
13 #include <iostream>
14 #include <iomanip>
15 using namespace std;
16
17
18 int main() {
19     Ringliste rListe(5);           // Kapazitaet=5
20     cout << rListe.toString() << endl; // toString-Methode
21
22     rListe << 1 << 2 << 3; // operator<< geeignet ueberladen
23     cout << rListe.toString() << endl;
24     rListe << 4 << 5 << 6;
25     cout << rListe.toString() << endl;
26     Ringliste vergleich(5);
27     vergleich << 2 << 3 << 4 << 5 << 6;
28     // erwartet: true
29     cout << boolalpha << (rListe==vergleich) << endl;
30
31     Ringliste drei; // Kapazitaet=10 (Default)
32     drei << 3 << 3 << 3;
33     drei+=6;          // operator+= geeignet ueberladen
34     cout << drei.toString() << endl;
35
36     rListe << drei; // operator<< geeignet ueberladen
37     cout << rListe.toString() << endl;
38     cout << drei.toString() << endl;
39
40     return 0;
41 }
42
```

```
1
2 #ifndef RINGLISTE_H_INCLUDED
3 #define RINGLISTE_H_INCLUDED
4
5 #include <utility>
6 #include <vector>
7 #include <string>
8 using namespace std;
9
10 class Ringliste {
11     int kapazitaet;
12     bool leer;
13     vector<int> ring;
14     int lesePos;
15     int schreibPos;
16
17     void nextPos(int &) const;
18
19 public:
20
21 //Konstruktoren
22     explicit Ringliste(int _kapazitaet = 10);
23
24     Ringliste(const Ringliste &_rListe);
25
26     ~Ringliste();
27
28 //Informationen
29     int anzElemente() const;
30
31     string toString() const;
32
33 //Vergleiche
34     bool operator==(const Ringliste &) const;
35
36 //Schreib und leseoperation
37     Ringliste &operator<<(int wert);
38
39     Ringliste &operator<<(const Ringliste &);
40
41     Ringliste &operator>>(int &wert);
42
43 //elementeweise Manipulation
44     void operator+=(int);
45
46 };
47
48 #endif
```

```
1 #include "Ringliste.h"
2 #include <iostream>
3 #include <sstream>
4 #include <string>
5
6 //Konstruktoren und Standardmethoden
7 Ringliste::Ringliste(int kapazitaet): kapazitaet(kapazitaet), leer(true), ring(kapazitaet), lesePos(0), schreibPos(0) {}
8
9 //Kopierkonstruktor
10 Ringliste::Ringliste(const Ringliste &_rListe) = default;
11
12 //Destruktor
13 Ringliste::~Ringliste() = default;
14
15 //Positionsinkrement
16
17 string Ringliste::toString() const {
18     stringstream strom;
19     strom << anzElemente() << "/" << kapazitaet << " | ";
20     if (leer){
21         return strom.str();
22     }
23     int pos = lesePos;
24     do {
25         strom << ring[pos] << " ";
26         nextPos(pos);
27     } while (pos != schreibPos);
28     return strom.str();
29 }
30
31 //Vergleiche
32 bool Ringliste::operator==(const Ringliste &rl) const {
33     string s1 = toString();
34     string s2 = rl.toString();
35     return (s1 == s2);
36 }
37
38 //Schreib und Leseoperationen
39 Ringliste &Ringliste::operator<<(int wert) {
40     if (anzElemente() == kapazitaet) nextPos(lesePos);
41     ring[schreibPos] = wert;
42     nextPos(schreibPos);
43     leer = false;
44     return *this;
45 }
46
47 Ringliste &Ringliste::operator>>(int &wert) {
48     if (anzElemente() == 1) leer = true;
49     wert = ring[lesePos];
50     nextPos(lesePos);
51     return *this;
52 }
53
54 Ringliste &Ringliste::operator<<(const Ringliste &rl) {
55     Ringliste rListe(rl);
56     int wert;
57     while (!rListe.leer) {
58         rListe >> wert;
59         operator<<(wert);
60     }
61     return *this;
62 }
63
```

```
64 //Elementweise Manipulation
65 void Ringliste::operator+=(int shift) {
66     if (leer) return;
67     int pos = lesePos;
68     do {
69         ring[pos] += shift;
70         nextPos(pos);
71     } while (pos != schreibPos);
72 }
73
74 void Ringliste::nextPos(int &pos) const {
75     pos = (pos + 1) % kapazitaet;
76 }
77 //Information
78 int Ringliste::anzElemente() const {
79     if (leer) return 0;
80     if (schreibPos == lesePos) return kapazitaet;
81     if (schreibPos > lesePos) return schreibPos - lesePos;
82     else return kapazitaet - (lesePos - schreibPos);
83 }
84
```

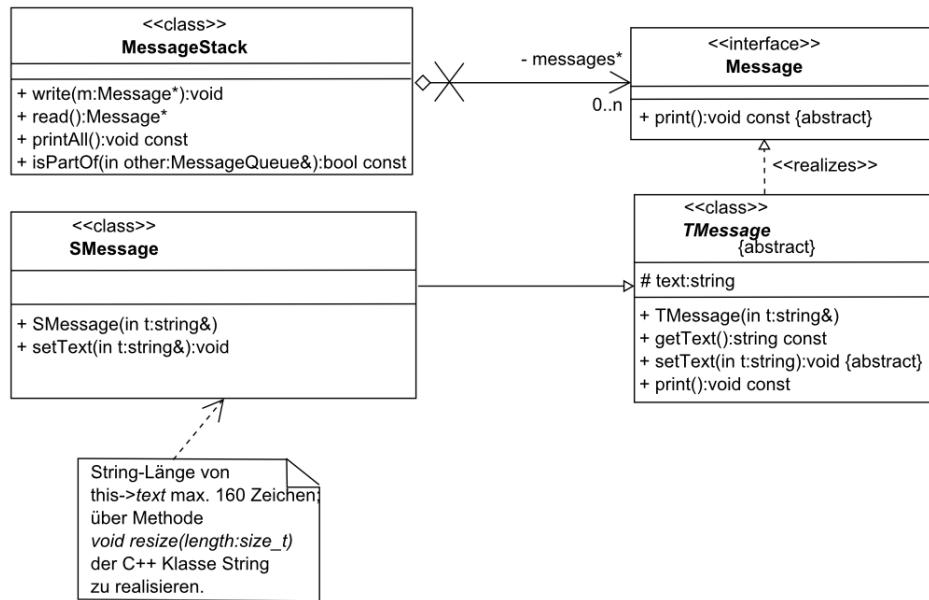
```
1 cmake_minimum_required(VERSION 3.17)
2 project(untilde1)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(untilde1 main.cpp Ringliste.cpp Ringliste.h)
```

Aufgabe 3: Vererbung (30 Punkte)

In dieser Aufgabe sollen für eine Software im Umfeld sozialer Medien einige Klassen definiert werden. Die Idee: ein Client kann in einem Message-Stack unterschiedliche Arten von Nachrichten (=Messages) verwalten und diese ausgeben lassen. Hierzu soll ein Interface namens `Message` mit einer `print`-Methode definiert werden. Das Interface `Message` wird von der abstrakten Klasse `TMessage` (kurz für Textual Message) realisiert. Es soll später mehrere unterschiedliche Arten von Textnachrichten geben. In dieser Aufgabe gibt es genau eine Spezialisierung von `TMessage`, nämlich die konkrete Klasse `SMessage` (kurz für Short Message).

Zur Erinnerung: ein Stack ist ein Container, der nach dem *Last-In First-Out* Prinzip arbeitet. Neue Elemente werden beim schreibenden Zugriff an das Ende des Containers eingefügt. Elemente werden beim lesenden Zugriff dem Ende des Containers entnommen und entfernt. Je später ein Element dem Stack hinzugefügt wird, desto früher wird es wieder entfernt.

Das folgende UML-Diagramm stellt die in dieser Aufgabe zu definierenden Klassen dar. Bei Abweichungen zum Text, sind die Vorgaben des Textes zu priorisieren.



Aufgaben:

- 3.1 Definieren Sie das Interface `Message`, mit einer rein abstrakten öffentlichen Methode `void print() const`. Überlegen Sie, ob das Interface `Message` Konstruktor(en), Destruktor und / oder `operator=` benötigt. Denken Sie zudem darüber nach, ob diese `virtual` sein sollten. Definieren Sie diese falls sie benötigt werden und begründen Ihre Entscheidung kurz.
- 3.2 Definieren Sie die abstrakte Klasse `TMessage` als öffentliche Ableitung von `Message`, mit der einzigen Instanzvariablen (Sichtbarkeit: `protected`):
 - `string text`: enthält den Text der Nachricht
 und folgenden Inline zu definierenden Methoden:
 - `TMessage(const string& t)`: der überladene Konstruktor weist der Instanzvariablen `text` das übergebene `string`-Objekt `t` über die Initialisierungsliste zu.
 - Fügen Sie einen Destruktor ein, wenn Sie der Meinung sind, dass dieser benötigt wird (`virtual?`).

Übungsklausur Fortgeschrittene Programmierung

- Definieren Sie die vom Interface Message geerbte print-Methode, so dass der Text über die Standardausgabe visualisiert wird. Stellen Sie zudem sicher, dass Ableitungen der Klasse die Implementierung erben, aber die Methode nicht überschreiben können.
- Definieren Sie eine get-Methode, über die der Wert der Instanzvariablen text ausgelesen werden kann. Es soll eine Kopie des Wertes zurückgegeben werden. Abgeleitete Klassen erben die Implementierung und sollen diese Methode überschreiben können.
- Deklarieren Sie eine rein virtuelle set-Methode, über die der Wert der Instanzvariablen text geändert werden kann.

3.3 Beantworten Sie die auf dem Lösungszettel definierten Verständnisfragen.

3.4 Definieren Sie die Klasse SMessage (kurz für Short Message) als öffentliche Ableitung der Klasse TMessage ohne zusätzliche Instanzvariablen und mit folgenden Inline-Methoden:

- SMessage (const string& t) : der Konstruktor der Klasse SMessage. Stellen Sie sicher, dass Instanzen dieser Klasse nur Texte mit max. 160 Zeichen verwalten. Verwenden Sie hierzu die vorhandene Methode void resize(size_t n) der Klasse string (s. Anlage 1). Überlegen Sie, ob und wie der Konstruktor von TMessage aufzurufen ist. Erläutern Sie kurz.
- Definition der geerbten setText-Methode. Stellen Sie sicher, dass Instanzen dieser Klasse nur Texte verwalten, die max. 160 Zeichen lang sind. Verwenden Sie hierzu wieder die vorhandene Methode void resize(size_t n) der Klasse string (s. Anlage 1).
- Frage: was passiert, wenn die setText-Methode in der Klasse SMessage nicht definiert wird? Erläutern Sie kurz.

3.5 Gegeben ist folgende Klasse MessageStack. Definieren Sie die deklarierten Methoden außerhalb des Klassenblocks.

```
class MessageStack {  
private:  
    vector<Message*> messages;  
public:  
    void write(Message* m);  
    Message* read();  
    void printAll() const;  
    bool isPartOf(const MessageStack& o) const;  
};
```

Erläuterung der Methoden:

- void write(Message* m) : fügt den übergebenen Zeiger auf ein Objekt der Klasse Message m als flache Kopie der Instanzvariablen messages hinzu.
- Message* read() : liest das letzte Element des vector-Containers messages aus, entfernt dieses aus dem Container und gibt das Element als flache Kopie zurück.
- void printAll() const: gibt alle Objekte auf die die Elemente des vector-Containers messages zeigen über die Standardausgabe aus und verwendet dazu die print-Methode der Instanzen des Interfaces Message.
- bool isPartOf(const MessageStack& o) : prüft, ob identische Instanzen von this->messages mindestens einmal in o.messages vorkommen (es wird keine Prüfung auf Wertegleichheit gefordert!). Ist dies der Fall, wird true zurückgegeben. Ansonsten false.

```
1
2 // #include "TMessage.cpp"
3 #include "MessageStack.cpp"
4
5
6 using namespace std;
7
8 int main() {
9
10    TMessage* msg1 = (TMessage *) "HalloDuEumel";
11    TMessage* msg2 = (TMessage *) "AMk";
12    TMessage* msg3 = (TMessage *) "Biermann ist der beste man der welt";
13
14    TMessage* msg4 = (TMessage *) "asdasd";
15    TMessage* msg5 = (TMessage *) "Apfelmuss";
16    TMessage* msg6 = (TMessage *) "qweqwe";
17
18    // SMessage berkan(msg1);
19    MessageStack klotz;
20    MessageStack knecht;
21
22    knecht.write(msg1);
23
24
25    klotz.write(msg1);
26    klotz.write(msg2);
27    klotz.write(msg3);
28    klotz.write(msg4);
29    klotz.write(msg5);
30    klotz.write(msg6);
31    //msg5->print();
32    // klotz.printAll();
33
34    cout<< boolalpha << klotz.isPartOf(knecht);
35
36    // Alles laeuft ausser Print
37
38    // 8/10 Punkte
39
40    return 0;
41 }
42
```

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4 #include <iostream>  
5 using namespace std;  
6  
7 // Abstrakte Klasse weil alle Methoden virtual sind  
8 class Message{  
9 public:  
10     Message(){  
11         cout << "Default Konstruktor von MSG" << endl;  
12     }  
13  
14     virtual string const print() = 0; // rein abstrakte Methode  
15  
16     virtual ~Message(){  
17         cout << "Destruktor von MSG" << endl;  
18     }  
19 };  
20  
21
```

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4  
5 #include "TMessage.cpp"  
6  
7 class SMessage : public TMessage{  
8 public:  
9  
10    SMessage(string& t);  
11  
12  
13    virtual ~SMessage(){  
14        cout << endl;  
15        cout << "Destruktor von SMSG" << endl;  
16    };  
17  
18  
19    void setText(string t);  
20};  
21  
22  
23 inline SMessage::SMessage(string &t) {  
24    if(t.size() > 10){  
25        t.resize(10);  
26        m_text = t;  
27    } else{  
28        m_text = t;  
29    }  
30    cout << "Konstruktor von SMSG" << endl;  
31}  
32  
33 inline void SMessage::setText(string t) {  
34    if(t.size() > 10){  
35        t.resize(10);  
36        m_text = t;  
37    } else{  
38        m_text = t;  
39    }  
40}
```

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4 #include "Message.cpp"  
5  
6 #include <sstream>  
7  
8  
9 class TMessage : public Message{  
10 protected:  
11     string m_text;  
12  
13 public:  
14     TMessage(){  
15         cout << "Default Konstruktor von TMSG" << endl;  
16     };  
17     TMessage(string &text);  
18     virtual ~TMessage(){  
19         cout << "Destruktor von TMSG" << endl;  
20     }  
21  
22     // Getter  
23     const string getText(){  
24         return m_text;  
25     }  
26  
27     // Setter  
28     virtual void setText(string text) = 0;  
29  
30  
31  
32     const string print(){  
33  
34         stringstream strom;  
35         strom << m_text;  
36         return strom.str();  
37     }  
38  
39  
40  
41 };  
42  
43 inline TMessage::TMessage(string &text): m_text(text){};  
44  
45
```

```
1 cmake_minimum_required(VERSION 3.19)
2 project(Vorbereitung2Vererbung)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(Vorbereitung2Vererbung main.cpp Message.cpp TMessage.cpp SMessage.cpp MessageStack.cpp)
```

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4  
5 //#include "Message.cpp"  
6 #include "SMessage.cpp"  
7 #include <vector>  
8  
9 class MessageStack{  
10 private:  
11     vector<Message*> messages;  
12  
13 public:  
14     void write(Message* m){  
15         // Flache kopie uebergeben  
16         messages.push_back(m);  
17     }  
18  
19  
20     Message* read(){  
21         Message* tmp;  
22         tmp = messages.back();  
23         messages.pop_back();  
24         return tmp;  
25     }  
26  
27  
28  
29  
30     void printAll() const{  
31         for(int i =0;i<messages.size();i++){  
32             cout << messages.at(i)->print() << endl;  
33             //cout << "cock" << endl;  
34         }  
35     }  
36  
37  
38     bool isPartOf(const MessageStack& o) const{  
39         for(int i =0; i<messages.size();i++){  
40             for (int j = 0; i<o.messages.size();j++){  
41                 if(messages.at(i)==o.messages.at(j)){  
42                     return true;  
43                 }  
44             }  
45         }  
46         return false;  
47     }  
48 };
```

2.1 Fifo [wichtig]

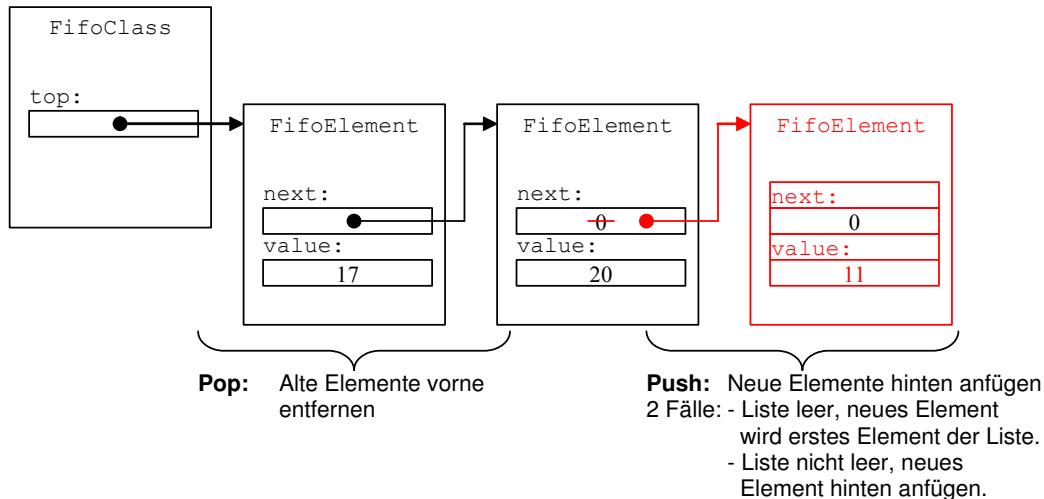
Erstellen Sie Klasse **Fifo**, die einen Fifo-Speicher für Objekte vom Typ `string` mit den Standardmethoden und mindestens folgenden öffentlichen Methoden realisiert:

Fifo()	erzeugt ein leeres Fifo
~Fifo()	Destruktor
Fifo& operator<<(const string&)	schreibt eine Kopie des <code>string</code> -Objektes in das Fifo ein „überlicherweise als <code>push</code> bezeichnet“
Fifo& operator>>(string&)	liest ein <code>string</code> -Objekt aus dem Fifo aus „überlicherweise als <code>pop</code> bezeichnet“
operator int () const	gibt den Füllstand des Fifos zurück

Ausnahmen sollen jeweils durch Werfen einer Zeichenkettenkonstante abgefangen werden. Neben unbekannten Ausnahmen sind insbesondere Ausnahmen für Fehler bei der dynamischen Speicherbeschaffung ^{>a} Fehler beim Öffnen von nichtexistierenden Dateien, sowie der Fifo-Unterlauf zu fangen.

Die einzelnen Elemente des Fifos sollen Objekte einer von Ihnen zu schreibenden Klasse **FifoElement** sein, die als verzeigerte Liste verkettet den Fifo im Speicher dynamisch abbilden. Nachstehendes Diagramm zeigt schematisch den Aufbau, sowie die grundlegenden Fifo-Operationen anhand eines Fifos für integer-Zahlen

^{>a}`new` wirft bei Fehlern Ausnahmeobjekte vom Typ `bad_alloc`, wenn der Header `new` eingebunden ist. In so einem Fall soll die Ausnahme lokal gefangen und stattdessen eine passende Fehlermeldung weitergeworfen werden



Quelle: Prof. Dr. B. Lang, HS Osnabrück

Die Fifo-Klasse muss mit dem Testprogramm `FifoClassTest.cpp` durchgetestet werden und die Ausgabe `Ausgabe.FifoClassTest` produzieren. Dem Testprogramm entnehmen Sie ebenfalls ggf. zu erstellende öffentliche Methoden der Fifo-Klasse.

Zusatzaufgabe: Überladen Sie den `<<`-Operator zusätzlich so, dass er als Argument auch Fifo-Objekte (als Referenz) akzeptiert und dessen Elemente in `*this` einfügt.

Hinweise: Für die Bearbeitung ist die Verwendung der STL nicht erlaubt. Vielmehr soll der Fifo mit einer von Ihnen geschriebenen dynamischen Speicherverwaltung versehen werden. Achten Sie darauf, dass Kopierkonstruktor und Zuweisungsoperator tiefe Kopien erzeugen und keine Speicherlecks entstehen.

1 So kann also die Mathematik definiert werden als diejenige
2 Wissenschaft, in der wir niemals das kennen, worüber wir sprechen, und
3 niemals wissen, ob das, was wir sagen, wahr ist.
4
5

```
1
2 #ifndef FIFO_FIFO_H
3 #define FIFO_FIFO_H
4
5 #endif //FIFO_FIFO_H
6 /* Includes */
7 #include <iostream>
8 #include "FifoElement.h"
9
10 /* usings */
11 using std::cout;
12 using std::cin;
13 using std::string;
14
15
16 class Fifo {
17     FifoElement *head{nullptr};
18     FifoElement *tail{nullptr};
19     int anzahl{0};
20 public:
21
22
23     /* Standardkonstruktor */
24     Fifo();
25
26     /* Copy-Konstruktor */
27     Fifo(Fifo &copy);
28
29     /* Destruktor */
30     ~Fifo();
31
32     void push(const string& text);
33
34     string pop();
35
36     Fifo operator<<(const string& text);
37
38     Fifo operator>>(string &text);
39
40     operator int();
41
42     int size() const;
43
44     void info() const;
45
46 };
47
48
49
50
```

```

1 #include "Fifo.h"
2 #include <iostream>
3
4 using std::string;
5
6 Fifo::Fifo() : head(nullptr), tail(nullptr), anzahl(0) {};
7
8 Fifo::~Fifo() = default;
9
10 Fifo::Fifo(Fifo &copy) {
11     FifoElement *newFifoElement = copy.head; // Initial erstmal ein Temp
12     FifoElement erstellen. // Solange das neue FifoElement auf
13     while (newFifoElement != nullptr) { // Dann inserte mit der Methode
14         push(newFifoElement->getMyString()); // ein neues FifoElement zeigt
15         push // kein Nullptr zeigt
16         newFifoElement = newFifoElement->getNext(); // nun auf next damit es zu einem "Zug" kommt
17     }
18
19
20 void Fifo::push(const string& text) { // Methode fuer das Pushen
21     FifoElement *neuesElement = new FifoElement(text); // Initial erstmal ein Temp FifoElement
22     erstellen. // Wenn kein FifoElement in der
23     if (head == nullptr) { // Dann ist es das Head-Element
24         Liste ist // und gleichzeitig das Tail-Element
25         head = neuesElement;
26         tail = neuesElement;
27     } else { // Man ueberschreibt den Nullpointer vom
28         tail->setNext( // Das neue FifoElement was erstellt
29             neuesElement); // Globale variable die hochgezaehlt wird
30         ende "next" des tails. Der bekommt das neue Ende.
31         tail = neuesElement; // falls man die Anzahl der FifoElemente braucht
32     }
33     anzahl++; // falls man die Anzahl der FifoElemente braucht
34
35     string Fifo::pop() { // Globale variable die hochgezaehlt wird
36         if (head == nullptr) { // Wenn kein FifoElement in der
37             throw "Es gibt nix in der Liste"; // Dann ist es das Head-Element
38         }
39         string ausgabe; // und gleichzeitig das Tail-Element
40         ausgabe = head->getMyString(); // Man ueberschreibt den Nullpointer vom
41         head = head->getNext(); // Das neue FifoElement was erstellt
42         anzahl--;
43         return ausgabe; // Globale variable die hochgezaehlt wird
44     }
45
46 Fifo Fifo::operator<<(const string& text) { //help reference
47     push(text); //help aus der Main wird zu dem
48     return *this; //help reference
49 }
50
51 Fifo Fifo::operator>>(string &text) { //help reference
52     text = pop(); //help aus der Main wird zu dem
53     return *this; //help reference
54 }

```

```
55 int Fifo::size() const{                                     // Gibt den Fuellstand des
    Fifos aus.
56     return anzahl;
57 }
58
59 Fifo::operator int() {
60     return size();
61 }
62
63 void Fifo::info() const{
64     FifoElement *newFifoElement = this->head;
65     while (newFifoElement != nullptr) {
66         std::cout << newFifoElement->getMyString() << std::endl;
67         newFifoElement= newFifoElement->getNext();
68     }
69 }
```

```

1 /*
2 * -----
3 *
4 * Programm: FifoTest.cpp
5 *
6 * Demonstration und Pruefung der Klasse Fifo
7 *
8 * Ursprung: Prof. Dr. B. Lang
9 *
10 * Praktikum Programmierung 2, O. Henkel, HS Osnabrueck
11 * -----
12 */
13
14 #include <iostream>
15 #include <string>
16 #include <fstream>
17 #include "Fifo.h"
18
19 using namespace std;
20
21
22 /*
23 * Testprogramm: Textdatei bitte als Argument uebergeben
24 *                 Als Default wird die Datei zitat.txt verwendet
25 */
26 int main(int nParameter, char *parameter[]) {
27
28     //
29     // Normaler Betrieb
30     //
31     cout << "*** Zunaechst wird der normale Betrieb der Fifo-Klasse abgetestet ***" << endl;
32     try {
33         Fifo s; // Leeres Fifo erstellen
34         cout << "> Fifo aus Textdatei fuellen" << endl;
35         {
36             string token;
37             ifstream in((nParameter > 1) ? parameter[1] : "zitat");
38             if (!in) throw "Kann Datei nicht oeffnen";
39             while (1) {
40                 in >> token;
41                 if (in.eof()) break;
42                 s << token;
43             }
44         }
45
46         cout << "> Fifo ausgeben" << endl;
47         while (s > 0) { // Elemente ausgeben
48             cout << s.pop() << endl;
49         }
50
51         cout << "> Fifo mittels Ein- und Ausgabeoperator fuellen" << endl;
52         s << "One";
53         s << "Two";
54         s << "Three";
55
56         cout << "> tiefe Kopie anlegen und zuweisen" << endl;
57         Fifo kopie1(s);
58         Fifo kopie2;
59         kopie2 = s;
60         kopie2 = kopie2;
61         kopie1 << "Four";
62         kopie2.push("Five");
63         cout << "> Fifo-Kopien ausgeben (die erste drei Werte muessen jeweils gleich sein)" << endl;
64         string help;

```

```

65      cout << " s:" << endl;
66      s >> help;
67      cout << help << endl;
68      s >> help;
69      cout << help << endl;
70      s >> help;
71      cout << help << endl;
72      cout << "Fifo sollte nun leer sein - tatsaechliche Anzahl der Elemente: " << s.size() <<
    endl;
73      cout << " kopie1:" << endl;
74      kopie1 >> help;
75      cout << help << endl;
76      kopie1 >> help;
77      cout << help << endl;
78      kopie1 >> help;
79      cout << help << endl;
80      kopie1 >> help;
81      cout << help << endl;
82      cout << "Fifo sollte nun leer sein - tatsaechliche Anzahl der Elemente: " << kopie1.size
() << endl;
83      cout << " kopie2:" << endl;
84      kopie2.info();
85      cout << "Fifo sollte noch alle Elemente enthalten - tatsaechliche Anzahl der Elemente: " <<
kopie2.size()
86          << endl;
87
88 } catch (const char *error) {
89     cout << "> Failure: Fifo Objekt sollte hier keine Ausnahme werfen:" << endl;
90     cout << " >> " << error << endl;
91     return 3;
92 } catch (...) {
93     cout << "> Failure: Unbekannte Ausnahme" << endl;
94     return 3;
95 }
96
97 //
98 // Fifo Unterlauf testen
99 //
100 cout << "*** Es werden nun Fifo-Unterlaufe passieren ***" << endl;
101 {
102     Fifo s; // Leeres Fifo einrichten
103     try {
104         cout << s.pop() << endl;
105     } catch (const char *error) {
106         cout << "> Erwartete Ausnahme wegen Fifo-Unterlauf:" << endl;
107         cout << " >> " << error << endl;
108     } catch (...) {
109         cout << "> Failure: Unbekannte Ausnahme" << endl;
110         return 3;
111     }
112     try {
113         string help;
114         cout << "> Fifo mittels Ein- und Ausgabeoperator fuellen" << endl;
115         s << "Eins";
116         s << "Zwei";
117         s << "Drei";
118         cout << "> Fifo ausgeben und Unterlauf herbeifuehren" << endl;
119         s >> help;
120         cout << help << endl;
121         s >> help;
122         cout << help << endl;
123         s >> help;
124         cout << help << endl;
125         s >> help;

```

```
126         cout << help << endl; // Hier muss ein Unterlauf passieren
127     } catch (const char *error) {
128         cout << "> Erwartete Ausnahme wegen Fifo-Unterlauf:" << endl;
129         cout << " >>> " << error << endl;
130     } catch (...) {
131         cout << "> Failure: Unbekannte Ausnahme" << endl;
132         return 3;
133     }
134 }
135 return 0;
136
137 }
138
```

1 So kann also die Mathematik definiert werden als diejenige
2 Wissenschaft, in der wir niemals das kennen, worüber wir sprechen, und
3 niemals wissen, ob das, was wir sagen, wahr ist.
4
5

```
1
2 #ifndef FIFO_FIFOELEMENT_H
3 #define FIFO_FIFOELEMENT_H
4
5 #endif //FIFO_FIFOELEMENT_H
6
7 #include <iostream>
8
9 using std::string;
10
11 class FifoElement {
12     FifoElement *next;
13     string myString;
14 public:
15     FifoElement(string text);
16     ~ FifoElement();
17
18     FifoElement *getNext() const;
19
20     void setNext(FifoElement *next);
21
22     const string &getMyString() const;
23
24 };
25
26
27
28
29
30
```

```
1 cmake_minimum_required(VERSION 3.17)
2 project(untilled4)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(untilled4 main.cpp Fifo.cpp Fifo.h FifoElement.cpp FifoElement.h)
```

```
1
2 #include "FifoElement.h"
3
4 FifoElement::FifoElement(string text) {
5     this->myString = text;
6     this->next = nullptr;
7 }
8
9 FifoElement::~ FifoElement() = default;
10
11 FifoElement *FifoElement::getNext() const {
12     return next;
13 }
14
15 void FifoElement::setNext(FifoElement *next) {
16     FifoElement::next = next;
17 }
18
19 const string &FifoElement::getMyString() const {
20     return myString;
21 }
22
```

```
1 #include <iostream>
2 #include "FifoClass.cpp"
3
4 int main() {
5
6     FifoClass fifo;
7
8     FifoElement* el1 = new FifoElement(1);
9     FifoElement* el2 = new FifoElement(2);
10    FifoElement* el3 = new FifoElement(69);
11
12    fifo.push(el1);
13    fifo.push(el2);
14    fifo.push(el3);
15
16    fifo.delete_position(2);
17
18
19
20
21    fifo.toString();
22
23    return 0;
24 }
25
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4  
5 #include "FifoElement.cpp"  
6 #include <iostream>  
7  
8 using namespace std;  
9  
10 class FifoClass{  
11 public:  
12     FifoElement* m_head;  
13     FifoElement* m_tail;  
14  
15  
16     FifoClass(){  
17         m_head = nullptr;  
18         m_tail = nullptr;  
19     }  
20  
21     void push(FifoElement* neuesElement){  
22         if(m_head == nullptr && m_tail== nullptr){  
23             m_head = neuesElement;  
24             m_tail = neuesElement;  
25         } else{  
26             m_tail->m_next = neuesElement;  
27             m_tail = neuesElement;  
28             neuesElement->m_next = nullptr;  
29         }  
30     }  
31  
32     void pop(){  
33         if(m_head==m_tail){  
34             m_head= nullptr;  
35             m_tail= nullptr;  
36         }  
37         if(m_head!= nullptr&&m_head!=m_tail){  
38             FifoElement* tmp = new FifoElement();  
39             tmp=m_head;  
40             m_head=m_head->m_next;  
41             delete(tmp);  
42         }  
43     }  
44 }  
45  
46  
47     // Loeschen an Position  
48     void delete_position(int pos){  
49         FifoElement *tmp=new FifoElement;  
50         FifoElement *prev=new FifoElement;  
51         tmp=m_head;  
52         for(int i=1;i<pos;i++){  
53             prev=tmp;  
54             tmp=tmp->m_next;  
55         }  
56         prev->m_next=tmp->m_next;  
57     }  
58  
59  
60     void toString(){  
61         FifoElement* tmp= new FifoElement();  
62         tmp=m_head;  
63         while(tmp != nullptr){  
64             cout<<tmp->m_value<<endl;
```

```
65         tmp=tmp->m_next;
66     }
67 }
68
69 };
```

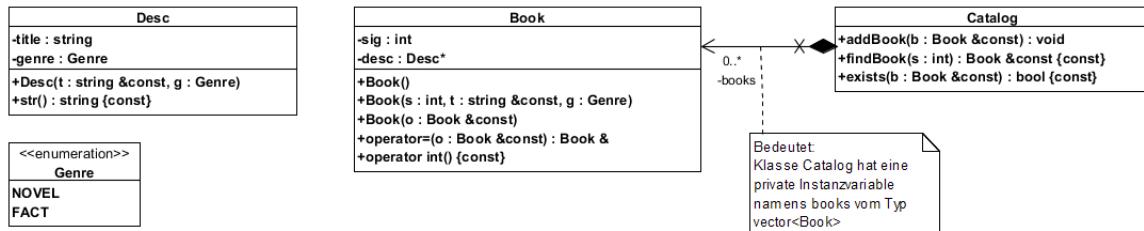
```
1 cmake_minimum_required(VERSION 3.19)
2 project(FifoWiederholung)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(FifoWiederholung main.cpp FifoClass.cpp FifoElement.cpp)
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4  
5 class FifoElement{  
6 public:  
7     FifoElement(){};  
8     FifoElement* m_next=nullptr;  
9     int m_value;  
10  
11    FifoElement(int value){  
12        m_value = value;  
13    }  
14  
15 };  
16
```

Teil II: Objektorientierte Programmierung (Programmiersprache C++)

Aufgabe 3: Klassen im Zusammenspiel, Ausnahmebehandlung (30 Punkte)

In dieser Aufgabe geht es darum, für eine Bibliothekssoftware mit C++ die Klassen `Book`, `Desc` (Kurzform für `Description`, also Buchbeschreibung) und `Catalog` zu definieren. Verwenden Sie das Schlüsselwort `const` wann immer möglich und sinnvoll.



Aufgaben:

- 3.1 Definieren Sie einen Aufzählungstyp (engl. Enumeration) namens `Genre` mit den beiden Konstanten `NOVEL=0` (deutsch Roman) und `FACT=1` (deutsch Sachbuch).
- 3.2 Definieren Sie die Klasse `Desc`. Ein Objekt der Klasse hält beschreibende Informationen eines Buch-Exemplars.

Private Instanzvariablen:

- `title` vom Typ `string` und
- `genre` vom Typ `Genre`.

Öffentliche Methode (Inline zu definieren):

- `Desc(const string& t, Genre g)`: die Zuweisung der Übergabeparameter soll über die Initialisierungsliste erfolgen.
- die konstante Methode `str()`, über die der Titel und das Genre als String-Objekt zurückgegeben werden. Folgendes Format soll verwendet werden: <Titel>, <Genre>, also z.B.: The C++ Developer, Fact.

- 3.3 Vervollständigen Sie die Definition der Klasse `Book`.

Private Instanzvariablen:

- `int sig`: dies ist die Signatur eines Buch-Exemplars.
- `Desc* desc`: diese Instanzvariable hält einen Zeiger auf ein Objekt vom Typ `Desc` mit den beschreibenden Informationen eines Buches.

Öffentliche Methoden (außerhalb des Klassenblocks zu definieren):

- Default-Konstruktor: weist `sig` den Wert 0 und `desc` einen NULL-Pointer zu.
- `Book(int s, const string& t, Genre g)`: der Parameter `s` ist der Instanzvariablen `sig` zuzuweisen. Erzeugen Sie anschließend dynamisch ein Objekt der Klasse `Desc` und weisen die Adresse der Instanzvariablen `desc` zu.
- Definieren Sie den Destruktor und stellen Sie sicher, dass dynamisch allokiert Speicher freigegeben wird.
- Definieren Sie den Zuweisungsoperator. Den Instanzvariablen sind tiefe Kopien der Variablen des übergebenen Objektes zuzuweisen. Gehen Sie davon aus, dass der Kopier-Konstruktor der Klasse `Desc` vorhanden ist. Achten Sie darauf, dass keine Speicherlecks entstehen.

Übungsklausur Fortgeschrittene Programmierung

- Definieren Sie den Kopierkonstruktor. Vermeiden Sie Code-Duplizierung und rufen den Zuweisungsoperator auf.
Vorgabe: stellen Sie sicher, dass der Zuweisungsoperator in jedem Fall korrekt durchläuft.
- Definieren Sie den Konvertierungsoperator nach `int`, der den Wert von `sig` zurückgibt.

3.4 Definieren Sie die Klasse Catalog.

- Die Klasse hat genau eine Instanzvariable namens `books` vom Typ `vector<Book>`.

Es sollen folgende Instanzmethoden Inline definiert werden:

- die `findBook(int s)`-Methode durchsucht den Container `books` und gibt eine konstante Referenz auf ein Objekt vom Typ `Book` zurück, wenn der Wert von `this->sig` mit dem Wert des übergebenen Parameters `s` identisch ist.
Wird kein `Book`-Objekt gefunden, soll eine `invalid_argument` `Exception` geworfen werden.
Hinweis: Um die Signatur eines `Book`-Objektes zu erhalten, können Sie den Konvertierungsoperator nach `int` der Klasse `Book` nutzen.
- der `exists`-Methode wird eine konstante Referenz auf ein vorhandenes `Book`-Objekt übergeben. Diese Methode verwendet `findBook()`. Wird eine `invalid_argument` `Exception` gefangen, so gibt diese Methode `false` zurück, ansonsten `true`.
Hinweis: Um die Signatur eines `Book`-Objektes zu erhalten, können Sie den Konvertierungsoperator nach `int` der Klasse `Book` nutzen.

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4 #include "Desc.cpp"  
5 class Book{  
6 private:  
7  
8     Desc* m_desc;  
9  
10 public:  
11     int m_sig;  
12     Book();  
13     Book(int s, const string& t, Genre g );  
14     Book(const Book& copy);  
15     Book& operator=(const Book& copy);  
16     Book& assign(const Book& copy);  
17     ~Book();  
18     operator int() const;  
19  
20     string toString(){  
21         stringstream strom;  
22         strom << m_desc->toString() << " " << m_sig << endl;  
23         return strom.str();  
24     }  
25  
26  
27 };  
28  
29 inline Book::Book() {  
30     m_sig = 0;  
31     m_desc = nullptr;  
32 }  
33  
34 inline Book::Book(int s, const string &t, Genre g) {  
35     m_sig = s;  
36     Desc* tmp = new Desc(t, g);  
37  
38     m_desc = tmp;  
39 }  
40  
41 inline Book::~Book(){  
42     m_desc= nullptr;  
43     delete (m_desc);  
44 }  
45  
46 inline Book & Book::operator=(const Book &copy) {  
47     if(this==&copy) return *this;  
48     return assign(copy);  
49 }  
50 }  
51  
52 // Speicherfreigabe in assign  
53 inline Book & Book::assign(const Book &copy) {  
54     // Der ist sicher, muss nicht geloescht werden  
55     m_sig=copy.m_sig;  
56  
57  
58     // Neues DESC erstellen!  
59     Desc* tmp;  
60     tmp=copy.m_desc;  
61  
62     //Altes DESC löschen!  
63     m_desc= nullptr;  
64     delete m_desc;
```

```
65
66     //Weisen neue DESC(tmp) this.m_desc zu
67     this->m_desc=tmp;
68
69     return *this;
70 }
71
72 inline Book::Book(const Book &copy) {
73     *this = copy;
74 }
75
76 inline Book::operator int() const {
77     return m_sig;
78 }
```

```
1 //  
2 // Created by boery on 17.07.2021.  
3 //  
4  
5 #include <iostream>  
6 #include <sstream>  
7  
8 using namespace std;  
9  
10 enum Genre{NOVEL = 0, FACT = 1};  
11  
12  
13 class Desc{  
14 private:  
15     string m_title;  
16     Genre m_genre;  
17  
18 public:  
19  
20     Desc(const string& t, Genre g);  
21     string toString();  
22     ~Desc(){cout<<"Desc Destruktor aufgerufen" << endl;}  
23  
24 };  
25  
26 inline Desc::Desc(const string &t, Genre g):m_genre(g),m_title(t){}  
27  
28 inline string Desc::toString() {  
29     stringstream strom;  
30     strom << m_title << ", " << m_genre << endl;  
31     return strom.str();  
32 }
```

```
1 #include <iostream>
2 #include "Catalog.cpp";
3
4
5
6 int main() {
7
8     Book DerHobitt(1,"Hobbit",FACT);
9     Book HerrDerRinge(2, "HerrDerRinge", NOVEL);
10    Book hallo(DerHobitt);
11
12    Catalog kat;
13
14    kat.addBook(DerHobitt);
15    //TRY AND CATCH VON INVALID EXCEPTION AUS class 'BuchNichtVorhanden: public std::exception'
16    try{
17        cout << boolalpha << kat.findBook(4);
18    } catch (BuchNichtVorhanden&){
19        cout <<"Buch existiert nicht du kek" << endl;
20    };
21
22
23
24    cout << boolalpha << kat.exists(HerrDerRinge) << endl;
25
26    return 0;
27 }
28
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4  
5 #include "Book.cpp"  
6 #include <Vector>  
7  
8 class BuchNichtVorhanden: public exception {  
9 public:  
10     BuchNichtVorhanden() = default;  
11 };  
12  
13  
14 class Catalog{  
15 private:  
16     vector<Book> m_books;  
17  
18     // Konstante Referenz  
19  
20 public:  
21     const Book& findBook(int s);  
22     void addBook(const Book& b);  
23     const bool exists(const Book& b);  
24 };  
25  
26  
27 inline const Book& Catalog::findBook(int s) {  
28     bool isInList=false;  
29     for(int i=0;i<m_books.size();i++){  
30         if(m_books.at(i).m_sig==s){ // KONVERTIERUNGSOOPERATOR AUS BOOK int(): '(m_books.at(i))'  
31             isInList=true;  
32             return m_books.at(i);  
33         }  
34     }  
35     //INVALID EXCEPTION THROW  
36     if(isInList==false){  
37         throw BuchNichtVorhanden();  
38     }  
39 }  
40  
41  
42 inline void Catalog::addBook(const Book& b){  
43     m_books.push_back(b);  
44 }  
45  
46 //METHODE MIT USE VON findBook - exception wird geworfen wenn findBook nichts findet  
47 inline const bool Catalog::exists(const Book &b) {  
48     try{  
49         if(findBook(b)) return true;  
50     } catch (BuchNichtVorhanden&){  
51         return false;  
52     }  
53 }  
54 }
```

```
1 cmake_minimum_required(VERSION 3.19)
2 project(Klausurvorbereitung3Klassen)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(Klausurvorbereitung3Klassen main.cpp Desc.cpp Book.cpp Catalog.cpp)
```

3.1 GeoObjekt [wichtig]

Implementieren Sie nachstehendes Klassendiagramm.

- **Punkt** ist ein Interface zur Bereitstellung von Punkten in der Ebene oder im Raum
- **MetrikVerhalten** ist ein Interface, dessen Implementationen die Berechnung des euklidischen Abstands zweier Punkte durch die Methode `abstand` bereitstellt
- **GeoObjekt** ist eine abstrakte Basisklasse. Das geschützte Element `metrik` ermöglicht in den konkreten Ableitungen passende Abstandsbestimmungen. Die öffentliche rein virtuelle Methode `inhalt` soll in den Ableitungen den Flächeninhalt bzw. das Volumen berechnen.
- Zu diesem Zweck besitzen die konkreten Ableitungen **Rechteck**, **Kreis**, **Quader**, **Ball** entsprechende Daten und Methoden

Die konkreten GeoObjekte sollen polymorph kopiert- und zugewiesen werden können. Die zugehörigen Standardmethoden müssen also noch zusätzlich implementiert werden. Inkompatible Zuweisungen sollen durch Werfen eines Ausnahmeobjektes vom Typ **OperandenPassenNicht** (leere Ausnahmeklasse) abgefangen werden.

Das Testprogramm `main.cpp` soll die folgende Ausgabe produzieren (die Zahl hinter den eckigen Klammern ist jeweils der berechnete Inhalt).

```
tatischer Test
Rechteck: [(0, 0), (2, 1)] 2

Polymorphietest - tiefe Kopie mittels clone
Rechteck: [(0, 0), (2, 1)] 2
Rechteck: [(-1, -1), (2, 1)] 6

Polymorphietest - tiefe Kopie mittels assign
Rechteck: [(0, 0), (2, 1)] 2
Rechteck: [(0, 0), (2, 2)] 4

Kreise
-----
statischer Test
```

```
Kreis: [(1, 1), 2] 12.5664
```

```
Polymorphietest - tiefe Kopie mittels clone
```

```
Kreis: [(1, 1), 2] 12.5664
```

```
Kreis: [(1, 1), 1] 3.14159
```

```
Polymorphietest - tiefe Kopie mittels assign
```

```
Kreis: [(1, 1), 2] 12.5664
```

```
Kreis: [(0, 0), 2] 12.5664
```

```
erwartete Ausnahme geworfen
```

```
erwartete Ausnahme geworfen
```

```
Quader
```

```
-----
```

```
statischer Test
```

```
Quader: [(0, 0, 0), (2, 1, 2)] 4
```

```
Polymorphietest - tiefe Kopie mittels clone
```

```
Quader: [(0, 0, 0), (2, 1, 2)] 4
```

```
Quader: [(-1, -1, -1), (2, 1, 2)] 18
```

```
Polymorphietest - tiefe Kopie mittels assign
```

```
Quader: [(0, 0, 0), (2, 1, 2)] 4
```

```
Quader: [(0, 0, 0), (2, 2, 2)] 8
```

```
erwartete Ausnahme geworfen
```

```
Baelle
```

```
-----
```

```
statischer Test
```

```
Ball: [(1, 1, 1), 2] 33.5103
```

```
Polymorphietest - tiefe Kopie mittels clone
```

```
Ball: [(1, 1, 1), 2] 33.5103
```

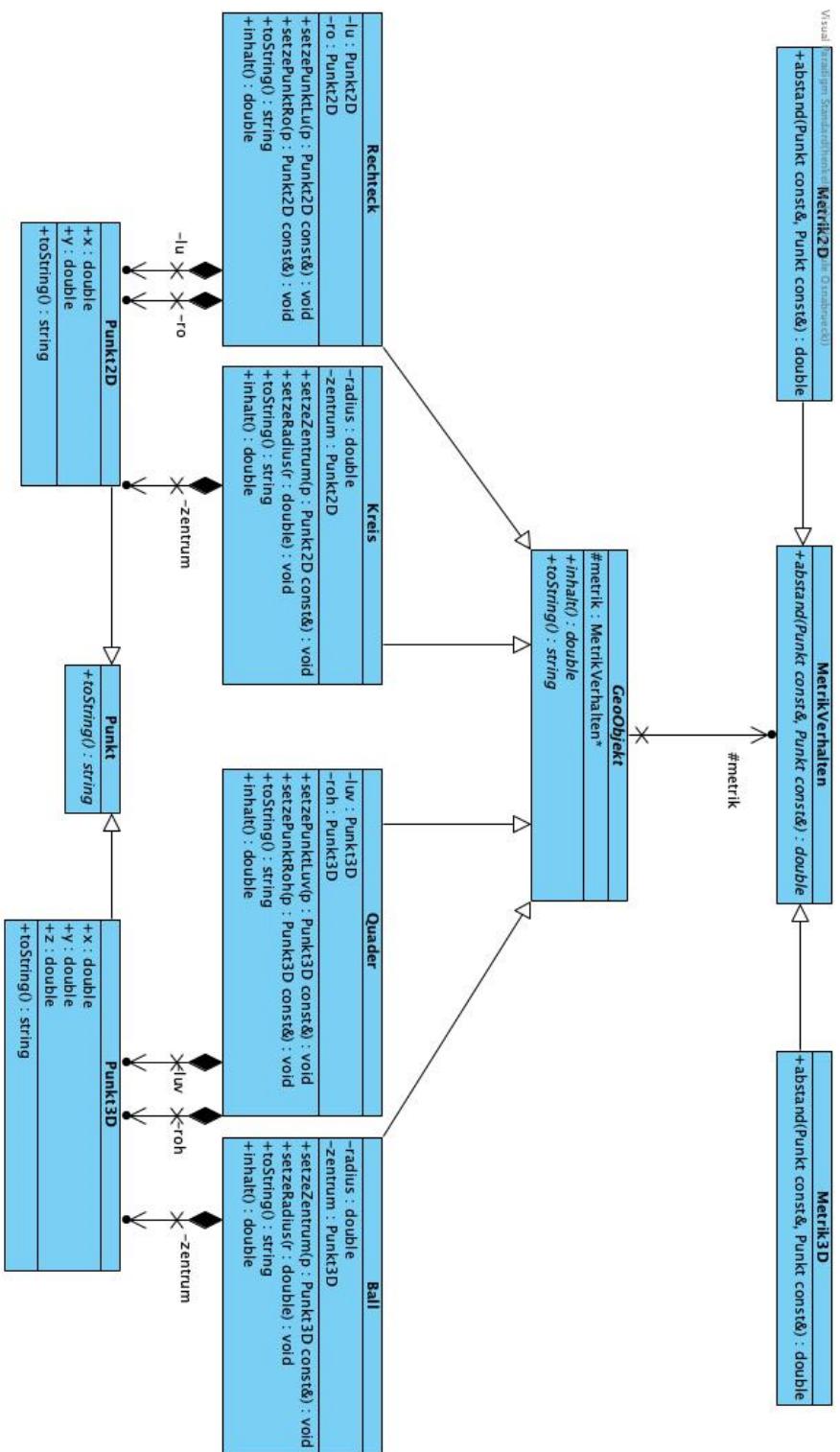
```
Ball: [(1, 1, 1), 1] 4.18879
```

```
Polymorphietest - tiefe Kopie mittels assign
```

```
Ball: [(1, 1, 1), 2] 33.5103
```

```
Ball: [(0, 0, 0), 2] 33.5103
```

```
erwartete Ausnahme geworfen
```



```
1 //  
2 // Created by Johannes on 31.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_BALL_H  
6 #define AUFGABE3_BALL_H  
7 #include "Punkt3D.h"  
8 #include "GeoObjekt.h"  
9  
10 class Ball:public GeoObjekt {  
11 public:  
12     double m_radius;  
13     Punkt3D m_zentrum;  
14  
15     /*Konstruktoren*/  
16     Ball();  
17     Ball( Punkt3D zentrum,double radius);  
18  
19     /*Setter*/  
20     void setzeZentrum(const Punkt3D& zentrum);  
21     void setzeRadius(double radius);  
22  
23     /*toString und Inhalt*/  
24     string toString();  
25     double inhalt();  
26  
27     /*clone Methode aus GeoObjekt*/  
28  
29     GeoObjekt* clone() const;  
30  
31     /*Ball& operator = (const Ball& rechts);*/  
32  
33     Ball& assign (const GeoObjekt& rechts);  
34 };  
35  
36  
37 #endif //AUFGABE3_BALL_H  
38
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_KREIS_H  
6 #define AUFGABE3_KREIS_H  
7 #include "Punkt2D.h"  
8 #include "GeoObjekt.h"  
9  
10 class Kreis:public GeoObjekt {  
11 public:  
12     double m_radius;  
13     Punkt2D m_zentrum;  
14  
15     Kreis();  
16     Kreis(Punkt2D p, double r);  
17  
18     void setzeZentrum(const Punkt2D& p);  
19     void setzeRadius(double r);  
20  
21     string toString();  
22     double inhalt();  
23  
24     /*clone Methode aus GeoObjekt*/  
25     GeoObjekt* clone()const;  
26  
27     /*operator ueberladung von = */  
28     /* Kreis& operator = (const Kreis& rechts);*/  
29  
30     /*assign Methode*/  
31     Kreis& assign(const GeoObjekt& rechts);  
32  
33  
34  
35 };  
36  
37  
38 #endif //AUFGABE3_KREIS_H  
39
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_PUNKT_H  
6 #define AUFGABE3_PUNKT_H  
7  
8 #include <iostream>  
9  
10 using namespace std;  
11  
12 class Punkt {  
13     virtual string toString()=0;  
14 };  
15  
16  
17 #endif //AUFGABE3_PUNKT_H  
18
```

```
1 //  
2 // Created by Johannes on 31.05.2021.  
3 //  
4  
5 #include "Ball.h"  
6 #include "math.h"  
7  
8 Ball::Ball() {};  
9  
10 Ball::Ball(Punkt3D zentrum, double radius) {  
11     m_radius = radius;  
12     m_zentrum = zentrum;  
13 }  
14  
15 void Ball::setzeZentrum(const Punkt3D &zentrum) {  
16     m_zentrum=zentrum;  
17 }  
18  
19 void Ball::setzeRadius(double radius) {  
20     m_radius=radius;  
21 }  
22  
23 string Ball::toString() {  
24     stringstream strom;  
25     strom<<"Kreis: [" << m_zentrum.toString() << ", "<< m_radius<<"]";  
26     return strom.str();  
27 }  
28  
29 // Formel: V = 4/3 * PI * r^3  
30 // Math.h includen!  
31 double Ball::inhalt(){  
32     return (4 * M_PI * pow(m_radius,3))/3;  
33 }  
34  
35 /*clone Methode aus GeoObjekt*/  
36 GeoObjekt * Ball::clone() const {  
37     return new Ball(*this);  
38 }  
39  
40 /*Ball& Ball::operator=(const Ball& rechts) {  
41     if(this==&rechts){  
42         return *this;  
43     }  
44     return assign(rechts);  
45 }*/  
46  
47 Ball& Ball::assign(const GeoObjekt &rechts) {  
48     const Ball* neu = dynamic_cast<const Ball*>(&rechts);  
49     if(!neu)throw OperandenPassenNicht();  
50     m_radius=neu->m_radius;  
51     m_zentrum=neu->m_zentrum;  
52     GeoObjekt::assign(rechts);  
53     return *this;  
54 }
```

```

1 //
2 //  main.cpp
3 //  GeoObjekt
4 //
5 //  Created by Oliver Henkel on 12.03.19.
6 //  Copyright © 2019 Oliver Henkel. All rights reserved.
7 //
8
9 #include "Punkt.h"
10 #include "GeoObjekt.h"
11 #include <iostream>
12 #include "Rechteck.h"
13 #include "Ball.h"
14 #include "Quader.h"
15 #include "Kreis.h"
16
17
18 using namespace std;
19
20 int main() {
21     cout << "Rechtecke" << endl;
22     cout << "-----" << endl;
23     cout << "statischer Test" << endl;
24     Rechteck rechteck( Punkt2D(0,0), Punkt2D(2,1));
25     cout << rechteck.toString() << " " << rechteck.inhalt() << endl;
26
27     cout << endl;
28     cout << "Polymorphietest - tiefe Kopie mittels clone" << endl;
29     GeoObjekt* pGeoR = &rechteck;
30     GeoObjekt* pGeoR2 = pGeoR->clone();
31     Rechteck* pRechteck2 = dynamic_cast<Rechteck*>(pGeoR2);
32     pRechteck2->setzePunktLu(Punkt2D(-1,-1));
33     cout << pGeoR->toString() << " " << pGeoR->inhalt() << endl;
34     cout << pGeoR2->toString() << " " << pGeoR2->inhalt() << endl;
35
36     cout << endl;
37     cout << "Polymorphietest - tiefe Kopie mittels assign" << endl;
38     (*pGeoR2) = (*pGeoR);
39     pRechteck2->setzePunktRo(Punkt2D(2,2));
40     cout << pGeoR->toString() << " " << pGeoR->inhalt() << endl;
41     cout << pGeoR2->toString() << " " << pGeoR2->inhalt() << endl;
42
43 //
44 // -----
45 //
46
47     cout << endl;
48     cout << "Kreise" << endl;
49     cout << "-----" << endl;
50     cout << "statischer Test" << endl;
51     Kreis kreis( Punkt2D(1,1), 2);
52     cout << kreis.toString() << " " << kreis.inhalt() << endl;
53
54     cout << endl;
55     cout << "Polymorphietest - tiefe Kopie mittels clone" << endl;
56     GeoObjekt* pGeoK = &kreis;
57     GeoObjekt* pGeoK2 = pGeoK->clone();
58     Kreis* pKreis2 = dynamic_cast<Kreis*>(pGeoK2);
59     pKreis2->setzeRadius(1);
60     cout << pGeoK->toString() << " " << pGeoK->inhalt() << endl;
61     cout << pGeoK2->toString() << " " << pGeoK2->inhalt() << endl;
62
63     cout << endl;
64     cout << "Polymorphietest - tiefe Kopie mittels assign" << endl;

```

```

65     (*pGeoK2) = (*pGeoK);
66     pKreis2->setzeZentrum(Punkt2D(0,0));
67     cout << pGeoK->toString() << " " << pGeoK->inhalt() << endl;
68     cout << pGeoK2->toString() << " " << pGeoK2->inhalt() << endl;
69
70     try {
71         (*pGeoK2) = (*pGeoR2);
72     } catch (OperandenPassenNicht&)
73     { cout << "erwartete Ausnahme geworfen" << endl; }
74
75     try {
76         (*pGeoR2) = (*pGeoK2);
77     } catch (OperandenPassenNicht&)
78     { cout << "erwartete Ausnahme geworfen" << endl; }
79
80     //
81     // -----
82     //
83
84     cout << endl;
85     cout << "Quader" << endl;
86     cout << "-----" << endl;
87     cout << "statischer Test" << endl;
88     Quader quader( Punkt3D(0,0,0), Punkt3D(2,1,2));
89     cout << quader.toString() << " " << quader.inhalt() << endl;
90
91     cout << endl;
92     cout << "Polymorphietest - tiefe Kopie mittels clone" << endl;
93     GeoObjekt* pGeoQ = &quader;
94     GeoObjekt* pGeoQ2 = pGeoQ->clone();
95     Quader* pQuader2 = dynamic_cast<Quader*>(pGeoQ2);
96     pQuader2->setzePunktLuv(Punkt3D(-1,-1,-1));
97     cout << pGeoQ->toString() << " " << pGeoQ->inhalt() << endl;
98     cout << pGeoQ2->toString() << " " << pGeoQ2->inhalt() << endl;
99
100    cout << endl;
101    cout << "Polymorphietest - tiefe Kopie mittels assign" << endl;
102    (*pGeoQ2) = (*pGeoQ);
103    pQuader2->setzePunktRoh(Punkt3D(2,2,2));
104    cout << pGeoQ->toString() << " " << pGeoQ->inhalt() << endl;
105    cout << pGeoQ2->toString() << " " << pGeoQ2->inhalt() << endl;
106
107    try {
108        (*pGeoQ2) = (*pGeoR2);
109    } catch (OperandenPassenNicht&)
110    { cout << "erwartete Ausnahme geworfen" << endl; }
111
112    //
113    // -----
114    //
115
116    cout << endl;
117    cout << "Baelle" << endl;
118    cout << "-----" << endl;
119    cout << "statischer Test" << endl;
120    Ball ball( Punkt3D(1,1,1), 2);
121    cout << ball.toString() << " " << ball.inhalt() << endl;
122
123    cout << endl;
124    cout << "Polymorphietest - tiefe Kopie mittels clone" << endl;
125    GeoObjekt* pGeoB = &ball;
126    GeoObjekt* pGeoB2 = pGeoB->clone();
127    Ball* pBall2 = dynamic_cast<Ball*>(pGeoB2);
128    pBall2->setzeRadius(1);

```

```
129     cout << pGeoB->toString() << " " << pGeoB->inhalt() << endl;
130     cout << pGeoB2->toString() << " " << pGeoB2->inhalt() << endl;
131
132     cout << endl;
133     cout << "Polymorphietest - tiefe Kopie mittels assign" << endl;
134     (*pGeoB2) = (*pGeoB);
135     pBall2->setzeZentrum(Punkt3D(0,0,0));
136     cout << pGeoB->toString() << " " << pGeoB->inhalt() << endl;
137     cout << pGeoB2->toString() << " " << pGeoB2->inhalt() << endl;
138
139     try {
140         (*pGeoB2) = (*pGeoK2);
141     } catch (OperandenPassenNicht&)
142     { cout << "erwartete Ausnahme geworfen" << endl; }
143
144
145 return 0;
146 }
147
```

```
1 //  
2 // Created by Johannes on 31.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_QUADER_H  
6 #define AUFGABE3_QUADER_H  
7 #include "Punkt3D.h"  
8 #include "GeoObjekt.h"  
9  
10 class Quader:public GeoObjekt {  
11 public:  
12     Punkt3D m_luv;  
13     Punkt3D m_roh;  
14  
15  
16     // Konstruktoren  
17     Quader();  
18     Quader(Punkt3D luv, Punkt3D roh);  
19  
20  
21  
22     // Setter  
23     void setzePunktLuv(const Punkt3D& luv);  
24  
25     void setzePunktRoh(const Punkt3D &roh);  
26  
27     // Hilfsmethoden  
28     string toString();  
29     double inhalt();  
30  
31  
32     /*clone Methode aus GeoObjekt*/  
33     GeoObjekt* clone() const;  
34  
35     /*Quader& operator=(const Quader& rechts);*/  
36  
37     Quader& assign( const GeoObjekt& rechts);  
38  
39  
40 };  
41  
42  
43  
44 #endif //AUFGABE3_QUADER_H  
45
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #include "Kreis.h"  
6 #include <sstream>  
7 #include <math.h>  
8  
9 using namespace std;  
10  
11 Kreis::Kreis(Punkt2D p, double r) {  
12     m_radius=r;  
13     m_zentrum =p;  
14 }  
15  
16 void Kreis::setzeRadius(double r) {  
17     m_radius=r;  
18 }  
19  
20 void Kreis::setzeZentrum(const Punkt2D &p) {  
21     m_zentrum=p;  
22 }  
23  
24 string Kreis::toString() {  
25     stringstream strom;  
26     strom<<"Kreis: [" << m_zentrum.toString() << ", "<< m_radius<<"]";  
27     return strom.str();  
28 }  
29  
30 double Kreis::inhalt() {  
31     return M_PI*m_radius*m_radius;  
32 }  
33  
34 /*clone Methode aus GeoObjekt*/  
35 GeoObjekt* Kreis::clone()const{  
36     return new Kreis(*this);  
37 }  
38  
39 /*Kreis& Kreis::operator=(const Kreis &rechts) {  
40     if(this == &rechts){  
41         return *this;  
42     }  
43     return assign(rechts);  
44 }*/  
45  
46 Kreis& Kreis::assign(const GeoObjekt &rechts) {  
47     const Kreis* neu = dynamic_cast<const Kreis*>(&rechts);  
48     if(!neu)throw OperandenPassenNicht();  
49     m_radius=neu->m_radius;  
50     m_zentrum=neu->m_zentrum;  
51     GeoObjekt::assign(rechts);  
52     return *this;  
53 }
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_PUNKT2D_H  
6 #define AUFGABE3_PUNKT2D_H  
7 #include "Punkt.h"  
8 #include <iostream>  
9 #include <sstream>  
10  
11 using namespace std;  
12  
13 class Punkt2D:public Punkt {  
14 public:  
15     double m_x;  
16     double m_y;  
17  
18     string toString();  
19  
20     /*Konstruktor Destruktor*/  
21  
22     Punkt2D(double x, double y);  
23     Punkt2D();  
24  
25     double getX();  
26     double getY();  
27 };  
28  
29  
30 #endif //AUFGABE3_PUNKT2D_H  
31
```

```
1 //  
2 // Created by Johannes on 31.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_PUNKT3D_H  
6 #define AUFGABE3_PUNKT3D_H  
7 #include "Punkt.h"  
8 #include <iostream>  
9 #include <sstream>  
10  
11 // Punkt3D erbt von der Klasse Punkt  
12 class Punkt3D : public Punkt{  
13 public:  
14     // Membervariablen  
15     double m_x;  
16     double m_y;  
17     double m_z;  
18  
19  
20     // Konstruktoren  
21     Punkt3D();  
22     Punkt3D(double x, double y, double z);  
23  
24  
25     // Getter  
26     double getX();  
27     double getY();  
28     double getZ();  
29  
30  
31     // Hilfsmethoden  
32     string toString();  
33  
34  
35 };  
36  
37  
38 #endif //AUFGABE3_PUNKT3D_H  
39
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #ifndef AUFGABE3_METRIK2D_H  
6 #define AUFGABE3_METRIK2D_H  
7  
8 #include <iostream>  
9 #include "MetrikVerhalten.h"  
10 #include "Punkt2D.h"  
11  
12 using namespace std;  
13 class Metrik2D:public MetrikVerhalten {  
14  
15 public:  
16     double abstand(const Punkt2D& p1,const Punkt2D& p2);  
17 };  
18  
19  
20 #endif //AUFGABE3_METRIK2D_H  
21
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #ifndef AUFGABE3_METRIK3D_H  
6 #define AUFGABE3_METRIK3D_H  
7  
8 #include <iostream>  
9 #include "Punkt3D.h"  
10  
11 using namespace std;  
12  
13 class Metrik3D {  
14 public:  
15     double abstand(const Punkt3D& p1, const Punkt3D& p2);  
16 };  
17  
18  
19 #endif //AUFGABE3_METRIK3D_H  
20
```

```

1 //
2 // Created by Johannes on 31.05.2021.
3 //
4
5 #include "Quader.h"
6 #include <sstream>
7 #include <iostream>
8
9 // Default Konstruktor
10 Quader::Quader() {};
11
12 // Ueberladener Konstruktor
13 Quader::Quader(Punkt3D luv, Punkt3D roh) {
14
15     m_luv=luv;
16     m_roh=roh;
17 }
18
19
20
21 // Setter
22 void Quader::setzePunktLuv(const Punkt3D &luv) {
23     m_luv=luv;
24 }
25
26 void Quader::setzePunktRoh(const Punkt3D &roh){
27     m_roh = roh;
28 }
29
30 string Quader::toString() {
31     stringstream strom;
32     strom << "[" << m_luv.toString() <<, "<< m_roh.toString()<<]";
33     return strom.str();
34 }
35
36 double Quader::inhalt() {
37
38     double yTotal;
39     double xTotal;
40     double zTotal;
41
42     yTotal= m_roh.getY()-m_luv.getY();
43     xTotal= m_roh.getX()-m_luv.getX();
44     zTotal= m_roh.getZ()-m_luv.getZ();
45     return xTotal*yTotal*zTotal;
46 }
47 }
48 /*clone Methode aus GeoObjekt*/
49 GeoObjekt* Quader::clone() const{
50     return new Quader(*this);
51 }
52
53 /*Quader& Quader::operator=(const Quader &rechts) {
54     if(this==&rechts){
55         return *this;
56     }
57
58     return assign(rechts);
59 }*/
60
61 Quader& Quader::assign(const GeoObjekt& rechts){
62     const Quader* neu = dynamic_cast<const Quader*>(&rechts);
63     if(!neu)throw OperandenPassenNicht();
64     m_luv=neu->m_luv;

```

```
65     m_roh=neu->m_roh;
66     GeoObjekt::assign(rechts);
67     return *this;
68 }
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #ifndef AUFGABE3_RECHTECK_H  
6 #define AUFGABE3_RECHTECK_H  
7  
8 #include "Punkt2D.h"  
9 #include "GeoObjekt.h"  
10  
11 class Rechteck:public GeoObjekt {  
12 public:  
13     /*muss private*/  
14     Punkt2D m_lu;  
15     Punkt2D m_ro;  
16  
17     Rechteck(Punkt2D lu,Punkt2D ro);  
18  
19     void setzePunktLu(const Punkt2D& p);  
20     void setzePunktRo(const Punkt2D& p);  
21  
22     string toString();  
23  
24     double inhalt();  
25  
26     /*clone Methode GEERBT von GeoObjekt*/  
27     GeoObjekt* clone() const;  
28  
29     /*Rechteck& operator = (const Rechteck& rechts);*/  
30  
31     Rechteck& assign (const GeoObjekt& rechts);  
32 };  
33  
34  
35  
36 #endif //AUFGABE3_RECHTECK_H  
37
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #ifndef AUFGABE3_GEOOBJEKT_H  
6 #define AUFGABE3_GEOOBJEKT_H  
7  
8 #include "MetrikVerhalten.h"  
9 #include "iostream"  
10 #include "OperandenPassenNicht.cpp"  
11 using namespace std;  
12  
13 class GeoObjekt {  
14 protected:  
15     MetrikVerhalten *metrik;  
16  
17 public:  
18     virtual double inhalt(){};  
19     virtual string toString(){};  
20  
21  
22     /*clone Methode*/  
23     virtual GeoObjekt* clone() const = 0;  
24  
25     /*Operator = ueberladen*/  
26     GeoObjekt& operator = (const GeoObjekt& rechts){  
27         if(this == &rechts){  
28             return *this;  
29         }  
30         return assign(rechts);  
31     };  
32  
33     /*assign Methode*/  
34     virtual GeoObjekt& assign(const GeoObjekt& rechts){  
35         const GeoObjekt* neu = dynamic_cast<const GeoObjekt*>(&rechts);  
36         if(!neu) throw OperandenPassenNicht();  
37  
38         metrik = neu->metrik;  
39         return *this;  
40     }  
41 };  
42  
43  
44 #endif //AUFGABE3_GEOOBJEKT_H  
45
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #include "Punkt2D.h"  
6  
7 string Punkt2D::toString() {  
8     stringstream strom;  
9     strom << "(" << m_x << ", " << m_y << ")";  
10    return strom.str();  
11 };  
12  
13 Punkt2D::Punkt2D(double x, double y) {  
14     m_x = x;  
15     m_y = y;  
16 };  
17  
18 Punkt2D::Punkt2D() {};  
19  
20 double Punkt2D::getX() {  
21     return m_x;  
22 }  
23  
24 double Punkt2D::getY(){  
25     return m_y;  
26 }  
27  
28  
29
```

```
1 //  
2 // Created by Johannes on 31.05.2021.  
3 //  
4  
5 #include "Punkt3D.h"  
6 #include <iostream>  
7  
8 // Default Konstruktor  
9 Punkt3D::Punkt3D() {};  
10  
11 // Ueberladener Konstruktor ohne Initialisierungsliste  
12 Punkt3D::Punkt3D(double x, double y, double z) {  
13     m_x = x;  
14     m_y = y;  
15     m_z = z;  
16 }  
17  
18 string Punkt3D::toString() {  
19     stringstream strom;  
20     strom << "(" << m_x << ", " << m_y << ", " << m_z << ")";  
21     return strom.str();  
22 }  
23  
24 double Punkt3D::getX() {  
25     return m_x;  
26 }  
27  
28 double Punkt3D::getY(){  
29     return m_y;  
30 }  
31  
32 double Punkt3D::getZ() {  
33     return m_z;  
34 }
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #include "Metrik2D.h"  
6 #include "iostream"  
7 #include "Punkt2D.h"  
8 #include "math.h"  
9  
10 double Metrik2D::abstand(const Punkt2D& p1, const Punkt2D& p2) {  
11     double xTotal = p1.m_x - p2.m_x;  
12     double yTotal = p1.m_y - p2.m_y;  
13  
14     return sqrt(xTotal*xTotal + yTotal*yTotal);  
15 }
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #include "Metrik3D.h"  
6 #include <iostream>  
7 #include <math.h>  
8  
9 double Metrik3D::abstand(const Punkt3D &p1, const Punkt3D &p2) {  
10    double xTotal = p1.m_x - p2.m_x;  
11    double yTotal = p1.m_y - p2.m_y;  
12    double zTotal = p1.m_z - p2.m_z;  
13  
14    return sqrt(xTotal*xTotal + yTotal*yTotal + zTotal*zTotal);  
15 }
```

```
1 //  
2 // Created by Johannes on 30.05.2021.  
3 //  
4  
5 #include "Rechteck.h"  
6 #include <sstream>  
7  
8 using namespace std;  
9  
10 Rechteck::Rechteck(Punkt2D lu, Punkt2D ro) {  
11     m_lu = lu;  
12     m_ro = ro;  
13 }  
14  
15 void Rechteck::setzePunktLu(const Punkt2D &p) {  
16     m_lu = p;  
17 }  
18  
19 void Rechteck::setzePunktRo(const Punkt2D &p) {  
20     m_ro = p;  
21 }  
22  
23 string Rechteck::toString() {  
24     stringstream strom;  
25     strom << "[" << m_lu.toString() <<, "<< m_ro.toString()<<"]";  
26     return strom.str();  
27 }  
28  
29 double Rechteck::inhalt(){  
30  
31     /*Braucht noch MOD für dummies*/  
32  
33     double yTotal;  
34     double xTotal;  
35  
36     yTotal= m_ro.getY()-m_lu.getY();  
37     xTotal= m_ro.getX()-m_lu.getX();  
38  
39     return xTotal*yTotal;  
40 }  
41  
42 GeoObjekt * Rechteck::clone() const {  
43     return new Rechteck(*this);  
44 }  
45  
46 /*Rechteck& Rechteck::operator=(const Rechteck &rechts) {  
47     if(this == &rechts){  
48         return *this;  
49     }  
50     return assign(rechts);  
51 }*/  
52  
53 Rechteck& Rechteck::assign(const GeoObjekt &rechts) {  
54     const Rechteck* neu = dynamic_cast<const Rechteck*>(&rechts);  
55     if(!neu)throw OperandenPassenNicht();  
56     m_lu = neu->m_lu;  
57     m_ro = neu->m_ro;  
58  
59     /*assign in Elternklasse aufrufen*/  
60     GeoObjekt::assign(rechts);  
61     return *this;  
62 }  
63  
64
```

65

66

```
1 # cmake_minimum_required(VERSION <specify CMake version here>)
2 project(Aufgabe3)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(Aufgabe3 main.cpp Punkt.h Punkt2D.cpp Punkt2D.h Rechteck.cpp Rechteck.h Kreis.cpp
Kreis.h Punkt3D.cpp Punkt3D.h Quader.cpp Quader.h Ball.cpp Ball.h GeoObjekt.h MetrikVerhalten.h
Metrik2D.cpp Metrik2D.h Metrik3D.cpp Metrik3D.h OperandenPassenNicht.cpp)
```

```
1 //  
2 // Created by Johannes on 05.06.2021.  
3 //  
4  
5 #ifndef AUFGABE3_METRIKVERHALTEN_H  
6 #define AUFGABE3_METRIKVERHALTEN_H  
7  
8 #include <iostream>  
9 #include "Punkt.h"  
10  
11 using namespace std;  
12  
13 class MetrikVerhalten {  
14 public:  
15     virtual double abstand(const Punkt& p1, const Punkt& p2)=0;  
16 };  
17  
18  
19 #endif //AUFGABE3_METRIKVERHALTEN_H  
20
```

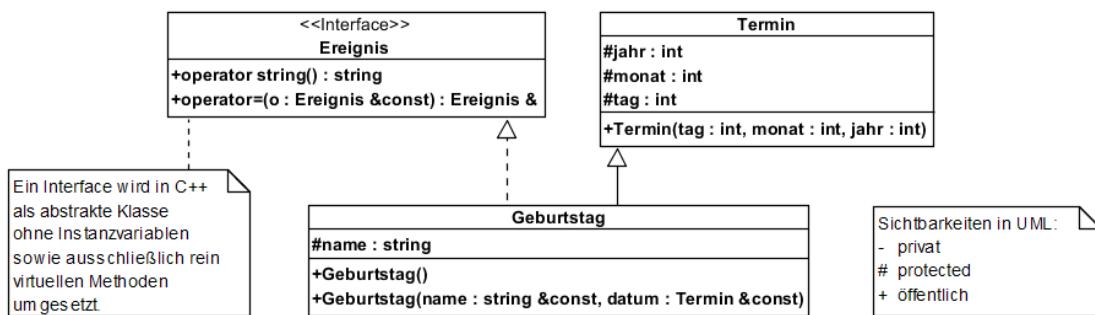
```
1 //
2 // Created by Johannes on 08.06.2021.
3 //
4 #include <exception>
5 #include "iostream"
6
7 using namespace std;
8
9 class OperandenPassenNicht:public exception{
10     virtual const char* what() const throw(){
11         return "Ausnahme bruder";
12     }
13 };
14
```

Aufgabe 4: Vererbung (20 Punkte)

Ein Organizer-Programm soll Ereignisse verwalten. Konkrete Ereignisse können z.B. Geburtstage, Termine, Aufgaben, usw. sein.

Die verschiedenen Ereignisse sollen innerhalb des Programms durch Objekte repräsentiert werden, die in gemeinsamen Containern gespeichert werden. Dazu werden alle konkreten Ereignisse von einer Basisklasse `Ereignis` abgeleitet.

Als konkrete Klasse wird in dieser Aufgabe beispielhaft die Klasse `Geburtstag` betrachtet. Nutzen Sie wann immer möglich und sinnvoll das Schlüsselwort `const`.



Aufgaben:

- 4.1 definieren Sie das Interface `Ereignis` mit folgenden Methoden:
 - öffentlicher Zuweisungsoperator,
 - öffentlicher Konvertierungsoperator nach `string`,
 - überlegen Sie, ob diese abstrakte Klasse ohne Instanzvariablen und ausschließlich rein virtuellen Methoden einen Destruktor benötigt. Falls ja, definieren Sie diesen.
- 4.2 definieren Sie die Klasse `Klasse Termin` mit den `int`-Variablen `jahr`, `monat` und `tag`, die mit der Sichtbarkeit `protected` gekapselt werden. Die Klasse besitzt lediglich einen öffentlichen Konstruktor, dem die Werte für `jahr`, `monat` und `tag` übergeben werden.
- 4.3 definieren Sie die konkrete Klasse `Geburtstag`, die von den Klassen `Ereignis` und `Termin` abgeleitet wird.

Sie enthält die Instanzvariable:

- `name` vom Typ `string` mit der Sichtbarkeit `protected`, und folgende Inline zu definierenden öffentlichen Methoden:
- einen Default-Konstruktor der die Instanzvariablen mit Default-Werten initialisiert (überlegen Sie sich sinnvolle Default-Werte),
- einen überladenen Konstruktor, der einen Namen und ein Termin - jeweils als konstante Referenz - als Parameter erwartet,
- den von der Basisklasse `Ereignis` geerbten Zuweisungsoperator:
`Geburtstag& operator=(const Ereignis& o)`

Es soll geprüft werden, ob ein Downcast des übergebenen Parameters vom Typ `const Ereignis&` möglich ist. Wenn nicht, wird vom `dynamic_cast` Operator eine `bad_cast` Exception geworfen. Bei Übereinstimmung ist der Standard-Zuweisungsoperator der Klasse `Geburtstag` aufzurufen (der automatisch vom Compiler erzeugt wird). Denken Sie daran, dass ein Objekt vom Typ `Geburtstag&` zurückzugeben ist.

- den von der Basisklasse `Ereignis` geerbten Konvertierungsoperator nach `string`. Der Konvertierungsoperator erzeugt aus der Variablen `name` und den geerbten Variablen des `Termin`-Objektes ein neues `string`-Objekt und gibt dieses zurück.

Beispiel: Mit den Werten ("Angela Merkel", 1954, 7, 17) für Name, Jahr, Monat, Tag soll ein `string`-Objekt mit dem Inhalt "Angela Merkel 17.7.1954" zurückgegeben werden.

Hinweis: Zur Konvertierung eines `int`-Wertes in ein `string`-Objekt soll die Klasse `stringstream` verwendet werden.

Übungsklausur Fortgeschrittene Programmierung

4.4 Beantworten Sie folgende Fragen:

- Erläutern Sie die Begriffe „statische Polymorphie“ und „dynamische Polymorphie“.
- Welche Auswirkungen hat es, wenn die Klasse Geburtstag den Zuweisungsoperator der Klasse Ereignis nicht definiert?
- Erläutern Sie, ob der Zuweisungsoperator in C++ polymorph verwendet werden kann. Denken Sie daran, dass der Compiler folgenden Zuweisungsoperator automatisch erzeugt, wenn dieser nicht explizit definiert wird:

```
Geburtstag& operator=(const Geburtstag& o) { ... }
```
- Die Anwendung des sizeof-Operators für die Klasse Geburtstag liefert den Wert 20 (Bytes). Erklären Sie, wie sich dieser Wert ergibt.

```
1 #include "Geburtstag.cpp"
2
3 int main() {
4
5     Termin berkanBday(8,10,1997);
6     Geburtstag berkan("Berkan", berkanBday);
7
8     Geburtstag defau;
9
10    cout << berkan();
11
12    cout << sizeof(berkan);
13
14    return 0;
15 }
16
17 /*
18 * Statische Polymorphie
19     Statische Polymorphie zeigt sich in Form des Überladens von Methoden, bei dem sich mehrere
20     Methodensignaturen nicht
21     in ihrem Bezeichner, jedoch in deren Parameterliste unterscheiden.
22     Die Wahl der auszuführenden Methode geschieht somit anhand von Anzahl und Typ der
23     Methodenparameter.
24 *
25 *Dynamische Polymorphie
26     Bei der dynamischen Polymorphie werden in mehreren Klassen einer Vererbungslinie gleiche, d.h.
27     mit identischen Signaturen versehene Methoden unterschiedlich implementiert,
28     sodass zur Laufzeit entschieden werden muss, welche der Methoden ausgeführt wird. Dies ist dann
29     der Fall, wenn die Kindklassen die Methoden der Elternklasse überschreiben (nicht überladen!).
30     Es muss dann zur Laufzeit entscheiden werden, ob die Methode der Eltern- oder diejenige der
31     Kindklasse ausgeführt wird.
32 **/
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4  
5 class Termin{  
6 protected:  
7     int m_jahr;  
8     int m_monat;  
9     int m_tag;  
10 public:  
11     Termin(int tag, int monat, int jahr){  
12         m_tag = tag;  
13         m_monat = monat;  
14         m_jahr = jahr;  
15     }  
16  
17     Termin(){};  
18  
19     int getJahr() const{  
20         return m_jahr;  
21     }  
22  
23     int getMonat() const{  
24         return m_monat;  
25     }  
26  
27     int getTag() const{  
28         return m_tag;  
29     }  
30 };
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4 #include <iostream>  
5  
6 using namespace std;  
7  
8 class Ereignis{  
9 public:  
10     virtual string operator()() = 0;  
11     Ereignis& operator=(const Ereignis& ereignis);  
12  
13  
14     virtual Ereignis& assign(const Ereignis& rhs){  
15         return *this;  
16     };  
17  
18 };
```

```
1 cmake_minimum_required(VERSION 3.19)
2 project(Klausurvorbereitung3Vererbung)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(Klausurvorbereitung3Vererbung main.cpp Ereignis.cpp Termin.cpp Geburtstag.cpp)
```

```
1 //  
2 // Created by boery on 18.07.2021.  
3 //  
4 #include "Termin.cpp"  
5 #include "Ereignis.cpp"  
6 #include "sstream"  
7  
8 class Geburtstag: public Termin, public Ereignis{  
9 protected:  
10     string m_name;  
11  
12 public:  
13     Geburtstag();  
14     Geburtstag(const string &name, const Termin &datum);  
15  
16  
17     Geburtstag& operator=(const Geburtstag& rhs){  
18         if(this == &rhs) return *this;  
19         return assign(rhs);  
20     }  
21  
22     Geburtstag& assign(const Ereignis& rhs){  
23         try{  
24             const Geburtstag* pG = dynamic_cast<const Geburtstag*>(&rhs);  
25             Ereignis::assign(rhs);  
26  
27             m_name = pG->m_name;  
28             m_tag = pG->m_tag;  
29             m_monat = pG->m_monat;  
30             m_jahr = pG->m_jahr;  
31             return *this;  
32         } catch(bad_cast&){  
33             cout << "Bad Cast du Hund" << endl;  
34         }  
35     }  
36 }  
37  
38  
39     string operator()(){  
40         stringstream strom;  
41         strom << m_name << " " << m_jahr << " " << m_monat << " " << m_tag << endl;  
42         return strom.str();  
43     }  
44  
45 };  
46  
47  
48 inline Geburtstag::Geburtstag(){  
49     m_tag=1;  
50     m_monat=1;  
51     m_jahr=1;  
52     m_name="Name eintragen.";  
53 }  
54  
55 inline Geburtstag::Geburtstag(const string &name, const Termin &datum) {  
56     m_name = name;  
57     m_tag = datum.getTag();  
58     m_monat = datum.getMonat();  
59     m_jahr = datum.getJahr();  
60 }  
61  
62
```

Inhaltsverzeichnis

1. Von Java zu C++

- i. Hello World
- ii. Programmieren ohne Klassen
- iii. Daten
 - 1. Datentypen
 - 2. Ströme
 - 3. Referenztypen
 - 4. Speicherklassen
 - 5. Ausnahmebehandlung
- iv. Funktionen
 - 1. Werte- und Referenzsemantik
 - 2. Operatorüberladung
 - 3. Überladungsauflösung
- v. Klassen
 - 1. Klassen in C++
 - 2. Operatormethoden
 - 3. benutzerdefinierte Typumwandlung
 - 4. Funktionsobjekte und Prädikate
- vi. sequentielle Container

2. Vererbung

- i. Objekte in Beziehung
- ii. Vererbung
 - 1. Umsetzung in C++
 - 2. Wertsemantik: Auflösen von Namenskonflikten
 - 3. Zugriffsrechte
 - 4. Standardmethoden
- iii. Polymorphismus
 - 1. Umsetzung in C++
 - 2. Auswahlregeln bei virtuellen Methoden
 - 3. Standardmethoden und Polymorphie
- iv. Abstrakte Klassen und Interfaces
 - 1. Abstrakte Klassen
 - 2. Interfaces
 - 3. Mehrfachvererbung

Inhaltsverzeichnis

1. Zeiger

i. Zeiger und Felder

1. Zeigervariablen
2. Felder
3. Zeigerarithmetik
4. Vergleich: Zeiger vs Felder & Referenzen
5. Anwendung: Online-Shop

ii. dynamische Speicherverwaltung

1. Speicheranforderung und Speicherfreigabe
2. flache und tiefe Kopie
3. Anwendung Favoritenliste

iii. Mehrdimensionale Felder

Von Java zu C++ - Hello World

Java

```
public class HelloWorld {  
    // Jedes Java-Programm besitzt eine Klasse  
    static public void main(String[] args) {  
        System.out.println("HelloWorld!");  
    }  
}
```

C++

```
#include <iostream> // Bibliothek fuer Ein-/Ausgabefunktionen  
using namespace std; // Namensraum fuer C++ Bibliotheksfunktionen  
  
int main() {  
    cout << "HelloWorld!" << endl;  
    return 0;  
}
```

Von Java zu C++ - Programmieren ohne Klassen

- In einem C++ Programm gibt es genau eine Funktion `int main()`, die bei Programmstart aufgerufen wird
- Methoden, die keiner Klasse zugeordnet sind, heißen Funktionen.
- Beim Aufruf einer Funktion muss deren **Signatur** (Funktionsname+Parameterliste) bekannt sein. Dies erfolgt entweder durch Implementation der Funktion vor dem Aufruf oder durch die Funktionsdeklaration am Anfang

```
int addiere(int, int); // Funktionsdeklaration  
                      // (kann entfallen, wenn die Funktion  
                      // direkt hier implementiert wird)  
  
int main() {  
    int a {4}, b {5}, c;  
    c = addiere(a,b);  
    // Die Werte von a,b werden als Kopie  
    // an die Funktion uebergeben  
    return 0;  
    // Rueckgabewert bei normaler  
    // Programmausfuehrung  
}  
  
int addiere(int a, int b) { // Funktionsimplementation  
    return a+b;  
}
```

Von Java zu C++ - Daten - Datentypen

Bez. Java	Größe	Min	Max	Bez. C++	Größe	Min	Max
<code>boolean</code>	1 bit	–	–	<code>bool</code>	8 Bit	<code>0 (false)</code>	<code>1 (true)</code>
<code>char</code>	16 Bit	Unicode 0	Unicode $2^{16} + 1$	<code>char</code>	8 Bit	–128	+127
–	–	–	–	<code>unsigned char</code>	8 Bit	0	255
<code>byte</code>	8 Bit	–128	127	–	–	–	–
<code>short</code>	16 Bit	-2^{15}	$2^{15} - 1$	<code>short</code>	16 Bit	-2^{15}	$2^{15} - 1$
–	–	–	–	<code>unsigned short</code>	16 Bit	0	$2^{16} - 1$
<code>int</code>	32 Bit	-2^{31}	$2^{31} - 1$	<code>int</code>	<code>32 Bit</code>	-2^{31}	$2^{31} - 1$
–	–	–	–	<code>unsigned int</code>	<code>32</code>	0	$2^{32} - 1$
<code>long</code>	64 Bit	-2^{63}	$2^{63} - 1$	<code>long</code>	<code>64 Bit</code>	-2^{63}	$2^{63} - 1$
–	–	–	–	<code>unsigned long</code>	<code>64 Bit</code>	0	$2^{64} - 1$
<code>float</code>	32 Bit	$1,4e - 45$	$34e + 38$	<code>float</code>	<code>32 Bit</code>	$1,1e - 38$	$3,4e + 38$
<code>double</code>	64 Bit	$4,9e - 324$	$1,7e + 308$	<code>double</code>	<code>64 Bit</code>	$2,2e - 308$	$1,7e + 308$
–	–	–	–	<code>long double</code>	<code>128 Bit</code>	$3,3e - 4932$	$1,1e + 4932$

Größen sind abhängig von der Rechnerarchitektur

Von Java zu C++ - Daten - Datentypen - Operatorprioritäten

Prio	Operator	Ass.	Bemerkung
0	<code>::</code>		Bereichsauflösung
1	<code>++ -- () [] -> .</code> <code>x.cast<Typ>() x=static, const, reinterpret, dynamic</code>	I	postfix, unär
2	<code>++ -- ! ~ + - * &</code> <code>sizeof() new delete</code>	r	präfix, unär
3	<code>.* ->*</code>	I	Elementselektion
4	<code>* / \%</code>	I	binär
5	<code>+ -</code>	I	arithmetisch
6	<code><< >></code>	I	shift
7	<code>< <= > >=</code>	I	
8	<code>== !=</code>	I	relational
9	<code>&</code>	I	
10	<code>\^</code>	I	bit
11	<code> </code>	I	
12	<code>\&\&</code>	I	
13	<code> </code>	I	logisch
14	<code>?:</code>	r	ternär
15	<code>= += -= *= /= %=</code> <code>&= ^= = <<= >>=</code>	r	Zuweisung
16	<code>throw</code>		Ausnahme werfen
17	<code>,</code>	I	Sequenz

Von Java zu C++ - Daten - Datentypen - Strings

Java

```
public class Klasse {
    public static void main(String [] args) {
        String str = "String";
        // inhaltlicher Vergleich + Elementzugriff; Ausgabe: r
        if(str.equals("String")) System.out.println(str.charAt(2));
        // Konkatenation + Zuweisung
        str+= "in Java";
        System.out.println(str.length() + " - " + str);
    } // Ausgabe: 14 String in Java
}
```

C++

```
#include <iostream>
#include <string>           // muss eingebunden werden
using namespace std;
int main(){
    string str {"String"}; // Initialisierung
    // inhaltlicher Vergleich + Elementzugriff; Ausgabe: r
    if(str == "String") cout << str[2] << endl;
    // Konkatenation + Zuweisung
    str += "in C++";
    cout << str.size() << " - " << str << endl;
    return 0;
} // Ausgabe: 13 String in C++
```

Operationen

Deklaration und Initialisierung	string s {"Hallo"};
Verkettung	string t="du da";
Vergleichsoperatoren	==, !=, <, >, <=, >=

Manipulationen

Leeren	s.clear()
t ab Position pos einfügen	s.insert(pos,t)
anz Zeichen ab pos löschen	s.erase(pos,anz)
anz Zeichen ab pos durch t ersetzen	s.replace(pos, anz, t)

Abfragen

Länge	s.length()
Kürzen/Verlängern (auffüllen mit 0)	s.resize(len)
anz Zeichen ab pos zurückgeben	s.substr(pos,anz)
Position von t in s	s.find(t, [start_pos])
— — Suche von rechts nach links	s.rfind(t, [start_pos])

Von Java zu C++ - Daten - Datentypen - Vektoren

Vektoren sind **Typschablonen**: Der Elementtyp wird in spitzen Klammern angegeben und gehört mit zum Typnamen! „primitive Elementtypen sind erlaubt!“

```

vector<double> vec {1.5, -2.3}; // Vektor mit 2 double-Werten initialisiert
vector<double> vec_kopie(vec); // Inhaltskopie von vec
vector<int> quadrat(10); // Vektor mit Platz fuer 10 int-Werte

// Elemente anfuegen/entfernen
vec.push_back(3.7); // 1.5, -2.3, 3.7
vec.push_back(5.4); // 1.5, -2.3, 3.7, 5.4
vec.pop_back(); // 1.5, -2.3, 3.7

// Elementzugriff
cout << vec.front() << endl; // 1.5
cout << vec.back() << endl; // 3.7

// wahlfreier Zugriff
for (int i=0; i<quadrat.size(); i++) quadrat[i] = i*i;
// 0 1 4 9 16 25 36 49 64 81
for (int i=10; i<15; i++) quadrat.push_back(i*i);
// 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196

// Globalzugriff
quadrat.resize(10); // schneidet Vektor ab
// 0 1 4 9 16 25 36 49 64 81
quadrat.resize(15); // verlaengert Vektor
// 0 1 4 9 16 25 36 49 64 81 0 0 0 0 0

vec_kopie = vec; vec_kopie[2]=3; // Zuweisung kopiert Inhalt!
cout << boolalpha << (vec_kopie==vec) << endl;
// lexikographischer Vergleich: true

```

logisches Speicherabbild

front			back
-------	--	--	------

Konstruktion (Elementtyp=T)

T steht für einen existierenden Datentypnamen.

leerer Vektor

vector<T> v;

Vektor mit n Elementen

vector<T> v(n);

Vektor mit n Elementen und Vorgabewert x

vector<T> v(n, x);

Vector v anlegen und initialisieren

vector<T> v {t1, t2, t3};

Kopie von v anlegen

vector<T> w(v);

Anfügen/Entfernen

x am Ende anfuegen

v.push_back(x);

Element am Ende entfernen

v.pop_back();

Elementzugriff

Element am Anfang

v.front();

Element am Ende

v.back();

i-tes Element

v[i];

i-tes Element mit Bereichsüberprüfung

v.at(i);

Globalzugriff

Anzahl Elemente

v.size();

Vektor abschneiden/auffüllen

v.resize(n);

Elemente löschen

v.clear();

Vektor leer?

v.empty();

Zuweisung

=

lexikographische Vergleiche

==, !=, <, <=, >, >=

Von Java zu C++ - Daten - Datentypen - Strukturen

```
struct Person { // Person ist jetzt ein neuer Typname
    string name;
    int alter;
    string adresse;
};           //! abschliessendes Semikolon nicht vergessen
Person person; // Vereinbarung einer struct-Variablen

// Komponentenzugriff
person.name = "Max_Muster";
person.alter = 30;
person.adresse = "Hauptstrasse_15";

// Zuweisung kopiert Inhalte!
Person neu {"Hans_Wurst", 25, "Bierstrasse_13"};
neu=person; // ueberschreibt die Komponenten von neu
            // durch die Komponenten von person
```

Von Java zu C++ - Daten - Datentypen - Enums

```
// Neuer Aufzaehlungstyp Ampel
enum Ampel { rot, gelb, gruen };

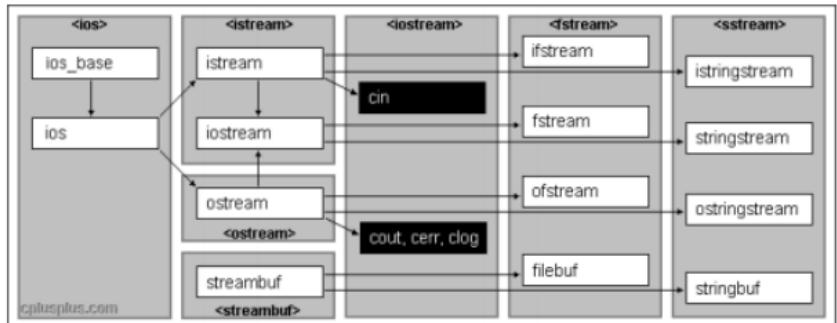
// eine Ampelvariable
Ampel ampel_ecke_lessingstrasse=rot;

//
if (ampel_ecke_lessingstrasse==rot) { /* warten */ }

// Ampel weiterschalten
ampel_ecke_lessingstrasse = gruen;
int interneNr {ampel_ecke_lessingstrasse};
cout << interneNr << endl; // 2
cout << gruen << endl; // 2
```

Von Java zu C++ - Daten - Ströme - Stromkonzept

In C++ ist die Ein-/Ausgabe über sogenannte *streams* realisiert. Die *iostream*-Bibliothek stellt Klassen, sowie Eingabe-/Ausgabemethoden bzw. -funktionen für jeden eingebauten Datentyp zur Verfügung. für benutzerdefinierte Datentypen ist es möglich, eigene Ein-/Ausgabefunktionen hinzuzufügen.



- Die Standardausgabe erfolgt über das Stream-Objekt `cout`, das eine formatierte Ausgabe der C++ Datentypen auf den Bildschirm ermöglicht.

`cout` ist ein globales Objekt der Klasse `ostream` und wird per `#include <iostream>` in eigenen Programme eingebunden.

- In Verbindung mit `cout` ist << ein Operator, der Werte von Variablen oder Konstanten ausgibt und eine Referenz auf den Strom zurückliefert.

Auf diese Weise sind in der Abarbeitung von links nach rechts Verketten möglich: `cout << "Werte..." << i << i << endl;`

- Mithilfe von Manipulatoren (`#include<iomanip>`) kann die Ausgabe formatiert werden.

Verwendet man anstelle von `cout` den Fehlerstrom `cerr` wird die Ausgabe in die Standard-Fehlerausgabe (`std::err`) umgeleitet (typischerweise ebenfalls die Konsole)

`fstream`-Objekte (`#include<iostream>`) ermöglichen die Manipulation von Dateien so, als wären es Ströme.

```

//Eingabestrom anlegen und mit Datei "eingabe.txt" verbinden
ifstream ein("eingabe.txt");
//Ausgabestrom anlegen und mit Datei "ausgabe.txt" verbinden
ofstream aus("ausgabe.txt");
if(!ein || !aus) //Pruefen ob die Dateien geoeffnet wurden
    cout << "Fehler-beim-Offnen-der-Dateien" << endl; exit(1);
}

//Variante - Dieselbe Datei fuer Ein- und Ausgabe:
//fstream strom("datei.txt");

// Datei kopieren und dabei Werte verdreifachen
double wert;
while(!ein >> wert) //Einlesen bis End of File
    aus << "drei*fertig" << setiosflags(ios::fixed) << 3*wert << endl;
ein.close(); aus.close();

```

```

-2.3 5.7
      Wert= -6.9
      Wert= 17.1
      Wert= 12.3
      Wert= 27
      Wert= 30

```

10.0

streamobjekte (`#include<sstream>`) ermöglichen die Manipulation von string-Objekten so, als wären es Ströme. Damit lassen sich komfortabel Datentypen konvertieren

```

string str = "12.345-45";
float doubleVar = 0; int intVar = 0;

//Konvertierung eines strings nach double und int
stringstream strObj(str); // Stromobjekt initialisieren
if(strObj >> doubleVar >> intVar) // Pruefen der Zuweisung
else { // hat geklappt - Ausgabe: 12.345 - 45
    cout << doubleVar << "-" << intVar << endl;
}

//Konvertierung von double und int in einen string
stringstream neuerStrom;
neuerStrom << doubleVar << ";" << intVar; // Konvertierung
cout << neuerStrom.str() << endl; // Umwandlung in string - Ausgabe: 12.345 - 45

```

<code>strom.good()</code>	letzte Operation war fehlerfrei
<code>strom.fail()</code>	letzte Operation war fehlerhaft wie <code>fail()</code> , aber Strom jetzt kaputt
<code>strom.bad()</code>	wie <code>fail()</code> , aber Strom jetzt kaputt
<code>strom.eof()</code>	Dateiende ist erreicht worden
<code>strom.clear()</code>	macht den Strom wieder aufnahmefähig

```

int zahl; // eingesetzende Zahl
char ungültigeZeichen {'-'}; // Zeichen, die keine Ziffern sind
char falschesZeichen {'.'}; // nimmt die Falscheingaben auf

cout << "Bitte eine Zahl, die Zahlen enthaelt, eingeben" << endl;
do { // Eingabe wird Zeichen fuer Zeichen durchgegangen
    // Pruefung ob es sich um eine normale Zeichen eingelesen
    if (cin >> zahl) { // Abfrage des Eingabestrom-Zustands
        cin.clear(); // macht den Strom wieder aufnahmefähig
        cin.ignore(); // ignoriert falschesZeichen; // alle vorherigen eingelegten Zeichen
        ungültigeZeichen; // ungültige Zeichen zaehlen
    } else { // Pruefung ob die Schleife erlaubt
        cout << "Achtung! Deine Zahl ist ungültig" << endl;
    }
} while(falschesZeichen != '\n'); // Pruefung ob die Schleife erlaubt
ungültigeZeichen = -1; // Das letzte '\n' war nicht ungültig

```

Von Java zu C++ - Daten - Referenztypen

Eine Referenz in C++ ist eine unlösbare Verknüpfung / ein Alias für eine bestehende Variable.

Referenztypen entstehen durch Anhängen von & an den Typnamen

```
int i {1};  
int& i_ref {i}; // i_ref ist Referenz auf i  
int j {i}; // Initialisierung von j mit dem Wert von i  
  
j = i_ref; // Dasselbe wie j = i;  
i_ref = 2; // Dasselbe wie i = 2;  
j = 3;  
i = j;  
cout << i << "-" << i_ref; // Ausgabe: 3 3
```

L- UND R-WERTE: KONSTANTE REFERENZEN

Die Verknüpfungsziele für Referenzen nennt man auch **L-Werte**. Sie besitzen einen Namen und eine damit verknüpfte Speicheradresse. Sie können auf der linken Seite einer Zuweisung stehen.

Im Gegensatz dazu haben Konstanten und temporäre Ausdrücke (z.B. **a+b**) nicht notwendig einen Namen und können nur auf der **rechten** Seite einer Zuweisung stehen (**R-Werte**).

const T& t_ref {wert}; vereinbart eine Referenz auf den L- oder R-Wert **wert** und erlaubt nur **lesenden** Zugriff.

Java	C++
Referenzen zeigen auf Objekte oder Arrays. Auf die Standarddatentypen kann nicht referenziert werden	Referenzen können für jeden Datentyp angelegt werden, dies wird durch Anhängen von & kenntlich gemacht
Referenzen können auf NULL verweisen	Referenzen können nur an existierende Variablen gebunden werden
Referenzen können jederzeit auf ein anderes Ziel verweisen	Eine einmal initialisierte Referenz ist unlösbar mit ihrem Ziel verbunden
Änderungen an der Referenz wirken sich auf das Original aus und umgekehrt	
Es können beliebig viele Referenzen für ein Original angelegt werden	
<code>==</code> vergleicht Identität, <code>equals</code> vergleicht Inhalte	<code>==</code> vergleicht Inhalte
<code>=</code> weist Speicheradressen zu	<code>=</code> weist Inhalte zu

Von Java zu C++ - Daten - Speicherklassen

Beim Start eines Programms wird Speicherplatz für folgende Segmente bereit gestellt

- **Code-Segment:** Enthält das Programm, sowie Konstanten
- **Stack-Segment:** Enthält automatische Variable
- **Heap-Segment:** Enthält dynamische Variable
- **Daten-Segment:** Enthält globale und statische Variable

- **Code- und Datensegment — statisch**
werden statisch zum Programmstart erzeugt

- **Stack und Heap — dynamisch**
werden dynamisch zur Laufzeit belegt und wieder freigegeben.
Dabei wird der Stack vom System verwaltet, während der Heap vom Benutzer verwaltet wird

- Dynamische Variablen werden durch den Benutzer auf dem Heap eingerichtet und verwaltet
- Die Lebensdauer bestimmt der Benutzer durch explizite (Bedarfs-)Anforderung und Freigabe von Speicherplatz ("dynamisch")
- Sie besitzen keinen eigenen Variablennamen, sondern werden über zugeordnete Zeiger angesprochen
- Sie werden im späteren Verlauf der Vorlesung thematisiert

```
#include <iostream>
using namespace std;
int test(){
    static int i=1; // wird beim Programmstart
                    // EINMALIG initialisiert
    return i++;
} // i ueberlebt das Ende der Verbundanweisung

int main(){
    cout << test() << endl; // 1
    cout << test() << endl; // 2
    cout << test() << endl; // 3
    //cout << i << endl; // i hier unbekannt
    return 0;
}
```

test.cpp

```
extern int global; // extern definierte Variable
static int statisch=1; // modulweit sichtbare Variable
void test(int n){

    global=statisch; // beide Variablen hier bekannt
    statisch=n;
}
```

main.cpp

```
#include <iostream>
using namespace std;
void test(int);
int global=10; // programmweite globale Variable
int main(){
    cout << global << endl; // 10
    test(3); cout << global << endl; // 1
    test(7); cout << global << endl; // 3
    test(0); cout << global << endl; // 7
    return 0;
}
```

- Auf dem Stack werden lokale Variablen einer Funktion oder Verbundanweisung abgelegt
- Sie besitzen nach der Deklaration einen undefinierten Wert
- und werden wieder vom Stack gelöscht, sobald der umgebende Block (Funktion, Verbundanweisung) verlassen wird ("automatisch")
- Bei jedem Aufruf einer Funktion wird auf dem Stack Speicher für die Parameter und die lokalen Variablen belegt
- Die Speicherung auf dem Stack geschieht nach dem LIFO (Last In First Out) Prinzip

Lokale statische Variablen

- werden am Beginn eines Funktionsrumpfes oder einer Verbundanweisung deklariert und durch das Schlüsselwort **static** gekennzeichnet
- werden im (statischen) Datensegment abgelegt und besitzen daher eine **feste** Speicheradresse
- daher bleibt ein einmal zugewiesener Wert während der gesamten Programmumlaufzeit erhalten, auch wenn der zugehörige Programmblock bereits verlassen wurde
- auf sie kann nur innerhalb des zugehörigen Programmblocks zugriffen werden (**lokal statisch**)
- Statische Variablen werden beim Programmstart **einmalig()** automatisch mit 0 initialisiert falls keine Initialisierung vom Benutzer vorgenommen wird

Globale Variablen

- werden außerhalb von Funktionen deklariert und im Datensegment abgelegt, so dass sie wie statische Variablen während der gesamten Laufzeit des Programms existieren
- Globale Variablen werden ebenfalls (einmalig) mit 0 initialisiert, falls keine Initialisierung vom Benutzer vorgenommen wird
- modulweite Sichtbarkeit:
Deklaration durch das Schlüsselwort **static** außerhalb von Funktionen macht die Variable **sichtbar** für das gesamte Modul (=Datei/Übersetzungseinheit)
- programmweite Sichtbarkeit:
Deklaration **ohne** Schlüsselwort außerhalb von Funktionen macht die Variable **programmweit global**. Anderen Modulen wird die externe Variablendefinition durch das Schlüsselwort **extern** bekannt gemacht. Es führt lediglich zu einer Deklaration, aber keiner Definition der Variablen.

test.cpp

```
extern int global; // extern definierte Variable
static int statisch=1; // modulweit sichtbare Variable
void test(int n){
    int statisch=1; // statisch jetzt lokal und
    global=statisch; // ueberdeckt den au遝neren Wert
    statisch=n;
}
```

main.cpp

```
#include <iostream>
using namespace std;
void test(int);
int global=10; // programmweite globale Variable
int main(){
    cout << global << endl; // 10
    test(3); cout << global << endl; // 1
    test(7); cout << global << endl; // 1
    test(0); cout << global << endl; // 1
    return 0;
}
```

Speicher- klasse	Schlüssel- wort	Segment	Initiali- sierung	Gültig- keit	Lebens- dauer
automatisch	-	Stack	nein	Block	Block
lokal, statisch	static	Daten	ja	Block	Programm
	static - extern			Modul Programm	Programm
dynamisch	-	Heap	nein	-	Benutzer

Von Java zu C++ - Daten - Speicherklassen - Ergänzung

Typ-Modifizierer — Fügt dem Datentyp eine Modifikation hinzu

signed	vorzeichenbehaftet
unsigned	vorzeichenlos
short	kürzer
long	länger

Typ-Qualifizierer — Fügt den Variablen neue Eigenschaften hinzu

const	Variable kann nach der Initialisierung nicht mehr verändert werden
volatile	Wert der Variablen kann durch Prozesse außerhalb des Programms verändert worden sein
mutable	Wert der Variablen kann durch KONSTANTE KLASSENMETHODEN verändert werden

Speicherklassen-Spezifizierer — Lebensdauer und Sichtbarkeit

auto	Lebensdauer und Sichtbarkeit: aktueller Block
static	Lebensdauer: Programm; Sichtbarkeit: Block der Definition
extern	Lebensdauer und Sichtbarkeit: Programm

Von Java zu C++ - Daten - Ausnahmebehandlung

Die Ausnahmebehandlung in C++ entspricht weitgehend der in Java.

Sie dient der Behandlung von durch äußere Einflüsse verursachten Fehlern im Programmablauf

STATIONEN DER AUSNAHMEBEHANDLUNG

- Fehlererkennung → "Werfen" eines Ausnahmeobjektes (**throw**)
- Code für Normalfall → Markiert durch **try-Block**
- Code für Ausnahmefall → "Fangen" von Ausnahmen im **catch-Block**

Beim Werfen einer Ausnahme wird der entsprechende Programmblock sofort verlassen, der zugehörige Programmstack abgewickelt und in die nächst höhere Programmebene gesprungen.

Dieser Vorgang wiederholt sich, bis:

- ein **try-Block** erreicht wird, der eine passende Ausnahme bereitstellt (d.h. **catch** fängt den passenden Ausnahmetyp)
- bis zur obersten Programmebene, wo das Programm verlassen wird

Von Java zu C++ - Daten - Ausnahmebehandlung

```
double kehrwert(int n) {
    if (n==0) throw string("Division durch Null");
    return 1./n;
}
int main() {
    int zahl;
    double inverse;
    do {
        cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
        cin >> zahl;
        try {
            inverse=kehrwert(zahl);
            cout << inverse << endl;
        }
        catch(const string& s) {
            cout << s << endl << "Bitte neuen Wert eingeben:" ;
        }
    } while(zahl!= -1);
    return 0;
}
```

Methode Kehrwert wirft ggf. ein Ausnahmeobjekt vom Typ string.

Schlüsselwort try markiert, dass der folgende Block Ausnahmen werfen kann, die im nächsten, zur Ausnahme passenden catch-Block (der in den Klammern stehende Typ muss dem Ausnahmetyp entsprechen) gefangen und behandelt oder weitergeworfen werden muss.

Der Benutzer gibt eine Zahl ein, dessen Inverse berechnet werden soll.

Anschließend wird versucht, den Kehrwert hierzu zu berechnen.

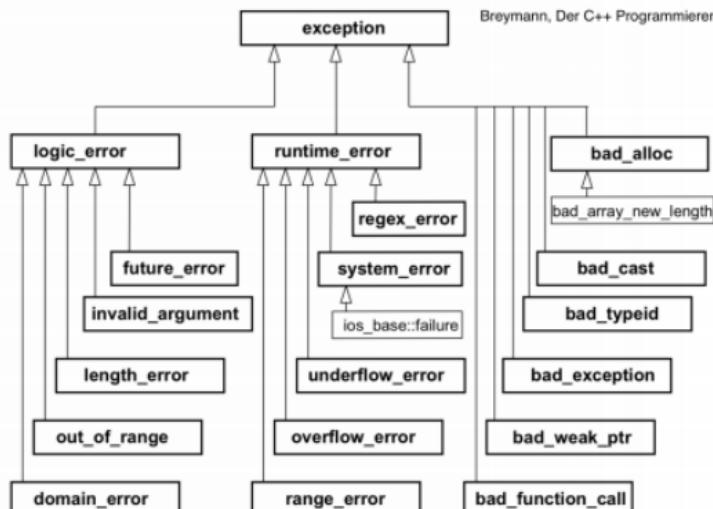
Bei Aufruf der Funktion Kehrwert wird eine Ausnahme vom Typ string geworfen, wenn die übergebene Zahl mit 0 übereinstimmt.

In diesem Fall wird die Ausführung des Codes bis zum nächsten, passenden catch-Block übersprungen.

Abschließend wird der Code des entsprechenden catch-Blockes ausgeführt und das Programm nach diesem Block regulär ausgeführt.

- Catch-Block sollte nach der Behandlung der Ausnahme wieder den ursprünglichen Programmzustand wiederherstellen (Exception-Sicherheit)
- In Java wird dies durch die finally -Klausel erreicht
- In C++ müssen dynamische Ressourcen so erzeugt/gelöscht werden, dass beim Auftreten von Ausnahmen keine Speicherlecks oder verwäiste Objekte entstehen

Von Java zu C++ - Daten - Ausnahmebehandlung - Klassen



Den erbenden Klassen kann im Konstruktor ein string-Objekt (Fehlermeldung) übergeben werden. Die geerbte Methode **const char* what()** liefert dieses als Zeichenkettenkonstante zurück

Tabelle 7.1: Bedeutung der Exception-Klassen

Breymann, Der C++ Programmierer

Klasse	Bedeutung	Header
exception	Basisklasse	<exception>
logic_error	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorebdingungen	<stdexcept>
invalid_argument	ungültiges Argument bei Funktionen	<stdexcept>
length_error	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	<stdexcept>
out_of_range	Bereichsüberschreitungsfehler	<stdexcept>
domain_error	anderer Fehler des Anwendungsbereichs	<stdexcept>
future_error	für asynchrone System-Aufrufe	<future>
runtime_error	nicht vorhersehbare Fehler, z.B. datenabhängige Fehler	<stdexcept>
regex_error	Fehler bei regulären Ausdrücken	<regex>
system_error	Fehlermeldung des Betriebssystems	<system_error>
range_error	Bereichsüberschreitung	<stdexcept>
overflow_error	arithmetischer Überlauf	<stdexcept>
underflow_error	arithmetischer Unterlauf	<stdexcept>
bad_alloc	Speicherzuweisungsfehler (Details siehe Abschnitt 7.2)	<new>
bad_typeid	falscher Objekttyp (vgl. Abschnitt 6.9)	<typeinfo>
bad_cast	Typumwandlungsfehler (vgl. Abschnitt 6.8)	<typeinfo>
bad_weak_ptr	kann vom shared_ptr-Konstruktor geworfen werden	<memory>
bad_function_call	wird ggf. von function::operator()() geworfen	<functional>

Von Java zu C++ - Funktion - Wert und Referenzparameter

- In Java können primitive Datentypen nur per Wert und Arrays/Objekte nur per Referenz an Methoden übergeben werden
- In C++ wird jeder primitive Datentyp und jedes Objekt stets per Wert übergeben. Für die Referenzübergabe müssen die Formalparameter einer Funktion/Methode Referenztypen sein.

```
// T bezeichnet einen beliebigen C++ Typ oder Klasse
void f(T t); // akzeptiert jeden nach T konvertierbaren
// Variablen (L-Wert). t ist eine Referenz
// auf den uebergebenen Wert. Aenderungen
// des Wertes wirken sich auf das Original aus

void f(T& t); // akzeptiert jede nach T konvertierbare
// Variable (L-Wert). t ist eine Referenz
// auf den uebergebenen Wert. Aenderungen
// des Wertes wirken sich auf das Original aus

void f(const T& t); // akzeptiert jeden nach T
// konvertierbaren L- oder R-Wert
// und besitzt kein Schreibrecht
// darauf
```

Java — call by value

```
public class K {
    static public // call by value fuer primitive Datentypen:
    void swap(int x, int y) { // x,y sind Kopien von a,b
        int hilfvar; x=hilf; // Kopien werden vertauscht
        int hilfvar; y=hilf; // ...und geloescht
    }

    static public void main(String[] args) {
        int a=1, b=2; // a==1, b==2
        swap(a,b); // a==1, b==2 -> call by value
    }
}

C++ — call by reference
```

void swap(int& x, int& y) { // x,y sind Verweise auf a,b
 int hilfvar; x=hilf; // Inhalte werden vertauscht
} // Verweise werden wieder geloescht

```
int main() {
    int a {1}, b {2}; // a==1, b==2
    swap(a,b); // a==2, b==1 -> call by reference
} // ohne &-Zeichen oben: Dasselbe Ergebnis wie im Java-Beispiel
```

Es sei obj ein Objekt einer Klasse Obj mit Attribut int i;

Java — call by reference

```
public class K {
    static public // call by reference fuer Referenztypen
    void set(Obj obj, int _i) { // obj ist Referenztyp
        obj.setze_i(_i); // Wrapper fuer Methode Obj.setze_i
    }
    static public void main(String[] args) {
        Obj obj = new Obj();
        set(obj, 10); // obj.i == 10 -> call by reference
    }
}
```

C++ — call by reference

```
void set(Obj& obj, int _i) { //!obj als Referenz uebergeben!
    obj.setze_i(_i); // Wrapper fuer Methode Obj::setze_i
}
int main() {
    Obj obj; // Wertesemantik, keine Referenz!
    set(obj, 10); // obj.i == 10 -> call by reference
} // ohne &-Zeichen oben: Attribut i in Objektkopie gesetzt!
```

Von Java zu C++ - Funktion - Operatorüberladung

Alle Operatoren in C++ sind als Operatorfunktionen implementiert

OPERATORÜBERLADUNG

Bis auf `::`, `.`, `*`, `?:`, `sizeof`, `x_cast` und `typeid` lassen sich alle Operatoren überladen.

<Rückgabetyp> **operator**+ (Parameterliste der Operanden);
↑ beliebiges Operatorzeichen

REGELN BEIM ÜBERLADEN VON OPERATOREN

- Operator-Prioritäten werden beim Überladen nicht verändert
- zusammengesetzte Operatoren (z.B. `+=`) passen sich nicht automatisch der neuen Bedeutung von `+` an, sie müssen dazu explizit überladen werden
- binäre Operatoren (z.B. `+`, `-`, `*`, `/`) müssen in der Regel einen neuen Ergebniswert erzeugen und zurückgeben

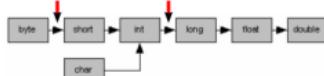
Von Java zu C++ - Funktion - Operatorüberladung

Erinnerung: Promotion und Standardkonvertierungen in C++

TYPKONVERTIERUNGEN

- Promotion (werterhaltende Konversion)
 - `bool`
 - `ganzahlig: char`
 - `short`
 - `float` → `double`
- Standardkonvertierungen (i.allg. nicht werterhaltend)
 - `int` → `long`
 - `int` → `float`/`double`

Zum Vergleich: In Java werden bei Bedarf die folgenden Konvertierungen durchgeführt:



Bei jedem Aufruf einer überladenen Funktion wird jeder Parameter des Aufrufs nach den folgenden Kriterien der Überladungsauflösung beurteilt

Krit.	Bezeichnung	Beschreibung	Beispiel
1	Übereinstimmung	Der Typ entspricht dem verlangten (bis auf triviale Umwandlungen, z.B. <code>T</code> → <code>const T</code> ; Zuordnung eines Schablonentyps)	<code>int</code> → <code>int</code> <code>int</code> → <code>const int</code>
2	Promotion	Werterhaltende Konversion	<code>float</code> → <code>double</code>
3	Standardkonvertierung	Im allgemeinen nicht werterhaltende Konversion	<code>int</code> ↔ <code>float</code>
4	Benutzerdefinierte Konvertierung	Typumwandlungsoperator	
5	Übereinstimmung mit variabler Parameterliste	Parameter lässt sich in variable Parameterliste übergeben	

Nach Beurteilung der Parameter nach den Kriterien der Überladungsauflösung wählt der Compiler den passendsten Kandidaten nach folgendem Auswahlverfahren:

WELCHER KANDIDAT WIRD GEWÄHLT?

Aufruf: `f(d, i)` mit `d: double, i: int`

Kandidat	Parameter 1	Parameter 2
<input type="checkbox"/> <code>f(double, long)</code>	1	3
<input type="checkbox"/> <code>f(float, int)</code>	3	1
<input type="checkbox"/> <code>f(float, long)</code>	3	3
<input checked="" type="checkbox"/> keine Auswahl		

AUSWAHLVERFAHREN

Eine Funktion wird aufgerufen, falls

- für jeden Parameter das Zuordnungskriterium minimal ist
- die so gefundene Signatur eindeutig ist

Andernfalls wird der Aufruf abgelehnt.

Aufruf: `f(i, s)` mit `i: int, s: short`

Kandidat	Parameter 1	Parameter 2
<input type="checkbox"/> <code>f(double, long)</code>	3	3
<input checked="" type="checkbox"/> <code>f(float, int)</code>	3	2
<input type="checkbox"/> <code>f(float, long)</code>	3	3
<input type="checkbox"/> keine Auswahl		

WARNUNG | Ausnahme bei Promotion

Von `int` nach `long` findet keine Promotion statt!

Von Java zu C++ - Klassen - Klassen in C++

In C++ wird der Quellcode einer Klasse auf 2 Dateien aufgeteilt

- Die **Headerdatei** ist die öffentliche Schnittstelle, die im wesentlichen nur Vereinbarungen von Klassen und Methoden enthält, sowie von Funktionen außerhalb von Klassen
- Die **Implementationsdatei** enthält alle Implementierungen. Zur korrekten Zuordnung einer Implementation muss mittels des Bereichsauflösungsoperators :: die zugehörige Klasse dem Methodennamen vorangestellt werden.
- Durch diese Aufteilung kann auch nachträglich die Implementation von Methoden verändert werden, ohne die Schnittstelle zu verändern

Test.h	Test.cpp	main.cpp
<pre>void f(int); class Test { int i {0}; // Init public: int gib_i(); };</pre>	<pre>f(int i) { cout << "f:::"; } int Test::gib_i() { return i; }</pre>	<pre>int main() { f(5); Test test; test.gib_i(); }</pre>

- Im Header werden in der Regel keine Implementierungen vorgenommen. Falls doch, werden Methoden als **inline** interpretiert, d.h. der Compiler ersetzt jeden Aufruf direkt durch die Implementation
„Vorsicht: Codeaufblähung“
- Die Implementationsdatei und Module, die den Code verwenden, müssen die zugehörige Headerdatei mittels **#include** einbinden. Dadurch kann der Compiler syntaktisch die korrekte Verwendung der eingebundenen Funktionen/Methoden prüfen
- Um Mehrfacheinbindungen zu verhindern, enthält die Headerdatei einen sogenannten **Wächter**, der vom Präprozessor ausgewertet wird und das Modul bedingungsabhängig dem Compiler zuführt

Test.h	Test.cpp	main.cpp
<pre>#ifndef Test_h #define Test_h // hier weiter #endif</pre>	<pre>#include "Test.h" // hier weiter</pre>	<pre>#include "Test.h" // hier weiter</pre>

Von Java zu C++ - Klassen - Klassen in C++

In C++ werden Sichtbarkeiten nicht für jede Methode/Attribut einzeln gesetzt, sondern gruppiert in Abschnitten hinter z.B. **public**:

Sichtbarkeit	Java	C++
public		öffentlicher Zugriff
protected	nur erbende Klassen, sowie Klassen im selben Paket haben Zugriff	nur erbende Klassen haben Zugriff
package	(Standard) — nur Klassen im selben Paket haben Zugriff	gibt es in C++ nicht!
private	nur die eigene Klasse kann zugreifen	(Standard) — nur die eigene Klasse kann zugreifen

SONDERFALL: **struct**

Strukturen in C++ sind Klassen, bei denen die Sichtbarkeit **public** als Standard voreingestellt ist

Der Spezifizierer **static** kennzeichnet Klassenvariablen und Klassenmethoden

- Klassenvariablen und -konstanten dürfen nur einmal außerhalb der Klasse initialisiert werden, oder **inline** innerhalb der Klassenvereinbarung
- Außerhalb der Klasse geschieht der Zugriff über den Bereichsauflösungsoperator ::
- statische Methoden dürfen nur Klassenvariablen und keine Objektvariablen verwenden

Test.h

```
class K {
    static int i;
    static inline int j {1};
    static const int k;
    static const int l {1};
    static int feld[1];
public:
    static void methode();
}
```

Test.cpp

```
int K::i {1};
const int K::k {1};
void K::methode() {}
```

 v— leitet Initialisierungsliste ein
K::K(size_t _anz) : anz(_anz), vec(_anz) {}
// Vektor mit vorgegebener Länge —^

```
K::K(const K& orig)
: anz(orig.anz), vec(orig.vec) {}

K& K::operator=(const K& rhs) {
    anz = rhs.anz;
    vec = rhs.vec;
    return *this;
}
```

- Initialisierungslisten initialisieren Objektvariablen bereits vor Betreten des Ausweisungsblocks
- Die Initialisierungsreihenfolge ist durch die Reihenfolge der Attribute in der Klassendeklaration festgelegt!
- Initialisierung ist bei Objekten anderer Klassen ein Konstruktoraufzug (einzige Möglichkeit, überladene Konstrukturen aufzurufen.)
- Zuweisungsoperator nimmt explizite Komponentenzuweisungen vor

Objekte werden, wie jede andere Variable auch, auf dem Stack erzeugt und nach Verlassen des Programmblocks wieder zerstört

```
K f(K k) { return k; }

int main() {
    K k1; // Anlegen mit SK
    K k2{1}; // Anlegen mit ueberladenem Konstruktur
    K k3 {1}; // Initialisierung: Auswahl passender Konstruktur
              // (K(int) )
    k1 = k3; // kopiert Inhalte
    k2 = f(k3); // 2xK + 2*D ! —> besser: Referenzuebergabe
    // Destruktoraufzug fuer k1, k2, k3
}
```

WARNUNG |

- Der Ausdruck **K k()**; erzeugt in C++ kein K-Objekt, sondern wird als Vereinbarung einer parameterlosen Funktion **k()** interpretiert, die ein K-Objekt zurückliefert. Solange diese "Funktion" nicht verwendet wird, gibt es keine Fehlermeldung des Compilers!
- Destruktoren sollten niemals Ausnahmen werfen

 Warum? ↴

In C++ werden Sichtbarkeiten nicht für jede Methode/Attribut einzeln gesetzt, sondern gruppiert in Abschnitten hinter z.B. **public**:

Java

```
public class K {
    private int i = 0; // Voreinstellung private
    protected int j = 0;
    public K() {}
    public void methode(){}
}
```

C++

```
class K { // immer public
    int i {0}; // Voreinstellung private
protected: // ab hier: protected
    int j {0};
public: // ab hier: public
    K() {}
    void methode() {}
}; // Semikolon nicht vergessen!
```

Die Sichtbarkeitsabschnitte können beliebig oft und in beliebiger Reihenfolge aufgeführt werden

Der Qualifizierer **const** ist eine Zusicherung an den Compiler:

```
const K::methode(const int& i) const {}
```

- Das erste **const** sichert zu, dass das zurückgegebene Objekt nicht verändert werden kann
- Das zweite **const** sichert zu, dass der Übergabeparameter eine konstante Referenz ist (d.h. nur mit Leserecht)
- Das dritte **const** bezieht sich auf das Objekt, an dem die Methode aufgerufen wird. Es sichert zu, dass die Methode keine Objektvariablen verändert.
- Ausnahme: Der Qualifizierer **mutable** vor einer Objektvariablen, lässt Änderungen von konstanten Methoden zu.
- Konstante Objekte können (wie andere Variablen auch) im Programm vereinbart werden. Für konstante Objekte stehen nur als konstant markierte Methoden zur Verfügung.
- Methoden können sowohl in einer konstanten und einer nicht-konstanten Version vorliegen. Daher muss der Qualifizierer **const** sowohl bei der Vereinbarung, als auch bei der Implementation angefügt werden.

In C++ gibt es keine Wurzelklasse **Object**, aber Methoden, die automatisch angelegt werden

- Standardkontruktor (SK)
- Kopierkontruktor (KK) bei Wertüber-/rückgabe stets aufgerufen!
- Zuweisungsoperator für Komponentenzuweisungen(!)
- Destruktor (D): "letzter Wille des Objekts vor seiner Zerstörung"

 Java: `public void finalize();` wichtig in Kapitel ↴ **ZEIGER**

```
class K {
    size_t anz {0};
    vector<int> vec;
public:
    K() = default; // Standardkontruktor behalten
    K(size_t);
    // Ueberladung
    K(const K&); // Kopierkontruktor
    K& operator=(const K&); // Zuweisungsoperator
    ~K() = default; // Standarddestruktur verwenden
};
```

- =**default**: übernimmt die Standardimplementation des Systems
- =**delete**: ermöglicht das Sperren eines Aufrufs

Es gibt standardmäßig KEINE der Methoden **clone**, **equals**, **toString**

Der Sequenzkontruktor erlaubt variable Initialisierungslisten: Vereinbart man zusätzlich die Methode `#include< initializer.list >;`

```
K::K(initializer_list<int> liste): anz(liste.size()) {
    for (int wert: liste) vec.push_back(wert);
}
```

```
int main() {
    K k1; // Anlegen mit SK
    K k2{1}; // Anlegen mit ueberladenem Konstruktur
    K k3 {1}; // Auswahl passender Konstruktur
              // Sequenzkontruktor hat Vorrang
    K k4 {1, 2, 3, 4, 5, 6};
}
```

erhält man die Belegungen

```
k1: 0
k2: 1| 0
k3: 1| 1
k4: 6| 1 2 3 4 5 6
```

Von Java zu C++ - Klassen - Operatormethoden

Operatormethoden sind überladene Operatoren in Klassen, der erste Operand wird dabei automatisch durch this vorbelegt und taucht nicht mehr explizit in der Parameterliste auf.

```
class K {
    vector<double> werte
public:
    //!
    //! typisches Beispiel I: Vergleichsoperatoren
    // in der Regel implementiert man nur
    bool operator==(const K&) const;
    bool operator<(const K&) const;
    // alle anderen Vergleichsoperatoren
    // sind davon abgeleitet
    bool operator!=(const K& rhs) { return !operator==(rhs); }
    bool operator<=(const K& rhs)
    { return ( operator==(rhs) || operator<(rhs) ); }
    bool operator>(const K& rhs) { return !operator<=(rhs); }
    bool operator>=(const K& rhs) { return !operator<(rhs); }
    //!
    //! typisches Beispiel II: Indexoperator
    // Variante mit Schreibrecht auf den Vektorinhalten
    double& operator[](int i) { return werte[i]; }
    // Variante ohne Schreibrecht auf den Vektorinhalten
    const double& operator[](int i) const { return werte[i]; }
    // normalerweise beeinhaltet die Implementation auch
    // noch eine Gueltigkeitspruefung des uebergebenen Index i
};

class K {
public:
    // Praefix-Operatoren
    K& operator++() { /* Implementation */ return *this; }
    K& operator--() { /* Implementation */ return *this; }

    // += und -= (ggf. auf ++/-- zurueckfuehren)
    K& operator+=(Param) { /* Implementation */ return *this; }
    K& operator-=(Param) { /* Implementation */ return *this; }

    // Weitere, z.B. *= und /= (ggf. auf +=/= zurueckfuehren)...
};
```

Von Java zu C++ - Klassen - Operatormethoden

```
class K {
public:
// Präfix-Inkrement
K& operator++() /* Implementation */ return *this; }

// + und (ggf. auf ++ zurückföhren)
K& operator+=(Param) /*Implementation*/ return *this; }

// ausserhalb der Klasse
// Postfix Inkrement
K operator++(K& k, int) {
K temp(k);
++k;
return temp;
}

// Addition mit neuem Ergebnis
K operator+(const K& lhs, const K& rhs) {
K ergebnis(lhs);
return ergebnis+=rhs;
}
```

- Operatoren wie `+`, `-`, `*`, `/`, ... müssen ein **neues** Ergebnisobjekt zurückliefern: Die rechte Seite der Anweisung `c=a+b` wird zunächst als temporäres Objekt erzeugt, bevor es der linken Seite zugewiesen wird
- Andererseits ist die Funktionalität der Operatoren oftmals bereits implementiert in den korrespondierenden Operatoren auf dem aktuellen Objekt (wie `+=`, `-=`, `*=`, `/=`, ...)
- allgemein gilt außerdem für symmetrische, binäre Operatoren wie `(+,-,*,-,/,...)`, dass sie **nicht** als Methode, sondern als **globale Operatorfunktion** implementiert werden sollen, um Typkonvertierungen in beiden Operanden gleichberechtigt zuzulassen
- damit verlieren sie den Zugriff auf alle privaten Daten der Klasse und müssen mithilfe öffentlicher Methoden (Konstruktoren, Operatormethoden) implementiert werden
- nach den Regeln der Überladungsauflösung sind dann auch Aufrufe für beide Operanden gleichberechtigt möglich

WARNUNG |

- Wenn Operatoren überladen werden, behalten sie ihre Assoziativität und Priorität
- Die Operatoren `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=` passen sich **NICHT** an, wenn die zugehörigen Operatoren `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `<<`, `>>` überladen wurden
- Die Operatoren `=`, `[]`, `()` und `->` können nur als nicht-statische Operatoren überladen werden, da sie **zwingend** an ein Objekt gebunden sind.
- Die Stromoperatoren `<<` und `>>` benötigen als ersten Operanden ein Objekt der Stromklassen und können daher nicht innerhalb eigener Klassen überladen werden. Sie müssen stets außerhalb der Klasse als Operatorfunktionen überladen werden.

Von Java zu C++ - Klassen - Benutzerdefinierte Typumwandlungen

gegeben: Typ T
Klasse K

gesucht: Umwandlung `T ↔ K`

Lösung: $T \rightarrow K$ `K(const T&);` (Konstruktor)
 $K \rightarrow T$ `operator T();` (Typumwandlungsoperator)

```
class Zahl {
private:
int _inhalt;
public:
Zahl(int _inhalt) : _inhalt(_inhalt) {}
operator int() { return _inhalt; }
// Der Rueckgabetyp des Typumwandlungsoperators ist
// sein Name (keine explizite Angabe!)
};
```

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

class X { // Klasse mit Typumwandlung durch den Konstruktor
public:
/*explicit*/ // verhindert:
X(int); // Benutzerdefinierte Konversion
{ cout << "X<" << endl; } // von int nach X durch Konstruktor
~X() { cout << ">X" << endl; }
};

void f(X) { // Eine Funktion auf Objekten der Klasse X
cout << "f aufgerufen" << endl;
}

int main() { // Demo
{(11); // Konvertierung int --> X
(static_cast<X>(11)); // mittels X(int)

(0.5); // Konvertierung double --> int --> X
(static_cast<X>(0.5));
}
```

Von Java zu C++ - Klassen - Benutzerdefinierte Typumwandlungen

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

class B; // Vorwaertsdeklaration! ---> macht Typ B bekannt

class A { // Klasse A, die B verwendet
public:
    A(int x): wert(x) {}
    A(const B&): wert=1; // Typwandlung B->A
    int get_v() const { return wert; }
};

class B { // Definition von B, die A verwendet
public:
    operator A() { return A(2); } // Typwandlung B->A
};

int main() { // Demo
    B b; // Welche Typwandlung findet statt?
    cout << static_cast<A>(b).get_v() //---> Compilerabhaengig
        << endl;
}
```

Von Java zu C++ - Klassen - Funktionsobjekte und Prädikate

Ein **Funktionsobjekt** entsteht durch Überladung des Klammer-Operators innerhalb einer Klasse:

Rückgabetyp **operator()** (**Parameterliste**);

- Der Aufruf des Operators ähnelt dem Funktionsaufruf, der mit dem aufrufenden Objekt verbunden ist
- Funktionsobjekte können in C++ die Rolle einfacher Interfaces übernehmen, die genau eine Methode bereitstellen (nämlich den Klammeroperator)
- Eine denkbare Anwendung sind "Funktionen mit Gedächtnis", d.h. Klassen mit überladenem Klammeroperator und Attributen, die im Konstruktor initialisiert und zur "Berechnung eines Funktionswertes" herangezogen werden

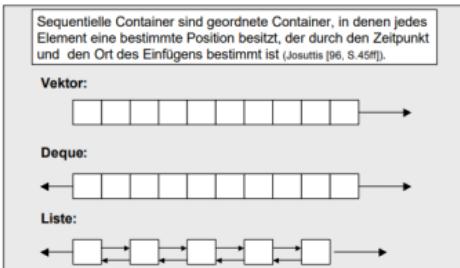
Ein **Prädikat** ist ein Funktionsobjekt mit spezieller Signatur:

bool operator() (**Parameterliste**) **const**;

Sie können zur Abfrage von Eigenschaften genutzt werden

Von Java zu C++ - Sequentielle Container

Container der STL - Sequentielle Container



ERLÄUTERUNG: CONTAINER

Abstrakte Datentypen zur Speicherung von Daten gleichen Typs

• realisiert als Klassenschablone (z.B.: `vector<double>`)

• andere Container erlauben z.B.

- Schnelles Einfügen von Elementen in der Mitte
- Schnelles Suchen nach Elementen
- ...

• Klassenbibliothek nennt sich

STL - Standard Template Library

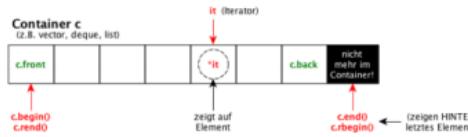
vector	deque	list
<pre>vector<int> v, vv; v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(0); // 4 1 2 3 0 // front: 1 // back: 0</pre>	<pre>deque<int> dq, dqq; dq.push_back(1); dq.push_back(2); dq.push_back(3); dq.push_front(0); // 4 0 1 2 3 // front: 0 // back: 3</pre>	<pre>list<int> l, ll; l.push_back(1); l.push_back(2); l.push_back(3); l.push_front(0); // 4 0 1 2 3 // front: 0 // back: 3</pre>
<pre>vvv; cout << (vv==v);</pre>	<pre>dqq=dq; cout << (dqq==dq);</pre>	<pre>ll.pop_front(); // 3 1 2 3 // front: 1 // back: 3 cout << (ll==l);</pre>
<pre>vv.pop_back();</pre>	<pre>dqq.pop_front(); // 3 1 2 3 // front: 1 // back: 3 cout << (dqq==dq);</pre>	<pre>ll.pop_front(); // 3 1 2 3 // front: 1 // back: 3 cout << (ll==l);</pre>
<pre>// true</pre>	<pre>// true</pre>	<pre>// true</pre>

Bibliothek	vector	deque	list
Beschreibung	dynamisches Feld	doppelseitige Schiene	Liste
logisches Speicherbild	front ... back	front ... back	front ... back
Konstruktion	<pre>vector<T> c; vector<T> c(n); vector<T> c(n,v); vector<T> c({...}); vector<T> c(d);</pre>	<pre>deque<T> d; deque<T> d(~);</pre>	<pre>list<T> l; list<T> l(~);</pre>
Anfügen/Entfernen	<pre>c.push_back(v); c.pop_back(); c.front(); c.back(); c[i]; c.at(i);</pre>	<pre>+c.push_front(v); +c.pop_front(); ~</pre>	<pre>+c.push_back(v); +c.pop_back(); ~</pre>
Elementzugriff			<pre>~c[i]; ~c.at(i);</pre>
Globalzugriff	<pre>nvc.size(); c.reserve(n); c.clear(); bvc.empty(); = ==, < (lex) ~=>, <=, >=</pre>	<pre>~</pre>	<pre>+c.unique(bvp); +c.remove(v); +c.remove_if(p); +c.sort(cmp); +c.reverse(); +c.merge(d,cmp);</pre>

Legende:
int n; T v; **initializer.list<T> { ... };** bool b; CxD d; wobei C den jeweiligen ContainerTyp bezeichnet
Optionaler Parameter: Prädikat p (unär), tp, cap (binär) für Argumente vom Typ T

~=: Dasseltiefe wie link: -: zuordnen; -: ohne
~~=: abgeleitet, Anforderung: Typ T muss ==, <, > unterstützen, lexicographischer Vergleich gemäß logischem Speicherbild

Von Java zu C++ - Sequentielle Container



WARNUNG | Referenzierung beim reverse_iterator

`reverse_iterator` geht von hinten nach vorne.

→ Referenziert immer auf vorausgehendes Objekt, da `c.rbegin()` nicht auf `c.back` zeigt, aber trotzdem referenziert werden kann!

Einfügen

```
list<int> l={1,2,3,4}, list<int> ll={7,8,9};  
list<int>::iterator pos, it; // Iteratoren  
pos=l.begin();pos++;pos++; // Startposition festlegen  
  
it=l.insert(pos,6); l.insert(it,5); // von rechts nach links  
// [6] 1 2 5 6 3 4 |  
l.insert(pos,3,-6); // mehrfaches Einfügen  
// [1] 1 2 5 6 7 8 -6 -6 -6 3 4 |  
  
pos=l.end(); // Einfügen einer Liste am Ende  
l.insert(pos,ll.begin(),ll.end());  
// [1] 1 2 5 6 7 8 -6 -6 -6 3 4 7 8 9 |
```

Funktionalität	Varianten	Beschreibung
Positionierung	<code>it=c.begin();</code> <code>it=c.end();</code> <code>l.begin();</code> <code>l.end();</code>	
Einfügen		fügt (n) Kopie(n) von e vor pos ein, bzw. die Sequenz** [first,last)***. it zeigt auf eingefügte Kopie
Entfernen		löscht Element *pos* bzw. Sequenz [first,last). it zeigt auf nachfolgende Position bzw. <code>end()</code>
Einsetzen	<code>c.splice(pos,liste);***</code> <code>c.splice(pos,liste, it);</code> <code>c.splice(pos,liste, first, last);</code>	setzt die liste, bzw. nur *it oder Listensequenz [first,last) vor pos ein. Eingesetzte Elemente werden liste entnommen

Entfernen

```
list<int> l={1,2,3,4};  
list<int>::iterator pos, it; // Iteratoren  
pos=l.begin();pos++; // Startposition festlegen  
  
it=l.erase(pos); // pos jetzt ungültig  
// [3] 1 3 4 |  
  
it=l.erase(it,l.end()); // Löschen bis zum Listenende  
// [1] 1 |  
  
cout << boolalpha << (it==l.end()) << endl;  
// true
```

Einsetzen

```
list<int> l={1,2,3,4}, list<int> ll={7,8,9};  
list<int>::iterator pos, it; // Iteratoren  
pos=l.begin();pos++;pos++; // Startposition festlegen  
  
l.splice(pos,ll); // ll vor pos einsetzen  
// [1: 7] 1 2 7 8 9 3 4 |  
// [1: 0] |
```

Die Anforderungen beim Elementzugriff bestimmen die Wahl des Containers

Anforderung	Container
indizierter Zugriff	<code>vector</code> , <code>deque</code>
Einfügen/Löschen nur am Anfang oder Ende	<code>deque</code>
Einfügen/Löschen überall	<code>list</code>

hybride Anforderungen:

- anfängliche umfangreiche Dateneingabe an vorgegebenen Positionen
- später indizierter Zugriff notwendig

Lösung:

- Daten in `list`-Container einschreiben
- relevanter Iteratorbereich `[first,last)` umkopieren in `vector`-Container mittels Konstruktor `c(first,last);`

```
list<int> dieListe;  
// Liste befüllen ...  
  
vector<int> derVektor(dieListe.begin(), dieListe.end());  
// Jetzt befinden sich die Daten im Vektor
```

Zeiger

```
// konstante Zeiger
char s[20];           // s ist konstanter Zeiger
char* const t = s;    // t ist konstanter Zeiger
s++; t++;             // geht beides nicht

// konstanter Inhalt
const char* u = "Halli";
u[3]=s[3];            // geht nicht
u = s;                // geht!

// konstanter Zeiger auf konstanten Inhalt
const char* const v = "Hallo";
v[3]=s[3];            // geht nicht
v = s;                // geht auch nicht mehr
```

- Eine Referenz ist ein Zeiger, der an einen existierenden Wert gebunden ist und diese Verbindung ist nicht lösbar
- Zeiger können überall hin zeigen und sind nicht an einen festen Wert gebunden (Iteratorenfunktionalität durch Zeigerarithmetik)
- Zeiger sind flexibler als Referenzen, aber auch fehleranfälliger, da der Programmierer verantwortlich dafür ist, dass die Zeigervariable auf einen definierten Wert zeigt
- Referenzen sind sicherer als Zeiger und syntaktisch angenehmer, da eine Referenz automatisch auf ihren Wert dereferenziert wird (kein Voranstellen von * notwendig)

- Ohne die (korrekte) Freigabe entstehen Speicherlecks!
- Die Lebensdauer von mit new erzeugten Objekten besteht bis zur Speicherfreigabe mit delete. Es ist darauf zu achten, dass der zugehörige Zeiger noch bis zu diesem Zeitpunkt 'lebt'
- delete auf einen Nullzeiger bewirkt nichts

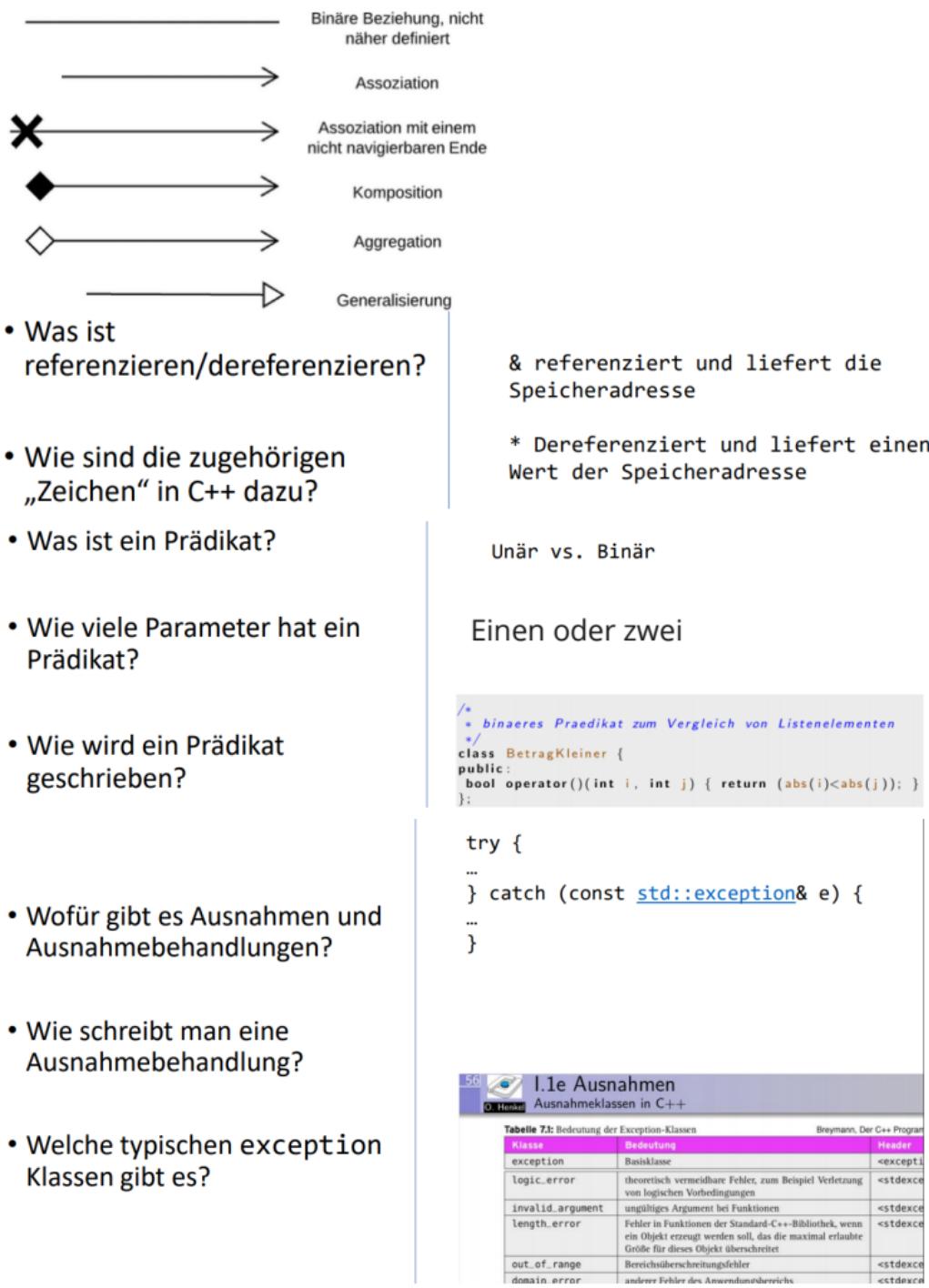
Falls wir Array haben. Operator= löscht alles

```
FavoritenListe::operator=(const FavoritenListe& rhs)
{
    if (this == &rhs) return *this;           // Selbstzuweisung
                                                // verhindern

    for (int i=0; i<favorit.size(); i++){   // alten Speicher
        delete favorit[i];                  // freigeben
    }
    favorit.clear();                        // tiefe Kopie wie

    for (int i=0; i<rhs.favorit.size(); i++) // im KK
        favorit.push_back(new Artikel(*rhs.favorit[i]));

    return *this;
}
```





III.3c Standardmethoden und Polymorphie

Beispielimplementation

BPP POLYMORPHE STANDARDMETHODEN

Kindklasse

Elternklasse

```
class B {  
    int x;  
public:  
    B(int _x): x(_x) {}  
    virtual ~B() {}  
    virtual B* clone()  
    { return new B(*this); }  
    B& operator=(const B& rhs) {  
        if (this==&rhs) return *this;  
        return assign(rhs);  
    }  
    virtual  
    B& assign(const B& rhs) {  
        x=rhs.x;  
        return *this;  
    }  
};
```

```
class U : public B {  
    int y;  
public:  
    U(int _x, int _y)  
    : B(_x), y(_y) {}  
  
    virtual ~U() {}  
    virtual U* clone()  
    { return new U(*this); }  
    U& operator=(const U& rhs) {  
        if (this==&rhs) return *this;  
        return assign(rhs);  
    }  
    virtual  
    U& assign(const B& rhs) {  
        const U* pU=  
            dynamic_cast<const U*>(&rhs);  
        if (!pU) return *this;  
        B::assign(rhs);  
        y = pU->y;  
        return *this;  
    }  
};
```



- öffentliche Vererbung als "ist ein"-Beziehung Folie 12f..
 - Jedes Objekt vom Typ U darf auch für ein Objekt vom Typ B stehen (insbesondere bei Parameterübergabe an Funktionen)
 - Die Unterklasse U erbt die Schnittstelle der Basisklasse und erweitert diese
- private Vererbung als "ist implementiert mit"-Beziehung Folie 21f..
 - U kann nicht überall dort stehen, wo B erwartet wird
 - U erbt jedoch die Implementation der B-Methoden (Verwendung mittels using-Direktive oder Delegation)
- Delegation als "benutzt"-Beziehung Folie 5..
 - modelliert eine Klassenbeziehung, die nicht als öffentliche Vererbung im Sinne von Schnittstellenerweiterung besteht
- Objektzugriff Folie 15ff..
 - Komponentenname wird, beginnend beim aktuellen Objekt, von unten nach oben gesucht
 - Überdeckung aller gleichnamigen Komponenten der in der Hierarchie "darüberliegenden" Klassen, unabhängig von der Signatur
 - Überladung findet nur zwischen den gleichnamigen Methoden statt, die auf der "Ebene" liegen, in der die erste Namensübereinstimmung gefunden wurde
 - "Ebenenübergreifende" Überladung kann mittels der using-Direktive realisiert werden
- Zeiger-/Referenzzugriff Folie 39..
 - ermöglicht generischen Zugriff auf alle Objektinstanzen von B und seiner Unterklassen
 - Komponentenauswahl gemäß statischen Objekttyp Folie 32..
 - Auswahl der Methodenimplementation gemäß dynamischen Objekttyp, wenn die Methode virtuell vereinbart wurde



- gewöhnliche Methoden in B definieren eine *obligatorische Implementation*, die in der Unterklassen nicht überschrieben werden soll
- virtuelle Methoden definieren eine *Schnittstelle und eine Standardimplementation*, die die Unterklasse erbt und überschrieben werden kann Folie 34
- rein virtuelle Methoden definieren *Schnittstelle* (ggf. eine *Teil-Implementation*), die in der Unterklasse implementiert werden *muss* Folie 57
- abstrakte Klassen sind Klassen, die mindestens eine *rein virtuelle Methode* haben. Von ihnen können keine Objekte angelegt werden, sie dienen als Blaupause für davon abgeleitete konkrete Klassen, die die Implementation zur Verfügung stellen.
Abstrakte Klassen sind allgemeine Konzepte, die jede andere (abgeleitete) Klasse der Klassenhierarchie konkretisiert Folie 56
- Interfaces sind Klassen mit ausschließlich *rein virtuellen Methoden* ohne Daten. Sie fungieren als Anforderungen, Schnittstellen oder Eigenschaften für die von ihnen abgeleiteten konkreten Klassen. Mehrfachvererbung von Schnittstellen ist kein Problem Folie 61



- Konstruktoren sind nie virtuell, da sie durch den Klassennamen stets eindeutig sind Folie 24f, 41
- Destruktoren können (als Ausnahme von der Namensregel) und sollten virtuell sein, damit Objekte der Unterklassen korrekt zerstört werden können Folie 24f, 41
- Kopierkonstruktoren sind nach Namensregel nicht virtuell. Eine polymorphe Objektkopie kann durch eine einheitliche virtuelle Methode (z.B. `clone()`) erreicht werden Folie 24f, 42f
- Der Zuweisungsoperator kann nicht vererbt werden, stattdessen überdeckt jeder Zuweisungsoperator der Unterklassen die "darüberliegenden" Versionen. Polymorphie wird ermöglicht durch Aufruf einer einheitlichen virtuellen Methode
virtual `U& assign(const B&)` aus dem Standard-Zuweisungsoperator heraus und Nutzung von **dynamic_cast** zur Typprüfung des Operanden. Folie 26f, 44f

Auf die Klammern kommt es an

```
const int n=10;  
T *p;      // Zeiger auf T-Objekte  
T (*p)[n]; // Zeiger auf ein Feld von n T-Objekten  
T *p[n];   // Feld von n Zeigern auf T-Objekten
```

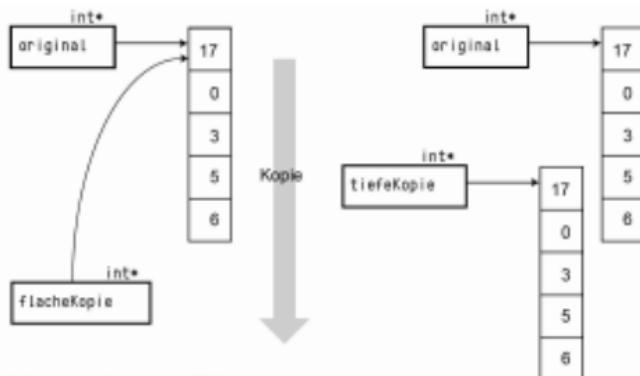
ALGORITHMUS ZUR ENTSCHEIDUNG VON ZEIGERDEKLARATIONEN

- ❶ Beginn ist der Variablenname
 - ❷ Interpretation von links nach rechts
 - ❸ Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
 - ❹ Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
 - ❺ Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert
-
- Felder sind konstante Zeiger auf das erste Feldelement und der Compiler reserviert bei der Vereinbarung automatisch einen zusammenhängenden Speicherbereich für das gesamte Feld
 - Der Zugriff auf Feldelemente erfolgt entweder über den []-Operator oder nach den Regeln der Zeigerarithmetik (und anschließender Dereferenzierung)
 - Die Zeigerarithmetik ist syntaktisch identisch mit der Nutzung von Random-Access Iteratoren für Container
 - Bei der Übergabe von Feldern an Funktionen, ist der Formalparameter i.d.R. ein nicht-konstanter Zeiger vom kompatiblen Grundtyp. Damit kann mittels Zeigerarithmetik an beliebige Stellen im Feld gesprungen werden.
Die Information über die Länge geht dabei verloren
(und muss ggf. als weiterer Parameter übergeben werden)

- Eine Referenz ist ein Zeiger, der an einen existierenden Wert gebunden ist und diese Verbindung ist nicht lösbar
- Zeiger können überall hin zeigen und sind nicht an einen festen Wert gebunden (Iteratorenfunktionalität durch Zeigerarithmetik)
- Zeiger sind flexibler als Referenzen, aber auch fehleranfälliger, da der Programmierer verantwortlich dafür ist, dass die Zeigervariable auf einen definierten Wert zeigt
- Referenzen sind sicherer als Zeiger und syntaktisch angenehmer, da eine Referenz automatisch auf ihren Wert dereferenziert wird (kein Voranstellen von * notwendig)

Referenzen sind Zeigern stets vorzuziehen, wenn die zusätzliche Funktionalität der Zeigern nicht benötigt wird

- Eine **flache Kopie** kopiert sämtliche Datenkomponenten einer Klasse.
Enthält die Klasse Zeiger, so werden lediglich der Zeiger kopiert
- Eine **tiefe Kopie** legt eine echte Inhaltskopie an, d.h. bei Bedarf wird passender Speicher angefordert und auch die Zeigerziele werden kopiert



Quelle: Bekeymann, Der C++ Programmierer, 106-8.2.

Beispiel: flache und tiefe Kopie eines durch einen Zeiger referenzierten Feldes