



Programmierung II

Vorlesungsskript

Mitschrift von Falk-Jonatan Strube

Vorlesung von Dr. Arnold Beck

12. Mai 2016

Inhaltsverzeichnis

1	C++	4
1.1	Ein- und Ausgabe	4
1.2	Defaultargumente	6
1.3	Überladen	6
1.4	Typisierte Konstanten	6
1.5	Referenzen	6
1.6	String0	7
1.7	String	7
1.8	const	7
1.9	Folge	7
1.10	Kopierkonstrukturen	7
1.11	Friend	7
1.12	Vererbung	7
1.13	Virtuelle Funktionen	7

Hinweise

Unterlagen unter:

```
1 cd /home/rex/fi1/nestler/Programmierung_II_2016/
```

Compiler

- Intel i16, i13 (für Linux oder Visual Studio) www.hocomputer.de (kostenpflichtig)
- gcc 5.3, 4.85 gcc.gnu.org

Zugriff auf Windows-Programme (Visual Studio 2013) in Linux-Laboren:

```
1 rdesktop -f its56 # oder its59
```

Empfohlene Literatur: Breymann[1]

1 C++

1.1 Ein- und Ausgabe

(siehe Folie CPP_01_stdio)

```
1 #include <iostream> // alternativ zu <stdio.h> in C
2 using namespace std; // namespace: für bestimmte Abkürzungen (bspw. cout anstatt std::
  cout)
3 // Hinweis: "::" zeigt, dass das davorstehende "static" ist (hier: std)
4
5 class integer {}; // class: Vergleichbar mit typedef
6
7 int main() {
8     integer i0; // i0: Instanz bzw. Objekt der Klasse integer
9     cin.get(); // Eingabe abwarten
10 }
```

(vgl. integer.cpp)

```
1 #include <iostream>
2 using namespace std;
3
4 class integer { // int – Variable in class verpacken
5     // private: // private ist default
6     int i; // this->i bzw. (*this).i
7     // private nur für andere Klassen, andere Instanzen der gleichen Klasse können drauf
  zugreifen
8 public: // wenn nichts steht, wird automatisch das vorherige angenommen. Hier: private
  integer(int i=0):i(i){ // Konstruktor und Defaultkonstruktor
9     // i=0 default Wert, wenn keiner Angegeben
10    // :i(i) übergebenes i wird dem i der Klasse zugewiesen: i_1(i_2) i_1 ist this->i,
11    und i_2 ist das übergebene i
12    cout<<"integer-Objekt i = "<<this->i<<endl;
13  }
14
15  int get(){ return i; }
16
17  void set(int i=0){ this->i = i; }
18
19  // statische Methode: aufrufbar ohne Instanziierung der Klasse
20  static integer add(integer i1, integer i2){ // Wertkopien von i1 und i2
21    // return integer(i1.i + i2.i); // alternativ und explizit: Konstruktor-Aufruf
22    // return erstellt eine Kopie (mit Konstruktor erstellt)!
23    // return i1.get()+i2.get(); // Umwandlung int nach integer, Aufruf Konstruktor
  implizit
24    return i1.i + i2.i;
25    // i1.i Möglich, da innerhalb der Klasse integer und somit privates i sichtbar
26  }
27 };
28
29 auto max(int x, int y) -> int { return x>y ? x : y; } // Lambda-Funktion
30 // auto: Rückgabetyt ergibt sich aus dem Kontext bzw. über das "-> int"
31
32 template<typename Typ1, typename Typ2> // Weiterentwicklung Makro: wählt automatisch Typ
  aus
33 auto quotient(Typ1 a, Typ2 b) -> decltype(a/b) { return a/b; }
```

```

34
35 auto main() -> int {
36     auto k = 0; // C++11: da 0 vom Typ int ist auch k vom Typ int
37     decltype(k) j = 5; // C++11: da k vom Typ int ist auch j vom Typ int
38     char *c = nullptr; //C++ 11: Zeigerliteral
39     int *ip = NULL;
40
41     integer i0(5), i1=4; // 2 (alternative) Initialisierungen von Objekten
42     // i1=4 nur möglich, wenn 1 Parameter gefordert ist.
43     cout<<"i0.i = "<<i0.get()<<endl;
44     // cout im Vergleich zu printf() typsicher.
45     cout<<"i0.i + i0.i = "<<integer::add(i0, i0).get()<<endl; // Aufruf static-Methode add
46     integer i3 = integer::add(i0, i0); // Initialisierung von i3
47     cout<<"i3.i = "<<i3.get()<<endl;
48     i0.set(22);
49     cout<<"i0.i = "<<i0.get()<<endl;
50     cout<<"max(3,5) = "<<max(3,5)<<endl;
51     cout<<"5/3 = "<<quotient(5, 3)<<endl;
52     cout<<"5.0/3.0 = "<<quotient(5.0, 3.0)<<endl;
53     cout<<"b / 1 = "<<quotient('b', '1')<<endl;
54     cin.get();
55 }

```

(vgl iostream.pdf)

- nach jeder cin Eingabe: „cin.clear();“, damit Fehler ignoriert werden um weiter cin's abhandeln zu können (vgl. robust_ea)

Einlesen:

```

1 char sc;
2 cout << "sc=";
3 cin >> sc;
4 cin.clear(); // clear, um die Eingabezeile freizumachen, damit man nicht an Falscher
    Eingabe hängen bleibt
5 cin.ignore(INT_MAX, '\n'); // braucht #include <limits.h>
6 cout << "sc" << dec << (int) sc << endl;
7
8 // alternativ:
9 char vb[128];
10 cout << "s=";
11 cin.getline(vb, sizeof(vb), '\n'); // lesen als String, dann wieder umwandeln (liest
    auch Leerzeichen ein)
12 sc = atoi(vb); // braucht #include <cstdlib>
13
14 // alternative zu getline:
15 cin.get(...); // lässt aber \n im Strom
16 cin.get();
17
18 // alternativ
19 cin >> setw(sizeof(vb)) >> vb; // verhindert Überlauf
20 sc = atoi(vb);
21
22 // alternativ
23 String s; // braucht #include <string>
24 size_t ie=0;
25 cin >> s;
26 unsigned int ni = stoi(s, &ie, 10);
27
28 // alternativ
29 getline(cin, s, '\n');
30 double d = stod(s, &ie);
31

```

```
32 // zum compilieren: g++ p2a1.cpp -std=c++11 -o a.out
```

robust_ea1.cpp:

```
1 do {
2     cout<<"d = "; cin>>d;    // einlesen
3     if (cin.eof()) break;    // break bei Strg+D oä.
4     if (cin.fail() || (cin.peek() != '\n')){ // ist nächstes Zeichen ungültig?
5         cin.clear(); cin.ignore(INT_MAX, '\n'); // Strom zurücksetzen und zum \n gehen
6         continue;
7     }
8     break; % Schleife verlassen, wenn korrekte Eingabe
9 } while(true);
10
11 if (cin.eof()){ cin.clear(); cout<<"eof\n"; }
12 else {
13     cin.clear(); cin.ignore(INT_MAX, '\n');
14     cout<<"Wert d = "<<d<<endl;
15 }
16 cin.ignore();
```

1.2 Defaultargumente

Defaultargumente müssen immer von rechts angefangen definiert sein:

```
1 myFunc(int i = 5, int j = 7) // korrekt
2 myFunc(int i, int j = 7)    // korrekt
3 myFunc(int i = 5, int j)    // falsch !!!
```

1.3 Überladen

overload.pdf

Hinweis: cast auf zwei Möglichkeiten:

```
1 int i = 5;
2 double d;
3 d = (long) i;
4 d = long(i);
```

1.4 Typisierte Konstanten

1.5 Referenzen

referenzen.pdf

Ein Speicherplatz wird mit mehreren Variablen-Namen beschrieben.

1.6 String0

1.7 String

1.8 const

const_mutable.pdf

Faustregel: Alle Funktionen, die nichts ändern, immer das const anfügen.

1.9 Folge

Initfolge/Initfolge.pdf

1.10 Kopierkonstrukturen

Kopierkonstrukturen.pdf

1.11 Friend

Zugriff auf private Klassen von außerhalb:

- Funktionen
- Alle Methoden anderer Klassen
- spezifische Methoden anderer Klassen

Muss von Klasse mit privater Funktion festgelegt werden. Freundschaft wird nicht „erwidert“, ist nicht reflexiv (bsp. friendreflex).

1.12 Vererbung

vererben.pdf

1.13 Virtuelle Funktionen

- überschriebene Funktionen
- virtuelle Funktionen \Rightarrow Späte Bindung
- rein virtuelle Funktionen
eine Klasse mit rein virtuellen Funktionen ist abstrakt
- VMT virtual method table ABB 124

besch1.cpp (Beschäftigte)

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Besch{
6     private:
7         string name;
8     public:
9         Besch(string name): name(name){} // :name(name) --> Member-Initialisierer
10        string getName() {return name;}
11        void setName(string name) {this->name = name;}
12        virtual void display(ostream& os) {os<<name;} // virtuelle Funktion
13        virtual double calc(){return 0.0;} // normalerweise Geld nicht mit double, sondern
14        // ganzzahlig rechnen!
15        // wenn Funktion eigentlich nicht benötigt wird: Polymorphes Interface:
16        // virtual double calc() = 0; // rein virtuelle Funktion
17        // diese rein virtuelle Funktion macht den Datentyp Besch abstract! D.h. davon kann
18        // keine Instanz erzeugt werden (wie in der Main passiert)!
19};
20ostream& operator<<(ostream& os, Besch& b) {b.display(os);}
21
22class Arbeiter:public Besch{ // public: durchsichtige Vererbung
23    private:
24        int stunden;
25        double stdLohn;
26    public:
27        Arbeiter(string name, double stdLohn):Besch(name){ // :Besch(name) --> Basis-
28            // Initialisierer
29            this->stdLohn = stdLohn;
30        }
31        int getStunden() { return stunden; }
32        void setStunden(int stunden) { this->stunden = stunden; } // usw. getter und setter
33        void display(ostream& os){ Besch::display(); os<<" "<<stdLohn<<" "<<stunden<<" "<<calc
34            ();}
35        double calc(){ return stdLohn * stunden; }
36};
37ostream& operator<<(ostream& os, Arbeiter& a) { a.display(os); }
38
39class Angest:public Besch{
40    private:
41        double gehalt;
42    public:
43        Angest(string name, double stdLohn):Besch(name){
44            this->gehalt = gehalt;
45        }
46        void display(ostream& os){ Besch::display()<<" "; os<<gehalt<<" ";}
47        double calc(){ return gehalt; }
48};
49ostream& operator<<(ostream& os, Angest& a) { a.display(os); }
50
51class Haendler:public Arbeiter{
52    private:
53        double prov;
54        double ums;
55    public:
56        Haendler(string name, double stdLohn, double prov):Arbeiter(name, stdLohn){this->prov
57            =prov;};
58        void setUms(double ums){this->ums = ums;}
59        double calc(){return Arbeiter::calc() ++ ums*prov;}
60        void display(ostream &os){Arbeiter::display(os);}
61}
62

```



```

58 int main() {
59     Besch* b1 = new Besch("Hans Huckebein");
60     b1->display(cout); cout<< " Euro"<<endl;
61     cout<< (*b1) << " Euro" << endl;
62     cout<< "====="<<endl;
63     Arbeiter* b2 = new Arbeiter("Moriz Lehmann", 8.99);
64     b2->setStunden(140);
65     b2->display(cout); cout<< " Euro"<<endl;
66     cout<< (*b2) << " Euro"<< endl;
67     cout<< "====="<<endl;
68     Angst* b3 = new Arbeiter("Friedrich Lempel", 3099.00);
69     b3->display(cout); cout<< "Euro"<<endl;
70     cout<< (*b3) << " Euro"<< endl;
71     cout<< "====="<<endl;
72     Haendler* b4 = new Handler("Bang Ohlufson", 15.80, 0.1);
73     b4->setStunden(350);
74     b4->setUms(12000);
75     b4->display(cout); cout<< "Euro"<<endl;
76     cout<< (*b3) << " Euro"<< endl;
77     cout<< "====="<<endl;
78
79     Besch* be[] = {b1,b2,b3};
80     for (int i=0; i<3; i++){
81         cout<<i<<": ";
82         be[i]->display(cout); cout<<endl; // gibt 0 für alle zurück, da nur calc() (und
            display()) vom Typ Besch ausgeführt, da das Array aus Besch besteht! Die Funktion
            calc() wurde überschrieben (Vgl. Überladen, wenn bei gleichen Namen verschiedene
            Paramater angegeben sind)!
83         // Pointer im Array bestimmt die ausgeführten Funktionen!
84         // Lösung: "virtual" als Schlüsselwort vor Funktion (s.o.). Angeben bei erstem
            Auftauchen in Vererbungshierarchie
85         cout<<(*be[i])<<endl<<"++++++"<<endl; // geht auch, da im Operator (die
            ja nicht virtuell gemacht werden kann, weil sie kein Member ist) die virtuelle
            display-Fkt aufgerufen wird.
86     }
87
88     return 0;
89 }

```

Vererbungs Hierarchie:

ABB 125

Potentielle Mehrfachvererbung (bei Manager):

- Hat mehrfache Namen (wegen unterschiedlichen Vererbungslinien)

Problemlösung: Arbeiter, Angst und Händler von virtual Besch erben lassen (dort, wo es sich aufzweigt):

```

1 class Arbeiter: virtual public Besch {};

```

Hat immer noch Einschränkungen!

Literatur

- [1] Ulrich Breymann und Ulrich Breymann. „Der C++ Programmierer“. In: *C++ lernen, professionell anwenden, Lösun* (2009).