Skript PVK Info Mo

1. Einführung
- Schwierige Prüfung
- PVK: Wiederholung von Schwerem, schwierigen Stoff. Keine vollständige Wiederholung der Vorlesung
- Auch böse Fallen, und Dinge, die man sonst vergisst.

1. Datentypen

int	mindestens 16 bit lang
unsigned int	mit & ohne Vorzeichen
char	mindestens 8 bit lang
unsigned char	mit & ohne Vorzeichen
long long int	mindestens 32 bit lang
float double	Fliesskommazahlen
bool	spezieller char, der alles, was nicht null ist, zu eins macht.

Beispielprogramm zu Bools laufen lassen.

Wie Benennen: A-Z, äöü, a-z, ß A 0-9, _ (Umlaute)

Nie am Anfang

Groß- und Kleinschreibung ist wichtig

2. Implizite und Explizite Konvertierung.

Implizite Konvertierung bei Zuordnung:

```
int i;
i = 1.4   (i == 1)
i = 1234.56  (i = 1234)
```

~~Alles~~ Fliesskomma zu integer: Kommas werden abgeschnitten.
Integer zu Fliesskomma: Intuitiv.

Bei Auswertung ~~a + b~~ a + b
wilder Typ hat das

char < int < unsigned int < float < double

~~Anderes~~ Programm: Hässliche Konvertierung. (uint_convert.cpp)

3. Darstellung von Info im Speicher:
8 bits = 1 byte unsigned char

1 0 0 0 1 0 1 0
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

$\sum 2^n$ · ~~Wh~~ bit$_n$

8
128 + ~~0~~ + 2 = 138

10010 | 14
−0100 | 2 · 4 : 18 · 4
1110 | = 14

−~~1000~~

−~~1000~~
1

Algorithmen zum Rechnen

1. Gerade oder Ungerade? Gerade: 0 Ungerade: 1, abziehen
2. Durch 2 teilen.

Bsp:

138	0
69	10
34	010
17	1010
8	01010
4	001010
2	0001010
1	10001010

signed char

1 000 10 10

+/− A

~~Positive Zahl~~ Nichtnegative Zahl (≥ 0)

0 | A | $\rightsquigarrow$ A

Negative Zahl: (< 0)

1 | A | $\rightsquigarrow -(2^7 - A)$

Integer addition:

$$\begin{array}{r} 11101011 \\ +\ 01000000 \\ \hline 100101011 \end{array}$$

→ Integer Overflow

Bild Wikipedia Integer Overflow.

Aufgabe: Warum macht

0 A | A
1 A | (2^7 - A)

Sinn?

Antwort: $-1 = -(2^7 - (2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0))$

$= (2^7 - \left(\frac{1-2^7}{1-2}\right)) = (2^7 + 1 - 2^7) = -1$

```
  1 1 1111111     -1
+ 0 0 0000001     +1
  ↑ ↑ ↑↑↑↑↑↑
= 1 0 0000000     =0
```

Funktioniert heute vielleicht anders, ~~aber grundsätzlich noch immer~~ sollte grundsätzlich aber, dass alle Programme noch funktionieren.

```
4
00000100
11111011
11111100
```

Algorithmus: - dezimal: Minus ignorieren, Umwandeln. Dann Alle stellen invertieren, 1 addieren

Integer ~~divisie~~ division

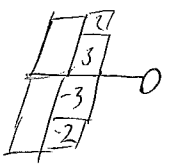
$5/3 \quad 5 = 1 \cdot 3 + 2 \quad 5/3 = 1 \cdot 3/3 + 2/3 = 1$

$A/B \quad A = nB + r, \quad n \in \mathbb{N}_0, \quad n < B, \quad n \in \mathbb{N}_0$

$\Rightarrow A/B = n.$

Modulo: $A \,\%\, B = r, \quad A, B > 0$

$5 \,\%\, 3 = 2$
$-5 \,\%\, 3 = -2$ } Symmetrisch um Null



$5 \,\%\, -3 = 2$
$-5 \,\%\, -3 = -2$ Minus wird zu plus. Effektiv ignoriert.

Modulo-Programm Programm: modulo.cpp

L- Und R- Werte

Mo

L- Wert: Kann man etwas zuordnen. Macht Sinn links von =
R- Wert: macht keinen Sinn

```
int a, b;
a = 5
a = b
(a = b) = c
```

L-Wert kann auch rechts stehen

Man kann auch (a=b) etwas zuordnen, weil (a=b) L-Wert A zurückgibt

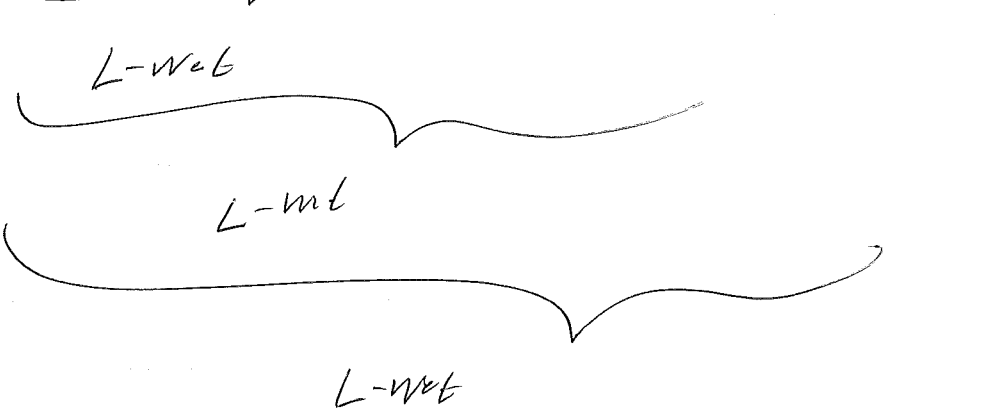
5 = a ??? Nein, das ergibt keinen Sinn.

Programm LRwerte.cpp

Wichtige Ausnahme

```
std::cout << "text" << a << "blabla"
```

std::cout: L-Wert
"text": R-Wert
a: L-Wert
"blabla": R-Wert.



Fliesskommazahlen $F(\beta, p, e_{min}, e_{max})$

$2 \leq \beta \in \mathbb{N}$ Basis

$1 \leq p \in \mathbb{N}$ Präzision

e_{min}, e_{max} minimaler und maximaler Exponent

$$s \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e \quad , \quad \pm \underbrace{d_0 . d_1 \dots d_{p-1}} \cdot \beta^e$$

Muss man ins Dezimalsystem konvertieren, um zu verstehen, was das heisst.

$d_0 . d_1 \dots d_{p-1}$

$\beta^0 \; \beta^{-1} \quad \beta^{-(p-1)}$

$\sum_{i=0}^{p-1} d_i \beta^{-i}$

IEEE754:

$F(2, 24, -126, 127)$ | Wichtig: Binärsystem

$F(2, 53, -1022, 1023)$ | Rest meist so wichtig

Algorithmus zum Berechnen: 1. Alles vor . separat berechnen

2. 0. schreiben

3. mit 2 mult.

4. 1 vorne { Ja: 1 abziehen, 1 schreiben / Nein: Weiter

0.625	0.
1.25	0.1
0.5	0.10
1.0	0.101

138.625

↓

138 | 0.625

10001010 | 0.101

↓ ↓

10001010.101

Auslöschung ≠: Endliche Präzision!

Zahlen nicht exakt:

0.1	0.0	0.4	0.0
0.2	0.00	0.8	0.00
0.4	0.000	1.6	0.001
0.8	0.0000	1.2	0.0011
1.6	0.00001	0.4	$0.00\overline{11}$
1.2	0.000011		
0.4			

$0.0\overline{0011}$

Mo (3.5) Einschub:

~~Floats~~ ~~Wenn~~ ~~A. Weil rundungsfehler~~

Regeln bei Fliesskommazahlen

- Keine Vergleiche (==, !=) mit fliesskommazahlen, weil Rundungsfehler wenn periodische Darstellung.

~~A Addie~~ (0.11) - 1.11 - 0.01 == 1.1 ist manchmal false

- Bei Addition von sehr grossen und sehr kleinen Zahlen kann es sein, dass kleine Zahlen ~~verloren~~ nicht eingerechnet werden.

(~~Hebt ihn in Arm~~

(Widerlager für Normeik, etwa Reihen)

Mo (4)

Operatoren:
Liste Zeigen

~~Auslöschung:~~ ~~1. ungerundete gerundete Zahlen Weil~~
~~2. A. Keine vergleiche bei Fließkommazahlen~~
~~3.~~

Für uns wichtig:

Post/ Prä-Inkrement/dekrement	x++ ++x, x-- --x
Prä Inkrement/ Dekrement	++x, --x
Zeichen	+x, -x
Logisches nicht	!x
Rechnen: Punkt	* / %
Strich	+ -
Gleichungen: Ungleich	< <= > >=
Gleich	== !=
Boolsche Operationen: Logisches Und	&&
Logisches Oder	||
Zuordnung	=, *= etc.

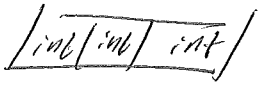
const, const-correctness:

const int i = 5;

- Ein konstanter typ lässt sich nach initialisierung nicht mehr verändern.
- Ein programm ist const-korrekt, wenn jede variable, die nicht verändert wird, const gesetzt wird.
- const int i; ohne zahl macht keinen Sinn, weil nicht mehr veränderbar => für immer undefiniert Compiler Motzt.

Arrays: Mehrere int nebeneinander im Speicher

int i[3]



Möglichkeiten: Beispielprogramm array1.cpp

Mehr Zugriff auf Arrays kommt beim nächsten mal.

unsigned int u=5;
~~int i=10;~~
int i=-10;

~~u~~ u-(--i)--

Erinnerung int < unsigned int

Was passiert hier?

~~u~~ i<0, i=-(2^{n-1}-A)
i=-2^{n-1}+A, A=i+2^{n-1}

(unsigned int)(i) ⇝ 2^{n-1}+A

u-i = u-(2^{n-1}+A) = u-(2^{n-1}+i+2^{n-1})
= u-(2·2^{n-1}+i) = u-i
(2·2^{n-1} overflow = 0)

Fazit: Auch wenn ~~int~~ int < unsigned int
funktioniert u-i, für i<0 wie erwartet.
(u+i aber nicht unbedingt).

Di Skript: ⊙

Kurzschlusslogik

Wenn man einen Ausdruck der Form (A und B) hat, und man weiss, dass A falsch ist, so braucht man gar nichts über B zu wissen. (A und B) ist nicht wahr, egal, ob B wahr oder falsch ist.
Dasselbe gilt für (A oder B). Wenn wir wissen, dass A wahr ist, dann brauchen wir nichts über B zu wissen. Die Aussage ist sowieso wahr.

Ctt Trägt dem mit der sogenannten Kurzschlusslogik Rechnung:
Wie wird (A) && (B), (A) || (B) evaluiert?

1. A wird evaluiert
2. Wenn A && ist, so gib false
   Wenn A || ist, so gib true
3. Evaluiere B
4. Wenn B true ist, gib true zurück, sonst false

(false && B), (true || B)

Wichtig, falls ~~&&~~ A (&& false) (|| true), ist, so wird B nicht evaluiert. Das ist dann wichtig, wenn in B etwas verändert wird oder eine potentiell unerlaubte Rechnung durchgeführt wird: int a; int b

(a == 0) || (b/a)

Teilung durch 0 findet nicht statt, da wenn a == 0, linke Seite true, rechte Seite wird nicht evaluiert.

(b) && (++a):

Wenn b == 0, dann ist linke Seite false: A wird nicht inkrementiert.

Wird in 75% der Prüfungen geprüft.

PVK PI SKRIPT ①

Arrays Zugriff mittels [] - Operator

```
int a1[5] = {1,2,3,4,5};
```

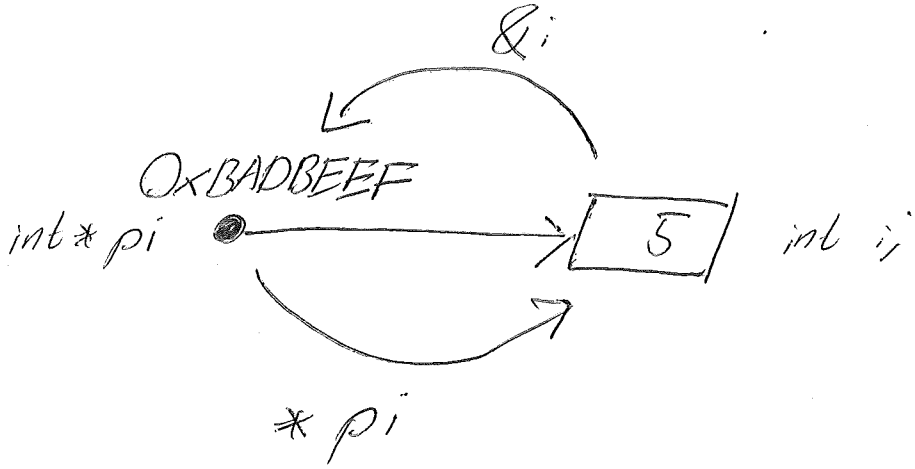
a1[2] = 3 (dritte Element, man beginnt bei 0 zu zählen) (Achtung Fehler-anfällig).

array2.cpp

Was passiert, wenn man auf einen Ort zugreift, bei dem noch nichts zugeordnet ist?

array3.cpp Segfault

1. Wenn nichts zugeordnet ist: Irgendwas, was halt dort liegt.
2. Wenn etwas zugeordnet ist: Betriebssystem schmeisst Programm ab



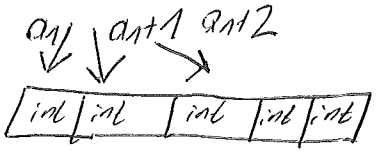
pointer1.cpp

Adresse ist immer etwas anderes.

Pointer und Arrays sind (beinahe) dasselbe. Für euch sind sie ganz dasselbe.

Ein Array ist ein Pointer auf den Beginn des Arrays mit der garantie, dass die Adresse array+n auf das (n+1)-te Element zeigt

```
int a1[5]
```



Übergabe von ranges: Wenn ihr Arrays ~~übergeben~~ einer Funktion übergeben möchtet, dann übergibt man i.a. 2 pointer.

- Ein pointer auf das erste Element
- Ein pointer <u>nach</u> dem letzten element.

Illustration Bild

Leere Range: begin == end

pointer3.cpp

References: - Koppelt Werte aneinander
- Kann nur bei Deklaration gesetzt werden
- Sehr nützlich bei Funktionen

Frage: Wie funktioniert das? Antwort: Braucht uns nicht zu interessieren!

references.cpp, references2.cpp.

Skript Di (2)

wmk-15/16

Aufgabe: Prüfung Aufgabe 8: Argumententypen
Ihr sollt nun in der Lage sein, diese Aufgabe zu lösen.
Lösen der Aufgabe mit Vorstellung.

Scopes und namespaces:

Sichtbarkeit von variabeln lässt sich mit geschweiften Klammern einschränken:
- Funktionen (auch main-Funktion)
- Schleifen (insbesondere For-Schleife)
- "Wilde" Klammern.

Faustregeln:
- Variabeln in Klammern werden gelöscht, wenn man die Klammer verlässt. (Gilt auch, wenn Schleife wiederholt wird => spöter)
- Wenn man auf eine variable zugreift, so bekommt man immer diejenige, die am weitesten innen deklariert wurde.

Programm: scopes.cpp

Diskutieren: Was passiert hier (5 min)

Namespaces: Häufig arbeiten viele Leute an einem Projekt. Namen werden manchmal mehrfach verwendet:

Wie wird das angestellt: scope resolution ::

Alle Objekte und Funktionen der Standardbibliothek befinden sich in std:: namespace

~~Namespaces klein~~

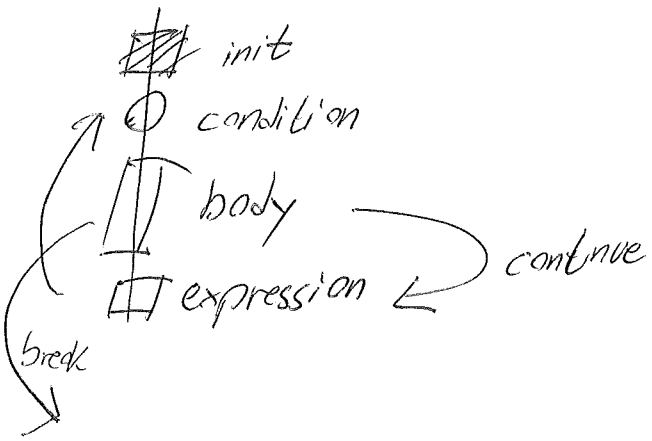
Programm: namespaces.cpp

using namespace NAME exportiert auch namespace::, aber kann Probleme machen.

Pause & Fragen

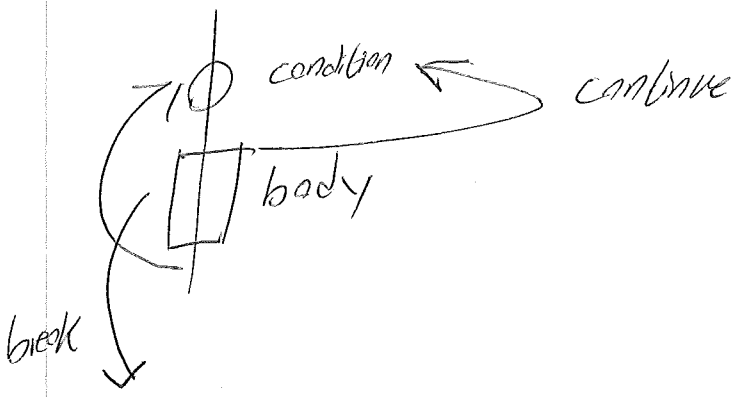
Loops

~~for~~ for(init; condition; expression)

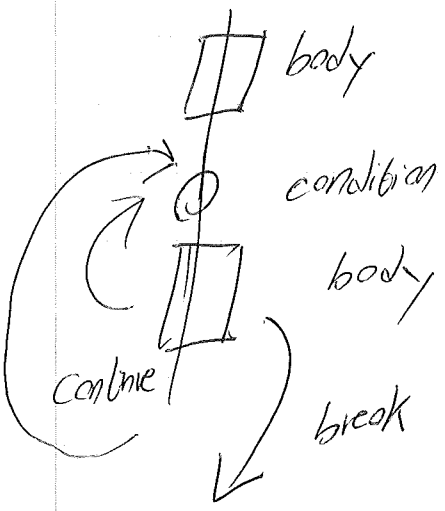


3)

while (~~expression~~ condition)



do – while



loop1, loop2, loop3. cpp

Rekursion & Iteration:

Sehr einfach: Funktion ruft sich selbst auf.
- Vor allem mit Aufgaben trainieren.
- Selbst Ideen für Rekursive Funktionen haben.

Winter 2015/2016 @ 3. a)/b)/c)

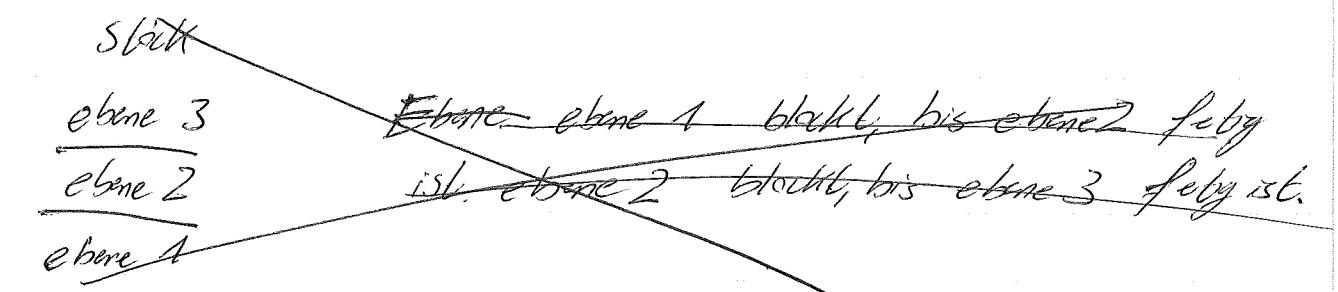
PVK Informatik Skript, M. ⑩

9 – Stacks bei Funktionenaufrufen:

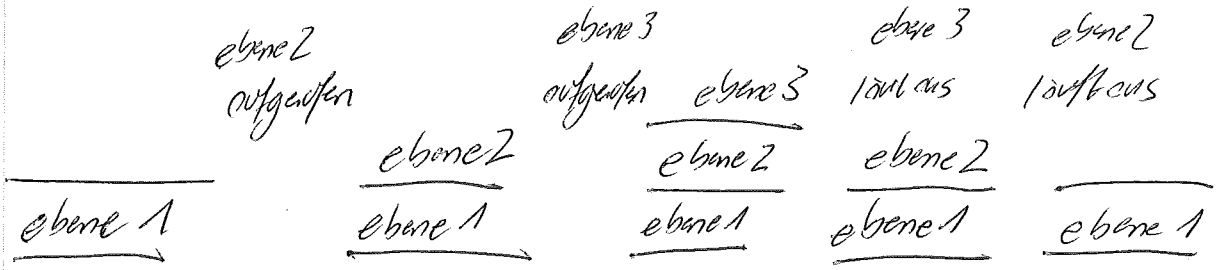
Wenn eine Funktion in ihrem Körper eine andere Funktion aufruft, dann blockiert die ursprüngliche Funktion, bis die neue Funktion abgeschlossen ist.

```
void ebene1(...) { ... ebene2(...); ... }

void ebene2(...) { ... ebene3(...); }
```



~~ebene 1 ruft ebene 2 auf. Ebene 2 ruft ebene 3 auf.~~
~~ebene 3 läuft aus. Dann macht ebene 2 weiter. Ebene 2~~



Funktionsaufrufe:

a) Einzelne Argumente

- Ich will das, was ich der Funktion übergebe, in der Funktion verändern.

typ funktionsname (typ & argument)

Wenn man ~~Argument verändert~~, argument in der Funktion verändert verändert sich auch das, was man übegeben hat.

- Ich will das, was ich der Funktion übegebe, in der Funktion sicher nicht übegeben:

typ funktionsname (typ argument)

Das, was wir übergeben, wird in argument kopiert. ~~Am ende~~ Ende der Funktion wird argument gelöscht. ~~Das, was~~ Wir können argument in der Funktion verändern, das verändert aber nichts an dem, was wir übegeben haben.

typ funktionsname (const typ & argument);

Wir übegeben eine Referenz. Es wird nichts kopiert, wenn man versucht argument innerhalb der Funktion zu verändern, meckert der compiler.

(01) M.

b) ~~Mehrere Argu~~ Ein Array:

typ funktionsname(typ * begin, typ * end)

Man übergibt ~~den Point~~ einen pointer an den Anfang des Arrays und einen past-the-end Pointer. Man kann über die Pointer die Werte des Arrays, das man übergeben hat, verändern. Wenn man an den Welen des Arrays nichts verändern will, so schreibt man

~~typ funktionsname(typ * begin, typ * end~~
typ funktionsname(~~typ~~ const typ * begin,
const typ * end)

Dann meldet der Compiler, wenn man versucht, etwas am Array zu verändern. Achtung: Im Gegensatz zu einzelnen Argumenten kann man nicht bereits im Funktionskopf sagen, dass man eine Kopie ~~machte. Das muss~~ des Arrays haben möchte. Das muss man im Funktionsrumpf machen.

Skript PVK Informatik Mi (1)

Klassen und Structs

Structs: Sind historisch erwachsen aus der Idee, mehrere Typen zusammen zu fassen.

class 1.cpp:

Beispiel: Wir definieren eine class point. Zugriff auf das, was drin ist: mit . - Operator.

Klassen: Erwachsen aus structs, mit der Idee, dass man gerne auch Funktionen, operatoren ect. haben möchte. Wichtige Fehlerquelle: Strichpunkt hinter Klassendefinition.

~~Funktionen in~~

Funktionen in einer Klasse (member ~~F~~unctions) heissen Methoden.

Schreibweise: class:: method(type argument 1, ...), :: wie bei namespaces.

In dieser Vorlesung nennt man sie aber member functions.

~~Unterschied zwischen~~

public/private: In Klassen gibt es Teile, die man von Aussen sehen soll, und Teile, die man nicht sehen soll.

Philosophie: Das was man sieht ist die Schnittstelle, sollte immer gleich bleiben. Das was dahinter ist ist die Implementierung, kann/soll man verändern/verbessern oder anpassen.

Unterschied zwischen struct und class: struct ist default public; class private. Sonst kein Unterschied.

Beispiel: class2.cpp

Zugriff auf ~~Private Info~~ private members:
~~Falls wir eine Funktion in einer Klasse haben und wir haben ein Argument, das vom selben Typ der Klasse ist, in der die~~, die ein Argument mit demselben ~~Typ hat, so kann~~
Von einer Methode die ein Argument mit demselben Typ wie die Klasse selbst hat, kann auf private members zugegriffen werden.

~~Beispiel: class3.cpp~~
*this: In jeder Methode gibt es einen Pointer *this, der einfach da ist. Er zeigt jeweils auf ~~die~~ das Objekt, das die Methode aufgerufen hat.

(*this).methode <=> this->methode

Insbesondere weil *this.methode
*(this.methode) geklammert wird

(C++ Precedence zeigen)

Beispiel: class3.cpp

Operatoren überladen:

Für diverse Klassen machen Ausdrücke wie
v1 + v2, v1 * s, Sinn. (Beispiel vektor)

Hier kann man operatoren überladen:

type operator=(argumente)

Normalerweise gibt man etwas Sinnvolles zurück:

politician& operator=(politician& RHS)
politician& operator=(politician& RHS)

1. Wir nehmen eine ~~Referenz~~ const referenz als Argument. Damit kann man das Argument zwar nicht verändern, aber man muss das Objekt nicht kopieren (call by reference)
2. Wir geben eine ~~eigene~~ referenz zu *this zurück, weil wir das erwarten. Wir wollen ~~1. eine Lvalue~~
1. eine Lvalue (++(a=b) soll a um 1 erhöhen)
2. Das, was ~~rechts~~ links vom = steht.

Allgemein:
Postfix & Suffix: Rvalue
Zuweisung: Lvalue, links der Zuweisung
Vergleich ==, >=: bool, Rvalue
<<, >> für Streams: std::ostream&

Konstruktor, Destruktor:

Konstruktor: Methode, die ~~ausgeführt~~ bei Deklaration aufgerufen wird.

Es gibt ~~drei~~ zwei wichtige Konstruktoren:

1. default constructor politician(), wenn ohne Argumente aufgerufen

2. copy constructor politician(politician &), wenn mit ~~anderem~~ ~~Ding~~ eigener Klasse ~~des~~ als Argument aufgerufen

Diese sind sehr wichtig!
1. default Konstruktor wird benutzt, wenn man eine Klasse ohne Argumente, oder sogar ganz ohne Klammern deklariert.

politician ann;
politician ann();

Wenn man ein Array der Klasse definiert

politician kabinett[80];

~~Wenn~~ Wenn man keinen default-Konstruktor hat, so baut der compiler einen. Dabei
- lässt er die grundlegenden types undefiniert (wie int;)
- ruft er die default-Konstruktoren der Klassen auf (bei uns etwa die Klasse der standard bibliothek std::string)

Das ist nicht immer was man will!

Mi (3)

copy constructor:

Wenn man eine Funktion
function(politician ~~long~~ argumentName)

aufruft so ~~greift man nicht auf~~ wird argumentName kopiert.
Dies geschieht mit dem copy constructor.

Insbesondere hat der copy Copy constructor die Form

politician(~~& politician~~ politician& Argument).

Wir müssen eine Referenz (call by reference) übergeben, denn falls wir das nicht machen würden, müsste man Argument kopieren, was der Compiler nicht kann, weil wir gerade bei dabei sind, zu sagen, wie ~~man etwas kopiert~~. das geht.

Basi
Falls kein copy constructor da ist, macht der compiler einen, der ~~alles kopiert, was~~ einfach alle Mitglieder kopiert.

1. Die basic types werden einfach kopiert (int i = j;)
2. Es werden copy constructors von Klassen aufgerufen, falls wir member dieser Klassen haben
3. Das ist auch nicht unbedingt was wir wollen.

Desbuktor ~Klasse()
~polticion()

- Hat keine Argumente,
- Wird aufgerufen, wenn Objekt gelöscht wird.
  - Läuft out of scope
  - Wird mit delete gelöscht. (Freitag)

Ist dazu da, aufzuräumen: Wenn kein desbuktor da ist definiert der compiler einen.
- basic types werden einfach freigegeben.
- für Klassen werden Desbuktoren aufgerufen.

Das ist auch nicht immer, was wir wollen.

Konsbuktor mit einem einzigen Argument, wie nicht wie oben, werden für implizite Konversionen benutzt

programm class6.cpp als Beispiel

Erklären, was passiert:

~~Zu [illegible] stat~~

std::ostream & operator<<(std::ostream & o, politicion RHS)

<< - Operator für Output. Braucht genau diese Argumente in dieser Reihenfolge.
std::ostream ist cout, welches typ std::ostream hat.
- Referenz weil Kopieren nicht möglich
- Referenz wird zurückgegeben weil erwartet.

friend functions

"Friend functions are like friends with benefits, they are allowed to touch your private parts!"

~~Wenn es nicht~~
Wenn man einen Operator als methode implementiert, so ist *this immer auf der LHS des operators.
~~Falls~~ Falls wir das nicht möchten, aber dennoch auf die private members zugreifen möchten, dann:

- deklariert man die Funktion im Block als Friend
- deklariert man den ~~function~~ function body ausserhalb der Klasse.

Beispiel: ~~operator~~ class6.cpp.

Aufgabe: Zu zweit Klasse implementieren:

(2x2)
Symmetrische reelle Matrizen haben 3 Freiheitsgrade

$\begin{pmatrix} a & c \\ c & b \end{pmatrix}$, sind ~~Gruppe~~ $\mathbb{R}$-Algebra unter Addition und Skalarmultiplikation.

Implementiert Klasse
- Copy constructor, ~~copy dest~~ default constructor
- Destructor ~~destructor~~ nicht nötig
- Ausgabe in sinnvoller Form.
- Zuweisungsoperator =
- 1 weiterer sinnvoller Konstruktor
- Addition, Subtraktion, Skalarmultiplikation

finals

~~Matrixmultiplikation~~

Skript PVK Informatik Do (1)

Heute wieder deutlich theoretischer:

Wir beginnen mit Lindenmeyer-Systemen und BNF:

Beides sehr, sehr einfache Prinzipien:

Lindenmeyer-Systeme: Alphabet Σ

Alle endlichen Wörter Σ^*

Produktionsvorschrift $p: \Sigma \to \Sigma^*$

Startwort s^*

$\mathcal{L}(\Sigma, p, s^*)$ ist ein Lindenmeyer-System.

$\Sigma = (A, B, C)$

$p: A \mapsto B$

$B \mapsto C$

$C \mapsto AB$

$s^* = A$

A, B, C, AB, BC, CAB, ABBC, BCCAB, CABABBC

BNF: Σ, Σ^*

factor = unsigned_number | "(" expression ")" | "-" factor

term = factor | factor "*" term | factor "/" term

expression = term | term "+" expression | term "-" expression

~~Term~~ Nichtterminalsymbole: Symbole, die man aufspalten kann, d.h. die ~~links~~ oben definiert sind

Terminalsymbole: Dinge, die nicht weiter definiert sind.

Plenum: Was sind hier nichtterminalsymbole, was sind terminalsymbole?

Idee der BNF: Möglichkeit, Rekursion zu definieren

-1+5*4

factor expression

+

term expression

factor factor term

factor term

* (-1) * factor term

1 factor term

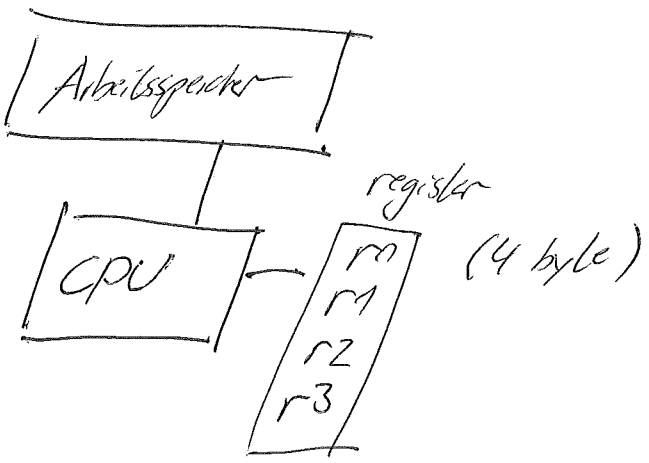
5 factor

4

An Prüfung: Meistens recht zu verstehen und eher heuristisch. Keine schweren Aufgaben, aber dennoch: üben.

Übungen werden wir Ende der Stunde machen.

Assembler



Do (2)

~~Aufgabe~~ Freiwillige Programmierübung: Assembler-Kommandos und was sie bewirken.

Ausdruck der jeweiligen Seiten verteilen.

~~Bisher war es so:~~

~~Es geht darum, ein~~ Ich möchte euch gerne beibringen, wie ein ~~Assembler~~ Compiler mit Assembler rechnet, damit ihr ein Gefühl ~~für~~ dafür, wie diese Simulierte Computer funktioniert, bekommt. Ich möchte jetzt daher 20 minuten lang mit euch die Aufgabe 3, Serie Donnerstag ausüben. Zweiergruppen, versuchen Aufgabe zu lösen,

Herunterladen der Programme, kompilieren und laufen lassen.
Aus gedruckte Seite Da Austeilen

~~Ansatz~~

~~Wir haben bereits~~

char-Manipulation:

```
#include <iostream>
```

std::istream ~~cin~~ std::cin

std::ostream ~~cout (, cerr)~~ std::cout(std::cerr)

Über cout braucht ihr nicht viel mehr zu wissen.

~~" " ' '~~

```
#include <string>
```

std::string: Fortgeschrittene Klasse für strings

C++-Reference öffnen.

Aber: Für Prüfung nicht so wichtig, weil zu kompliziert.

std::isbeam std::cin

operator >> : Formatierter Input
Liest ints direkt in ints ein.
~~Liest~~ Kann gut mit std::strings arbeiten
Liest auch einzelne chars ein.

Was und wie viel eingelesen wird hängt vom Argument rechts ab. Verschiedene overloads.

operator >> (std::istream is, int in)
operator >> (std::istream is, char c)
operator >> (std::istream is, std::string st)

~~ein Bsp~~ Programm char3.cpp
Ist sehr intuitiv, es ist nicht davon auszugehen, dass das an der Prüfung intensiv getestet wird.

Konvertierung (implizit oder explizit) bool (& std::cin)

→ Ist false wenn ein Einleseversuch gescheitert ist.
→ Ist sonst true.

Programme, rätseln. char1.cpp, char2.cpp

Hier: Erinnerung, dass bools mit 0 oder 1 ausgegeben werden

Do (3)

~~Zahlensysteme: Es gibt jetzt neu~~

Zahlensysteme: Wir kennen das Dezimalsystem und das Binärsystem. Neu: Hexadezimal (16er) und Oktalsystem (8er-System)

$$\sum_{i=0} \text{ziffer}_i \cdot (\text{basis})^i$$

Schreibweise

Oktalsystem: 0137

$$8^2\ 8^1\ 8^0$$

$$64 + 3 \cdot 8 + 7 = 95$$

Hexadezimalsystem $0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ \underset{10}{A}\ \underset{11}{B}\ \underset{12}{C}\ \underset{13}{D}\ \underset{14}{E}\ \underset{15}{F}$

$$0x1FA$$

$$16^2\ 16^1\ 16^0$$

$$256 + 15 \cdot 16 + 10 = 506$$

"Einstellungen" und andere Dinge mit std::iscream / std::cin:

std::cin >> std::noskipws;

~~Setzt~~ Liest nichts ein, sondern besagt, dass auch whitespace (Enter, ~~Lab, te~~ Tab, Leertasten usw.) eingelesen wird.

std::cin >> std::oct Oktal zahlen einlesen
std::cin >> std::hex Hexadezimal zahlen einlesen.

programm * oct hex. cpp

Schliesslich: gibt std::istream::get
std::istream::peek

char wird ausgelesen / char wird an ausgegeben, aber nicht aus dem Stream entfernt.

Nicht so wichtig, aber kommt in Winter 14/15, Aufgabe 5/6 vor.
Nun: Zusammen Aufgaben BNF Lösen

Winter 14/15	5/6	Zu Zweit lösen, Ich bespreche mit Euch alle Seiten.
Sommer 15	4/5	
Winter 15/16	4	
Do. Seite		

Informatik PVK-Skript Freitag ⑨

Switch-Statement:

```
switch( condition) {
    case 1 :
        dosomething();
        break;
    case 2:
        dosomething();
        break;
    default:
        dosomething;
```

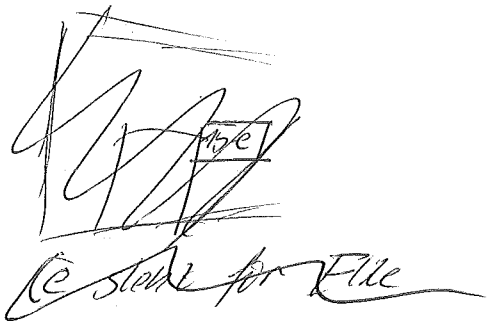
```
if (condition == 1) {
        dosomething();}
else if (condition == 2) {
    do something(); }
else {
do default();
}
```

Geht nur mit konstanten Ausdrücken im Switch.

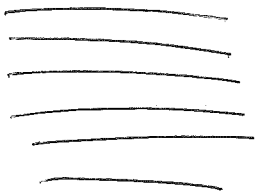
Informatik PVK Skript Freitag (1)

Heute: Klassen - Eliteausbildung

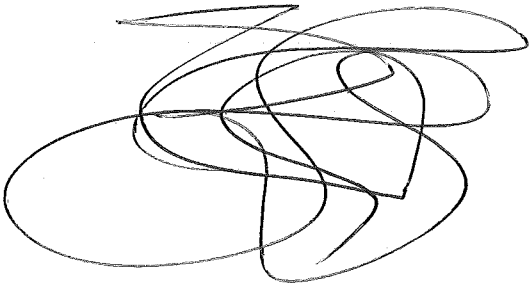
new, delete, Speicher management



Stack
Dt. Stapel



Heap
Dt. Haufen



Wenn wir in einem Programm eine Variable definieren, so ist diese auf dem Stack. Sie wird gelöscht sobald ~~sie~~ wir aus dem Gültigkeitsbereich (Scope) hinausfallen

new int: Wir sagen dem Betriebssystem, dass wir gerne einen Pointer an einen Ort im Speicher hätten, an dem es einen int gibt. Dieser ist auf dem Heap und läuft per Definition NIE aus, ausser ~~nur~~ das Programm wird beendet oder wir löschen ihn.

Vorteile: Man braucht nicht jeden int, den man braucht, explizit in den code zu schreiben. Unglaublich nützlich bei Klassen.

Nachteile: Wenn ich den int nicht lösche, dann bleibt er im Speicher und blockiert dort Speicher.

Programm new1.cpp

delete: Es gibt die Möglichkeit, mit new erschaffene Elemente wieder zu löschen. Das sollte man auch unbedingt tun, da es sonst einen Memory Leak gibt: Dinge, die man nicht löscht, sind bis zum Ende im Speicher.

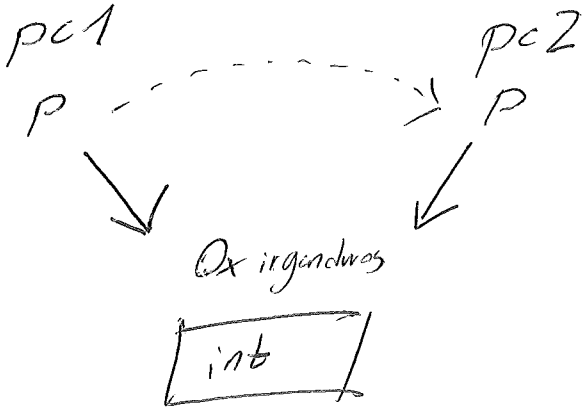
Man kann auch arrays mit new, delete generieren:
new2.cpp

Man kann mit Pointern, new und delete viel falsch machen:
~~Viel~~
~~Viel~~ Wenn ein Pointer an einen falschen Ort zeigt, und man ihn dereferenziert, dann stürzt das Programm ab.
~~Es gibt einen Unterschied zwis~~
(Ein) Unterschied zwischen Pointer und Arrays:
Wenn ein (nicht dynamisch) generiertes Array ausläuft, dann wird der Speicher freigegeben. Bei Pointer ist das i.a. nicht so.
(z. B.) in new2.cpp.

Zeit, ein Memory leak zu demonstrieren: Was macht new3.cpp?

~~Hier Ich bin~~ New und delete sind angeblich nützlich für Klassen, können aber auch immense Probleme bereiten.

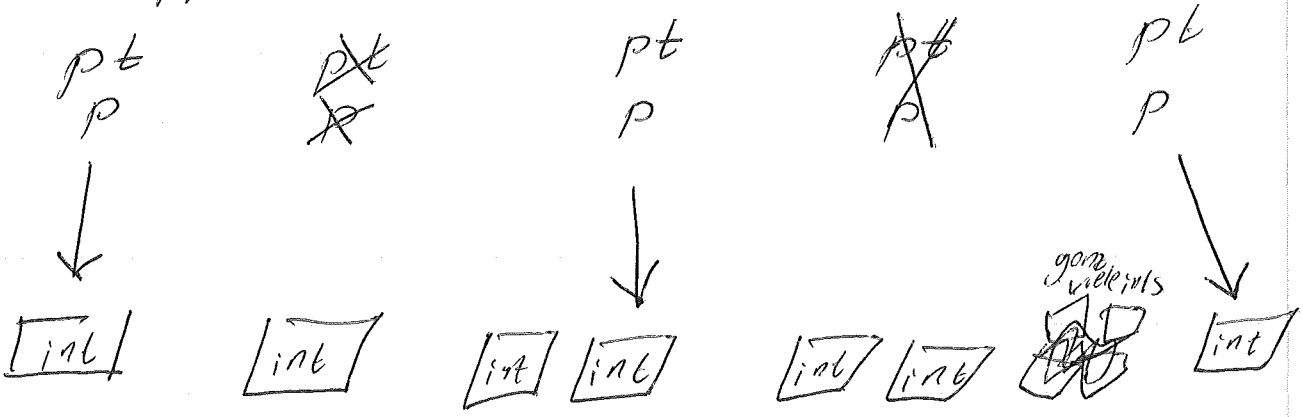
new4.cpp



Fr. (2)

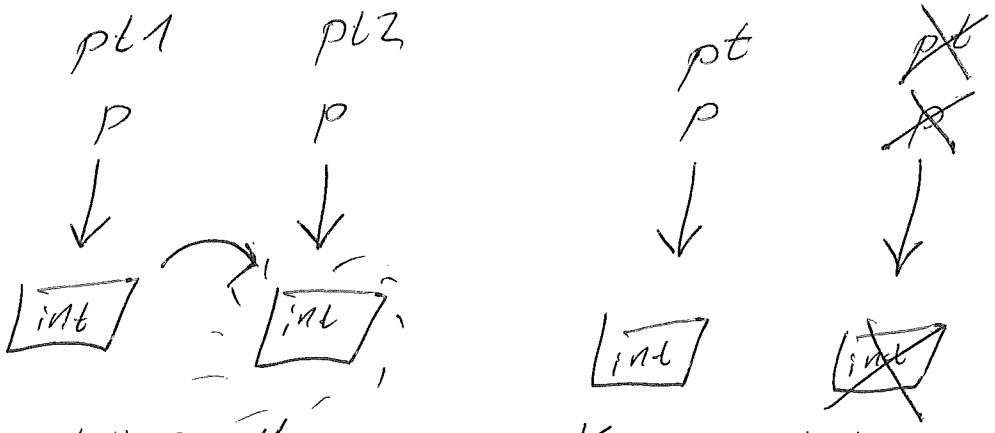
Der Standard-Konstruktor kopiert nur den pointer, nicht das Element auf dem Heap. Die beiden pointer greifen auf dieselbe Sache im Speicher zu: Sehr gefährlich und komplex zu verstehen.

new5.cpp



Standard-Destruktor löscht nur den pointer, nicht den int auf dem heap. ⇒ Memory Leak, Ebenfalls sehr gefährlich, da komplex zu verstehen. Wenn nur wenige ~~aufgerufen werden~~ ~~Konstruct~~ objekte konstruiert werden: Speicher wird langsam immer voller. (z.B. Spielabstürze nach zweieinhalbstündigen Gamingsessions).

Lösung: new6.cpp. Neue copy-~~K~~onstruktor (Achtung Referenz!)
Neue destruktor

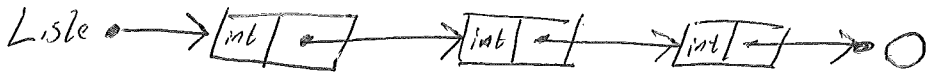


Kein geteilter Zugriff

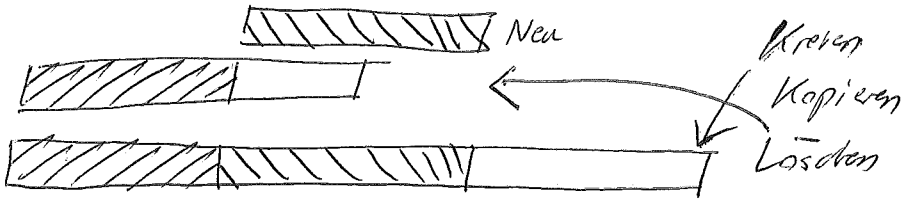
Kein memory Leak

Fazit: Wenn man new, delete braucht, muss man eigene Konstruktoren und Destruktoren schreiben!

Hier: Einfaches Beispiel, man hätte einfach einen mint nehmen können.
Aber: Sehr nützlich für Klassen, die eine Beliebige Menge von etwas beinhalten sollen:



Linked List: ~~Beispiel~~



Array mit dynamischer Grösse

Oder, wenn man irgendein Objekt auf Kommando generieren muss:
Generiere neue Daten basierend darauf wenn etwas neu kommt.
Generiere neues wildes Problem wenn Kampf beginnt.

~~Es ist immer gefährlich, wenn~~
Es besteht immer die Gefahr, dass ein pointer ausläuft, ohne dass ein (oder geändert wird)
Objekt auf dem Heap, auf der der pointer zeigt, gelöscht wird.
Es gibt intelligente pointer, die wissen wann sie auslaufen/geändert werden und ob noch ein anderer pointer auf das objekt, auf das sie gerade zeigen, zeigt. Falls nicht mehr darauf zeigen würde, wird es automatisch gelöscht. Nicht wichtig für Prüfung! Aber für das Leben vielleicht sehr wichtig!

Fr. (3)

The many uses of const:

1. Konstante variablen im Funktionskörper: Werden benutzt, um Konstanten zu definieren : const double versionNumber = 1.5;

2. Konstante Pointer: (pj anderer pointer, j anderer int)

const int *pi } *pi = j ✗ (Keine Änderung der Zahl)

int const *pi } pi = pj ✓ (Änderung pointer ist O.K.)

int * const pi *pi = j ✓ (Änderung der Zahl ist O.K.)

pi = pj ✗ (Keine Änderung des Pointes)

3. Pointer als Argumente von Funktionen

int average(const int * begin, int const * end)

(Keine Änderung was sich hinter den beiden Pointer befindet, innerhalb der Funktion, Änderung wohin pointer zeigen ist O.K.)

4. Konstante Referenzen: float abenteuer(Katze the Cat)

Eine Katze ist 4GB gross!

Wir Kopieren die Katze ⇒ 4GB speicher weg während abenteuer()

Lösung: Konstante Referenz: float abenteuer(const Katze & the Cat)

- Wir Kopieren die Katze nicht
- Der Katze passiert nichts, weil const auf dem Abenteuer, weil const.

const Katze & } dasselbe

Katze const &

Katze & const ← Kein macht keinen Sinn

5. Konstante Klassen-Member: Können nach Klassendeklaration nicht mehr verändert werden. Müssen im Konstruktor gesetzt werden, können aber nicht im Funktionskörper gesetzt werden.

Initialisier-Syntax

```
csl Class (int k, int l) : i(k), j(l) { ... }
```

initialisier.cpp

6. Konstante Methoden:

```
Void mysteryFunction(int &m) const { ... }
```

- Darf nichts an den Membes der Klasse verändern
- Nur const-Methoden können aufgerufen werden, wenn unsere Klasse const ist.
- Sagt nicht über Argumente aus (int &m wird hier verändert)

Beispiel: constClass.cpp

Fallen

Bei std::cout werden bools mit 0, 1 ausgegeben, nicht true/false.

Typen-Koersion: char < int < unsigned int < float < double

Wenn u unsigned int, i signed int, dann !!! kommt ständig!

wenn u+i >0, hat typ unsigned int, verhält sich aber normal; ← kommt ständig!

u+i <0, dann overflowt es (kommt wahrscheinlich nicht an Prüfung, weil Vorgang unbekannt)

Fließkomma: 1.1, 1.1 kann nicht exakt dargestellt werden ⇒ Auslöschung ⇒

1.1-1.1 == 1.0 ist false! (auch IEEE 754) (Ev. bei euch Konvention, Prüfung: False)

Was darstellbar ist: 0.5, 0.25, 0.125 (da $2^{-1}, 2^{-2}, 2^{-3}$), was anderes wird nicht kommen.

for-Schleife: for(a; b; c) Strichkommas!!!

Hinter Klassendefinition: Strichkommas!

Um sicher zu sein: Hinter jede Funktion, Deklaration, Forward declaration in Klassen: Strichkomma!

Bei copy-Constructor: Referenz als Argument!

Wenn new, delete verwendet wird: Immer eigene Konstruktoren, Destruktoren schreiben!