PVK

Informatik für Mathematiker und Physiker

Janet Greutmann

15.01. - 17.01.2020

Vorstellungsrunde

Wir wollen uns etwas kennenlernen...

- Name
- Studiengang
- Hobby

Probeprüfung vom 09.01.2020

Prüfungsanalyse

In der Regel hat die Prüfung folgenden Inhalt:

- 1 Aufgabe vom Typ „Typen und Werte“
- 1 Aufgabe vom Typ „Konstrukte“
- 1 Aufgabe vom Typ „Fliesskommazahlen“
- 1 Aufgabe vom Typ „EBNF “
- 3 bis 4 Programmieraufgaben

Aufbau PVK

Vormittag: Unterricht (Theorie und Beispiele)
Nachmittag: Serien lösen

Die Serien bestehen in der Regel aus drei Teilen:

1. Aufwärmen: Aufgaben zu bereits besprochenen und geübten Themen (sollen unter Zeitdruck wie an der Prüfung gelöst werden)
2. Repetition / Zusammenfassung: letzte Fragen formulieren und klären, ZF schreiben oder ergänzen
3. Übungen: Aufgaben zu den besprochenen Themen (alte Prüfungsaufgaben oder neue Aufgaben)

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Aufgabentyp I: Typen und Werte

Informatik I 2/12 Prüfung 9.2.2018

Aufgabe 1: Typen und Werte (6P)

Geben Sie für jeden der Ausdrücke jeweils C++-Typ und Wert an. *For each of the expressions provide C++type and value.*

/1P (a) `1 / 2 * 2.0`

Typ / *Type*	Wert / *Value*

/1P (b) `12 - 7 % 5`

Typ / *Type*	Wert / *Value*

/1P (c) `true && ! true == false || false`

Typ / *Type*	Wert / *Value*

/1P (d) `0x10 + 0x12 == 32`

Typ / *Type*	Wert / *Value*

/1P (e) `3 / 2.0f + 10 / 2.0`

Typ / *Type*	Wert / *Value*

/1P (f) `4u - 5 < 0`

Typ / *Type*	Wert / *Value*

Abbildung: Aufgabe 1, Prüfung vom Februar 2018

Aufgabentyp I: Typen und Werte

Das sind Gratispunkte :)

- VL: Kap. 1 bis 3 und 7
- Themen: int, unsigned int, bool, float, double (Operationen, Eigenschaften, Konvertierungen)
- Das musst du dafür wissen:
  1. Unterschied Integer-Division und double-Division
  2. Konvertierungen (int $\rightleftarrows$ bool, int $\rightleftarrows$ unsigned int)
  3. Präzedenzen
  4. Unterschied Prä- und Postinkrement
  5. Short-Circuiting

Die verschiedenen Typen

Die verschiedenen Typen

Typ	Wertebereich
bool	{true, false}
int	entspricht ungefähr $\mathbb{Z}$ maximaler und minimaler Wert: Zugriff mit std::numeric_limits<int>::max() (bzw. min) erfordert: #include<limits>)
unsigned int	entspricht ungefähr $\mathbb{N}$ maximaler Wert: Zugriff mit std::numeric_limits<unsigned int>::max() erfordert: #include<limits>)
float	entspricht ungefähr $\mathbb{R}$ maximaler und minimaler Wert: Zugriff mit std::numeric_limits<float>::max() (bzw. min) erfordert: #include<limits>)
double	entspricht ungefähr $\mathbb{R}$ maximaler und minimaler Wert: Zugriff mit std::numeric_limits<double>::max() (bzw. min) erfordert: #include<limits>)

Zugriff auf Grenzen

```
#include <iostream>
#include <limits>

int main() {
  std::cout <<
    std::numeric_limits<int>::max() << "\n";

  std::cout <<
    std::numeric_limits<int>::min() << "\n";

  std::cout <<
    std::numeric_limits<unsigned int>::max() << "\n";

  std::cout <<
    std::numeric_limits<float>::max() << std::endl;

  std::cout <<
    std::numeric_limits<double>::max();
}
```

Unterschied float und double

Der Unterschied liegt in der Präzision:

Typ	Anzahl Stellen	Exponent
float	$\sim$7 Stellen	$\pm$38
double	$\sim$15 Stellen	$\pm$308

Hierarchie

MERKE:

bool $<$ int $<$ unsigned int $<$ float $<$ double

$<$ entspricht „weniger allgemein als“

Operatoren

Operatoren

Typ	**Operationen**
bool	unäre (!) und binäre logische Operatoren (&&, ‖)
int unsigned int	binäre arithmetische Operatoren (+,-,/,*,%,...) relationale Operatoren ($<$, $>$, $\leq$, ...)
float double	binäre arithmetische Operatoren (+,-,/,*,...) relationale Operatoren ($<$, $>$, $\leq$, ...)

Operatoren

binäre logische Operatoren:
bool $\circ$ bool $\rightarrow$ bool

binäre arithmetische Operatoren:
T $\circ$ T $\rightarrow$ T
mit T $\in$ {unsigned int, int, float, double}

relationale Operatoren:
T $\circ$ T $\rightarrow$ **bool**
mit T $\in$ {unsigned int, int, float, double}

Merke: Operator bestimmt Typ von Resultat! Ggf. Terme konvertieren.

Unterschied Integer-Division und double-Division

Unterschied Integer-Division und double-Division

Bei der Integer-Division ist das Ergebnis immer ganzzahlig (die Kommastellen werden einfach nicht beachtet).
Ausserdem gibt es für Integer den Modulo-Operator.

Beispiele:

5 / 2 == ?
5.0 / 2.0 == ?
5 % 2 == ?

Konvertierungen (int $\rightleftarrows$ bool, int $\rightleftarrows$ unsigned int)

Konvertierungen (int $\rightleftarrows$ bool, int $\rightleftarrows$ unsigned int)

int $\rightleftarrows$ bool

int $\rightarrow$ bool
0 $\mapsto$ false
$\mathbb{Z} \setminus \{0\} \rightarrow$ true

bool $\rightarrow$ int
false $\mapsto$ 0
true $\mapsto$ 1

int $\rightleftarrows$ unsigned int

unsigned int $\rightarrow$ int
x u $\mapsto$ x

int $\rightarrow$ unsigned int
$\mathbb{Z}^{\geq 0} \rightarrow \mathbb{Z}^{\geq 0}$
$x \mapsto x$ u

$\mathbb{Z}^{\leq 0} \rightarrow \mathbb{Z}^{>0}$
$x \mapsto (x + 2^{32})$ u

Operatoren und Konvertierungen

Beispiel 1:

true + 2 == ?

Operatoren und Konvertierungen

Beispiel 1:

true + 2 == ?

Operator: + (binärer arithmetischer Operator)
ein binärer arithmetischer Operator verknüpft z.B. zwei Integer oder zwei double etc. miteinander. Aber: keine bool.
D.h. der bool true muss konvertiert werden. Da 2 ein Integer ist, soll auch true dahin konvertiert werden.
true konvertiert nach int wird zu 1 und unsere Gleichung ergibt sich zu:
1 + 2 == 3

Operatoren und Konvertierungen

Beispiel 2:

$(1 < 2)$ && $2 ==$?

Operatoren und Konvertierungen

Beispiel 2:

$(1 < 2)$ && $2 ==$?

Zuerst werten wir die Klammer aus: $(1 < 2) ==$ true
Anschliessend betrachten wir den Ausdruck true && 2. && ist ein binärer logischer Operator und verknüpft somit zwei bool. D.h. die 2 muss von int nach bool konvertiert werden: da $2 \neq 0$, wird 2 auf true abgebildet und wir erhalten true && true $==$ true.

Short-Circuiting

Wahrheitstabellen von logischem UND und ODER:

Short-Circuiting

Wahrheitstabellen von logischem UND und ODER:

&&	true	false
true	true	false
false	false	false

<table>
<tr><th>||</th><th>true</th><th>false</th></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td></tr>
</table>

Short-Circuiting

Wahrheitstabellen von logischem UND und ODER:

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

Die Auswertung binärer logischer Operatoren erfolgt von **links nach rechts**. Falls nach der Auswertung des linken Ausdruckes das Resultat bereits klar ist, wird der rechte Ausdruck nicht berechnet.

Short-Circuiting

Wahrheitstabellen von logischem UND und ODER:

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

Die Auswertung binärer logischer Operatoren erfolgt von **links nach rechts**. Falls nach der Auswertung des linken Ausdruckes das Resultat bereits klar ist, wird der rechte Ausdruck nicht berechnet.

false && X ⇒ false (unabhängig von X)
true || X ⇒ true (unabhängig von X)

Dies ist v.a. dann sehr praktisch, wenn X ein sehr komplizierter Ausdruck ist.

Präzedenzen

Präzedenzen

MERKE:

unäre logische Operatoren (!)
VOR
binäre logische Operatoren (+,-,*,...)
VOR
relationale Operatoren (<,>,...)
VOR
binäre logische Operatore (&&, ||)

Beachte: && bindet stärker als ||

Unterschied Prä- und Postinkrement

++i erhöht i um 1 und nimmt den neuen Wert

i++ nimmt den alten Wert und erhöht dann um 1

Unterschied Prä- und Postinkrement

++i erhöht i um 1 und nimmt den neuen Wert

i++ nimmt den alten Wert und erhöht dann um 1

Beispiel:

Gegeben: int i = 1, int j = 2;
Berechne (++i)*j und (i++)*j.

Unterschied Prä- und Postinkrement

++i erhöht i um 1 und nimmt den neuen Wert

i++ nimmt den alten Wert und erhöht dann um 1

Beispiel:

Gegeben: int i = 1, int j = 2;
Berechne (++i)*j und (i++)*j.

(++i) heisst i wird um eins erhöht und dann wird der Ausdruck mit dem neuen Wert ausgewertet, also mit i = 2. D.h. (++i)*j == 2*2 == 4.

(i++) heisst der Ausdruck wird mit dem alten Wert ausgewertet und dann wird i um eins erhöht. Sprich für die Multiplikation nimm i = 1 und der gesamte Ausdruck wird zu 1*2 == 2. Anschliessend erhöhe i um eins (jetzt: i = 2).

Üben wir ein bisschen...

```
int i = 10;
double d = 2;
int m = 7;
int pre = 1;
int post = 1;
unsigned int u = 0;
```

(a) `1 / 4 * d` /1P

Typ / *Type* | Wert / *Value*

(b) `m % m` /1P

Typ / *Type* | Wert / *Value*

(c) `i = 10` /1P

Typ / *Type* | Wert / *Value*

(d) `i += i` /1P

Typ / *Type* | Wert / *Value*

(e) `++pre / post--` /1P

Typ / *Type* | Wert / *Value*

(f) `u-5 < 6` /1P

Typ / *Type* | Wert / *Value*

Abbildung: Aufgabe 1, Prüfung vom August 2017

Lösungen

(a) `1 / 4 * d` /1P

Typ / *Type*	Wert / *Value*
`double`	`0`

(b) `m % m` /1P

Typ / *Type*	Wert / *Value*
`int`	`0`

(c) `i = 10` /1P

Typ / *Type*	Wert / *Value*
`int`	`10`

(d) `i += i` /1P

Typ / *Type*	Wert / *Value*
`int`	`20`

(e) `++pre / post--` /1P

Typ / *Type*	Wert / *Value*
`int`	`2`

(f) `u-5 < 6` /1P

Typ / *Type*	Wert / *Value*
`bool`	`false`

Abbildung: Aufgabe 1, Prüfung vom August 2017

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Aufgabentyp II: Fliesskommazahlen

Informatik I 12/12 Prüfung 9.2.2018

Aufgabe 7: Zahlendarstellungen (10P)

/6P (a) Geben Sie ein möglichst knappes normalisiertes Fliesskommazahlensystem an, mit welchem sich die folgenden dezimalen Werte gerade noch genau darstellen lassen: jede Verkleinerung von p, $e_{\max}$ oder $-e_{\min}$ muss dazu führen, dass mindestens eine Zahl nicht mehr dargestellt werden kann.
Hinweis: p zählt auch die führende Ziffer.
Tipp: Schreiben Sie sich die normalisierte Binärzahldarstellung der Werte auf, wenn sie für Sie nicht offensichtlich ist.

Provide the smallest possible normalized floating point number system that can still represent the following values exactly: any decrease of the numbers p, $e_{\max}$ or $-e_{\min}$ must imply that at least one of the numbers cannot be represented any more.
Hint: p does also count the leading digit.
Tip: Write down the normalized binary representation of the values, if it is not obvious for you.

Werte / *Values*: $34,\ 1.75,\ 65/64,\ 1/128$

$F^*(\beta, p, e_{\min}, e_{\max})$ mit / *with*
$\beta = 2$, $p =$, $e_{\min} =$, $e_{\max} =$.

/4P (b) Die Dezimalzahlen $x = 40$ und $y = 1$ sind im normalisierten Fliesskommasystem $F^*(2,5,0,5)$ ($p = 5$) darstellbar.
Geben Sie das Resultat der Berechnung von $x + y$ in F^* an. Geben Sie die Antwort in dezimaler und binärer Darstellung an. Hinweis: Verwenden Sie arithmetisches Runden.
Geben Sie jeden einzelnen Schritt der Berechnung an.

The decimal numbers $x = 40$ and $y = 1$ are representable in the normalized floating point system $F^*(2,5,0,5)$ ($p = 5$).
Provide the result of $x + y$ when computed in F^*. Provide the answer in binary und decimal representation. Hint: Use arithmetic rounding to find this number.
Provide each single step of the computation.

Abbildung: Aufgabe 7, Prüfung vom Februar 2018

Aufgabentyp II: Fliesskommazahlen

Das sind Gratispunkte :)

- VL: Kap. 7 und 8
- Themen: Fliesskommazahlen
- Das musst du dafür wissen:
  1. Konvertierung von ganzen Zahlen ins Binärsystem und umgekehrt (oder andere Zahlensysteme)
  2. Konvertierung von Dezimalzahlen ins Binärsystem
  3. Normalsystem $F^*(b, p, e_{min}, e_{max})$

Konvertierung von ganzen Zahlen ins Binärsystem und umgekehrt (oder andere Zahlensysteme)

Dezimalsystem → x-System

x-System → Dezimalsystem

Konvertierung von ganzen Zahlen ins Binärsystem und umgekehrt (oder andere Zahlensysteme)

Dezimalsystem → x-System

Dezimalzahl : x = A Rest a
A : x = B Rest b
B : x = C Rest c
⋮
Y : x = 1 Rest z

Dann ergibt sich als Resultat: 1z...cda im x-System (Buchstaben symbolisieren beliebige Ziffern).

x-System → Dezimalsystem

Sei abcdef...yz eine Zahl im x-System mit einer bekannten Anzahl Ziffern n.

Die korrespondierende Dezimalzahl ergibt sich wie folgt:

Dezimalzahl = $z \cdot x^0 + y \cdot x^1 + ... + f \cdot x^{n-6} + e \cdot x^{n-5} + ... + a \cdot x^{n-1}$

Beispiel: Konvertierung von ganzen Zahlen ins 3er-System und umgekehrt

Dezimalsystem → 3er-System

Sei 35 eine Zahl im Dezimalsystem.

3er-System → Dezimalsystem

Sei 12012 eine Zahl im 3er-System.

Beispiel: Konvertierung von ganzen Zahlen ins 3er-System und umgekehrt

Dezimalsystem → 3er-System

Sei 35 eine Zahl im Dezimalsystem.

$35 : 3 = 11$ Rest 2
$11 : 3 = 3$ Rest 2
$3 : 3 = 1$ Rest 0

Dann ergibt sich als Resultat: 1022 im 3er-System.

3er-System → Dezimalsystem

Sei 12012 eine Zahl im 3er-System. #Ziffern $n = 5$.

Die korrespondierende Dezimalzahl ergibt sich wie folgt:

$2 \cdot 3^0 + 1 \cdot 3^1 + 0 \cdot 3^2 + 2 \cdot 3^3 + 1 \cdot 3^4 =$
$2 + 3 + 0 + 54 + 81 = 140$

Konvertierung von Dezimalzahlen ins Binärsystem

Benutze folgende Tabelle:

Sei Z eine Dezimalzahl.

x	b_i	$x - b_i$	$2(x - b_i)$
Z	$b_0 = \ldots$	$Z - b_0$	$2(Z - b_0) =: Z_1$
Z_1	$b_1 = \ldots$	$Z_1 - b_1$	$2(Z_1 - b_1) =: Z_2$
...			

Die Binärzahl ergibt sich zu $b_0.b_1b_2b_3b_4\ldots$ (nicht zwingend normalisiert!)

Beispiel: Konvertierung von 1.15_{10} ins Binärsystem

x	b_i	$x - b_i$	$2(x - b_i)$
1.15			

Beispiel: Konvertierung von 1.15_{10} ins Binärsystem

x	b_i	$x - b_i$	$2(x - b_i)$
1.15	$b_0 = 1$	0.15	0.3
0.3	$b_1 = 0$	0.3	0.6
0.6	$b_2 = 0$	0.6	1.2
1.2	$b_3 = 1$	0.2	0.4
0.4	$b_4 = 0$	0.4	0.8
0.8	$b_5 = 0$	0.8	1.6
1.6	$b_1 = 1$	0.6	**1.2**

Die Zahl ist periodisch!
Die Binärzahl ergibt sich zu $1.00\overline{1001}$

Das typischste Beispiel für eine periodische Darstellung im Binärsystem ist die Dezimalzahl 0.1

Abkürzung

Für alle Dezimalzahlen der Form $\frac{n}{2^m}$, $n, m \in \mathbb{N}$, benutze folgende Tabelle, um abzukürzen.

binary	1	1	1	1	.	1	1	1	1
decimal	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Abkürzung

Für alle Dezimalzahlen der Form $\frac{n}{2^m}$, $n, m \in \mathbb{N}$, benutze folgende Tabelle, um abzukürzen.

binary	1	1	1	1	.	1	1	1	1
decimal	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Vorgehen:

1. Zerlege deine Dezimalzahl und wandle die einzelnen Summanden direkt um.
2. Addiere im Binärsystem
3. Normalisiere die Zahl (falls verlangt).

Abkürzung

Für alle Dezimalzahlen der Form $\frac{n}{2^m}$, $n, m \in \mathbb{N}$, benutze folgende Tabelle, um abzukürzen.

binary	1	1	1	1	.	1	1	1	1
decimal	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Vorgehen:

1. Zerlege deine Dezimalzahl und wandle die einzelnen Summanden direkt um.
2. Addiere im Binärsystem
3. Normalisiere die Zahl (falls verlangt).

Beispiel: Konvertiere $\frac{3}{4}_{10}$ ins Binärsystem.

1. $\frac{3}{4}_{10} = \frac{1}{2}_{10} + \frac{1}{4}_{10} = 0.1_2 + 0.01_2$
2. $0.1_2 + 0.01_2 = 0.11_2$

Konvertierung vom Dezimalsystem ins Binärsystem

Zusammengefasst haben wir bei der Konvertierung von Dezimalzahlen ins Binärsystem 3 Fälle:

1. **Ganze Zahlen**
   Konvertierung mit gezeigtem Algorithmus
2. **Dezimalzahlen der Form $\frac{n}{2^m}$, $n, m \in \mathbb{N}$**
   Verwende Abkürzung
3. **alle anderen Fliesskommazahlen (meistens periodische Darstellung im Binärsystem)**
   Verwende das Vorgehen mit der Tabelle

Normalsystem $F^*(\beta, p, e_{min}, e_{max})$

Normalsystem $F^*(\beta, p, e_{min}, e_{max})$

Definition:

Basis $\beta \geq 2$
Präzision $p \geq 1$
kleinster Exponent e_{min}
grösster Exponent e_{max}

Normalisierte Darstellung:

$\pm d_0.d_1...d_{p-1} \cdot \beta^e$
mit:
$e \in \{e_{min}, ..., e_{max}\}$
$d_i \in \{0, ..., \beta - 1\}$
$d_0 \neq 0$

Runden (Binärzahlen):

Wenn auf n Stellen gerundet werden soll, betrachte die (n+1)-te Stelle. Ist sie gleich 1, wird aufgerundet, ist sie gleich null, wird abgerundet bzw. die n-te Stelle so belassen.

Mögliche Aufgabenstellungen

Typ 1

Aufgabenstellung:

Gegeben sind verschiedene Zahlen im Dezimalsystem.
Gesucht wird das kleinst mögliche Normalsystem ($\beta = 2$), so dass alle Zahlen eine exakte Darstellung besitzen.

Vorgehen:

1. Konvertiere alle Zahlen ins Binärsystem (beachte die 3 versch. Fälle)
2. Normalisiere die Zahlen (nicht runden)
3. Bestimme für alle Zahlen die Präzision p und den Exponenten e.
4. Setze $p =$ grösste Präzision, $e_{min} =$ kleinster Exponent und $e_{max} =$ grösster Exponent

Mögliche Aufgabenstellungen

Typ 2

Aufgabenstellung:

Gegeben sind verschiedene Zahlen im Dezimalsystem und ein Normalsystem ($\beta = 2$).
Man soll beurteilen, welche der gegebenen Zahlen eine exakte Darstellung im gegebenen Normalsystem haben.

Vorgehen:

1. Konvertiere alle Zahlen ins Binärsystem (beachte die 3 versch. Fälle)
2. Normalisiere die Zahlen (nicht runden)
3. Bestimmte für alle Zahlen die Präzision p und den Exponenten e
4. Vergleiche p und e. Falls diese nicht im durch das Normalsystem definierten Intervall liegen, liegt die exakte Zahl nicht im Normalsystem.

Mögliche Aufgabenstellungen

Typ 3

Aufgabenstellung:

Gegeben sind verschiedene Zahlen im Dezimalsystem.
Man soll beurteilen, welche Zahlen im Binärsystem eine periodische Darstellung besitzen und welche nicht.

Vorgehen:

1. Konvertiere alle Zahlen ins Binärsystem (beachte die 3 versch. Fälle)

Üben wir ein bisschen...

/6P (a) Geben Sie ein möglichst knappes normalisiertes Fliesskommazahlensystem an, mit welchem sich die folgenden dezimalen Werte gerade noch genau darstellen lassen: jede Verkleinerung von p, $e_{\max}$ oder $-e_{\min}$ muss dazu führen, dass mindestens eine Zahl nicht mehr dargestellt werden kann.
Hinweis: p zählt auch die führende Ziffer.
Tipp: Schreiben Sie sich die normalisierte Binärzahldarstellung der Werte auf, wenn sie für Sie nicht offensichtlich ist.

Provide the smallest possible normalized floating point number system that can still represent the following values exactly: any decrease of the numbers p, $e_{\max}$ or $-e_{\min}$ must imply that at least one of the numbers cannot be represented any more.
Hint: p does also count the leading digit.
Tip: Write down the normalized binary representation of the values, if it is not obvious for you.

Werte / *Values*: $34,\ 1.75,\ 65/64,\ 1/128$

$F^*(\beta, p, e_{\min}, e_{\max})$ mit / *with*

$\beta = 2 \quad , p = \qquad , e_{\min} = \qquad , e_{\max} = \qquad .$

Abbildung: Aufgabe 7, Prüfung vom Februar 2018

Lösung

(a) Geben Sie ein möglichst knappes normalisiertes Fliesskommazahlensystem an, mit welchem sich die folgenden dezimalen Werte gerade noch genau darstellen lassen: jede Verkleinerung von p, $e_{\max}$ oder $-e_{\min}$ muss dazu führen, dass mindestens eine Zahl nicht mehr dargestellt werden kann.
Hinweis: p zählt auch die führende Ziffer.
Tipp: Schreiben Sie sich die normalisierte Binärzahldarstellung der Werte auf, wenn sie für Sie nicht offensichtlich ist.

Provide the smallest possible normalized floating point number system that can still represent the following values exactly: any decrease of the numbers p, $e_{\max}$ or $-e_{\min}$ must imply that at least one of the numbers cannot be represented any more.
Hint: p does also count the leading digit.
Tip: Write down the normalized binary representation of the values, if it is not obvious for you.

Werte / *Values*: 34, 1.75, $65/64$, $1/128$

$F^*(\beta, p, e_{\min}, e_{\max})$ mit / *with*

$\beta = 2$, $p = 7$, $e_{\min} = -7$, $e_{\max} = 5$.

Abbildung: Aufgabe 7, Prüfung vom Februar 2018

Üben wir ein bisschen...

(a) Geben Sie ein möglichst knappes normalisiertes Fliesskommazahlensystem an, mit welchem sich die folgenden dezimalen Werte gerade noch genau darstellen lassen: jede Verkleinerung von p, $e_{\max}$ oder $-e_{\min}$ muss dazu führen, dass mindestens eine Zahl nicht mehr dargestellt werden kann.
Hinweis: p zählt auch die führende Ziffer.
Tipp: Schreiben Sie sich die normalisierte Binärzahldarstellung der Werte auf, wenn sie für Sie nicht offensichtlich ist.

Provide a smallest possible normalized floating point number system that can still represent the following values exactly: any decrease of the numbers p, $e_{\max}$ or $-e_{\min}$ must imply that at least one of the numbers cannot be represented any more.
Hint: p does also count the leading digit.
Tip: Write down the normalized binary representation of the values, if it is not obvious for you.

/3P

Werte / *Values*: $2.25,\ \frac{1}{8},\ 0.5,\ 16.5,\ 2^3$

$F^*(\beta, p, e_{\min}, e_{\max})$ mit / *with*
$\beta = 2$, $p =$, $e_{\min} =$, $e_{\max} =$.

(b) Markieren die Werte, die eine exakte Darstellung in einem beliebigen (endlichen) normalisierten binären Fliesskommazahlensystem besitzen.

Mark the values that have an exact representation in an arbitrary (finite) normalized binary floating point number system.

/3P

○ 123.4 ○ 0.025 ○ $1/10$ ○ 1000.5 ○ $1/16$ ○ $1.5/32$

(c) Die Zahl in der linken Spalte der nachfolgenden Tabelle ist jeweils als Literal der Sprache C++zu verstehen. Berechnen Sie, was jeweils verlangt ist.

The number displayed to the left in the following table shall be considered a number literal in C++language. Compute what is requested.

/2P

0x7FF	Dezimalzahl / *decimal number*	=	
-16	Binärzahl mit 6 Bits im Zweierkomplement / *Binary number with 6 bits in two's complement*	=	

Abbildung: Aufgabe 3, Prüfung vom August 2017

Lösung

(a) Geben Sie ein möglichst knappes normalisiertes Fliesskommazahlensystem an, mit welchem sich die folgenden dezimalen Werte gerade noch genau darstellen lassen: jede Verkleinerung von p, $e_{\max}$ oder $-e_{\min}$ muss dazu führen, dass mindestens eine Zahl nicht mehr dargestellt werden kann.
Hinweis: p zählt auch die führende Ziffer.
Tipp: Schreiben Sie sich die normalisierte Binärzahldarstellung der Werte auf, wenn sie für Sie nicht offensichtlich ist.

Provide a smallest possible normalized floating point number system that can still represent the following values exactly: any decrease of the numbers p, $e_{\max}$ or $-e_{\min}$ must imply that at least one of the numbers cannot be represented any more.
Hint: p does also count the leading digit.
Tip: Write down the normalized binary representation of the values, if it is not obvious for you.

/3P

Werte / *Values*: $2.25,\ \frac{1}{8},\ 0.5,\ 16.5,\ 2^3$

$F^*(\beta, p, e_{\min}, e_{\max})$ mit / *with*
$\beta = 2$, $p = 6$, $e_{\min} = -3$, $e_{\max} = 4$.

(b) Markieren die Werte, die eine exakte Darstellung in einem beliebigen (endlichen) normalisierten binären Fliesskommazahlensystem besitzen.

Mark the values that have an exact representation in an arbitrary (finite) normalized binary floating point number system.

/3P

○ 123.4 ○ 0.025 ○ $1/10$ √ 1000.5 √ $1/16$ √ $1.5/32$

(c) Die Zahl in der linken Spalte der nachfolgenden Tabelle ist jeweils als Literal der Sprache C++zu verstehen. Berechnen Sie, was jeweils verlangt ist.

The number displayed to the left in the following table shall be considered a number literal in C++language. Compute what is requested.

/2P

0x7FF	Dezimalzahl / *decimal number*	= 2047
-16	Binärzahl mit 6 Bits im Zweierkomplement / *Binary number with 6 bits in two's complement*	= 110000

Abbildung: Aufgabe 3, Prüfung vom August 2017

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Aufgabentyp III: Konstrukte

Informatik I 3/12 Prüfung 9.2.2018

Aufgabe 2: Konstrukte (8P)

Geben Sie zu folgenden Codestücken jeweils die erzeugte Ausgabe an.

Provide the output for each of the following pieces of code.

(a)

```
int a = 10;
int& b = a;
b -= a;
std::cout << a + b;
```

Ausgabe / *Output*:

/2P

(b)

```
int a[] = {1,8,0,2,7,4,3,6,5};
for (int i = 3; a[i] != 5; i = a[i]){
    std::cout << *(a+i);
}
```

Ausgabe / *Output*:

/3P

(c)

```
int res = 0;
int d = 1;
int val = 101010;
while (val > 0){
    res += val % 10 * d;
    d *= 2;
    val /= 10;
}
std::cout << res;
```

Ausgabe / *Output*:

/3P

Abbildung: Aufgabe 2, Prüfung vom Februar 2018

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Loops

Es gibt drei verschiedene Loops:

1. for

```
for (init; cond; expr){
  BODY
}
```

2. while

```
while (cond) {
  BODY
}
```

3. do while

```
do {
  BODY
} while (cond);
```

Anwendungen

1. for
2. while
3. do while

Anwendungen

1. for
   wenn Anzahl Iterationen bekannt sind
   Beispiel: Ausgabe eines Vektors der Länge n
2. while
3. do while

Anwendungen

1. for
   wenn Anzahl Iterationen bekannt sind
   Beispiel: Ausgabe eines Vektors der Länge n
2. while
   solange bis die Bedingung nicht mehr wahr ist (Anzahl Iterationen nicht unbedingt bekannt)
   Beispiel: Berechnung des Rests einer Division
3. do while

Anwendungen

1. for
   wenn Anzahl Iterationen bekannt sind
   Beispiel: Ausgabe eines Vektors der Länge n
2. while
   solange bis die Bedingung nicht mehr wahr ist (Anzahl Iterationen nicht unbedingt bekannt)
   Beispiel: Berechnung des Rests einer Division
3. do while
   Loop soll mindestens einmal ausgeführt werden
   Beispiel: Etwas berechnen und dann fragen, ob man nochmal will.

Anwendungen

1. for
   wenn Anzahl Iterationen bekannt sind
   Beispiel: Ausgabe eines Vektors der Länge n
2. while
   solange bis die Bedingung nicht mehr wahr ist (Anzahl Iterationen nicht unbedingt bekannt)
   Beispiel: Berechnung des Rests einer Division
3. do while
   Loop soll mindestens einmal ausgeführt werden
   Beispiel: Etwas berechnen und dann fragen, ob man nochmal will.

Man kann mit verschiedenen Loops auch genau dasselbe machen (wenn man will).

Äquivalente Loops

1. for

```
for (unsigned int i = 0; i < 10; ++i)
  std::cout << i << " ";
```

2. while

```
unsigned int i = 0;
while (i < 10) {
  std::cout << i << " ";
  ++i;
}
```

3. do while

```
unsigned int i = 0;
do {
  std::cout << i << " ";
  ++i;
} while (i < 10);
```

Übung

Gibt es bei den folgenden Loops Unterschiede? Falls ja, welche?

```
for (unsigned int i = 0; i < 10; ++i)
  std::cout << i << " ";
```

```
unsigned int i = 0;
while (i < 10) {
  std::cout << ++i << " ";
}
```

```
unsigned int i = 0;
do {
  std::cout << i++ << " ";
} while (i <= 10);
```

Lösung

Sie unterscheiden sich im Output:

```
for (unsigned int i = 0; i < 10; ++i)
  std::cout << i << " ";

// 0 1 2 3 4 5 6 7 8 9
```

```
unsigned int i = 0;
while (i < 10) {
  std::cout << ++i << " ";
}

// 1 2 3 4 5 6 7 8 9 10
```

```
unsigned int i = 0;
do {
  std::cout << i++ << " ";
} while (i <= 10);

// 0 1 2 3 4 5 6 7 8 9 10
```

if und else

Um eine Fallunterscheidung zu machen, verwende if und else.

```
if (cond) {
  BODY
}
else if (cond) {
  BODY
}
else {
  BODY
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Vektoren

Ein Vektor ist ein zusammenhängender Speicherblock. Der Index des ersten Eintrags ist 0.

Deklaration und Initialisierung:

```
#include <iostream>
#include <vector>


int main() {
  std::vector<int> vec; // leerer Vektor deklariert
  std::vector<int> vec2(2,0); // 0 0
  std::vector<int> vec3 = { -1, 2, -3, 4 }; // -1 2 -3 4
  std::vector<char> vec4 = { 'a', 'b', 'c' }; // a b c

  return 0;
}
```

Funktionen

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> vec = { -1, 2, -3, 4 };
  unsigned int length = vec.size(); // Laenge des Vektors
  vec.push_back(5); // Element 5 wird an Vektor angehaengt
  // -1 2 -3 4 5
  vec.pop_back(); // letztes Element wird geloescht
  // -1 2 -3 4
  vec.at(2) = 7;  // Element bei Index 2 wird auf 7 gesetzt
  // -1 2 7 4
  vec[3] = 9; // Element bei Index 3 wird auf 9 gesetzt
  // -1 2 7 9
  return 0;
}
```

Benutze bevorzugt vec.at(i) anstatt vec[i], denn beim Aufruf vec.at(i) wird zusätzlich geprüft, ob es ein erlaubter Zugriff ist (bzw. ob der Vektor mind. Länge $i + 1$ hat).

Mehrdimensionale Vektoren

Matrizen sind in C++ Vektoren, wobei in jedem Eintrag ein Vektor steht.

```
#include <iostream>
#include <vector>

int main() {
  std::vector<std::vector<int>>
    mat(n_rows, std::vector<int>(n_cols, value));

  std::vector<std::vector<int>>
    matrix(2, std::vector<int>(3,0));
  // 0 0 0
  // 0 0 0

  number_of_rows = matrix.size();
  number_of_cols = matrix[0].size();

  return 0;
}
```

Mehrdimensionale Vektoren

Iteriere durch die Matrix mit einem Doppelloop.

```
#include <iostream>
#include <vector>
using irow = std::vector<int>;
using matrix = std::vector<irow>;

void output(const matrix& m) {
  for (unsigned int row = 0; row < m.size(); ++row) {
    for (unsigned int col = 0; col < m[0].size(); ++col)
      std::cout << m.at(row).at(col) << " ";
    std::cout << "\n";
  }
}

int main() {
  matrix mat(2, irow(3,0));
  output(mat);
  return 0;
}
```

Aufgabe

Schreibe ein Programm, das eine Matrix vom Standardinput liest, sie transponiert und wieder ausgibt.

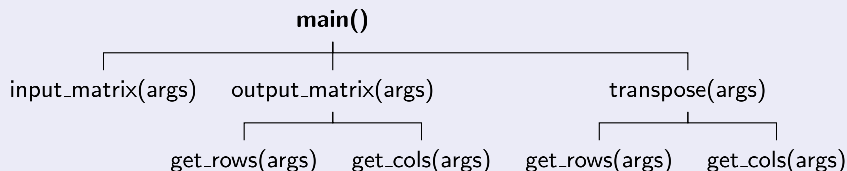
Format Eingabe: Zuerst zwei Zahlen, die die Anzahl Zeilen und die Anzahl Spalten angeben. Anschliessend die Zeilen der Matrix.

Aufgabe

Schreibe ein Programm, das eine Matrix vom Standardinput liest, sie transponiert und wieder ausgibt.

Format Eingabe: Zuerst zwei Zahlen, die die Anzahl Zeilen und die Anzahl Spalten angeben. Anschliessend die Zeilen der Matrix.

Struktur

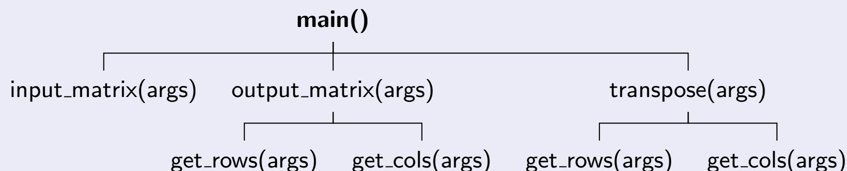


Aufgabe

Schreibe ein Programm, das eine Matrix vom Standardinput liest, sie transponiert und wieder ausgibt.

Format Eingabe: Zuerst zwei Zahlen, die die Anzahl Zeilen und die Anzahl Spalten angeben. Anschliessend die Zeilen der Matrix.

Struktur



Die Lösung findet ihr in code expert.

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Funktionen

Wieso soll man Funktionen verwenden?

- Vermeidung von „Spaghetticode“
- Auslagerung von Aufgaben
- Code sparen (Funktion mehrmals aufrufen anstatt Copy Paste)
- Zeit sparen (bei Copy Paste müsste man es bei jeder Kopie ändern)
- Übersichtlichkeit

Pre- und Postconditions

Die Precondition schränkt die Argumente, die der Funktion übergeben werden dürfen, ein.

Die Postcondition beschreibt, was die Funktion tut und zurückgibt.

Preconditions sollen möglichst schwach sein, sodass die Funktion oft angewendet werden kann. Argumente, die nicht in die Funktion rein dürfen, sind mit assert abzufangen.

Postconditions sollen möglichst stark sein, sodass klar ist, was zurück kommt.

Eine Precondition ist z.B. sinnvoll:

- Division durch Null vermeiden, indem Null als Argument ausgeschlossen wird.
- grösster Integer ausschliessen, um Endlosschleifen zu vermeiden.
- Relationen zwischen Argumenten (z.B. $a < b$ oder Ähnliches)

Kleine Beispiele

Schreibe ein Programm, welches folgende Funktionalitäten besitzt:

1. Maximum von zwei beliebigen ganzen Zahlen berechnen
2. eine beliebige ganze Zahl quadrieren
3. Betrag einer beliebigen ganzen Zahl

Lösung

```
int max(int a, int b) {
  if (a < b)
    return b;
  else
    return a;
}

int betrag(int a) {
  if (a >= 0)
    return a;
  else
    return -a;
}

int pow(int a) {
  return a * a;
}
```

Anderes Beispiel

Schreibe eine Funktion, die prüft, ob eine natürliche Zahl durch 2, 3 und 5 teilbar ist.

Anderes Beispiel

Schreibe eine Funktion, die prüft, ob eine natürliche Zahl durch 2, 3 und 5 teilbar ist.

```
// Verschachtelte if's (ACHTUNG!)
bool is_what_i_want(unsigned int a) {
  if (a % 2 == 0) {
    if (a % 3 == 0) {
      if (a % 5 == 0) {
        return true;
      }
    }
  }
  return false;
}

// Boolean und Short-Circuiting
bool is_what_i_want_v2(unsigned int a) {
  return (a % 2 == 0) && (a % 3 == 0) && (a % 5 == 0);
}
```

FALSCH!

```
// Verschachtelte if's (FALSCH!)
bool is_what_i_want_falsch(unsigned int a) {
  if (a % 2 == 0) {
    if (a % 3 == 0) {
      if (a % 5 == 0) {
        return true;
      }
    }
  }
  else
    return false;
}
```

Funktion hat nicht in jedem Fall einen Rückgabewert und das führt zu undefined behaviour!

Referenzen

Einführungsbeispiel: Was ist der Output?

```
int main() {

  int a = 3;
  int& b = a;
  b = 2;
  std::cout << a;

  return 0;
}
```

Pass by value vs. pass by reference

Worin besteht der Unterschied? Was ist der Output?

```
void foo(int i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

```
void foo(int& i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

Pass by value vs. pass by reference

Worin besteht der Unterschied? Was ist der Output?

```
void foo(int i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

Output: 4

```
void foo(int& i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

Output: 5

Pass by value vs. pass by reference

Worin besteht der Unterschied? Was ist der Output?

pass by value

```
void foo(int i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

Output: 4

pass by reference

```
void foo(int& i) {
  i = 5;
}

int main() {
  int i = 4;
  foo(i);
  std::cout << i <<
    std::endl;
}
```

Output: 5

Wieso Referenzen?

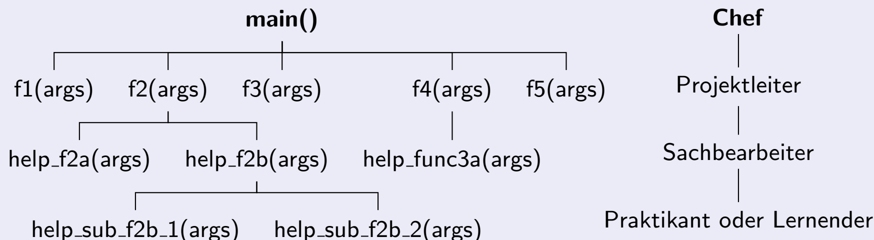
Wieso Referenzen?

- mehr als ein Rückgabewert bei einer Funktion
- keine Kopien (insbesondere bei Matrizen, Vektoren etc.)
- nützlich bei Objekten, die nicht kopiert werden können

Funktionen: Hierarchie

Eine Funktion übernimmt **eine** Aufgabe. Entweder für die main() Funktion oder für eine andere Funktion.

Vorstellung:



Nach unten: Aufgaben zum Lösen abgeben
Nach oben: Resultat / Lösung zurückgeben
main() koordiniert Reihenfolge der Funktionsaufrufe der Funktionen der nächsten Ebene.

Die Hilfsfunktionen einer Funktion werden entweder vor der Funktion (in welcher sie dann aufgerufen werden) deklariert oder bereits definiert.

Beispiel

Es soll eine Folge von Zahlen (beendet mit -1) eingelesen und in umgekehrter Reihenfolge ausgegeben werden.

Beispiel

Es soll eine Folge von Zahlen (beendet mit -1) eingelesen und in umgekehrter Reihenfolge ausgegeben werden.

Zu vergebende Aufgaben:

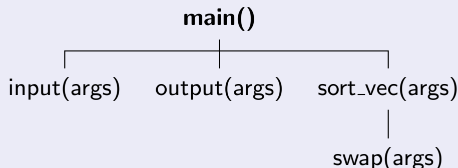
- Input:
  Zahlen einlesen und Vektor erstellen
- Output:
  Vektor ausgeben
- Sortieren:
  Reihenfolge der Vektoreinträge umkehren
- Swap:
  Zwei Einträge tauschen

Beispiel

Es soll eine Folge von Zahlen (beendet mit -1) eingelesen und in umgekehrter Reihenfolge ausgegeben werden.

Zu vergebende Aufgaben:

- Input:
  Zahlen einlesen und Vektor erstellen
- Output:
  Vektor ausgeben
- Sortieren:
  Reihenfolge der Vektoreinträge umkehren
- Swap:
  Zwei Einträge tauschen

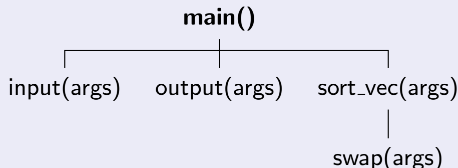


Beispiel

Es soll eine Folge von Zahlen (beendet mit -1) eingelesen und in umgekehrter Reihenfolge ausgegeben werden.

Zu vergebende Aufgaben:

- Input:
  Zahlen einlesen und Vektor erstellen
- Output:
  Vektor ausgeben
- Sortieren:
  Reihenfolge der Vektoreinträge umkehren
- Swap:
  Zwei Einträge tauschen



Implementiere diese Aufgabe in einem leeren Projekt (z.B. Playground) in code expert.

Lösungsvorschlag (Teil 1)

```
#include <iostream>
#include<vector>

std::vector<int> input() {
  std::vector<int> vec;
  int n;
  std::cin >> n;
  while (n != -1) {
    vec.push_back(n);
    std::cin >> n;
  }
  return vec;
}

void output(const std::vector<int>& vec) {
  for (unsigned int i = 0; i < vec.size(); ++i)
    std::cout << vec.at(i) << " ";
}
```

Lösungsvorschlag (Teil 2)

```
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}

void sort_vec(std::vector<int>& vec) {
  for (unsigned int i = 0; i < 0.5*vec.size();++i)
    swap(vec.at(i), vec.at(vec.size() - 1 - i));
}

int main() {
  std::vector<int> v;
  v = input();
  sort_vec(v);
  output(v);

  return 0;
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Rekursion Mathematik

Beispiel aus der Mathematik: *Rekursive Definition Fakultät*

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

oder Folgen, etc.

Rekursion Informatik

Analoges Prinzip für Informatik:

1. base case (sonst terminiert die Rekursion nicht!)
2. Rekursionsaufruf $n = function(n-1)$

Fakultät

Fakultät rekursiv implementieren:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
unsigned int fac(unsigned int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n - 1);
}
```

Program Tracking

Was ist der Output des folgenden Programms?

```
#include <iostream>

void func(const int n) {
  if (n > 1) {
    func(n / 2);
    func(n / 2);
  }

  for (int i = 0; i < n; ++i)
    std::cout << "*";
  std::cout << " ";
}

int main() {
  int n = 4;
  func(4);
  return 0;
}
```

Program Tracking

Was ist der Output des folgenden Programms? [* * ** * * ** ****]

```
#include <iostream>

void func(const int n) {
  if (n > 1) {
    func(n / 2);
    func(n / 2);
  }

  for (int i = 0; i < n; ++i)
    std::cout << "*";
  std::cout << " ";
}

int main() {
  int n = 4;
  func(4);
  return 0;
}
```

Aufgabe

Schreibe ein Programm, welches eine natürliche Zahl einliest und diese als Binärzahl ausgibt. Es müssen keine zusätzlichen Nullen ausgegeben werden.

Verwende Rekursion.

Lösungsvorschlag

```
#include <iostream>

void convert(unsigned int n) {
  if (n > 0) {
    convert(n / 2);
    if (n % 2 == 1)
      std::cout << "1";
    else
      std::cout << "0";
  }
}

int main() {
  unsigned int n;
  std::cin >> n;
  convert(n);
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Structs

Verwende structs, um Objekte zu definieren. Für diese Objekte können anschliessend auch Operationen definiert werden.

Beispiel:

```
struct vec {
  double x;
  double y;
  double z;
};

// POST: returns the sum of two vectors
vec sum(const vec& a, const vec& b) {
  vec tmp;
  tmp.x = a.x + b.x;
  tmp.y = a.y + b.y;
  tmp.z = a.z + b.z;
  return tmp;
}
```

Structs sind default public und der Zugriff auf eine Variable des structs wird durch structname.variablename ermöglicht.

Beispiel

Definiere ein struct matrix für eine 2x2-Matrix. Definiere anschliessend die Summe zweier Matrizen, die Determinante und die Inverse. Schreibe ausserdem eine Funktion, um die Matrix auszugeben.

Hinweise:
Für 2x2-Matrizen gelten:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$det(A) = ad - cb$$

$$A^{-1} = \frac{1}{det(A)} \cdot \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Lösungsvorschlag (Teil 1)

```
struct matrix {
  double a;
  double b;
  double c;
  double d;
};

matrix sum(const matrix& m1, const matrix& m2) {
  matrix tmp;
  tmp.a = m1.a + m2.a;
  tmp.b = m1.b + m2.b;
  tmp.c = m1.c + m2.c;
  tmp.d = m1.d + m2.d;
  return tmp;
}

double det(const matrix& m) {
  return m.a * m.d - m.c * m.b;
}
```

Lösungsvorschlag (Teil 2)

```
matrix inverse(const matrix& m) {
  matrix tmp;
  double mult = 1 / det(m);
  tmp.a = mult * m.d;
  tmp.b = -mult * m.b;
  tmp.c = -mult * m.c;
  tmp.d = mult * m.a;
  return tmp;
}

void print(const matrix& mat) {
  std::cout << mat.a << " " << mat.b << "\n" << mat.c << "
    " << mat.d << "\n";
}
```

Lösungsvorschlag (Teil 3)

```
int main() {
  matrix mat = { 1, 2, 3, 4 };
  print(sum(mat, mat));
  std::cout << det(mat) << "\n";
  print(inverse(mat));
  return 0;
}
```

Operator Overloading

Zwei Funktionen können denselben Namen haben, wenn

- eine unterschiedliche Anzahl Argumente haben.
- das Argument einen unterschiedlichen Typ hat.

Zwei Funktionen können **nicht** denselben Namen haben, wenn

- die Variablennamen unterschiedlich sind.
- einen unterschiedlichen Funktionstyp haben.

Beispiel

```
#include <iostream>
#include <vector>

struct vec {
  double x;
  double y;
  double z;
};

void print(const vec& v) {
  std::cout << v.x << "\n" << v.y << "\n" << v.z << "\n";
}

void print(const std::vector<int> v) {
  for (unsigned int i = 0; i < v.size(); ++i)
    std::cout << v.at(i) << "\n";
}
```

Beispiel

```
int main() {
  vec vector = { 1,2,3 };
  std::vector<int> standard_vector = { 4,5,6 };
  print(standard_vector);
  std::cout << "\n";
  print(vector);
  return 0;
}

/*
OUTPUT:
1
2
3

4
5
6
*/
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Pointer

Bei Pointer kann man sowohl den Wert der Variabel, auf welche der Pointer zeigt, ändern, als auch den Ort, wo der Pointer hinzeigt. Dazu benötigen wir das Konzept von Adressen (normalerweise 64-bit integers).

Pointer

Bei Pointer kann man sowohl den Wert der Variabel, auf welche der Pointer zeigt, ändern, als auch den Ort, wo der Pointer hinzeigt. Dazu benötigen wir das Konzept von Adressen (normalerweise 64-bit integers).

Es gibt zwei Operatoren, die wir für die Implementierung von Pointer brauchen. Den Adressoperator & und den Dereferenzieroperator *.

Pointer

Bei Pointer kann man sowohl den Wert der Variabel, auf welche der Pointer zeigt, ändern, als auch den Ort, wo der Pointer hinzeigt. Dazu benötigen wir das Konzept von Adressen (normalerweise 64-bit integers).

Es gibt zwei Operatoren, die wir für die Implementierung von Pointer brauchen. Den Adressoperator & und den Dereferenzieroperator *.

Beispiele:

```
//Initialisiere einen Pointer
int* pointer = nullptr; // leerer Pointer
int x = 2;
int* p = &x;
std::cout << "Wert von *p: ";
std::cout << *p << "\n";
```

Arrays

Arrays sind zusammenhängende Speicherblöcke, die wie folgt erstellt werden:

```
//Alloziere Speicherblock (Array)
int* array2 = new int[7]; // leer
int* array3 = new int[7]{ 1,2,3,4,5,6,7 }; // mit Werten
```

Das stellen wir uns so vor:

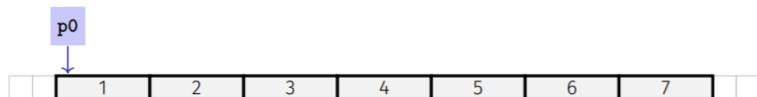


Abbildung: aus Vorlesungsfolien IFMP

Pointer Arithmetic

Folgende Operationen sind für Pointer zulässig:

```
int* array = new int[7]{ 1,2,3,4,5,6,7 };
int* p = nullptr;

// Pointer umhaengen
p = array;

// Pointer dereferenzieren
int x = *p;
std::cout << *array << "\n";

// Pointer inkrementieren
++p;
std::cout << ++array << "\n";

// Pointer vergleichen
std::cout << p == array << "\n"; // Zeigen sie an
  denselben Ort?
```

Pointer Arithmetic

```
// Distanz zwischen zwei Pointer
unsigned int size = 2;
int* begin = new int[size]{ 1,2 };
int* end = begin + size;
std::cout << "Laenge des Arrays: " << end - begin << "\n";

// const
int a = 2;
int b = 3;
const int* ptr = &a; // kein Schreibzugriff auf Wert am
  Zielort
int* const ptr2 = &b; // kein Schreibzugriff auf Zeiger
  (Zeiger kann nicht umgehaengt werden)
```

Pointer und Arrays: Beispiel

Schreibe ein Programm, welches folgende Funktionalitäten besitzt:

- `output_array(args)`: Ausgabe eines Arrays
- `fill(args)`: ein Array mit einem Wert füllen (z.B. an jeder Stelle steht eine 2)
- `fill(args)`: ein Array mit Werten füllen (an jeder Stelle steht ein Wert, der über die Konsole eingegeben wird)

Dabei ist die Funktion `fill` zu überladen.
Überlege dir bei jedem Pointer, ob der Wert und/oder der Zeiger konstant sein sollen.

Lösungsvorschlag (Teil 1)

```
#include <iostream>

void fill(int* const begin, int* const end, int value) {
  for (int* p = begin; p != end; ++p)
    *p = value;
}

void fill(int* const begin, int* const end) {
  std::cout << "Please fill your array of size " << end -
    begin << "\n";
  for (int* p = begin; p != end; ++p) {
    int value;
    std::cin >> value;
    *p = value;
  }
}
```

Lösungsvorschlag (Teil 2)

```
void output(const int* const begin, const int* const end) {
  for (const int* p = begin; p != end; ++p)
    std::cout << *p << " ";
  std::cout << "\n";
}

int main() {
  int* array = new int[7]{ 1,2,3,4,5,6,7 }; // mit Werten
  int* array2 = new int[3]; // leeres Array
  int* array3 = new int[4]; // leeres Array

  // Fuelle leeres Array
  fill(array2, array2 + 3, 2);
  fill(array3, array3 + 4);

  // Output Arrays
  output(array, array + 7);
  output(array2, array2 + 3);
  output(array3, array3 + 4);
  return 0;
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Üben wir ein bisschen...

```
void func_a(int* start, int* end) {
    for(int* it = start; it != end; ++it)
        std::cout << *it << " ";
}
```

(a) Was gibt das Programm mit folgender main-Funktion aus.
What is the output of the program with the following main-function.

2 P

```
int main() {
    int src[] = { 1, 2, 3 };
    int dst[8];
    for (int i = 7; i >= 0; --i)
        dst[i] = src[i % 3];
    func_a(dst + 1, dst + 7);
    return 0;
}
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

Lösung

(a) Was gibt das Programm mit folgender `main`-Funktion aus.
What is the output of the program with the following `main`-function.

2 P

```
int main() {
    int src[] = { 1, 2, 3 };
    int dst[8];
    for (int i = 7; i >= 0; --i)
        dst[i] = src[i % 3];
    func_a(dst + 1, dst + 7);
    return 0;
}
```

```
2 3 1 2 3 1
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

Üben wir ein bisschen...

```
void func_b(int &a, int b, int& c) {
    if (b > 0)
        a += b * c;
    c *= 2;
    std::cout << a;
}
```

(b) Was gibt das Programm mit folgender `main`-Funktion aus.
What is the output of the program with the following `main`-function.

3 P

```
int main() {
    char values[8] = { 1, 0, 0, 1, 0, 1, 0, 1 };
    int res = 0;
    int x = 1;
    for(int i = 0; i < 8; ++i)
        func_b(res, values[i], x);
    return 0;
}
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

Lösung

(b) Was gibt das Programm mit folgender `main`-Funktion aus.
What is the output of the program with the following `main`-function.

3 P

```
int main() {
    char values[8] = { 1, 0, 0, 1, 0, 1, 0, 1 };
    int res = 0;
    int x = 1;
    for(int i = 0; i < 8; ++i)
        func_b(res, values[i], x);
    return 0;
}
```

```
1 1 1 9 9 41 41 169
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

Üben wir ein bisschen...

```
void func_c(const int nr) {
    if (nr > 1) {
        func_c(nr / 2);
        func_c(nr / 2);
    }

    for(int i = 0; i < nr; ++i)
        std::cout << "*";
    std::cout << " ";
}
```

(c) Was gibt das Programm mit folgender `main`-Funktion aus.
What is the output of the program with the following `main`-function.

3 P

```
int main() {
    func_c(4);
    return 0;
}
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

Lösung

(c) Was gibt das Programm mit folgender `main`-Funktion aus.
What is the output of the program with the following `main`-function.

3 P

```
int main() {
    func_c(4);
    return 0;
}
```

```
* * ** * * ** ****
```

Abbildung: Aufgabe 3, Prüfung vom Januar 2016

1. Aufgabentyp I: Typen und Werte
2. Aufgabentyp II: Fliesskommazahlen
3. Aufgabentyp III: Konstrukte
   - Loops
   - Vektoren und Strings
   - Funktionen und Referenzen
   - Rekursion
   - Structs und Operator Overloading
   - Pointer und Arrays
   - Übungen
4. **Aufgabentyp IV: EBNF**
5. Aufgabentyp V: Programmieraufgaben
   - Klassen
   - Dynamische Datenstrukturen und Memory Management
   - Linked Lists und Bäume

Aufgabentyp IV: EBNF

Aufgabe 3: EBNF (8P)

Die folgende EBNF definiert gewisse Ausdrücke in einer Prefixschreibweise.
Beantworten Sie die nachfolgenden Fragen.

Anmerkung: Leerschläge sind im Rahmen der EBNF bedeutungslos.

The following EBNF defines certain expressions in a prefix notation.
Answer the questions below.

***Remark**: Whitespaces are irrelevant in the context of this EBNF.*

```
Expression = Operator Operand ',' Operand.
Operator   = '+' | '-' | '/' | '*'.
Operand    = Number | Expression.
Number     = Integer | Integer '.' Integer.
Integer    = Digit {Digit}.
Digit      = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

/6P (a) Geben Sie in folgender Matrix an, ob die Zeichenkette einem gültigen Ausdruck (Expression) gemäss der EBNF entspricht oder nicht. Falsche Antworten ergeben keine negativen Punkte.

Provide in the following matrix if the string corresponds to a valid expression according to the EBNF. Wrong answers do not result in negative points.

Zeichenkette *String*	gültig *valid*	ungültig *invalid*
+ + 3 , 5 , 6		
+ (3.5 , 2)		
+ * 3.3 , + 5 , 6 , 2.2, 1.1		
3 - * 5 , 6		
+ * 3 , 5 , * 6.2 , 0.7		
+ 3.7 - 4.7		

/2P (b) Ändern Sie genau eine Produktionsregel der EBNF ab, so dass folgender Ausdruck (expression) gültig wird.

Modify one and only one production rule of the EBNF such that the following expression becomes valid.

```
( ( 3.5 + 6) * ( 2 + 7.2 ) )
```

Geänderte Zeile / *Modified line*:

Abbildung: Aufgabe 3, Prüfung vom Februar 2018

Aufgabentyp IV: EBNF

- VL: Kap. 15 EBNF
- Themen: EBNF (und Rekursion)
- Mögliche Teilaufgaben
  1. Verständnisteil (Gratispunkte, immer untersuchen)
  2. EBNF Regeln schreiben (immer versuchen)
  3. Programmierteil (nicht immer dabei)
- Das musst du dafür wissen:
  1. Wie liest man EBNF? (für Verständnisteil wichtig)
  2. Wie schreibt man EBNF?
  3. Wie implementiert man EBNF? (für Programmierteil wichtig)

Wie liest man EBNF?

- | bedeutet „oder“
- Ausdrücke in geschweiften Klammern dürfen beliebig oft wiederholt werden (auch 0 Mal)
- Ausdrücke in eckigen Klammern dürfen 0 oder 1 Mal wiederholt werden
- Alles, was in Gänsefüsschen steht, ist ein Zeichen der Zeichenfolge.
  ACHTUNG: $\{ \neq "\{"$
- Ausdrücke, die **nicht** in Gänsefüsschen stehen, sind Aufrufe von anderen Zeilen.
- Leerschläge sind bedeutungslos.

Beispiel: EBNF verstehen

4 BNF I *(6 Punkte)*

Die folgende BNF definiert eine Sprache zur Steuerung einer Schildkröte für Turtle-Grafiken. Das Kommando (ein *Command* der BNF) `X 10 [R 5 F 8]` steht zum Beispiel für die zehnfache Repetition von: Drehung nach rechts um 5 und Bewegung nach vorne um 8 Einheiten.

Beantworten Sie die Fragen auf der rechten Seite!

Anmerkung: Leerschläge sind im Rahmen der BNF bedeutungslos.

```
Command = Movement | Repetition.
Movement = "F" Number | "L" Number | "R" Number.
Repetition = "X" Number "[" Commands "]".
Commands = Command | Command Commands.
Number = unsigned int.
```

Abbildung: Aufgabe 4, Prüfung vom Januar 2016

Beispiel: EBNF verstehen

```
Command = Movement | Repetition.
Movement = "F" Number | "L" Number | "R" Number.
Repetition = "X" Number "[" Commands "]".
Commands = Command | Command Commands.
Number = unsigned int.
```

Welche Ausdrücke sind gemäss dieser EBNF gültig?

- F 4
- X 10 [F 4 L 3 R 3 F 5]
- X 10 []
- X 10 F 4
- X 10 [F 4 X 10 [F 4] L 3]

Lösung: EBNF verstehen

```
Command = Movement | Repetition.
Movement = "F" Number | "L" Number | "R" Number.
Repetition = "X" Number "[" Commands "]".
Commands = Command | Command Commands.
Number = unsigned int.
```

Welche Ausdrücke sind gemäss dieser EBNF gültig?

- F 4 **gültig**
- X 10 [F 4 L 3 R 3 F 5] **gültig**
- X 10 [] **ungültig**
- X 10 F 4 **ungültig**
- X 10 [F 4 X 10 [F 4] L 3] **gültig**

Wie schreibt man EBNF?

Vorgehen:

1. Definiere verschiedene Teile deines Ausdrucks (Funktionen) und schreibe diese untereinander.
2. Überlege dir, wie die verschiedenen Ausdrücke voneinander abhängen und verschachtelt sind.
3. Schreibe die entsprechenden Regeln hin, indem du die erlaubten Zechen verwendest.
4. Setzte am Ende jeder Zeile einen Punkt. **nicht vergessen!**
5. Nimm ein Beispiel und überprüfe deine EBNF.

Beispiel: EBNF schreiben

Definiere eine EBNF zur Beschreibung der Bedienung eines einfachen Verkaufsautomaten:
Der Kunde kommt an ("arrive"), zahlt Geld ("coin"), fordert Schokolade ("choc") oder Getränk ("drink") oder möchte das eingeworfene Geld zurück ("reject") und verlässt die Maschine wieder ("leave").
Schokolade kostet eine Münze, ein Getränk zwei.

Beispiele gültiger Sequenzen:

- arrive coin choc leave
- arrive coin coin drink leave
- arrive coin reject leave
- arrive coin choc coin choc coin coin drink leave

Lösung: EBNF schreiben

```
S            = "arrive" Use.
Use          = "coin" SingleCredit | "leave".
SingleCredit = "choc" Use | "reject" Use | "coin" DoubleCredit.
DoubleCredit = "drink" Use | "reject" Use.
```

Abbildung: Aufgabe 5, Prüfung vom Februar 2017

Wie implementiert man EBNF?

In der Regel gibt es drei mögliche Fälle:

1. Rekursiver Aufruf der Funktion oder Aufruf einer anderen Funktion
2. Aufruf von consume oder has
3. Aufruf von lookahead

Hinweise:

- Dieses Thema kann man sich durch Üben (mind. teilweise) gut aneignen.
- Wenn ihr bei diesem Thema nicht sattelfest seid, aber noch Zeit habt, ratet hier.

Wie implementiert man EBNF?

Fall 1: Rekursiver Aufruf der Funktion oder Aufruf einer anderen Funktion

Idee: Aufgerufene Funktion soll prüfen, ob die Zeichenfolge, die gerade kommt, ein XY ist.

Beispiel: (aus Trains)

```
bool open(std::istream& is) {
  // open = loco cars .
  return loco(is) && cars(is);
}
```

Bemerkung: Die Reihenfolge wird auch geprüft, da die Auswertung einer Ausdrucks mit && von links nach rechts erfolgt.

Wie implementiert man EBNF?

Fall 2: Aufruf von consume oder has

Idee: Es wird ein bestimmtes Zeichen erwartet und consume prüft, ob dieses Zeichen kommt. (Genau lesen, wann und ob das Zeichen gelesen wird oder nicht!)

Beispiel: (aus Trains)

```
bool loco(std::istream& is) {
  // loco = "*" | "*" loco
  bool valid = consume(is, '*');
```

Wie implementiert man EBNF?

Fall 3: Aufruf von lookahead

Idee: Es gibt zwei Möglichkeiten und man will eine Fallunterscheidung machen. Man ruft lookahead auf, um zu wissen, was für ein Zeichen als nächstes kommt, denn lookahead gibt diesen char zurück. D.h. man kann diesen char vergleichen, um die Fallunterscheidung umzusetzen.

Beispiel: (aus Trains)

```
bool loco(std::istream& is) {
  // loco = "*" | "*" loco
  bool valid = consume(is, '*');

  if (lookahead(is) == '*') {
    valid = valid && loco(is);
  }
  return valid;
}
```

Beispiel: EBNF implementieren

4 BNF I *(6 Punkte)*

Die folgende BNF definiert eine Sprache zur Steuerung einer Schildkröte für Turtle-Grafiken. Das Kommando (ein *Command* der BNF) `X 10 [R 5 F 8]` steht zum Beispiel für die zehnfache Repetition von: Drehung nach rechts um 5 und Bewegung nach vorne um 8 Einheiten.

Beantworten Sie die Fragen auf der rechten Seite!

Anmerkung: Leerschläge sind im Rahmen der BNF bedeutungslos.

```
Command = Movement | Repetition.
Movement = "F" Number | "L" Number | "R" Number.
Repetition = "X" Number "[" Commands "]".
Commands = Command | Command Commands.
Number = unsigned int.
```

Abbildung: Aufgabe 4, Prüfung vom Januar 2016

Öffne ein leeres Projekt in code expert. Lösche das main.cpp File und füge das file turle.cpp ein. Ergänze die Lücken.

Lösungsvorschlag (Teil 1)

```
bool Command(std::istream& is) {
  // Command = Movement | Repetition.
  if (lookahead(is) == 'X') return Repetition(is);
  else return Movement(is);
}

bool Movement(std::istream& is) {
  // Movement = "F" Number | "L" Number | "R" Number.
  if (has(is, 'F') || has(is, 'L') || has(is, 'R'))
    return Number(is);
  return false;
}

bool Repetition(std::istream& is) {
  // Repetition = "X" Number "[" Commands "]".
  if (has(is, 'X'))
    return Number(is) && has(is, '[') && Commands(is) &&
   has(is, ']');
  return false;
}
```

Lösungsvorschlag (Teil 2)

```
bool Commands(std::istream& is) {
  // Commands = Command | Command Commands
  if (!Command(is))
    return false;
  if (lookahead(is) == ']')
    return true;
  return Commands(is);
}

bool Number(std::istream& is) {
  // Number = unsigned int.
  int temp;
  is >> temp;
  if (temp < 0)  // unsigned int only
    return false;
  if (is.fail() || is.bad()) // numbers only
    return false;
  return true;
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Von struct zu class

- Unterschied liegt in der default protection.
- class ohne zusätzliche Angabe ist private und struct public.
- private Variablen einer Klasse sind nur durch Memberfunktionen zugänglich.
- public Variablen einer Klasse sind für alle Funktionen zugänglich.

Von struct zu class: Beispiel

```cpp
struct matrix {
  double a; double b;
  double c; double d;
};

matrix sum(const matrix& m1, const
     matrix& m2) {
  matrix tmp;
  tmp.a = m1.a + m2.a;
  // [...]
  return tmp;
}

double det(const matrix& m) {
  return m.a * m.d - m.c * m.b;
}

matrix inverse(const matrix& m) {
  matrix tmp;
  double mult = 1 / det(m);
  tmp.a = mult * m.d;
  tmp.b = -mult * m.b;
  // [...]
  return tmp;
}

void print(const matrix& mat) {
  std::cout << mat.a << " " <<
     /*[...]*/;
}
```

```cpp
class matrix {
  double a; double b;
  double c; double d;
public:
  matrix() : a(0), b(0), c(0), d(0) {}
  matrix(int A, int B, int C, int D) :
     a(A), b(B), c(C), d(D) {}

  matrix sum(const matrix& m2) const {
    matrix tmp;
    tmp.a = this->a + m2.a;
    // [...]
    return tmp;
  }
  double det() const {
    return this->a * this->d - this->c
     * this->b;
  }
  matrix inverse() const {
    matrix tmp;
    double mult = 1 / det();
    tmp.a = mult * this->d;
    tmp.b = -mult * this->b;
    // [...]
    return tmp;
  }
  void print() const {
    std::cout << this->a << /*[...]*/;
  }
};
```

Vergleich im Detail

Als erstes fällt auf, dass die Funktionen zur matrix jetzt in der class mit drin sind.

Wieso?

Vergleich im Detail

Als erstes fällt auf, dass die Funktionen zur matrix jetzt in der class mit drin sind.

Wieso?

Die Variablen a bis d sind in der class Umgebung nicht mehr öffentlich, sondern private. Und auf die privaten Variablen können nur Funktionen, die Teil der Klasse sind, zugreifen. Deshalb müssen wir diese Funktionen in die class Umgebung hinein schreiben.

Memberfunktionen

Eine Memberfunktion ist eine Funktion, die Teil einer Klasse ist. Die Memberfunktion lässt sich zusammen mit einem Objekt derselben Klasse aufrufen.

Memberfunktionen

Eine Memberfunktion ist eine Funktion, die Teil einer Klasse ist. Die Memberfunktion lässt sich zusammen mit einem Objekt derselben Klasse aufrufen.

```
class matrix {
  double a; double b;
  double c; double d;
public:
  matrix() : a(0), b(0), c(0), d(0) {}
  matrix(int A, int B, int C, int D) : a(A), b(B), c(C),
    d(D) {}

  matrix sum(const matrix& m2) const {
    matrix tmp;
    tmp.a = this->a + m2.a;
    // [...]
    return tmp;
  }
```

Beispiel für eine Memberfunktion: `matrix sum(const matrix& m2) const{}`

Vergleich im Detail

Was weiter ins Auge sticht, sind folgende beiden Zeilen:

```
matrix() : a(0), b(0), c(0), d(0) {}
matrix(int A, int B, int C, int D) : a(A), b(B), c(C),
  d(D) {}
```

Was ist das und wofür brauchen wir das?

Vergleich im Detail

Was weiter ins Auge sticht, sind folgende beiden Zeilen:

```
matrix() : a(0), b(0), c(0), d(0) {}
matrix(int A, int B, int C, int D) : a(A), b(B), c(C),
  d(D) {}
```

Was ist das und wofür brauchen wir das?

Das sind Konstruktoren. Sie legen fest, wie unser Objekt matrix initialisiert werden soll. structs können (v.a. wenn sie komplexer sind) auch Konstruktoren haben.

Konstruktoren

Konstruktoren sind spezielle Memberfunktionen. Ihre Aufgabe ist es, sicherzustellen, dass die Objekte der Klasse korrekt **initialisiert** werden. So werden undefined values vermieden. Konstruktoren tragen den Namen der Klasse.

Konstruktoren

Konstruktoren sind spezielle Memberfunktionen. Ihre Aufgabe ist es, sicherzustellen, dass die Objekte der Klasse korrekt **initialisiert** werden. So werden undefined values vermieden. Konstruktoren tragen den Namen der Klasse.

Normalerweise gibt es zwei Arten von Konstruktoren:

1. default constructor (für leere Elemente)
2. Konstruktor für Elemente mit Werten

Konstruktoren

Konstruktoren sind spezielle Memberfunktionen. Ihre Aufgabe ist es, sicherzustellen, dass die Objekte der Klasse korrekt **initialisiert** werden. So werden undefined values vermieden. Konstruktoren tragen den Namen der Klasse.

Normalerweise gibt es zwei Arten von Konstruktoren:

1. default constructor (für leere Elemente)
2. Konstruktor für Elemente mit Werten

```
class matrix {
  double a; double b;
  double c; double d;
public:
  matrix() : a(0), b(0), c(0), d(0) {}
  matrix(int A, int B, int C, int D) : a(A), b(B), c(C),
    d(D) {}
```

Beispiel für einen default Konstruktor: `matrix()`

Beispiel für einen Konstruktor mit Werten: `matrix(int A, int B, ...)`

Aufbau einer Klasse

```
class classname
```

Aufbau einer Klasse

```
class classname
  private
```

Aufbau einer Klasse

```
class classname
  private

  public
```

Aufbau einer Klasse

```
class classname
  private

  public
    Konstruktoren
```

Aufbau einer Klasse

class classname

private

public

Konstruktoren

Memberfunktionen

Vergleich Struktur Funktion vs. Memberfunktion

struct

```
matrix sum(const matrix& m1,
    const matrix& m2) {
  matrix tmp;
  tmp.a = m1.a + m2.a;
  // [...]
  return tmp;
}
```

class

```
matrix sum(const matrix&
  m2) const {
  matrix tmp;
  tmp.a = this->a + m2.a;
  // [...]
  return tmp;
}
```

Vergleich Struktur Funktion vs. Memberfunktion

struct

```
matrix sum(const matrix& m1,
    const matrix& m2) {
  matrix tmp;
  tmp.a = m1.a + m2.a;
  // [...]
  return tmp;
}
```

`matrix sum(const matrix& m1, const matrix& m2)` ist eine normale Funktion und hat 2 Argumente.
Da jedes Argument ein struct ist, wird auf die Variablen desselben mittels `objectname.varname` zugegriffen.

class

```
  matrix sum(const matrix&
    m2) const {
    matrix tmp;
    tmp.a = this->a + m2.a;
    // [...]
    return tmp;
  }
```

`matrix sum(const matrix& m2) const` ist eine Memberfunktion und hat ebenfalls 2 Argumente, wobei jedoch nur 1 explizit übergeben werden muss. Auf die Variablen des übergebenen Argumentes `m2` kann wie bekannt mittels `m2.a` etc. zugegriffen werden.
Das Argument, das nicht explizit übergeben werden muss, ist das sogenannte **implizite** Argument. Auf dessen private Variablen kann mittels `this->varname` (oder hier auch möglich: `varname`).
`this->varname = (*this).varname`

Das implizite Argument

Eine Memberfunktion kann direkt durch `this->var` auf die privaten Variablen des impliziten Arguments zugreifen. Eine Nicht-Memberfunktion kann das nicht! Die übrigen Argumente werden wie bisher normal übergeben und der Zugriff erfolgt analog wie wir es bei structs gelernt haben.

```
double det() const {
  return this->a * this->d - this->c * this->b;
}

matrix inverse() const {
  matrix tmp;
  double mult = 1 / det();
  tmp.a = mult * this->d;
  tmp.b = -mult * this->b;
  tmp.c = -mult * this->c;
  tmp.d = mult * this->a;
  return tmp;
}
```

Memberfunktionen und const

Um das implizite Argument als read-only an eine Memberfunktion zu übergeben, schreibe `const` vor die geschweiften Klammern:

```
double det() const {
  return this->a * this->d - this->c * this->b;
}
```

Vergleich Aufrufe in main()

struct

```
int main() {
  matrix mat = { 1, 2, 3,
    4 };
  print(sum(mat, mat));
  std::cout << det(mat) <<
    "\n";
  print(inverse(mat));
  return 0;
}
```

class

```
int main() {
  matrix mat(1, 2, 3, 4);
  (mat.sum(mat)).print();
  std::cout << mat.det()
    << "\n";
  (mat.inverse()).print();
  return 0;
}
```

Vergleich Aufrufe in main()

struct

```
int main() {
  matrix mat = { 1, 2, 3,
    4 };
  print(sum(mat, mat));
  std::cout << det(mat) <<
    "\n";
  print(inverse(mat));
  return 0;
}
```

In unserem Beispiel haben wir für den struct keinen Konstruktor definiert, daher erstellt man ein Objekt vom Typ matrix mit den geschweiften Klammern und den Werten.
Die Funktionen werden normal mit ihrem Namen aufgerufen und die Argumente übergeben.

class

```
int main() {
  matrix mat(1, 2, 3, 4);
  (mat.sum(mat)).print();
  std::cout << mat.det()
    << "\n";
  (mat.inverse()).print();
  return 0;
}
```

Wir haben einen Konstruktor definiert und dessen Struktur muss beim Erstellen eines Objektes vom Typ matrix berücksichtigt werden. Daher wird mat für eine Matrix mit Werten mit den runden Klammern und den Werten initialisiert.
Die Memberfunktionen werden mit `objectname.funcname` aufgerufen und es werden nur die nicht-impliziten Argumente übergeben. Beachte die spezielle Schreibweise bei Verschachtelungen.

Et voilà: die Umwandlung von struct zu class ist vollbracht!

```
class matrix {
  double a; double b;
  double c; double d;
public:
  matrix() : a(0), b(0), c(0), d(0) {}
  matrix(int A, int B, int C, int D) : a(A), b(B), c(C), d(D) {}

  matrix sum(const matrix& m2) const {
    matrix tmp;
    tmp.a = this->a + m2.a;
    // [...]
    return tmp;
  }
  double det() const {
    return this->a * this->d - this->c * this->b;
  }
  matrix inverse() const {
    matrix tmp;
    double mult = 1 / det();
    tmp.a = mult * this->d;
    tmp.b = -mult * this->b;
    // [...]
    return tmp;
  }
  void print() const {
    std::cout << this->a << /*[...]*/;
  }
};
int main() {
  matrix mat(1, 2, 3, 4);
  (mat.sum(mat)).print();
  (mat.inverse()).print();
```

Deklaration und Definition splitten

Nun möchten wir unsere Klasse auch anderen zugänglich machen. Da wir den Nutzern nicht unbedingt unseren Quellcode weitergeben möchten, splitten wir die Deklaration und die Definition unserer Klasse in zwei Dokumente auf. Die Deklarationen kommen ins .h file und die Definitionen ins .cpp file. Der Nutzer soll dann nur das .h file lesbar erhalten.

Deklaration und Definition splitten

.h file

```
class matrix {
  double a;
  double b;
  double c;
  double d;

public:
  matrix();
  matrix(int A, int B,
    int C, int D);

  matrix sum(const
    matrix& m2) const;

  double det() const;

  matrix inverse()
    const;

  void print() const;
};
```

.cpp file

```
#include <iostream>
#include "class_kurz_split.h"

matrix::matrix() : a(0), b(0), c(0), d(0) {}
matrix::matrix(int A, int B, int C, int D) :
     a(A), b(B), c(C), d(D) {}

matrix matrix::sum(const matrix& m2) const {
  matrix tmp;
  tmp.a = this->a + m2.a;
  // [...]
  return tmp;
}
double matrix::det() const {
  return this->a * this->d - this->c * this->b;
}
matrix matrix::inverse() const {
  matrix tmp;
  double mult = 1 / det();
  tmp.a = mult * this->d;
  tmp.b = -mult * this->b;
  // [...]
  return tmp;
}
void matrix::print() const {
  std::cout << this->a << /*[...]*/;
}
```

Deklaration und Definition splitten

Beachte:

- im .h file stehen nur Deklarationen der Klasse
- das .h file muss sowohl ins main file als auch ins .cpp file mittels `#include "classname.h"`eingebunden werden.
- im .cpp file stehen alle Definitionen der Memberfunktionen (und Nicht-Memberfunktionen).
- Alle Memberfunktionen brauchen im .cpp file beim Namen einen Zusatz (damit klar ist, dass sie Memberfunktionen sind). D.h. alle Funktionen heissen im .cpp file `classname::funcname`.

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Container und Iteratoren

Container und Iteratoren

Beispiel:

```
#include <iostream>
#include <set>

int main() {
  std::set<int> cont = { 8,3,1,4,6,9 };
  for (std::set<int>::iterator it = cont.begin();
    it != cont.end(); ++it) {
    std::cout << *it << " ";
  }
  return 0;
}
```

Memory Management: new

new T(...)	Speicher für ein neues Objekt vom Typ T wird alloziert. Wert: Adresse des neuen T-Objekt Typ: Zeiger T* Objekte, die mit new erzeugt worden sind, haben dynamische Speicherdauer: sie "leben", bis sie explizit gelöscht werden.
new T[expr]	Neuer zusammenhängender Bereich im Speicher für n Elemente vom Typ T wird alloziert. expr ist vom Typ int mit Wert n.

Memory Management: delete

delete expr	Objekt wird dekonstruiert und Speicher freigegeben. expr ist ein Zeiger vom Typ T*, der auf ein vorher mit new erzeugtes Objekt zeigt.
delete [] expr	Array wird gelöscht und Speicher freigegeben. expr ist ein Zeiger vom Typ T*, der auf ein vorher mit new erzeugtes Array verweist.

Richtlinie "Dynamischer Speicher":
Zu jedem new gibt es ein passendes delete!

Destruktor

Der Destruktor einer Klasse T ist die eindeutige Memberfunktion mit Deklaration ∼T ( );

Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts vom Typ T endet, z.B. bei Aufruf von delete auf einem Objekt vom Typ T* oder wenn der Gültigkeitsbereich eines Objektes vom Typ T endet.

Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf.

Copy-Konstruktor

Der Copy-Konstruktor einer Klasse T ist der eindeutige Konstruktor mit Deklaration T (const T& varname);

Wird automatisch aufgerufen, wenn Werte vom Typ T mit Werten vom Typ T initialisiert werden:
T varname = t; (t vom Typ T) oder T varname (t);

Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise).

Beispiel: Box

```
class Box {
  int* ptr;
public:
  // POST: create a Box containing the same things as other.
  Box(const Box& other);
  // POST: assign other to this
  Box& operator=(const Box& other);
  // POST: Destroy the box and its contents.
  ~Box();
  // PRE: v point to an object allocated by new.
  //      Deallocation responsibility of v has not been
    given to anyone yet.
  // POST: create a Box containing the object pointed by v.
  //       Give deallocation responsibility to the Box.
  Box(int* v);
  // POST: Access the value.
  int& value();
};
```

Beispiel: Box

```
#include <iostream>
#include "box.h"

Box::Box(const Box& other) {
  ptr = new int(*other.ptr);
  std::cerr << "copy constructor: value=" << *ptr << " at " << ptr << std::endl;
}

Box& Box::operator= (const Box& other) {
  *ptr = *other.ptr;
  std::cerr << "assignment operator: value=" << *ptr << " at " << ptr << std::endl;
  return *this;
}

Box::~Box() {
  std::cerr << "destructor: value=" << *ptr << " at " << ptr << std::endl;
  delete ptr;
  ptr = nullptr;
}

Box::Box(int* v) {
  ptr = v;
  std::cerr << "overloaded constructor: value=" << *ptr << " at " << ptr << std::endl;
}

int& Box::value() {
  return *ptr;
}
```

1 Aufgabentyp I: Typen und Werte

2 Aufgabentyp II: Fliesskommazahlen

3 Aufgabentyp III: Konstrukte
- Loops
- Vektoren und Strings
- Funktionen und Referenzen
- Rekursion
- Structs und Operator Overloading
- Pointer und Arrays
- Übungen

4 Aufgabentyp IV: EBNF

5 Aufgabentyp V: Programmieraufgaben
- Klassen
- Dynamische Datenstrukturen und Memory Management
- Linked Lists und Bäume

Linked Lists

Eine Linked List besteht in der Regel aus aneinandergereihten Knoten. Einen simplen Knoten können wir z.B. wie folgt darstellen:

```
struct Node {
  int value;
  Node* next;

  Node(int v) : value(v), next(nullptr) {}
};

int main() {
  Node* newNode = new Node(2);
  return 0;
}
```

Linked List

Reihen wir die Knoten aneinander, erhalten wir eine simple Linked List:

```
class linkedlist {
  Node* head;

public:
  linkedlist();

  void addNode(Node* newNode);
};
```

Ergänze die Funktion `void addNode(Node* newNode)`

void addNode(Node* newNode)

```
void linkedlist::addNode
    (Node* newNode) {
  // if list is empty
  if (/*TODO*/)
    /*TODO*/
  // list is not empty
  else {
    // Define two helping
    pointers
    Node* left = nullptr;
    Node* right = head;
    // As long as right
    hasn't reached the end of
    the list
    while (/*TODO*/) {
      if (left != nullptr)
        assert(left->next ==
    right);
      // Compare values
      if (/*TODO*/) {
        // If newNode should
    be the first node
```

```
          if (/*TODO*/)
            /*TODO*/
          // newNode is not the
    first node
          else
            // insert newNode
    after left node
            /*TODO*/
          // connect newNode
    with right node
          /*TODO*/
          break;
      }
      // iterations
      /*TODO*/
    }
    // if newNode should be
    at the end
    if (right == nullptr) {
      /*TODO*/
}}}
```

Lösung

```
void linkedlist::addNode(Node* newNode)
    {
  // if list is empty
  if (head == nullptr)
    head = newNode;
  // list is not empty
  else {
    // Define two helping pointers
    Node* left = nullptr;
    Node* right = head;
    // As long as right hasn't reached
     the end of the list
    while (right != nullptr) {
      if (left != nullptr)
        assert(left->next == right);
      // Compare values
      if (newNode->value <=
     right->value) {
        // If newNode should be the
     first node
        if (left == nullptr)
          head = newNode;
```

```
          // newNode is not the first node
          else
            // insert newNode after left
     node
            left->next = newNode;
          // connect newNode with right
     node
          newNode->next = right;
          break;
       }
       // iterations
       left = right;
       right = right->next;
     }
     // if newNode should be at the end
     if (right == nullptr) {
       left->next = newNode;
       newNode->next = nullptr;
     }
   }
}
```

++Schwierigkeitsgrad

Jetzt wollen wir Node als class anstatt struct schreiben.
Was müssen wir alles anpassen?

++Schwierigkeitsgrad

Jetzt wollen wir Node als class anstatt struct schreiben.
Was müssen wir alles anpassen?

- in class Node eine Unterscheidung zwischen private und public machen.
- eine Memberfunktion schreiben, um an die private Variable zu kommen.
- files splitten
- in der Klasse linkedlist Zugriff auf private Variablen von Node anpassen (über Memberfunktion anstatt direkt)

Lösung

```
class Node {
  int value;

public:
  Node* next;

  Node(int v);

  int getValue() const;
};
```

```
#include <iostream>
#include "Node_class.h"

Node::Node(int v) : value(v),
    next(nullptr) {}

int Node::getValue() const {
  return this->value;
}
```

In der Memberfunktion `void linkedlist::addNode(Node* newNode)` ändere folgende Ausdrücke:

`newNode->value` ⇒ `newNode->getValue()`

++Schwierigkeitsgrad

Jetzt möchten wir an einem Knoten mehrere Informationen speichern:

- Vorname
- Familienname
- Alter
- Geburtsdatum

Wir können die Grundstruktur der Linked List übernehmen.
Du findest die detaillierte Aufgabenstellung auf Serie 3.

Bäume

Ein Baum ist vom Konzept her sehr ähnlich wie eine Linked List. Man hat wieder Knoten, die man aneinanderreiht. Der Unterschied besteht darin, dass die Knoten des Baums zwei Pointer haben:

```
struct Node {
  int value;
  Node* left;
  Node* right;

  Node(int v);
};
```

class Tree

Wir beginnen mit einem Knoten root. Dieser hat einen Wert value. Wenn man den nächsten Knoten (mit Wert v) einfügen will, vergleicht man die Werte miteinander. Ist v $<$ value, so soll der neue Knoten links von root hinkommen. Falls v $>$ value rechts.

class Tree

Wir beginnen mit einem Knoten root. Dieser hat einen Wert value. Wenn man den nächsten Knoten (mit Wert v) einfügen will, vergleicht man die Werte miteinander. Ist v $<$ value, so soll der neue Knoten links von root hinkommen. Falls v $>$ value rechts.

```
class Tree {
  Node* root;

public:
  Tree();

  Node* minValueNode(Node* root);

  Node* deleteNode(Node* root, const int value);

  void insert(int v);

  void print(Node* tmp) const;

  void print() const;

  void call_deleteNode(unsigned int val);
};
```

class Tree

```
class Tree {
  Node* root;

public:
  Tree();

  Node* minValueNode(Node* root);

  Node* deleteNode(Node* root, const int value);

  void insert(int v);

  void print(Node* tmp) const;

  void print() const;

  void call_deleteNode(unsigned int val);
};
```

Aufgabe: Schreibe Tree.cpp

class Tree

Schreibe den default Konstruktor im .cpp file.

class Tree

Schreibe den default Konstruktor im .cpp file.

```
#include <iostream>
#include <cassert>
#include "Node_tree.h"
#include "Tree_simple.h"

Tree::Tree() : root(nullptr) {}
```

class Tree

Schreibe den default Konstruktor im .cpp file.

```
#include <iostream>
#include <cassert>
#include "Node_tree.h"
#include "Tree_simple.h"

Tree::Tree() : root(nullptr) {}
```

Schreibe die Memberfunktion `MinValueNode(Node* root)` .

class Tree

Schreibe den default Konstruktor im .cpp file.

```
#include <iostream>
#include <cassert>
#include "Node_tree.h"
#include "Tree_simple.h"

Tree::Tree() : root(nullptr) {}
```

Schreibe die Memberfunktion `MinValueNode(Node* root)` .

```
Node* Tree::minValueNode(Node* root) {
  while (root->left != nullptr)
    root = root->left;
  return root;
}
```

deleteNode

```
Node* Tree::deleteNode(Node* root, const int value) {
  assert(root != nullptr);
  // If the value to be deleted is less than the root's
    value,
  // recurse in the left subtree
  if (/*TODO*/)
    /*TODO*/
  // If the value to be deleted is greater than the root's
    value,
  // recurse in the right subtree
  else if (/*TODO*/)
    /*TODO*/
  // the value of root equals the value to be deleted.
  else
  {
```

deleteNode

```
    // Case 1: root is a leaf.
    if (/*TODO*/) {
      delete root;
      return nullptr;
    }
    // Case 2: root is a node with only one child.
    if (/*TODO*/) {
      Node* temp;
      if (/*TODO*/)
        /*TODO*/
      else
        /*TODO*/
      delete root;
      return temp;
    }
    // Case 3: root is a node with two children.
    // Find the inorder successor of this node
    Node* temp = minValueNode(root->right);
    // Copy the inorder successor's content to this node
    /*TODO*/
    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->value);
  }
  return root;
}
```

Üben wir ein bisschen...

Prüfung: Strategien

Ende

Viel Erfolg an der Prüfung! :)

Quellen

Vorlesung Informatik für Mathematiker und Physiker (252-0847-00L)
https://lec.inf.ethz.ch/ifmp/2019/