Informatik I PVK

Pascal Schärli

pascscha@student.ethz.ch

ETH zürich

Typen und Werte

Variablen

- Variablen können sich im Laufe des Programms ändern.
- Jede Variable hat ihren eigenen Typ (z.B. int, *float*, *char*)
- Jede Variable muss vor dem Gebrauch deklariert werden.

```
int a = 5;
b = a + a;
c = b + 3;
```

```
error: ‘b’ was not declared in this scope
```

```
int a = 5;
int b = a + a;
int a = b + 3;
```

```
error: error: redeclaration of ‘int a’
```

```
int a = 5;
int b = a + a;
int c = b + 3;
```

Primitive Datentypen – Boolean

Beschreibung

Booleans werden verwendet um Wahrheitswerte (wahr/falsch) darzustellen.

Eigenschaften

Bits	1
Von	0
Bis	1

Abkürzung

```
bool
```

Beispiel

```
bool a = true;
bool b = false;
```

Primitive Datentypen – Character

Beschreibung

Characters werden verwendet um Zeichen darzustellen

Eigenschaften

Bits	8
Von	0
Bis	255

Abkürzung

```
char
```

Beispiel

```
char a = 'f';
char b = '\n';
```

Primitive Datentypen – Integer

Beschreibung

Integers werden verwendet um ganze Zahlen darzustellen

Eigenschaften

Bits	32
Von	-2^{31}
Bis	$2^{31} - 1$

Abkürzung

```
int
```

Beispiel

```
int a = 1;
int b = -10;
```

Primitive Datentypen – Unsigned Integer

Beschreibung

Unsigned Integers werden verwendet um natürliche Zahlen darzustellen

Eigenschaften

Bits	32
Von	0
Bis	2^{32} - 1

Abkürzung

```
unsigned int
```

Beispiel

```
unsigned int a = 1;
unsigned int b = 5;
```

Primitive Datentypen – Float

Beschreibung

Floats werden verwendet um Fliesskommazahlen darzustellen.

Eigenschaften

Bits	32
Von	$-(2^{23})^{2^8}$
Bis	$(2^{23})^{2^8}$

Abkürzung

```
float
```

Beispiel

```
float a = 1.5;
float b = -1.9;
```

Primitive Datentypen – Double

Beschreibung

Doubles sind wie Floats, doch sie haben doppelt so hohe Präzision.

Eigenschaften

Bits	64
Von	$-(2^{52})^{2^{11}}$
Bis	$(2^{52})^{2^{11}}$

Abkürzung

```
double
```

Beispiel

```
double a = 1.5;
double b = -1.9;
```

Casting

- Das umwandeln von einem Datentyp zu einem anderen Datentyp nennt man casten
- Beim Casten kann es passieren, dass sich der Wert der Zahl verändern kann.

Casting – Boolean

- Irgendein Typ → Boolean
  - 0 → false
  - sonst → true
- Boolean → Irgendein Typ
  - false → 0
  - true → 1

Beispiele

Irgendein Typ → Boolean

```
5.8   →   true
  0   →   false
```

Boolean → Irgendein Typ

```
false   →   0
 true   →   1
```

Casting – Character

Die letzen 8 Bits werden zum entsprechenden Zeichen umgewandelt:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Casting – Integer $\leftrightarrow$ Float, Double

- Float, Double $\rightarrow$ Integer
  - Die Zahl wird abgerundet
- Integer $\rightarrow$ Float, Double
  - Die Zahl bleibt (meistens*) unverändert

**bei grossen Zahlen kann es sein, dass sie nicht exakt als float darstellbar sind*

Beispiele

Float,Double $\rightarrow$ Integer

`0.4`	$\rightarrow$	0
`9.99`	$\rightarrow$	9

Integer $\rightarrow$ Float,Double

5	$\rightarrow$	5
7	$\rightarrow$	7

Casting – Integer ↔ Unsigned Integer

Bei der Umwandlung von *int* zu *unsigned int* bleibt die Binäre darstellung gleich, die Zahl wird einfach anders interpretiert.

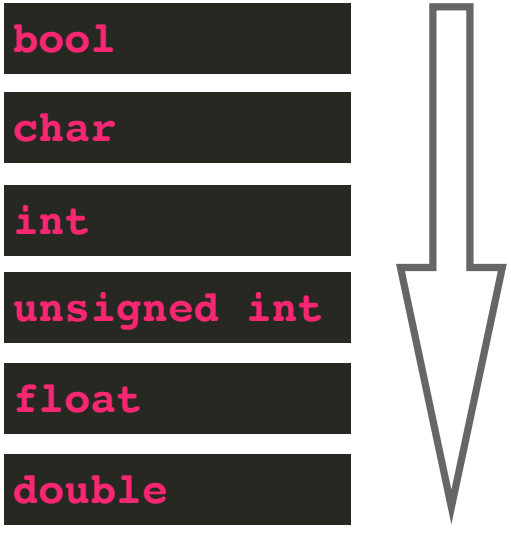
Beispiel: 4 Bit Zahlensystem

Binär	int	unsigned int
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binär	int	unsigned int
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Rechnen mit verschiedenen Typen

Beim Rechnen mit verschiedenen Datentypen werden die Werte immer in den "Allgemeineren" Datentyp umgewandelt.



Rechnen mit verschiedenen Typen


```
bool
char
int
unsigned int
float
double
```

Typ	Wert
double	46

Rechnen mit verschiedenen Typen

	Typ	Wert
`2 / 4.0f + 7 / 2.0`	`double`	4
`1 / 2 * 2.0`	`double`	0
`12 - 7 % 5`	int	10
`3 / 2.0f + 10 / 2.0`	`double`	6.5

Increment

- Der Pre- Post Increment Operator erhöht eine Variable um 1.
- Man kann das Increment vor oder nach der Variabel schreiben.

Pre Increment

*Die Variabel wird erhöht und der **neue** Wert wird zurückgegeben*

```
int a = 5;
std::cout << ++a;
```

Output

6

Wert von a

6

Post Increment

*Die Variabel wird erhöht und der **alte** Wert wird zurückgegeben*

```
int a = 5;
std::cout << a++;
```

Output

5

Wert von a

6

Increment

Pre Increment

*Die Variabel wird erhöht und der **neue** Wert wird zurückgegeben.*

Das Pre-increment für Integer ist äquivalent zu dieser Funktion:

```
int& pre(int& i){
    i = i + 1;
    return i;
}
```

```
++a
```

```
pre(a)
```

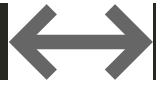
Post Increment

*Die Variabel wird erhöht und der **alte** Wert wird zurückgegeben.*

Das Post-increment für Integer ist äquivalent zu dieser Funktion:

```
int post(int& i){
    i = i + 1;
    return i - 1;
}
```

```
a++
```



```
post(a)
```

Vergleiche

In C++ können Variablen wie in der Mathematik miteinander verglichen werden.

Boolean Expressions

AND `&&`

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

OR `||`

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

XOR `!=`

a	b	a != b
false	false	false
false	true	true
true	false	true
true	true	false

NOT `!`

a	!a
false	true
true	false

Kurzschluss Evaluation

C++ wertet die rechte Seite von einem boolschen nicht mehr aus, falls das Resultat bereits bekannt ist.

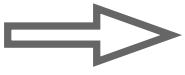
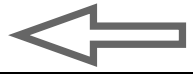
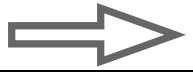
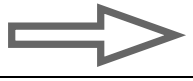
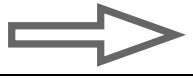
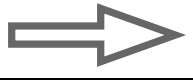
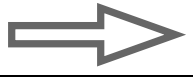
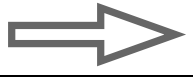
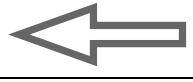
```
false && (1u-5<-1u*x)
```

```
false
```

```
true || (1u-5<-1u*x)
```

```
true
```

Präzedenz

Präzedenz	Operator	Assoziativität
2	`a++` `a--`	
3	`++a` `--a` `!`	
5	`a*b` `a/b` `a%b`	
6	`a+b` `a-b`	
9	`<` `<=` `>` `>=`	
10	`==` `!=`	
14	`&&`	
15	`||`	
16	`=` `+=`	

Komplette Tabelle: https://en.cppreference.com/w/cpp/language/operator_precedence

Präzedenz

```
int x = 1;
```

```
 x == 1  ||   1 / (x - 1)  < 1
```

```
 x == 1  ||  (1 / (x - 1)) < 1
```

```
 x == 1  || ((1 / (x - 1)) < 1)
```

```
(x == 1) || ((1 / (x - 1)) < 1)
```

```
(1 == 1) || ((1 / (x - 1)) < 1)
```

```
    true || ((1 / (x - 1)) < 1)
```

```
    true
```

P	Op	Assoz.
2	`a++` `a--`	⇨
3	`++a` `--a` `!`	⇦
5	`a*b` `a/b` `a%b`	⇨
6	`a+b` `a-b`	⇨
9	`<` `<=` `>` `>=`	⇨
10	`==` `!=`	⇨
14	`&&`	⇨
15	`||`	⇨
16	`=` `+=`	⇦

Präzedenz

```
int x = 1;

 !(1 && x)   + 1

(!(1 && x)) + 1

(!(1 && 1)) + 1

(!( true )) + 1

(  false  ) + 1

     0      + 1

            1
```

P	Op	Assoz.
2	`a++` `a--`	⇨
3	`++a` `--a` `!`	⇦
5	`a*b` `a/b` `a%b`	⇨
6	`a+b` `a-b`	⇨
9	`<` `<=` `>` `>=`	⇨
10	`==` `!=`	⇨
14	`&&`	⇨
15	`||`	⇨
16	`=` `+=`	⇦

Präzedenz

```
  8 > 4  > 2  > 1
((8 > 4) > 2) > 1
( true   > 2) > 1
(   1    > 2) > 1
       false  > 1
         0    > 1
              false
```

P	Op	Assoz.
2	`a++` `a--`	⇨
3	`++a` `--a` `!`	⇦
5	`a*b` `a/b` `a%b`	⇨
6	`a+b` `a-b`	⇨
9	`<` `<=` `>` `>=`	⇨
10	`==` `!=`	⇨
14	`&&`	⇨
15	`||`	⇨
16	`=` `+=`	⇦

Präzedenz

```
int a = 1;
int b = 1;
```

	Typ	Wert
`true && ! true == false || false`	`bool`	true
`++a / b--`	int	2
`1. + (2u - 5 < 6)`	double	1
`1 / 10 * 1. == 1. * 1 / 10`	`bool`	false

R/L – Values

- Operanden in Expressions können in R und L values unterteilt werden.
- L-Values dürfen links von einem Assignment Operator (=) stehen, R-Values nicht.

L-Value

Darf links oder rechts von = stehen.

Hat eine Adresse an welcher Werte gespeichert werden können.

R-Value

Darf nur rechts vom = stehen.

Kann keine Werte speichern.

R/L – Values

```
int a = 1;
int b = 2;
```

L-Value

```
a
```

```
a += 3
```

```
a = b
```

```
++a
```

R-Value

```
3
```

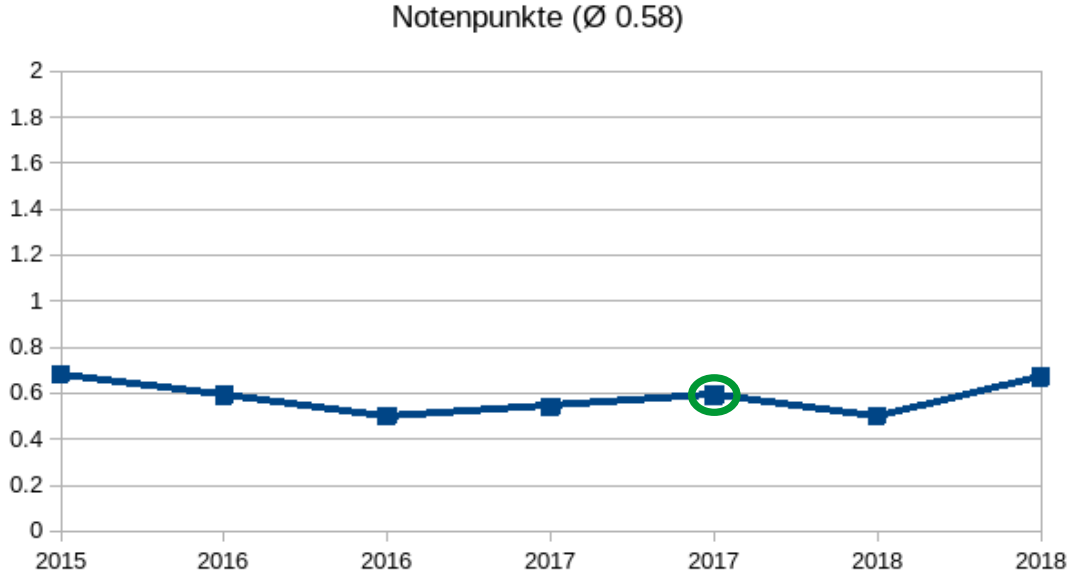
```
a + 3
```

```
a == b
```

```
a++
```

Prüfungsaufgabe

Typen und Werte



Sommer 2017, 0.59 Notenpunkte

Prüfungsaufgabe

Typen und Werte (Basistypen)

Basisprüfung Sommer 2017, 0.59 Notenpunkte

Geben Sie für jeden der Ausdrücke auf der nächsten Seite jeweils C++Typ und Wert an. Wenn der Wert nicht bestimmt werden kann, schreiben Sie “undefiniert”.

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

Prüfungsaufgabe

Typen und Werte (Basistypen)

Basisprüfung Sommer 2017, 0.59 Notenpunkte

	Typ	Wert
`1 / 4 * d`	`double`	0
`m % m`	`int`	0
`i = 10`	`int`	10
`i += i`	`int`	20
`++pre / post--`	`int`	2
`u-5 < 6`	`bool`	false

Zahlendarstellung

Fliesskomma $\rightarrow$ Binär

$$ab.cd \rightarrow a * 2 + b * 1 + c/2 + d/4$$

00.00 $\rightarrow$ 0
00.01 $\rightarrow$ 0 + 1/4
00.10 $\rightarrow$ 0 + 1/2
00.11 $\rightarrow$ 0 + 3/4
01.00 $\rightarrow$ 1
01.01 $\rightarrow$ 1 + 1/4
01.10 $\rightarrow$ 1 + 1/2
01.11 $\rightarrow$ 1 + 3/4

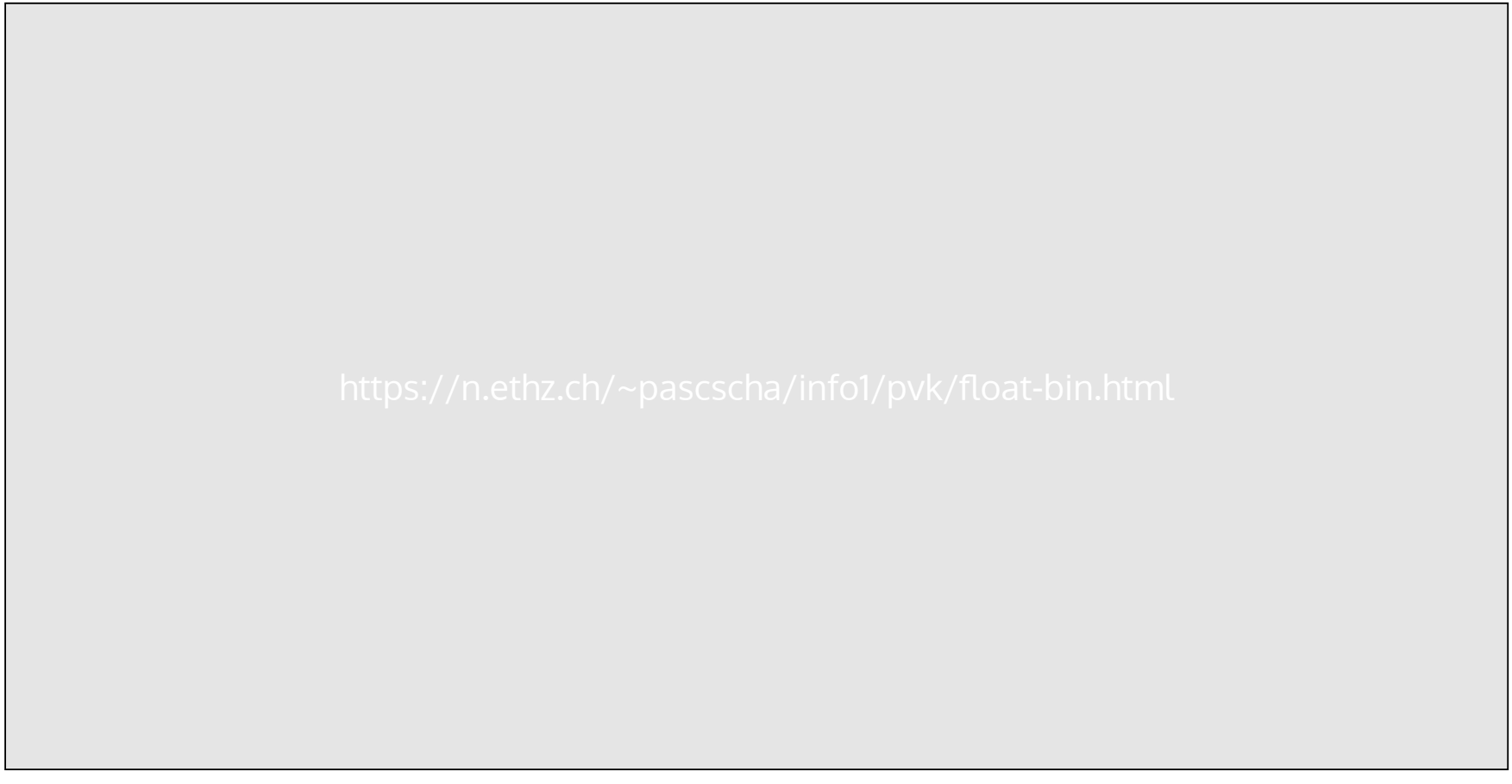
10.00 $\rightarrow$ 2
10.01 $\rightarrow$ 2 + 1/4
10.10 $\rightarrow$ 2 + 1/2
10.11 $\rightarrow$ 2 + 3/4
11.00 $\rightarrow$ 3
11.01 $\rightarrow$ 3 + 1/4
11.10 $\rightarrow$ 3 + 1/2
11.11 $\rightarrow$ 3 + 3/4

Fliesskomma $\rightarrow$ Binär

x_i	b_i	$x - b_i$	$x_{i+1} = 2 * (x - b_i)$
1.9	1	0.9	1.8
1.8	1	0.8	1.6
1.6	1	0.6	1.2
1.2	1	0.2	0.4
0.4	0	0.4	0.8
0.8	0	0.8	1.6
1.6	1	0.6	1.2

$$1.9 \rightarrow \qquad 1\,.\,1\,\overline{1\,1\,0\,0}\,\cancel{1}$$

Fliesskomma → Binär



FP-Systeme

$F^*(b, p, e_{min}, e_{max})$ ➡ $\pm b_0.b_1 b_2 \cdots b_{p-1} * b^e$

$b_i \in \{0, \ldots b-1\}$ *(Basis des Zahlensystems)*

$b_0 ! = 0$ *(Normalisiert)*

$p \rightarrow$ Anzah Ziffern

$e \in \{e_{min}, e_{min}+1, \ldots, e_{max}\}$ *(Bereich für Exponent)*

FP-Systeme

Sind diese Zahlen sind im folgenden Zahlensystem enthalten: $F^*(2, 4, -2, 2)$? $\pm b_0.b_1 b_2 \cdots b_{p-1} * b^e$

`1.111 * 2^2`	✓
`0.000 * 2^1`	✗ normalisiert $\rightarrow$ Zahl startet nicht mit 0
`1.001 * 2^-1`	✓
`1.0001 * 2^-1`	✗ zu viele Nachkommastellen
`1.111 * 2^5`	✗ Exponent ist nicht zwischen -2 und 2.
`1.000 * 2^1`	✓

FP-Systeme

Was sind die folgenden Zahlen in $F^*(2, 4, -2, 2)$?

$$\pm b_0.b_1 b_2 \cdots b_{p-1} * b^e$$

die grösste

```
1.111 * 2^2 ->  7.5
```

die kleinste

```
-1.111 * 2^2 -> -7.5
```

die kleinste nicht-negative.

```
1.000 * 2^-2 -> 0.25
```

FP-Systeme

$F^*(b, p, e_{min}, e_{max})$?

Was	Formel
Anzahl Positiver Werte	$(b-1) * b^{p-1} * (e_{max} - e_{min} + 1)$
Anzahl Werte	$(b-1) * b^{p-1} * (e_{max} - e_{min} + 1) * 2$
grösste Zahl	$(b^p - 1) * b^{e_{max}-p+1}$
kleinste positive Zahl	$b^{e_{min}}$

FP-Systeme

https://n.ethz.ch/~pascscha/info1/pvk/fp.html

Rechnen mit FP-Systemen

Wie kann man Zahlen in einem FP-System addieren?

- Beide Zahlen auf den selben Exponenten bringen
- Binärzahlen addieren (wie schriftliche Addition)
- Summe re-Normalisieren (1. . . .)
- Runden falls nötig

Rechnen mit FP-Systemen

```
1.001 * 2^-1 + 1.111 * 2^-2
```

$F^*(2, 4, -2, 2)$

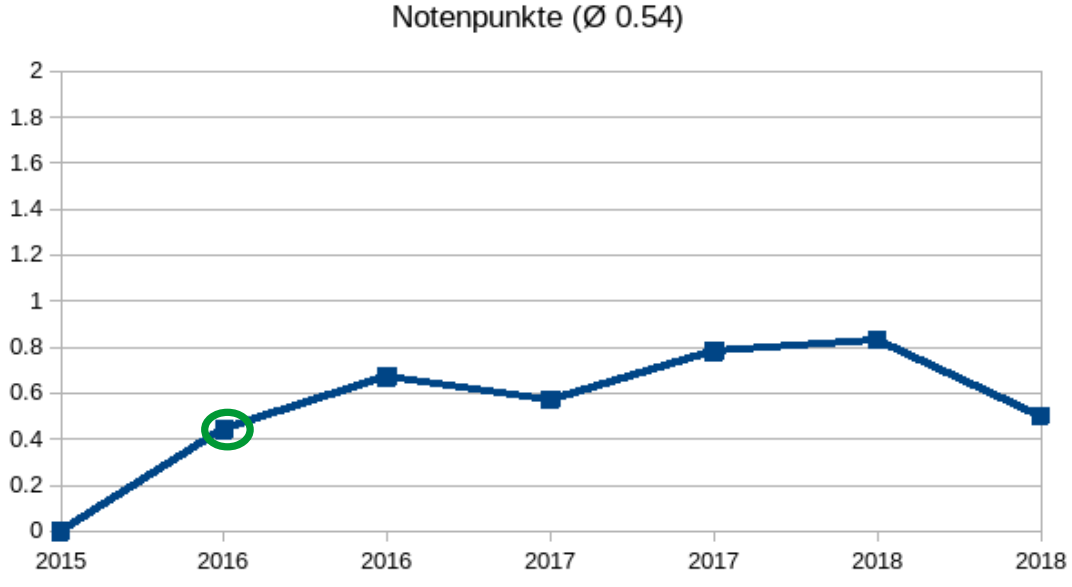
```
    1.001   * 2^-1
+   0.1111  * 2^-1
-------------------
= 10.0001   * 2^-1

= 1.00001   * 2^0

=> 1.000    * 2^0
```

1. Beide Zahlen auf den selben Exponenten bringen
2. Binärzahlen addieren (wie schriftliche Addition)
3. Summe re-Normalisieren (1. ...)
4. Runden falls nötig

$\rightarrow$ Resultat: 1 (Exakt: 1.03125)

Prüfungsaufgabe

Zahlendarstellungen



Winter 2016, 0.44 Notenpunkte

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

Basisprüfung Winter 2016, 0.44 Notenpunkte

$$F^*(b, p, e_{min}, e_{max})$$

$$b = 2$$

$$p = 5$$

$$e_{min} = -2$$

$$e_{max} = 2$$

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

$$F^*(2, 5, -2, 2)$$

Wie viele unterschiedliche positive Werte beinhaltet das normalisierte Fliesskommasystem F^*?

$$(b-1)b^{p-1} * (e_{max} - e_{min} + 1)$$
$$= (2-1) * 2^{5-1} * (2 - (-2) + 1)$$
$$= 2^4 * (5)$$
$$= \mathbf{80}$$

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

$$F^*(2, 5, -2, 2)$$

Welches ist die grösste Zahl in F^*?

$$
\begin{aligned}
&(b^p - 1) * b^{e_{max} - p + 1} \\
&= (2^5 - 1) * 2^{2-5+1} \\
&= (32 - 1) * 2^{-2} \\
&= 32/4 - 1/4 \\
&= \mathbf{7.75}
\end{aligned}
$$

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

$$F^*(2, 5, -2, 2)$$

Welches ist die maximale Präzision in F^*?

$b^{e_{min}}$

$= 2^{-2}$

$= \mathbf{\frac{1}{4}}$

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

$$F^*(2, 5, -2, 2)$$

Welche Zahl $\hat{x}$ aus F^* ist am nächsten an $x = 2.5625$?

(Arithmetisch Runden)

$2 \rightarrow 10_{(2)}$

$0.5625 \rightarrow 0.1001_{(2)}$

$2.5625 \rightarrow 10.1001_{(2)}$

$10.1001_{(2)}$

⬇ Normalisieren

$1.01001_{(2)} * 2^1$

⬇ Arithmetisch Runden

$\boldsymbol{\hat{x} = 1.0101_{(2)} * 2^1}$

Prüfungsaufgabe

Normalisiertes Fliesskommasystem

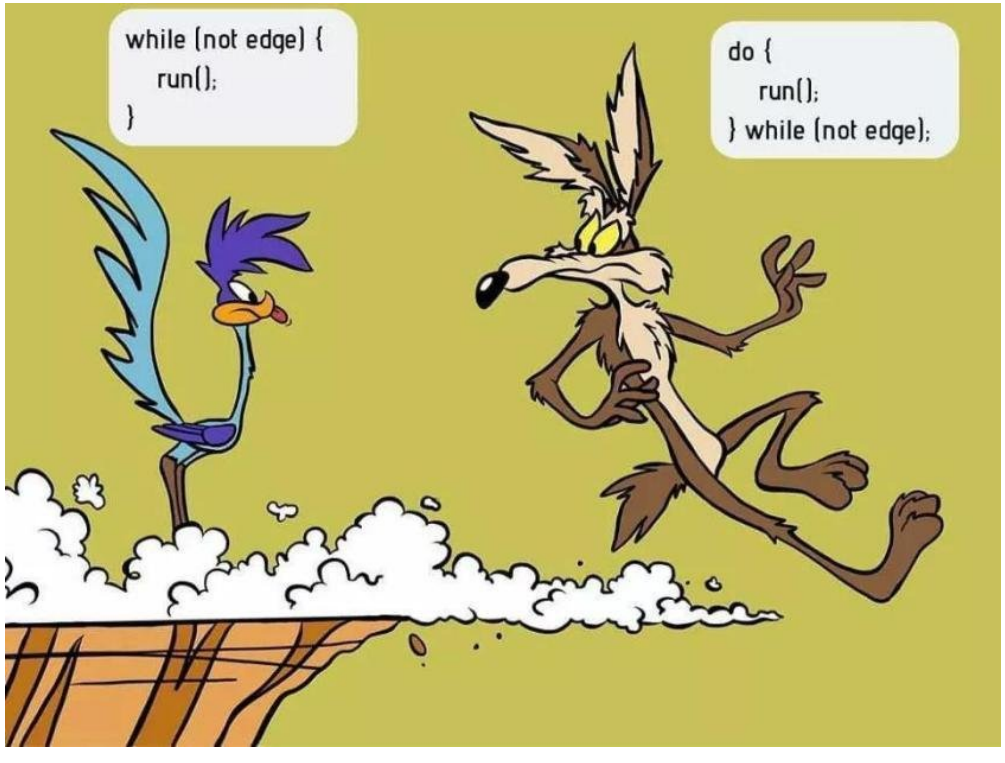
$$F^*(2, 5, -2, 2)$$

Was ist der Fehler $|\hat{x} - x|$?

$$\hat{x} = 1.0101_{(2)} * 2^1$$
$$= (1 + \tfrac{1}{4} + \tfrac{1}{16}) * 2^1$$
$$= 2.625$$

$$\Rightarrow |\hat{x} - x| = |2.625 - 2.5625|$$
$$\mathbf{= 0.0625}$$

Programmfluss



If – Statement

- Falls die Condition in den Runden () Klammern erfüllt ist, wird der Code in den geschweiften {} Klammern ausgeführt.

- Jedes If-Statement besteht aus:
  1. Genau einem *if*
  2. Beliebig vielen *else if*
  3. Höchstens einem *else*

```
int n = 50;

if(n > 128){
    std::cout << "n is larger than 128";
}
else if(n > 64){
    std::cout << "n is larger than 64";
}
else if(n > 32){
    std::cout << "n is larger than 32";
}
else if(n > 16){
    std::cout << "n is larger than 16";
}
else {
    std::cout << "n is smaller than 17";
}
```

```
n is larger than 32
```

Scopes

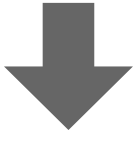
- Variablen sind immer nur in einem bestimmten Bereich (=scope) gültig
- Wenn dieser Bereich verlassen wird, wird die Variable wieder gelöscht

```
int a = 0;

if(a == 0){
    int b = 3;
}
else{
    int b = 4;
}

std::cout << b << std::endl;
```

error: ‘b’ was not declared in this scope

Scopes

```
int a = 2;

if (x < 7) {
    int a = 8;
    std::cout << a;
}

std::cout << a;
```



```
82
```

```
int a = 2;

if (x < 7) {
    a = 8;
    std::cout << a;
}

std::cout << a;
```

```
88
```

While-Loop

Solange die Condition in den Runden () Klammern erfüllt ist, wird der Code in den geschweiften {} Klammern wiederholt.

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

    i++;
}
```

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

For-Loop

- While-Loops verfolgen häufig die selbe Struktur:

1. Initialization
2. Condition
3. Increment

```
int i = 0;
```

```
i < 10
```

```
i++
```

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

    i++;
}
```

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

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

- Diese Struktur kann man daher auch schöner mit einem For-Loop schreiben.
- Der einzige unterschied ist, dass i nun im inneren Scope ist.

Do-While-Loop

- Alternative zum while-Loop
- Kondition wird erst am Ende geprüft
- Schleifenkörper wird mindestens ein Mal ausgeführt

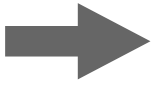
```
int i = 1;
do{
    std::cout << i << " ";
    i*=2;
} while(i < 10);
```

```
1 2 4 8
```

Abkürzung

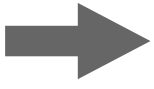
Falls ein Statement nur eine Zeile lang ist, kann man die geschweiften {} Klammern weglassen

```
if(i > 10){
    std::cout << "yes" << std::endl;
}
```



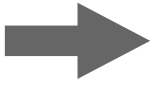
```
if(i > 10)
    std::cout << "yes" << std::endl;
```

```
while (i < n){
    std::cout << ++i << "\n";
}
```



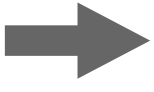
```
while (i < n)
    std::cout << ++i << "\n";
```

```
for (int i = 1; i <= n; ++i){
    std::cout << i << "\n";
}
```



```
for (int i = 1; i <= n; ++i)
    std::cout << i << "\n";
```

```
do{
    std::cout << i++ << "\n";
}while (i <= n);
```



```
do
    std::cout << i++ << "\n";
while (i <= n);
```

Aufgabe

- Welche drei Unterschiede gibt es zwischen diesen drei Loops?

```
// loop 1
for (int i = 1; i <= n; ++i){
    std::cout << i << "\n";
}
```

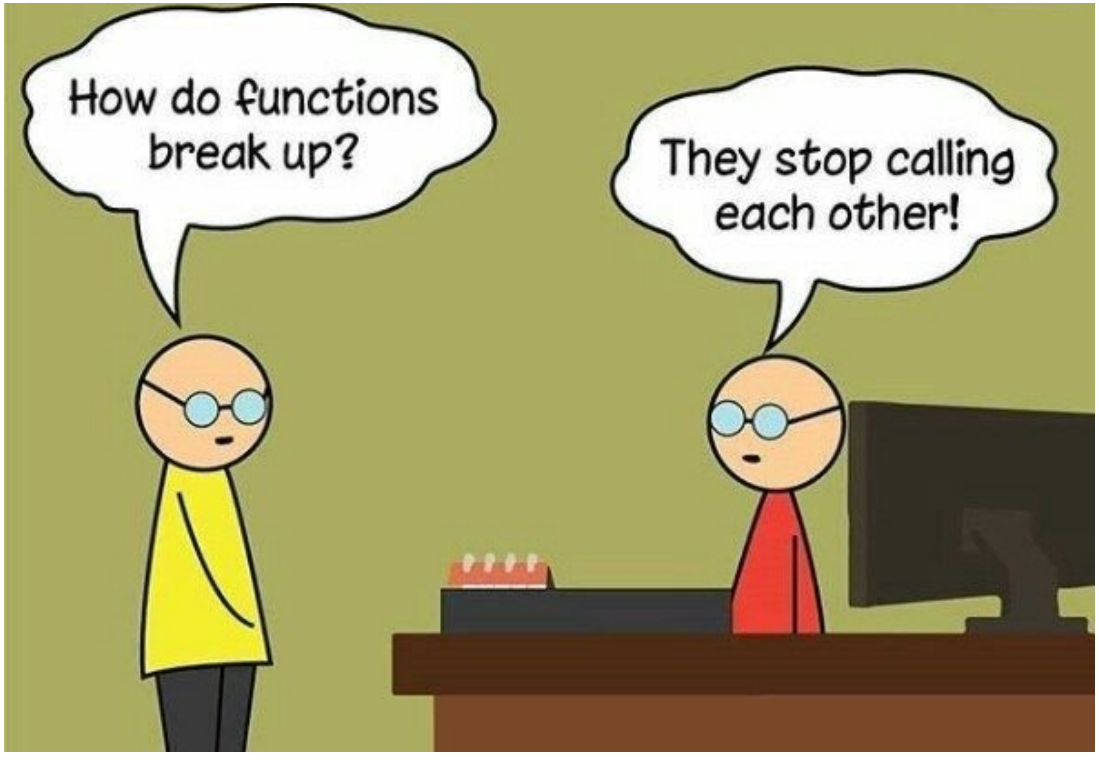
```
// loop 2
int i = 0;
while (i < n){
    std::cout << ++i << "\n";
}
```

```
// loop 3
int i = 1;
do{
    std::cout << i++ << "\n";
}while (i <= n);
```

Lösung:

- Der letzte Loop hat den Output 1 falls $n \leq 0$
- Loops 1 und 3 terminieren nie wenn n der grösstmögliche Integer ist
- Bei Loop 1 ist i nach dem Ausführen out of scope

Funktionen



Motivation

Schreibe ein Programm, welches für drei Zahlen a, b und c die Summe a! + b! + c! zurück gibt.

- Wir müssen drei mal denselben Code kopieren um die Fakultät zu berechnen

```
unsigned int a,b,c;
std::cin >> a >> b >> c;

unsigned int a_fact = 1;
for(int i = 1; i <= a; i++){
        a_fact *= i;
}

unsigned int b_fact = 1;
for(int i = 1; i <= b; i++){
        b_fact *= i;
}

unsigned int c_fact = 1;
for(int i = 1; i <= c; i++){
        c_fact *= i;
}

std::cout << a_fact + b_fact + c_fact << std::endl;
```

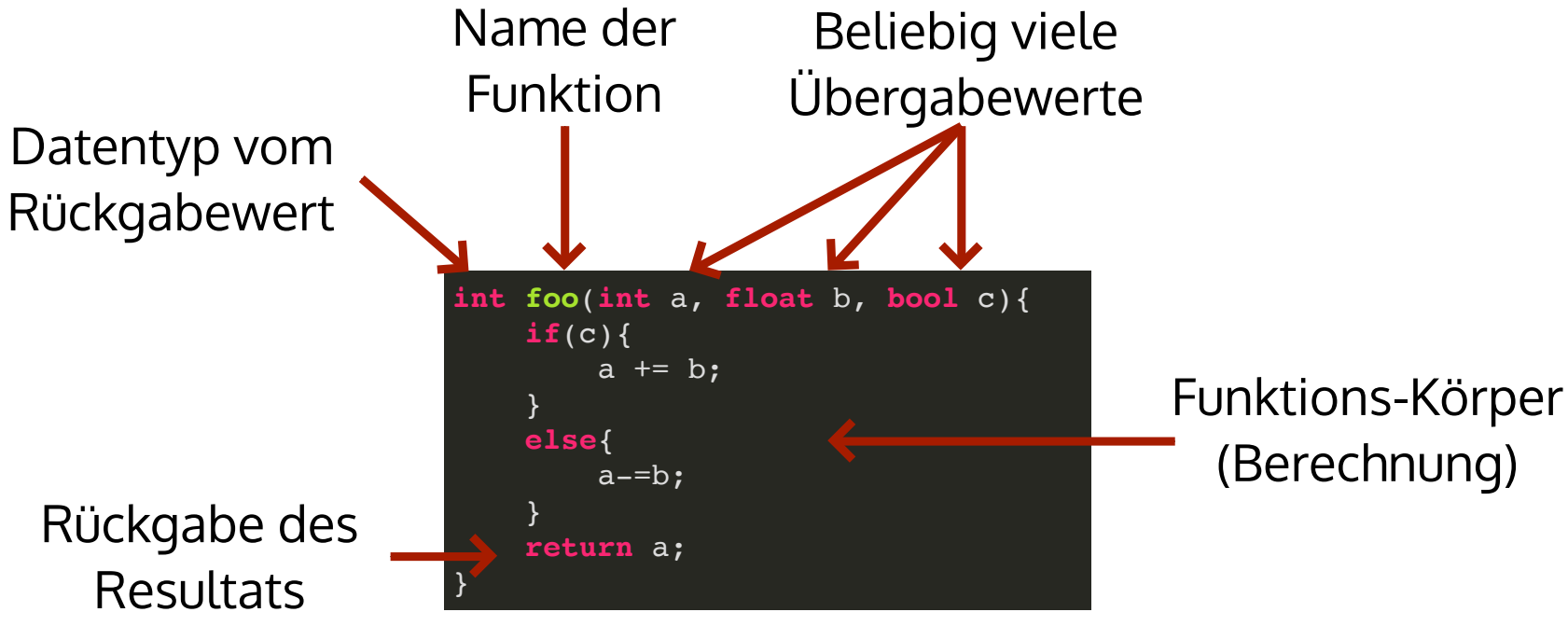
Motivation

Wie können wir verhindern, immer wieder den selben Code zu kopieren?

- Wir können die Fakultät in eine "Funktion" packen, so dass wir den Algorithmus nur einmal schreiben müssen.

```
#include <iostream>

unsigned int fact(unsigned int n){
  unsigned int out = 1;
  for(int i = 1; i <= n; i++){
    out *= i;
  }
  return out;
}


int main(){
  unsigned int a,b,c;
  std::cin >> a >> b >> c;

  std::cout << fact(a)+fact(b)+fact(c)<<std::endl;
  return 0;
}
```

Syntax


Name der Funktion
Beliebig viele Übergabewerte
Datentyp vom Rückgabewert
int foo(int a, float b, bool c){
if(c){
a += b;
}
else{
a-=b;
}
return a;
}
Funktions-Körper (Berechnung)
Rückgabe des Resultats

Aufgabe

Findet die 10 Fehler!

```
bool isPrime(unsigned int n){
    for(unsigned int i = 2; i <= n/2; i++){
        if(n % i == 0){
                return false;
        }
    }
    return true;
}
```

Pre- / Postcondition

```
//PRE:  left <= right
//POST: returns true iff x is in the Interval [left, right]
bool inInterval(double x, double left, double right){
        return x >= left && x <= right;
}
```

Precondition:

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert Definitionsbereich der Funktion.

Postcondition:

- Was gilt nach Funktionsaufruf?
- Spezifiziert Wert und Effekt des Funktionsaufrufes.

Forward Declaration

Was machen wir, wenn wir zwei Funktionen haben, welche sich im Kreis aufrufen?

```
 1
 2
 3 int f(int i){
 4     if(i < 0) return 1;
 5     return g(i-1);
 6 }
 7
 8 int g(int i){
 9     return 2*f(i-1);
10 }
```

error: ‘g’ was not declared in this scope

Forward Declaration

Wir müssen eine Funktion bereits vorher deklarieren, aber noch nicht implementieren. Dies nennt man *forward declaration*.

```
 1 int g(int i);
 2
 3 int f(int i){
 4     if(i < 0) return 1;
 5     return g(i-1);
 6 }
 7
 8 int g(int i){
 9     return 2*f(i-1);
10 }
```

Referenzen


CARTOONSTOCK
Search ID: jcon1035
"DO YOU HAVE ANY OTHER REFERENCES BESIDES YOUR MOM AND SANTA CLAUS?"

Namensalias

In C++ können wir existierende Variablen "Verlinken"

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

```
2
```

Call by Reference

Es ist möglich Funktionsargumente als Referenzen zu definieren.

```cpp
#include <iostream>

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

int main(){
    int a = 2;
    int b = 4;
    swap(a, b);
    std::cout << a << b;
    return 0;
}
```

```
24
```

```cpp
#include <iostream>

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

int main(){
    int a = 2;
    int b = 4;
    swap(a, b);
    std::cout << a << b;
    return 0;
}
```

```
42
```

Motivation

Warum braucht es Referenzen überhaupt?

- Referenzen erlauben uns mehrere Werte zurückzugeben in einer Funktion.

```
void scale(double& x, double& y, double amount){
    x *= amount;
    y *= amount;
}
```

- Die Übergebenen Parameter müssen nicht kopiert werden.

```
void read_i (Vector& v, unsigned int i);
```

- Manche Objekte können nicht kopiert werden.

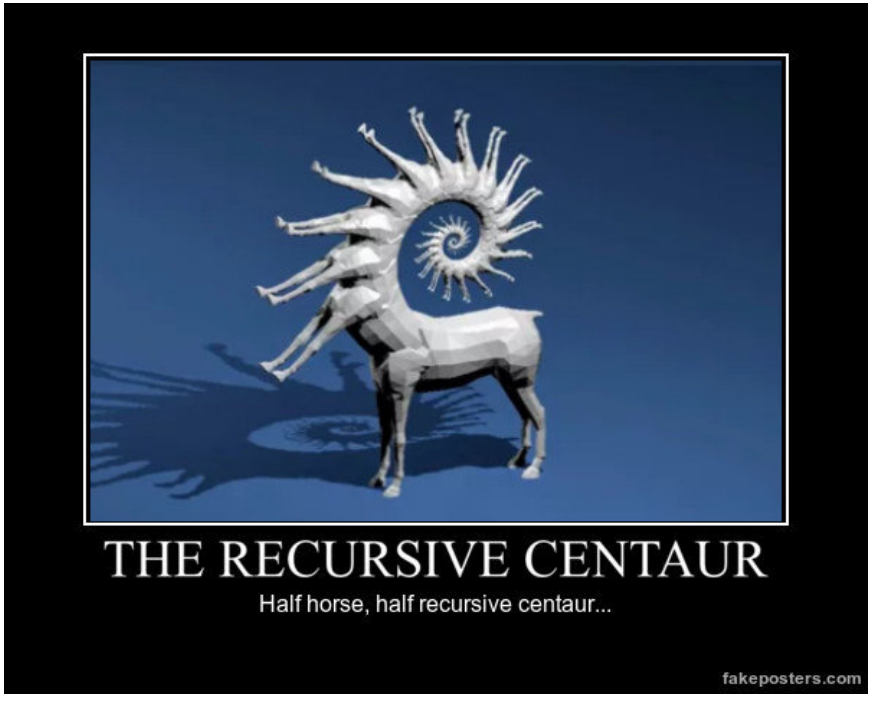
```
std::ostream o = std::cout;
o << "It works!\n";
```

```
error: use of deleted function
```

```
std::ostream& o = std::cout;
o << "It works!\n";
```

```
It works!
```

Rekursion


THE RECURSIVE CENTAUR
Half horse, half recursive centaur...
fakeposters.com

Fibonacci

In C++ können sich Funktionen selbst "rekursiv" aufrufen. Dies vereinfacht manchmal die Implementation einer Funktion.

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{sonst} \end{cases}$$

```cpp
unsigned int f(unsigned int n){
    if(n == 0) return 0;
    if(n == 1) return 1;
    return f(n-1) + f(n-2);
}
```

```cpp
int main(){
    for(unsigned int i = 0; i < 10; i++){
        std::cout << f(i) << " ";
    }
    return 0;
}
```

```
0 1 1 2 3 5 8 13 21 34
```

Beispiel 1

Was ist der Output von diesem Programm?

Welche Mathematische Operation ist es?

```
int foo(double b, unsigned int e){
    if(e == 0){
        return 1;
    }
    else{
        return b * foo(b, e-1);
    }
}
```

```
int main(){
    std::cout << foo(2,2) << std::endl;
    std::cout << foo(2,3) << std::endl;
    std::cout << foo(5,2) << std::endl;
    std::cout << foo(10,9) << std::endl;
    return 0;
}
```

Beispiel 1 – Erklärung

```
int foo(double b, unsigned int e){
    if(e == 0){
        return 1;
    }
    else{
        return b * foo(b, e-1);
    }
}
```

$$foo(2,3) \rightarrow b = 2, e = 3 \rightarrow 2 * foo(2,2) \quad = 2 * 4 = 8$$

$$foo(2,2) \rightarrow b = 2, e = 2 \rightarrow 2 * foo(2,1) \quad = 2 * 2 = 4$$

$$foo(2,1) \rightarrow b = 2, e = 1 \rightarrow 2 * foo(2,0) \quad = 2 * 1 = 2$$

$$foo(2,0) \rightarrow b = 2, e = 0 \rightarrow \quad 1$$

Beispiel 1 – Lösung

Die Funktion Implementiert das Potenzieren der Basis b mit dem Exponent e.

$$foo(b, e) = b^e$$

```cpp
#include <iostream>

int foo(double b, unsigned int e){
    if(e == 0){
        return 1;
    }
    else{
        return b * foo(b, e-1);
    }
}

int main(){
    std::cout << foo(2,2) << std::endl;
    std::cout << foo(2,3) << std::endl;
    std::cout << foo(5,2) << std::endl;
    std::cout << foo(10,9) << std::endl;
    return 0;
}
```

```
4
8
25
1000000000
```

Beispiel 2

Was ist der Output von diesem Programm?

Welche Mathematische Operation ist es?

```
int foo2(unsigned int l, unsigned int b){
    if(l < b){
        return 0;
    }
    else{
        return 1 + foo2(l / b, b);
    }
}
```

```
int main(){
    std::cout << foo2(2,2) << std::endl;
    std::cout << foo2(16,4) << std::endl;
    std::cout << foo2(30,3) << std::endl;
    std::cout << foo2(10000,10) << std::endl;
    return 0;
}
```

Beispiel 2 – Erklärung

```
int foo2(unsigned int l, unsigned int b){
    if(l < b){
        return 0;
    }
    else{
        return 1 + foo(l / b, b);
    }
}
```

$$foo2(30,3) \rightarrow l = 30, b = 3 \rightarrow 1 + foo2(10,3) = 1 + 2 = 3$$

$$foo2(10,3) \rightarrow l = 10, b = 3 \rightarrow 1 + foo2(3,3) = 1 + 1 = 2$$

$$foo2(3,3) \rightarrow l = 3, b = 3 \rightarrow 1 + foo2(1,3) = 1 + 0 = 1$$

$$foo2(1,3) \rightarrow l = 1, b = 3 \rightarrow 0$$

Beispiel 2 – Lösung

Die Funktion Implementiert das der abgerundete Logarithmus von l zur Basis b.

$$foo2(l, b) = \lfloor log_b(l) \rfloor$$

```cpp
#include <iostream>

int foo2(unsigned int l, unsigned int b){
    if(l < b){
        return 0;
    }
    else{
        return 1 + foo2(l / b, b);
    }
}

int main(){
    std::cout << foo2(2,2) << std::endl;
    std::cout << foo2(16,4) << std::endl;
    std::cout << foo2(30,3) << std::endl;
    std::cout << foo2(10000,10) << std::endl;
    return 0;
}
```

```
1
2
3
4
```

Rekursive Funktionen Schreiben

- Rekursive Funktionen haben grundsätzlich immer die selbe Struktur:
  - Abbruchsbedingungen
  - Rekursiver Funktionsaufrufe

```
1 unsigned int f(unsigned int n){
2     if(n == 0) return 0;
3     if(n == 1) return 1;
4     return f(n-1) + f(n-2);
5 }
```

```
1 int foo(double b, unsigned int e){
2     if(e == 0) return 1;
3
4     return b * foo(b, e-1);
5 }
```

```
1 int foo2(unsigned int l, unsigned int b){
2     if(l == 1) return 0;
3
4     return 1 + foo(l / b, b);
5 }
```

Rekursive Funktionen Schreiben

- Die Funktion ruft sich selbst mit einer einfacheren Variante des Problems auf
- Der Trick zum Schreiben von rekursiven Funktionen ist, dass man davon ausgehen muss, dass die Funktion welche man schreibt bereits Funktioniert.

Beispiel – Print reverse

Wir wollen eine Funktion schreiben, welche einen String in umgekehrter Reihenfolge printen kann:

```
//POST: Prints the interval [start, end) of a string
//      in reverse.
void print_reverse(std::string s, int start, int end){



}

int main(){
    std::string s = "KVP olleH";
    print_reverse(s, 0, s.size());
    return 0;
}
```

Beispiel – Print reverse

Wir wollen eine Funktion schreiben, welche einen String in umgekehrter Reihenfolge printen kann:

Abbruchsbedingung:

Falls unser Intervall leer ist, müssen wir nichts printen.

```
//POST: Prints the interval [start, end) of a string
//      in reverse.
void print_reverse(std::string s, int start, int end){
    if(start >= end) return;


}

int main(){
    std::string s = "KVP olleH";
    print_reverse(s, 0, s.size());
    return 0;
}
```

Beispiel – Print reverse

Wir wollen eine Funktion schreiben, welche einen String in umgekehrter Reihenfolge printen kann:

Rekursiver Funktionsaufruf:

Falls das Intervall nicht leer ist, können wir zuerst unseren String ohne das erste Zeichen rückwärts ausgeben

```
//POST: Prints the interval [start, end) of a string
//      in reverse.
void print_reverse(std::string s, int start, int end){
    if(start >= end) return;
    print_reverse(s, start+1, end);

}

int main(){
    std::string s = "KVP olleH";
    print_reverse(s, 0, s.size());
    return 0;
}
```

Beispiel – Print reverse

Wir wollen eine Funktion schreiben, welche einen String in umgekehrter Reihenfolge printen kann:

Nachdem wir den rest vom String rückwärts ausgegeben haben, müssen wir nur noch das erste Zeichen von unserem Intervall ausgeben.

```
//POST: Prints the interval [start, end) of a string
//      in reverse.
void print_reverse(std::string s, int start, int end){
    if(start >= end) return;
    print_reverse(s, start+1, end);
    std::cout << s[start];
}

int main(){
    std::string s = "KVP olleH";
    print_reverse(s, 0, s.size());
    return 0;
}
```

Beispiel – Print reverse

```cpp
#include <iostream>

//POST: Prints a string in reverse from start to end
void print_reverse(std::string s, int start, int end){
    if(start >= end) return;
    print_reverse(s, start+1, end);
    std::cout << s[start];
}

int main(){
    std::string s = "KVP olleH";
    print_reverse(s, 0, s.size());
    return 0;
}
```

Output:

```
Hello PVK
```

Beispiel – Print reverse

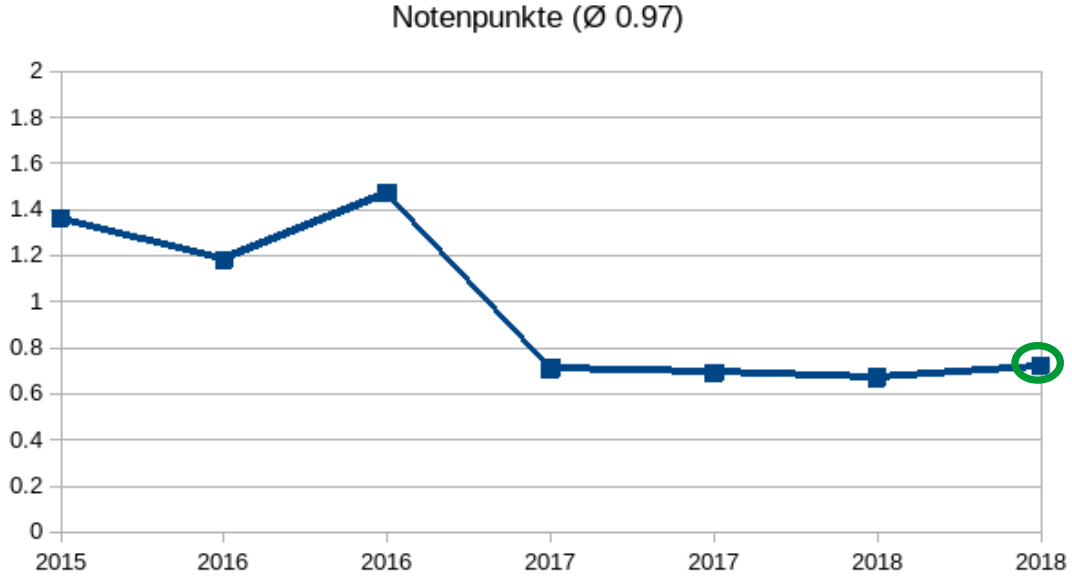
```cpp
//POST: Prints a string in reverse from start to end
void print_reverse(std::string s, int start, int end){
    if(start >= end) return;
    print_reverse(s, start+1, end);
    std::cout << s[start];
}
```

```
1 print_reverse("KVP",0,3)
2     print_reverse("KVP",1,3)
3         print_reverse("KVP",2,3)
4             print_reverse("KVP",3,3)
5             start == end -> return
6         std::cout << 'P'
7     std::cout << 'V'
8 std::cout << 'K'
```

P V K

Prüfungsaufgabe

Rekursion



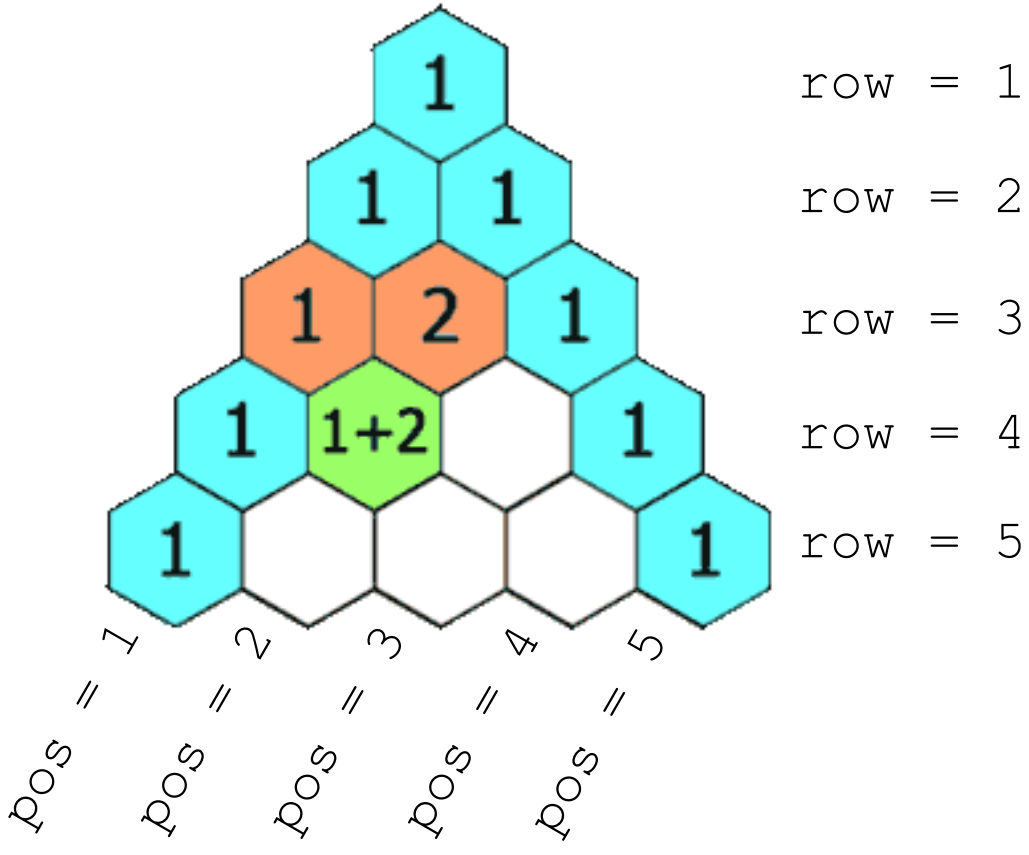
Sommer 2018, 0.72 Notenpunkte

Prüfungsaufgabe

Rekursion: Pascalsches Dreieck

Basisprüfung Sommer 2018, 0.72 Notenpunkte

Vervollständigt das gegebene Programm, welches die Zahl für eine gegebene Position berechnet.



Prüfungsaufgabe

Rekursion: Pascalsches Dreieck

Basisprüfung Sommer 2018, 0.72 Notenpunkte

```
int main(){
    //Input
    std:: cout<< "Please input a row and a position along the row: ";
    int row, pos;
    std::cin >> row >> pos;

    //Check whether the input integers correspond to a valid entry
    assert (check_input_validity(row, pos));

    //Display the result
    std::cout << "The value at row " << row << " and position " << pos
              << " is "<< compute_pascal(row, pos);

    return 0;
}
```

Prüfungsaufgabe

Rekursion: Pascalsches Dreieck

Basisprüfung Sommer 2018, 0.72 Notenpunkte

```
bool check_input_validity(int row, int pos){
    if (row < pos || pos < 1){
        std::cout << "Invalid: no element at the given row-position pair.";
        return false;
    }
    else {
        return true;
    }
}
```

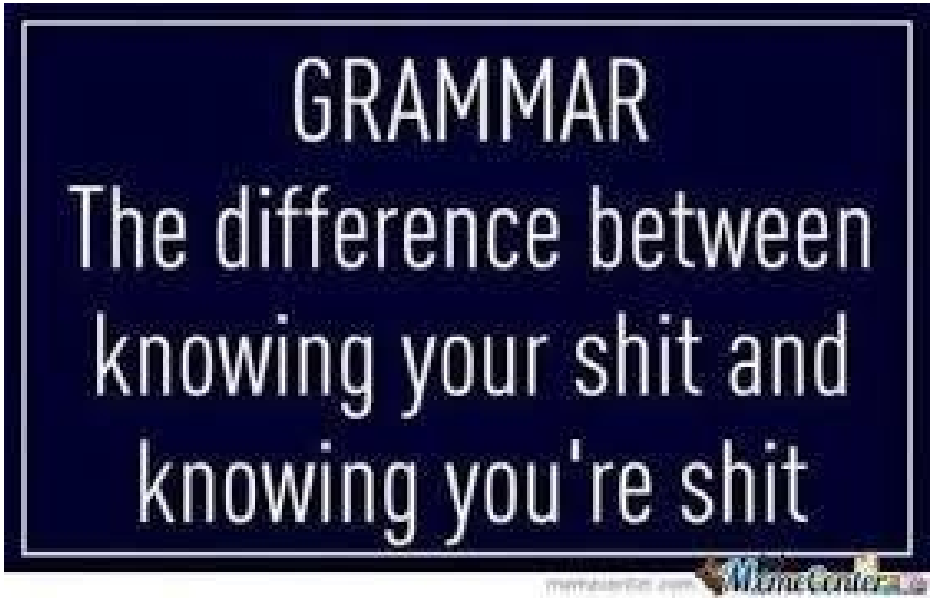
Prüfungsaufgabe

Rekursion: Pascalsches Dreieck

Basisprüfung Sommer 2018, 0.72 Notenpunkte

```
int compute_pascal(int row, int pos){
    if (pos == 1){
        return 1;
    }
    else if (pos == row){
        return 1;
    }
    else {
        return compute_pascal(row-1, pos) + compute_pascal(row-1, pos-1);
    }
}
```

BNF/EBNF


GRAMMAR
The difference between
knowing your shit and
knowing you're shit

BNF

- Die **B**ackus-**N**aur-**F**orm (BNF) ist ein Rekursiver Weg um Elemente aus einem "Alphabet" mit gegebenen Regeln zusammenzusetzen.

- Sie ist eine Sammlung von Substitutionsregeln, mit welchen alle erlaubten Sätze erstellt werden können.

BNF – Prefix Expression Parser

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

BNF – Prefix Expression Parser

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

Digit

Irgend eine beliebige Ziffer von '0' bis '9'

Beispiele

0, 5, 9

BNF – Prefix Expression Parser

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

Integer

Zwischen 1 und ∞ viele Ziffern aneinander gehängt

Beispiele

1, 0034, 089457029348720398712304190857

BNF – Prefix Expression Parser

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

Number

Entweder 1 Integer oder 2 Integer mit '.' getrennt

Beispiele

0, 1.1, 01.1200, 3.14159

BNF – Prefix Expression Parser

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

Operand

Entweder eine Number oder eine Expression

Beispiele

0, 1, 2.2, [+ 1 2, + 1 * 2 3]

BNF – Prefix Expression Parser

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

Operator

Irgend ein beliebiger Operator '+', '-', '/', '*'

Beispiele

+, -, /, *

BNF – Prefix Expression Parser

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

Expression

Ausrücke in der Prefix-Schreibweise. In der Prefix-Schreibweise wird jeweils der Operator zuerst geschrieben.
$(1 + 2 \rightarrow +12)$

Beispiele

+ 2 3.12 , * 1 - 2.5 0, - 1 * 2 3 / 2 3

EBNF

- Die **E**xtended-**B**ackus-**N**aur-**F**orm hat zusätzliche Syntax
  -> kompaktere Notationen
- {...} -> Inhalt kann beliebig viel (inkl. 0) Mal wiederholt werden.
- [...] -> Inhalt wird entweder 0 oder 1 Mal wiederholt

BNF

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

EBNF

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

BNF – Beispiel

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

Entspricht der folgende String einer gültigen Expression?

\+ * 3 2 9

BNF – Beispiel

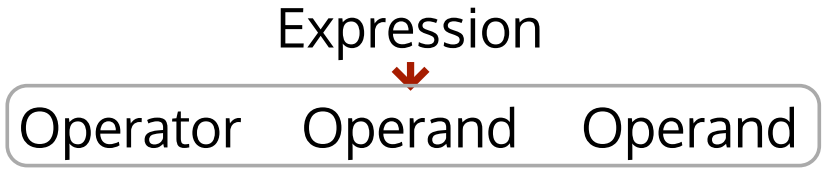
```
1 Expression = Operator ' ' Operand ' ' Operand
2 Operator   = '+' | '-' | '/' | '*'
3 Operand    = Number | Expression
4 Number     = Integer ['.' Integer]
5 Integer    = Digit { Digit }
6 Digit= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

\+ * 3 2 9 Expression

BNF – Beispiel

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

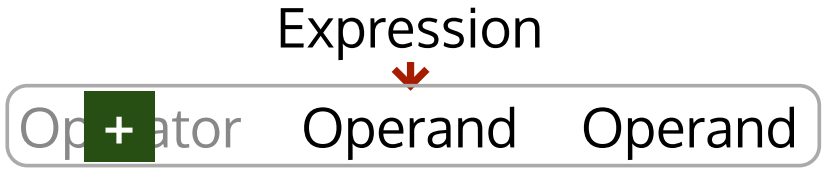
\+ * 3 2 9



BNF – Beispiel

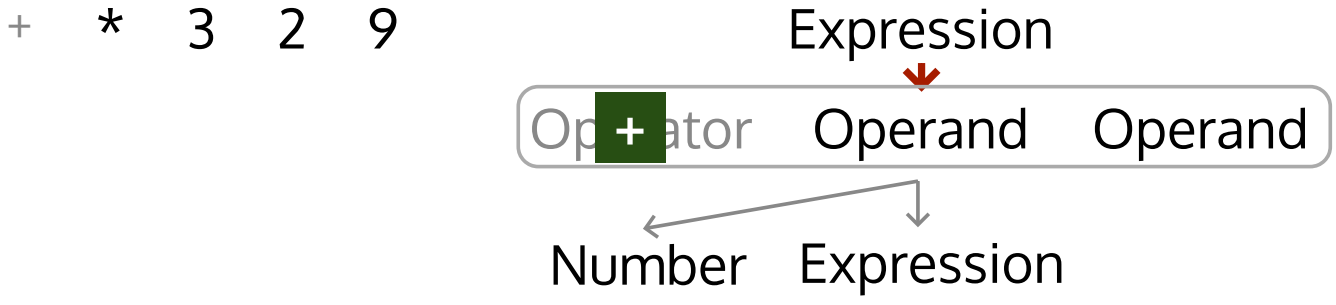
```
1 Expression = Operator ' ' Operand ' ' Operand
2 Operator   = '+' | '-' | '/' | '*'
3 Operand    = Number | Expression
4 Number     = Integer ['.' Integer]
5 Integer    = Digit { Digit }
6 Digit= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

\+ * 3 2 9



BNF – Beispiel

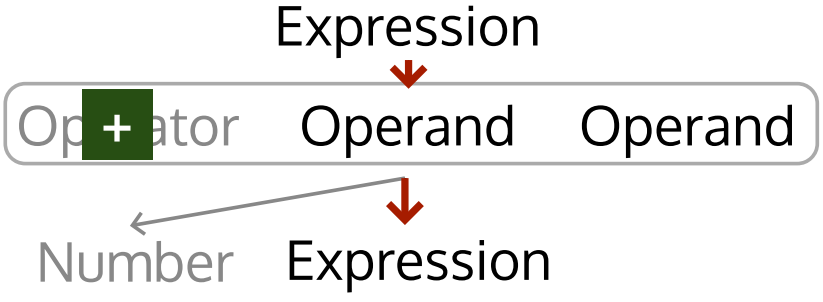
```
1 Expression = Operator ' ' Operand ' ' Operand
2 Operator   = '+' | '-' | '/' | '*'
3 Operand    = Number | Expression
4 Number     = Integer ['.' Integer]
5 Integer    = Digit { Digit }
6 Digit= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```



BNF – Beispiel

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

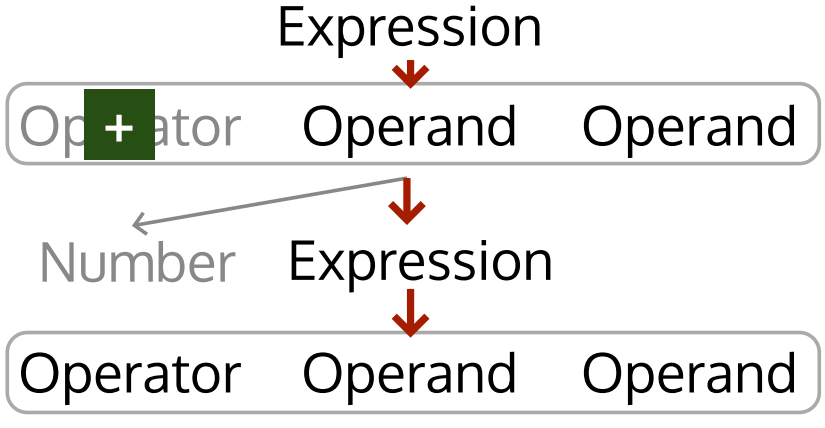
\+ * 3 2 9



BNF – Beispiel

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

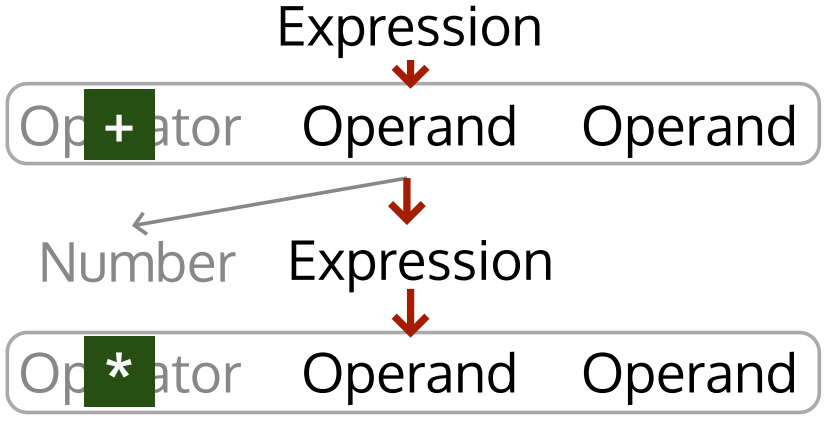
\+ * 3 2 9



BNF – Beispiel

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

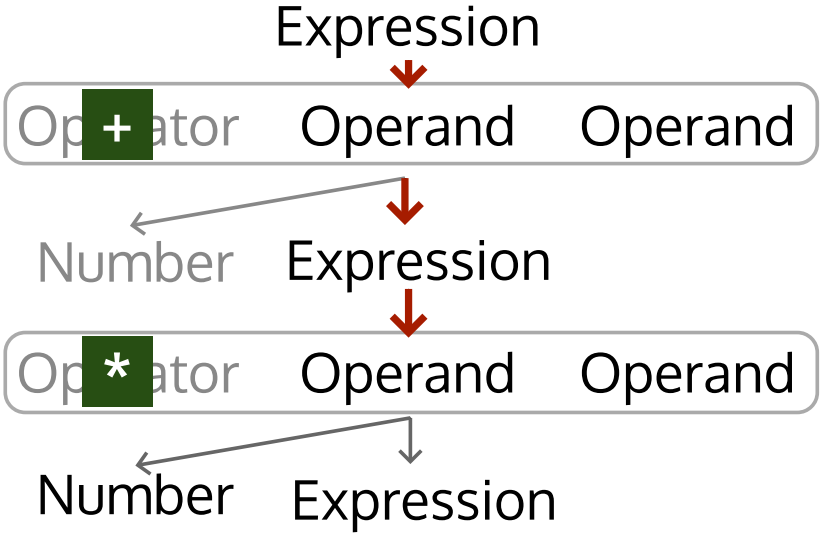
\+ * 3 2 9



BNF – Beispiel

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

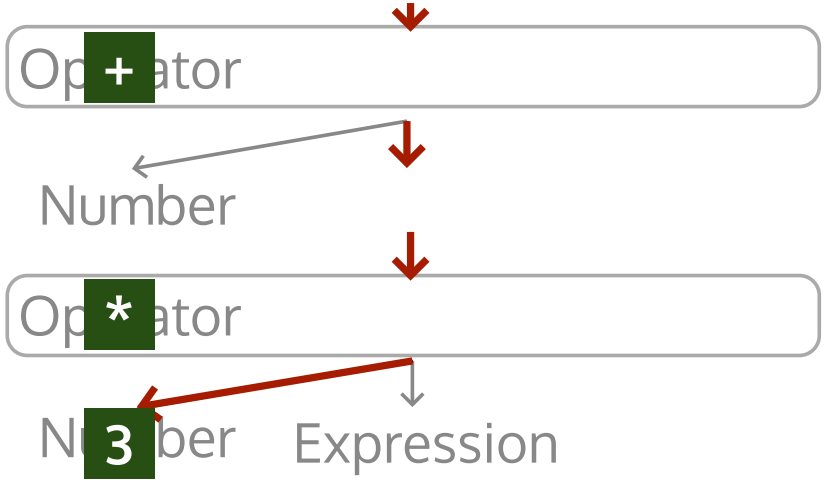
+ * 3 2 9



BNF – Beispiel

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

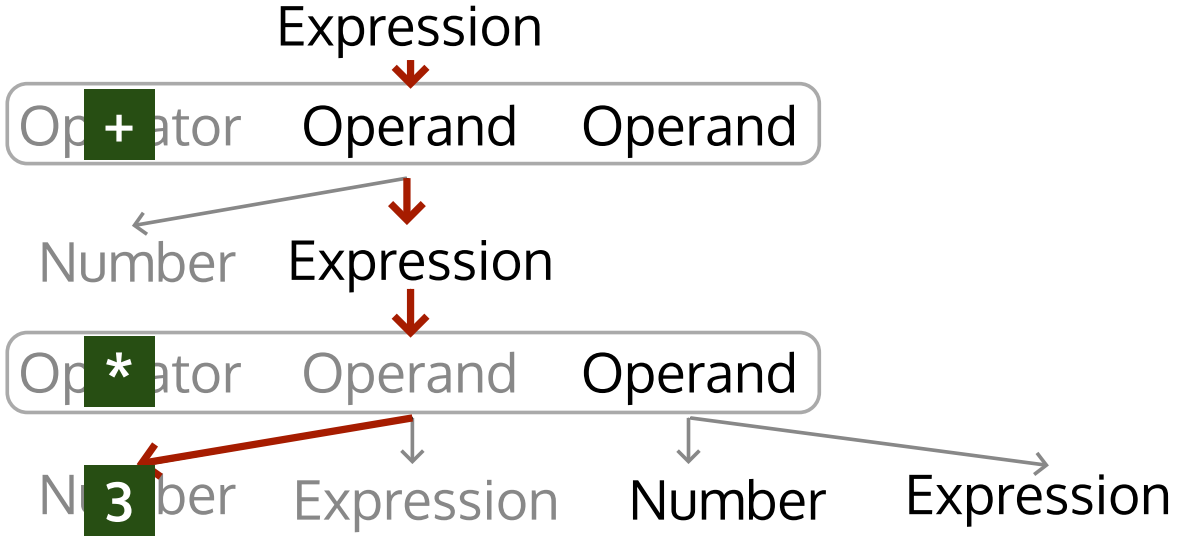
+ * 3 2 9



BNF – Beispiel

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

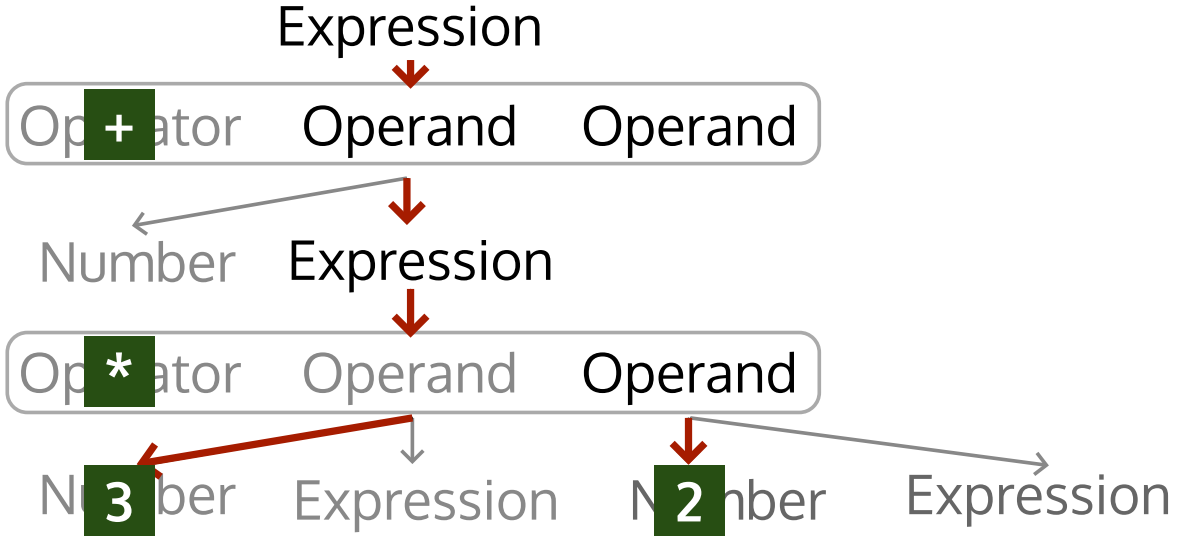
+ * 3 2 9



BNF – Beispiel

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

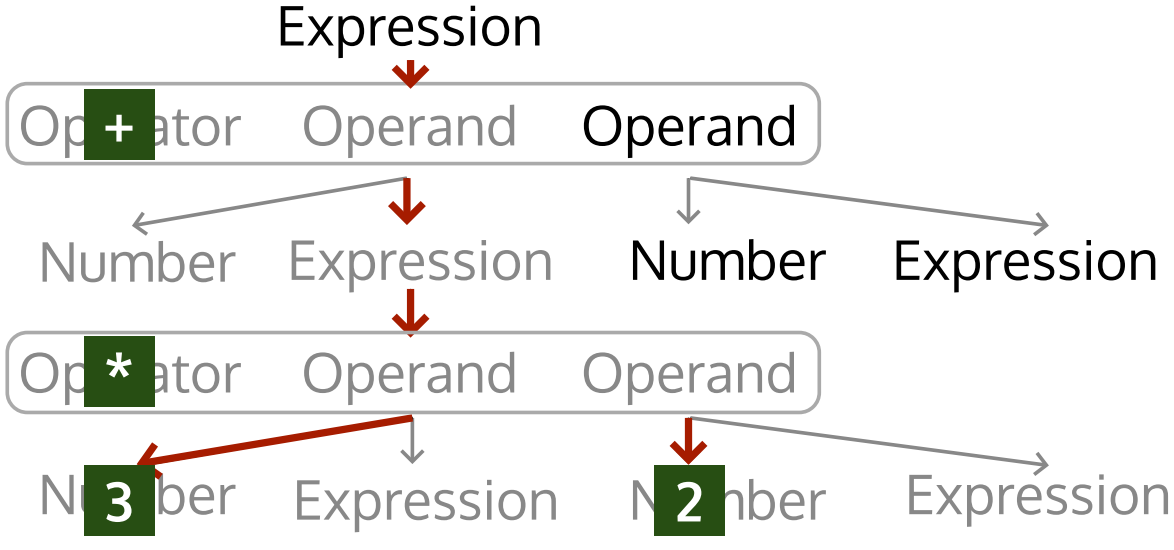
+ * 3 2 9



BNF – Beispiel

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

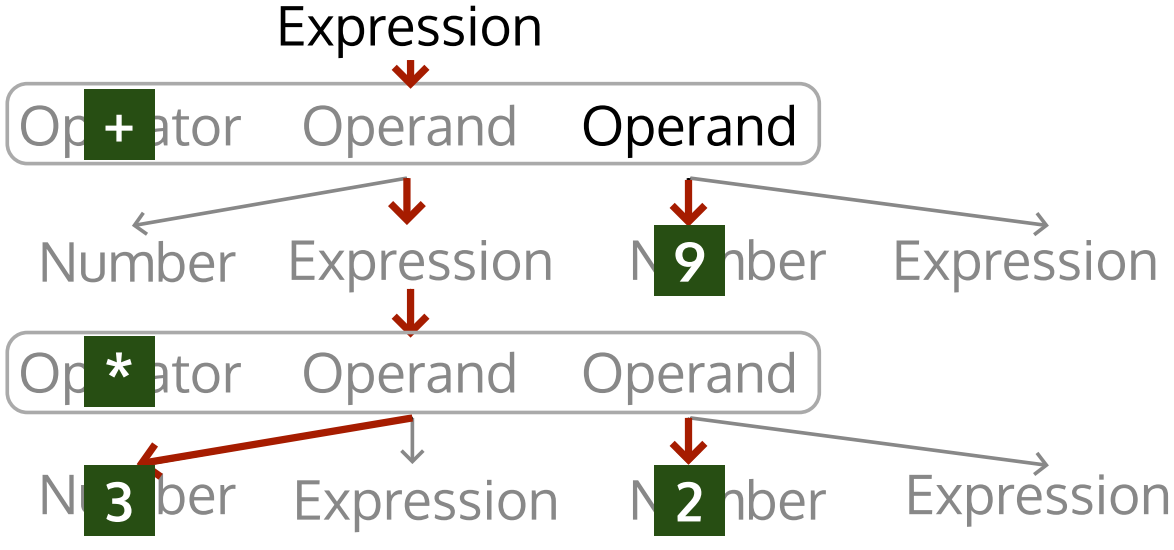
+ * 3 2 9



BNF – Beispiel

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

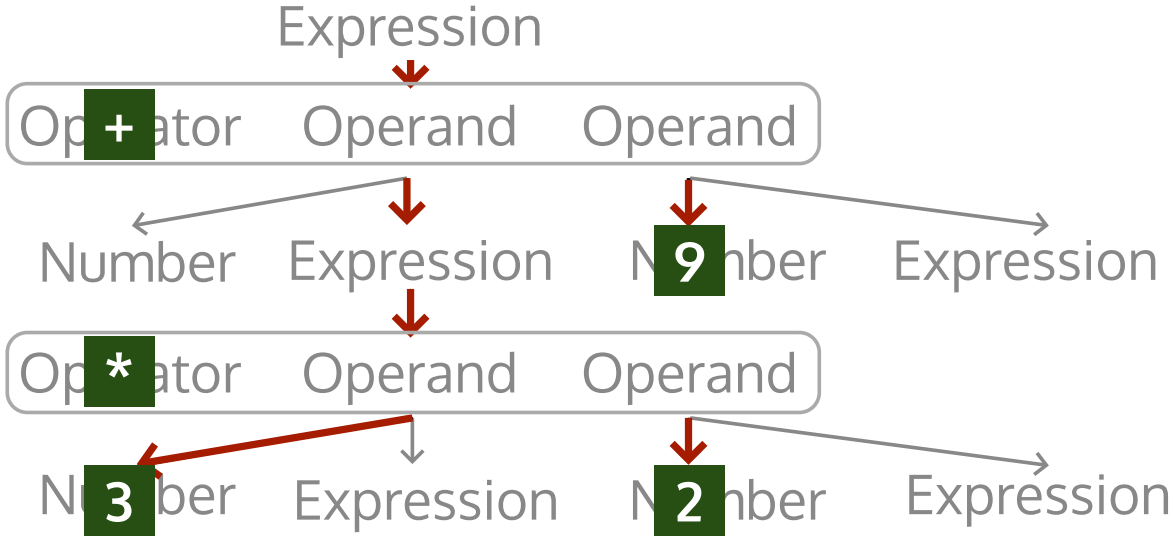
\+ * 3 2 9



BNF – Beispiel

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

\+ * 3 2 9



BNF – Beispiel

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

Entspricht der folgende String einer gültigen Expression?

/ 7 5 - 0

BNF – Beispiel

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

/ 7 5 - 0 Expression

BNF – Beispiel

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

/ 7 5 - 0

Expression

Operator Operand Operand

BNF – Beispiel

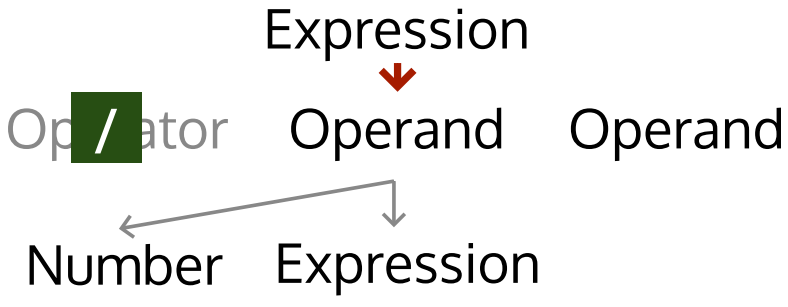
```
1 Expression = Operator ' ' Operand ' ' Operand
2 Operator   = '+' | '-' | '/' | '*'
3 Operand    = Number | Expression
4 Number     = Integer ['.' Integer]
5 Integer    = Digit { Digit }
6 Digit= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

/ 7 5 - 0

BNF – Beispiel

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

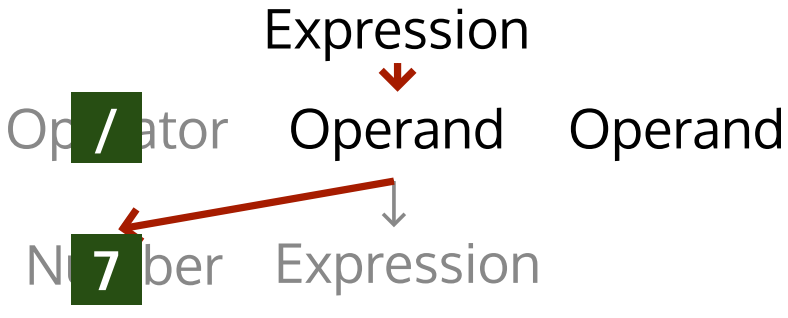
/ 7 5 - 0



BNF – Beispiel

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

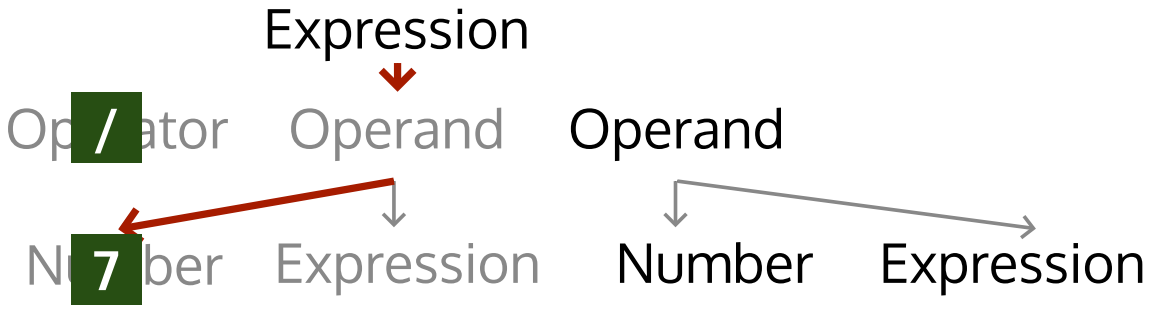
/ 7 5 - 0



BNF – Beispiel

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

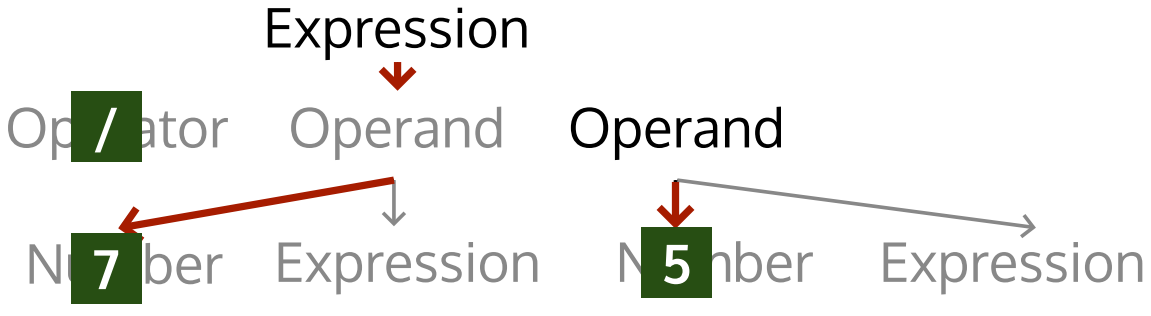
/ 7 5 - 0



BNF – Beispiel

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

/ 7 5 - 0



BNF – Beispiel

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

/ 7 5 - 0

Die BNF Konnte nicht den ganzen String Parsen.

→ invalid

EBNF in C++

- Für jede Regel erstellen wir eine Funktion mit Rückgabetyp *bool* und einem *instream* als Übergabetyp.
- Wir returnen *true*, falls unsere Expression am Anfang vom String steht
- Ansonsten returnen wir *false*

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

EBNF in C++

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

```
 1 // POST: returns true if a digit could
 2 //  be consumed from in
 3 // digit= '0' | '1' | '2' | '3' | '4' |
 4 //          '5' | '6' | '7' | '8' | '9'
 5 bool digit(std::istream& is){
 6     char next = lookahead(is);
 7     if(next >= '0' && next <= '9'){
 8         is >> next;
 9         return true;
10     }
11     return false;
12 }
```

112.3 + 2

112.3 + 2

true

+ 5 - 1 2.2

+ 5 - 1 2.2

false

EBNF in C++

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

```
 1 // POST: returns true if an unsigned integer
 2 //        could be consumed from in
 3 // integer = digit { digit }
 4 bool integer(std::istream& is){
 5     if(!digit(is)){
 6         return false;
 7     }
 8     while(digit(is));
 9     return true;
10 }
```

112.3 + 2

~~112~~.3 + 2

true

+ 5 - 1 2.2

+ 5 - 1 2.2

false

EBNF in C++

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

```
 1 // POST: returns true if an unsigned
 2 //       floating point number could
 3 //       be consumed from in
 4 // number = integer ['.' integer]
 5 bool number(std::istream& is){
 6     if(!integer(is)){
 7         return false;
 8     }
 9     if(lookahead(is) == '.'){
10         char dot;
11         is >> dot;
12         return integer(is);
13     }
14     else{
15         return true;
16     }
17 }
```

112.3 + 2

~~112.3~~ + 2

true

+ 5 - 1 2.2

+ 5 - 1 2.2

false

EBNF in C++

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

```cpp
// POST: returns true if an operator
//       could be consumed from in
// operator  = '+' | '-' | '/' | '*'
bool operator_(std::istream& is){
    std::vector<char> o = {'+','-','*','/'};
    char next = lookahead(is);
    for(unsigned int i = 0; i < 4; i++){
        if(next==o[i]){
            consume(is, next);
            return true;
        }
    }
    return false;
}
```

112.3 + 2

112.3 + 2

false

+ 5 - 1 2.2

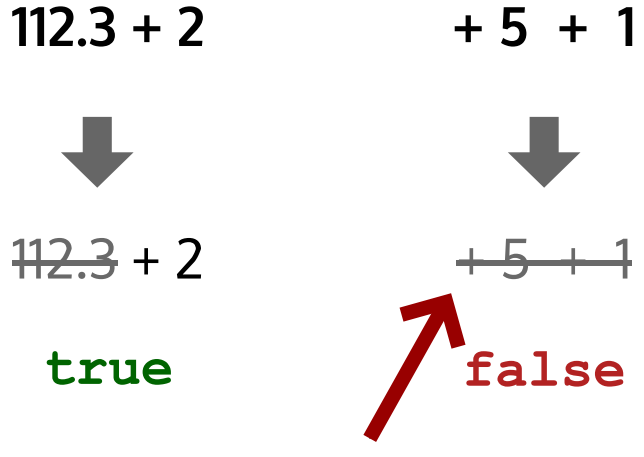
+ 5 - 1 2.2

true

EBNF in C++

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

```
 1 // POST: returns true if an operand could
 2 //       be consumed from in
 3 // operand   = number | expression
 4 bool operand(std::istream& is){
 5     char next = lookahead(is);
 6     if(next == '+' || next == '-' ||
 7        next == '*' || next == '/'){
 8         return expression(is);
 9     }
10     else{
11         return number(is);
12     }
13 }
```



Das Programm versuchte eine Expression zu parsen, aber es fehlt das Ende

EBNF in C++

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

```
// POST: returns true if an expression
//       could be consumed from in
// expression = operator_ operand operand
bool expression(std::istream& is){
    if(!operator_(is)) return false;
    if(!consume(is,' ')) return false;
    if(!operand(is)) return false;
    if(!consume(is,' ')) return false;
    if(!operand(is)) return false;
    return true;
}
```

112.3 + 2

112.3 + 2

false

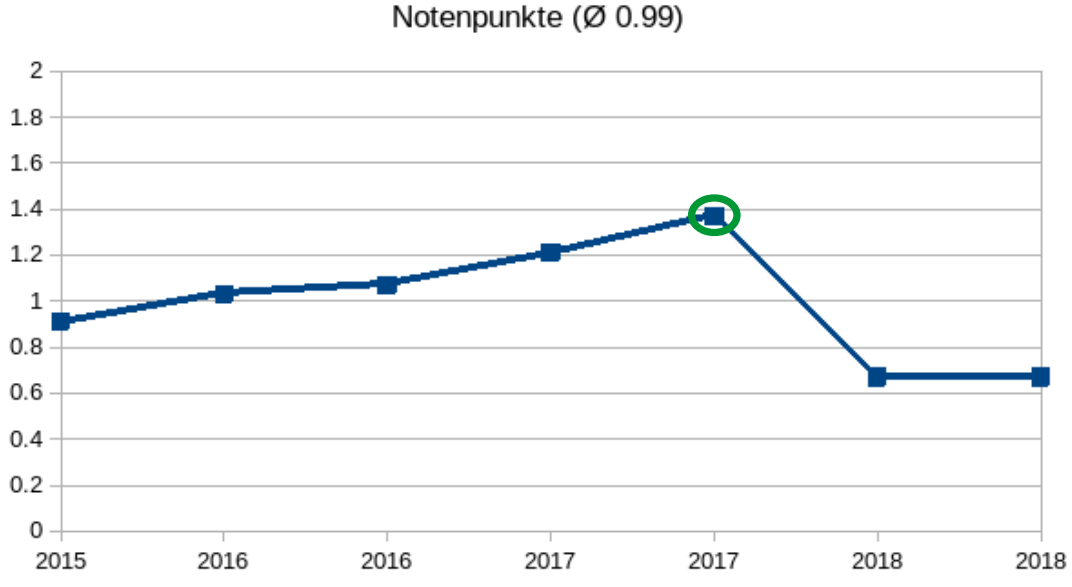
+ 5 - 1 2.2

~~+ 5 - 1 2.2~~

true

Prüfungsaufgabe

EBNF



Sommer 2017, 1.37 Notenpunkte

Prüfungsaufgabe

EBNF I

Basisprüfung Sommer 2017, 0.59 Notenpunkte

```
1  Statement = Expression ';'.
2  Expression = Simple [':' Simple].
3  Simple = Designator | Integer.
4  Designator = Identifier {'.' Identifier | '(' [ExpressionList] ')'}
5  ExpressionList = Expression { ',' Expression }.
6
7  Integer = Digit {Digit}
8  Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
9
10 Identifier = Letter { Letter }
11 Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' .
```

Welche der folgenden Zeichenketten sind gültige Statements gemäss der EBNF?

`a(2:5);` ✓

`a(3).3;` ✗

`a(b).c:d;` ✓

`a(3):3;` ✓

Prüfungsaufgabe

EBNF I

Basisprüfung Sommer 2017, 0.59 Notenpunkte

```
1  Statement = Expression ';'.
2  Expression = Simple [':' Simple].
3  Simple = Designator | Integer.
4  Designator = Identifier {'.' Identifier | '(' [ExpressionList] ')'}
5  ExpressionList = Expression { ',' Expression }.
6
7  Integer = Digit {Digit}
8  Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
9
10 Identifier = Letter { Letter | Digit }
11 Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' .
```

Ändern Sie genau eine Produktionsregel der EBNF ab, so dass folgende Anweisung (Statement) gültig wird:

```
a3(c3):a4(c4);
```

Prüfungsaufgabe

EBNF II

Basisprüfung Sommer 2017, 0.78 Notenpunkte

Wir haben diesen Code, mit welchem identifiziert werden soll, ob eine Zeichenkette im Sinne der vorherigen EBNF ein gültiges Statement darstellt.

```
bool Expression (std::istream& is);

// ExpressionList = Expression { ',' Expression }.
bool ExpressionList(std::istream& is){
    if (!Expression(is)) return false;

    // TODO

    return true;
}

// Designator = Identifier {'.' Identifier
//            | '(' [ExpressionList] ')'}.
bool Designator(std::istream& is){
    if (!Identifier(is)) return false;
    while(true){
        if (has(is, '.')){
            // TODO
        }
        else if (has(is, '(')){
            bool ignore = ExpressionList(is);
            // TODO
        } else
            return true;
    }
}
```

```
// Simple = Designator | Integer.
bool Simple(std::istream& is){
    return Integer(is) || Designator(is);
}

// Expression = Simple [':' Simple].
bool Expression(std::istream& is){
    if (!Simple(is)) return false;
    return // TODO ;
}

// Statement = Expression ';'.
bool Statement(std::istream& is){
    return Expression(is) && has(is, ';');
}
```

Prüfungsaufgabe

EBNF II

Basisprüfung Sommer 2017, 0.78 Notenpunkte

Folgender Code, welcher nur als Funktionsdeklaration vorliegt, soll als korrekt implementiert vorausgesetzt werden:

```
1  // POST: when the next available non-whitespace character equals c,
2  //it is consumed and the function returns true,
3  //otherwise the function returns false.
4  bool has(std::istream& is, char c);
5
6  // Integer = Digit {Digit}.
7  bool Integer (std::istream& is);
8
9  // Identifier = Letter {Letter}.
10 bool Identifier (std::istream& is);
```

Prüfungsaufgabe

EBNF II

Basisprüfung Sommer 2017, 0.78 Notenpunkte

Warum wird die Funktion *Expression* anscheinend zweimal deklariert?

Es gibt eine zirkuläre Abhängigkeit zwischen den Funktionen. Daher muss mindestens eine Funktion vor ihrer Definition deklariert werden (forward declaration).

Sonst wäre sie in mindestens einer anderen Funktion nicht sichtbar.

Prüfungsaufgabe

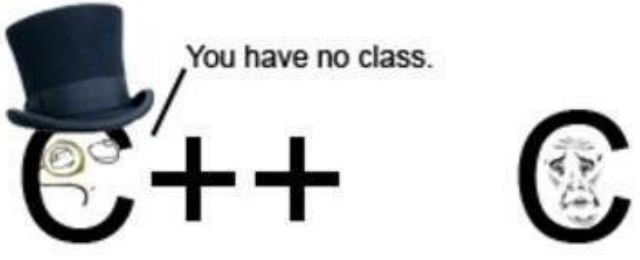
EBNF II

Basisprüfung Sommer 2017, 0.78 Notenpunkte

```
bool Expression (std::istream& is);

// ExpressionList = Expression { ',' Expression }.
bool ExpressionList(std::istream& is){
    if (!Expression(is)) return false;
    while (has(is, ',')){
        if (!Expression(is)) return false;
    }
    return true;
}

// Designator = Identifier {'.' Identifier
//              | '(' [ExpressionList] ')'}.
bool Designator(std::istream& is){
    if (!Identifier(is)) return false;
    while(true){
        if (has(is, '.')){
            if (!Identifier(is)) return false;
        }
        else if (has(is, '(')){
            bool ignore = ExpressionList(is);
            if (!has(is,')')) return false;
        } else
            return true;
    }
}
```

```
// Simple = Designator | Integer.
bool Simple(std::istream& is){
    return Integer(is) || Designator(is);
}

// Expression = Simple [':' Simple].
bool Expression(std::istream& is){
    if (!Simple(is)) return false;
    return !has(is,':') || Simple(is) ;
}

// Statement = Expression ';'.
bool Statement(std::istream& is){
    return Expression(is) && has(is, ';');
}
```

Structs & Klassen



Objekte

- Abgesehen von den Primitiven Datentypen gibt es auch komplexere Datentypen.
- Diese können mehr Funktionalität beinhalten.
- Diese Typen sind dem Compiler nicht standardmässig bekannt
  -> Sie müssen entweder selbst geschrieben oder importiert werden

Vektor

- Ein Beispiel für ein solches Objekt ist der Vektor
- Vor der Verwendung muss dieser mit #include<vector> importiert werden.
- Beim erstellen von einem neuen Vektor muss immer noch der zu speichernde Datentyp angegeben werden.
- Bei Bedarf kann auch die Anzahl Elemente, der Initialisierungswert oder eine Liste gegeben werden.

Beispiele:

```
1 #include <vector>
2
3 std::vector<int> a;           // empty vector of ints
4 std::vector<float> b(5);      // five floats with value 0
5 std::vector<double> c(5,3);   // five doubles with value 3
6 std::vector <int> d = {1, 2};// two integers with values 1, 2
```

Vektor

- Mit der Funktion *at* oder dem [] operator kann man auf Elemente im Vektor zugreifen.
- Wir haben Indizierung bei 0, das heisst das vorderste Element hat den Index 0.

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

int main(){
    std::vector<char> v = {'<', '1', '3', 'n', 'i', ' ', 'f'};
    std::cout << v[4] << v[3] << v.at(6) << v[1] << v.at(5) << v[0] << v[2];
    return 0;
}
```

Output

```
inf1 <3
```

Vektor

Vektoren sind *keine* Primitive Datentypen. Sie haben noch weitere Funktionalitäten:

Name	Returns
`size`	Grösse (Länge) vom Vektor
`empty`	*true* falls Vektor leer ist, sonst *false*
`push_back`	Fügt Element am ende dazu
`pop`	Löscht letztes Element
`begin`	Iterator auf Start vom Vektor
`end`	Iterator auf Ende vom Vektor

Eine komplette Liste findet ihr hier unter "Member Functions"

Structs

- Structs erlauben uns eigene "Datentypen" zu erstellen.
- Sie sind eine Kollektion von Variablen und Funktionen
- Nach der Deklaration braucht es ein Semicolon ;

```
struct my_struct_t{
    int n;
    bool b = true;
};
```

Unser eigener Datentyp mit dem Namen my_struct_t

Der Struct beinhaltet *eine Variable n*

Die Variable b hat standardmässig den Wert *true*

Semicolon nach der Deklaration

Initialisierung

- Einen Struct kann man initialisieren in dem man alle Initialisierungswerte in geschwungenen {} Klammern schreibt.
- Diese Werte werden dann in der entsprechenden Reihenfolge in die Variablen geschrieben
- Elemente können mit "." ausgelesen werden

```
struct my_struct_t{
    int n;
    bool b = true;
};
```

```
int main(){
    my_struct_t s = {2, true};
    if(s.b){
        s.n = 3;
    }
}
```

Initialisierung mit {} Klammern

Auf b kann mit s.b zugegriffen werden

Man kann die Variablen schreiben und lesen

Initialisierung

- Man kann auch eine Funktion schreiben, welche zum erstellen von einem Struct dient.
- Diese Funktion nennt man Konstruktor.
- Sie hat denselben Namen wie der Struct und sie hat keinen Rückgabetyp.

```
struct my_struct_t{
    int n;
    bool b = true;
    my_struct_t(int a){
        n = a;
    }
};
```

Konstruktor hat keinen Rückgabetyp und denselben Namen wie der Struct

```
int main(){
    my_struct_t s(2);
    std::cout << s.n << " " << s.b;
}
```

Bei der Initialisierung wird der Konstruktor aufgerufen

Output

```
2 1
```

Initialisierung

- Die Initialisierung von Variablen kann abgekürzt werden
- Man Initialisiert alle Variablen auf der selben Linie wie der Konstruktor.

```
struct my_struct_t{
    int n;
    bool b;
    my_struct_t(int a): n(a), b(true){}
};
```

Die Variable *n* wird mit *a* initialisert und die Variable *b* wird mit *true* initialisiert.

```
int main(){
    my_struct_t s(2);
    std::cout << s.n << " " << s.b;
}
```

Der Konstruktor kann immer noch gleich aufgerufen werden.

Output

```
2 1
```

Funktionen

- Structs können auch Funktionen beinhalten
- Diese können genau gleich wie die Variabeln mit einem "." benutzt werden

```
struct my_struct_t{
    int n;
    bool b;
    my_struct_t(int a): n(a), b(true){}
    void print_conditional(){
        if(b){
            std::cout << this->n;
        }
    }
};
```

Wir definieren eine Funktion innerhalb von unserem Struct

Mit dem Keyword this bekommen wir einen Pointer auf uns selbst

```
int main(){
    my_struct_t s(2);
    s.print_conditional();
}
```

Diese Funktion können wir dann auf ein bestimmtes Objekt angewendet aufrufen.

Output

2

Kapselung

- Mit Kapselung ist es möglich zu steuern, wer alles Zugriff auf ein Member (Variable / Funktion) hat.
- Das Keyword ***private*** macht, dass alle folgenden Funktionen / Variablen nur von innerhalb vom Objekt benutzt werden können.
- Das Keyword ***public*** macht, dass alle folgenden Funktionen / Variablen auch von ausserhalb vom Objekt benutzt werden können.

Kapselung

```
struct my_struct_t{
private:
    int n;
    bool b;
public:
    my_struct_t(int a): n(a), b(true){}
    void print_conditional(){
        if(b){
            std::cout << n;
        }
    }
};
```

```
int main(){
    my_struct_t s(2);
    std::cout << s.n << " " << s.b;
}
```

```
int main(){
    my_struct_t s(2);
    s.print_conditional();
}
```

Output

2

Klassen

- Klassen können in C++ genau gleich wie Structs verwendet werden.
- Der Unterschied ist, dass Members von Klassen standardmässig private sind, während Members von Structs standardmässig public sind.

Überladung

Überladung

Function Overloading

In C++ ist es möglich, dass mehrere Funktionen den gleichen Namen haben, solange sich die Übergabewerte unterscheiden.

richtig

```
int foo1(int a){ ... }
int foo1(int a, int b){ ... }
```

```
int foo2(int a){ ... }
int foo2(float a){ ... }
```

falsch

```
int foo3(int a){ ... }
int foo3(int b){ ... }
```

Es reicht nicht, wenn sich nur die Variabelnamen unterscheiden.

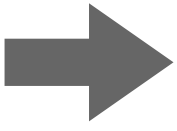
```
int foo4(int a){ ... }
float foo4(int a){ ... }
```

Es reicht nicht, wenn sich nur der Rückgabetyp unterscheidet.

Überladung

Function Overloading

```
#include <iostream>

void out (const int i) {
    std::cout << i << " (int)\n";
}

void out (const double i) {
    std::cout << i << " (double)\n";
}

int main () {
    out(3.5);
    out(2);
    out(2.0);
    out(0);
    out(0.0);
    return 0;
}
```



```
3.5 (double)
2 (int)
2 (double)
0 (int)
0 (double)
```

Überladung

Operator Overloading

Genau wie Funktionen können auch Operatoren wie **+-*/** überschrieben werden.

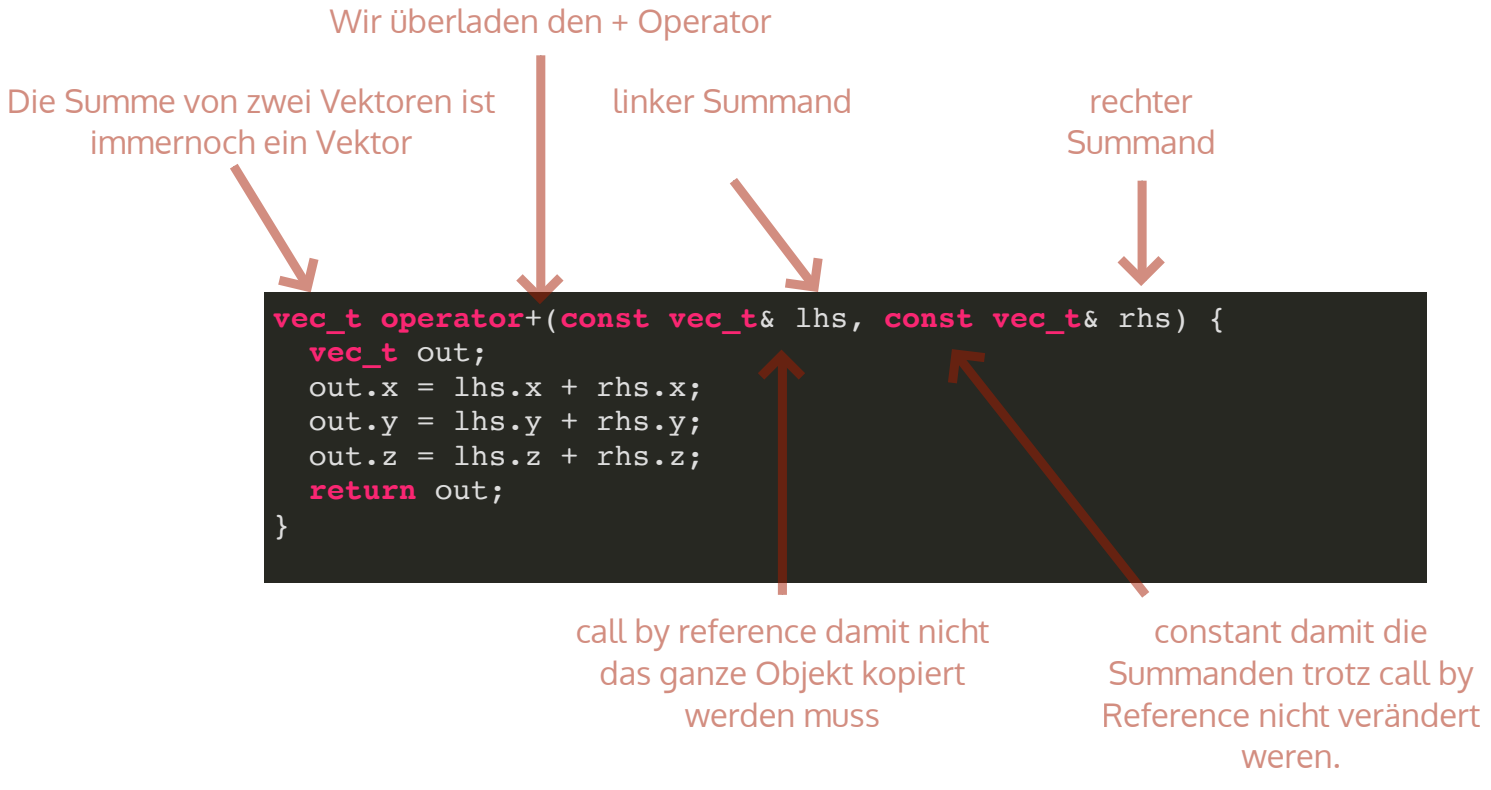
Als Beispiel überladen wir Operatoren für diesen Struct, welcher Punkte im dreidimensionalen Koordinatensystem darstellt:

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

Überladung

Addition

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



Überladung

Output

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

Gibt den Vektor als String aus: zB *(1,2.5,3.5)*

Überladung

Output

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

Warum müssen wir wieder einen Outstream als Return-Wert haben?

```
std::cout << vec1 << vec2;
```

Dies ist nötig, damit wir mehrere Objekte auf derselben Zeile ausgeben können.

Überladung

Output

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

Warum müssen wir wieder einen Outstream als Return-Wert haben?

```
std::cout << vec1 << vec2;
```

Weil der << Operator von links nach rechts ausgewertet wird, wird zuerst vec1 eingelesen.

```
1 std::ostream& operator<< (std::ostream &out, const vec_t& vec) {
2     out << "(" << vec.x << "," << vec.y << "," << vec.z << ")";
3     return out;
4 }
```

Überladung

Output

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

Warum müssen wir wieder einen Outstream als Return-Wert haben?

```
std::cout << vec1 << vec2;
```

Weil der << Operator von links nach rechts ausgewertet wird, wird zuerst vec1 eingelesen.

```
1 std::ostream& operator<< (st std::cout , c vec1 ) {
2     out << "(" << vec.x << "," << vec.y << "," << vec.z << ")";
3     return out;
4 }
```

Überladung

Output

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

Warum müssen wir wieder einen Outstream als Return-Wert haben?

```
std::cout  << vec2;
```

Nach dem vec1 ausgegeben wurde wird unser Outstream, in diesem Beispiel std::cout zurückgegeben.

```
1 std::ostream& operator<< (std::ostream &out, const vec_t& vec) {
2     out << "(" << vec.x << "," << vec.y << "," << vec.z << ")";
3     return out;
4 }
```

Überladung

Output

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

Warum müssen wir wieder einen Outstream als Return-Wert haben?

```
std::cout  << vec2;
```

So haben wir wieder einen Outstream, mit welchem wir vec2 ausgeben können.

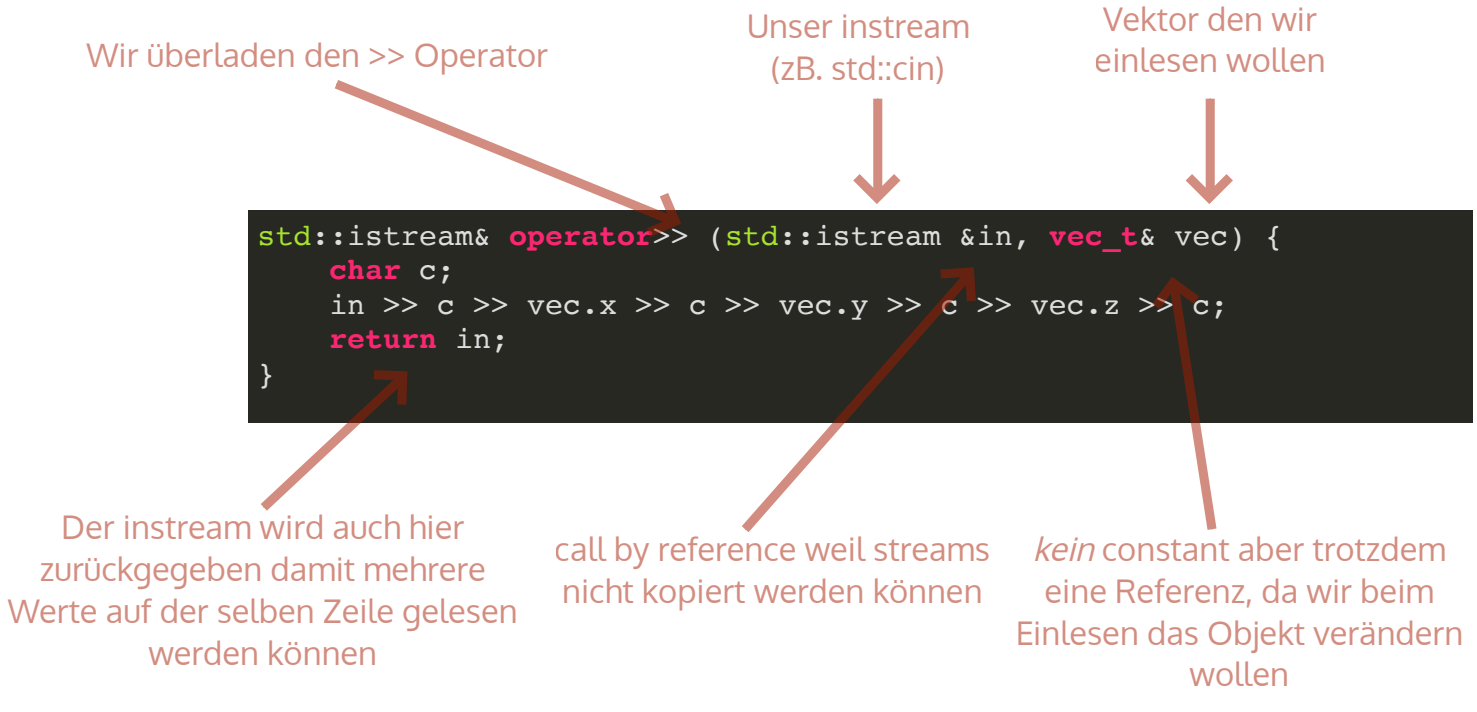
```
1 std::ostream& operator<< (st std::cout , c vec2 ) {
2     out << "(" << vec.x << "," << vec.y << "," << vec.z << ")";
3     return out;
4 }
```

Überladung

Input

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

Liest einen Vektor ein zB *(1,2.5,3.5)*



Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

Wir definieren einen wegwerf Charakter c, welcher alle Klammern und Kommas "schluckt"

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

(1,2.5,3.5)

char c

?

vec_t vec

x = ?

y = ?

z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

(1,2.5,3.5)

char c

'('

vec_t vec

x = ?

y = ?

z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1~~,2.5,3.5)

char c

'('

vec_t vec

x = 1

y = ?

z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1,~~2.5,3.5)

char c

','

vec_t vec

x = 1

y = ?

z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1,2.5~~,3.5)

char c

','

vec_t vec

x = 1

y = 2.5

z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1,2.5,~~3.5)

char c

','

vec_t vec

x = 1
y = 2.5
z = ?

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1,2.5,3.5)~~

char c

','

vec_t vec

x = 1

y = 2.5

z = 3.5

Überladung

Input

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

```
1 std::istream& operator>> (std::istream &in, vec_t& vec) {
2     char c;
3     in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
4     return in;
5 }
```

std::instream & in

~~(1,2.5,3.5)~~

char c

')'

vec_t vec

x = 1
y = 2.5
z = 3.5

Überladung

Operator Overloading

Das Überladen dieser Operatoren erlaubt es uns unsere eigenen Objekte mit den normalen Operatoren zu manipulieren:

```cpp
#include <iostream>

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

vec_t operator+(const vec_t& lhs, const vec_t& rhs) {
  vec_t out;
  out.x = lhs.x + rhs.x;
  out.y = lhs.y + rhs.y;
  out.z = lhs.z + rhs.z;
  return out;
}

std::ostream& operator<< (std::ostream &out, vec_t vec) {
    out << "(" << vec.x << "," << vec.y << "," << vec.z << ")";
    return out;
}

std::istream& operator>> (std::istream &in, vec_t& vec) {
    char c;
    in >> c >> vec.x >> c >> vec.y >> c >> vec.z >> c;
    return in;
}
```

```cpp
int main () {
  vec_t a, b;
  std::cin >> a >> b;
  std::cout << a + b << std::endl;
}
```

Input:

```
(1,2.5,3.5)
(1,2,3)
```

Output:

(2,4.5,6.5)

In-Class Overloading

- Operatoren können auch in-class Überladen werden.
- Diese wird für das Objekt auf der linken Seite des Operators aufgerufen

Wir überladen den -= Operator

Es wird ein L-Wert zurückgegeben

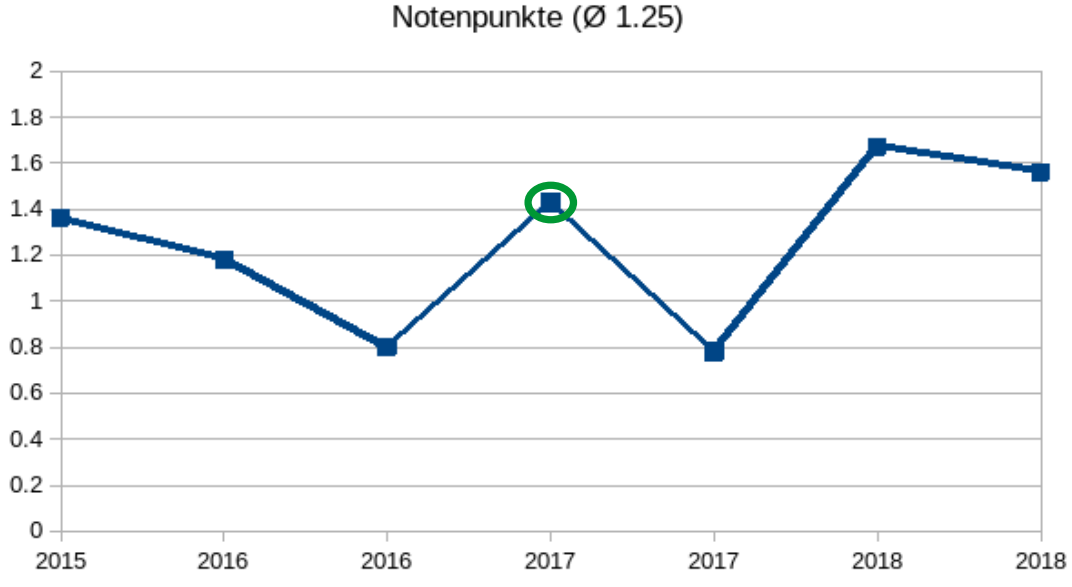
```
struct vec_t {
  double x;
  double y;
  double z;

  vec_t& operator-=(const vec_t& v){
    x -= v.x;
    y -= v.y;
    z -= v.z;
    return *this;
  }
};
```

Wir können unsere eigenen Variabeln ohne prefix verändern

Der Rückgabewert ist das Objekt selbst

Prüfungsaufgabe

Klassen/Datenstrukturen



Winter 2017, 0.83 Notenpunkte

Prüfungsaufgabe

Klassen und Operator Overloading

Basisprüfung Winter 2017, 0.83 Notenpunkte

Überladet die nötigen Operatoren, damit die Mainfunktion so funktioniert.

```
class Time{
    int sec; // total number of seconds
    public:
    // construct time from hours, minutes and seconds
    Time (int h, int m, int s): sec(s+m*60+h*3600){}

    // write time to stream out
    void write (std::ostream& out) const {
        out << sec / 3600 % 24 << ":"
            << sec / 60 % 60 << ":" << sec % 60;
    }
};
```

Output:

```
14:4:58
14:4:59
14:5:0
14:5:1
...
14:7:2
```

```
int main(){
    Time second (0,0,1);
    Time examStart (14,04,58);
    Time examDuration (01,02,05);
    Time examEnd = examStart + examDuration;
    for (Time t=examStart; t != examEnd; t += second){
        std::cout << t << std::endl;
    }

    return 0;
}
```

Prüfungsaufgabe

Klassen und Operator Overloading

Basisprüfung Winter 2017, 0.83 Notenpunkte

```
class Time{
    int sec; // total number of seconds
    public:
    // construct time from hours, minutes and seconds
    Time (int h, int m, int s): sec(s+m*60+h*3600){}

    // add t to this time
    Time& operator+= (const Time& t){
        sec += t.sec;
        return *this;
    }

    // two times match when they represent the same time during a day
    // Example: 0:0:5 must match 24:0:5
    bool operator==(const Time& t){
        return sec % (3600*24) == t.sec % (3600*24);
    }

    // write time to stream out
    void write (std::ostream& out) const {
        out << sec / 3600 % 24 << ":"
            << sec / 60 % 60 << ":" << sec % 60;
    }
};
```

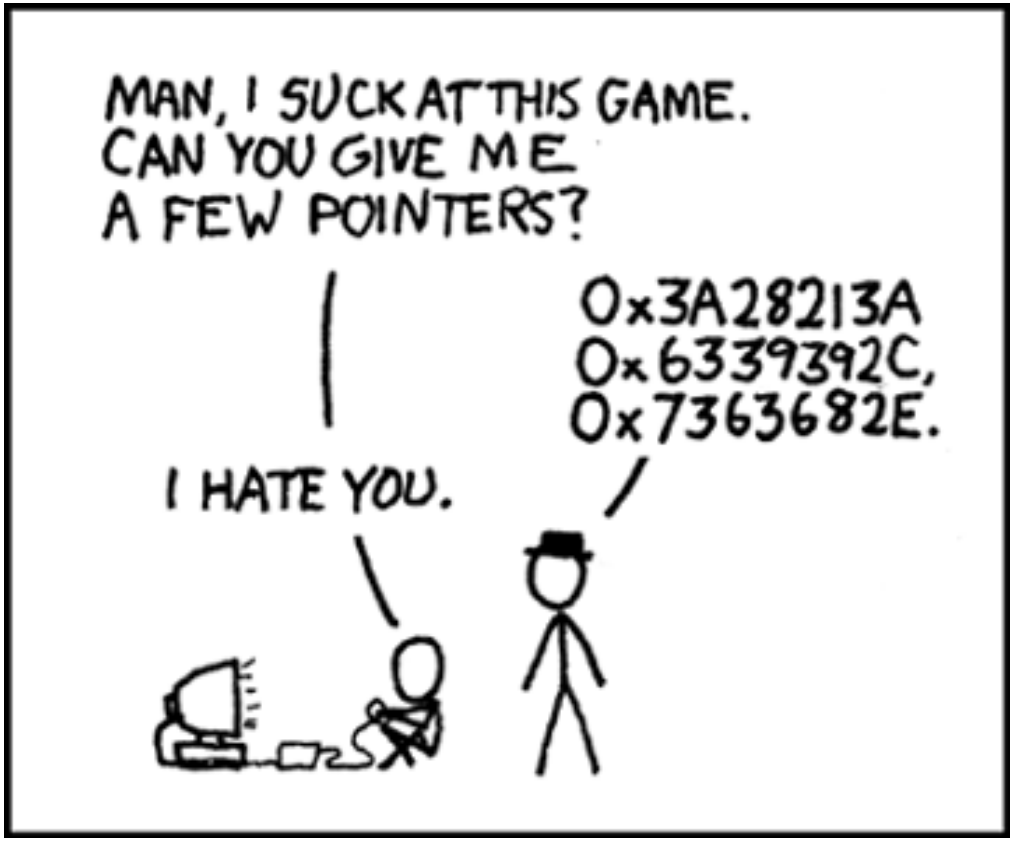
Prüfungsaufgabe

Klassen und Operator Overloading

Basisprüfung Winter 2017, 0.83 Notenpunkte

```
// output time on stream o
std::ostream& operator<< (std::ostream& o, const Time t){
    t.write(o);
    return o;
}

// return if times are not equal
bool operator!=(const Time& t1, const Time& t2){
    return !(t1 == t2);
}

// add time t1 and time t2
Time operator+(Time t1, const Time & t2){
    return t1 += t2;
}
```

Pointer / Iteratoren



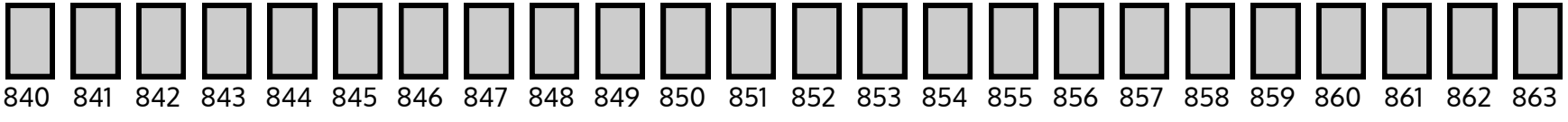
Pointers

- Der Speicher in Computern ist in Bytes (8 Bit) aufgeteilt.
- Jedes Byte hat eine Adresse
- Ein Pointer speichert nicht einen Wert sondern die Adresse von einem Wert.

```
int a = 5;
int* x = &a;

char c = 'd';
char* z = &c;
```

Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```

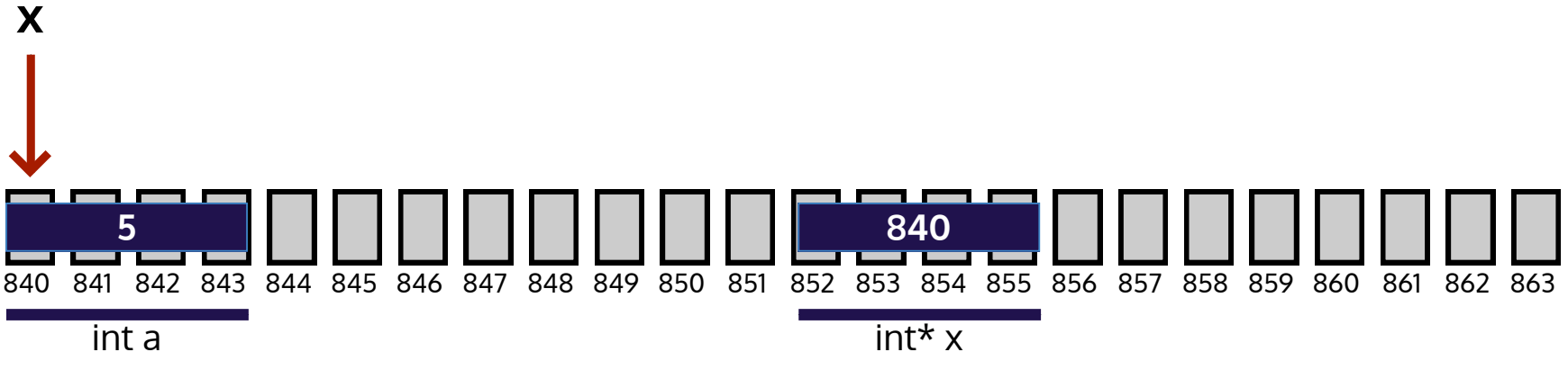


Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```

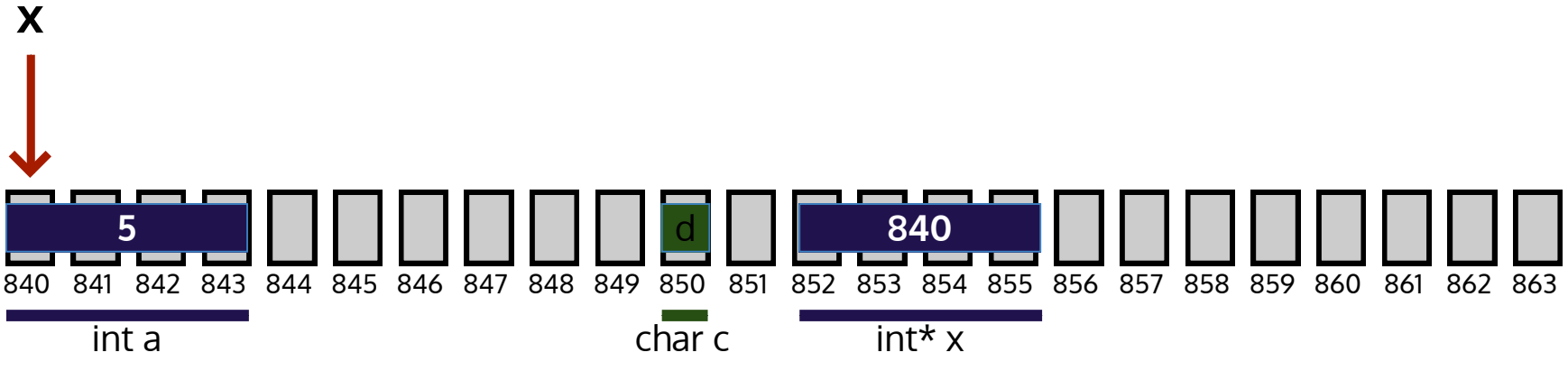
Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```



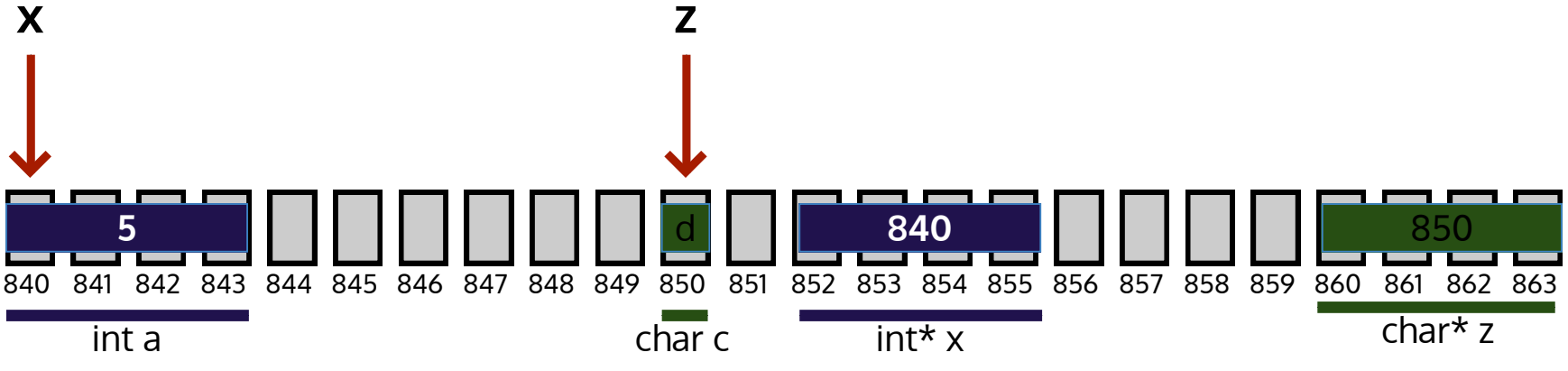
Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```



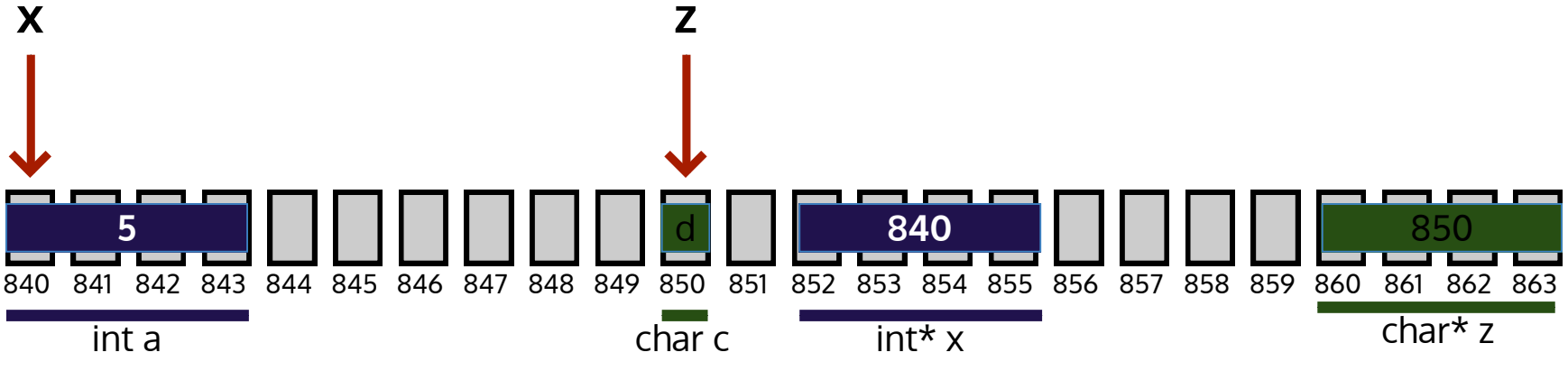
Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```



Pointers

```
1  int a = 5;
2  int* x = &a;
3
4  char c = 'd';
5  char* z = &c;
```



Operatoren

&

1. Bitwise AND

```
z = x & y;
```

2. Variabel als Referenz deklarieren

```
int& y = x;
```

3. Adresse einer Variabel herausfinden

```
int* ptr_a = &a;
```

*

1. Multiplikation

```
z = x * y;
```

2. Variabel als Pointer deklarieren

```
int* ptr_a = &a;
```

3. Inhalt von einem Pointer herausfinden

```
int a = *ptr_a;
```

New & Delete

- Mit dem Keyword *new* kann man Pointer erstellen, welche auf Daten zeigen die niemals out of scope gehen.
- Im Gegenzug muss man selber die Daten mit dem Keyword *delete* löschen falls man diese nicht mehr braucht.

New & Delete

```cpp
#include <iostream>

int* create_value(){
    int x = 5;
    return &x;
}

int main(){
    int* ptr = create_value();

    std::cout << *ptr << "\n";
    return 0;
}
```

```
warning: address of local variable
‘x’ returned

Segmentation fault (core dumped)
```

```cpp
#include <iostream>

int* create_value(){
    int* x = new int(5);
    return x;
}

int main(){
    int* ptr = create_value();

    std::cout << *ptr << "\n";

    delete ptr;

    return 0;
}
```

```
5
```

New & Delete (Arrays)

```cpp
#include <iostream>

int* create_array(){
    int* a = new int[5];

    for(unsigned int i = 0; i < 5; i++){
        *(a + i) = i;
    }

    return a;
}

int main(){
    int* a = create_array();

    std::cout << *a << " ";
    std::cout << *(a + 1) << " ";
    std::cout << *(a + 2) << " ";
    std::cout << *(a + 3) << " ";
    std::cout << *(a + 4) << " ";

    delete[] a;

    return 0;
}
```

Stellt speicher für 5 Integer bereit

Pointer dereferenzieren um den dort gespiecherten Wert zu ändern

Pointer dereferenzieren um den dort gespiecherten Wert zu lesen

Output:

```
0 1 2 3 4
```

Arrays – Syntaktischer Zucker

`*(a + 1)` ⟺ `a[1]`

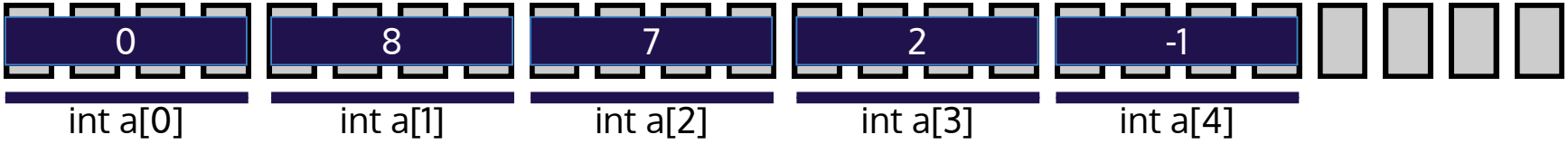
```cpp
#include <iostream>

int* create_array(){
    int* a = new int[5];

    for(unsigned int i = 0; i < 5; i++){
        *(a + i) = i;
    }

    return a;
}

int main(){
    int* a = create_array();

    std::cout << *a << " ";
    std::cout << *(a + 1) << " ";
    std::cout << *(a + 2) << " ";
    std::cout << *(a + 3) << " ";
    std::cout << *(a + 4) << " ";

    delete[] a;

    return 0;
}
```

⟺

```cpp
#include <iostream>

int* create_array(){
    int* a = new int[5];

    for(unsigned int i = 0; i < 5; i++){
        a[i] = i;
    }

    return a;
}

int main(){
    int* a = create_array();

    std::cout << a[0] << " ";
    std::cout << a[1] << " ";
    std::cout << a[2] << " ";
    std::cout << a[3] << " ";
    std::cout << a[4] << " ";

    delete[] a;

    return 0;
}
```

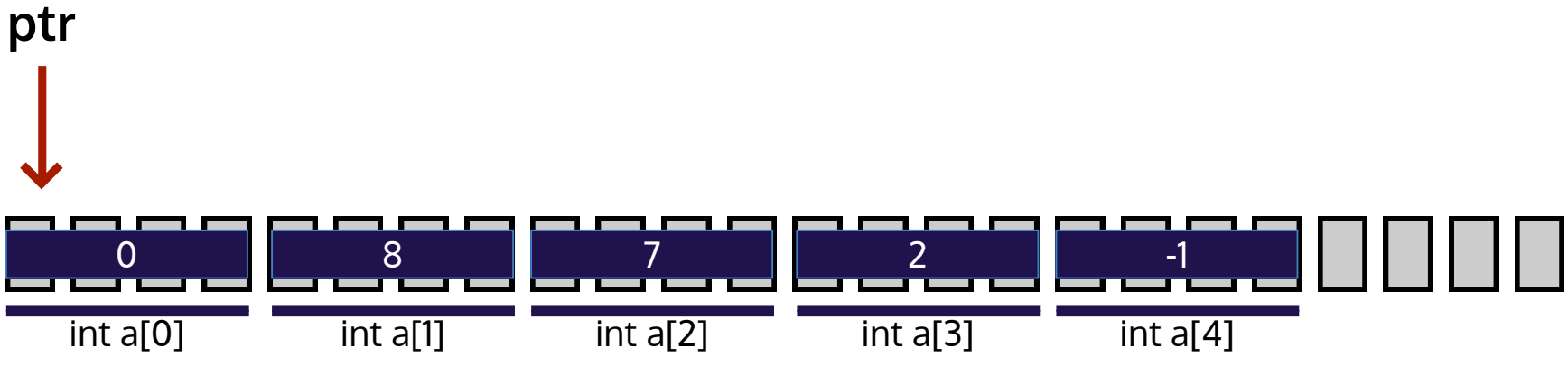
Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



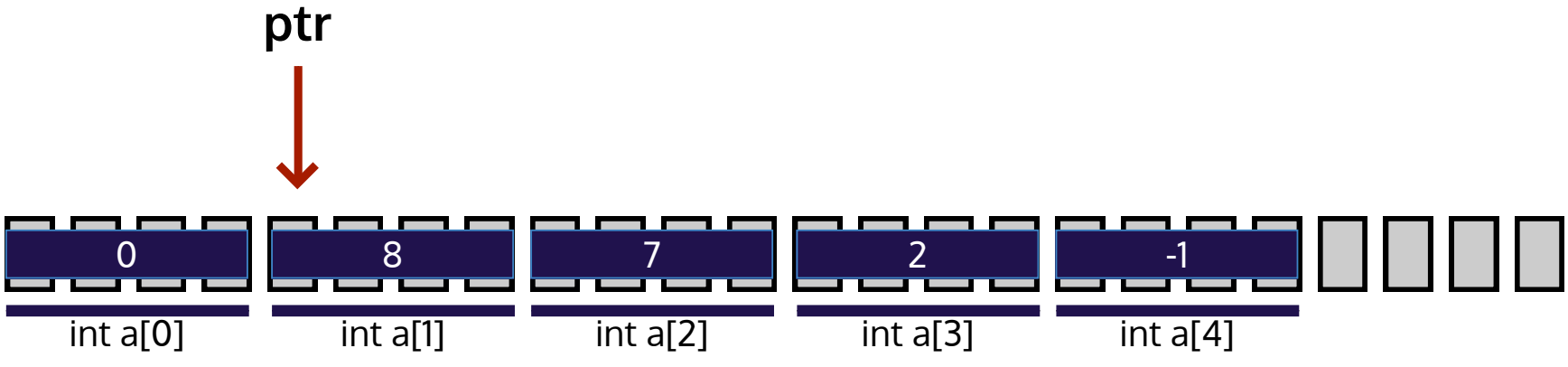
Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;          // pointer assignment
3 ++ptr;                 // Shift pointer to right
4 int my_int = *ptr;     // read target
5 ptr += 2;              // shift by 2 elements
6 *ptr = 18;             // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



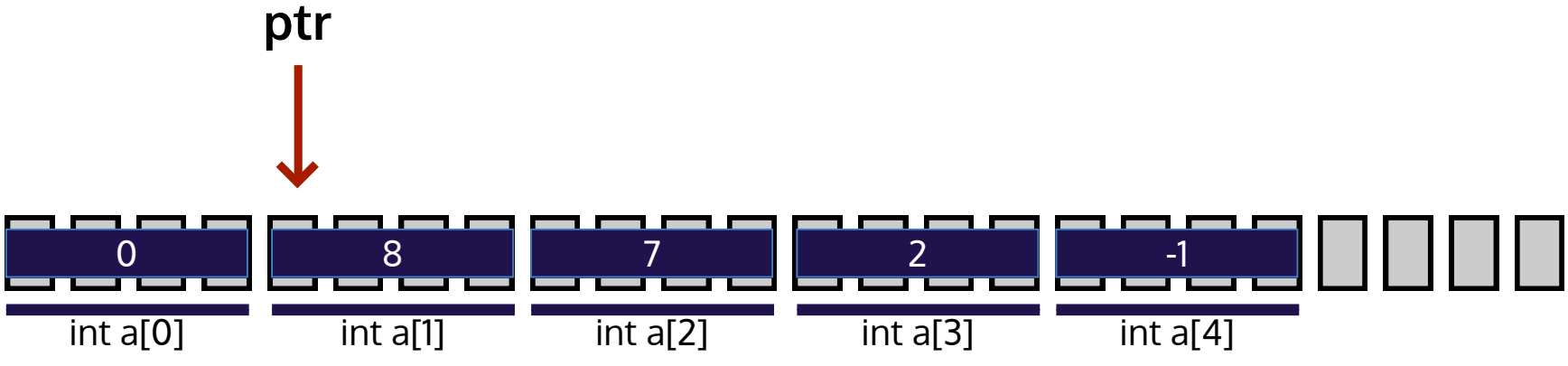
Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;          // pointer assignment
3 ++ptr;                 // Shift pointer to right
4 int my_int = *ptr;     // read target
5 ptr += 2;              // shift by 2 elements
6 *ptr = 18;             // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



Beispiel

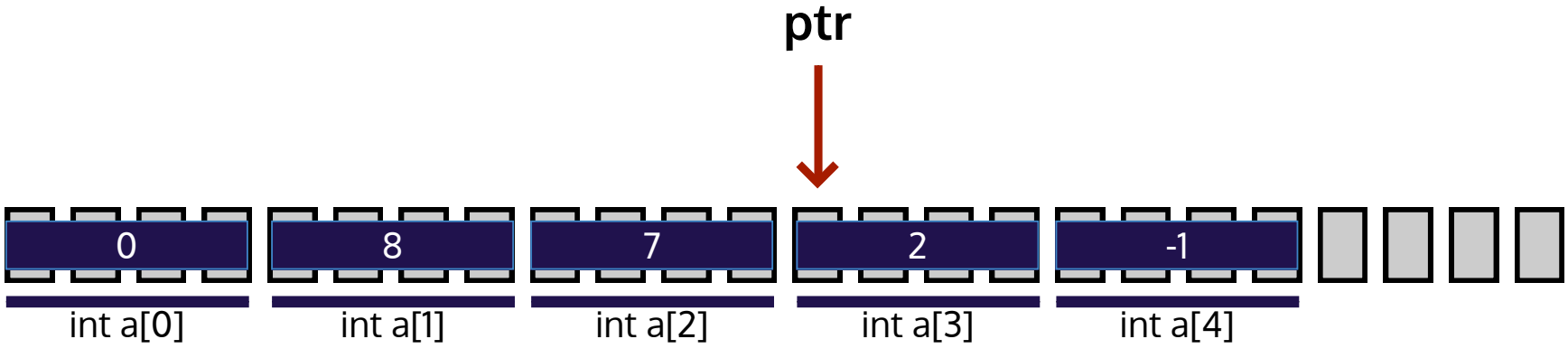
```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



my_int = 8

Beispiel

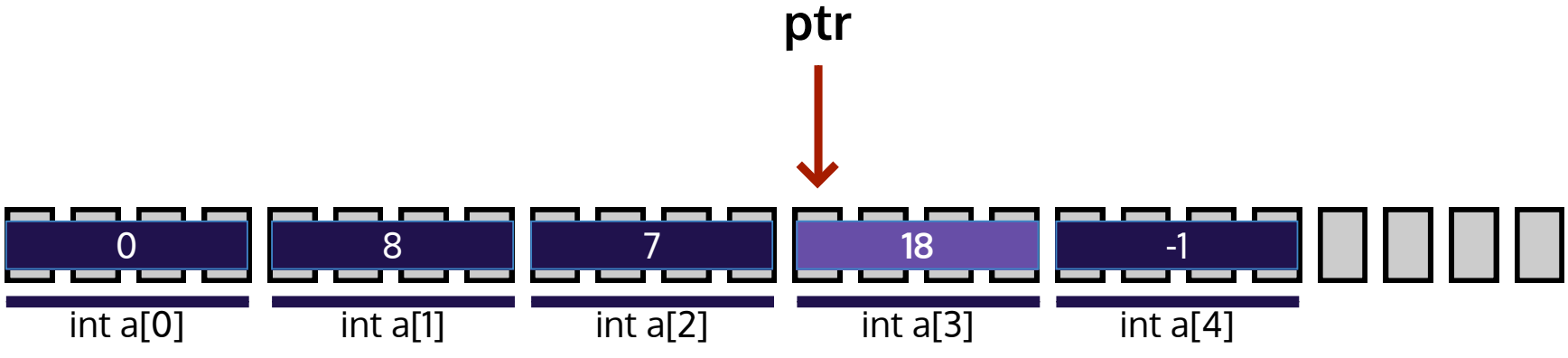
```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



my_int = 8

Beispiel

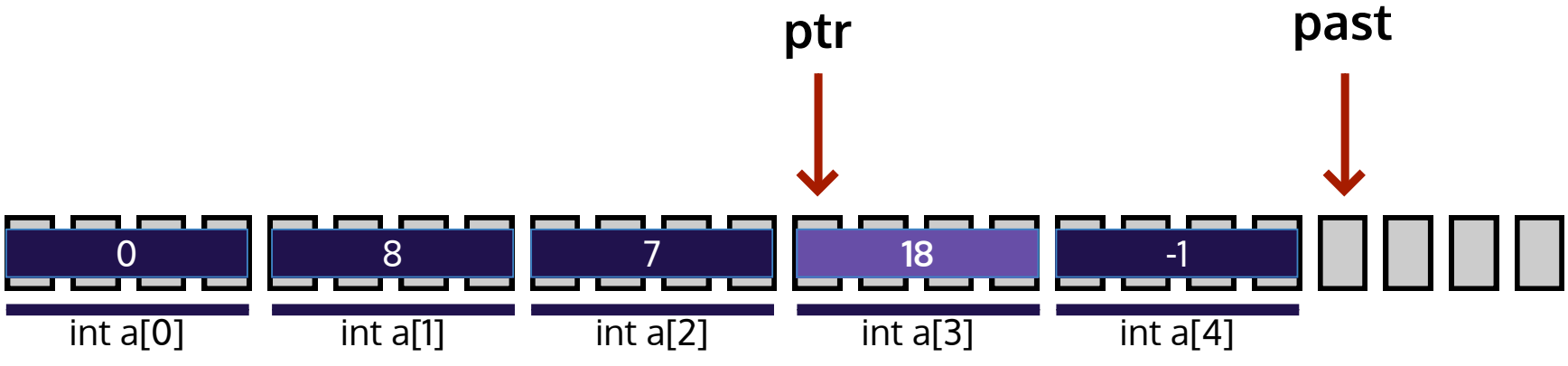
```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



my_int = 8

Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```



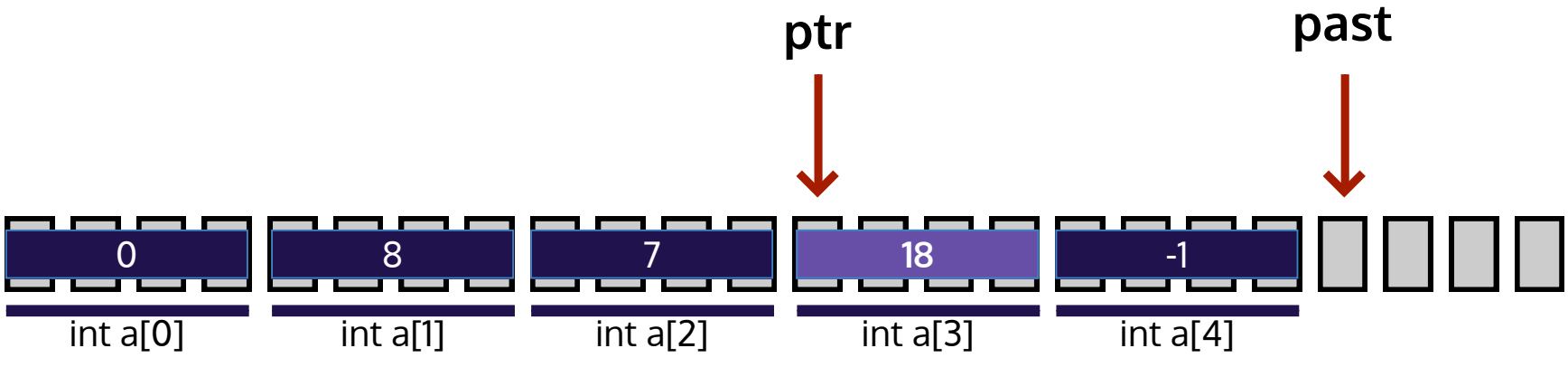
my_int = 8

Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```

Output:

```
8 1
```

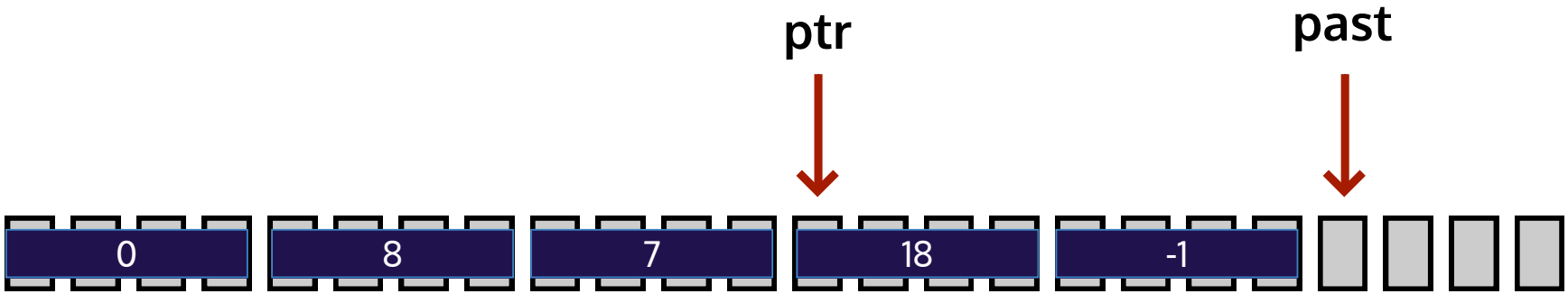


my_int = 8

Beispiel

```
1 int* a = new int[5]{0, 8, 7, 2, -1};
2 int* ptr = a;           // pointer assignment
3 ++ptr;                  // Shift pointer to right
4 int my_int = *ptr;      // read target
5 ptr += 2;               // shift by 2 elements
6 *ptr = 18;              // overwrite target
7 int* past = a+5;
8 std::cout << my_int << " " << (ptr < past) << "\n";
9 delete[] a;
```

delete Löscht die Werte noch nicht, aber es gibt sie zum Überschreiben frei.



my_int = 8

Nullpointer

- Wie kann man es schaffen, dass ein Pointer auf nichts zeigt?
- Es gibt in C++ den sogenannten *null* Pointer, welcher verwendet werden kann um eine Variable effektiv auf nichts zeigen zu lassen.

```cpp
#include <iostream>

int main() {

    int* p = new int(10);
    std::cout << p << std::endl;

    delete p;
    std::cout << p << std::endl;

    p = nullptr;
    std::cout << p << std::endl;

    return 0;
}
```

```
0x55beafb36e70
0x55beafb36e70
0
```

Pointer auf Objekte

Beide Schreibweisen sind äquivalent, aber
das Rechte *(->)* wirkt eleganter

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

int main() {

    std::vector<int>* v = new std::vector<int>;

    (*v).push_back(3);

    std::cout << (*v).size() << std::endl;

    return 0;
}
```

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

int main() {

    std::vector<int>* v = new std::vector<int>;

    v->push_back(3);

    std::cout << v->size() << std::endl;

    return 0;
}
```

Container

- In C++ gibt es verschiedene Arten eine Sammlung von Daten zu speichern.
- Alle diese Arten nennt man Container
- Ihr kennt bereits ein paar Container: Arrays, Strings, Vektoren
- Es gibt aber noch andere Container:
  - Set
  - Queue
  - usw... *(Liste aller Container)*

Iteratoren

- Pointers erlauben es uns zusammenhängende Speicherbereiche zu traversieren
- Andere Container speichern aber je nach dem nicht alle Daten in einem zusammenhängendem Speicherbereich (zB. Linked List) und somit nicht mit normalen Pointern traversiert werden
- Aus diesem Grund stellen uns Containers *Iteratoren* zur Verfügung, welche uns diese Funktionalität ermöglichen.

Iteratoren

Der Datentyp von einem Iterator erhält man mit dem Format

container::iterator

Beispiele

Vektor mit Integer gefüllt

```
std::vector<int>::iterator
```

String

```
std::string::iterator
```

Vektor mit Floats gefüllt

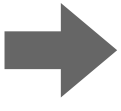
```
std::vector<float>::iterator
```

Set mit Booleans gefüllt

```
std::set<bool>::iterator
```

Iteratoren – Beispiel

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

int main(){
    std::vector<int> cont = {8,3,1,4,6,9};

    for (std::vector<int>::iterator it = cont.begin(); it != cont.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

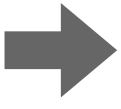


```
8 3 1 4 6 9
```

Iteratoren – Beispiel

Der selbe Code funktioniert genau gleich bei anderen Containertypen, wie z.B. bei einem set:

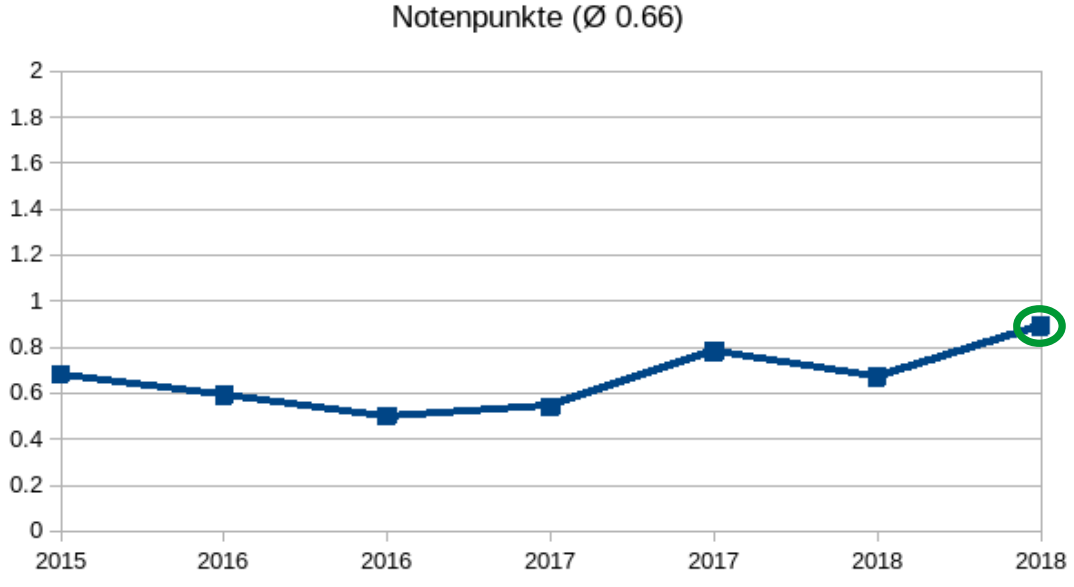
```cpp
#include <set>
#include <iostream>

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;
}
```



```
1 3 4 6 8 9
```

Prüfungsaufgabe

Typen und Werte (Konstrukte)



Sommer 2018, 0.89 Notenpunkte

Prüfungsaufgabe

Konstrukte

Basisprüfung Sommer 2018, 0.89 Notenpunkte

```
int a[6] = {1, 3, 5, 7, 9, 11};
int* x = a;
int& k = *x;
x++;
std::cout << *(x+k);
```

```
5
```

Prüfungsaufgabe

Konstrukte

Basisprüfung Sommer 2018, 0.89 Notenpunkte

```
float p[6] = {0.5, 1.0, 1.5, 2.0, 0.5, 0.0};
float* s = &p[4];
float* r = &p[2];
std::cout << (r[2] + *(s+1));
```

```
0.5
```

Prüfungsaufgabe

Konstrukte

Basisprüfung Sommer 2018, 0.89 Notenpunkte

```
void my_function(int* m) {
    *m = 2;
}

int values[] = {1, 3, 5};
int* v = values;
my_function(v++);
std::cout << v[0];
```

```
3
```

Prüfungsaufgabe

Konstrukte

Basisprüfung Sommer 2018, 0.89 Notenpunkte

```
struct Node {
    Node* next;
    int value;
    Node (int v, Node* n) : next(n), value(v) {};
};

Node s = Node(10, nullptr);
Node t = Node(20, &s);
s.next = &t;
std::cout << (s.next->next->value);
```

```
10
```

Prüfungstipps

Generell

- Macht euch einen Lernplan für die ganze Lernphase.
- Löst nochmals viele Serien
- Zu wenig lernen ist schlecht, zu viel aber genau so
  -> Gönnt euch Freizeit
- Wenn Ihr an einer Prüfung stecken bleibt, geht weiter und löst die Schwierigen aufgaben am Schluss Iterativ Stück für Stück

Prüfungstipps

Informatik I (♥)

- Es gibt viele Alte Prüfungen, löst so viele wie möglich.
- Der Prüfungsstil war bis jetzt gleich wie bei Prof. Friedrich
- Viele Aufgaben sind jedes Jahr gleich
- Programmiert in eurer Freizeit
- Wenn Ihr bei einem bestimmten Thema mühe habt, schaut euch das PVK Skript von Luzian Bieri an

Viel Spass!