PVK Präsentation

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Aufbau

- 9 Lektionen à 55'
- 3 Lektionen pro Tag
- Fragestunde 12.00 - ?

Lektion:

1. Theorie: 30'
2. Aufgaben Lösen: 15'
3. Aufgaben besprechen: 10'
4. Pause 5'

1. Datentypen, Literale, Operatoren, Präzedenz
2. Pointer, Referenzen, Dynamische Allokation
3. Arrays, vector, struct
4. Syntax, ASCII Code, string
5. Verzweigungen, Schleifen
6. Funktionen, Gültigkeitsbereich, Überladungen
7. Stellenwertsysteme, EBNF
8. Klassen, Dynamische Datenstrukturen
9. Vererbung

PVK Lektion 1

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 5' Datentypen
- 5' Literale
- 10' Operatoren
- 10' Präzedenz
- 15' Übungen
- 10' Besprechung

Namen

Erlaubte Namen

- Eine Sequenz von Ziffern, Unterstrichen, Klein- und Grossbuchstaben,
- die **nicht mit einer Ziffer** beginnt,
- die **mit zwei oder mehr Unterstrichen** beginnt,
- die **nicht mit einem Unterstrich gefolgt von einem Grossbuchstaben** beginnt,
- die nicht ein Schlüsselwort von c++ ist.

Primitive Datentypen

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void

L- und R-Werte

- Ein L-Wert ist ein veränderbarer Wert mit Speicheradresse.
- Ein R-Wert ist ein Ausdruck der kein L-Wert ist. Ein R-Wert kann nicht verändert werden. Jeder L-Wert kann als R-Wert verwendet werden.

Modifier	Effect
signed	variable interpreted as signed
unsigned	variable interpreted as unsigned
short	half number of allocated bits if possible
long	double number of allocated bits if possible

Primitive Datentypen

Modifier	Typical Bit Width	Typical Range
char	1byte	-127 to 127
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4byte	-2'147'483'648 to 2'147'483'647
unsigned int	4bytes	0 to 4'294'967'295
signed int	4bytes	-2'147'483'648 to 2'147'483'647
short int	2bytes	-32'768 to 32'767
unsigned short int	2bytes	0 to 65'535
signed short int	2bytes	-32'768 to 32'767
long int	4bytes	-2'147'483'648 to 2'147'483'647
signed long int	4bytes	-2'147'483'648 to 2'147'483'647
unsigned long int	4bytes	0 to 4'294'967'295
float	4bytes	+/- 3.4e +/- 38 (7 digits)
double	8bytes	+/- 1.7e +/- 308 (15 digits)
long double	8bytes	+/- 1.7e +/- 308 (15 digits)

Bool Literale

```
1 true     // 1
2 false    // 0
```

Typenumwandlung

- `int->bool` 0 wird zu `false`, alles andere zu `true`.
- `bool->int` true wird zu 1, false wird zu 0.

Integer Literale

```
1 212       // Decimal number
2 212ul     // Long unsigned decimal number
3 0xFeeL    // Long hexadecimal number
4 0b101     // Binary representation of 5
5 011       // Octal representation of 9
6 078       // Illegal: 8 is not an octal
    digit
```

Floating point Literale

```
1 3.14195      // Decimal repr. of pi
2 314195E-5L   // Exponential repr. of pi
3 510E         // NO: incomplete exponent
4 210f         // NO: no decimal or exponent
5 .e55         // NO: missing integer or
    fraction
```

Suffixes

- Floating point literals sind immer vom Typ double.
- Ausser mit dem Suffix `f,F` wird der Typ float.
- Oder mit dem Suffix `l,L` wird der Typ long double.

Implizite Typenumwandlung

Wann immer in einem Kontext Typ `T2` erwartet, aber `T1` vorhanden ist, macht c++ eine Typenumwandlung.

Fälle

- Als Funktionsargument.
- Bei Operatorauswertung (Zum "allgemeineren" Typ).
- Bei der Wertrückgabe in einer Funktion.
- In Anweisungen wie `switch -> int`.
- In Anweisungen wie `if, while, for -> bool`.

$$\text{char, bool} < \text{int} < \text{unsigned int} < \text{float} < \text{double}$$

Falls keine passende Umwandlung möglich ist: Programmabbruch.

Operatoren

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
::	.*	.	?:		

Arithmetische Operatoren

```
+   -   *   /   %   ++   --
```

Signatur

- Ein oder zwei Argumente.
- Bei zwei Argumenten unterschiedlichen Typs wird zum "allgemeineren Typ" konvertiert.
- Rückgabetyp entspricht "allgemeinstem" Eingabetyp.

$$\text{char, bool} < \text{int} < \text{unsigned int} < \text{float} < \text{double}$$

Arithmetische Operatoren

```
1  int      a = 2, b = 7;
2  int      d = a - b;          // binary minus
3  int      e = -a;             // unary minus
4  int      f = a++;            // f = 2; a = 3;
5  --a;                         // a = 2;
6  int      g = ++a;            // g = 3; a = 3;
7  int      h = b % 2;          // h = 1;
8  int      i = b/a;            // i = 3;
9  int      j = b/2.0;          // j = 3;
10 double   k = b/a;            // k = 3;
11 double   l = b/double(a)// l = 3.5;
```

$$a/b * b + a\%b == a$$

Vergleichsoperatoren

`==` `!=` `>` `<` `>=` `<=`

Signatur

- Zwei Argumente.
- Bei zwei Argumenten unterschiedlichen Typs wird zum "allgemeineren Typ" konvertiert.
- Rückgabetyp ist IMMER `bool`.

char, bool $<$ int $<$ unsigned int $<$ float $<$ double

Vergleichsoperatoren

```
1 0 = 1;          // NO! = is assignment
    operator
2 0 == 1;         // false 0 != 1
3 3.0 == 3;       // true
4 8>4>1;          // false
5 0>-1>0;         // true
6 8>4 && 4>1;     // true
```

Logische Operatoren

`&&` `||` `!`

Signatur

- Ein oder zwei Argumente.
- Argumente werden immer zu `bool` konvertiert.
- Rückgabetyp ist IMMER `bool`.
- `&&` und `||` machen **Kurzschlussauswertung.**

Logische Operatoren

x	y	AND(x,y)	OR(x,y)
0	0	false	false
0	1	false	true
1	0	false	true
1	1	true	true

x	NOT(x)
0	true
1	false

XOR(x,y)=AND(OR(x,y),NOT(AND(x,y)))

x	y	XOR(x,y)
0	0	false
0	1	true
1	0	true
1	1	false

```
1 !(a && b)==(!a || !b)
2 !(a || b)==(!a && !b)
```

Präzedenz

P.	Operator	Description	Associativity
1	`::`	Scope resolution	
2	`a++ a--`	Suffix/postfix increment and decrement	Left-to-right
	`type() type{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Locigal NOT and bitwise NOT	
	`*a`	Dereference	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator	
9	`< <= > >=`	Relational operators	
10	`== !=`	Relational operators	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR	
13	`|`	Bitwise OR	
14	`&&`	Logical AND	
15	`||`	Logical OR	
	`a?b:c`	Ternary conditional	

Präzedenz

```
1  cout<<a&&b;      //(cout<<a)&&b;
2
3  *p++             //*(p++);
4
5  a = b = c = d;   //a = (b =(c = d)));
6
7  a + b - c;       //(a + b) - c;
8
9  delete ++*p;     //delete(++(*p))
```

Präzedenz

Schlecht geschriebene Ausdrücke sind in der Auswertung undefiniert.

```
1 f(++i, ++i);
2 n = ++i + i;
3 b = ++a - a++;
```

Vermeide mehrfache Veränderung der gleichen Variable in einem Ausdruck.

PVK Lektion 2

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 10' Pointer
- 10' Referenzen
- 10' Pointer vs Referenzen
- 6' Dynamische Allokation
- 12' Übungen
- 7' Besprechung

Pointer

Pointer sind Variablen die Adressen anderer Variablen speichern.

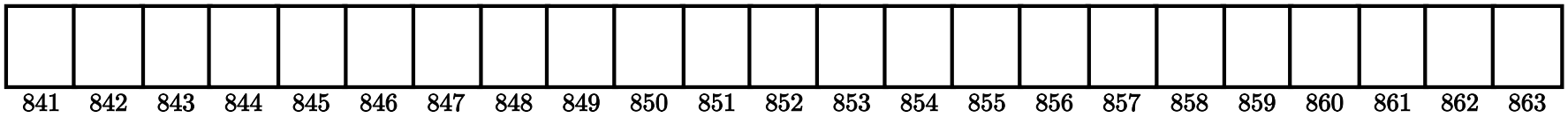
```
1 int a = 6;
2 int * b = &a;
3
4 (*b)++; //a == 7
```

Operatoren

- Dereferenzierung: `*`
- Referenzierung: `&`
- Neuer pointer: `<type> * name`
- Inkrementierung: `++`

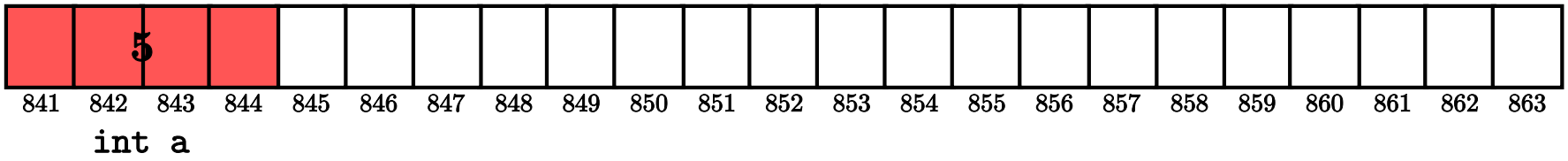
Pointer Program

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

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



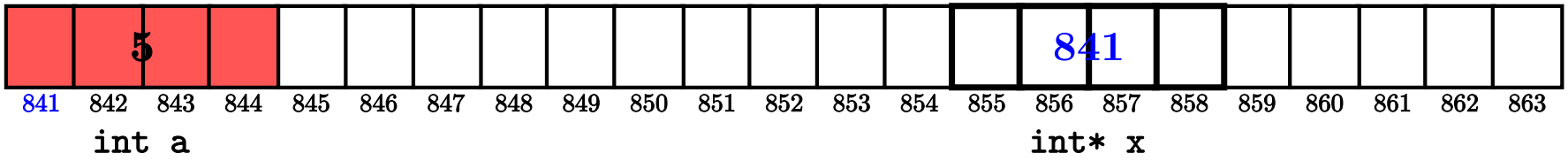
Pointer Program

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

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



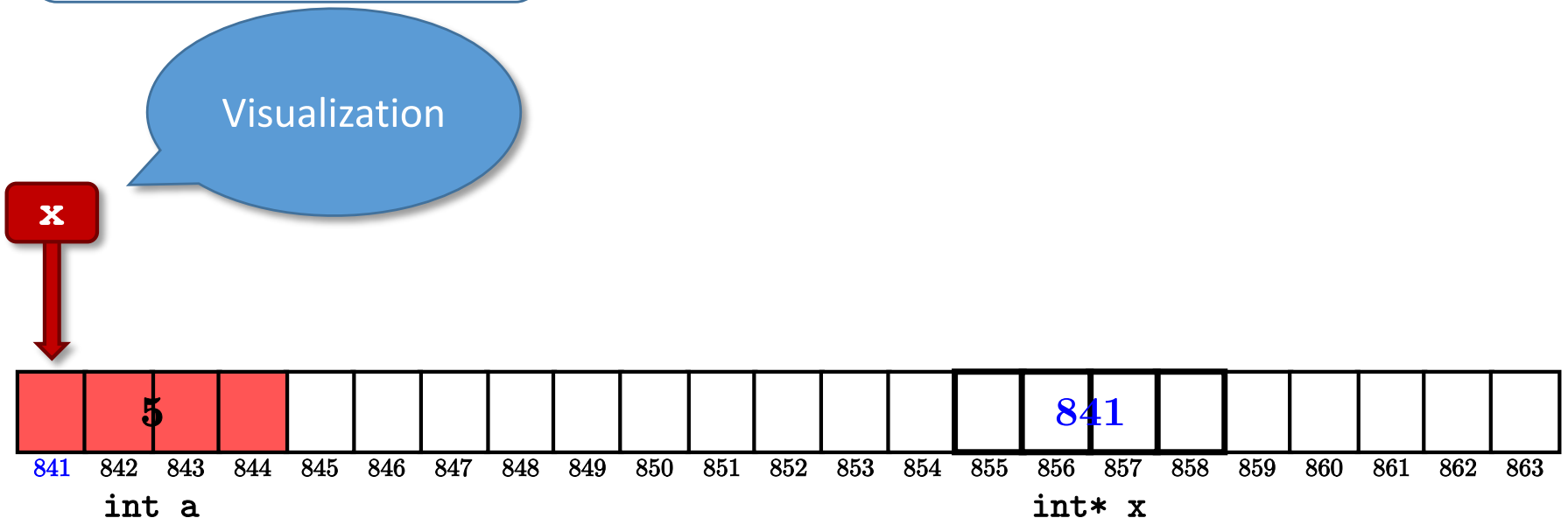
Pointer Program

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

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



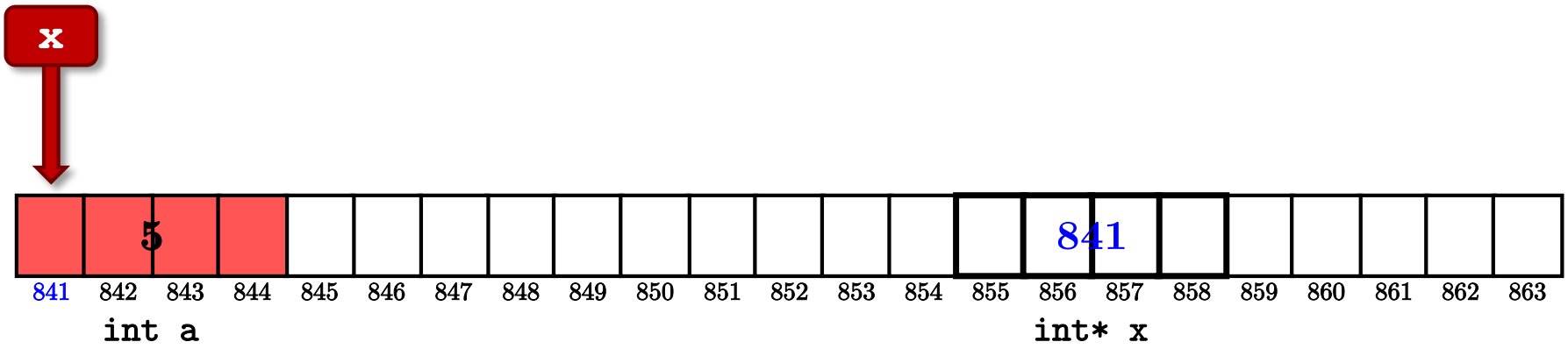
Pointer Program

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

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



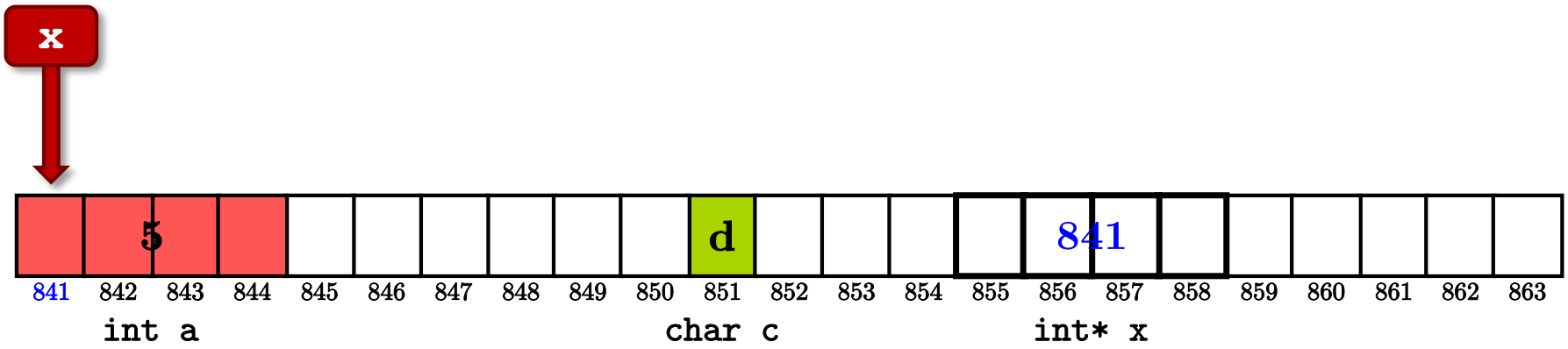
Pointer Program

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

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



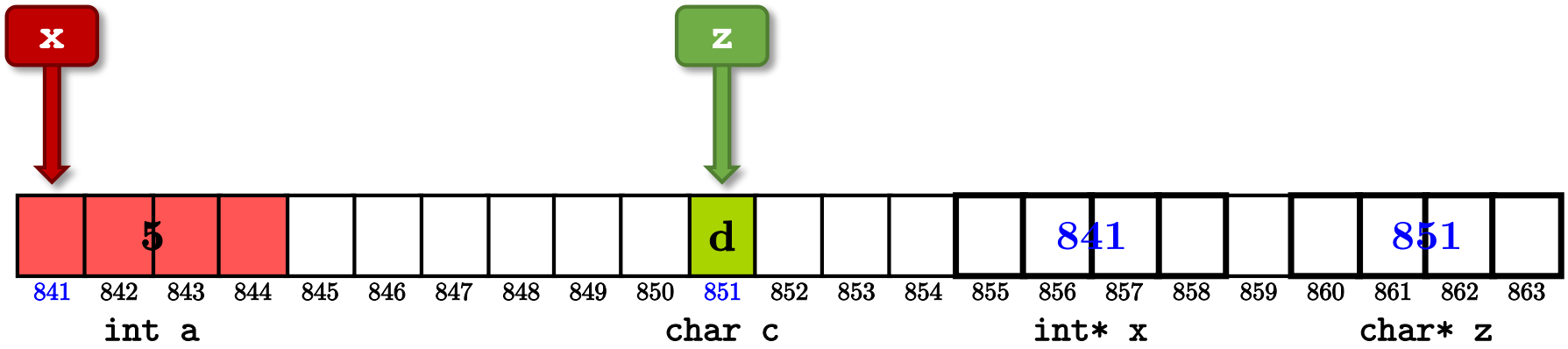
Pointer Program

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

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



Pointer Program

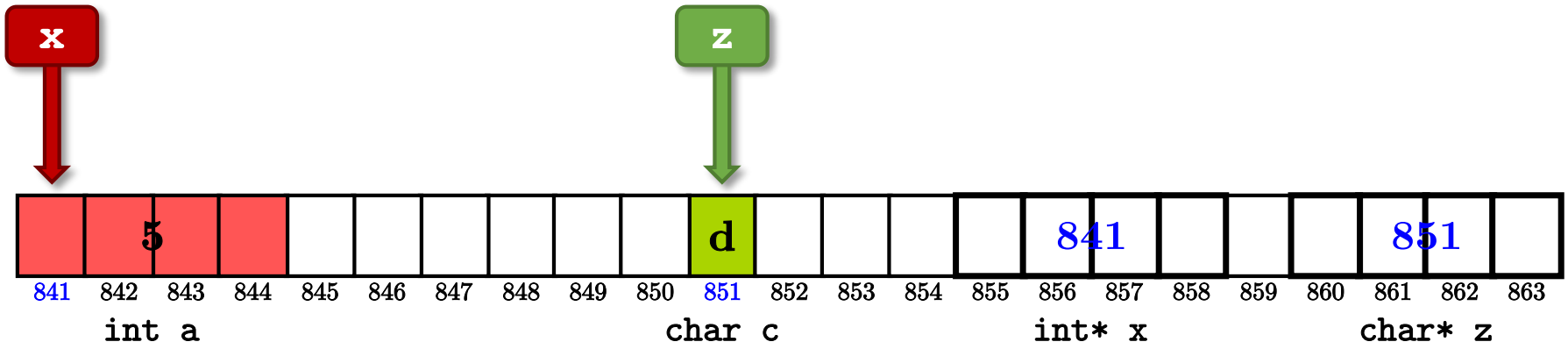
```
int a = 5;
int* x = &a;

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



Shifting Pointers

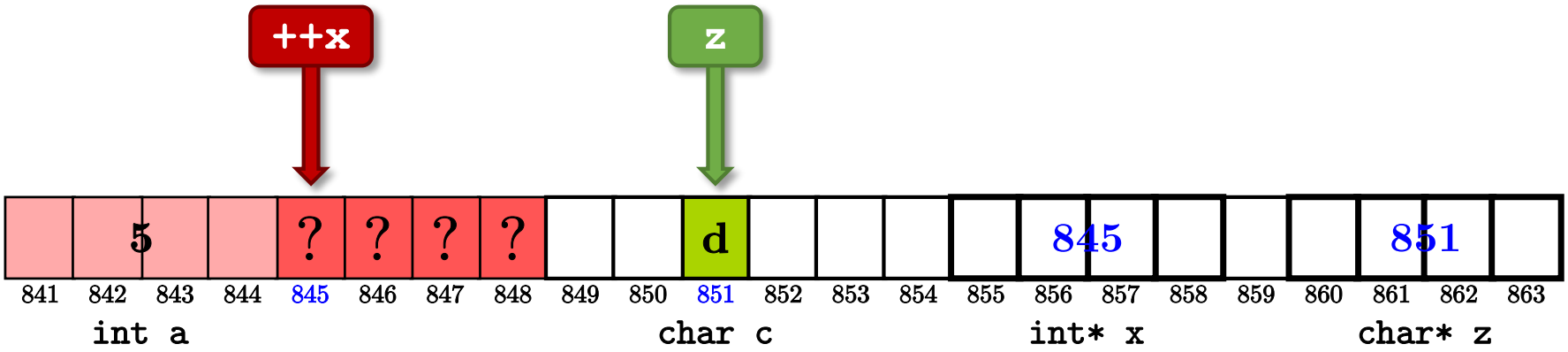
Pointer Program

```
++x;
++z;
```



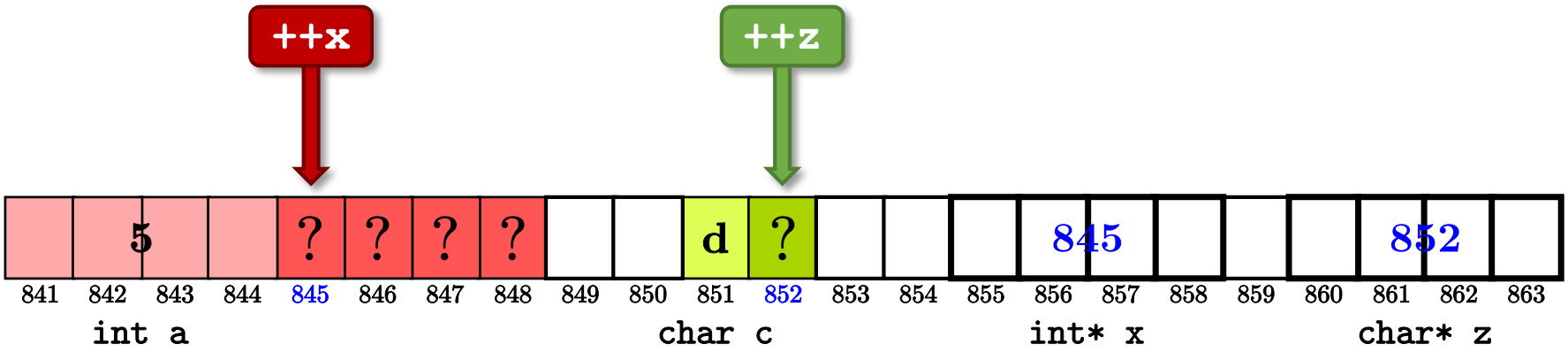
Pointer Program

```
++x;
++z;
```



Pointer Program

```
++x;
++z;
```

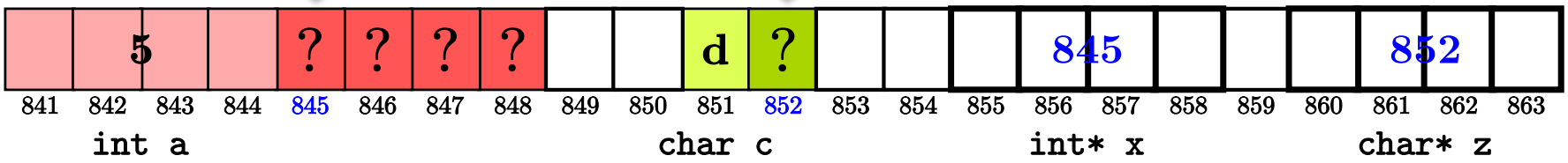


Pointer Program

```
++x;
++z;
```

Warning:
We don't know the value at

?



Const Pointers

Lies von rechts nach links! Lies * als 'Pointer auf'.

- `int * iptr = &i;`
  iptr: Pointer auf int.
- `const int* icptr = &i;`
  icptr: Pointer auf const int.
- `int const * ic2ptr = &i;`
  icptr2: Pointer auf const int.
- `int * const iptrc = &i;`
  iptrc: const Pointer auf int.
- `const int * const icptrc = &i;`
  icptrc: const Pointer auf const int.

Const Pointers

Regeln

- Ein const Pointer kann nach der Initialisierung nicht umgesetzt werden.
- Ein Pointer auf eine const Variable kann die Variable nicht verändern.
- Ein Pointer auf eine nicht const Variable kann nicht auf eine const Variable zeigen.

```
1 int i = 1;
2 int * const pointer_to_i = &i;
3 // ++pointer_to_i; //not allowed
4 int const * cpointer_to_i = &i;
5 // *cpointer_to_i = 3; //not allowed
6 const int j = 2;
7 // int * pointer_to_j = &j; //not allowed
8 const int * pointer_to_j = &j;
```

Referenzen

Referenzen sind neue Namen für bestehende Variablen.

```
1 int a = 6;
2 int & b = a;
3
4 b++; // a == 7
```

Operatoren

- Neue Referenz: `<type> & name`

Referenzen

Regeln

- Referenzen müssen zum Deklarationszeitpunkt initialisiert werden.
- Referenzen müssen mit einem L-Wert initialisiert werden.
- Das referenziert Objekt muss mindestens so lange wie die Referenz existieren.

```
1 int i = 1;
2 int & ref_to_i = i;
3 // int & ref_to_nothing; //not allowed
```

Const Referenzen

Regeln

- Const Referenzen dürfen auf R-Werte zeigen.
- Nur const Referenzen dürfen auf const Variablen zeigen.

```
1 //int & h = 3; //not allowed
2 const int & i = 7;
3
4 const int n = 5;
5 // int & i = n; //not allowed
6 const int & i = n;
```

Dynamische Allokation

Normalerweise wird Speicherplatz, anhand der Variablendefinitionen, vor der Programmausführung gemacht.

Something new

- Jedes `new` benötigt ein passendes `delete`.
- Wenn man dynamischen Speicherplatz nicht aufräumt kann es zur Überfüllung des Programmspeichers kommen.

new

```
1  /*
2    new <type> (<constructor args>);
3  */
4
5  //allocation of a single variable
6  int * pointer;
7  pointer = new int;
8
9  //allocation of a whole array
10 //length does not have to be defined
11 //at compiletime!
12 int * another_pointer;
13 another_pointer = new int [length];
```

delete

```
1 /*
2    delete <pointer>
3 */
4
5 //delete the objects allocated above
6 delete pointer;
7
8 //Use delete [] for dynamically allocated
     arrays
9 delete [] another_pointer;
```

PVK Lektion 3

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 10' Arrays
- 10' vector
- 10' struct
- 15' Übungen
- 10' Besprechung

Arrays

Arrays speichern eine Sequenz fixierter Länge von Objekten gleichen Typs.

```
1 //Arraylaenge bekannt zu Compilezeit!
2 const int Length = 10;
3 //Deklarierung
4 int my_array[Length];
5 //Zugriff
6 my_array[9] = 100;
```

Beachte

- Arrays enthalten Zufallswerte wenn uninitialisiert.
- Zugriff auf Werte ausserhalb ist möglich. Programmverhalten undefiniert.

Arrays: Initialisierung

```
1 int b[8] = {1,2,3,4}; //[1 2 3 4 0 0 0 0]
2 int c[4]; //[w x y z]
3 int a[] = {7,5,0,3,8}; //[7 5 0 3 8]
```

Nutze Schleifen um nacheinander auf alle Elemente zuzugreifen:

```
1 for(int i = 0; i<4; i++){
2   c[i] = 0;
3 }
```

Arrays: Mehrdimensionalität

```
1 //2 element array of 4 element arrays
2 int A[2][4];
3 int B[][3] = {{1,2,3},{4,5,6},{7,8,9}};
4 int C[][2] = {1,2,3,4,5,6,7,8};
5
6 cout<<C[3][1];
```

Arrays: Mehrdimensionalität

```
1 //2 element array of 4 element arrays
2 int A[2][4];
3 //3 element array of 3 element arrays
4 int B[][3] = {{1,2,3},{4,5,6},{7,8,9}};
5 //4 element array of 2 element arrays
6 int C[][2] = {1,2,3,4,5,6,7,8};
7
8 cout<<C[3][1]; //output: 8
```

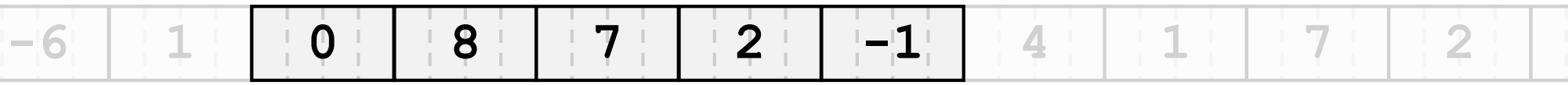
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                          // array-to-pointer conv
++ptr;                                 // shift to the right
int my_int = *ptr;                     // read target
ptr += 2;                              // shift by 2 elements
*ptr = 18;                             // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```

-6	1	3	-8	1	5	-3	4	1	7	2	

Pointer Program

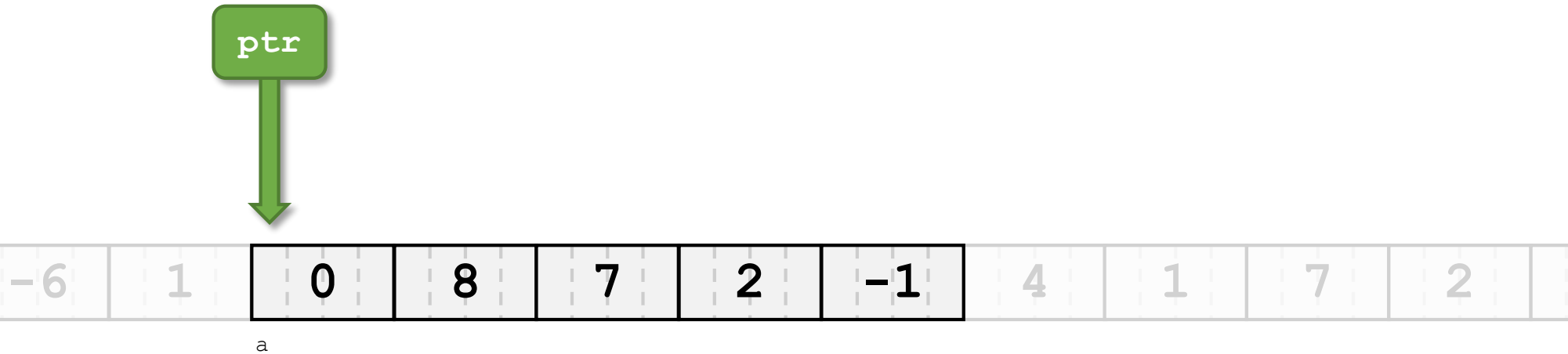
```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                              // array-to-pointer conv
++ptr;                                     // shift to the right
int my_int = *ptr;                         // read target
ptr += 2;                                  // shift by 2 elements
*ptr = 18;                                 // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



a

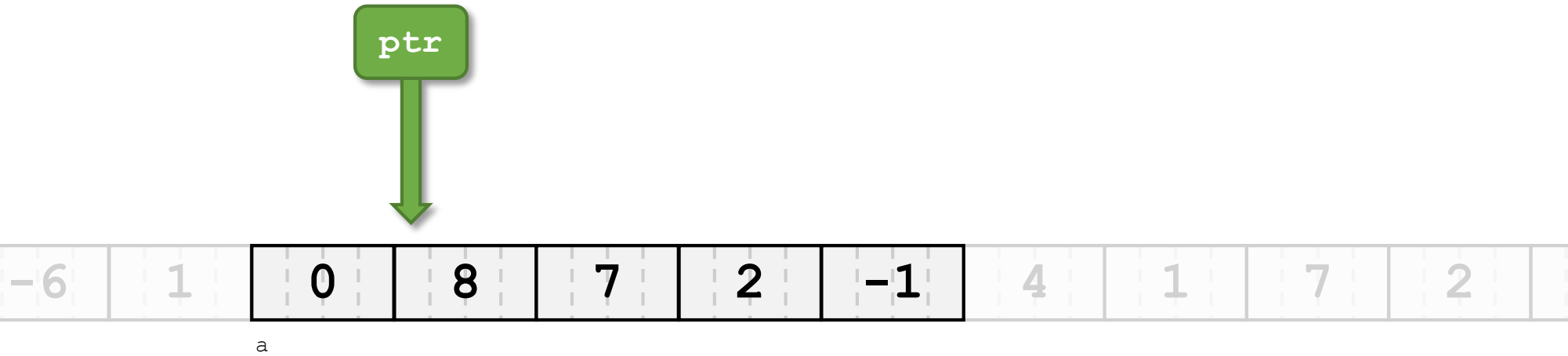
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                          // array-to-pointer conv
++ptr;                                 // shift to the right
int my_int = *ptr;                     // read target
ptr += 2;                              // shift by 2 elements
*ptr = 18;                             // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



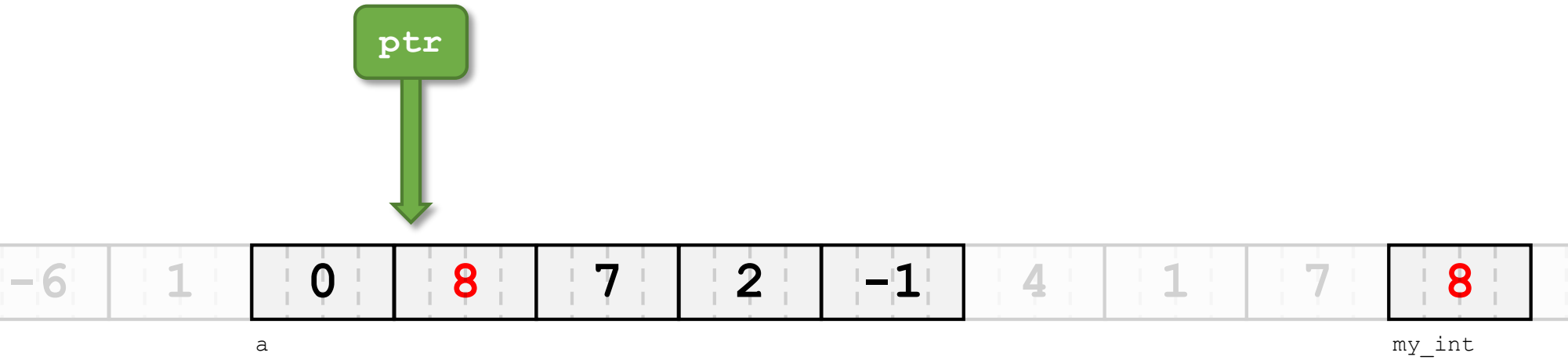
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                              // array-to-pointer conv
++ptr;                                     // shift to the right
int my_int = *ptr;                         // read target
ptr += 2;                                  // shift by 2 elements
*ptr = 18;                                 // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



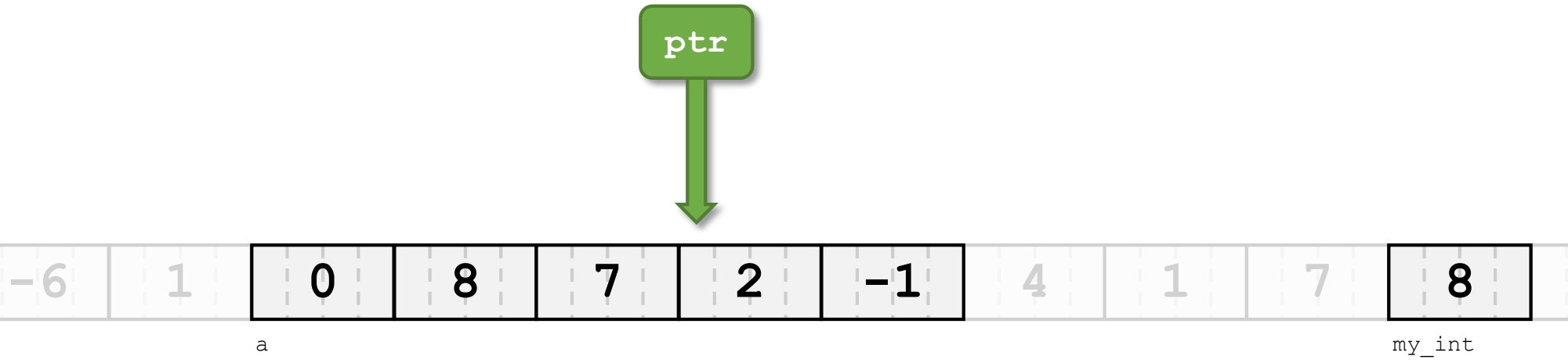
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                              // array-to-pointer conv
++ptr;                                     // shift to the right
int my_int = *ptr;                         // read target
ptr += 2;                                  // shift by 2 elements
*ptr = 18;                                 // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



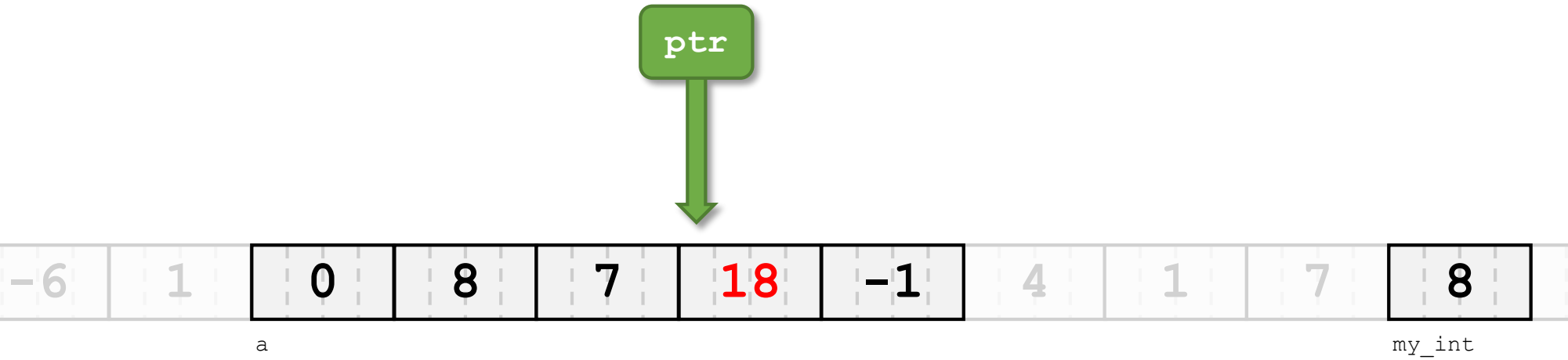
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                          // array-to-pointer conv
++ptr;                                 // shift to the right
int my_int = *ptr;                     // read target
ptr += 2;                              // shift by 2 elements
*ptr = 18;                             // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



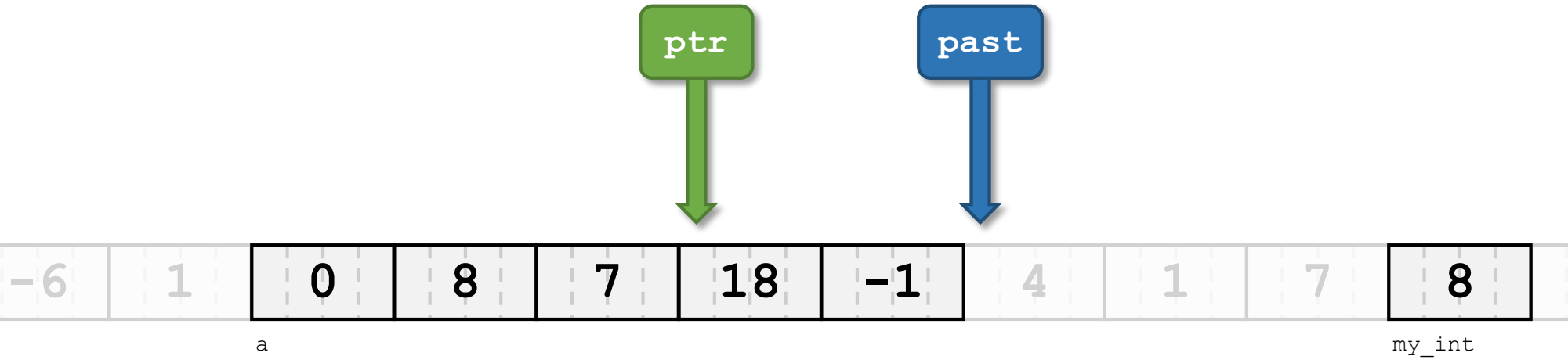
Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                          // array-to-pointer conv
++ptr;                                 // shift to the right
int my_int = *ptr;                     // read target
ptr += 2;                              // shift by 2 elements
*ptr = 18;                             // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



Pointer Program

```
int a[5] = {0, 8, 7, 2, -1};
int* ptr = a;                              // array-to-pointer conv
++ptr;                                     // shift to the right
int my_int = *ptr;                         // read target
ptr += 2;                                  // shift by 2 elements
*ptr = 18;                                 // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```



Vectors

```
1  #include <vector>
2
3  int main(){
4     int n;
5     cin>>n;
6     std::vector<int> numbers(n,0);
7
8     for (int i = 0; i<n; i++)
9        std::cin>>numbers[i];
10
11    for (int i = 0; i<n; i++)
12       std::cout<<numbers[i]<<" ";
13
14    std::vector<int> copy = numbers;
15
16 }
```

Vectors

```
1 cout<<numbers.size(); //Laenge des Vektors
2
3 numbers.push_back(7); //Verlaengerung des
    Vektors
4
5 cout<<numbers[11]; //garbage /
    segmentation fault
6
7 cout<<numbers.at(11); //Prueft Index auf
    Validitaet
```

vector

2D Vectors

```
1  int m = 4;
2  int n = 3;
3  std::vector< std::vector<int> > peter (m,
       std::vector<int>(n));
4
5  for (unsigned int i = 0; i<m; i++){
6    for(unsigned int j = 0; j<n; j++){
7      std::cin>>peter[i][j];
8    }
9  }
10
11 std::vector< std::vector<int> > copy =
       peter;
```

struct

```
1  struct rational{
2    int n;
3    int d;
4  };
5
6  int main (){
7    rational r;
8    r.n = 1;
9    r.d = 2;
10
11   return 0;
12 }
```

struct - Direkte Instantiierung

```
1  struct rational{
2    int n;
3    int d;
4  }r,s;
5
6  int main (){
7    r.n = 1;
8    r.d = 2;
9
10   return 0;
11 }
```

struct - Als Funktionsargument

```
1 //POST: deliver solution for quadratic
    equation and return number of solutions
2 int quad_solve(double a, double b, double
    c, double & x1, double & x2);
```

struct - Als Funktionsargument

```
1 struct solution{
2   double x1;
3   double x2;
4 };
5
6 //POST: return solution as struct
7 solution quad_solve(double a, double b,
    double c);
```

PVK Lektion 4

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 10' Syntax
- 10' ASCII Code
- 10' string
- 15' Übungen
- 10' Besprechung

Syntax: Definitionen

Algorithmus

Ein Rezept zur Lösung eines Problems.

Programmiersprache

Eine Sammlung von Instruktionen, die genutzt wird um Programme zu schreiben die Algorithmen implementieren.

Syntax

Eine Sammlung von Regeln die beschreibt welche Symbole einer Sprache wie kombiniert werden dürfen. Grammatikalische Korrektheit?

Semantik

Die Semantik beschreibt wie die Programmiersprache interpretiert wird. Was ist die Bedeutung einer Programmzeile?

Syntax: Definitionen

Editor

Ein spezieller Editor vereinfacht das Schreiben in einer Programmiersprache.

Compiler

Ein Compiler übersetzt von Programmiersprache zu Maschinencode, sodass der Code auf einer Maschine ausgeführt werden kann. (Assembler)

Computer

Ein Computer ist fähig Maschinencode auszuführen.

Syntax: Definitionen

Deklaration

Eine Deklaration führt einen neuen Namen ins Programm ein.

Definition

Eine Definition gibt einem namen einen Körper im Programm. (für Variablen und Funktionen)

Initialisierung

Eine Initialisierung gibt einem definierten Namen einen Wert. (nur für Variablen)

Syntax: Definitionen

Literal

Ein Literal repräsentiert einen konstanten Wert im Programm. Ein Literal hat Typ und Wert.

Variable

Eine Variable repräsentiert einen veränderlichen Wert im Programm. Eine Variable hat Typ, Wert, Name und Adresse.

Objekt

Ein Objekt repräsentiert Werte im Speicher. Objekte haben Typ, Wert und Adresse. Beispiele: Variable, Datenstruktur, Funktion.

Syntax: Definitionen

Lvalue

Ein L-Wert ist ein veränderlicher Ausdruck der eine Adresse hat.

Rvalue

Ein R-Wert ist ein Ausdruck, der kein L-Wert ist. Ein R-Wert kann nicht verändert werden. Jeder L-Wert kann zum R-Wert umgewandelt werden.

Block

Ein Block ist ein von geschweiften Klammern umfasster Programmteil.

Syntax: Programmaufbau

```
1  #include <iostream>
2  //#include "local_header_file.h"
3
4  //Function declarations (and definitions)
5
6  int main(int argc, char * argv [])
7  {
8      //Function calls
9
10     std::cout << "Hello World!" << std::
           endl;
11     return 0;
12 }
13
14 //Function definitions
```

Schriftzeichen

char

- 1 byte - 7 bits verfügbar (Spezialrolle 1. bit)
- Speichert Symbole

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

ASCII code

Schriftzeichen

```
1 char letter 'a';
2 int number = letter; // conversion
3 //number = 97
4
5 int number = 66;
6 //implicite conversion
7 char letter = number;
8 //letter = 'B'
```

Schriftzeichen

Das Alphabet

$$
\begin{aligned}
(65)- \quad & 10\ 00001 = && 'A' \\
(66)- \quad & 10\ 00010 = && 'B' \\
& \vdots \\
(97)- \quad & 11\ 00001 = && 'a' \\
(98)- \quad & 11\ 00010 = && 'b'
\end{aligned}
$$

string: C-style Character String

```
1 //initialisation with string literal
2 char text = "bool";
3 //equivalent to:
4 char text = {'b','o','o','l','\0'};
```

Weaknesses

- Konstante Länge (ist ein Array)
- Ist für die meisten Operatoren nicht überladen.
- Kennt die eigene Länge nicht.

string

```
1  #include <string>
2  std::string text1 = "bool";
3  //A string knows its length
4  text1.length();
5
6  //initialize with variable length n
7  //and fill with 'a'
8  std:string text2 (n, 'a');
9  //string understands comparisons
10 //and many other operations
11 text1 == text2;
```

More information on strings

PVK Lektion 5

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 15' Verzweigungen
- 15' Schleifen
- 15' Übungen
- 10' Besprechung

Verzweigungen: if ... else

```
1 if (condition)
2   statement1;
3 else{
4   statement2;
5   statement3;
6 }
```

Verzweigungen: switch

```
1  switch (var) {
2    case 1: cout<<"one"<<endl;
3    anotherStatement;
4    break;
5
6    case 2: cout<<"two"<<endl;
7    case 3: cout<<"three"<<endl;
8    break;
9
10   case 4: cout<<"four"<<endl;
11   break;
12
13   default: cout<<"default"<<endl;
14 }
```

Schleifen: Welche Schleife in welchen Fall?

Motivation

- So wenig code wie möglich.
- Einfach lesbarer code.

Möglichkeiten

- **for**: Es wird ein Zähler benötigt, Zähler wird nach der Schleife nicht mehr benötigt.
  Wiederhole ein statement n mal.
- **while**: Die Bedingung hängt von einer Variable ab, die bereits vor der Schleife existiert.
  Dekrementiere x bis es ein Vielfaches von 5 ist.
- **do**: Die Bedingung hängt von einer Variable ab, die erst in der Schleife erhalten wird.
  Führe cin >> x aus bis x > 3

Schleifen: while

```
1 while(condition)
2   statement;
3
4 while(condition){
5   statement1;
6   statement2;
7 }
```

Wichtig

- Bedingung, Veränderung, Terminierung!
- Eine leere Bedingung wird nicht kompiliert!

Schleifen: for

```
1 for(int i = 0; i<10; i = i+2){
2   statement1;
3   statement2;
4 }
5
6
7 for(;not_done;){
8   statement1;
9 }
```

Wichtig

- Der Ausdruck (i = i+2) wird erst nach den statements ausgeführt!
- Eine leere Bedingung entspricht true!

Schleifen: do while

```
1 do{
2    statement1;
3    statement2;
4 }
5 while(condition);
```

PVK Lektion 6

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 15' Funktionen
- 5' Gültigkeitsbereich
- 10' Überladungen
- 15' Übungen
- 10' Besprechung

Funktionen: Struktur

```
1 <r_type> <function_name> (<ptype> <pname>)
2 {
3   // Koerper / Funktionsdefinition
4 }
```

Struktur

- Rückgabetyp
- Funktionsname
- Funktionsparameter / Funktionsargumente
- Funktionskörper

Funktionsdefinition und -deklaration

```
1  void g (...); //Deklaration von g
2
3  void f (...)
4  {
5     g(...);
6  }
7
8  void g (...) // Definition von g
9  {
10    f(...);
11 }
```

Funktionen: Argumentübergabe

```
1 void change(int a){
2   a = 4;
3 }
4
5 void change_ref(int & a){
6   a = 4;
7 }
```

```
1 int main(){
2   int b = 3;
3   change(b); //kein effekt
4   change_ref(b); //b = 4;
5 }
```

Funktionen: Call by Reference

```
1 void change_ref(int & a){
2   a = 4;
3 }
4
5 void change_poi(int * a){
6   *a = 4;
7 }
```

```
1 int main(){
2   int b = 3;
3   change_poi(&b); //b = 4;
4 }
```

Funktionen: Return by Reference

```
1 int & increment ( int & i){
2   i = i + 1;
3   return i;
4 }
```

```
1 int main(){
2   int i = 3;
3   increment(increment(i));
4   ++(++i));
5 }
```

Funktionen: Rekursion

```
1 void f(){
2   f(); //Endlosaufruf
3 }
```

Funktionen: Rekursion

```
1 // POST: return value n!
2 unsigned int fac (unsigned int n)
3 {
4   //Abbruchbedingung
5   if (n <= 1) return 1;
6
7   //Rekursiver Aufruf
8   return n* fac(n-1);
9 }
```

Gültigkeitsbereich

Bereiche

1. In einer Funktion oder in einem Block (lokale Variablen)
2. Als Parameter einer Funktion (formaler Parameter)
3. Ausserhalb aller Funktionen (globale Variablen)

Regel

Es wird immer die nähere Definition genutzt!

Gültigkeitsbereich

```
1  int i = 2; //Globale Variable
2
3  void fun(){cout<<i;}
4
5  int main(){
6    int i = 5;
7    { //Block
8      int i = 3;
9      cout<<i;
10   }
11   cout<<i;
12   fun();
13 }
```

Regel

Es wird immer die nähere Definition genutzt!

Überladungen: Funktionen

```
1  void print_variable(int a){
2    cout<<"This is an int."<<endl;
3  }
4
5  void print_variable(double a){
6    cout<<"This is a double."<<endl;
7  }
8
9  int print_variable(int a, int b){
10   cout<<"Two ints."<<endl;
11   return 2;
12 }
```

Überladungen: Operatoren

Motivation

- Operatoren sind nur für primitive Datentypen definiert.
- Operatoren sind einfacher zu benutzen als Funktionen.
- **Neudefinition für jeden neuen Datentyp!**

```
1 <r_type> operator<target_op> (<ptype> <
    pname>)
2 {
3   //Koerper / Funktionsdefinition
4 }
```

Überladungen: Operatoren

```
1 class rational {
2   int n,d;
3   public:
4   rational& operator+= (const rational b);
5 };
```

$$\frac{a_n}{a_d} \leftarrow \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d}{a_d \cdot b_d} + \frac{b_n \cdot a_d}{b_d \cdot a_d}$$

Überladungen: Operatoren

```
1  rational& rational::operator+=(const
       rational b){
2    this->n = this->n*b.d + this->d*b.n;
3    this->d = this->d*b.d;
4    return *this;
5  }
6
7  rational operator+ (const rational& a,
       const rational& b){
8    rational temp = {0,1};
9    temp += a;
10   temp += b;
11   return temp;
12 }
```

PVK Lektion 7

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 20' Stellenwertsysteme
- 10' EBNF
- 15' Übungen
- 10' Besprechung

Stellenwertsysteme: Binäre Darstellung

$$91310 =$$

Stellenwertsysteme: Binäre Darstellung

$$91310 = 10 * 9131 + \qquad 0$$

Stellenwertsysteme: Binäre Darstellung

$$
\begin{aligned}
91310 &= 10 * 9131 + & 0 \\
9131 &= 10 * 913 + & 1
\end{aligned}
$$

Stellenwertsysteme: Binäre Darstellung

$$
\begin{aligned}
91310 &= 10 * 9131 + & 0 \\
9131 &= 10 * 913 + & 1 \\
913 &= 10 * 91 + & 3 \\
91 &= 10 * 9 + & 1 \\
9 &= 10 * 0 + & 9
\end{aligned}
$$

Stellenwertsysteme: Binäre Darstellung

$$
\begin{aligned}
91310 &= 10 * 9131 + & 0 \\
9131 &= 10 * 913 + & 1 \\
913 &= 10 * 91 + & 3 \\
91 &= 10 * 9 + & 1 \\
9 &= 10 * 0 + & 9
\end{aligned}
$$

$$
\begin{aligned}
61 &= 2 * 30 + & 1 \\
30 &= 2 * 15 + & 0 \\
15 &= 2 * 7 + & 1 \\
7 &= 2 * 3 + & 1 \\
3 &= 2 * 1 + & 1 \\
1 &= 2 * 0 + & 1
\end{aligned}
$$

Stellenwertsysteme: Binäre Darstellung

						$\sum$
Ziffern	9		1		3	1	0	
Multiplikator								
Wert								

Stellenwertsysteme: Binäre Darstellung

						$\sum$
Ziffern	9		1	3	1	0	
Multiplikator	10000		1000	100	10	1	
Wert							

Stellenwertsysteme: Binäre Darstellung

						$\sum$
Ziffern	9		1	3	1	0	
Multiplikator	10000		1000	100	10	1	
Wert	90000		1000	300	10	0	91310

Stellenwertsysteme: Binäre Darstellung

						$\sum$
Ziffern	9		1		3	1	0	
Multiplikator	10000		1000		100	10	1	
Wert	90000		1000		300	10	0	91310

							$\sum$
Ziffern	1	1	1	1	0	1	
Multiplikator	32	16	8	4	2	1	
Wert	32	16	8	4	0	1	61

Stellenwertsysteme: Negative Binärzahlen

bin	uint	int	bin	uint	int
0000	0	0	1000	8	-8
0001	1	1	1001	9	-7
0010	2	2	1010	10	-6
0011	3	3	1011	11	-5
0100	4	4	1100	12	-4
0101	5	5	1101	13	-3
0110	6	6	1110	14	-2
0111	7	7	1111	15	-1

Stellenwertsysteme: Dezimalzahlen im Binärsystem

									$\sum$
digit:	1	1	0	1	.	1	0	1	
factor:	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	
value	8	4	0	1		$\frac{1}{2}$	0	$\frac{1}{8}$	13.625

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.8

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.8
1.6	1	0.6

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.8
1.6	1	0.6
	⋮	

Stellenwertsysteme: Dezimalzahlen im Binärsystem

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.8
1.6	1	0.6
	⋮	

$$1.1\overline{1100}$$

Stellenwertsysteme: Unser kleines 10bit Fliesskommasystem

Beschreibung von Fliesskommasystemen

$$\mathcal{F}(\underbrace{\beta}_{\text{Basis} \geq 2}, \overbrace{p}^{\text{Anzahl Stellen} \geq 1}, \underbrace{e_{min}, e_{max}}_{\text{Kleinster und Grösster Exponent}})$$

$\mathcal{F}(2, 6, 0, 15)$

0000000000

- Vorkommastelle
- Nachkommastellen
- Exponent

Beispiele

$\underbrace{1.11111 \cdot 2^{15}}_{\text{Grösste Zahl}}$ $\underbrace{0.00001 \cdot 2^{0}}_{\text{Kleinste Zahl}}$

$\mathcal{F}(2, 6, -8, 7)$

0000000000

- Vorkommastelle
- Nachkommastellen
- Exponent

Beispiele

$\underbrace{1.11111 \cdot 2^{7}}_{\text{Grösste Zahl}}$ $\underbrace{0.00001 \cdot 2^{-8}}_{\text{Kleinste Zahl}}$

Stellenwertsysteme: $\mathcal{F}^*(2, 6, -8, 7)$

$$0\,0\,0\,0\,0\,0\,0\,0\,0\,0$$

- Vorzeichenstelle
- Nachkommastellen
- Exponent

Beispiele

$\underbrace{+1.11111 \cdot 2^{7}}_{\text{Grösste Zahl}}$ $\underbrace{-1.11111 \cdot 2^{7}}_{\text{Kleinste Zahl}}$ $\underbrace{+1.00000 \cdot 2^{-8}}_{\text{Kleinste positive Zahl}}$

$\mathcal{F}^*(2, 6, -7, 7)$

0000000000

- Vorzeichenstelle
- Nachkommastellen
- Exponent

Beispiele

$\underbrace{0000000000}$	$\underbrace{0000000001}$	$\underbrace{0000000010}$	$\underbrace{0000000011}$
Null	$+\infty$	$-\infty$	NaN

Tipps zu Fliesskommazahlen

```
1  //Kein Vergleich gerundeter Zahlen
2  double  a = 1.1;
3  if(100*a == 110) cout<<true<<endl;
4
5  //Keine Add. versch. grosser Zahlen
6  float a = 67108864.0f + 1.0f
7  //output: 67108864
8
9  //Keine Subtr. aehnlich grosser Zahlen
10 float x_0 = 0.2;
11 //represented as: 0.20000000298
12 float x_1 = 6*x_0 - 1; //is not 0.2
```

Backus-Naur-Form

Kurz und simpel

Die Backus-Naur-Form ist eine Sprache die mit einfacher Syntax beschreibt, welche Sätze mit den Wörtern einer Sprache gebildet werden dürfen.

Aufbau

- Alphabet = Terminalsymbole
- Satzbau = Produktionsregeln = Nichtterminalsymbol

```
1 ZifferAusserNull = "1" | "2" | "3" | "4" |
      "5" | "6" | "7" | "8" | "9" ;
2 Ziffer = "0" | ZifferAusserNull ;
```

Erweiterte Backus-Naur-Form: Beispiel

```
1 ZifAussNull = "1" | "2" | "3" | "4" | "5"
    | "6" | "7" | "8" | "9" ;
2 Zif = "0" | ZifAussNull ;
3
4 Zwoelf = "1", "2" ;
5 Dreihundertzwoelf = "3", Zwoelf ;
6
7 NatZahl = ZifAussNull, { Zif } ;
8 GanzeZahl = "0" | [ "-" ], NatZahl ;
```

Vorteile der EBNF

```
1  seq = term | term "_" seq
2  term = "A" | "A" lowerterm | lowerterm
3  lowerterm = "a" | "a" lowerterm
4
5
6  seq = term | term "_" seq
7  term = "A" { "a" } | "a" { "a" }
8
9  seq = term [ "_" seq ]
10 term = "A" { "a" } | "a" { "a" }
```

PVK Lektion 8

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 15' Klassen
- 15' Dynamische Datenstrukturen
- 15' Übungen
- 10' Besprechung

Klassen: Motivation

Was

- Klassen: Kombination von Variablen und Funktionen
- Instanzen
- Objekte

Objektorientiertes Programmieren

Die Grundidee besteht darin, die Architektur einer Software an den Grundstrukturen desjenigen Bereichs der Wirklichkeit auszurichten, der die gegebene Anwendung betrifft.

Wieso

- Verkapselung
- Wiederverwendbarkeit

Objektorientiertes Programmieren

Klassen: Grundlagen

Mitglieder

- **Mitgliedsfunktion**:
  Funktion die in der Klasse angelegt wird.
- **Mitgliedsvariable**:
  Variable die in der Klasse angelegt wird.

Mitglieder können nur über eine Instanz der Klasse aufgerufen werden.

```
1 //Definition der Klasse
2 class Vector {...};
3 Vector v1; //Deklaration einer Instanz
4 v1.memberVariable;
5 v1.memberFunction();
6 Vector* pv1 = &v1;
7 pv1->memberVariable;
```

Klassen: Grundlagen

Verkapselung

- **Private Mitglieder**:
  Nach dem Schlüsselwort `private` kommen alle Variablen die versteckt sein sollen.
- **Öffentliche Mitglieder**:
  Nach dem Schlüsselwort `public` kommen alle Variablen die öffenlich sein sollen.

Nur öffenlichte Mitglieder können über einen der beiden Zugriffsoperatoren erreicht werden.

Mitgliedsfunktionen haben Zugriff auf private Migliedsvariablen.

Klassen: Grundlagen

```
1 class Vector {
2 private:
3   double x;
4   double y;
5 };
```

Codeboard

Zusätzliches Beispiel

Klassen: Konstruktoren

Konstruktor

- Öffentliche Mitgliedsfunktion
- Wird automatisch beim Erstellen einer Instanz der Klasse aufgerufen!
- Wird eingesetzt um Mitgliedsvariablen zu initialisieren.

```
1 public:
2   Vector (){
3     x = 0;
4     y = 0;
5   }
6   Vector (double _x, double _y) : x(_x), y
      (_y) {}
```

Klassen: Konstruktoren

```
1  class Vector {
2    double x;
3    double y;
4  public:
5    Vector () : x(0),y(0){}
6    Vector (double _x, double _y)
7       : x(_x),y(_y) {}
8  };
9
10 Vector v1;
11 Vector v2();
12 Vector v3(1.0, 2.3);
```

Klassen: Zugriffsmethoden

Zugriff

- Der Zugriff ist durch Verkapselung eingeschränkt.
- Lösung: Sicherer Zugriff mit Zugriffsmethoden.

```
1 double get_x() const {return x;}
2 double get_y() const {return y;}
3
4 void set_x(const double _x) {x = _x;}
5 void set_y(const double _y) {y = _y;}
```

Klassen: Zugriff auf Mitglieder

this

- Der `this` Pointer speichert die Adresse seiner Instanz einer Klasse.
- Er ist in jeder Klasse vorhanden und kann in Mitgliedsfunktionen benutzt werden um auf die aktuelle Instanz zuzugreifen.

```
1 double get_x() const {return this->x;}
2 double get_y() const {return (*this).y;}
```

Klassen: Arithmetische Operatoren

Argumentübergabe

- Die Operatoren, die als Mitgliedsfunktion überladen werden, erhalten die aufrufende Instanz als erstes Argument.
- Der Rückgabetyp hängt vom überladenen Operator ab. Für Zuweisungen wird eine Referenz zurückgegeben.

```
1 Vector& operator+= (const Vector& b){
2     x += b.get_x();
3     y += b.get_y();
4     return *this;
5 }
6 //Im main:
7 Vector v3(3,4),v4(1,2);
8 v3 += v4;
```

Klassen: Arithmetische Operatoren

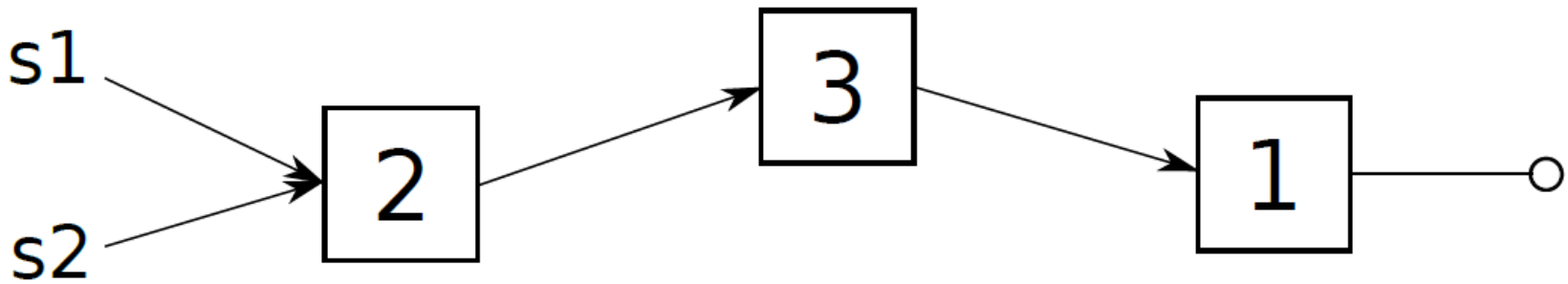
```
1  Vector& operator+= (const Vector& b){
2      x += b.get_x();
3      y += b.get_y();
4      return *this;
5  }
6
7  //Ausserhalb der Klasse
8  Vector operator+ (const Vector& a, const
       Vector& b) {
9    Vector res = a;
10   res += b;
11   return res;
12 }
```

Dynamische Datentypen - Stack

```
1  class stack {
2  public:
3    void push (int value){...}
4    int pop (){}
5    ...
6    void print (){...}
7
8  private:
9    ln* top_node; //ln = list node
10 };
11
12 struct ln {
13   int key;
14   ln * next;
15 }
```

Dynamische Datentypen - Stack

```
1 stack s1;
2 s1.push(1);
3 s1.push(2);
4 s1.push(3);
5 s2(s1);
```



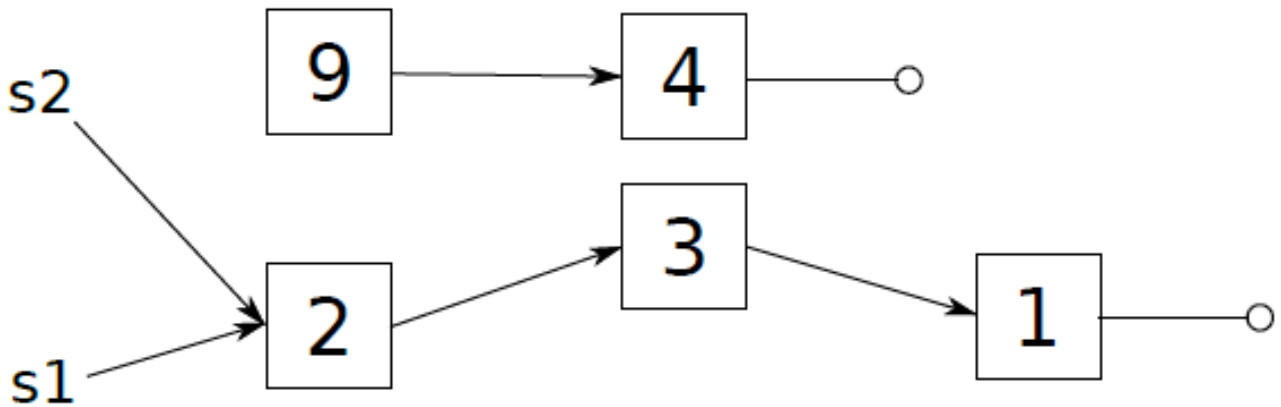
Dynamische Datentypen - Stack - Tiefe Kopie

```
1  stack::stack(const stack& s) : top_node(0)
       {
2    copy(s.top_node, top_node);
3  }
4
5  void stack::copy(const ln* from, ln*& to){
6    assert (to == 0);
7    if(from != 0){
8      to = new ln(from->key);
9      copy(from->next, to->next);
10   }
11 }
```

```
1  s2(s1);
```

Dynamische Datentypen - Stack - Zuweisung

```
1 stack s2;
2 s2.push(4);
3 s2.push(9);
4 s2 = s1;
```



Dynamische Datentypen - Stack - Zuweisung

```
1 void stack::clear(ln* from){
2   if (from != 0){
3     clear(from->next);
4     delete from;
5   }
6 }
```

Dynamische Datenstrukturen - Stack - Zuweisung

```
1 stack& stack::operator= (const stack& s) {
2   if (top_node != s.top_node) { // test
        for self-assignment
3     clear(top_node);
4     top_node = 0; // fix dangling pointer
5     copy(s.top_node, top_node);
6   }
7 return *this;
8 }
```

```
1 s1 = s2;
```

Dynamische Datenstrukturen - Stack - Destruktor

```
1 void useStack(){
2   stack temp;
3   temp.push(2);
4   temp.pop();
5 } //end of scope, destruction
6
7 stack::~stack() {
8   clear(top_node);
9 }
```

Klassen: Spezielle Mitgliedsfunktionen

Standardmitglieder

- Defaultkonstruktor
- Kopierkonstruktor
- Zuweisungsoperator
- Defaultdestruktor

Regel der drei: Wenn entweder Destruktor, Kopierkonstruktor und Zuweisungsoperator neu definiert werden sollten alle drei neu definiert werden.

PVK Lektion 9

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Time Schedule

- 30' Vererbung
- 15' Übungen
- 10' Besprechung

Vererbung: Grundlagen

```
1  class A{
2     ... //Basisklasse
3  };
4
5  class B: public A{
6     ... //Abgeleitete Klasse
7  };
8
9  class C: public B{
10    ...
11 };
```

Vererbung: Zugriffskontrolle

Vererbung \ Mitglied	`public`	`protected`	`private`
`public`	`public`	`protected`	n/a
`protected`	`protected`	`protected`	n/a
`private`	`private`	`private`	n/a

Polymorphismus

```
1  class A {
2    virtual void print()
3    {cout<<"A"<<endl;}
4  };
5
6  class B : public A {
7    void print()
8    {cout<<"B"<<endl;}
9  };
10
11 A instantce1;
12 instance1.print();
13 B instance2;
14 instance2.print();
15 A * pointer1 = &instance2;
16 pointer1->print();
```

Polymorphismus und dynamische Bindung

Ausgangspunkt: Abgeleitete Klasse die virtuelle Mitgliedsfunktion ihrere Basisklasse überschreibt.
Frage: Wann wird welche Version der Mitgliedsfunktion aufgerufen?

Faustregeln

1. Bei direktem Aufruf über eine Instanz wird immer die dem Typ entsprechende Mitgliedsfunktion aufgerufen.
2. Bei indirektem Aufruf über einen Pointer wird für nicht virtuelle Mitgliedsfunktionen immer die dem Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.
3. Bei indirektem Aufruf über einen Pointer wird für virtuelle Mitgliedsfunktionen immer die dem dereferenzierten Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.

Abstrakte Basisklasse

```
1  class A {
2    virtual void print() = 0;
3  };
4
5  class B : public A {
6    void print()
7    {cout<<"B"<<endl;}
8  };
9
10 //A instance1; //Forbidden
11 A * pointer1 = new B; //Allowed
```

Konstruktoren

```
1  class A {
2    int a;
3  public:
4    A(int _a) : a(_a){}
5  };
6
7  class B : public A {
8    int b;
9  public:
10   B(int _a, int _b) : A(_a), b(_b) {}
11 };
```

is a - has a

```
1  class point {
2  public:
3    double x;
4    double y;
5  };
6
7  class circle : private point {
8  private:
9    double radius;
10 };
```

is a - has a

```
1  class point {
2  public:
3    double x;
4    double y;
5  };
6
7  class circle {
8  private:
9    double radius;
10   point p;
11 };
```

is a - has a

```
1  class University {
2  private:
3    std::vector<Student> students_;
4  };
5
6  class Student {
7  private:
8    Legi legi_;
9  };
10
11 class Phys_Student : public Student {};
12
13 class Legi {
14   int immatriculation_year_;
15 };
```

Backup Slides

GianAndrea Müller
mailto:muellegi@student.ethz

July 5, 2018

Überladungen: Operatoren

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

Gewisse operatoren können nicht überladen werden:

::	.*	.	?:

Back

Überladungen: Operatoren

Wie überladen?

1. **Operatorsignatur festlegen**
   - Welcher Rückgabetyp?
   - Wie viele Argumente?
   - Welche Argumenttypen?
   - Tip: Signatur online nachschauen.
2. **Inside class oder outside class?**
   - Inside class definition:
     Operator hat Zugriff auf private Mitglieder.
   - Outside class definition:
     Operator hat keinen Zugriff auf private Mitglieder, kann dafür Typenkonversionen machen.

Back