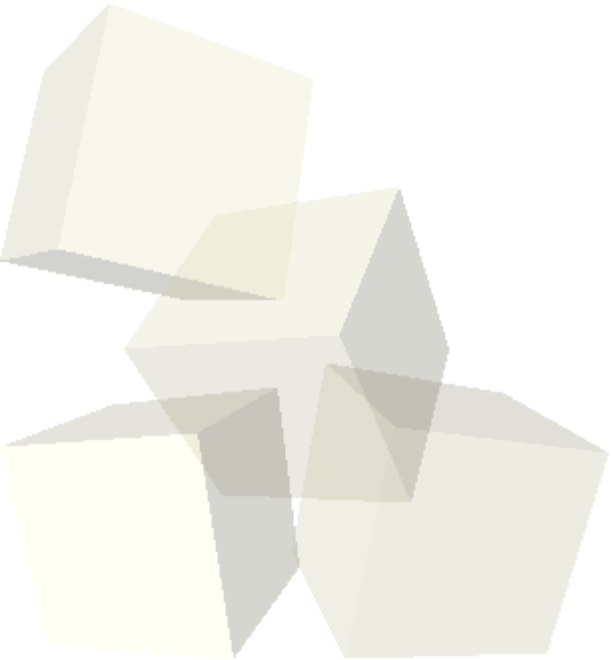
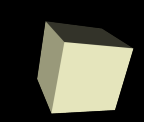
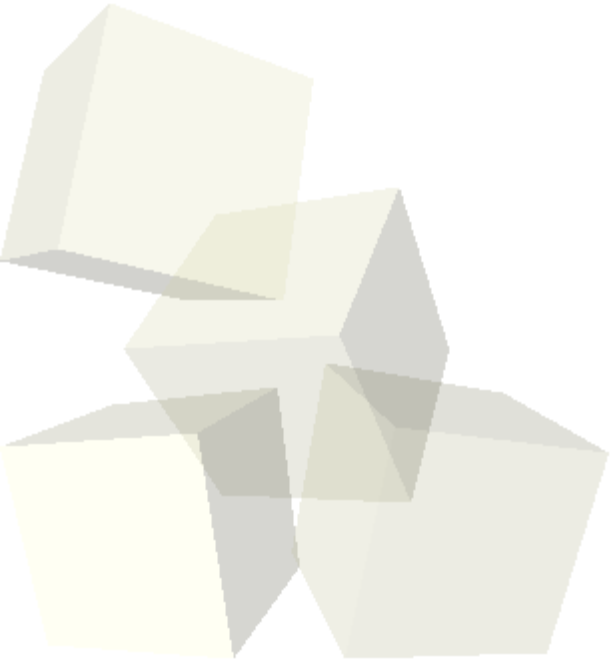
Herzlich willkommen





Agenda

- Include-Guards
- Mehrdimensionale Arrays
- Gleitkommazahlen



Include-Guards

- #include “*file*” kopiert den Inhalt der Datei *file* in die aktuelle Übersetzungseinheit hinein.
- Dabei kann es leicht passieren, dass ein und dieselbe Datei mehrfach eingebunden wird:

b.h

```
struct B
{
    int i;
};
```

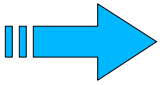
c.h

```
#include "b.h"

struct C
{
    struct B b;
};
```

a.c

```
#include "b.h"
#include "c.h"
```



```
struct B
{
    int i;
};

struct B
{
    int i;
};

struct C
{
    struct B b;
};
```

Include-Guards

- Das mehrfache Einbinden lässt sich mit Hilfe des Präprozessors verhindern. Dazu benötigt man ein eindeutiges Symbol, einen sog. *Include-Guard*:

b.h

```
#ifndef B_INCLUDED
#define B_INCLUDED

struct B
{
    int i;
};

#endif
```

c.h

```
#ifndef C_INCLUDED
#define C_INCLUDED

#include "b.h"

struct C
{
    struct B b;
};

#endif
```

Falls das Symbol noch nicht definiert wurde, wird die Header-Datei gerade zum ersten Mal in die aktuelle Übersetzungseinheit eingebunden.

Dann definieren wir das Symbol, so dass alle späteren Einbindungen wirkungslos bleiben.

Include-Guards

- Mehrfach*definitionen* sind illegal.
- Mehrfach*deklarationen* sind dagegen erlaubt.
- Pragmatisch ist es sinnvoll, grundsätzlich Include-Guards in Header-Dateien zu verwenden.

Deklarationen

```
int compare(char, char);


struct person;


extern int x;
```

Definitionen

```
int compare(char x, char y)
{
    return x - y;
}

struct person
{
    char name[60];
    int alter;
};

int x;
```

Mehrdimensionale Arrays

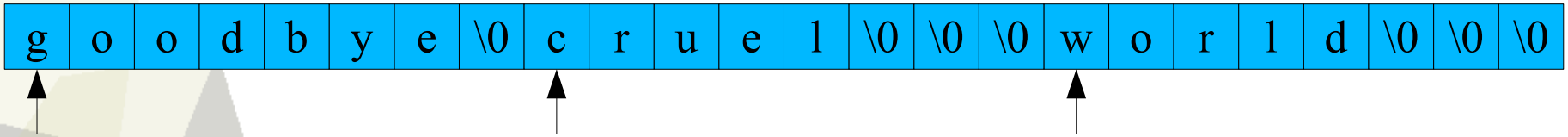
- Arrays sind beliebig tief schachtelbar:
  - int buchstabenzaehler[26];
  - char viergewinnt_spielfeld[6][7];
  - float temperatur[2000][12][31][24]; // 71 MB!
- Mehrdimensionale Arrays sind “row-major” im Speicher ausgelegt:
  - char tictactoe[3][3];

Zweidimensionale Arrays

```
int compare(const void * v, const void * w)
{
    return strcmp((const char *)v, (const char *)w);
}

int main(void)
{
    char a[3][8] = {"goodbye", "cruel", "world"};
    qsort(a, 3, 8, compare);

    return 0;
}
```

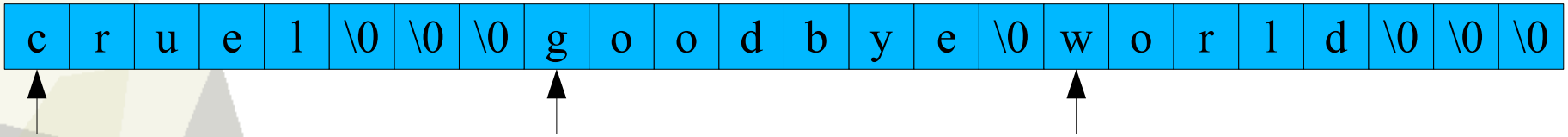


Zweidimensionale Arrays

```
int compare(const void * v, const void * w)
{
    return strcmp((const char *)v, (const char *)w);
}

int main(void)
{
    char a[3][8] = {"goodbye", "cruel", "world"};
    qsort(a, 3, 8, compare);

    return 0;
}
```

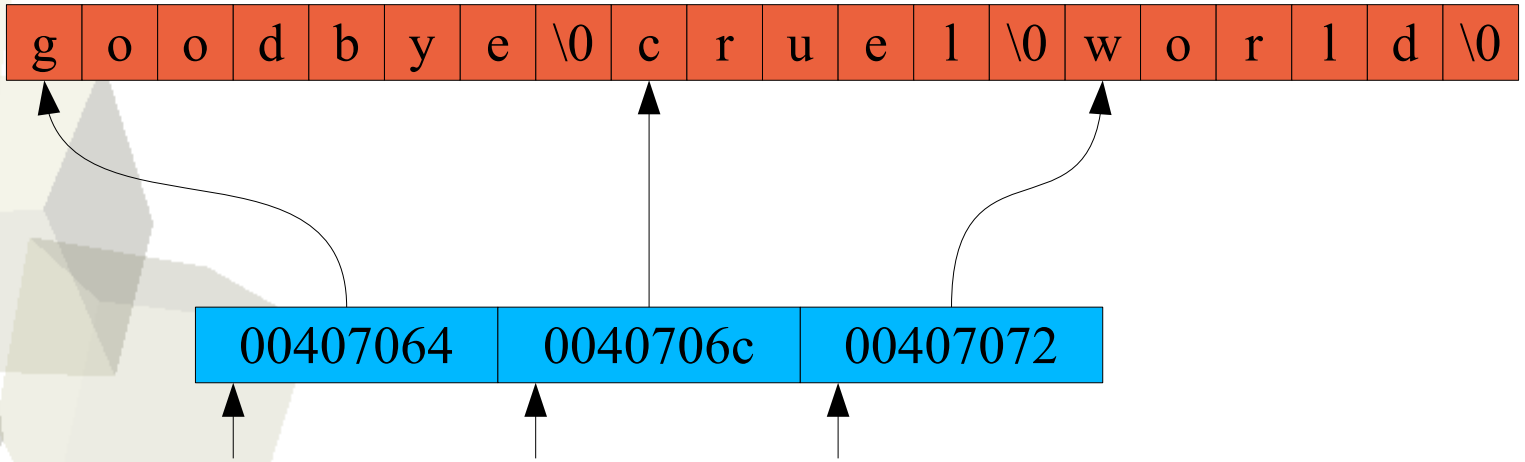


Arrays von Zeigern

```
int compare(const void * v, const void * w)
{
    return strcmp(*(const char * const *)v, *(const char * const *)w);
}

int main(void)
{
    const char * b[3] = {"goodbye", "cruel", "world"};
    qsort(b, 3, sizeof(const char *), compare);

    return 0;
}
```

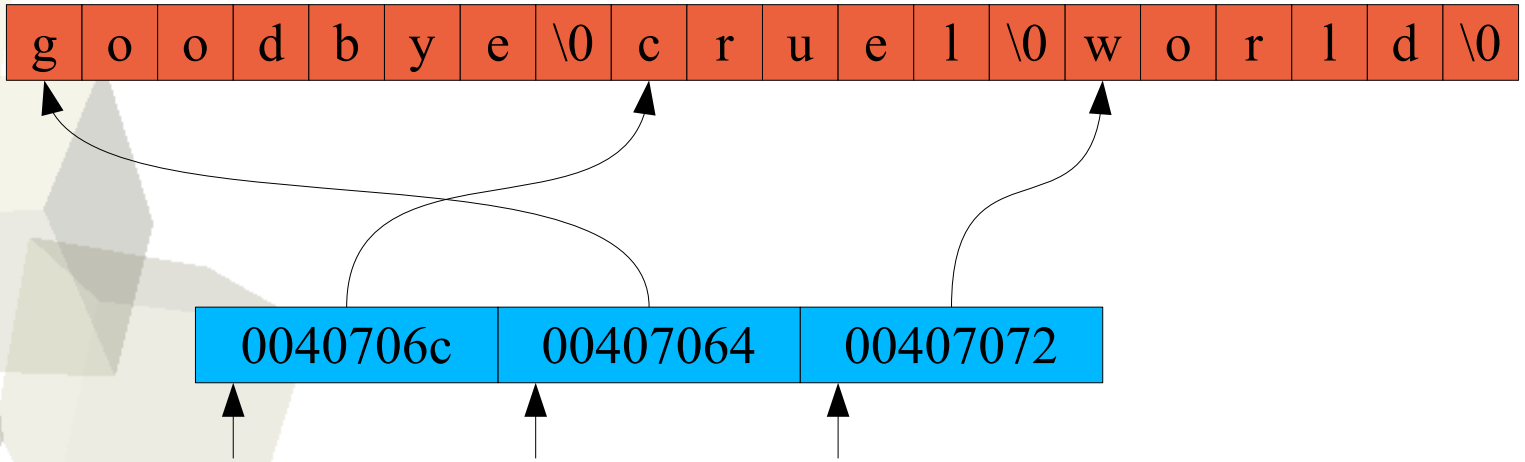


Arrays von Zeigern

```
int compare(const void * v, const void * w)
{
    return strcmp(*(const char * const *)v, *(const char * const *)w);
}

int main(void)
{
    const char * b[3] = {"goodbye", "cruel", "world"};
    qsort(b, 3, sizeof(const char *), compare);

    return 0;
}
```



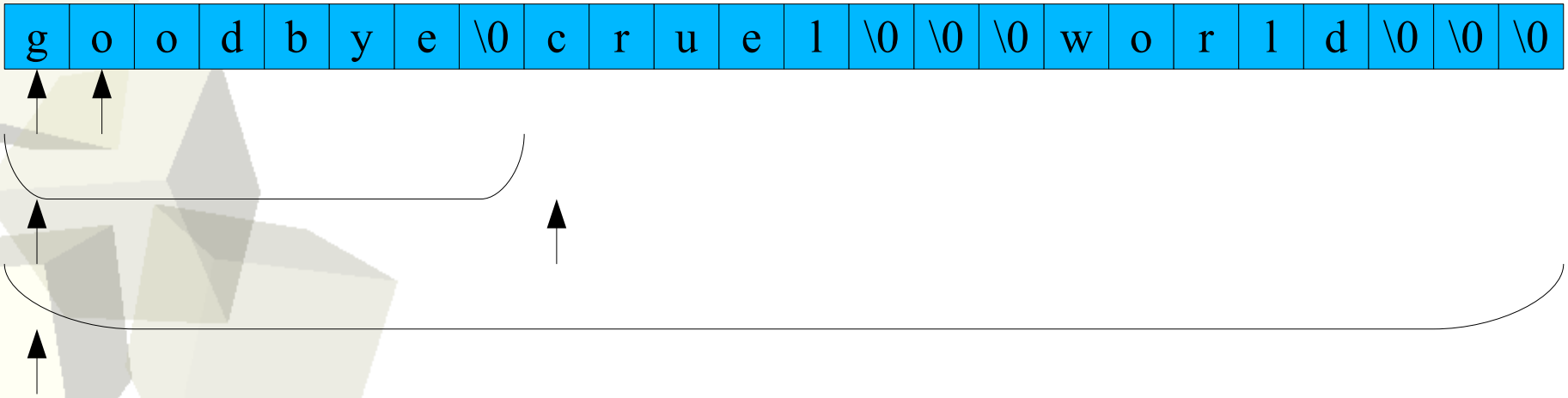
Zeiger vs. Adressen

Die Adresse eines Arrays ist die Adresse seiner ersten Zelle. Trotzdem ist ein Zeiger auf ein Array ein anderer Typ als ein Zeiger auf seine erste Zelle:

```
char a[3][8] = {"goodbye", "cruel", "world"};

char * p  = &a[0][0];   /* Zeiger auf 'g' */
char (*q)[8] = &a[0];   /* Zeiger auf "goodbye" */
char (*o)[3][8] = &a;   /* Zeiger auf {"goodbye", "cruel", "world"} */

printf("%p %p\n", p, p + 1);   /* 0022fefc 0022fefd */
printf("%p %p\n", q, q + 1);   /* 0022fefc 0022ff04 */
printf("%p %p\n", o, o + 1);   /* 0022fefc 0022ff14 */
```



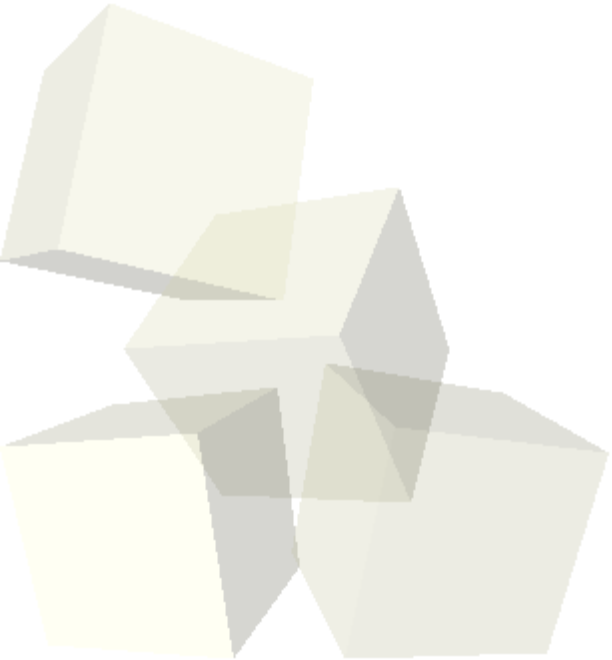
Arrays und Zeiger

- Die implizite Wandlung von Arrays zu Zeigern bezieht sich nur auf die erste Dimension:
  - char a[3][8] = {"goodbye", "cruel", "world"};
  - char (*q)[8] = a; /* q zeigt auf "goodbye" */
- Bei Parametern wird entsprechend nur die erste Dimension ignoriert. Drei identische Signaturen:
  - void f(int a[12][31]);
  - void f(int a[ ][31]);
  - void f(int (*a)[31]);
- Alle anderen Dimensionen müssen zur Übersetzungszeit feststehen. Illegale Signaturen:
  - void g(int a[12][ ]);
  - void g(int a[ ][ ]);
  - void g(int (*a)[ ]);

Gleitkommazahlen

```
import java.math.BigDecimal;

public class DoubleTest
{
    public static void main(String[ ] args)
    {
        System.out.println(new BigDecimal(0.1));
    }
}
```



Gleitkommazahlen

```
import java.math.BigDecimal;

public class DoubleTest
{
    public static void main(String[ ] args)
    {
        System.out.println(new BigDecimal(0.1));
    }
}
```

0.1000000000000000055511151231257827021181583404541015625

Gleitkommazahlen

```
import java.math.BigDecimal;

public class DoubleTest
{
    public static void main(String[ ] args)
    {
        System.out.println(new BigDecimal(0.1));
    }
}
```

Translates a double into a BigDecimal which is **the exact decimal representation** of the double's binary floating-point value.

The results of this constructor can be somewhat unpredictable. [...] 0.1 cannot be represented exactly as a double (or, for that matter, as a **binary fraction** of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1, appearances notwithstanding.

Festkommazahlen

9 , 2 5

Festkommazahlen

9	,	2	5

8	4	2	1

Festkommazahlen

9	,	2	5

8	4	2	1
1	0	0	1

Festkommazahlen

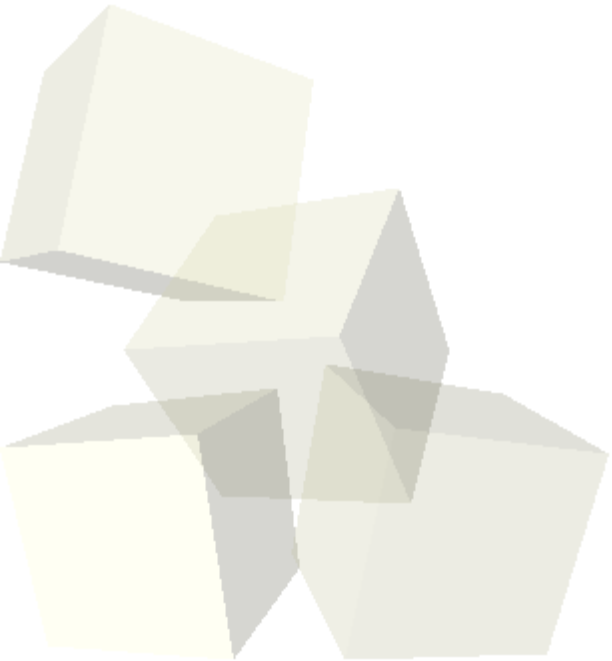
9	,	2	5

8	4	2	1		½	¼
1	0	0	1			

Festkommazahlen

9	,	2	5

8	4	2	1		½	¼
1	0	0	1	,	0	1



Festkommazahlen

Festkommazahlen

0	,	6	8	7	5		0	,	
1	,	3	7	5			0	,	1
0	,	7	5				0	,	1	0
1	,	5					0	,	1	0	1
1							0	,	1	0	1	1

Festkommazahlen

0 , 1

0 , 2

0 , 4

0 , 8

1 , 6

1 , 2

0 , 4

0 ,

0 , 0

0 , 0 0

0 , 0 0 0

0 , 0 0 0 1

0 , 0 0 0 1 1

Festkommazahlen

0 , 1	0 ,
0 , 2	0 , 0
0 , 4	0 , 0 0
0 , 8	0 , 0 0 0
1 , 6	0 , 0 0 0 1
1 , 2	0 , 0 0 0 1 1

0 , 4

Wissenschaftliche Notation

- Für alltägliche Zahlen wie 1,5 ist die bisher verwendete Notation völlig ausreichend.

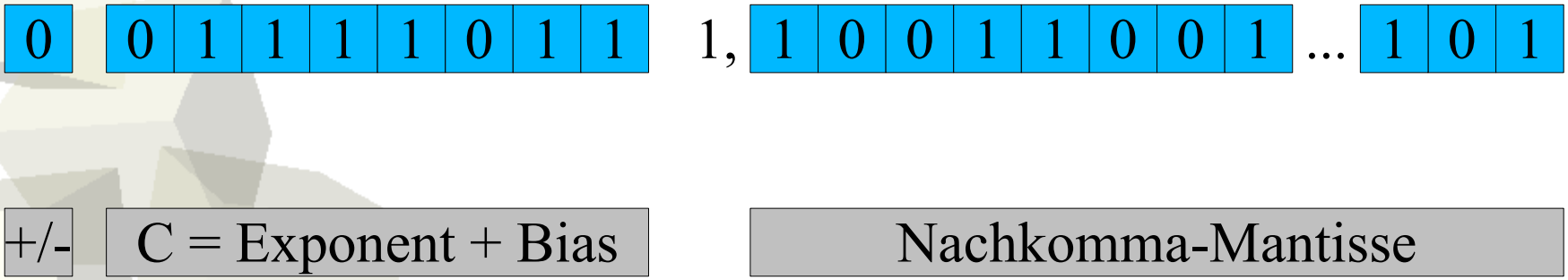
- Für sehr kleine oder sehr große Zahlen verwendet man stattdessen eine andere Notation:
  - Elementarladung: 1,602176487 * 10^-19 C
  - Sonnenmasse: 1,9891 * 10^30 kg
  - Allgemein: Mantisse * Basis^Exponent

- Eindeutige Zahlendarstellung: normierte Mantisse
  - Nur eine Ziffer vor dem Komma, größer als 0
    - ➔ Im Binärsystem also immer eine 1 vor dem Komma
  - Für die Zahl 0 braucht man offenbar eine Sonderregel.

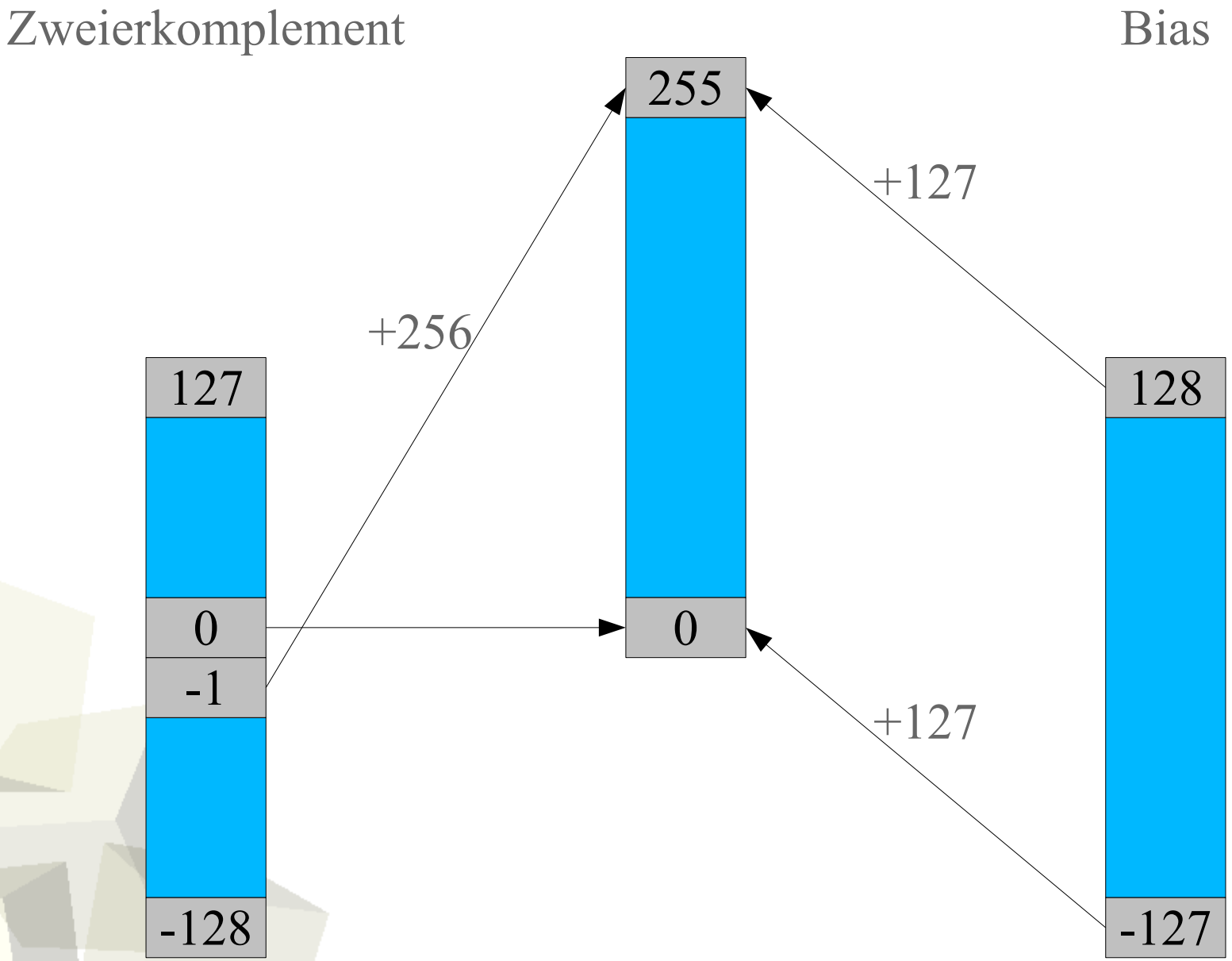
- Übung normierte Darstellung von 15 16 0,75

- Die Repräsentation von float bzw. double basiert auf der wissenschaftlichen Notation:
  - 1 Bit für das Vorzeichen
  - 8 bzw. 11 Bit für den Exponenten
  - 23 bzw. 52 Bit für die Mantisse

- Um negative Exponenten zu ermöglichen, wird auf den Exponent ein Bias B (127 bzw. 1023) addiert.



Negative Zahlen



Normalisiert				1,M * 2^(C-B)
+/-	0 < C < max		1,	M
Denormalisiert				0,M * 2^(1-B)
+/-	0		0,	M > 0
Null				0
+/-	0		0,	0
Unendlich				∞
+/-	max			0
Keine Zahl				NaN
+/-	max			M > 0

Beobachtungen

- Die Anzahl der signifikanten Ziffern hängt von der Breite der Mantisse ab: (ld10 = 3,322)
  - 24 / ld10 = 7,225 (float)
  - 53 / ld10 = 15,95 (double)
- Die Größenordnung vom Exponenten:
  - 2^127 = 10^38 (float)
  - 2^1023 = 10^307 (double)

- Die meisten endlichen Dezimalbrüche sind in binärer Repräsentation periodisch.

- Für wissenschaftliche Zwecke ist diese Ungenauigkeit irrelevant, für Finanzsoftware nicht.
  - Java hat für Dezimalbrüche die Klasse BigDecimal.