Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

- numbers can be elements from various sets. e.g. $x \in \mathbb{Z}$, $x \in \mathbb{N}$.
- each number has various representations. e.g.
  - 17
  - XVII
  - IIIII IIIII IIIII II

- usually, we encode numbers with the help of base-10 digits, i.e. the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
  A string $s = (a_{n-1}a_{n-2}... a_1a_0) \in \Sigma^n$, is then interpreted as

  $$\Sigma^{n-1}_{i=0}\, a_i \cdot 10^i \quad \text{Example: } 17 = 1\cdot10^1 + 7\cdot10^0$$

- What happens, if we use another base, another alphabet?
  with bits:
  $\Sigma_2 = \{0, 1\}$ $\quad 17_{10} = 1\cdot2^4 + 0\cdot2^3 + 0\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = 10001_2$

  $\Sigma_{16} = \{0, 1, 2, 3, 4, 5 , 6, 7, 8, 9, a, b, c, d, e, f\}$
  $17_{10} = 1\cdot16^1 + 1\cdot16^0 = 11_{16} = 0x11$ hex numbers

TECHNISCHE UNIVERSITÄT DARMSTADT

Integers II

integer variables of fixed length are the most natural and mostly used kind of variables

Bitstrings are interpreted as numbers in the dual number system.

The value then is $bit_{31} \cdot 2^{31} + bit_{30} \cdot 2^{30} \ldots bit_0 \cdot 2^0$.

TECHNISCHE UNIVERSITÄT DARMSTADT

Integers II

Computing with binary numbers

sum:

```
  1 0 1 1
+   1 1 1
-------
  1 1 1 0
```

product:

```
1011 · 101
-----
1011
0000
1011
-----
110111
```

base-10

```
  9 9
+   3
-----
  1 0 2
```

Integers II

Computing with binary numbers of fixed length

sum:

$$\begin{array}{rrrrr} & 1 & 0 & 1 & 1 \\ + & 0 & 0_1 & 1_1 & 1 \\ \hline & 1 & 1 & 1 & 0 \end{array}$$

sum with ignored overflow:

$$\begin{array}{rrrrr} & 1 & 0 & 1 & 1 \\ + & 1 & 0_1 & 1_1 & 1 \\ \hline (1) & 0 & 1 & 1 & 0 \end{array}$$

$(11_{10} + 11_{10}) \bmod 16_{10} =$
$22_{10} \bmod 16_{10} = 6_{10}$

n bits -> 2^n values -> can be interpreted as a ring of equivalence classes. Any bit combination, representing the number $0 \le j < 2^n$, also represents the integers modulo 2^n, i.e. the set $\{ j + k \cdot 2^n$ with $k \in \mathbb{Z} \}$. Sums and products are well defined.

TECHNISCHE UNIVERSITÄT DARMSTADT

Integers II

What about negative numbers?

Idea 1: take 1 bit for the sign, the rest is left as it is.

Idea 2: use the so called two's-complement for representation of integers in the range of -2^{N-1} to $+ 2^{N-1}-1$, N being the number of bits.

Example: 8-bit twos-complement integers

0 1111111 = 127
0 1111110 = 126
0 0000001 = 1
0 0000000 = 0
1 1111111 = -1
1 1111110 = -2
1 0000000 = -128

How to build a negative number:
- build the bit inverse
- add 1

Example: $2_{10} = 00000010_2$

~00000010 = 11111101

```
  11111101
+ 00000001
----------
  11111110 = -2
```

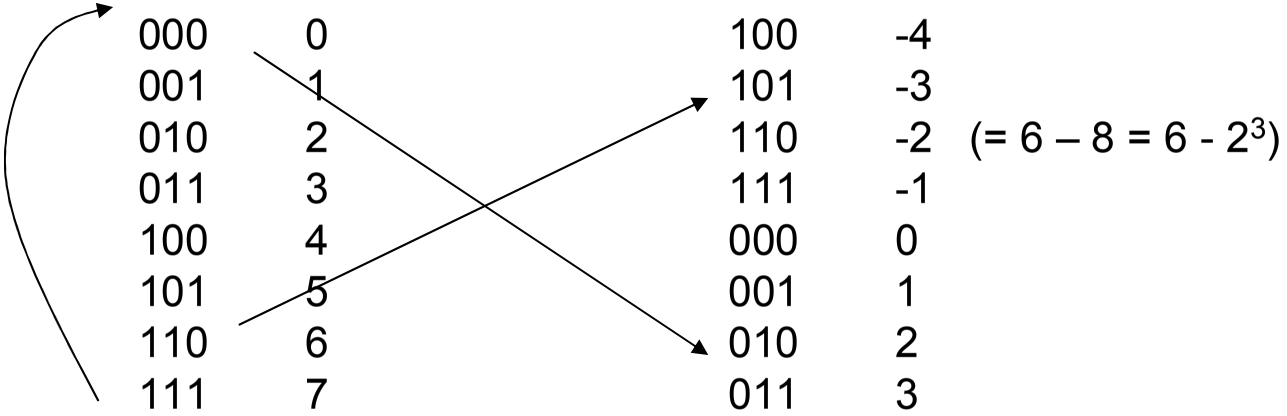
TECHNISCHE UNIVERSITÄT DARMSTADT

Integers II

Why does the two's complement work?

Simple idea: substract 2^n from the upper half of the numbers, i.e.

rotate the order of the bitstrings and re-interprete the bitstrings with MSB 1



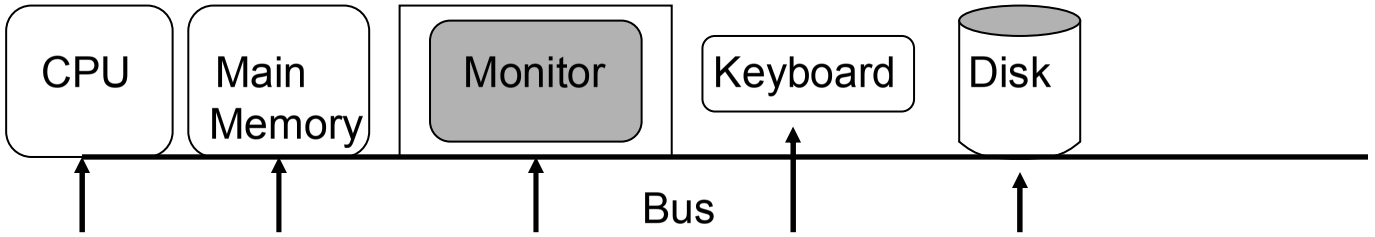
bin	dec	bin	dec	
000	0	100	-4	
001	1	101	-3	
010	2	110	-2	(= 6 – 8 = 6 - 2^3)
011	3	111	-1	
100	4	000	0	
101	5	001	1	
110	6	010	2	
111	7	011	3	

two' complement(x) = bit-complement(x) + 1 = ((2^n – 1) – x) + 1 = 2^n – x
und (2^n – x) mod 2^n = (2^n – 2^n – x) mod 2^n
Example: two's complement(3_{10}) = bit-complement(11_2) + 1_2 = 111 – 011 + 001 =
[1]000 – 011 = -11_2 = -3_{10}

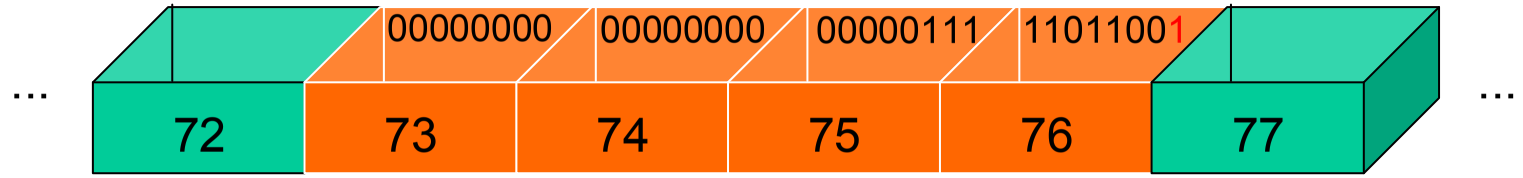
My idealized computer

TECHNISCHE UNIVERSITÄT DARMSTADT

- For our programming course, a computer mainly consists of



Memory:



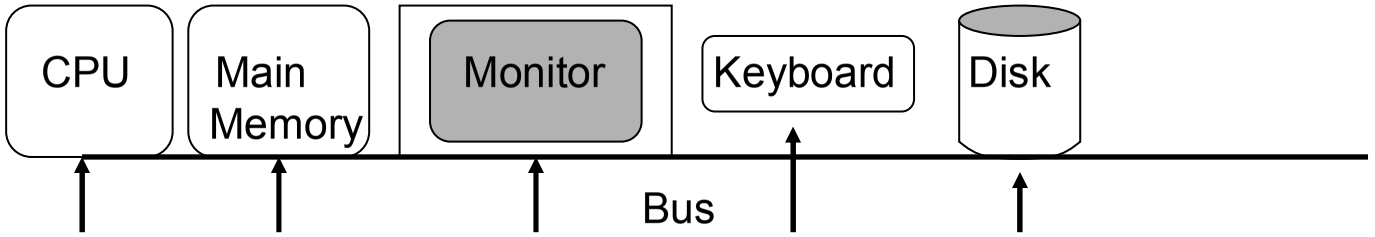
CPU:

- picks a number from a memory cell
- computes an operation from that number and from another cell
- writes back the result to one of the memory cells

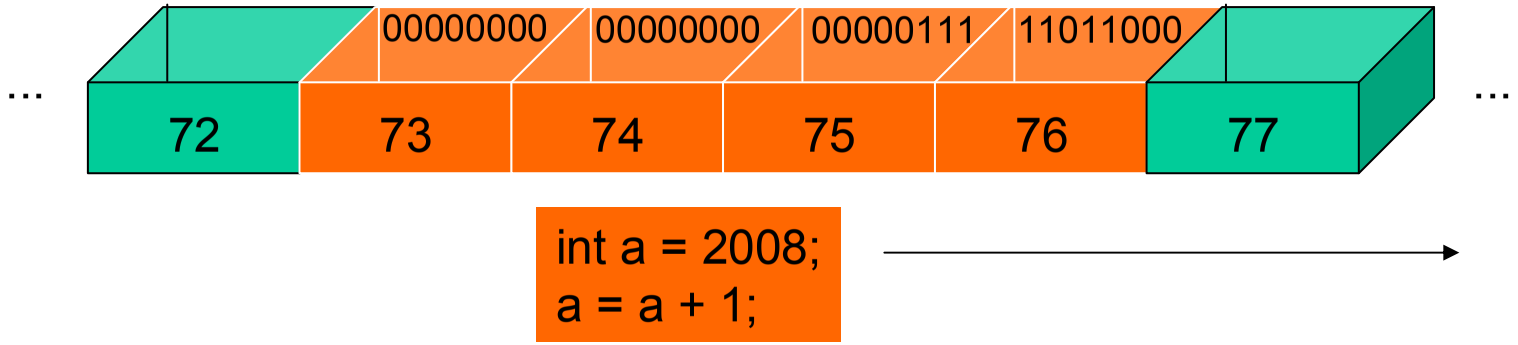
My idealized computer

TECHNISCHE UNIVERSITÄT DARMSTADT

- For our programming course, a computer mainly consists of



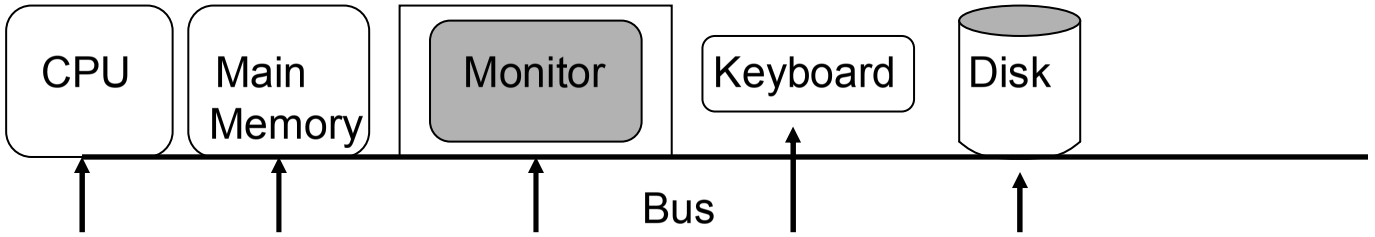
Memory:



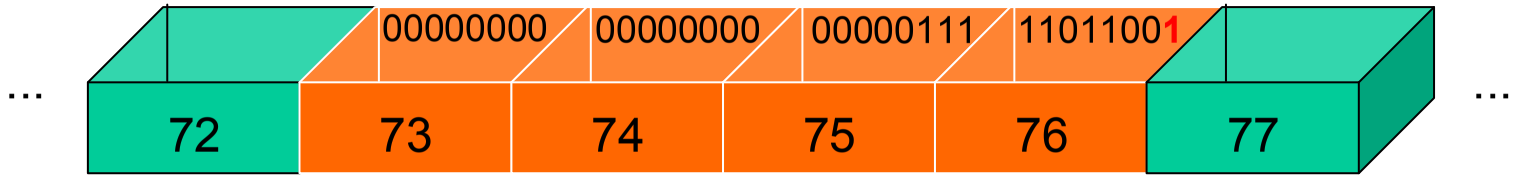
My idealized computer

TECHNISCHE UNIVERSITÄT DARMSTADT

- For our programming course, a computer mainly consists of



Memory:

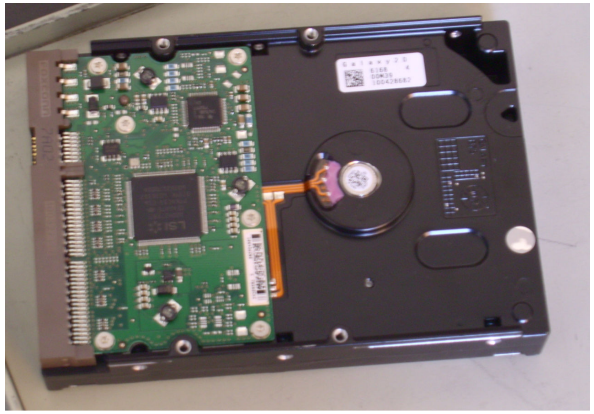


The real computer

TECHNISCHE UNIVERSITÄT DARMSTADT

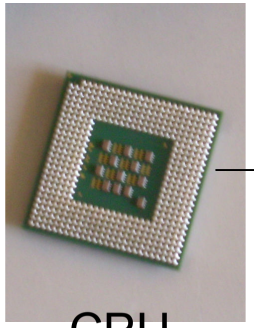
- A real CPU
  - is quite a complex device
  - contains a few memory cells, so called registers
  - interpretes sequences of 0s and 1s as sequences of instructions, like “set a register to 0”, or “add the content of a register to a memory cell“
  - ...
- real main memory
  - Consists of many so called bits with value 0 or 1. Typically, bits are grouped to so-called bytes (8 bits) or words (32 or 64 bits) and the main memory size is expressed with the help of its number of bytes. Typically, a personal computer has between 512 Mbyte and 32 Gigabyte RAM (random access memory) and 1TByte Diskspace.
- quite a complicated circuit manages data transfer between CPU, memory and other components.

RAM memory module

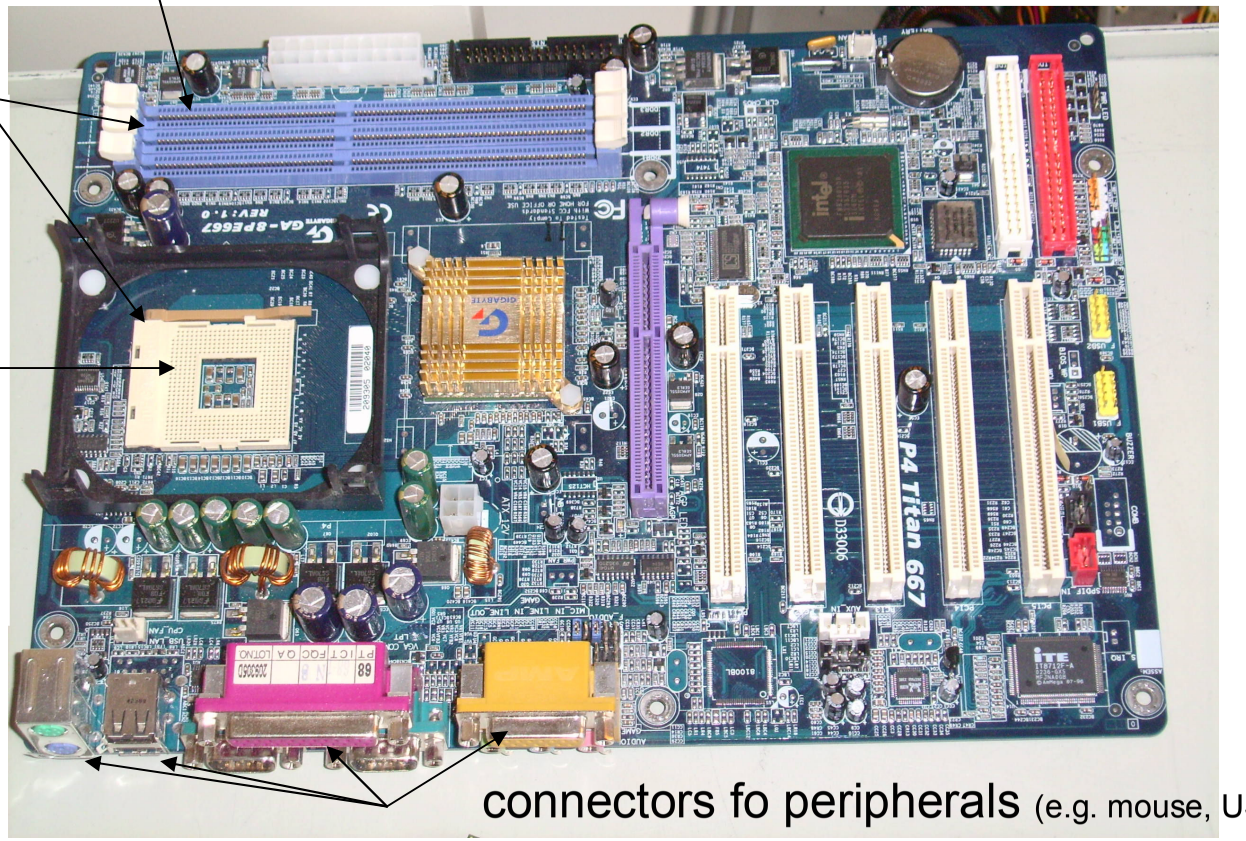


disk

sockets



CPU



connectors fo peripherals (e.g. mouse, USB, ethernet ...)

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Generalized binary fixed-point and floating-point numbers

0.75

$0.75 = 1 * 1/2 + 1* 1/4 = 0.11_2$

0.7

$0.7 = 1 * 1/2 + 0* 1/4 + 1 * 1/8 + 1/16 + ...$

the first 64 bits:

0.1011001100110011001100110011001100110011001100110011001100110011

0.7 is a periodic number in the binary system.

-> representation errors in IEEE format is not avoidable

-> x = 0.7; x = 11.0 · x – 7.0; increases the error by a factor of 10

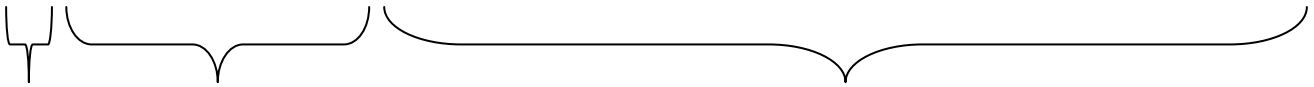
TECHNISCHE UNIVERSITÄT DARMSTADT

Interpretation of the type float

floating point variables

0/1 sequences are interpreted as sign (s) , mantissa (m) and exponent (p)

0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 1

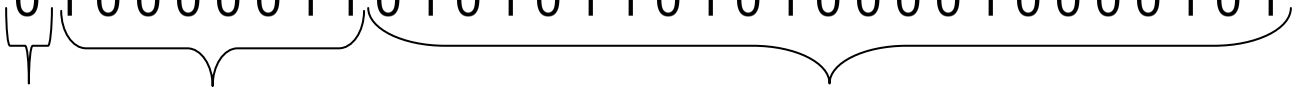


s p m

the resulting number then is $s \cdot m \cdot 2^p$

In the IEEE-754 standard, 127 is added to the exponent, and the leading 1 of the mantissa is not stored. The exponent has 8 bits and the mantissa 23 explicit bits, thus 24 implicit bits.

0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 1



s p‘ m‘

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

The type of a variable determines how much memory is reserved.

	sign	smallest value	largest value	size in bits
char	yes	-128	127	8
unsigned char	no	0	255	8
short	yes	-32768	32767	16
unsigned short	no	0	65535	16
long	yes	-2147483648	2147483647	32
unsigned long	no	0	4294967295	32
float	yes	$1.2 * 10^{-28}$	$3.4 * 10^{38}$	32
double	yes	$2.32 * 10^{-308}$	$1.8 * 10^{308}$	64
int	yes	depends on machine and OS		
unsigned int	no			
size_t	no			
ssize_t	yes			
off_t	no			
long long	yes	-2^{63}	$2^{63} -1$	64
unsigned long long	no	0	$2^{64} - 1$	64
long double	yes	$3.4 * 10^{-4932}$	$1.2 * 10^{4932}$	96

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory II: Pointers and arrays

The statement

int a[10];

defines an array with name ‘a’ with 10 elements, somewhere in memory:

a : 

a[0] a[1] ... a[9].

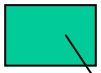
a[i] is the i-th element of the array, counting from 0.

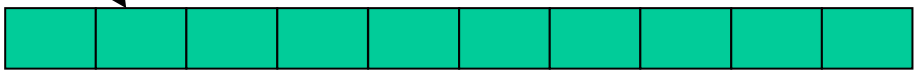
int *pa; defines a ‘pointer’ to an integer variable.

We can initialize it e.g. with

pa = &a[1];

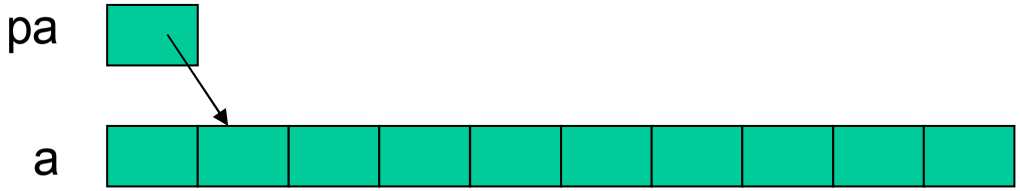
This means, pa holds the memory address of a[1].

pa 

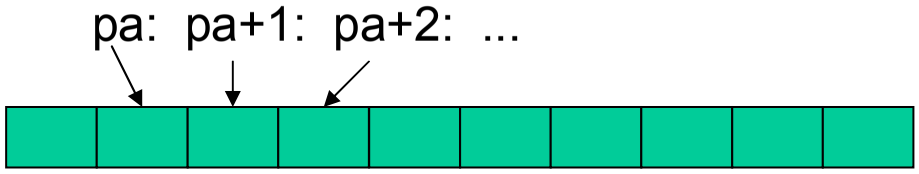
a 

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT



x = *pa; dereferences pa and assigns a[1] to x.



This works independently of the array's / pointer's data type:

Programming language C

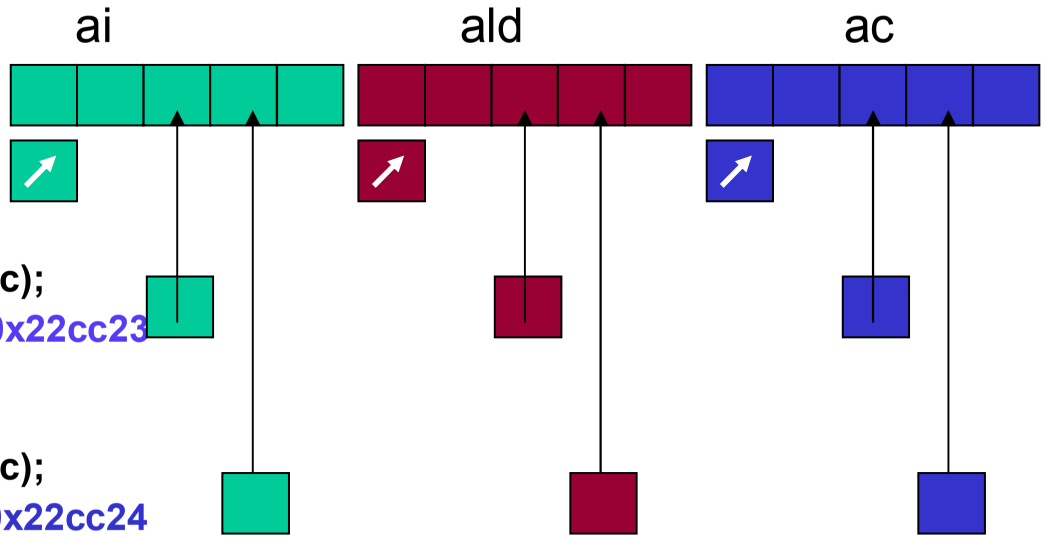
TECHNISCHE UNIVERSITÄT DARMSTADT

```
int ai[5]; long double ald[5]; char ac[5];
int *pi;    long double *pld;    char *pc;

pi = &ai[2]; pld = &ald[2]; pc = &ac[2];
printf("pi=%p, pld=%p, pc=%p\n",pi,pld,pc);
output: pi=0x22ccbc, pld=0x22cc54, pc=0x22cc23

pi = &ai[3]; pld = &ald[3]; pc = &ac[3];
printf("pi=%p, pld=%p, pc=%p\n",pi,pld,pc);
output: pi=0x22ccc0, pld=0x22cc60, pc=0x22cc24

printf("#bytes: int:%d, long double:%d, char:%d\n",
      (unsigned int)(pi+1)-(unsigned int)pi,
      (unsigned int)(ald+1)-(unsigned int)ald,
      (int)(pc+1)-(int)pc);
output: #bytes: int:4, long double:12, char:1
```



- "&" is the address operator. It returns the numerical value of the storage location where the value of a certain variable isstored in memory.
- "(type) variable" casts (= transforms) a variable of one type to another type