Einführung in C - Introduction to C

7. Pointers and memory management

Prof. Dr. Eckhard Kruse

DHBW Mannheim

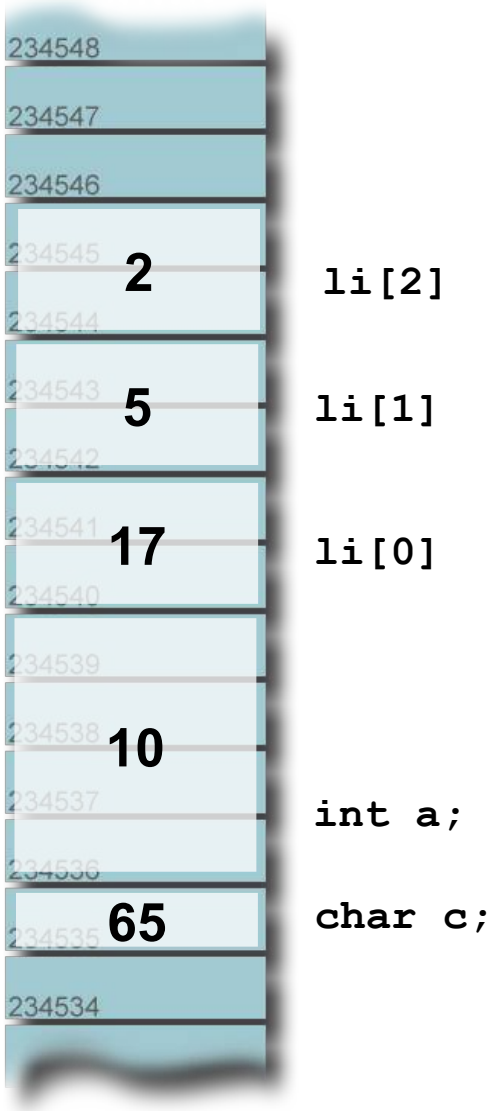
Variables and memory

A Variable is a place in computer memory, where values can be stored.

- The size of required memory depends on the type.
- How and where the memory is reserved is not directly controlled by the programmer.
  - Local variables: memory is reserved when the scope is entered and freed when it is left
  - Static/global variables: memory is reserved throughout the program's lifetime.

```
int a;            // reserves 4 bytes of memory
a=10;             // write 10 into the 4 bytes

char c='A';       // 1 byte
short li[3]={ 17, 5, 2 };  // 3*2 Bytes
```



Sizeof and &

Definition

The **sizeof** operator determines the size (in bytes) a data type or variable is using in memory.

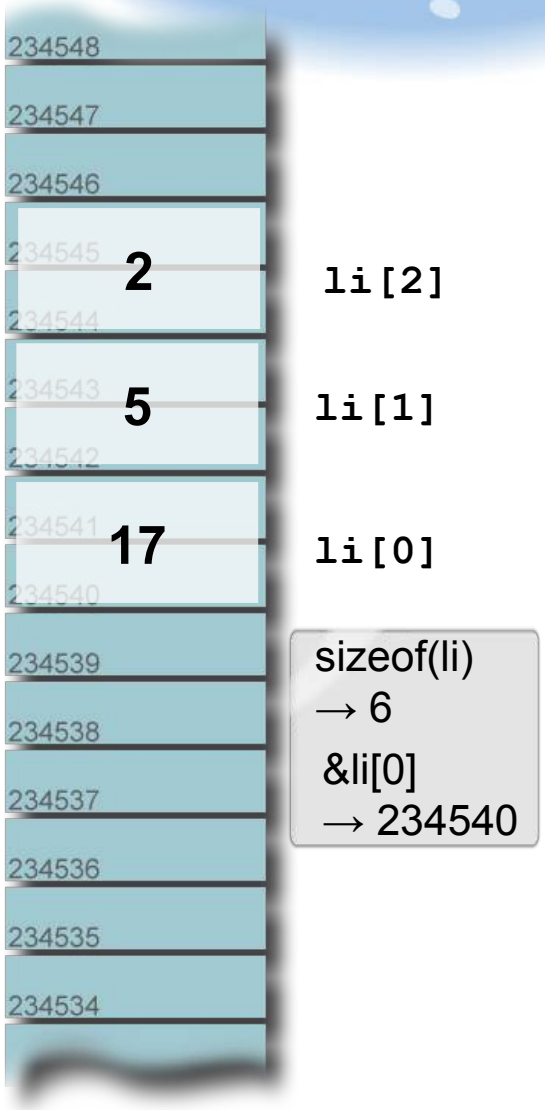
```
short s;
int array[4];

printf("%d", sizeof(short));
printf("%d", sizeof(s));
printf("%d", sizeof(array));
printf("%d", sizeof(array[0]));
printf("%d", sizeof("Hallo"));
```

compile-time vs. run-time evaluation...

The **address operator &** provides the address, where a variable is stored in memory.

```
printf("%d", &s);
printf("%p", &s); // pointer format: hex
printf("%d", &array[0]);
printf("%d", array);  // same as &array[0]
printf("%d", &"Test");
```



Variables and memory

Exercise

`variables_and_memory.c`

Code snippet 701

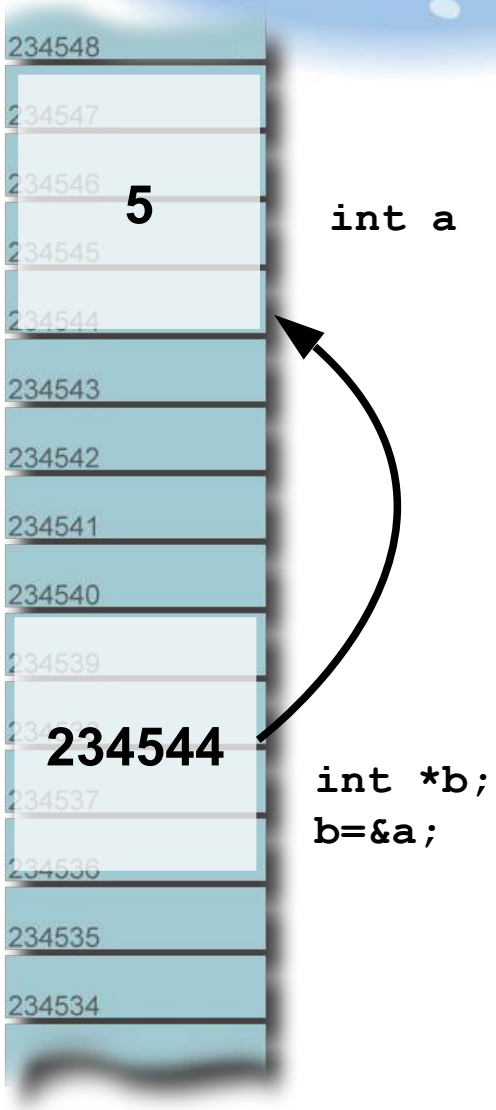
Pointers

Definition

A pointer is a variable (or constant) pointing to an address in memory where a value (of some datatype) is stored:

- Declaration: **`datatype *pointer_name`**
- * deferences the pointer, i.e. not the pointer but the value in the address it is pointing to is accessed.

```
int a;
int *b, *c;  // pointers to int values
b=&a;        // let b point to address of a
*b=5;        // store a 5 at this address
// null pointer: indicate invalid pointer:
c=0; /* or */ c=NULL;
```



use pointers to:

- pass variable parameters to functions (call by reference)
- create dynamic data structures, i.e. which are stored in memory allocated at run-time
- access information stored in arrays/strings (as alternative to using the index with […])

Call by value

Example

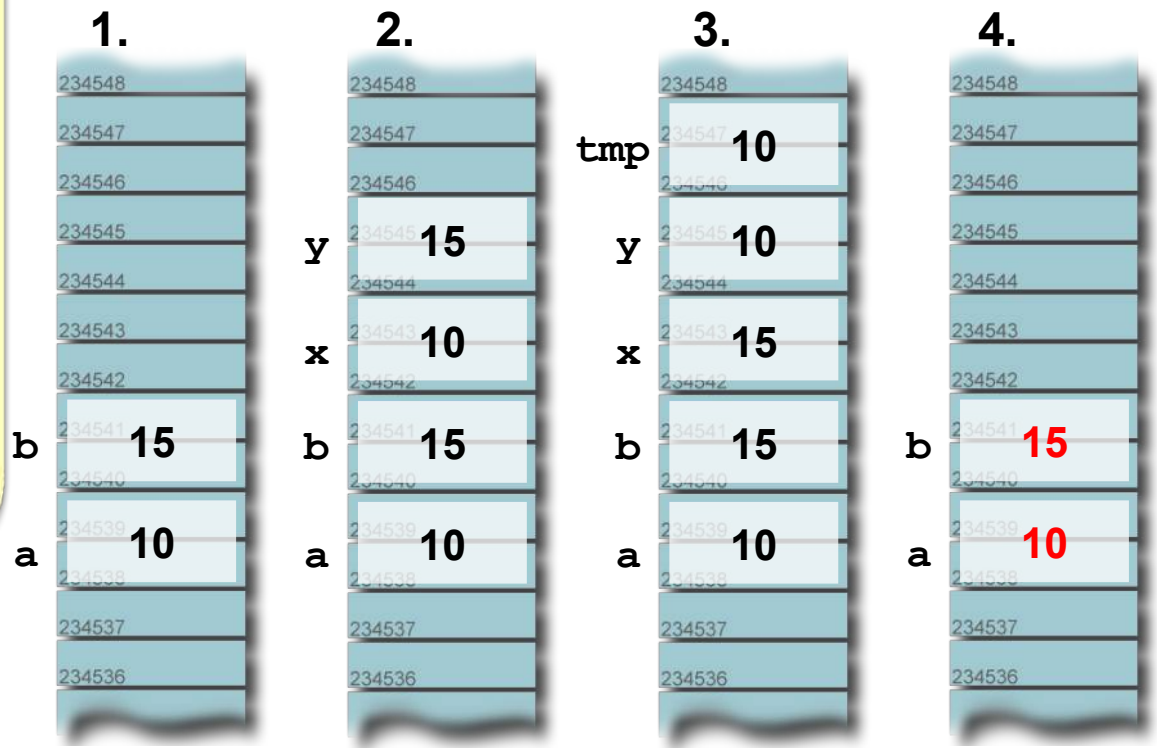
```
main()
{
    short a, b;
    a=10; b=15;
    swap(a,b);
    printf("%d %d",a,b);
}

swap(short x, short y)
{
    short tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

1
4
2
3

This version of swap does not work:

- x, y are **copies** of a and b
- a and b are not touched in swap!



Call by reference

Example

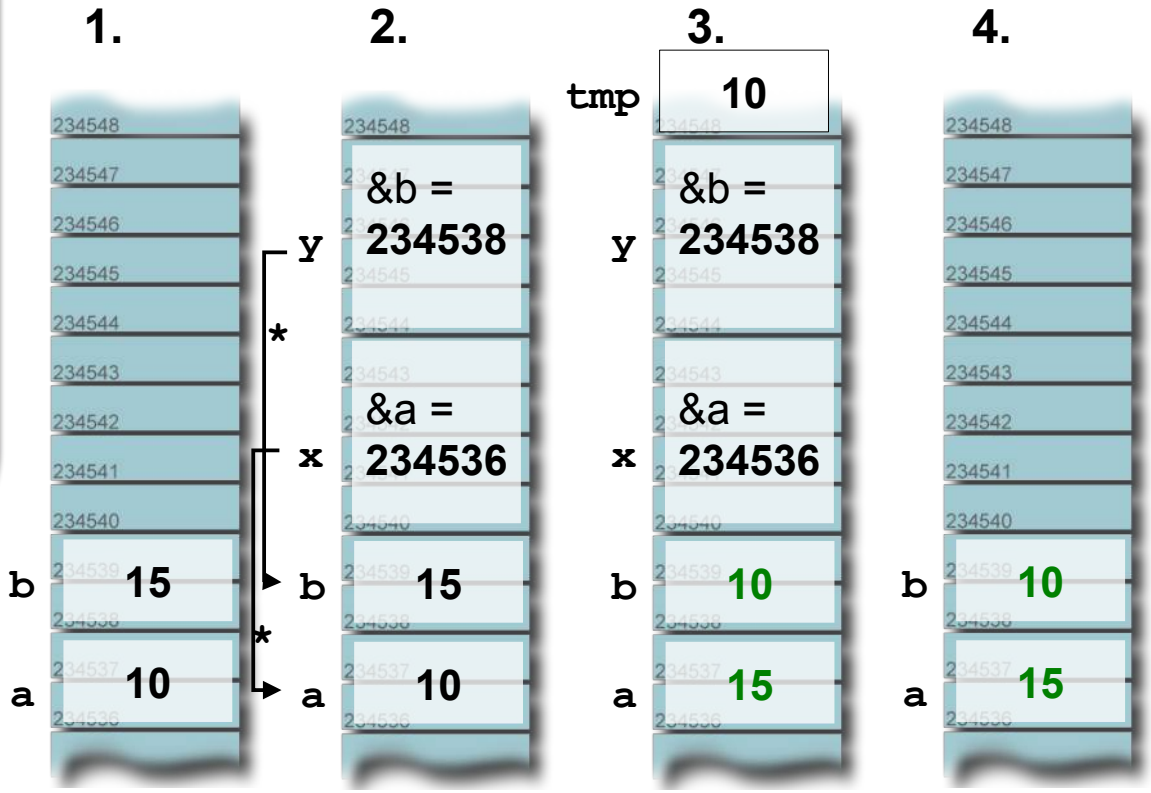
```
main()
{
    short a, b;
    a=10; b=15;
    swap(&a,&b);
    printf("%d %d",a,b);
}

swap(short *x, short *y)
{
    short temp;
    temp=*x;
    *x=*y;
    *y=temp;
}
```

1
4
2
3

This version of swap does work:

- Not the values, but the addresses of a and b are passed!



Pointers to structs

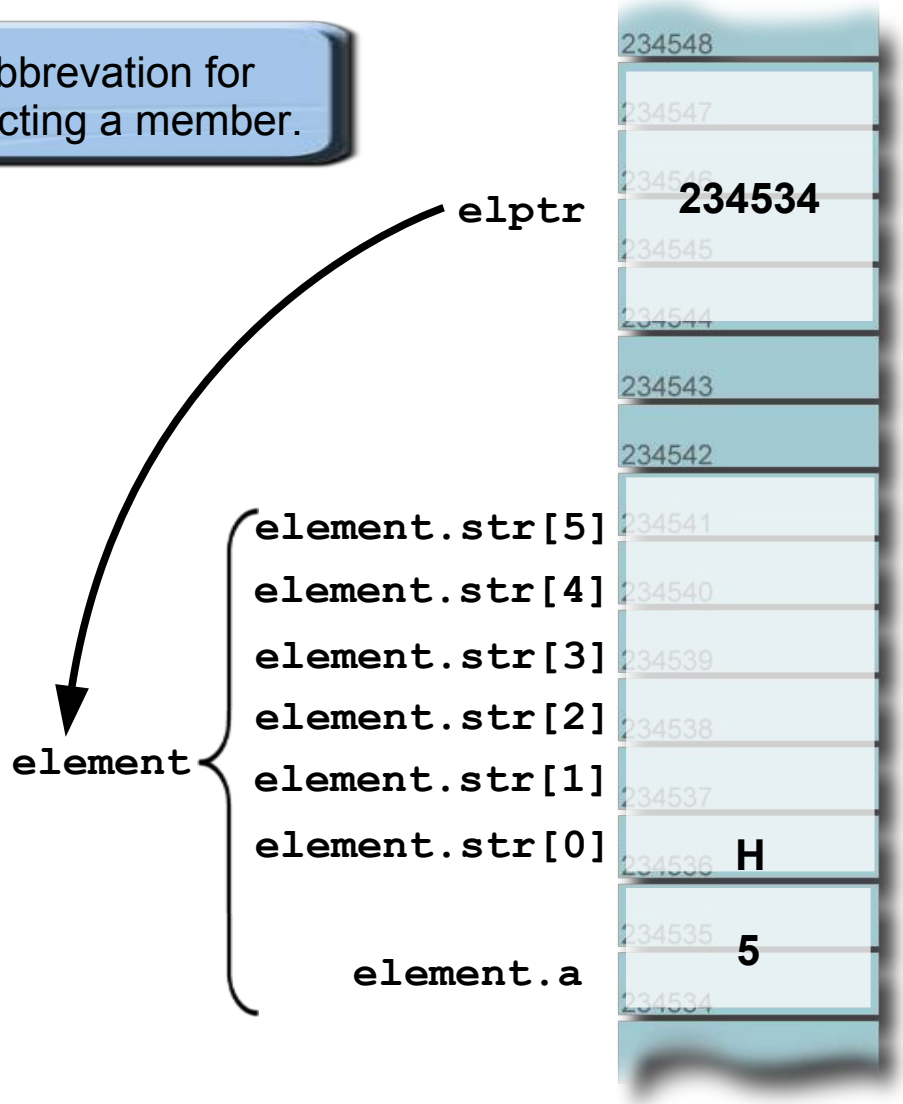
The **arrow operator ->** is a convenient abbrevation for derefencing a pointer to a struct and selecting a member.

```
struct test {
   short a;
   char str[6];
};

main()
{
  struct test element;
  struct test *elptr;

  element.a= ...;
  element.str[0]= ...;

  elptr=&element;
  (*elptr).a = 5;
  // or shorter:
  elptr->a = 5;
  elptr->str[0] = 'H';
}
```



Pointers and structs

Exercise

`pointers.c`

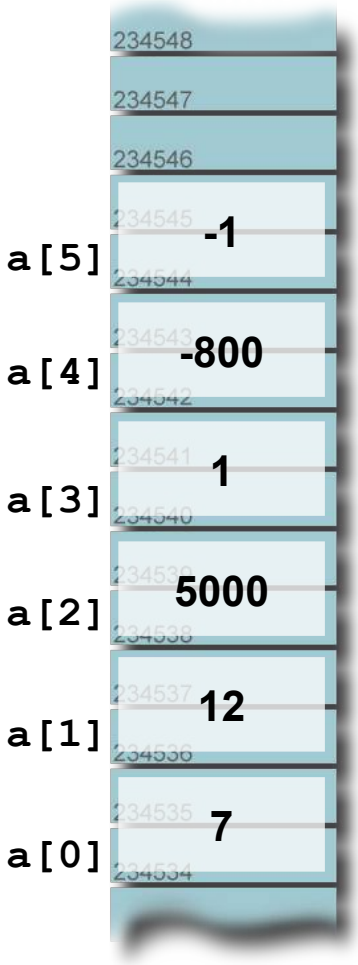
Code snippet 702

Pointers and arrays

```
short a[6]= { 7, 12, 5000, 1, -800, -1 };
short *ptr;

ptr=a;                    // or: ptr=&a[0];
printf("%d",*ptr);        // -> 7
printf("%d",ptr[0]);      // -> 7
printf("%d",ptr[2]);      // -> 5000
printf("%d",*(ptr+2));    // -> 5000
//  +2 does not simply add 2 to the ptr address, but it
    rather adds 2*sizeof(short), because ptr is short*


// pointers can be cast to point
// to different types:
unsigned char *ptr2;
ptr2=(unsigned char *)a;
printf("%d", *ptr2);
```



Good idea?

```
char *getPointerToName() {
   char name[100];

   scanf("%s", name);
   return name;
}

main()
{
   ...
   char *n;
   n=getPointerToName();
   printf(n);
}
```

Comments?

Malloc and free

The C function **malloc** allocates memory dynamically, i.e. during program execution. The required size needs to be provided and a pointer to the memory area is returned (0/NULL indicates an error): *void *malloc(size_t size);*
The C function **free** frees memory which is no longer needed: *void free(void *ptr);*

```
#include <stdlib.h>

int *ptr;

ptr = malloc(100 * sizeof(int));

if(ptr==NULL) // pointer==0 -> error
   printf("Could not allocate");
else {
  *ptr=25;
  ptr[99]=10;
  ...
  free(ptr); // free memory
}
```

A *void* pointer indicates that the type of data the pointer points to is unknown. Before accessing data with *, the pointer needs to be cast or assigned to a typed pointer.

wrong use of malloc and free is a major source of program crashes:

- allocate the correct size
- check for null pointers
- each malloc() should have it's free()
- don't use pointers after free

Malloc and free

Exercise

`malloc_eratosthenes.c`

Code snippet 703