Reminder

- Wahl der Übungsgruppe in Moodle
- Ausfüllen und hochladen des Clusterantrags in Moodle
- Abgeben des Clusterantrags im Original

Programming in C

C compared

- C is old (developed 1969–1973)
- Is “simple” as in
  - A compiler can have < 10,000 lines of code
  - Has just a few features
- Used in many toy examples
- Gets ‘regular’ standards
  - C95, C99, C11, C17, C23

Total

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

C compared with Java

- Syntax and keywords are very similar
- Main differences
  - C compiles to machine code
  - C-Pointers allow for pointer arithmetic
  - No classes and objects in C
  - Manual memory management
  - No built-in string data type in C
  - C has preprocessor
  - Different standard library functions (affects, e.g., I/O)

Disclaimer

- We are cutting corners everywhere
- We are simplifying many things
- If you want to be a language-lawyer, you have to consult the standard

Let’s start simple

```
int multiple_of_23(int number) {
        for (int i = 0; i < 100; i++) {
                int k = i * 23;
                if (k = number) {
                        return 1;
                }
        }
        return 0;
}
```

Functions are similar

```
int multiple_of_23(int number) {






      return 0;
}
```

The return type is like Java, you can also use `void`

The parameter is like Java

The naming is like Java

The return statement is like Java

for-loops are similar

```
for (int i = 0; i < 100; i++) {



}
```

The syntax is like Java

First clause initializes a variable

Second clause is the condition

Third clause is the post-iteration statement

if-conditions … oh no

```
int k = i * 23;
if (k = number) {
        return 1;
}
```

Arithmetic statements are like Java

If conditions are similar to Java

if-conditions in C

- C did not have `bool` as a datatype until 1999
- In C, conditions are evaluated to an arithmetic value
  - `0` represents `false`
  - Everything else represents `true`

What happened?

```
int k = i * 23;
if (k = number) {
        return 1;
}
```

Have you seen:
`int i = j = k = 2;`
before?

Same here:
`k = number`
returns number

The output?

```
int multiple_of_23(int number) {
        for (int i = 0; i < 100; i++) {
                int k = i * 23;
                if (k = number) {
                        return 1;
                }
        }
        return 0;
}
```

- If `number == 0`
  - `0`
- If `number != 0`
  - `1`

`==, <, >, <=, >=, !=`
work as in Java

Types in C

Type name	Minimum size in bits	Explanation
`(signed/unsigned) char`	8	Smallest addressable unit. Is an integer type. Holds characters.
`(signed) short (int)`	16	[-32,767; +32,767]
`unsigned short (int)`	16	[0; 65,535]
`long`	32	At least $[-2^{31}+1, +2^{31}-1]$
`long long`	64	At least $[-2^{63}+1, +2^{63}-1]$
…	…	…
`float`	Usually IEEE 754 32 bit	
`double`	Usually IEEE 754 64 bit	
…	…	…

Combined types in C

- Assume you already have some types `t1` and `t2`

```
struct new_type {
        t1 name1;
        t2 name2;
        ...
}

struct super_new_type {
        struct new_type no_cookie;
}
```

Array types in C

- Assume you already have some type `t1`

```
t1 my_array[32];
char name[64];


float rotation_matrix[4][4];
float other_matrix[4][5];
```

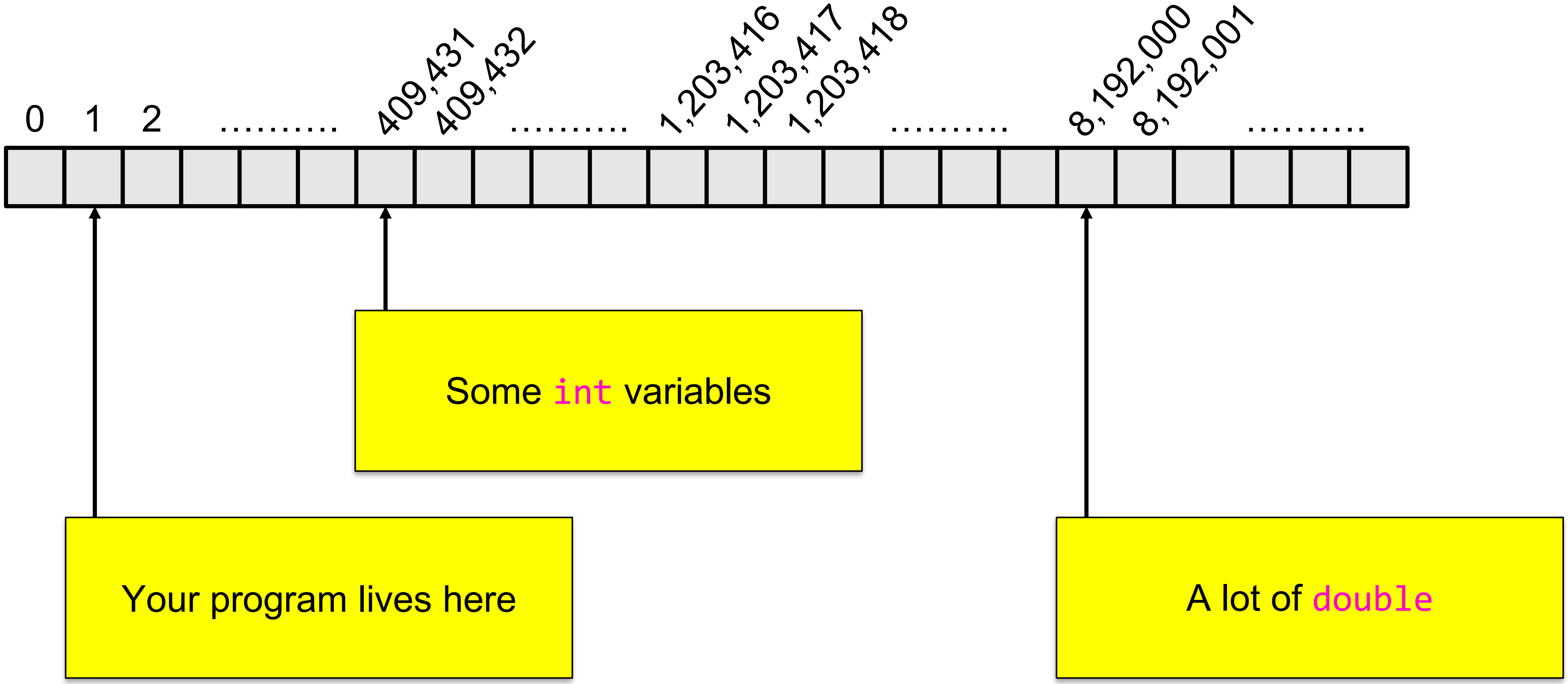
`other_matrix`
has 4 elements,
each has 5 elements

Memory layout

- In your machine, everything resides in memory
  - The program code
  - The data
  - I/O devices (memory-mapped I/O)

- Memory is a large ‘list’ of numbered cells
  - Usually 8bit wide

Exemplary memory layout



Good-to-knows

- If you have two addresses, a1 and a2, then they point to the same memory location if and only if they are equal
  - **and come from the same memory pool*
  - More in the CUDA lectures
- If you have two addresses, a1 and a2, the addresses between them might not be accessible for you

=> Segmentation fault

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Good-to-knows two

- Variables have automatic or dynamic lifetime
  - If you explicitly ask for memory, you are responsible (dynamic lifetime)
  - If you receive the memory automatically, you must not do anything

All automatically taken care of

```
float do_math(int number) {
	float rotation_matrix[4][4];
	if (number == 3) {
		rotation_matrix[2][0] = 0.0;
	} else {
		int k = number * 4;
		return (float)k;
	}
	return rotation_matrix[2][3];
}
```

The infamous pointer types in C

- Assume you already have some type t1

```
t1 *my_pointer;
char *name;


float **rotation_matrix;
float *other_matrix[4];
```

my_pointer holds the address of one t1.

name holds the address of one char

rotation_matrix holds the address of an address of a float

other_matrix is an array of size 4, each element is an address of a float

Getting a pointer

```
int funky_pointer(int number) {
      int *pointer = &number;
      int *other_pointer = pointer;
      *other_pointer = 0;
      return number;
}
```

No harm done

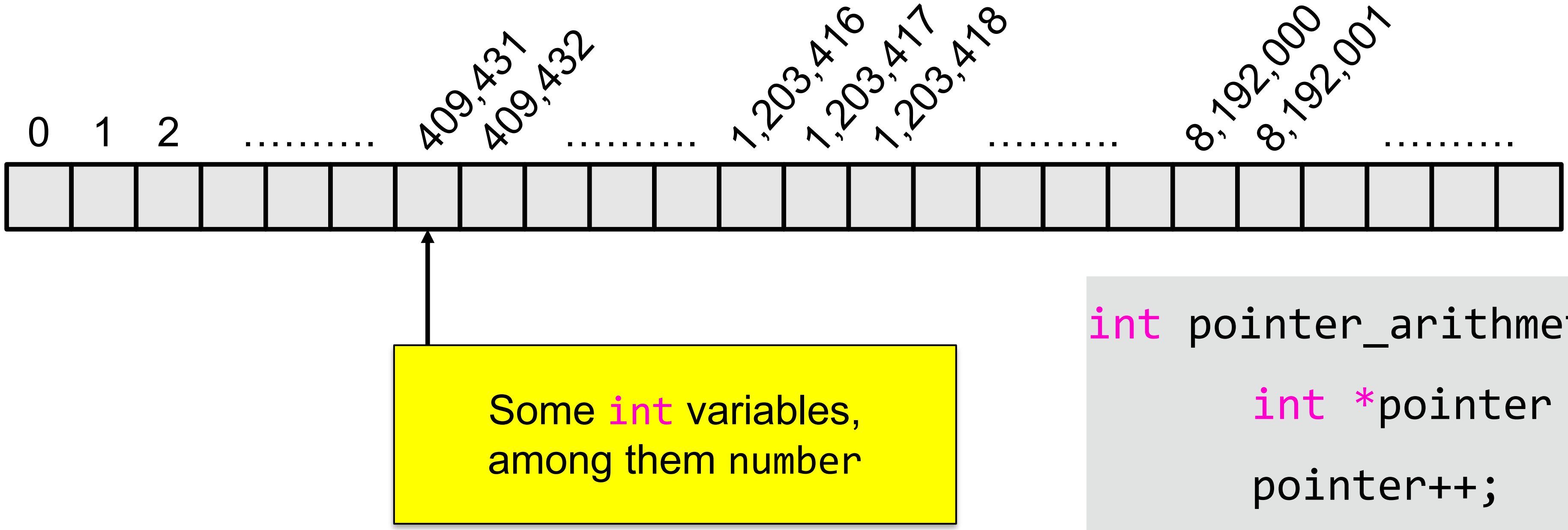
Crashing the program

```
int harmful_access(int number) {
      int *pointer = &number;
      int *other_pointer;
      *other_pointer = 0;
      return number;
}
```

other_pointer is not initialized.

We cannot follow an address that is not there.

Remember this?




```c
int pointer_arithmetic(int number) {
    int *pointer = &number;
    pointer++;
    *pointer = 0;
    return number;
}
```

“Fact”

Pointers are not harmful in and of themselves!
Using them to access addresses that do not hold data is

Dynamic memory management

- You might need a number of `float` that the user specifies
  - Cannot do so during compile time

- You can ask your operating system for a block of memory

```
void *malloc(size_t number_bytes);
```

- You are now responsible!

```
void free(void *ptr);
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Let’s waste resources

```c
void i_m_feeling_generous(int number) {
    float **pointers = malloc(sizeof(float*) * number);
    for (int i = 0; i < number; i++) {
        pointers[i] = malloc(sizeof(float) * 1024);
    }
}
```

Memory leak!

“Fact”

Pointers are not harmful in and of themselves!
Not returning allocated resources is

C workaround for strings

- C does not offer a built-in type for strings
  - Strings are really, really hard to get right and efficient
- Modeled as `char*` with a null-terminating character

h	e	l	l	o	_	s	t	u	d	e	n	t	s	!	\0	?	?	?	?	?	?	?	?

Counting the characters

```
size_t count_chars(
        const char *string) {
        size_t length = 0;
        while (*string != ‘\0’) {
                length++;
                string++;
        }
        return length;
}
```

```
size_t return_length(void) {
        const char *string =
                “hello students!”;
        return count_chars(string);
}
```

Towards compilation

- A C compiler reads a file from top to bottom
  - If it encounters a function it does not know, it throws an error

- A C compiler does not know any function

#include

```
#include “file.h”

#include <other_file.h>
```

#include takes the specified file and copies it as it exactly at the position

“…” means:
From the current directory

<…> means:
From the system directory

Common library functions

- <stdlib.h>
  - `malloc`, `free`
  - `system`
  - `abs`, `div`, `rand`
- <string.h>
  - `memcpy`, `memcmp`
  - `strcpy`, `strlen`
- <stdio.h>
  - `printf`, `scanf`

Doubly-recursive functions

```
void func1(void) {
	...
	func2();
	...
}

void func2(void) {
	...
	func1();
	...
}
```

```
void func2(void) {
	...
	func1();
	...
}

void func1(void) {
	...
	func2();
	...
}
```

Forward-declaring functions

```
void func1(void);
void func2(void);

void func1(void) {
        ...
        func2();
        ...
}
```

Now, the compiler knows of those two functions

The compiler might never encounter the function

Further towards compilation

- A C compiler reads a file from top to bottom
  - If it encounters a function multiple times, it throws an error
  - Cannot distinguish based on arguments

```
#pragma once


...
...
...
```

```
#ifndef KEY
#define KEY


...
...
...


#endif
```

File naming convention

- my_source.h
  - “Header”
  - Contains function declarations
  - Contains type definitions
- my_source.c
  - “Compilation unit”
  - Includes headers
  - Contains function definition

Compilation workflow

1. The preprocessor resolves its directives

   `#include`, `#define`, `#ifdef`, …

2. The compiler compiles each compilation unit

   Some functions might not be resolved

3. The linker combines all compiled files

   Unresolved functions now result in linking errors

Some output

```
int printf(const char *format, ...);
```

- Takes a pointer to constant characters (i.e., does not change them)
- Takes arbitrarily many arguments
- Returns the number of written characters (negative if failure)

```
char c = ‘h’; int i = 304; char *string = “ students!\n”;
printf(“%cello %i%s”, c, i, string);
```

- More details: https://www.tutorialspoint.com/c_standard_library/c_function_printf.htm

Some input

```c
int scanf(const char *format, ...);
```

- Takes a pointer to a string specifying the types to read
- Takes arbitrarily many arguments
- Returns the number of successfully read arguments

```c
char c; int i; char string[20];
scanf("%c%d%19s", &c, &i, string);
```

- More details: https://www.tutorialspoint.com/c_standard_library/c_function_scanf.htm

Some smaller topics

© Alex Becker

The main function

```c
int main(int argc, char **argv) {
        …
}
```

Your program starts at the `main` function

`argc` contains the number of arguments passed to the program

`argv` is a pointer to the arguments as `char*`

The first argument is always the program name

The main function

```
int main(int argc, char **argv) {
        for(int i = 0; i < argc; i++){
                printf("Arg %d: %s\n", i, argv[i]);
        }
        return 0;
}
```

```
./a.out Hello World!

Arg 0: ./a.out
Arg 1: Hello
Arg 2: World!
```

Your program starts at the `main` function

`argc` contains the number of arguments passed to the program

`argv` is a pointer to the arguments as `char*`

The first argument is always the program name

Special characters in strings

Character	Description
\n	newline
\t	horizontal tab
\v	vertical tab
\\	backslash
\b	backspace
\’	single quote
\”	double quote

Expressions with mixed types

- The shorter type is converted to the longer type
- Integers are converted to floating points
- Assignments convert to the target type
- If an automatic conversion is impossible, there is an error

```
float nooo(long num, short den) {
        float quotient = num / den;
        return quotient;
}
```

Missing loops

- There are also `while () {}` and `do {} while ();`
- There are `break` and `continue`

Switching on values

```
float nooo(long number) {
        int i = 0;
        switch (number) {
                case 0: case 1:
                        i++;
                case 2:
                        i--; break;
                default:
                        return 2;
        }
        return i;
}
```

Can only target constant integer values

Pay attention for fall-throughs!

Return values:

nooo(0) = 0.0
nooo(1) = 0.0
nooo(2) = -1.0
else: 2.0

Copying dynamically-managed variables

```
float *failed_copy(long number) {
        float *values = malloc(sizeof(float) * number);
        float *copy = values;
        free(values);
        return copy;
}
```

Shallow copy for everything

Array-to-pointer decay

```
size_t get_elements(float *vals){
        return sizeof(vals) / sizeof(vals[0]);
}

void decay(float *vals);

int main() {
        float values[3] = {1.0, 2.0, 4.1};
        decay(values);
        return 0;
}
```

Arrays decay to pointers

Type shenanigans

```
float q_rsqrt(float number) {
        float x2 = number * 0.5F;
        float y = number;
        long i = * (long *) &y;
        i  = 0x5f3759df - ( i >> 1 );
        y = * (float *) &i;
        y = y * (1.5F – (x2 * y * y);
        return y;
}
```

Can cast pointer arbitrarily

Different levels of const

```
int main() {
        int value = 0;
        const int k = value;
        const int *pointer1 = &k;
        int * const pointer2 = &value;
        const int * const pointer3 = &k;
        return k;
}
```

Cannot change k

Can change pointer1, cannot change *pointer1

Cannot change pointer2, can change *pointer2

Cannot change pointer3, cannot change *pointer3

Defining type aliases

```
typedef int PARPROG_INTEGER;

struct point {
        float x;
        float y;
}

typedef struct point Point;
```

A case study

Implementing a linked list

© Alex Becker

Our list nodes

```
typedef struct node {
        struct node *next;
        void *data;
} Node;
```

Creating a new list

```
Node *linked_list_create(void *first_data) {
        Node *list = malloc(sizeof(Node));
        list->next = 0;
        list->data = first_data;
        return list;
}
```

-> to access members of the type pointed to

a->b is equivalent to (*a).b

Appending values

```
Node *append_int(Node *list, int value) {
        int *ptr = malloc(sizeof(int));
        *ptr = value;
        Node *new_node = malloc(sizeof(Node));
        new_node->data = ptr; new_node->next = 0;

        if (list == 0) return new_node;
        Node *curr = list
        for (; curr->next != 0; curr = curr->next) { }
        curr->next = new_node;
        return list;
}
```

Counting nodes

```
int count_nodes(Node *list) {
	int number_nodes = 0;
	while (list != 0) {
		number_nodes++;
		list = list->next;
	}
	return number_nodes;
}
```

Freeing a list

```
void clear_node(Node *node) {
        free(node->data);
        free(node);
}

void clear_list(Node *list) {
        while (list != 0) {
                Node *copy = list->next;
                clear_node(list);
                list = copy;
        }
}
```