TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Reminder

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Moodle course

- If you are not enrolled in moodle, write an e-mail to Lukas
  - lukas.rothenberger@tu-darmstadt.de
  - https://moodle.tu-darmstadt.de/course/view.php?id=40840

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Hand-ins

- Choose a group in moodle
- Put your matriculation number into moodle
- Hand in the signed forms for Lichtenberg
  - Not strictly required for course, but exercises will be graded on the system
  - Offers possibility of CUDA-compatible graphics card and many threads

# Exercises

- Theoretical, in moodle:
  - On your own
  - Covers all material up to that week
- Practical labs:
  - In groups of three
- Oral tests (Testate):
  - On your own
  - Tutors might offer slots for groups nevertheless

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Consultation hours

- Find the consultation hours of the tutors in moodle:
  - Allgemeines > Sprechstunden
- Tutors can change location during the semester

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Programming in C++

# C++ is the upgrade

- Bjarne Stroustrup started in 1979 to extend C with classes
- Standardized in 1998 and 2003
- Since 2011 standardized every three years
  - Most recent standard: C++23
  - C++26 is a work in progress

# Where to look

https://en.cppreference.com

# Features of C++

- You cannot solve a complex problem with simple tools
- We will highlight multiple interesting features
  - Most advanced or dangerous topics avoided

# Philosophy

- "Zero-overhead principle"
  - You do not have to pay for what you do not use
  - If you were to program a feature by hand, you would not achieve better

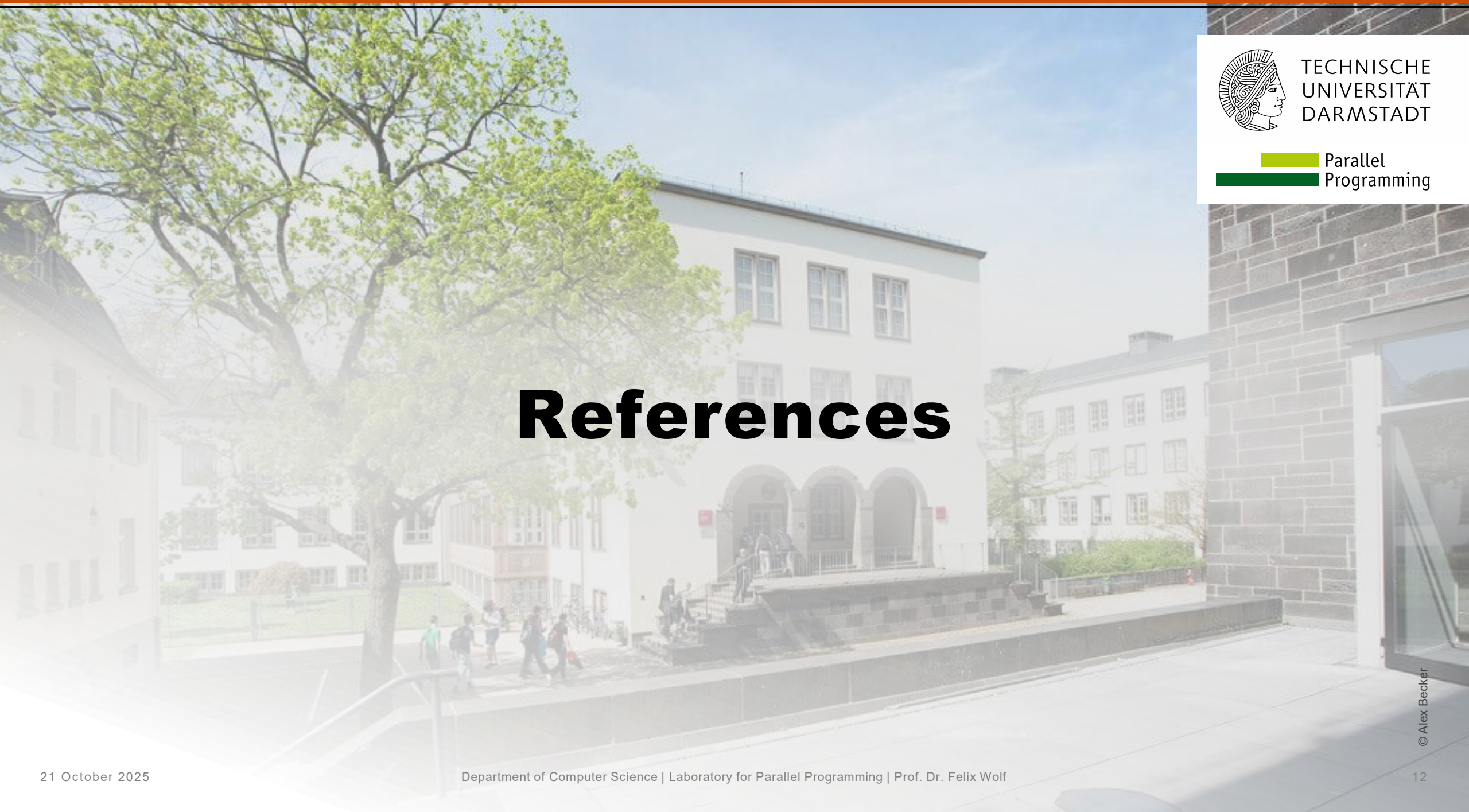
- A 'large' standard
  - What is impossible to achieve on your own
  - What is hard to get right
  - What is popular

# Warning

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

We do not follow (our own) code-style guidelines!



TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# References

# The drawbacks of pointers

- Pointers can be uninitialized
- Pointers can point to garbage (incremented too many times)
- Pointers allow type shenanigans

# The idea of a refence

- Have different names for the same memory cell
  - “Lukas”
  - “Herr Rothenberger”
  - “You” (furiously point finger)
- Think of it as a tool/way to access the memory cell

# Let's start simple

```
int return_two(int number) {
        int& name = number;
        name = 2;
        return number;
}
```

`type&` is the reference for `type`

`name` now refers to `number`

`name` does not have an own memory location!

# References preserve const

```
int return_two_failure(int number) {
        const int& name = number;
        name = 2;
        int const& nickname = number;
        nickname = 2;
        return number;
}
```

`const int&` and `int const&` are similar

Cannot change the value via `name` or `nickname`

# The benefits of references

- They are always initialized
- Under the hood, they are an address
  - No hefty copy

# Throws

# The drawbacks or return values

- When we encounter a bad state in C, we need to propagate it via the return value of the function
- You always had to check for the return values
  - Obfuscated actual meaning of function

# The idea of exceptions

- For exceptional behavior, we should not clutter the control path
- Distinguish return value from error code

# Let's start simple

```
std::uint16_t convert(int number) {
        if (number < 0) {
                throw -1;
        }
        if (number > 65535) {
                throw "Too high";
        }

        return std::uint16_t{number};
}
```

We can throw anything

# catch the culprit

```
std::uint16_t maybe() {
        try {
                return convert(-12);
        } catch (const char *c) {
                return 12;
        } catch (int& i) {
                return 13;
        }
}
```

If an expression in a try block throws, search for the suitable catch

We can catch anything

If there is no suitable catch, we keep throw-ing

There is also finally

# The benefit of throwing

- If it is not your responsibility, you can ignore it
  - System out of memory?
  - Passed illegal argument?

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Templates

# The drawback of `void*`

- In C, we were not able to handle different types in the same function
  - We can take an argument as `void*`, but then we had to cast it manually
  - Some performance issues with `void*`

# The idea of a template

- We provide a prototype for a function (later class) with some ‘blank spaces’ to be filled in later
- We specify what we need from the blank spaces

# Let's start simple

```
template<typename _Ty>
std::size_t get_size(_Ty object) {
        std::size_t s = sizeof(object)
        return s;
}
```

We provide this function for every type

The compiler will generate an instance for each invocation

# Calling templated functions

```
void print_sizes() {
      std::size_t s1 = get_size<int>(32);
      std::size_t s2 = get_size<std::int64_t>(32);
      std::size_t s3 = get_size(32);

      ... (printing comes later)
}
```

# Restrictions on templates

```
template<typename _Ty>
_Ty increment_to_one() {
        _Ty zero = _Ty(0);
        zero++;
        return zero;
}
```

We require that there is a constructor of _Ty taking an int

We require that we can call ++ on an object of type _Ty

We could also call any method imaginable

# Value templates

- Our ‘blank spaces’ were types up until now
- But there is not much hindering as to template on values
  - Other than rounding

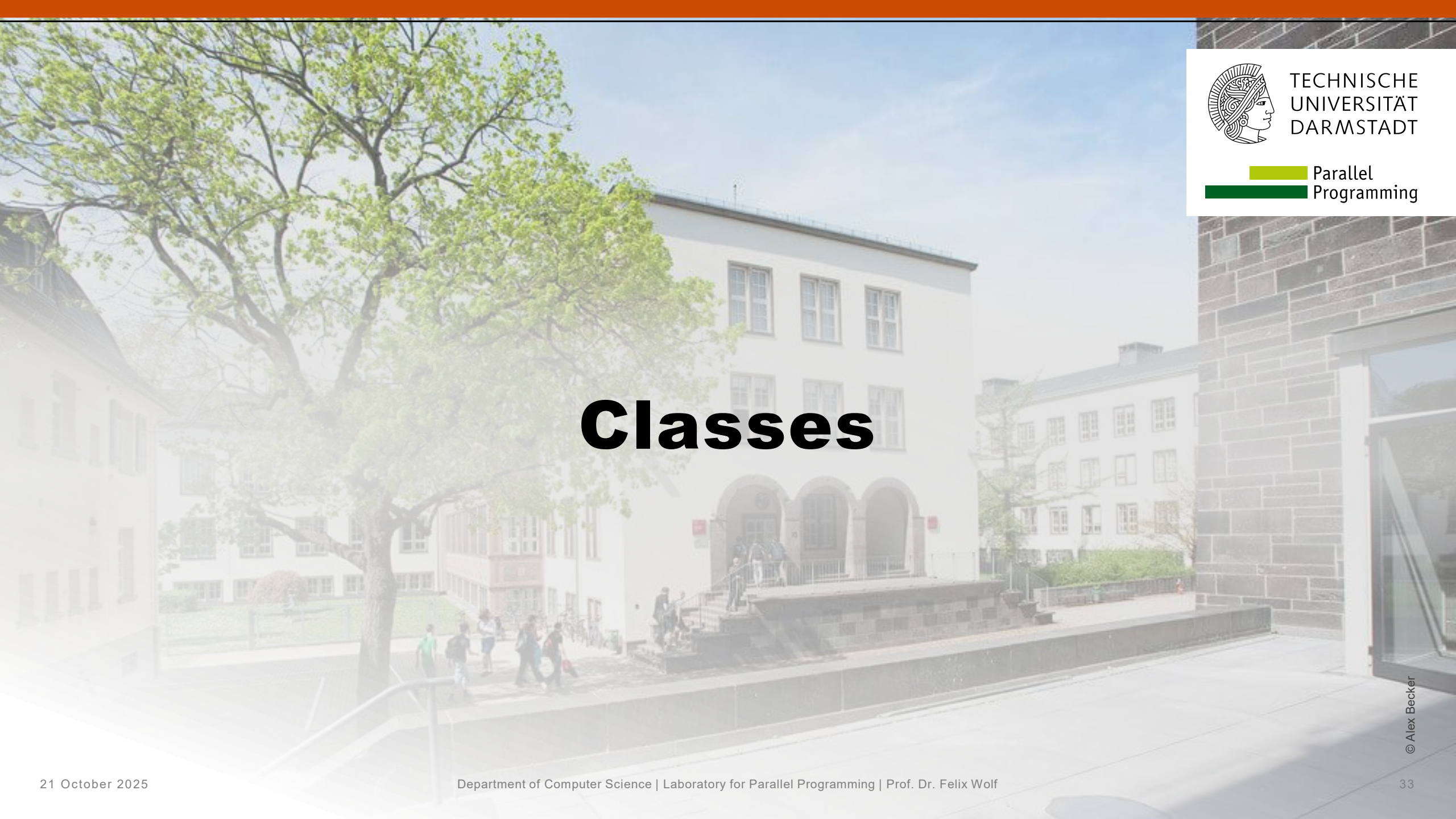
# Taking a value

```
template<int _val, typename _Ty>
int increment(_Ty value) {
        value += _val;
        return value;
}
```

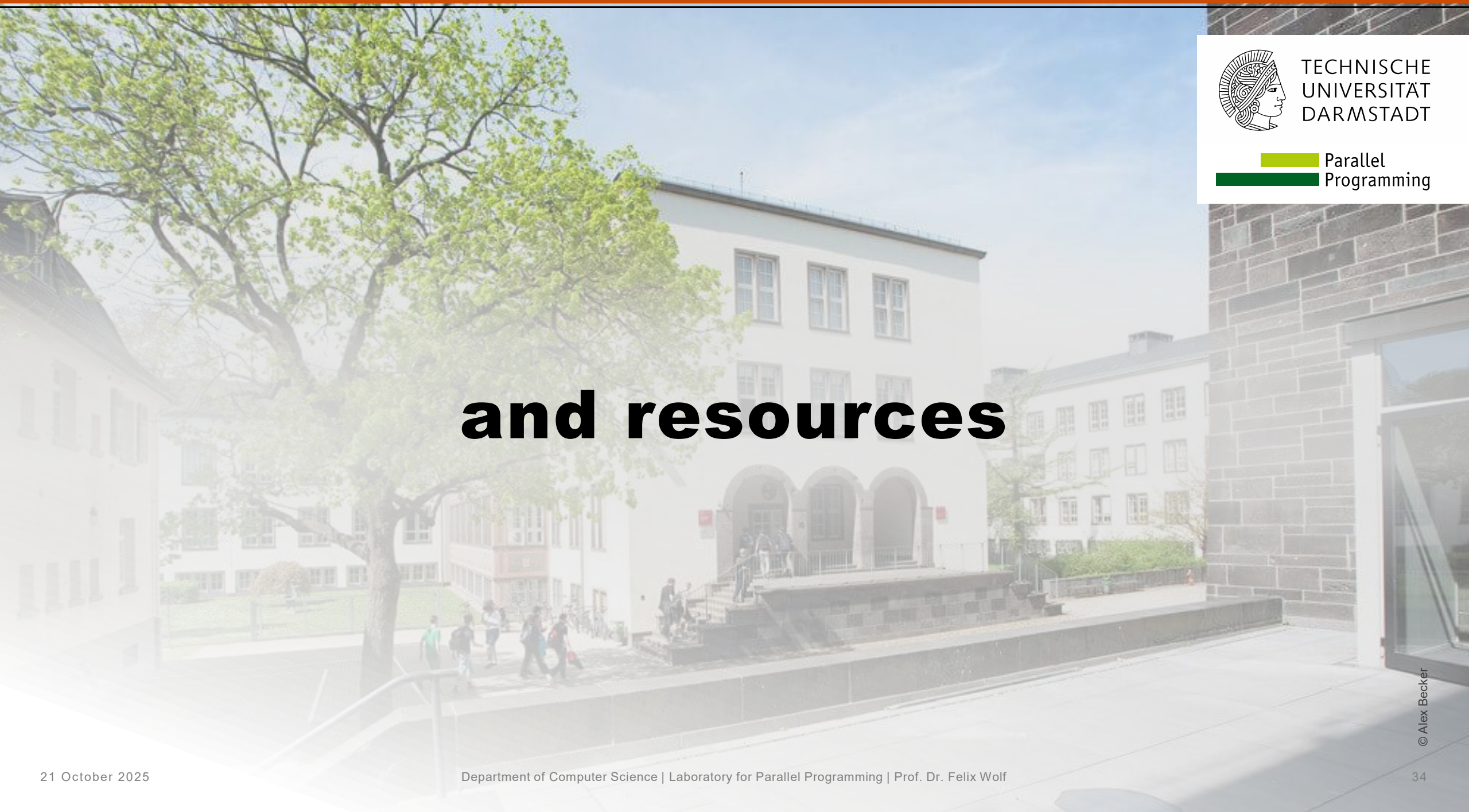
More work during compile time

# The benefits of templates

- Now, we can write containers, adaptable algorithms, etc.
- We catch type errors during compile time



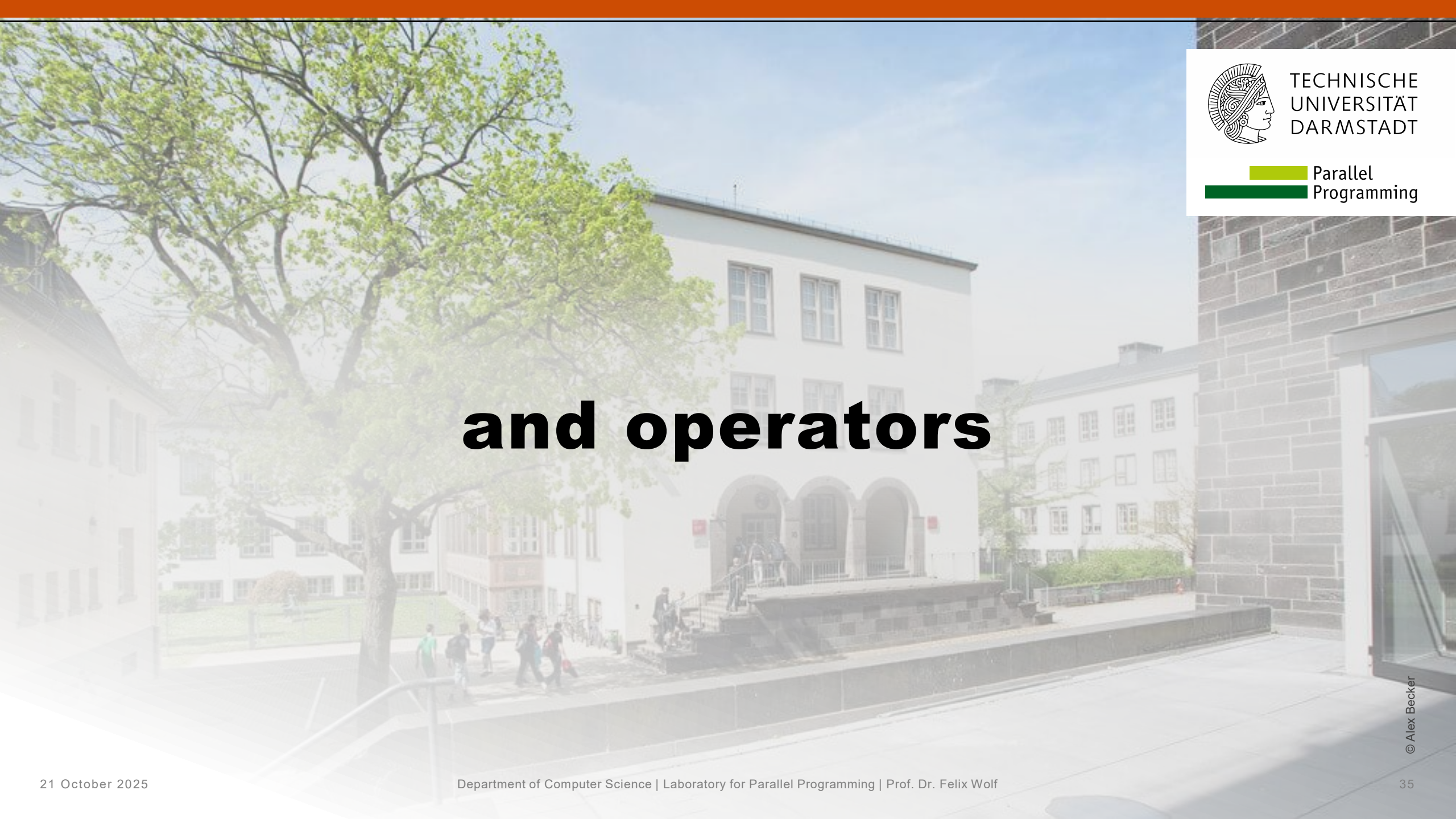
# Classes



# and resources

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming



# and operators

# The drawbacks of structures

- Disconnect between data and functionality
- Dependencies between data not expressible
- No customization of operators

# The drawbacks of resources

- You always have to keep book about the resources you opened
- What's the convention if you call other libraries?

# The idea of classes

- Group data and functionality together
- Hide unnecessary information from others
- (Allow polymorphism, dynamic dispatch, etc.)
- Have fancy syntax

# Let's start 'simple'

```
template<typename _Ty1, typename _Ty2>
class pair {
_Ty1 val1;
_Ty2 val2;

public:
pair(const _Ty1& v1, const _Ty2& v2)
      : val1{ v1 }, val2{ v2 }
      { }
};
```

val1 and val2 are private members

Note the member variable initialization

# Remember this class

```
template<typename _Ty1, typename _Ty2>
class pair {
_Ty1 val1;
_Ty2 val2;

public:
pair(const _Ty1& v1, const _Ty2& v2)
      : val1{ v1 }, val2{ v2 }
      { }
};
```

# Declaring methods

```
...
_Ty1& get_first_ref() {
        return val1;
}

const _Ty1& get_first_ref() const {
        return val1;
}
...
```

The compiler chooses the method based on the object

Differing implementations possible

# Fun with operators

- In C++, you can define operators for classes
  - Well known: `=`, `==`, `<=`, `++`, `--`, `[]`
  - Curious: casting, `&`, `(…)`, `,`

# Copying

- What must I do to copy an object of my class?
- The default way is to recursively copy member-wise
  - C-types are copied trivially

# Copying resources

```
template<typename _Ty1, std::size_t _si>
class my_array {
_Ty1 *pointer;

public:
my_array() : pointer{ new _Ty1[_si] }
        { }
};
```

new is C++'s way of malloc – it is typed!

A trivial copy fails

# Let's fail

```
int main(int argc, char **argv) {
        my_array<int, 10> arr{};
        my_array<int, 10> copy = arr;
        if (argc < 2) {
                my_array<int, 10> other{};
                copy = other;
        }

        return 3;
}
```

We copied shallowly

We should ask in the forum!

# Remember this class

```
template<typename _Ty, std::size_t _si>
class my_array {
_Ty *pointer;

public:
my_array() : pointer{ new _Ty[_si] }
       { }
};
```

# Let's construct successfully

```
...
my_array(const my_array& other)
        : pointer{ new _Ty1[_si] } {
        for (std::size_t i = 0; i < _si; i++) {
                pointer[i] = other.pointer[i];
        }
}
...
```

We take `other` by constant reference

If we were to take it by value, we would need to construct `other` first…

# State of the union

```
int main(int argc, char **argv) {
        my_array<int, 10> arr{};
        my_array<int, 10> copy = arr;
        if (argc < 2) {
                my_array<int, 10> other{};
                copy = other;
        }

        return 2;
}
```

We have correctly constructed copy

# Let's copy successfully

```
...
my_array& operator=(const my_array& other) {
        for (std::size_t i = 0; i < _si; i++) {
                pointer[i] = other.pointer[i];
        }
        return *this;
}
...
```

`this` is always a pointer to the class

`*this` is a reference

Values are a lie

# State of the union

```
int main(int argc, char **argv) {
	my_array<int, 10> arr{};
	my_array<int, 10> copy = arr;
	if (argc < 2) {
		my_array<int, 10> other{};
		copy = other;
	}

	return 1;
}
```

We have correctly constructed copy

We have correctly assigned copy

# Returning memory

- Remember your responsibility from C?
  - If you ask for memory yourself, you are responsible
  - `malloc()` needed a call to `free()`
- We asked for memory with `new[]`
  - We now need to return it with `delete[]`
- Do not mix and match!
  - There is also the pair `new` and `delete`

# Let’s destruct successfully

```
...
~my_array() {
        delete[] pointer;
}
...
```

The runtime knows the size of the memory

# The destructor

- There can be many ways to construct an object
  - Different overloads of the constructor allowed
- There always can only be one way to destruct an object
  - No arguments
  - Called immediately when it goes out of scope
  - Clean up all you are responsible for

# State of the union

```
int main(int argc, char **argv) {
        my_array<int, 10> arr{};
        my_array<int, 10> copy = arr;
        if (argc < 2) {
                my_array<int, 10> other{};
                copy = other;
        }

        return 0;
}
```

We have correctly constructed copy

We have correctly assigned copy

We have correctly cleaned up

# Accessing our data

```
...
_Ty operator[](std::size_t index) {
        if (index >= _si) {
                throw std::exception{};
        }

        return pointer[index];
}
...
```

Some operators have fixed parameters, some not

C++23: Allows multiple `std::size_t`

Before: cheating possible

# The benefit of classes

- A whole new world opens up!
  - Automatic resource management
  - Great code quality

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Various smaller topics

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Standard containers

# The all 'rounder: `std::vector<_Ty>`



- Managed block of contiguous memory for objects of type _Ty
- Constant access time
- Amortized constant time for pushing elements at the end
- Constant time for deleting elements at the end

# The all ‘ugly: `std::list<_Ty>`

- Linked list of nodes containing one object each
- Linear access time
- Constant merging
- Nearly always slower than `std::vector<_Ty>`

# The fixer:
# `std::array<_Ty, _si>`

- You get exactly what you expect

# The bouncer:
# `std::queue<_Ty>`

- First in, first out
- Nothing more

# The dishwasher: `std::stack<_Ty>`

- First in, last out
- Nothing more

# The odd uncle: `std::deque<_Ty>`

- List of contiguous memory blocks
- Constant access time
- Constant push and pop at both ends
- No one really knows what's going on

# The matchmaker: `std::map<_Ke, _Val>`



- In other programming languages known as a dictionary
- Associative container mapping `_Ke` to `_Val`
  - Uses `operator[]`
- Logarithmic access time, stored as red–black tree

# The better matchmaker: `std::unordered_map<_Ke, _Val>`



- In other programming languages known as a hash map
- Amortized constant insert/access/removal time

# The savior: `std::string`

- Yeah, there is a string type in C++
- It offers a lot of functionality

# Iterating a container

- Most containers offer:
  - `begin()`, `begin() const`, `cbegin() const`
  - `end()`, `end() const`, `cend() const`
- They return a (constant) iterator
  - Proxy object to access the container
  - Point to the first/one-after-last element
  - Can increment iterator with ++
  - Can retrieve element with *

# Let's start 'simple'

```
template<typename _Contain>
double sum(const _Contain& container) {
        auto temp = 0.0;
        for (auto it = container.begin();
                it != container.end(); ++it) {
                temp += *it;
        }

        return temp;
}
```

auto means:
Compiler, do your thing!

We require a lot of _Contain

# `auto` in action

- `auto` matches greedily any type
  - Might want to use `const auto` or `const auto&`
- `auto` is necessary for types that cannot be named

# Let's get complex

```
template<typename _Ke>
double sum(const std::map<_Ke, double>& m) {
        auto temp = 0.0;
        for (const auto& [_, val] : m) {
                temp += val;
        }

        return temp;
}
```

`for (type name : c)` is C++'s foreach

`auto [a, b]` 'matches' trivial pairs

`auto [a, b, c]` 'matches' trivial triples

# The benefits of standard container



- They are really hard to get right!
- Tuned by many smart minds
  - Small-vector optimization for some libraries
  - Exception-safe

# I/O

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Template + I/O

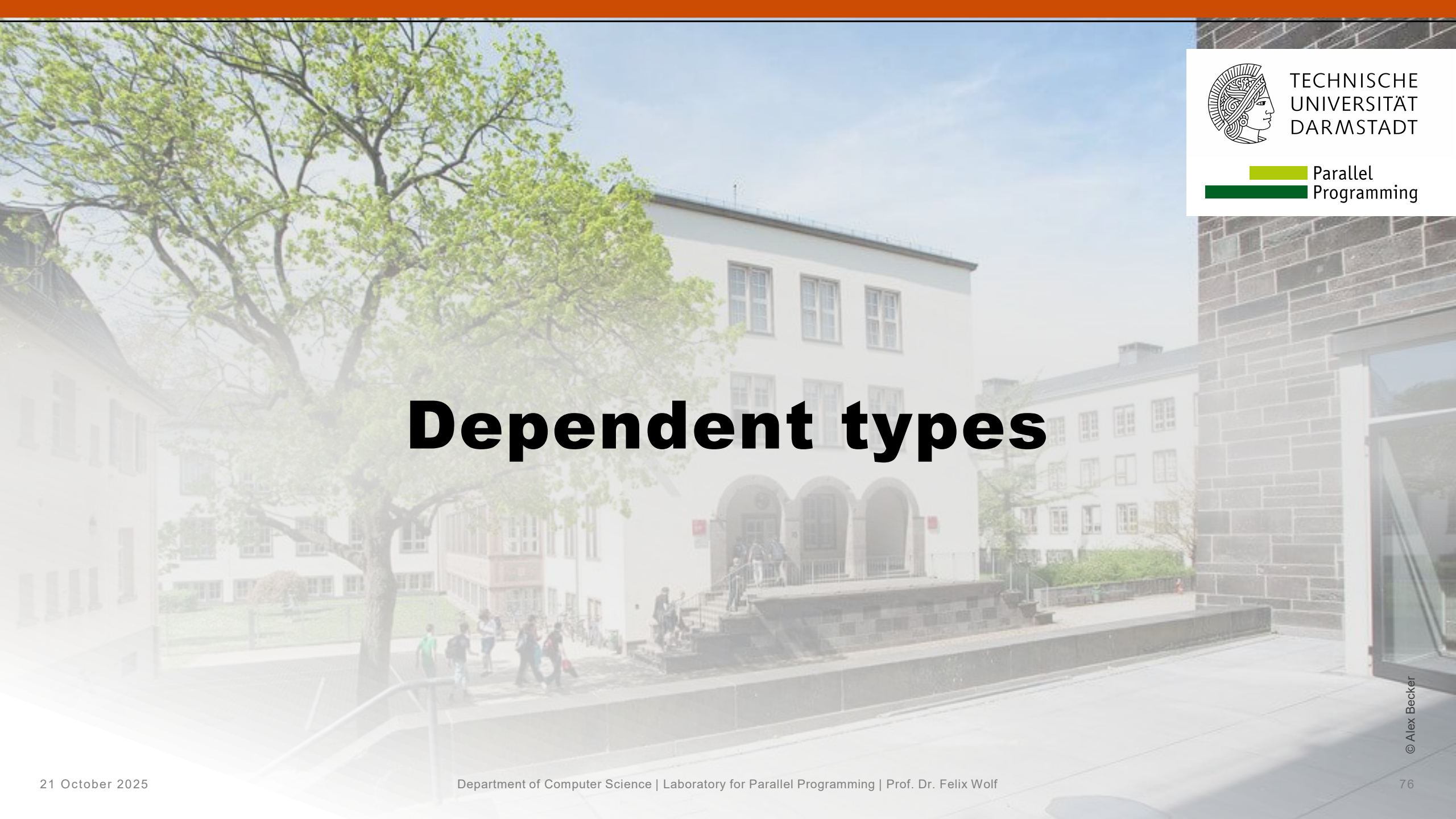
- Methods like `printf` and `scanf` are kind-of incompatible with templates
  - What specifier would you insert into the formatting string?

# C++'s I/O

```
template<typename _Ty>
void print(const _Ty& val) {
      std::cout << val << '\n';
      _Ty copy {};
      std::cin >> copy;
      std::cout << (val == copy) << std::endl;
}
```

<< and >> shove data into our out of the stream

Can overload `std::cout` and `std::cin` for custom types



TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Dependent types

# What is necessary?

- C++ goes hard when it comes to types
  - You have to convert between types all the time
  - In C, all had the same type (and you had to care for the values)

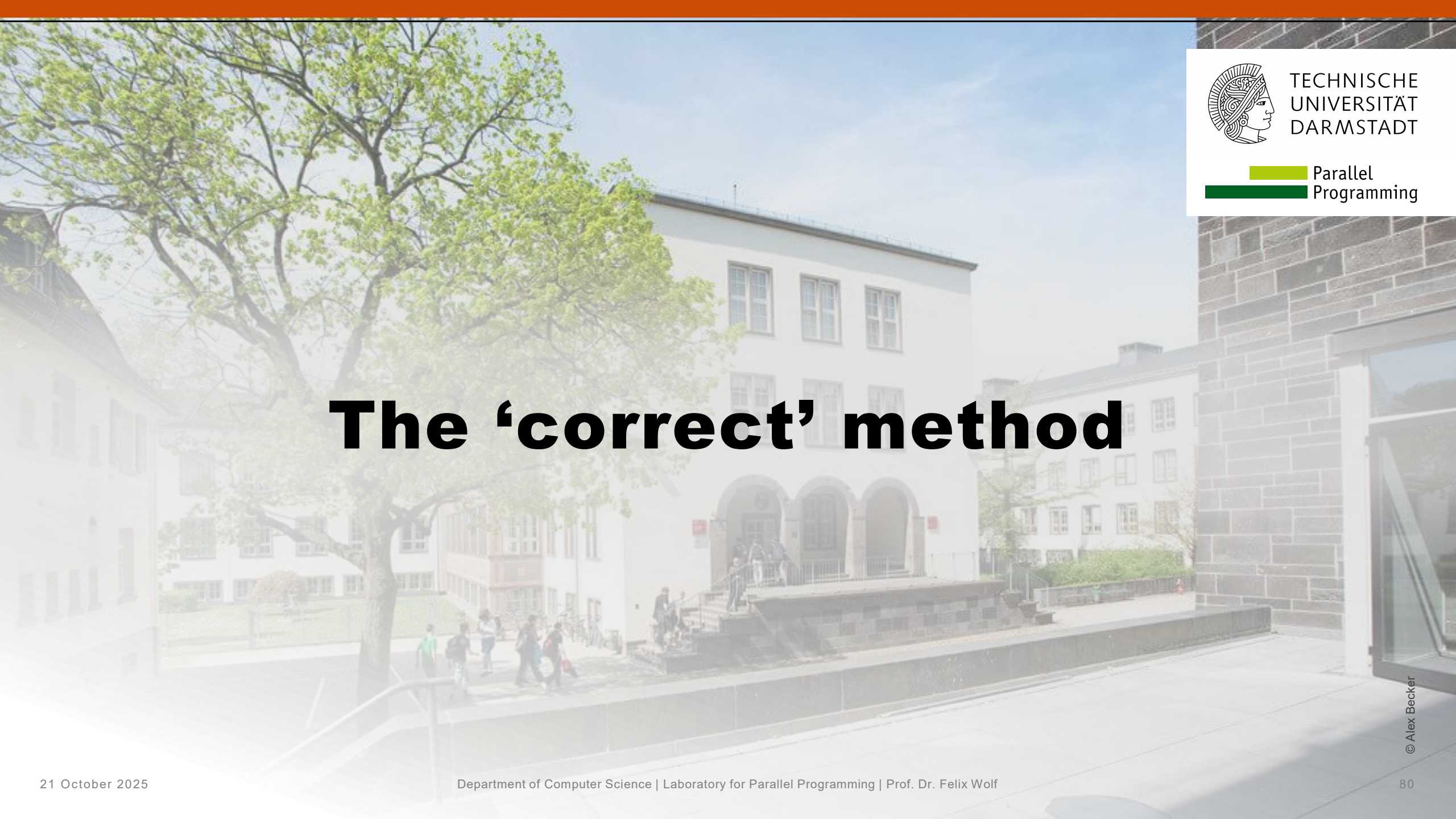
# std::vector<T>

## Member types

| Member type | Definition |
|---|---|
| value_type | T |
| allocator_type | Allocator |
| size_type | Unsigned integer type (usually std::size_t) |
| difference_type | Signed integer type (usually std::ptrdiff_t) |
| reference | value_type& |
| const_reference | const value_type& |
| pointer | Allocator::pointer (until C++11)<br>std::allocator_traits<Allocator>::pointer (since C++11) |
| const_pointer | Allocator::const_pointer (until C++11)<br>std::allocator_traits<Allocator>::const_pointer (since C++11) |
| iterator | *LegacyRandomAccessIterator* and *LegacyContiguousIterator* to value_type (until C++20)<br>*LegacyRandomAccessIterator*, contiguous_iterator, and *ConstexprIterator* to value_type (since C++20) |
| const_iterator | *LegacyRandomAccessIterator* and *LegacyContiguousIterator* to const value_type (until C++20)<br>*LegacyRandomAccessIterator*, contiguous_iterator, and *ConstexprIterator* to const value_type (since C++20) |
| reverse_iterator | std::reverse_iterator<iterator> |
| const_reverse_iterator | std::reverse_iterator<const_iterator> |

# The index types

- All indexed containers offer `::size_type`
  - Usually `std::size_t`
  - `std::size_t` is large enough to index any contiguous memory
  - `std::vector<HUGE_TYPE>::size_type` can be smaller



# The ‘correct’ method

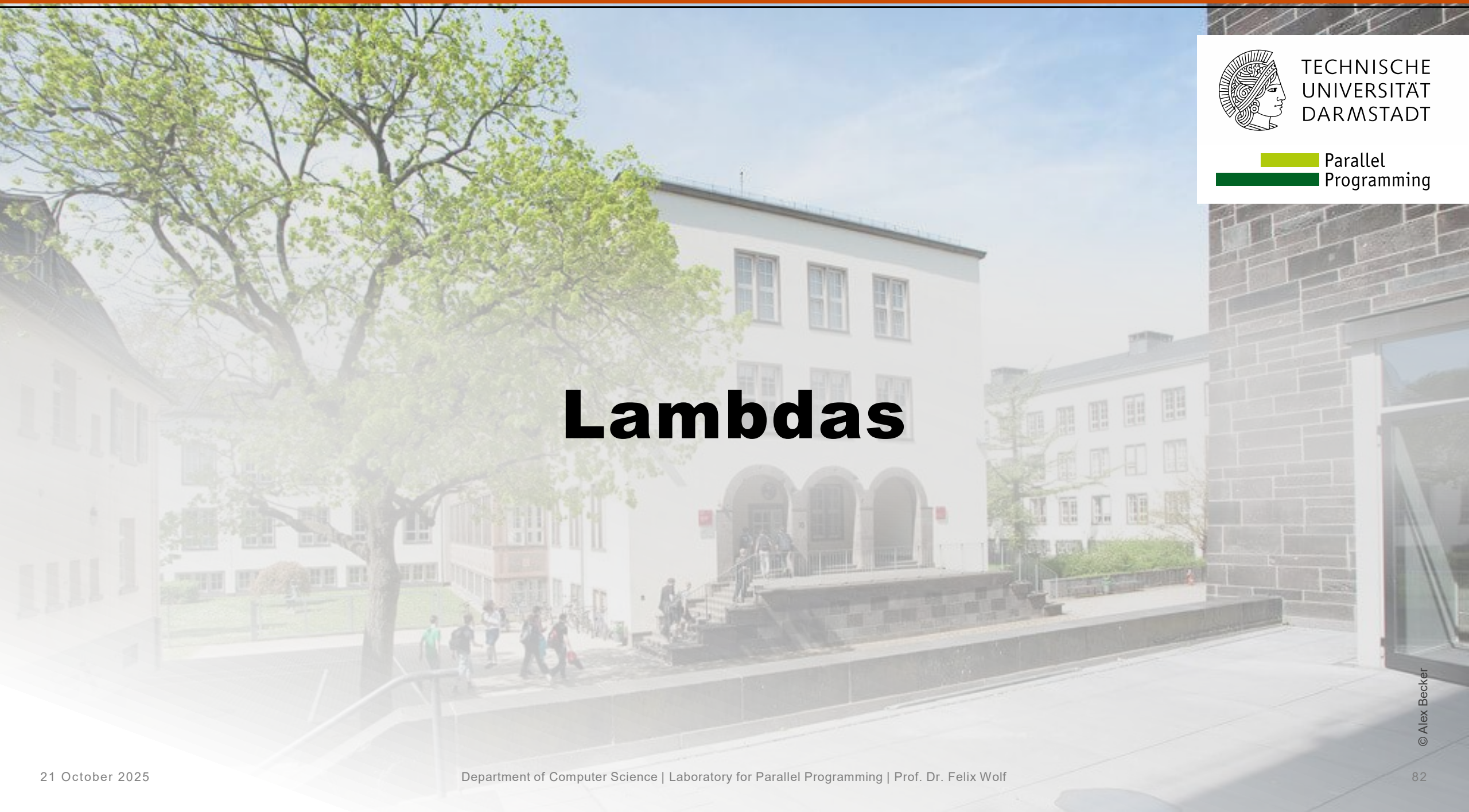
# C++ is opt-in

```
class holder {
public:
[[nodiscard]] constexpr double get_value(
      const int argument) const noexcept {
            return 0.0;
      }
}
```

[[nodiscard]]: Warning if value not saved when called

constexpr: Can be evaluated during compile time

noexcept: Does not throw any value



TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

# Lambdas

# Purpose of lambdas

- Sometimes, you need a small function
  - Not necessarily a name
  - Usually not passed around

# Let's start simple

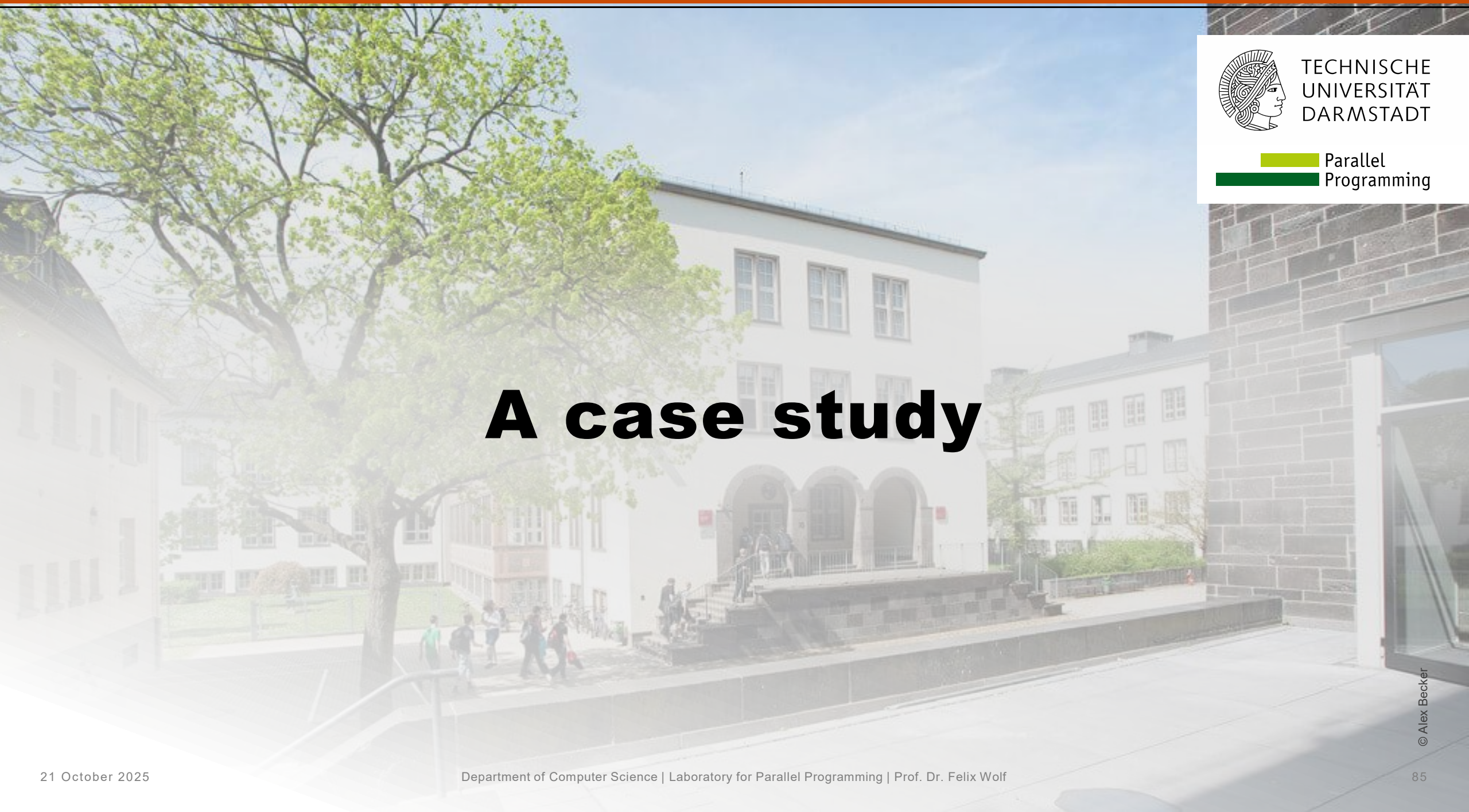
```
int main() {
      const std::vector<int> v {1, 2, 3, 4, 5};
      const int mod_value = 3;

      const auto pos = std::find_if(
            v.begin(), v.end(),
            [mod_value] (const int x)
            {return x % mod_value == 0;});
      return pos != v.end();
}x
```

[...] contains the values from outside that are made available. & possible

(...) contains the arguments

{...} contains the instructions



# A case study

# Our setting

- We have a directed graph G = (V, A) whose nodes have some position in Euclidean space
- We want to calculate the average Euclidean length of the arcs
- The graph and the calculation are distributed with MPI
  - As MPI is not part of this year's syllabus, we accept the functions as black boxes

# The class for positions : 1

```
template <typename T>
    requires std::is_arithmetic_v<T>
class Vec3 {
    T x{0};
    T y{0};
    T z{0};

public:
    using value_type = T;

    constexpr Vec3() = default;
    constexpr ~Vec3() = default;
```

We do not accept any type, it must be arithmetic

We initialize the members to 0, no matter the constructor call

We provide the dependent type `value_type`

We want default construction/destruction of our class

# The class for positions : 2

```
constexpr explicit Vec3(const T& val) noexcept
    : x(val),
      y(val),
      z(val) {
}

constexpr Vec3(const T& _x, const T& _y, const T& _z)
    noexcept : x(_x), y(_y), z(_z) { }

constexpr Vec3(const Vec3<T>& other)
    = default;
constexpr Vec3<T>& operator=(const Vec3<T>& other)
    = default;
```

We allow construction by one or three values

We do not want implicit conversion, so we use `explicit`

We have no resources in the class, so we use `default`

# The class for positions : 3

```
[[nodiscard]] constexpr friend
    Vec3<T>  operator-
    (const Vec3<T>& lhs, const Vec3<T>& rhs)
    noexcept {
    return {
        lhs.x - rhs.x,
        lhs.y - rhs.y,
        lhs.z - rhs.z
    };
}
```

We want to subtract vectors, `friend` makes the operator access the private members

`return {...}` can often deduce the constructor

# The class for positions : 4

```
[[nodiscard]] double calculate_2_norm() const noexcept {
    return std::sqrt(calculate_squared_2_norm());
}

[[nodiscard]] constexpr double calculate_squared_2_norm()
    const noexcept {
    const auto xx = x * x;
    const auto yy = y * y;
    const auto zz = z * z;

    const auto sum = xx + yy + zz;
    return sum;
}
```

Sometimes, you want to store intermediate values for debuggability

Let the optimizer to the heavy-lifting

`std::sqrt` is not `constexpr`

# Calculate distance for one node

```
[[nodiscard]] static double compute_arc_length
(const DistributedGraph& graph, const node_id_type node_id) {
    const auto my_rank = MPIWrapper::get_my_rank();

    auto accumulated_distance = 0.0;
    const auto& node_position = graph.get_node_position(my_rank, node_id);
    const auto out_arcs = graph.get_out_arcs(my_rank, node_id);

    for (const auto& [target_rank, target_id, weight] : out_arcs) {
        const auto& position = graph.get_node_position(target_rank, target_id);

        const auto difference = position - node_position;
        const auto distance = difference.calculate_2_norm();

        accumulated_distance += distance;
    }

    return accumulated_distance;
}
```

`get_node_position()` returns `Vec3<double>`

`get_out_arcs()` returns a `std::span`

`std::span` functions as a view on contiguous data

# Calculating distance for all

```
[[nodiscard]] static double all_compute_arc_length(const DistributedGraph& graph) {
    const auto number_local_nodes = graph.get_number_local_nodes();

    auto accumulated_distance = 0.0;

    for (auto node_id = node_id_type{0}; node_id < number_local_nodes; ++node_id) {
        const auto distance = compute_arc_length(graph, node_id);
        accumulated_distance += distance;
    }

    const auto total_distance = MPIWrapper::all_reduce_sum(accumulated_distance);
    const auto number_total_arcs = OutArcCounter::all_count_out_arcs(graph);

    const auto& average_distance = total_distance / number_total_arcs;

    return average_distance;
}
```

`all_reduce_sum()` takes local values and returns the sum

`all_count_out_arcs()` returns the number of arcs in the graph

# Takeaways

- Let the compiler do the heavy lifting
- Name your functions/methods/variables with **common sense**
- Break your problem into suitable sub-problems