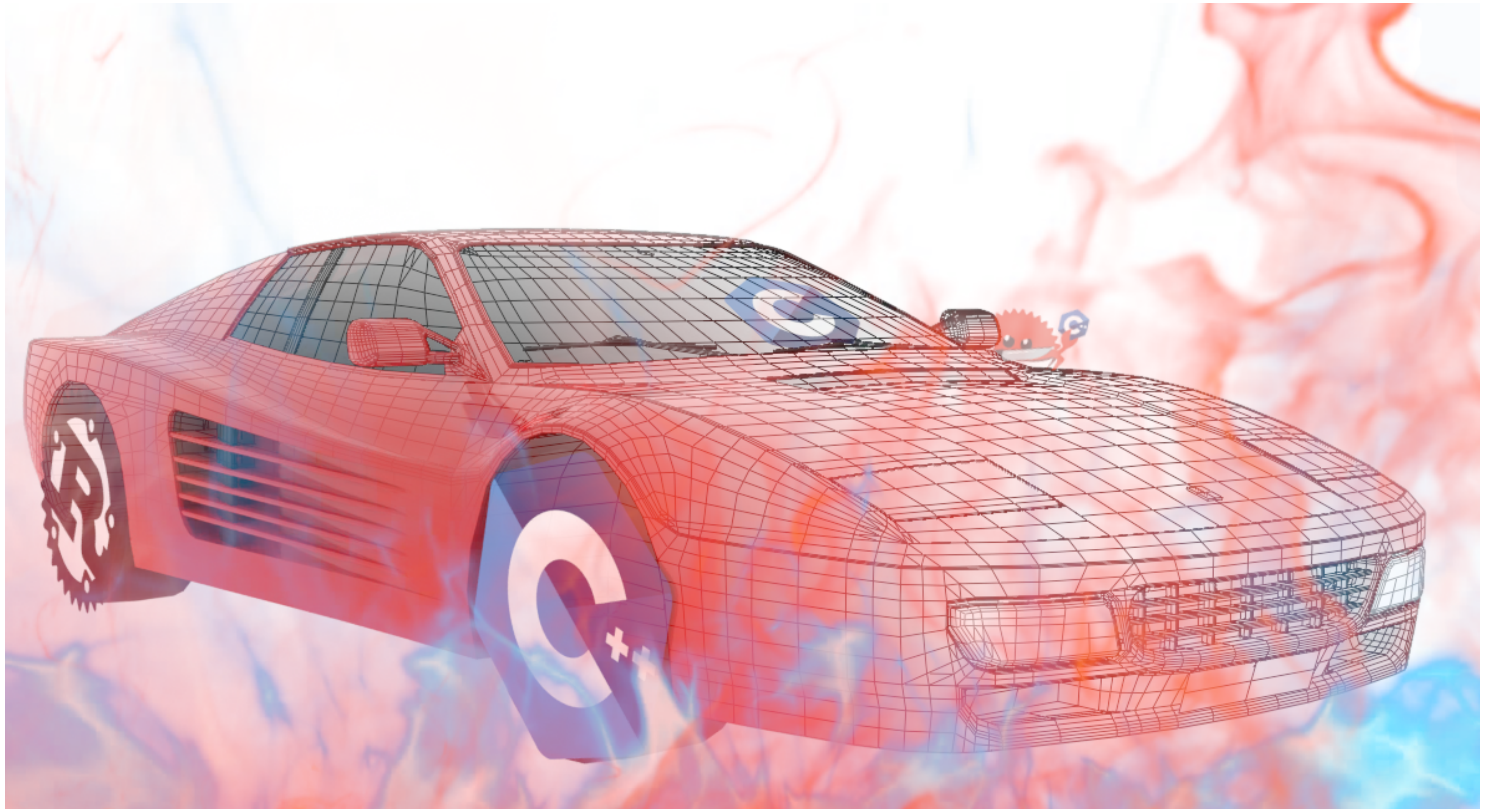
Effizientes Programmieren in C, C++ und Rust

C++ Templates

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Recap

1. What are the main benefits of exceptions?
2. What is the major difference between Java generics and C++ templates?

Table of Contents

Why we Need Templates

What are templates again?

- Code that is generic over different types
- "Parameterized" classes and functions
- Compiled into separate, specialized machine code for each variant (template instantiation/ monomorphization)

```
template<typename T>
class WithGenericData {
    // ...
    std::vector<T> _data;
};
```

What can we use templates for?

- Generic data structures and algorithms
- Zero-cost abstractions 💸
- Lifting computations to compile time
- Enhancing type safety
- Building extremely generic library interfaces

- **Black magic!** 🪄
  (aka the dark arts of template metaprogramming)

Why we Need Templates

Generic algorithms without templates

```
void qsort(void* ptr, size_t count, size_t size,
           int (*comp)(const void*, const void*));
```

- Data is behind a `void` pointer
- Each usage requires casting
- Function pointers for generic functionality
- Data structures: even the destructor of the data must be a function pointer

- Note: We **can** build vtable-like functionality atop function pointers

Problems?

- Not fun to use *(in my opinion at least)*
- No type safety
- Many indirections $\Rightarrow$ not efficient!

C Programmers

style="display: block; margin-left: 0px; margin-top: 0px; width: 60%; float: left;"/>

A Story about Graphs, Algorithms and Templates

Ralf Templatus, a mostly average C++ programmer, is hired by Dr. S. Fina* to work on a graph algorithm library.

- Should provide efficient data structures for different graph types
  - Currently only one basic graph implementation
- Support common algorithms for working on graphs

** what's your problem, these are completely normal names!*

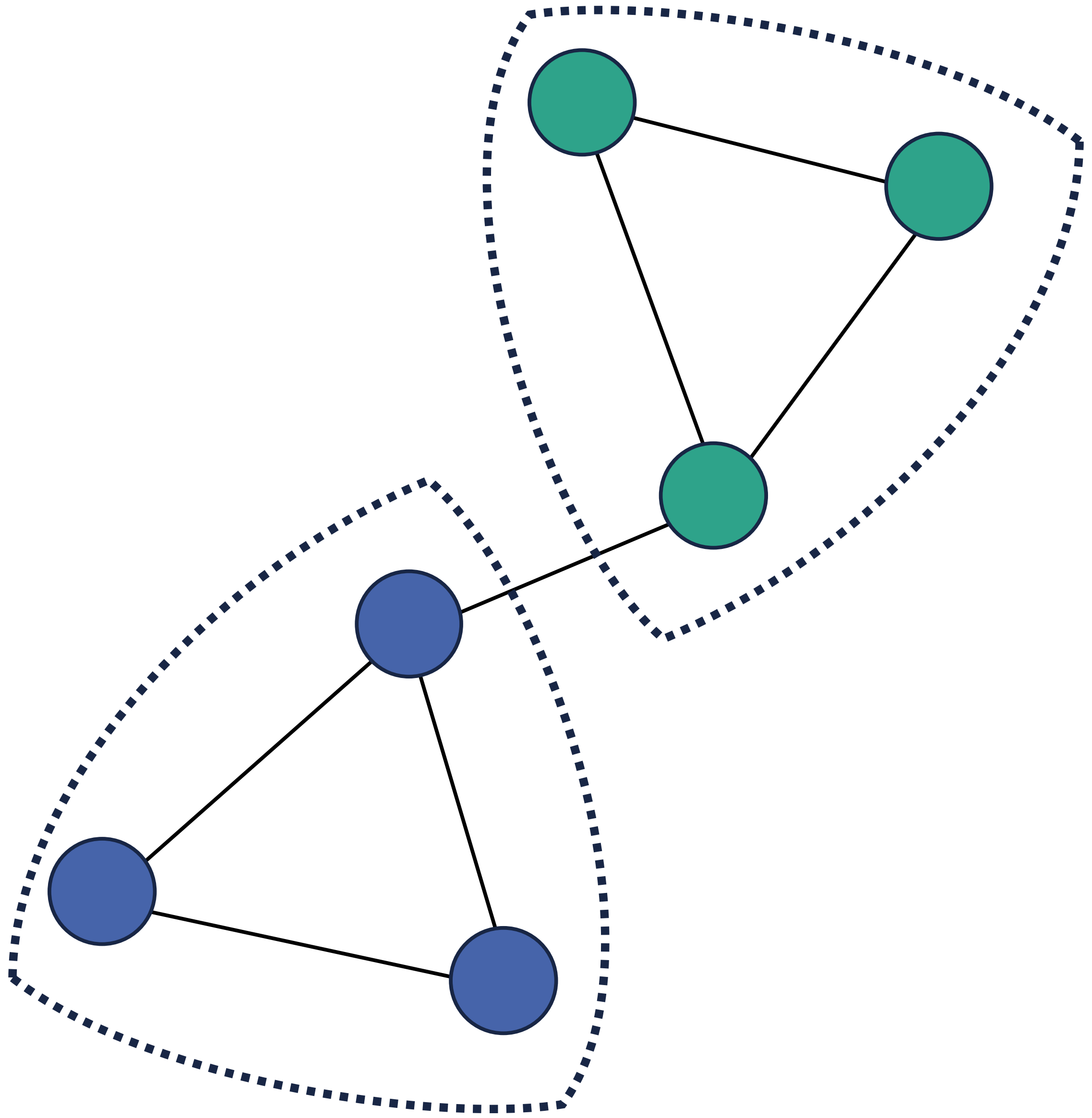
```
class Graph {
  public:
    NodeId num_nodes() const { /* ... */  }
    EdgeId num_edges() const { /* ... */  }
    int32_t node_weight(NodeId node_id) const
    { /* ... */ }
    int32_t edge_weight(EdgeId edge_id) const
    { /* ... */ }
    NodeView nodes() const { /* ... */  }
    EdgeView adjacent_edges(NodeId node_id) const
    {
        EdgeId l = _nodes[node_id].first_edge;
        EdgeId r = _nodes[node_id + 1].first_edge;
        auto it = _edges.cbegin();
        return EdgeView{it + l, it + r};
    }

  private:
    std::vector<NodeData> _nodes;
    std::vector<EdgeData> _edges;
};
```

A Story about Graphs, Algorithms and Templates

Ralf's first task is implementing a simple clustering algorithm.

With some help, he finds a simple and efficient solution.



```
void clustering_round(const Graph& graph,
            std::span<uint32_t> cluster_ids,
            ResettableArray<int32_t>& accumulator) {
    for (NodeId node_id: graph.nodes()) {
        accumulator.reset();
        uint32_t best_cluster = cluster_ids[node_id];
        int32_t best_rating = 0;
        // accumulate ratings for all neighbors
        for (auto [target, weight]:
             graph.adjacent_edges(node_id)) {
            auto& rating = accumulator.get_ref(
                                   cluster_ids[target]);
            rating += weight;
            if (rating > best_rating) {
                best_cluster = cluster_ids[target];
                best_rating = rating;
            }
        }
        cluster_ids[node_id] = best_cluster;
    }
}
```

A Story about Policies

> The implementation works well, but unfortunately it tends to create some overly heavy clusters.

- Typical approach for improving algorithms: test many different strategies
- Final version might support a selection of the best strategies
- Strategies? Did someone just say **strategy pattern**?

A Strategic Algorithm

- Goal: penalties for node weights

```
struct RatingPolicyInterface {
    virtual float get_rating(int32_t edge_weight,
        int32_t node_weight) const = 0;
};
```

```
struct WeightPenalty: public RatingPolicyInterface {
    float get_rating(int32_t edge_weight,
                     int32_t node_weight) const {
        return edge_weight /
            static_cast<float>(node_weight);
    };
};
```

```
struct NoPenalty: public RatingPolicyInterface {
    float get_rating(int32_t edge_weight,
                     int32_t) const {
        return edge_weight;
    };
};
```

A Story about Policies

Ralf comes up with two variants.*

```
void clustering_round_dynamic(const Graph& graph,
              std::span<uint32_t> cluster_ids,
              ResettableArray<int32_t>& accumulator,
              const RatingPolicyInterface& policy) {
    for (NodeId node_id: graph.nodes()) {
        /* ... */
        for (auto [target, weight]:
               graph.adjacent_edges(node_id)) {
              auto& rating = /* ... */;
/* ====> */ rating += policy.get_rating(weight,
                            graph.node_weight(target));
              /* ... */
        }
        cluster_ids[node_id] = best_cluster;
    }
}
```

```
struct RatingPolicyInterface {
    virtual float get_rating(int32_t edge_weight,
        int32_t node_weight) const = 0;
};
```

```
template<class RatingPolicy>
void clustering_round_template(const Graph& graph,
              std::span<uint32_t> cluster_ids,
              ResettableArray<int32_t>& accumulator)
    for (NodeId node_id: graph.nodes()) {
        /* ... */
        for (auto [target, weight]:
              graph.adjacent_edges(node_id)) {
              auto& rating = /* ... */;
/* ====> */ rating += RatingPolicy::get_rating(weig
                            graph.node_weight(target));
              /* ... */
        }
        cluster_ids[node_id] = best_cluster;
    }
}
```

WHICH ONE IS FASTER? 🚴

A Story about Policies

```
void clustering_round_dynamic(/* ... */
    const RatingPolicyInterface& policy);
```

```
template<class RatingPolicy>
void clustering_round_template(/* ... */);
```

Which one is faster?

- 5 rounds, 1k nodes, 8k edges
- Dynamic dispatch (both): 295 μs
- Templated (with penalty): 200 μs
- Templated (no penalty): 175 μs
- Similar for graphs of different size

... okay okay, no one is suprised

- Another data point: templated, but forbid inlining
  - With penalty: 300 μs
  - No penalty: 245 μs

But why is it faster?

- Assembly for dynamic vs. static dispatch:

```
mov     rcx, QWORD PTR [r12]   # read vtable-ptr
call    [QWORD PTR [rcx]]      # read and call function
```

```
call    NoPenalty::get_rating(int, int)  # direct call
```

- Actually, this is only a very minor difference (two hot reads, predictable)
- Dynamic dispatch also requires more shuffling between registers and stack (caller-saved registers)
- Largest difference: inlining!

The Policy Pattern

- Dynamic dispatch is generally less efficient then static dispatch
  - *Particularly notable in hot loops*
- Instead: policies as template parameters
  - *Basically a templated version of the strategy pattern*
- Dispatch to policies either via static functions (if the policy has no additional data) or a method on an instance (if data is required)
- Applicable to functions and classes
- Same efficiency as writing each combination by hand! 🚴

```
template<class RatingPolicy, class MaxWeightPolicy,
         class Accumulator>
void clustering_round_policized(const Graph& graph,
            std::span<uint32_t> cluster_ids,
            Accumulator& accumulator) {
    for (NodeId node_id: graph.nodes()) {
        /* ... */
        for (auto [target, weight]:
            graph.adjacent_edges(node_id)) {
            auto& rating = accumulator.get_ref(
                                cluster_ids[target]);
            rating += RatingPolicy::get_rating(weight,
                        graph.node_weight(target));
            if (MaxWeightPolicy::allows(
                total_cluster_weight)) {
                /* ... */
            }
        }
        cluster_ids[node_id] = best_cluster;
    }
}
```

A Story about Specialization

The next task is generalizing the graph class: the two-digit userbase has varying requirements for node and egde weights.

- Classical use case for templates: support different kinds of data
- C++20: Specify requirements for weight types via concepts ↑
  - *Trivially copyable? If not, we should probably return references instead of values*
- Note: algorithms now need to be templated over the graph type

```
template<typename NodeWeight, typename EdgeWeight>
class Graph {
  public:
    NodeId num_nodes() const { /* ... */  }
    EdgeId num_edges() const { /* ... */  }

    NodeWeight node_weight(NodeId node_id) const {
        return _nodes[node_id].weight;
    }

    EdgeWeight edge_weight(EdgeId edge_id) const {
        return _edges[edge_id].weight;
    }

  private:
    std::vector<NodeData> _nodes;
    std::vector<EdgeData> _edges;
};
```

A Story about Specialization

Dr. S. Fina notices that many users only need unit weights. Therefore, these should be supported with maximum efficiency.

Current Inefficiencies?

- Unnecessary reads and writes of weight data: we know the value is always 1
- Unnecessary space usage
- Clustering (with weight penalty): division by 1 could be removed

Marker Types to the Rescue

- Introduce specific type for unit weight

```
struct UnitWeight {};
```

- We can make this the **default** type

```
template<typename NodeWeight = UnitWeight,
         typename EdgeWeight = UnitWeight>
class Graph { /* ... */ };
```

- Holds no data, thus no reads or writes
- No memory overhead (?)
- Problem solved?

A Story about Specialization

```
struct UnitWeight {};
```

Let's double-check!

```
template<typename NodeWeight>
struct NodeData {
    EdgeId first_edge;
    NodeWeight weight;
};
```

```
static_assert(sizeof(NodeData<int32_t>) == 8);
// true, as expected
static_assert(sizeof(NodeData<UnitWeight>) == 4);
// error: static assertion failed
// (actual size is 8)
```

- ... what happened?

  ⇒ Unique address property of C++ objects requires extra space

- Problem 1: `UnitWeight` still takes up space
- Problem 2: Algorithms expect concrete numbers for weights

```
template<typename NodeWeight, typename EdgeWeight>
class Graph {
    ??? node_weight(NodeId node_id) const;
    ??? edge_weight(EdgeId edge_id) const;
};
```

- Inserting `UnitWeight` would break calling code

  ⇒ So what is the return type?

A Story about Specialization

An Overview of Specialization

- Idea: write separate code for specific values of template parameters

```
template<typename T>  // primary template
void foo() {
    std::println("general case");
}

template<>  // (full) specialization
void foo<int>() {
    std::println("this is an int");
}
```

- Works analogously for classes
- Example: `std::vector<bool>` is a completely separate implementation
- Class templates even support complex patterns (partial specialization)
  - *Functions not, probably because overload resolution is a nightmare even without this*

```
template<typename T>  // primary template
class Bar { };

template<>  // full specialization
class Bar<int> { };

template<typename T>  // partial specialization
class Bar<std::vector<T>> { };
//        ^^^^^^^^^^^^^^
// pattern is "matched" against primary template
```

A Story about Specialization

Back to Graphs

- We can just write two versions!

```cpp
template<typename NodeWeight>
struct NodeData {
    uint32_t first_edge;
    NodeWeight weight;

    using Weight = NodeWeight;

    Weight get_w() const {
        return weight;
    }
};
```

```cpp
template<>
struct NodeData<UnitWeight> {
    uint32_t first_edge;

    // int32_t as default
    using Weight = int32_t;

    Weight get_w() const {
        return 1;
    }
};
```

```cpp
static_assert(sizeof(NodeData<UnitWeight>) == 4);
// Hooray, the assertion holds!
```

- We can even provide a unified interface

- Next: redeclare types in graph

  ⇒ Users can work with `Graph::NodeWeightT`

```cpp
template<typename NodeWeight = UnitWeight,
         typename EdgeWeight = UnitWeight>
class Graph {
  public:
    using NodeWeightT = NodeData<NodeWeight>::Weight;
    using EdgeWeightT = EdgeData<EdgeWeight>::Weight;

    NodeWeightT node_weight(NodeId node_id) const {
        return _nodes[node_id].get_w();
    }

    EdgeWeightT edge_weight(EdgeId edge_id) const {
        return _edges[edge_id].get_w();
    }

  private:
    std::vector<NodeData<NodeWeight>> _nodes;
    std::vector<EdgeData<EdgeWeight>> _edges;
};
```

A Story about Specialization

As time goes by, more and more specific graph types are supported.

Ralf thinks specialization could be perfect to select specialized algorithms at compile time.

```
template<class Graph>
void computeComponents(const Graph& graph,
    std::vector<NodeId>& output) {
    // use CMG algorithm*
}
template<NW, EW>
void computeComponents<UndirectedGraph<NW, EW>>(
    const UndirectedGraph<NW, EW>& graph,
    std::vector<NodeId>& output) {
    // graph is undirected: use simple DFS
}
```

** a simple and efficient single-pass SCC algorithm*

- Unfortunately, undirected graphs are not implemented as a separate class
- Specialization is rather cumbersome and hard to read
- Better: constants and `if constexpr`

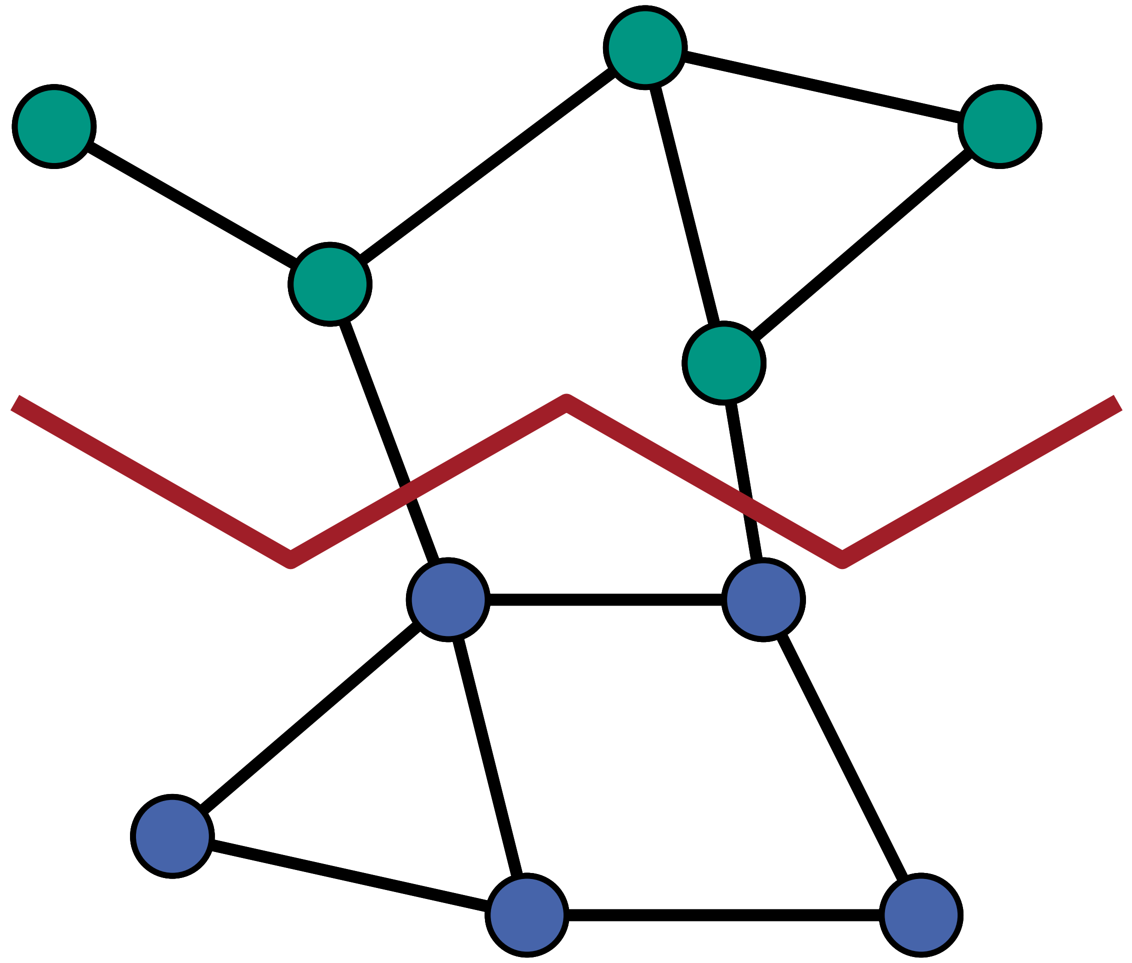
```
template< /* ... */ >
class Graph {
    // compile time flag whether graph is directed
    static constexpr bool is_directed = /* ... */;
};

template<class Graph>
void computeComponents(const Graph& graph,
    std::vector<NodeId>& output) {
    // branch is selected at compile time
    if constexpr (Graph::is_directed) {
        // use CMG algorithm
    } else {
        // use simple DFS
    }
}
```

A Story about Algorithms and Interfaces

Due to the success of the library,
Dr. S. Fina wants to extend it with a partitioning algorithm.

- **Goal:** compute blocks of equal size, cut as few edges as possible



Composing Algorithms

- The algorithms start with a clustering
- Using the clustering, an initial solution is computed
- Finally, the solution is refined with a local search algorithm

$\Rightarrow$ Contains clustering and refinement as sub-algorithms

A Story about Algorithms and Interfaces

"Should be easy", Ralf thinks.

- The last refactoring encapsulated the clustering algorithm in a `class`
- Just add the refinement and remaining logic in the same way
- Policy pattern ensures really good efficiency

```cpp
template<class RatingPolicy,
         class MaxWeightPolicy,
         class PriorityPolicy>
class Clustering { /* ... */ };

template<class Objective, class TieBreakPolicy>
class Refinement { /* ... */ };

template<class ClusteringAlg, class RefinementAlg>
class Partitioning {
  public:
    template<class Graph>
    void partition(const Graph& graph,
        uint32_t num_blocks,
        std::vector<uint32_t>& partition_ids) {
        /* ... */
    }

  private:
    ClusteringAlg _clustering;
    RefinementAlg _refinement;
};
```

A Story about Algorithms and Interfaces

Oh no! Although the algorithm is fast, many users are unhappy.

Some report compile times of **multiple hours** after integrating the partitioning algorithm.*

Others complain that the debug binary has a size of almost **10 GB**.

- What is going on?

** of course, these are only the select few with enough RAM to finish compiling at all*

```
template<class ClusteringAlg, class RefinementAlg>
class Partitioning {
    template<class Graph>
    void partition(/* ... */) {
        /* ... */
    }
};
```

Revisiting the Implementation

- `Partitioning` reuses both the clustering and refinement and is also templated by the graph class
- The clustering uses 3 policies and the refinement 2
- Assuming 5 graph variants and 3 variants per policy, this results in a total of $5 \times 3^5 = 1.215$ variants

A Story about Algorithms and Interfaces

The Dilemma

- Too many template parameters cause an exponential explosion of compile time and binary size
- But using dynamic dispatch for the policies is also inefficient

A Solution

- Better: use a healthy mixture of dynamic and static dispatch
- We can lift the dynamic dispatch out of the hot loop

$\Rightarrow$ The correct interface for dynamic dispatch is the sub-algorithms

```cpp
template<class Graph>
struct ClusteringInterface {
    virtual void clustering_round(
        const Graph& graph,
        std::span<uint32_t> cluster_ids) = 0;
};

template<class Graph, class RatingPolicy,
         class MaxWeightPolicy, class PriorityPolicy>
struct Clustering: public ClusteringInterface<Graph> {
    void clustering_round(const Graph& graph,
        std::span<uint32_t> cluster_ids) override {
        /* ... */
    }
};


template<class Graph>
struct RefinementInterface {
    /* ... */
};

template<class Graph, class Objective,
         class TieBreakPolicy>
struct Refinement: public RefinementInterface<Graph> {
    /* ... */
}
```

A Story about Algorithms and Interfaces

Interfaces

- The sub-algorithms get a single interface which erases the type information of the policies
  - *We can't erase the graph type, since it includes arbitrary user-provided types (would also require downcasting)*
  - *Note: virtual methods can't be templated*
- Correct algorithm variant is selected once via dynamic dispatch
- The inner loops are perfectly efficient due to static dispatch
- Reduced to one code variant per graph

$\Rightarrow$ Moral of the story:
choose your interfaces wisely

```cpp
template<class Graph>
struct ClusteringInterface {
    /* ... */
};

template<class Graph>
struct RefinementInterface {
    /* ... */
};

template<class Graph>
class Partitioning {
  public:
    void partition(const Graph& graph,
        uint32_t num_blocks,
        std::vector<uint32_t>& partition_ids) {
        /* call clustering, compute initial
           partition, call refinement */
    }

  private:
    using Clustering = ClusteringInterface<Graph>;
    using Refinement = RefinementInterface<Graph>;

    std::unique_ptr<Clustering> _clustering;
    std::unique_ptr<Refinement> _refinement;
};
```

Multitudes of Templates

One feature is still missing: users would like a simple way to select algorithms at runtime.

- Currently: type parameters must be manually specified
- **Goal:** function that takes arguments representing policies (enum!) and returns according algorithm

```
template<class Graph>
std::unique_ptr<ClusteringInterface<Graph>>
create_clustering_alg(
    RatingPolicy rp, MaxWeightPolicy mwp,
    PriorityPolicy pp) {
    // TODO
}
```

```
enum class RatingPolicy {
    WeightPenalty,
    NoPenalty
};

class WeightPenalty { /* ... */  };

class NoPenalty { /* ... */  };
```

- Brute force approach: write a lot of nested `switch` statements
- For large number of combinations a bit ugly and error prone
  - *Could be "improved" with macros*
- There is surely a generic solution?

Multitudes of Templates

Variadic Templates

- Allow to handle lists of type parameters

```
// a "meta" type for creating lists of types
template<typename... T>  // parameter pack
struct Typelist;

template<typename T, typename... Ts>
struct Typelist<T, Ts...> {
    using Head = T;
    using Tail = Typelist<Ts...>;  // pack expansion
};

template<>
struct Typelist<> { };
```

- Build definitions recursively with a "base case" (empty list) and a "regular case"

  ⇒ Case distinction via specialization!

- Also usable for more regular code, e.g. allowing variadic function arguments
- Example: `std::vector::emplace_back`

```
// how to generically call a constructor*
template<class Constructed, typename... Ts>
Constructed construct_generic(Ts... args) {
    // pack expansion of args
    return Constructed { args... };
}
```

* *though we need perfect forwarding ↑ to do this properly*

Multitudes of Templates

- Step 1: connect type to enum value

```
enum class RatingPolicy {
    WeightPenalty,
    NoPenalty
};

class WeightPenalty {
    static constexpr RatingPolicy value =
        RatingPolicy::WeightPenalty;
};

class NoPenalty {
    static constexpr RatingPolicy value =
        RatingPolicy::NoPenalty;
};
```

- Step 2: use a variadic mapper `struct` to do the template magic 🪄
  - *Note: we only consider a single policy to keep it remotely readable*
  - *Multiple policies: use type lists, since only one variadic per definition is possible*

```
template<class Graph>
using AlgPtr = std::unique_ptr<ClusteringInterface<Graph>>;

template<class Graph, class... Policies>
struct AlgFactory;

template<class Graph, class Policy, class... Policies>
struct AlgFactory<Graph, Policy, Policies...> {
    static AlgPtr<Graph> create(RatingPolicy rp) {
        if (Policy::value == rp) {
            return std::make_unique<
                        Clustering<Graph, Policy>>();
        } else {  // "recursive" call
            return AlgFactory<Graph, Policies...>
                    ::create(rp);
        }
    }
};

template<class Graph>
struct AlgFactory<Graph> {
    static AlgPtr<Graph> create(RatingPolicy) {
        throw "Value did not match any policy!";
    }
};

AlgPtr<Graph> create_clustering_alg(RatingPolicy rp) {
    return AlgFactory<Graph, WeightPenalty, NoPenalty>
            ::create(rp);
}
```

Multitudes of Templates

Some Remarks

- Variadic templates are a powerful metaprogramming tool
- Combination with specialization allows "recursive" logic
- Code gets quickly arcane and unreadable

  ⇒ Use with caution

The Dark Path

- This is really powerful ... what else can we do with it?
- Is there a general framework for compile time logic?

```
template<class Graph, class... Policies>
struct AlgFactory;

template<class Graph, class Policy, class... Policies>
struct AlgFactory<Graph, Policy, Policies...> {
    /* .... */
};

template<class Graph>
struct AlgFactory<Graph> {
    /* .... */
};

AlgPtr<Graph> create_clustering_alg(RatingPolicy rp) {
    return AlgFactory<WeightPenalty, NoPenalty>
           ::create(rp);
}
```

Template Metaprogramming

- **Basic insight:** a struct corresponds to a function that maps types to another type

```
template<typename T>  // function input
struct Identity {
    using Result = T;  // function output
};
```

- Specialization gives us pattern matching

```
template<typename S, typename T>
struct IsEqual {  // default case
    static constexpr bool Result = false;
};

template<typename T>
struct IsEqual<T, T> {  // specific pattern
    static constexpr bool Result = true;
};
```

- Pattern matching enables other conditionals
  - *Note: template parameters can also be values of primitive types*

```
template<bool Cond, typename Then, typename Else>
struct IfThenElse;

template<typename Then, typename Else>
struct IfThenElse<true, Then, Else> {
    using Result = Then;
};

template<typename Then, typename Else>
struct IfThenElse<false, Then, Else> {
    using Result = Else;
};
```

$\Rightarrow$ **Meta-language** where types are values and templates are functions
(and of course it is turing complete)

Template Metaprogramming

- Template template parameters (i.e., template parameters which are themselves templates) even allow to reproduce the full functional pipeline

```
template<template<typename> class Mapper,
         class InputList>
struct Map;   // apply Mapper to every element

template<template<typename, typename> class Op,
         class Init, class InputList>
struct Fold;  // fold the list via Op
```

Meta Language	C++
Value	Type
Member value	`using` declaration
Member function	Templated `using` declaration
List	Variadic template or Cons/Nil
Function	Templated class
Function parameter	Template parameter
Higher-order function parameter	Template template parameter

Template Metaprogramming

Meta Language Properties

- Functional
- Interpreted
- Dynamically typed
  (at least without concepts)
- Only recursion, no iteration
- No mutation*
- Powerful pattern matching

... and extremely ugly syntax

** We would hope there is no mutation, since types can't change during compilation, right? Well ... turns out the world is a cruel place*

More Use Cases for Templates

- Matrices and vectors with dimensions known at compile time
- Lazy evaluation of mathematical formulas via "expression templates"
- Representing physical units (e.g. seconds, meters) in the type system
- ...

The Constant of the Story

After two years of working on a compile time graph library, Ralf has quit his job in despair.

He now works at an insurance – with only regular, boring code, right?

- Szenario: a lot of statistical calculations which need $\binom{n}{k}$
- In most cases, n has the same value (say $n = 30$)
- How to speed this up?

It's just a lookup

- We can compute a lookup table with all values at compile time *(at runtime might be good enough, but that is boring)*
- Computation is a single read of cached memory
- Restriction: we can't use template meta programming
  (we want to preserve Ralf's mental health)

The Constant of the Story

Compile Time Computation

- A `constexpr` variable is a compile-time constant
- `constexpr` functions can be evaluated at compile time...

```
template<typename T>
constexpr size_t num_bits() {
    return sizeof(T) * CHAR_BIT;
}

constexpr size_t BITS_OF_INT = num_bits<int>();
```

- ...but have additional restrictions *(`noexcept`, only literal types, transitive calls are `constexpr`)*

Yet Another Way of Metaprogramming

- `constexpr` is surprisingly powerful despite the restrictions
- Specifically, it is turing complete even without using templates

  ⇒ Preferable to templates for anything involving calculations

The Constant of the Story

A Lookup Table

- Goal: lookup table for $\binom{n}{k}$ with fixed n

```
constexpr __uint128_t partial_fac(__uint128_t current,
                                  __uint128_t lower) {
    if (current == lower) {
        return 1;
    }
    return current * partial_fac(current - 1, lower);
}

template<size_t N>
constexpr std::array<__uint128_t, N + 1>
lookup_table() {
    std::array<__uint128_t, N + 1> result;
    for (size_t k = 0; k <= N; ++k) {
        result[k] = partial_fac(N, k)
                  / partial_fac(N - k, 0);
    }
    return result;
}
```

A Note on Datatypes

- Use 128 bit integers since faculties get really la
  really quickly
- Unfortunately, `__uint128_t` is implementatio
  defined*

```
__uint128_t faculty(size_t n, size_t k) {
    constexpr auto LOOKUP_N30 = lookup_table<30
    if (n == 30) {
        return LOOKUP_N30[k];
    }
    return partial_fac(n, k) / partial_fac(n -
}
```

- Benchmark**: almost 6x speedup

* *but couldn't we use `uintmax_t`? No! `uintmax_t` has 64 bit, f*
... reasons

** *https://godbolt.org/z/az7Ma7nrc*

Summary

Many New Tools

- The policy pattern
- Specialization
- Variadic templates for lists of types
- Beware exploding complexity (and compile times) when using templates!
- Use a mixture of static and dynamic dispatch with good interfaces

Two Kinds of Metaprogramming

- Template metaprogramming can allow super generic code
- `constexpr` for calculations
- Mostly useful for libraries