Abstract approach to templates

Abstract description of templates in C++

- A template is a function evaluated by the compiler
  - Its arguments may be:
    - integral constants

```
template< std::size_t N> /*...*/;
```

    - types

```
template< typename T> /*...*/;
```

→ tohle je nejčastější

    - other templates (voilà: functional programming)

```
template< template< /*...*/> class T> /*...*/;
```

Abstract description of templates in C++

- A template is a function evaluated by the compiler
  - Its return value may be one of:
    - an integral constant
      - indirectly (convention: static member constant named "value")

```
template< /*...*/> class F { static constexpr int value = /*...*/; };
```

      - directly [C++14] (convention: named "F_v" if defined as F::value)

```
template< /*...*/> constexpr int F_v = /*...*/;
```

    - a type
      - a class - directly

```
template< /*...*/> class F {/*...*/};
```

      - any type indirectly (convention: member named "type")

```
template< /*...*/> class F { typedef /*...*/ type; };
```

      - any type directly [C++11] (convention: named "F_t" if defined as F::type)

```
template< /*...*/> using F_t = /*...*/;
```

    - a template
      - indirectly

```
template< /*...*/> class X { template< /*...*/> using type = /*...*/; };
```

    - a run-time function (a function template or a static member function of a class template)

Abstract description of templates in C++

- A template is a function evaluated by the compiler
  - Its arguments may be:
    - integral constants
    - types
    - templates, i.e. compile-time functions
  - Its return value may (directly or indirectly) be one of:
    - an integral constant
    - a type
    - a template, i.e. a compile-time function
    - a run-time function
  - A template may also "return" a "compile-time structure":

```
template< /*...*/> class F {
  using A = /*...*/;
  static const int B = /*...*/;
  static int C(int x, int y) {/*...*/}
};
```

tohle můžu chápat funkci, co vrací víc hodnot

Abstract description of templates in C++

- There is a *compile-time programming language* inside C++
  - The language can operate on the following atomic "values":
    - integral constants
    - types
    - run-time functions
  - The operations available are
    - integral operators of C++ (plus part of standard library marked *constexpr*)
    - type constructions (creating pointers, references, arrays, classes, ...)
    - defining a new run-time function (which may call others, including compile-time "values")
  - The "values" may be combined into "structures"

- The compile-time language is *functional*
  - no variables which could change their value
  - "compile-time functions" (i.e. class templates) are first-class "values"

- **constexpr** functions — > u proměnné jde o konstantu vyhodnocenou překladačem
  - The compiler will *interpret* the code of the function during compilation
  - Limitations are gradually being lifted
    - C++11: A constexpr function must consist of single return statement
    - C++14: No virtual functions, no dynamic allocation, no goto/asm/throw/try-catch
    - C++17: No virtual functions, no goto/asm/throw/try-catch
    - C++20: No goto/asm, no static variables, few other limitations
  - No access to run-time objects allowed

2 -> můžu dynamicky alokovat, ale musím mít možnost po sobě uklidit
– proto má std::string i constexpr destruktor

Nelze si ale připravit při překladu data pro runtime...

- **constexpr** functions, variables:
  - Must be evaluated by compiler when used in template arguments etc.
    - In this case, any objects must have their lifetime confined within the evaluation
  - May be evaluated by compiler as an optimization elsewhere
  - [C++20] **consteval, constinit**: every call/init must return a constant
- Cf. **const** variables, references, pointers, functions:
  - Compiler prohibits run-time modification (via this access path only)

- **constexpr** functions vs. template metaprogramming
  - constexpr function always obey the rules for run-time functions
    - Types can not be manipulated by constexpr functions
    - Expressions must be statically typed
      - A loop can not work with elements of different types

```
using T = std::tuple<int,double,string>;
/*constexpr*/ T for_each(T p)
{
  for (auto && x : p)       // ILLEGAL - std::tuple does not have begin()/end()
    x += 'A';               // ILLEGAL - x is not statically typed here
  return p;
}
```

⇒ v každé iteraci je to jiný typ, to nikdy překladač nedá

- Using templates, implementation of such loops is possible, although tricky

Abstract description of templates in C++

- There is a *compile-time programming language* inside C++
- The compile-time language is *functional*
- There are *Prolog*-like features
  - A "compile-time function" (i.e. a template) may be defined by several independent rules (using partial specialization)

```
template< typename T, int N> class F;
template< typename T> class F<T, 0> { /*...*/ };
template< typename U, int N> class F<U*,N> { /*...*/ };
template< typename P> class F<X<P>, 1> { /*...*/ };
```

Jistá forma prologovské unifikace
– podle specializace zvolí impl.

- There is unification of terms representing type arguments
- But unlike Prolog, there is no priority or try-next-if-failed mechanism

- With [C++20] concepts we can even emulate Prolog '!'
  - Before [C++20], it required decltype(), function templates, and SFINAE

Abstract description of templates in C++

- There is a *compile-time programming language* inside C++
- C++ types form a complex universe of "compile-time values"
  - Classes (class/struct/union) are compared *by name* → dvě stejné struktury se stejným obsahem nejsou „stejné“
    - New unique "values" are generated by each occurrence of structure in source code
    - Empty structures may be employed as unique opaque values (usually called *tags*)
  - Other type constructs (*,&,&&,[],()) are compared by contents
    - This allows unification in partial specializations → pointer bude i po přiřazení furt stejný..
  - Instances of the same class template are considered equal if their arguments are equal
  - Aliases created by typedef/using are considered equal to their defining types

Abstract description of templates in C++

- There is a *compile-time programming language* inside C++
- The output of the compile-time program is a run-time program
  - *Compile-time functions* may return *run-time types* and *run-time functions*
    - They may also generate static_assert and other error messages

      → ; když tam error bude, tak nemám runtime, do kterého bych to poslal!
- Elements used only during compile-time
  - integral constants
  - structures containing types, constants, static functions
    - often called *policy class*
    - even empty structures are important as *tags*
  - templates perceived as compile-time functions
    - those returning policy classes are often called *traits*
- Compile-time elements converted to run-time representations
  - C++ types
  - run-time functions

Template tricks

C++11: <ratio>

- Compile-time arithmetics
  - Rational numbers

```
template< intmax_t n, intmax_t d = 1> struct ratio;
```

  - Example

```
using platform = ratio_add<ratio<9>, ratio<3,4>>;      // 9   // 3/4
```

→ jde pouze o název, nedrží žádnou hodnotu

```
static_assert( ratio_equal< platform, ratio< 975, 100>>::value, "?");
```

- Evaluated by the compiler
  - Including computing the greatest common divisor!
- No longer needed because of stronger constexpr functions

Metaprogramming

„Něčím si nechám generovat můj runtimový program"

Example

- Type transformation
  - Goal: Transform, e.g., this triplet ...

```
using my_triplet = std::tuple< int, my_class, double>;
```

  - ... into the triplet std::tuple< F(int), F(my_class), F(double)>

→ označuje něco jako funkci co dostane typ a vrátí jiný typ

- What it is good for?
  - column-stores:

→ například na vstupu je typ seznamu, na výstupu je typ vectoru

```
std::tuple< std::vector<int>, std::vector<my_class>, std::vector<double>>
```

  - passing of references:

```
std::tuple< int &, my_class &, double &>
```

Example

- Type transformation
  - Goal: Transform, e.g., this triplet ...

```
using my_triplet = std::tuple< int, my_class, double>;
```

  - ... into the triplet std::tuple< F(int), F(my_class), F(double)>

How to invoke the transformation?

- Standard C++ library uses the "::type" convention:

```
using my_transformed_triplet = type_transform< my_triplet, F>::type;
```

  - In case of dependent types, „typename" is required

```
template< typename some_triplet> class X {
  using transformed_triplet = typename type_transform< some_triplet, F>::type;
};
```

- C++14 adds the "_t" convention:

```
using my_transformed_triplet = type_transform_t< my_triplet, F>;
```

  - No typename required

Example

- Type transformation
  - Goal: Transform, e.g., this triplet ...

```
using my_triplet = std::tuple< int, my_class, double>;
```

  - ... into the triplet std::tuple< F(int), F(my_class), F(double)>

```
using my_transformed_triplet = type_transform_t< my_triplet, F>;
```

- How can we specify the F "function" for type_transform_t?

A. Directly:

až někde hlouběji bude použití takového argumentu s <...>

```
using my_reference_triplet = type_transform_t< my_triplet, std::add_lvalue_reference_t>;
```

  - Failed before C++17 because std::add_lvalue_reference_t is not a class template

```
using my_vector_triplet = type_transform_t< my_triplet, std::vector>;
```

  - Failed before C++17 because std::vector has 3 arguments but type_transform_t expects 1

Template template arguments

- Template template arguments before C++17

```
template< template< typename> class P>
class example { using a_class = P<int>; }
```

  - The actual argument was required to be a class/struct template

```
template< typename T> using A1 = std::vector< T>;
using my_type1 = example< A1>;
```

    - Failed because A1 is an alias template
  - The arguments of the actual argument were required to exactly match the arguments of the formal argument

```
using my_type2 = example< std::vector>;
```

    - Failed because std::vector has three arguments (although two of them have default values)
- C++17 changed the behavior
  - Alias templates are allowed as actual arguments, default arguments considered
  - The compatibility of the actual argument has a new, hardly readable definition:
    - Formally, a template template-parameter P is at least as specialized as a template template argument A if ...
  - "typename" may be used instead of "class" in the formal argument declaration
- Both the examples work in C++17

Example

- How can we specify the F function for type_transform_t?
  A. Directly (requires C++17):
     - The argument is a template class/struct or template alias

```
using my_reference_triplet = type_transform_t< my_triplet, std::add_lvalue_reference_t>;

using my_vector_triplet = type_transform_t< my_triplet, std::vector>;
```

  B. Indirectly, using the well-known “::type” convention:
     - The argument is a template class/struct containing an alias member named "type"

```
using my_reference_triplet = type_transform_t< my_triplet, std::add_lvalue_reference>;

template< typename T> struct add_vector { using type = std::vector< T>; };

using my_vector_triplet = type_transform_t< my_triplet, add_vector>;
```

  C. Indirectly, using a nested template
     - The argument is a class/struct containing a template alias member named "type"
     - This is an equivalent of a functor, in the realm of meta-functions returning types

```
struct add_vector { template< typename T> using type = std::vector< T>; };

using my_vector_triplet = type_transform_t< my_triplet, add_vector>;
```

položka smluveného jména, šablonovaná

Example

- The implementation of type_transform must match the selected convention
  A. Directly (requires C++17):
     - The argument is a template class/struct or template alias

```
template< typename T, template< typename> typename F>
struct type_transform {
  using example_usage = F< int>;
};
```

  B. Indirectly, using the well-known "::type" convention:
     - The argument is a template class/struct containing an alias member named "type"

```
template< typename T, template< typename> class F>
struct type_transform {
  using example_usage = typename F< int>::type;
};
```

  C. Indirectly, using a nested template
     - The argument is a class/struct containing a template alias member named "type"

```
template< typename T, typename F>
struct type_transform {
  using example_usage = typename F::template type<int>;
};
```

- type_transform_t is a wrapper with the same arguments as type_transform

Example

- The full implementation (C++17)
  - General template struct declaration

takže třeba vector jde, protože lze volat s jediným argumentem

```
template< typename T, template< typename> typename F>
struct type_transform;
```

→ tohle říká, že to musí být šablona volatelná s jediným parametrem

  - Partial specialization for std::tuple
    - The partial specialization allows to extract the type list L from std::tuple< L...>
    - A type list can exist only as a template argument – accessible only inside such a template

```
template< typename ... L, template< typename P> class F>
struct type_transform< std::tuple< L ...>, F>
{
    using type = std::tuple< F< L> ...>;
};
```

    - We may also implement specializations for std::pair or std::array
  - Wrapper alias template

```
template< typename T, template< typename> typename F>
using type_transform_t = typename type_transform< T, F>::type;
```

- In the other two conventions, more complex type-ids are used instead of F<L>

Iterating through n-tuple elements

Example

- Working with tuple values

```
using my_triple = std::tuple< int, my_class, double>;
using my_transformed_triple = type_transform_t< my_triple, std::vector>;


my_triple a;
my_transformed_triple c;
auto my_value_function = /*...*/;
static_transform(a, c, my_value_function);
```

chci, aby mi to transformovalo int -> std::vector<int>, my_class -> std::vector<my_class>...

- static_transform shall be an equivalent of std::transform for tuples
  - The intent is to apply f to every element of x and store the results in r
  - Implementation like this will **not** compile:

```
template< typename T1, typename T2, typename F>
void static_transform( T1 && x, T2 && r, F f)
{ for ( std::size_t i = 0; i < std::tuple_size_v< T1>; ++i)
    std::get<i>(r) = f( std::get<i>(x));
}
```

tohle nepůjde přeložením

→ tady ale potřebuju constantu, což řídící proměnná for-cyklu není.

- my_value_function is a polymorphic functor
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors

Naopak chci kód pro každý typ specializovaný.

Nikdy to nebude jedna runtimová instance, která by se volala na všechny typy.

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
- The simplest case is a polymorphic lambda (C++14)

```
auto my_value_function = [](auto && x){
  return std::vector</*???*/>( 1, x);
};
```

  - Problem: How to derive the type for the vector?

```
std::vector<std::remove_cv_t<std::remove_reference_t<decltype(x)>>> x
```

tohle nepotřebuju

místo toho

  - Problem: The argument x shall be passed using `std::forward</*???*/>( x)`
- Since C++20, lambda may contain explicit template arguments

```
auto my_value_function = []<typename T>(T && x){
  return std::vector<std::remove_cvref_t<T>>( 1, std::forward<T>(x));
};
```

→ tohle mi vyrobí std::vector<T> z nějakého typu T, který jí předám referencí

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
- Explicitly defined functor

```
struct my_value_functor {
  template< typename T1>
  std::vector<T1> operator()( T1 && x) const
  { return std::vector<T1>( 1, std::forward<T1>(x));
  }
};
auto my_value_function = my_value_functor{};
```

  - BEWARE: It does NOT work correctly!
  - If the actual argument is an lvalue of type T, T1 would be T &
    - Reference collapsing rules apply
  - The result type would be a vector of references!

- The problem encountered here is quite frequent and often forgotten!

Example

- Polymorphic functors
  - applied to arguments of different types, may return diferent types
  - in this case, it transforms single values into vectors
- Explicitly defined functor – corrected implementation

```
struct my_value_functor {
  template< typename T1>
  auto operator()( T1 && x) const
  { return std::vector<std::remove_cv_t<std::remove_reference_t<T1>>>
      ( 1, std::forward<T1>(x));
  }
};
```

  - std::remove_reference_t removes & or && if present
  - std::remove_cv_t removes const or volatile if present
  - C++20: std::remove_cvref_t combines the two
  - Using auto in the operator declaration avoids repeating the ugly type-id

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
- Polymorphic lambda – corrected implementation

```
auto my_value_function = [](auto && x){
  using x_t = decltype(x);
  using e_t = std::remove_cv_t< std::remove_reference_t< x_t>>;
  return std::vector<e_t>( 1, std::forward<x_t>(x));
};
```

<= tohle je trochu podvod, ale výsledek je stejný...

  - decltype(x) is the type of x, always a (lvalue or rvalue) reference
  - std::forward is used differently from the usual practice, but it works:
    - Our code is equivalent to

```
template< typename U> auto f(U && x) { /*...*/ std::forward<U &&>(x) /*...*/ }
```

    - std::forward<T>(x) is by definition equivalent to static_cast<T&&>(x)
    - Therefore, std::forward<U&&> is equivalent to std::forward<U> due to reference collapsing

Example

- Iterating through n-tuple elements
  - A non-working implementation

```
template< typename A, typename R, typename F>
void static_transform(A && a, R && r, F f)
{
    using atuple = std::remove_cv_t< std::remove_reference_t< A>>;
    using rtuple = std::remove_cv_t< std::remove_reference_t< R>>;
    static constexpr auto asize = std::tuple_size_v< atuple>;
    static constexpr auto rsize = std::tuple_size_v< rtuple>;
    static_assert(asize == rsize, "tuples must be of equal length");

    for ( std::size_t i = 0; i < std::tuple_size_v< T1>; ++i)
        std::get<i>(r) = f( std::get<i>(x));
}
```

dělostřelecká příprava

—> tohle furt nelze přeložit

  - std::get<i> requires constant i
  - The compiler must be able to instantiate f() for the correct types

Example

- Iterating through n-tuple elements
  - Reformulated using a compile-time version of std::for_each
  - static_for_each_index calls transform_ftor for each index in the given range

```
template< typename A, typename R, typename F>
void static_transform(A && a, R && r, F f)
{
    using atuple = std::remove_cv_t< std::remove_reference_t< A>>;
    using rtuple = std::remove_cv_t< std::remove_reference_t< R>>;
    static constexpr auto asize = std::tuple_size_v< atuple>;
    static constexpr auto rsize = std::tuple_size_v< rtuple>;
    static_assert(asize == rsize, "tuples must be of equal length");

    static_for_each_index< 0, rsize>(
        transform_ftor< A, R, F>(
            std::forward< A>(a), std::forward< R>(r), std::move(f)));
}
```

jako argumenty jsou tu konstanty

Example

- static_for_each_index - interface

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f);
```

konstanty odsud-dosud

- Logically, it shall call f(i) for every i in [b,e)
- Problem: i must be a constant inside f *kulaté závorky jsou runtimová věc*
  - The compiler must create a separate instantiation for every f(i)
- Problem: f<i>() does not work because f is a variable, not a function *→ tohle normálně nejde...*
  - f.operator()<i>() does not work either *→ tohle taky funguje*
- Possible solution: call a normal member function instead of operator()

```
f.call<i>();
```

  - Requires a functor like

```
struct ftor {
    template< std::size_t i>
    void call() const
    { /*...*/ }
};
```

Example

- static_for_each_index - interface

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f);
```

  - Logically, it shall call f(i) for every i in [b,e)
  - Problem: i must be a constant inside f
  - A better solution: pass the constant through the type of an argument

```
f(static_index<i>{});
```

=> minimální prázdný tag, kterým dodáme typ

  - The argument type is an empty tag class

```
template< std::size_t i> struct static_index {};
```

=> jednodušší pomocí struktury

  - Requires a functor like

```
struct ftor {
    template< std::size_t i>
    void operator()(static_index<i>) const
    { /*...*/ }
};
```

Example

- A better solution: pass the constant through the type of an argument
  - We can also use std::integral_constant as the tag class

```
template< std::size_t i>
using static_index = std::integral_constant<std::size_t, i>;
```

→ už vytvořené ve std::, nemusím dělat sám!

  - With integral constant, we may even use a (C++14) lambda:

```
auto ftor = [](auto tag_value){
  constexpr std::size_t i = tag_value;
};
```

→ tohle je objekt typu s konverzním operátorem na size_t, s přívlastkem constexpr

to „i" je schované v typu toho „tag_value"

  - std::integral_constant defines a conversion operator to std::size_t which, although not declared static, does not access the object and returns the template argument i
  - therefore, the member function may be evaluated in constant context although tag_value is a run-time object
  - at run-time, the tag_value is an empty object and most compilers will not reserve any space for it, consequently, there will be no run-time cost for dealing with this object
  - tag objects shall always be passed by value – passing by reference may have a run-time cost

Example

- A functor for static_transform

```
template< typename A, typename R, typename F> struct transform_ftor
{
    transform_ftor(A && a, R && r, F f)
    : a_(std::forward<A>(a)), r_(std::forward<R>(r)), f_(std::move(f))
    {}
    template< std::size_t i> void operator()(static_index<i>) const
    {   std::get< i>(std::forward<R>(r_)) = f_( std::get< i>(std::forward<A>(a_)));
    }
private: A && a_; R && r_; F f_;
};
```

tohle může vypadat, že to bude volat n-krát, tak to ve skutečnosti jen jednou.

kopie malého funktoru, co dělá tu transformaci typů

→ když bych dostal r-value, tak i tady ponechám r-value.

- Formally, A && and R && in the constructor are NOT universal references
  - However, we will supply the type arguments A, R from a context where A && and R && are universal references (and consistent with the actual arguments to the constructor)
  - Therefore, the use of forward in the constructor is correct
  - Since the member variables a_ and r_ have the same type as the arguments, they also act as if they were universal references
  - The use of forward in operator() is correct because it is invoked only once for each i
- if A && is rvalue reference, f_ may steal from it
  - std::get propagates the rvalue/lvalue distinction from its argument

Example

- Static for-loop – an implementation by recursion
  - A class/struct template is required because we need partial specialization

```
template< std::size_t b, std::size_t n> struct for_each_index_helper {
    template< typename F> static void call(F && f)
    {   f(static_index< b>());
        for_each_index_helper< b + 1, n - 1>::call( std::forward<F>(f));
    }
};
```

  - Partial specialization stops the recursion

```
template< std::size_t b> struct for_each_index_helper< b, 0> {
    template< typename F> static void call(F &&) {}
};
```

→ konec rekurze, kde už neudělám nic

začátek *konec* *<b; e)*

  - Public wrapper

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f)
{   for_each_index_helper< b, e - b>::call(std::move(f));
}
```

→ tady to změním na počet, abych mohl dojít ke konci <b,0>

  - The public function receives the functor f by value, similarly to all std algorithms
  - Internally, we pass it by universal reference to avoid excess copying

Example

- Static for-loop – implementation without recursion
  - we need to generate the indices {0,...,n-1}

  - C++14 library contains this:

```
template< std::size_t ... IL>
using index_sequence = std::integer_sequence<std::size_t, IL...>;
```

    - index_sequence is an alias to a more general tag class

```
template< std::size_t N>
using make_index_sequence = /* black magic */;
```

    - The black magic ensures that make_index_sequence<N> == index_sequence< 0, 1, ..., N-1>

- Usage:
  - Use make_index_sequence<N> to generate a list of indices
    - The list is not accessible directly
  - Use partial specialization (or a function template) as a context in which the list is visible as a template parameter pack
    - Similar to the implementation of type_transform; instead of a type list argument of a tuple, here we have a constant list argument of an index_sequence

Example

- Static for-loop – implementation without recursion
  - A helper class

```
template< std::size_t b, typename S>
struct for_each_index_helper2;
```

  - A partial specialization is used to access the template parameter pack L

```
template< std::size_t b, std::size_t ... L>
struct for_each_index_helper2< b, std::index_sequence< L ...> >
{   template< typename F> static void call(F & f)
    {   (f( static_index< b + L>()), ...);   —> takhle mi to překladač vytvořil n-krát.
    }
};
```

  - A public wrapper

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F && f)
{
    for_each_index_helper2< b, std::make_index_sequence< e - b> >::call(f);
}
```

Example

- Implementation of make_index_sequence
  - A tag struct template

```
template< typename T, T ... IL> struct integer_sequence {};
```

  - A helper struct to recursively generate a list of integers
    - Defines member type = integer_sequence< T, 0, 1, /*...*/, N-1, (IL+N) ...>

```
template< typename T, T N, T ... IL> struct integer_sequence_generator
: integer_sequence_generator< T, N-1, 0, (IL+1)...> {};
```

  - A partial specialization stops the recursion

```
template< typename T, T ... IL>
struct integer_sequence_generator< T, 0, IL...> {
    using type = integer_sequence< T, IL...>;
};
```

  - The public wrapper

```
template< std::size_t N>
using make_index_sequence = typename integer_sequence_generator< std::size_t, N>::type;
```

- This is recursive at compile time, allowing to avoid recursion at run time
  - The compile-time complexity (N^2) may be improved to (N*log(N))
    - Hint: <IL..., (IL+sizeof...(IL))...> doubles the length of the sequence