EXPLORATION OF C++20 METAPROGRAMMING

INBAL LEVI

WHO AM I?

- A C++ enthusiast.
- An embedded software engineer at Solar Edge working on smart home.
- One of the organizers of CoreCpp conference and user group.
- A Member of WG21.
- One of the founders of the Israeli NB.
- I've studied physics, I also love math. ☺

MOTIVATION

- Templates are a powerful tool for C++ programmers.
- Each C++ version has evolved templates, allowing better usage.
- We'll see the value of adding templates to your code.

MOTIVATION

- Some environments discourage extensive use of templates

Template metaprogramming

Avoid complicated template programming.

(...)

Decision:

Template metaprogramming sometimes allows cleaner and easier-to-use interfaces than would be possible without it, but it's also often a temptation to be overly clever. It's best used in a small number of low level components where the extra maintenance burden is spread out over a large number of uses.

And I don't blame them...

- We'll see how C++20 creates a paradigm shift in the way we use metaprogramming.

OUTLINE

- Part 0: (Prologue) What are templates?
- Part I: Adding templates to existing code
- Part II: Overload resolution and ADL
- Part III: Conditions at compile time pre C++20
- Part IV: Conditions at compile time using C++20 (and beyond…)
- Part V: Advanced methods for controlling compile time logic

PART 0: WHAT ARE TEMPLATES?

PART 0: WHAT ARE TEMPLATES

- A template allows creating a class or a function

```
template <typename T>
class TemplatedClass {

    // Define members, using T
};

template <typename T>
void TemplatedFunction() {

    // Use T
};

template <typename T>
void TemplatedFunction(T x) {

    // Use T
};
```

PART 0: WHAT ARE TEMPLATES

- A Function template is usable only if a definition exists.

```
template <typename T>              // declaration, not enough *
void foo(T t);

template <typename T>              // definition
void foo(T t) {
    T * T;
}
```

- Instantiation of a function template can be explicit or implicit.

```
template void foo <int>(int i);    // Explicit instantiation
template void foo(int i);          // Explicit instantiation
```

```
int main()
{
    int i; float j;
    foo<int>(i);                   // Implicit instantiation by call
    foo(i);                        // Implicit by call (+ argument deduction)
    void(*ptr)(int) = foo;         // Implicit by pointer (+ argument deduction)
}
```

* C++20 adds syntactic sugar for declaring a function template.

PART 0: WHAT ARE TEMPLATES

- Templates are evaluated at **compile time.**
- Templates are Turing – Complete:
  you can achieve any functionality of a Turing machine with templates.
- Templates have a few layers of usage:
  - They can be used to avoid rewriting repeated logic.
  - They can be used to write a generic code, which can be provided as library.
  - They provide a mechanism for moving logic to compile-time.
- Templates, in their simplest use do not generate extra code.
  In fact, they have very similar size and compilation time to regular code.

PART I: ADDING TEMPLATES TO EXISTING CODE

PART I: ADDING TEMPLATES TO EXISTING CODE

- Consider the following template free code.
- Overload resolution is by parameters only.

```
void foo(int i);            // Overload 1
void foo(float s);          // Overload 2
void foo(SomeClass c);      // Overload 3
int foo(int i);             // ambiguous, will not compile

int main()
{
    int i;
    float j;
    SomeClass c;
    foo(i);                 // Overload 1
    foo(j);                 // Overload 2
    foo(c);                 // Overload 3
}
```

PART I: ADDING TEMPLATES TO EXISTING CODE

- We could replace our overloads with a single template function.
- This way, we get two main advantages:
  1. In Code: Save coding time, code duplication, changes can be done in a single place.
  2. In Binary Size: The advantage of code generated only for used types. (lower space overhead)

```cpp
template <typename T>
void foo(T t) {
    // do something with T
}

int main()
{
    int i;
    float j;
    foo(i);          // (implicit) Instantiation for int
    foo(j);          // (implicit) Instantiation for float
}
```

```asm
.LC2:
void foo<int>(int):
        push    rbp
        mov     rbp, rsp
        push    rbx
void foo<float>(float):
        push    rbp
        mov     rbp, rsp
        push    rbx
```

PART I: ADDING TEMPLATES TO EXISTING CODE

- Partial specialization of a class

```cpp
template<typename T, typename U>
struct FullType {
    FullType() {
        cout << "FullType CTOR" << '\n';
    };
};

template<typename T>
struct FullType<T,T> {
    FullType() {
        cout << "FullType same type CTOR" << '\n';
    };
};

int main()
{
    FullType<int, bool> i;          // FullType CTOR
    FullType<int, int> j;           // FullType same type CTOR
}
```

```asm
.LC0:
        .string "FullType CTOR   "
FullType<int, bool>::FullType() [base object
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
.LC0:
        .string "FullType same type CTOR   "
FullType<int, int>::FullType() [base object
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
```

PART I: ADDING TEMPLATES TO EXISTING CODE

- Partial specialization of a function

```cpp
template <typename T>
struct Wrapper {
    Wrapper(T t_): t(t_) {
        cout << "Wrapper: " << typeid(T).name() << '\n';
    };
    T t;
};

template <typename T>
void Foo(T t) {
    cout << "Foo: " << typeid(T).name() << '\n';
}

template <typename T>
void Foo(Wrapper<T> t) {
    cout << "Foo with wrapper: " << typeid(T).name() << '\n';
}

int main()
{
    int i; foo(i);                    // Foo: i
    Wrapper<int> j; foo(j);           // Foo with wrapper: i
}
```

```asm
.LC0:
        .string "Foo: "
void foo<int>(int):
        push    rbp
        mov     rbp, rsp
        push    rbx
.LC1:
        .string "Foo with wrapper: "
void foo<int>(Wrapper<int>):
        push    rbp
        mov     rbp, rsp
        push    rbx
```

PART II: OVERLOAD RESOLUTION AND ADL

PART II: OVERLOAD RESOLUTION AND ADL

Resolution of overloaded template functions:

1. The compiler creates a set of candidates (all functions f that can be accessed from where the call is)
2. The compiler creates a subset of viable functions (according to the number of parameters):
    1. Name Lookup (NL)
    2. Partial Ordering of Function Templates (POFT)
    3. Template Argument Deduction (ADL)
3. The best viable function is the one whose parameters all have either better or equal-ranked implicit conversion sequences.

Notes:

- Member functions (excluding constructors) are treated as if they had an extra parameter appear before the first actual parameter.

PART II: OVERLOAD RESOLUTION AND ADL

Overload resolution includes the following processes and rules:

1. Name Lookup (NL)

   All the functions with a suitable name are associated to the function call.

2. Partial Ordering of Function Templates (POFT)

   Choosing a function template is done according to the most specialized version.

3. Template Argument Deduction (ADL)

   Including the namespace of the arguments in addition to the function call scope.

Notes:

- Ambiguity will be created when there is <u>more than one viable function</u> (compiler error).
- <u>We will not get an error</u> if no candidate function is found (Substitution Failure Is Not An Error).
- <u>We will get an error</u> if we try to <u>call a function</u> with no viable candidates.

PART III: CONDITIONS AT COMPILE TIME PRE C++20

PART III: CONDITIONS AT COMPILE TIME PRE C++20

<type_traits>:

- type_traits are part of <types> library.
- Verified at compile time.
- A class type trait (since C++11):
  - Identifies whether a condition is true or false.
  - Every type trait class has a value member.
  - The member can be retrieved by
    - (C++11) trait_name<A,B,...>::value
    - (C++17) trait_name_v<A,B,...>
- A function type trait (since C++20):
  - A constexpr function.
  - Identifies a situation.

PART III: CONDITIONS AT COMPILE TIME PRE C++20

<type_traits>: overview

- Classes:
  - Queries
    - Helper classes (is constant)
    - Primary type categories (void, class, etc.)
    - Composite type categories (reference, etc.)
    - Type properties (CV, final, aggregate, etc.)
  - Query supported operations (trivially_assignable, etc.)
  - Query properties (alignment_of, rank, extent, etc.)
  - Type relationships (is_same, is_nothrow_convertible, etc.)
  - Operations
    - Const volatility (remove_cv, add_cv, etc.)
    - References (remove_reference, etc.)
    - Pointers (remove_pointer, add_pointer)
    - Sign (make_signed, make_unsigned)
    - Arrays (remove_extent, remove_all_extent)
  - Transformation related functionality
  - Operations on traits (conjunction, disjunction, negation)

- Functions (C++20):
  - Member relationships (is_corresponding_member, is_pointer_interconvertible_with_class)
  - Constant evaluation context (is_constant_evaluated)

PART III: CONDITIONS AT COMPILE TIME PRE C++20

Two main types of conditions exist

1. Conditions using bool

```
template<typename T, typename U>
bool IsSame = std::is_same_v<T, U>;

int main()
{
    cout << "IsSame: " << IsSame<int, double>;    // IsSame: 0
}
```

2. Conditions using function

```
template<typename T, typename U>
void IsSame() {
    cout << "IsSame: "<< std::is_same_v<T,U> << '\n';
};

int main()
{
    IsSame<int, bool>();                                // IsSame: 0
}
```

PART III: CONDITIONS AT COMPILE TIME PRE C++20

Function conditions using overloads and enable_if (C++11):

```
template <class T>                                       // Trivial types
std::enable_if_t<std::is_trivial<T>, void>::value        // optional
PrintTriviality(T t) {
    cout << "Trivial case: " << typeid(t).name() <<'\n';
}

template <class T>                                       // Non trivial types
std::enable_if_t<!std::is_trivial<T>, void>::value       // optional
PrintTriviality(T t) {
    cout << "Non Trivial case: " << typeid(t).name() <<'\n';
}
```

Function conditions using constexpr if (C++17):

```
template <typename T>
void PrintTriviality(T t) {
    if constexpr (std::is_trivial_v<T>)  {              // Trivial types
        cout << "Trivial case: " << typeid(t).name() <<'\n';
    } else {                                            // Non Trivial types
        cout << "Non Trivial case: " << typeid(t).name() <<'\n';
    }
}
```

PART III: CONDITIONS AT COMPILE TIME PRE C++20

<type_traits>: (C++17) conjunction, disjunction, negation

std::conjunction (&&)

```
template<typename T, typename... Ts>
bool IsAllSame() {
    cout << "conjunction: "<< std::conjunction_v<std::is_same<T,Ts>...> << '\n';
    return std::conjunction_v<std::is_same<T,Ts>...>;
};

int main()
{
    IsAllSame<int, int, int>();          // conjunction: 1
}
```

```
template<typename T, typename... Ts>
bool IsAllSame(T, Ts...) {
    cout << "conjunction: " << std::conjunction<std::is_same<T,Ts>...> << '\n';
    return std::conjunction<std::is_same<T,Ts>...>;
};

int main()
{
    int i; int j;
    IsAllSame<>(i, j);                   // conjunction: 1
}
```

PART III: CONDITIONS AT COMPILE TIME PRE C++20

<type_traits>: conjunction, disjunction, negation

std::disjunction (||)

```
template<typename T, typename... Ts>
bool IsTwoSame() {
    cout << "disjunction: "<< std::disjunction_v<std::is_same<T,Ts>...> << '\n';
    return std::disjunction_v<std::is_same<T,Ts>...>;
};

int main()
{
    IsTwoSame<int, string, bool>();           // disjunction: 0
}
```

std::negation (!)

```
template<typename T, typename U>
bool IsNotSame() {
    cout << "!IsSame: "<< std::negation_v<std::is_same<T,U>> << '\n';
    return std::negation_v<std::is_same<T,U>>;
};

int main()
{
    IsNotSame<int, int>();                    // IsNotSame: 0
}
```

- Using std:: versions over (&&, ||, !) will optimize by avoiding instantiation of parameters pack if condition is met.

PART IV: CONDITIONS AT COMPILE TIME USING C++20

PART IV: CONDITIONS AT COMPILE TIME USING C++20

- C++20 created a paradigm shift in metaprogramming.
- Concepts and constraints allow the user to introduce a new 'type category' (vs. type) to the compiler.

- In the following slides, we will review:
  - C++20's new type-traits.
  - New syntax for controlling the overload set.
  - New tools to use in C++20 template metaprogramming, and beyond.

PART IV: CONDITIONS AT COMPILE TIME USING C++20

- Classes:
  - Type properties:
    - is_bounded_array<A>;
    - is_unbounded_array<A>;
  - Type relationships:
    - is_nothrow_convertible<A, B>;
    - is_layout_compatible<A,B>;
    - is_pointer_interconvertible_base_of<A,B>;

- Functions:
  - Member relationships:
    - is_pointer_interconvertible_with_class<A,B>
    - is_corresponding_member<A, B, C, D>
  - Constant evaluation context:
    - is_constant_evaluated

- Transformations related functionality:
  - remove_cvref<A>;

PART IV: CONDITIONS AT COMPILE TIME USING C++20

is_layout_compatible<A,B>;

is_pointer_interconvertible_base_of<A,B>;

is_pointer_interconvertible_with_class<A,B>

is_corresponding_member<A, B, C, D>

is_constant_evaluated();

```
constexpr int foo(int a) {
    if (is_constant_evaluated()) {
        cout << “Evaluated at compile time!" << '\n';
    }
}
```

- Provides greater reflection abilities!

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Concepts:

- Concepts are a way for the developer to define restrictions on the "type category".

```
template <typename T>
concept concept_name = (condition);
```

- The expression is not evaluated, only validated.
- The condition can be:
  - A simple condition
  - Requires expression, which will return true if valid, false otherwise

```
template <typename T>
concept concept_name = requires(T t) { valid_expression; };
```

  - Requires expression, which describes requirements on the return value of a function (compound):

```
template <typename T>
concept concept_name = requires(T t) {
    {valid_expression} -> condition_on_return_type;
};
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Concepts:

- Concepts can be used to identify whether a type belongs to the "type category" *

```
template <typename T>
concept concept_name = (condition);
```

```
template <typename T>
concept trivial = std::is_trivial_v<T>;

int main()
{
    cout << "Trivial: " << trivial<int> << '\n';      // Trivial: 1
}
```

- A concept cannot be constrained (for example, by using another concept to restrict its params).
- A concept cannot recursively refer to itself.

* A popular way do declare a concept:

```
template <typename... Ts>
constexpr auto concept_name = (condition);
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Concepts:

Core language concepts:

- same_as
- derived_from
- convertible_to
- common_reference_with
- common_with
- integral
- signed_integral
- unsigned_integral
- floating_point
- assignable_from
- swappable
- swappable_with
- destructible
- constructible_from
- default_initializable
- move_constructible
- copy_constructible

Comparison concepts:

- boolean-testable
- equality_comparable
- equality_comparable_with
- totally_ordered
- totally_ordered_with

Object concepts:

- movable
- copyable
- semiregular
- regular

Callable concepts:

- invocable
- regular_invocable
- predicate
- relation
- equivalence_relation
- strict_weak_order

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Constraints:

- Constraints are used to restrict the types which can instantiate the template

```
template <typename T>
void foo() requires integral<T> {
    // do something requires integral types
}
```

- Constraints can also be defined in a form that resembles enable_if

```
template <typename T>
  requires integral<T>
void foo() {
    // do something requires integral types
}
```

```
int main()
{
    foo<int>();               // works
    foo<string>();            // Compilation error: no matching function found
}
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Constraints:

- Constraints express additional data on the types
- Constraints can be used to restrict:
  - Template parameters
  - Variables
- Constraints normalization is the process of ‘unpacking’ the constraints
- Constraints can be added to the code as follows:
  1. In the declaration of the template parameters.
  2. Using requires clause after the template parameter list.
  3. Using concept on the placeholder type in an abbreviated function template declaration.
  4. Using trailing requires clause.

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Constraints:

Evaluation will be in the following order, with conjunction between the expressions

1. In the declaration of the template parameters

```
template <integral T>
void foo(T t) {
    // do something with T
}
```

2. Using requires clause after the template parameter list

```
template <typename T>
 requires integral<T>
void foo(T t) {
    // do something with T
}
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Constraints:

3. Using concept on the placeholder type in an abbreviated function template declaration.

```
void foo(some_concept auto T) {
    // do something with T
}
```

```
void foo(some_concept auto... T) {
    // do something with T
}
```

4. Using requires clause after the template parameter list.

```
template <typename T>
void foo(T t) requires integral<T> {
    // do something with T
}
```

* Remember this list, we will go back to that later.

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Real world use case: wrapper type

- The “type category” can be used in a broader way than the type.
    - Concept: contains_data

```
template <class T>
concept contains_data = requires(T t)
{
    t.data;
};
```

- DataType which satisfies that concept:

```
template <typename T>
struct DataType                                    // satisfies contains_data
{
    DataType(T t) : data(t) {
        cout << "DataType created" << '\n';
    };
    T data;
};
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Real world use case: wrapper type

- A wrapper accepts only types which satisfy contains_data

```
template <typename...>
class wrapper
{
public:
    template <typename T>
    wrapper(T t) = delete;

    template <contains_data T>      // create from any contains_data type
    wrapper(T t) {
        cout << "wrapper from any contains_data, with data: " << t.data << '\n';
    };
};
```

- Wrap1 contains DataType<int>, Wrap2 fails.

```
int main()
{
    wrapper<> Wrap1{DataType{3}};       // wrapper <DataType>
    wrapper<> Wrap2{3};                 // error: use of deleted function
}
```

PART IV: CONDITIONS AT COMPILE TIME USING C++20

Explicit circuit braking:

- P2199R0: Concepts to Differentiate Types / Isabella Muerte

  Not in the standard yet, suggests concepts which can solve the problem of recursive evaluation.

```
template <class T, class U>
  concept different_from = not same_as<T, U>;


template <class T, class U>
   concept similar_to = same_as<remove_cvref_t<T>, remove_cvref_t<U>>;


template <class T, class U>
  concept distinct_from = not similar_to<T, U>;
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Lazy evaluation:

- Temporaries are created and returned from an operation.
- We can reduce the instantiation of temps, by avoiding their creation.
- As a result, we can improve the compile time of our program, by avoiding unnecessary overhead.

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Expression templates

```
template<typename T, typename InternalVec = std::vector<T>>
struct VecWrapper {
    VecWrapper(size_t n);                                       // CTOR VecWrapper
    VecWrapper(const InternalVec& other);                       // CCTOR VecWrapper

    template<typename T1, typename V1>
    VecWrapper& operator=(const VecWrapper<T1, V1>& other);     // VecWrapper op=
    template<typename T1, typename V1>
    VecWrapper operator+(const VecWrapper<T1, V1>& other);      // VecWrapper op+
    (...)                                                       // VecWrapper op[]
    InternalVec vec;
};
```

```
int main()
{
    VecWrapper<int> x(3);                                       // CTOR VecWrapper
    VecWrapper<int> y(3);
    x[0] = x[1] = x[2] = 0;                                     // op[]
    y = y + x;                                                  // op+
}
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Expression templates

```
template<typename T, typename InternalVec = std::vector<T>>
struct VecWrapper {
    VecWrapper(size_t n);                                          // CTOR VecWrapper
    VecWrapper(const InternalVec& other);                          // CCTOR VecWrapper

    template<typename T1, typename V1>
    VecWrapper& operator=(const VecWrapper<T1, V1>& other);        // VecWrapper op=
    template<typename T1, typename V1>
    VecWrapper operator+(const VecWrapper<T1, V1>& other);         // VecWrapper op+
    (...)                                                          // VecWrapper op[]
    InternalVec vec;
};
```

```
template<typename T, typename A, typename B>
VecWrapper<T, Sum<T, A, B>> operator+ (const VecWrapper<T, A>& a, const VecWrapper<T, B>& b){
    cout << "Operator+ of VecWrapper" << '\n';
    return VecWrapper <T, Sum<T, A, B>>(Sum<T, A, B>(a.vec, b.vec));
}
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Expression templates

```
template<typename T, typename A , typename B>                    // Takes T
struct Sum{                                                      // Proxy
  Sum(const A& a_, const B& b_): a(a_), b(b_) {
        cout << "CTOR Sum" << '\n';
  }

  T operator[](const std::size_t i) const {                     // Lazy +
    return a[i] + b[i];
  }

    const A& a;                                                  // By ref
    const B& b;
};
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Expression templates

```
int main()
{
    VecWrapper<int> x(3);
    VecWrapper<int> y(3);
    VecWrapper<int> z(3);

    x[0] = x[1] = x[2] = 0;
    y[0] = y[1] = y[2] = 1;
    z[0] = z[1] = z[2] = 2;

    VecWrapper<int> sum(3);
    sum = x + y + z;                          // Lazy evaluation
}
```

```
Operator+ of VecWrapper
CTOR Sum
CCTOR VecWrapper
Operator+ of VecWrapper
CTOR Sum
CCTOR VecWrapper
Op= VecWrapper
Op= VecWrapper
Op= VecWrapper
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Ranges (But not just yet…)

- Ranges were added to C++20
- Input range adaptors, along with zip, zip_with, stayed out.
- You can experiment with it by adding Eric Niebler's range-v3 library.

std::plus< >::operator( )

```
template< class T, class U>
constexpr auto operator()( T&& lhs, U&& rhs ) const
-> decltype(std::forward<T>(lhs) + std::forward<U>(rhs));
```

Using zip_with for lazy evaluation

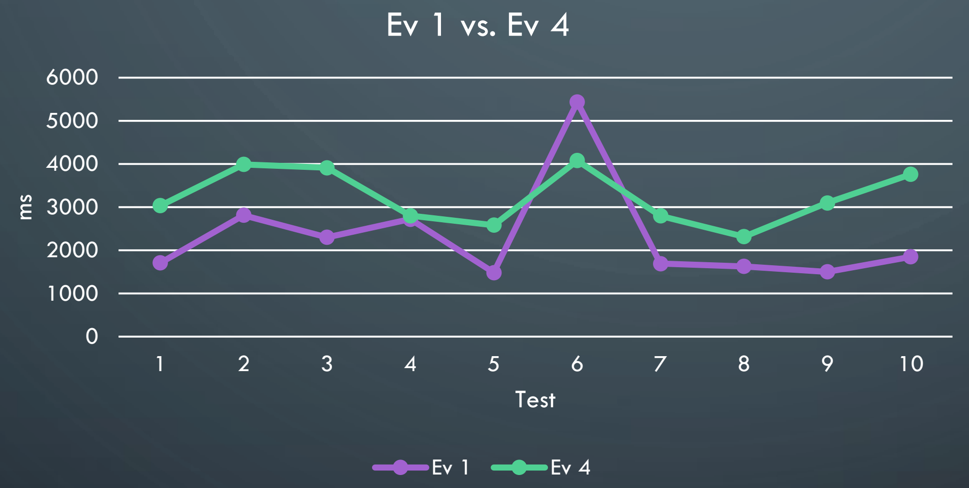
```
int main()
{
    vector<int> vec1{0, 0, 0};
    vector<int> vec2{1, 1, 1};
    vector<int> vec3{2, 2, 2};
    auto sum = ranges::view::zip_with(plus<>{}, ranges::view::zip_with(plus<>{}, ve1, vec2), vec3);
    cout << "Sum: " << sum;                                    // Sum: [3,3,3]
}
```

PART V: ADVANCED METHODS FOR COMPILE TIME LOGIC

Constraints evaluation

```
template <typename T>                               // Ev 4
    wrapper(T t) requires contains_data<T> {};

template <contains_data T>                          // Ev 1
    wrapper(T t){};
```



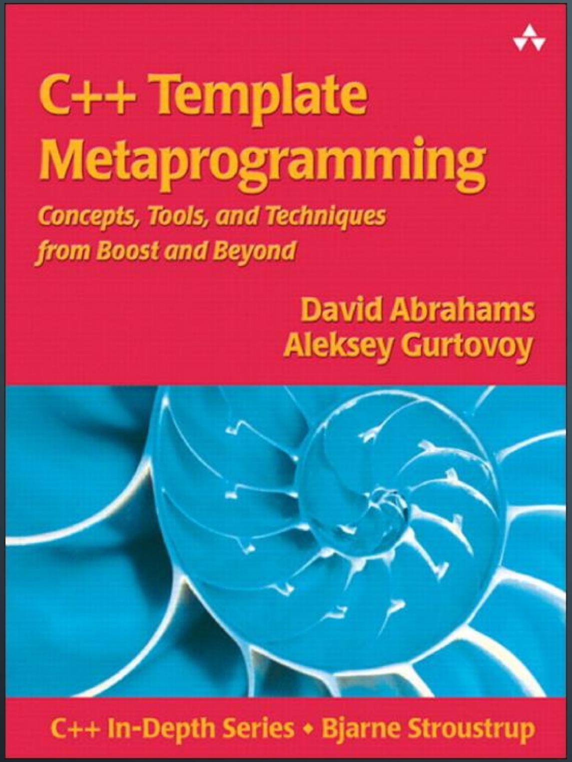
Gcc (trunk)
x86-64
-std=c++2a
-stdlib=libc++
-O3

```
template <braking_condition T>
    wrapper(T t)requires additional_condition<T> {};
```

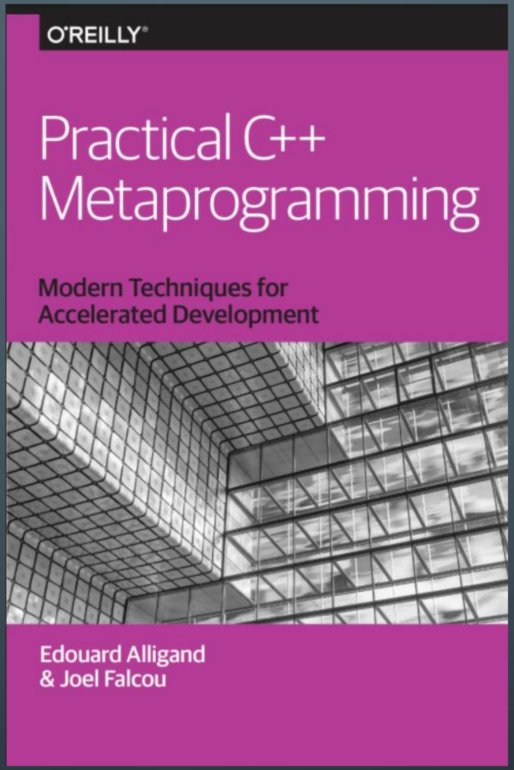
TAKEAWAYS

- Template metaprogramming is a powerful tool.
- C++20 extends the vocabulary for template metaprogramming.
- C++20's Concepts & constraints are a step towards increasing readability of template code.
- Some more improvements are expected in the following years.
- C++20 added powerful compile-time tools, use them.
- Let's strive to extend the usage of templates!

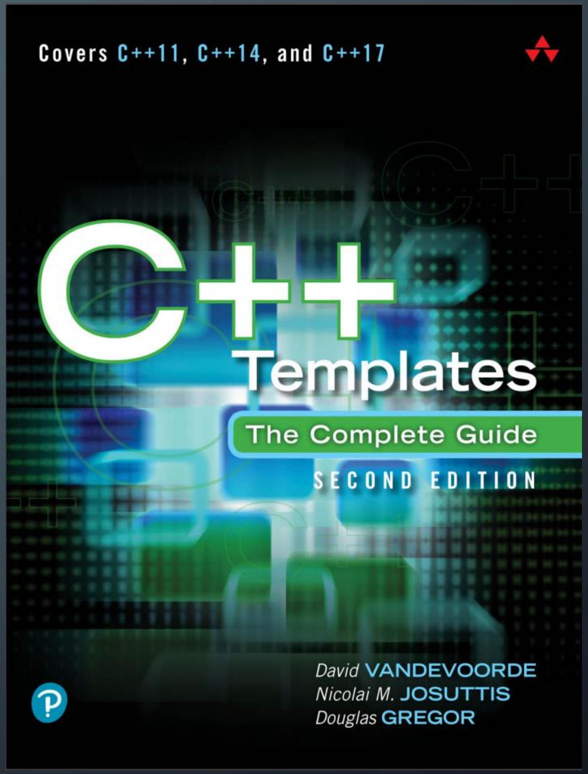
BOOKS



C++ Template Metaprogramming
Concepts, Tools, and Techniques from Boost and Beyond
David Abrahams
Aleksey Gurtovoy
C++ In-Depth Series • Bjarne Stroustrup



O'REILLY
Practical C++ Metaprogramming
Modern Techniques for Accelerated Development
Edouard Alligand & Joel Falcou



Covers C++11, C++14, and C++17
C++ Templates
The Complete Guide
SECOND EDITION
David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR

LINKS

- CppCon talks:
  - Template Metaprogramming: Type Traits / Jody Hagins
  - Back to Basics: Templates / Andreas Fertig
  - Constructing Generic Algorithms: Principles and Practice / Ben Deane
  - How C++20 Changes the Way We Write Code / Timur Doumler
  - CppCon 2019: Modern Template Techniques / Jon Kalb: https://www.youtube.com/watch?v=MLV4IVc4SwI
  - CppCon 2019: Range Algorithms, Views and Actions: A Comprehensive Guide / Dvir Yitzchaki: https://www.youtube.com/watch?v=qQtS50ZChN8
  - MeetingC++ 2014: Expression Templates Revisited / Klaus Iglberger: https://www.youtube.com/watch?v=hfn0BVOegac
- Technical data:
  - P0734R0: Working Draft, C++ extensions for Concepts / Andrew Sutton (2017): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf
  - Partial ordering of function templates / Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/partial-ordering-of-function-templates-cpp?view=vs-2019
  - Guidelines For snake_case Concept Naming / Jonathan Müller: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1851r0.html
  - Requires-expression / Andrzej Krzemienski: https://akrzemi1.wordpress.com/2020/01/29/requires-expression/
  - Cpp Insights: https://cppinsights.io/
- Papers:
  - P2199R0: Concepts to Differentiate Types / Isabella Muerte: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2199r0.html
  - P1035R4: Input range adaptors / Christopher D. B., Casey C., Corentin J.: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1035r4.html#zip_with-in-c20

THANK YOU!

I hope you're now inspired to go and explore C++20 templates ☺

Inbal Levi

Mail:	sinbal2l@gmail.com
Twitter:	@Inbal_l
Linkdin:	/inballevi
Github:	/inbal2l