ŁUKASZ GUT

Compile-time programming

AGENDA

1. Introduction to template metaprogramming
2. STL and <type_traits> header
3. New specifier: constexpr and it's implications
4. Novelties from C++20
5. Compile-time std::vector

INTRODUCTION TO TMP

WHAT IS (TEMPLATE) METAPROGRAMMING?

"**Metaprogramming** - a programming technique in which computer programs have the ability to treat other programs as their data." [Wikipedia]

Template metaprogramming - a C++ programming technique that uses template instantiation to drive compile-time evaluation.

MOTIVATION?

PERFORMANCE (OPTIMIZATIONS OFF)

Method	Time
Classic pow(x, n)	14.48 [s]
Metaprogramming pow<x, n>	0.77 [s]

For x = 17, n = 64. Repeated 100,000,000 times. Compiled with g++8.3.0 -O0.

PERFORMANCE (OPTIMIZATIONS ON)

Method	Time
Classic pow(x, n)	0.73 [s]
Metaprogramming pow<x, n>	0.68 [s]

For x = 17, n = 64. Repeated 100,000,000 times. Compiled with g++8.3.0 -Ofast.

MOTIVATION

- Performance (not so much anymore with template metaprogramming, constexpr is the new king)
- Type traits

TEMPLATE INSTANTIATION

When we use the name of a template where a **function**, **type** or **variable** is expected, the compiler instantiates the expected entity from that template.

```
template<typename T>
void f(T param) {}

int main() {
  f<int>(7);
}
```

SHIFTING WORK TO COMPILE-TIME

Compile-time absolute value **metafunction**.

```cpp
template<int X>
struct absolute {
  static_assert(X != INT_MIN);
  // initializing value, NOT assigning
  static const int value = (X < 0) ? -X : X;
};

int main() {
  // metafunction calls
  const int val0 = absolute<-5>::value;
  const int val1 = absolute<7>::value;

  static_assert(val0 == 5);
  static_assert(val1 == 7);
}
```

Code snippet

COMPILE-TIME RECURSION VIA SPECIALIZATION

Compile-time pow **metafunction**.

Primary template:

```
template<unsigned X, unsigned N>
struct pow {
    static const unsigned value = X * pow<X, N - 1>::value;
};
```

Partial specialization:

```
template<unsigned X>
struct pow<X, 0> {
    static const unsigned value = 1;
};
```

Code snippet

TYPE AS A METAFUNCTION'S PARAMETER

Compile-time array dimension checker **metafunction**.

Primary template:

```
template<typename T>
struct rank {
    static const std::size_t value = 0u;
};
```

Partial specialization for bounded array type:

```
template<typename T, std::size_t N>
struct rank<T[N]> {
    static const std::size_t value = 1u + rank<T>::value;
};
```

Partial specialization for non-bounded array type:

```
template<typename T>
struct rank<T[]> {
    static const std::size_t value = 1u + rank<T>::value;
};
```

Code snippet

TYPE AS A METAFUNCTION'S RETURN VALUE

Another great thing about metafunctions is that they can also return **type**.

```
template<typename T>
struct type_is {
    using type = T;
};
```

Code snippet

TYPE AS A METAFUNCTION'S RETURN VALUE

Remove const qualifier **metafunction**.
It doesn't really remove the const, it returns the same type but **without** the const.

```
template<typename T>
struct remove_const {
    using type = T;
};

template<typename T>
struct remove_const<const T> {
    using type = T;
};
```

The same can be done for volatile.

Code snippet

TYPE AS A METAFUNCTION'S RETURN VALUE

Combining the two we can remove both const-volatile qualifiers.

```cpp
template<typename T>
struct remove_cv {
    using type =
    typename remove_const<
      typename remove_volatile<T>::type
    >::type;
};
```

Code snippet

MAKE YOUR LIFE EASIER #1

C++ standard library calling syntax.

```
remove_cv<T>::type;
remove_cv_t<T>; // C++14 and above
```

Previous example:

```
std::is_same<int, remove_cv<const volatile int>::type>::value
std::is_same_v<int, remove_cv_t<const volatile int>>
```

Code snippet

MAKE YOUR LIFE EASIER #2

Let's define a simple identity **metafunction** (credits to Walter E. Brown).

```cpp
template<typename T>
struct type_is {
    using type = T;
};

template<typename T>
using type_is_t = typename type_is<T>::type;
```

Code snippet

REIMPLEMENTING REMOVE_CONST<T>

We can use our identity **metafunction** as a base class to make our lives even easier.

```
template<typename T>
struct remove_const : type_is<T> {};

template<typename T>
struct remove_const<const T> : type_is<T> {};
```

Code snippet

COMPILE-TIME DECISION MAKING

Let's implement a compile-time **if statement**! It will allow us to write self-configuring code.

Possible use cases:

```
conditional<(val < 0), int, unsigned> i;
conditional<(sizeof(T) < 4), StackBox<T>, HeapBox<T>> box;
conditional<(val < 0), Fun1, Fun2>{}(); // Instantiate correct function
```

COMPILE-TIME DECISION MAKING

The implementation is trivial. Please notice that certain template arguments do **not** have names!

```
template<bool, typename T1, typename>
struct conditional : type_is<T1> {};

template<typename T1, typename T2>
struct conditional<false, T1, T2> : type_is<T2> {};

template<bool B, typename T1, typename T2>
using conditional_t = typename conditional<B, T1, T2>::type;
```

Code snippet SBO Code snippet

SFINAE

"**Substitution failure is not an error (SFINAE)** - refers to a situation in C++ where an invalid substitution of template parameters is not in itself an error. David Vandevoorde first introduced the acronym SFINAE to describe related programming techniques." [Wikipedia]

In simpler terms: compiler has a right to reject template-instantiated code that would be **ill formed (not compile)** for a given type.

SFINAE EXAMPLE

Let's take a look at SFINAE in action.

```
struct S {
  using type = double;
  inline static const type value = 7.0;
};

template<typename T>
typename T::type fun(const T& obj) { return T::value; }

double fun(int val) { return val; }

int main() {
    fun(S());
    fun(0); // Why no errors? Clearly int::value is ill formed...
}
```

OVERLOAD RESOLUTION SET AND SFINAE

To understand SFINAE we need to understand how the compiler produces an **overload resolution set (a list of viable functions)**.

1. Perform a name lookup
2. For templates the template argument values are deduced from the types of the actual arguments passed in to the function
   - All occurrences of the template parameter (in the return type and parameters types) are substituted with those deduced types.
   - When this process leads to invalid type (like int::value) the particular function/class is removed from the overload resolution set. (SFINAE)
3. At the end we have a list of viable functions that can be used for the specific call. If this set is empty, then the compilation fails. If more than one function is chosen, we have an ambiguity. In general, the candidate function, whose parameters match the arguments most closely is the one that is called.

SFINAE IN USE

Imagine having a way to provide different function implementations for types T1 and type T2 but having a compile-time error when invoked with different type. Let's implement enable_if!

```
template<bool, typename T = void>
struct enable_if : type_is<T> {};

template<typename T>
struct enable_if<false, T> {};
```

Code snippet

SFINAE - WHY DO WE CARE?

Using SFINAE and to be precise tricks like enable_if allow us to decide EXACTLY which templates should be discarded and which should be not.

This is super important if we don't want templates to be a "catch-all" tool and still have them provide us with all the goodness of generic code.

enable_if is basically a way for us to use C++20 concepts in a very ugly way.

ENABLE_IF IN REAL WORLD SCENERIO

Real use case of enable_if.

```
template <class T, enable_if_t<std::is_integral_v<T>>* = nullptr>
void fun(const T& t) {
    // an implementation for integral types (int, char, unsigned, etc.)
}

template <class T, enable_if_t<std::is_class_v<T>>* = nullptr>
void fun(const T& t) {
    // an implementation for class types
}
```

Code snippet

IS_CLASS AS A HARDCORE USE OF SFINAE

Let's take a look and embrace the implementation of is_class.

```
namespace detail {
  template <class T> char test(int T::*);
  struct two { char c[2]; };
  template <class T> two test(...);
}

template <class T>
struct is_class : std::integral_constant<bool,
    sizeof(detail::test<T>(0))==1 &&
    !std::is_union<T>::value>
{};
```

Code snippet

METAFUNCTIONS CONVENTIONS SUMMARY #1

1. When a metafunction returns value make it **static const(expr)** and name it **value**.

```
template<int X>
struct absolute {
    static_assert(X != INT_MIN);
    static constexpr int value = (X < 0) ? -X : X;
};
```

2. When a metafunction returns a type use an **alias** and name it **type**.

```
template<typename T>
struct type_is {
    using type = T;
};
```

METAFUNCTIONS CONVENTIONS SUMMARY #2

3. Provide additional aliases for your classes to make it more user friendly.

```
template<typename T>
using type_is_t = typename type_is<T>::type;
```

4. Make use of inheritance.

```
template<typename T>
struct remove_const : type_is<T> {};

template<typename T>
struct remove_const<const T> : type_is<T> {};
```

5. ...?

STL AND <TYPE_TRAITS>

STL AND <TYPE_TRAITS>

There is a lot of metafunctions defined in **<type_traits>** header.

CONSTEXPR

CONSTEXPR

constexpr - specifies that the value of a variable or function can appear in constant expressions.

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at **compile time**.

Such variables and functions can then be used where only **compile time constant expressions** are allowed.

CONSTEXPR

constexpr specifier can be applied to either a variable or a function.

```
constexpr auto x = 0;
constexpr auto fun() {}
```

CONSTEXPR VARIABLE REQUIREMENTS

A constexpr variable must satisfy the following requirements:

- its type must be LiteralType.
- it must be immediately initialized.
- the full-expression of its initialization, including all implicit conversions, constructors calls, etc, must be a constant expression.

CONSTEXPR FUNCTION REQUIREMENTS

A **constexpr function** must satisfy the following requirements:

- it must not be virtual (until C++20).
- its return type must be LiteralType.
- each of its parameters must be a LiteralType.
- there exists at least one set of argument values such that an invocation of the function could be an evaluated subexpression of a core constant expression (for constructors, use in a constant initializer is sufficient) (since C++14). No diagnostic is required for a violation of this bullet. In simpler terms: There **must** exist one valid set of arguments (the empty set is also a valid set).

CONSTEXPR FUNCTION REQUIREMENTS

The function body must not contain:

- a goto statement
- a statement with a label other than case and default
- a try-block (until C++20)
- an asm declaration (until C++20)
- a definition of a variable of non-literal type
- a definition of a variable of static or thread storage duration
- a definition of a variable for which no initialization is performed. A function body that is =default; or =delete; contains none of the above.

ENOUGH

WHY DO WE CARE?

It makes compile-time programming much easier to reason about - even for beginners!

```cpp
// Old-school C++
template<int X, int N>
struct pow { static const int value = X * pow<X, N - 1>::value; };

template<int X>
struct pow<X, 0> { static const int value = 1; };

// New-school C++
constexpr auto pow_fun(int v, int n) {
    auto result = 1;
    for(int i = 0; i < n; i++)
        result *= v;
    return result;
}
```

Code snippet

IF CONSTEXPR

Since C++17 we have a new statement available: **if constexpr**.

In a constexpr if statement, the value of condition **must** be a contextually converted constant expression of type bool.

If the value is true, then statement-false is discarded (if present), otherwise, statement-true is discarded.

The return statements in a discarded statement do not participate in function return type deduction!

IF CONSTEXPR IS AWESOME

This tool is super useful, primarily because of two reasons:

1. It lets us write cleaner code when we want to "control the flow" of the program at compile time.
2. It lets us write functions that return different types!

IF CONSTEXPR SYNTAX

The syntax of **if constexpr** is really simple.

```
if constexpr(statement) {
    // ...
} else if (statement) {
    // ...
} else {
    // ...
}
```

CONTROLLING THE FLOW AT COMPILE TIME

Imagine writing a function that would behave differently when provided different types.

You could use such technique to e.g. provide an optimized implementation for certain types that posses certain properties.

Before C++14 you had to rely on partial template specialization. Now you can use **if constexpr**.

Code snippet

RETURNING DIFFERENT TYPES FROM FUNCTIONS.

Because the return statements in a discarded statement do not participate in function return type deduction we can use this to write functions that return different types.

```
template<typename T>
auto get_cont(const T& obj) {
    if constexpr(sizeof(obj) < sizeof(int)) {
        return std::pair{obj, obj};
    } else {
        return std::vector{obj, obj};
    }
}
```

Code snippet

CONSTEXPR SUMMARY

1. Constexpr is a super powerful and easy to use tool (especially in C++17/C++20).
2. Constexpr makes compile-time programming much easier and more programmer-friendly.
3. Constexpr allows us to optimize code without sacrificing on readability.

My recommendation for friday evening

NOVELTIES FROM C++20

NOVELTIES FROM C++20

C++20 introduces a lot of new toys and improvements over the older standards.

1. Concepts.
2. constexpr <algorithm> header (and a few constexpr algorithms from <numeric> header... hopefully :)).
3. constexpr virtual methods.
4. constexpr std::vector and constexpr std::string.
5. constexpr new operator.

CONCEPTS SNEAK PEAK.

Concepts allow us to put **easy to reason about** constraints on template parameters.

```
// C++20
template<typename T>
concept Integral = std::is_integral<T>::value;

Integral auto gcd(Integral auto a, Integral auto b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}

// C++17
template<typename T, std::enable_if_t<std::is_integral_v<T>>* = nullptr>
auto gcd_old(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Code snippet

COMPILE-TIME STD::VECTOR

CONSTEXPR NEW AND COMPILE-TIME ALLOCATION

In C++20 **most probably** new operator will be constexpr. Currently, this functionality is implemented only in the experimental versions of GCC and Clang.

It allows compile-time dynamic allocation.

```cpp
constexpr auto fun() {
  auto ptr = new int(5);
  // do something with ptr
  delete ptr;
}
```

Allocated memory in constexpr context **must** be deleted before transitioning to runtime context!

Code snippet

CONSTEXPR ALLOCATOR

Using constexpr new we can implement allocators that work at compile-time.

```
template<typename T>
class ConstexprNewAllocator {
    public:
        using value_type = T;
        using pointer = T*;
        using const_pointer = const T*;
        using size_type = std::size_t;
    public:
        [[nodiscard]] constexpr pointer allocate(size_type n);
        constexpr void deallocate(pointer p, size_type n);
};
```

CONSTEXPR ALLOCATOR IMPLEMENTATION

The most trivial allocator you can imagine.

```
[[nodiscard]] constexpr pointer allocate(size_type n) {
    return new value_type[n]{};
}

constexpr void deallocate(pointer p, size_type n) {
    delete[] p;
}
```

CONSTEXPR STD::VECTOR IMPLEMENTATION

Using our custom allocator we can implement std::vector that works at compile-time.

Code snippet

BIBLIOGRAPHY

- cppreference.com
- bfilipek.com
- Effective Modern C++ by Scott Meyers
- CppCon 2014: Walter E. Brown "Modern Template Metaprogramming: A Compendium, Part I/II"

THANK YOU!