Seminar "C++ Tomorrow"
Summer Term 2017

Template Metaprogramming

Sascha Sauermann
Technische Universität München

08.06.2017

Abstract

Template metaprogramming (TMP) provides a way to use the compiler to generate code or perform computations at compile time. While TMP was possible in C++03 by using tools like the *Boost MPL* library, it was hard to use, requiring in-depth knowledge. This paper describes recent additions to the language standard, like *variadic templates*, *fold expressions* and the *constexpr-if* and how they simplified the application of TMP.

1 Introduction

Templates in C++ not only allows to implement classes or functions that take generic parameters but they can be used to perform compile-time computations. Templates in C++ provide generics by generating code, i.e. instantiating the templated class or function with the specified types. As the templating system is Touring-complete, it can also be used to do functional programming which is discussed in another paper of this seminar.

In C++03 templates were added to the language and libraries like Boost MPL were written to provide required functionality for template metaprogramming. However, Boost MPL requires deep knowledge of the library and the templating engine to write code [5]. We will therefore ignore the existence of external libraries for this paper.

Let us look at the limits of the C++03 standard in regards to templates. The code in figure 1 shows the implementation of a method `cartesian` that applies another function[1] `f` to each pair combination in the Cartesian product of its parameters. As the method should be type-safe templates are used so that there is no need to define the method for each type of arguments separately. But the main problem of the code is, that we still have to duplicate the function for different parameter counts as we have no type-safe way to define a function that can take an arbitrary amount of parameters.

Libraries that needed functions that support a varying number of parameters had to emulate this behavior. Either the preprocessor had been used to generate the required code, or a (hopefully) sufficient amount of overloads were written by hand — for example up to 10 parameters. Code written that way gets hard to maintain as much code is duplicated and increases the compile time [4].

With the addition of *variadic templates* in C++11, we got a means to define arbitrary parameter amounts by using parameter packs and process them. Fold expressions in C++17 provided more and easier ways to do calculations with parameter packs and the constexpr-if allows taking different execution paths for some types without the need for overloading a templated function.

[1] For simplicity's sake, the function f is fixed, but it can easily be replaced with a functor. See figure 13 in the Appendix.

```cpp
#include <iostream>
template<typename A, typename B>
void f(A a, B b) const {
    // Do sth useful in f
    std::cout << "(" << a << ", " << b
    ↪  << ")" << std::endl;
}
template<typename T>
void cartesian(T t){
    f(t,t);
}
template<typename T1, typename T2>
void cartesian(T1 t1, T2 t2){
    f(t1,t1);    f(t1,t2);
    f(t2,t1);    f(t2,t2);
}
int main(){
    cartesian("Hello");
    cartesian("Hello", "World");
    // Next one does not work as it has
    ↪  no matching template
    // cartesian("Hello", "World", 42);
}
```

Figure 1: The Cartesian product for varying parameter counts without variadic templates

Related Work

Multiple programming languages have a feature for type-safe generic methods and classes. C++ has the templating system for this purpose. Other languages like Java, C#, Scala, Haskell or F# have similar features, like generics. Although all of those languages support a variable number of function arguments with the same (super-) type, none of them supports multiple arguments with different types. For example, in Java, the use of a *VarArg* parameter of the common type of all objects — *Object* — accepts all types but does not preserve them, and is therefore not type-safe[4].
The language D, that is very similar to C++, does provide a syntax for variadic function arguments though [3, 4].
In some languages, C++11 and up included, tuples provide an alternative for such parameters. Though, most of the time there is no (type-safe) way to iterate over the content of a tuple — especially not at compile time — or tuples may be restricted to a maximum amount of elements.
We will show the uses of **variadic templates** and other additions to the language standard that improved the application of TMP such as the introduction of **fold expressions** and **constexpr-if**.

2 Basics

2.1 Template instantiation and type deduction

When a template with a specified type is used, the compiler instantiates the template with this type. So a template like `template<typename T> T foo(T t) {return t;}` is compiled to `int foo(int t){return t;}` when used like `foo<int>(5);`. The compiler replaces all occurrences of the type arguments with the specified types. This instantiation is never done more than once per type for each function or class [6, §17.8.2.1] but not yet evaluated templates can result in further and even recursive instantiations.
Including the type argument in the function call is not always mandatory as the compiler can deduce the missing arguments. So calling `foo(5)` will create the same instantiation as `foo<int>(5)`. Since C++17 this also works with class templates by inferring the types based on the constructor arguments.
When specified and deduced template parameters are mixed the specified arguments are substituted before the deduction process begins. The specified arguments must fit the given parameters though [6, §17.8.2].

2.2 Overload resolution

When the compiler resolves a function call, it has first to do a name lookup where it checks if functions with matching arguments exist. For templated functions, the compiler tests if the inferred type can be substituted for the template parame-

ters. If the replacement fails, the method is not further considered [6, §6.4].

If multiple so-called *candidate functions* exist, the overload resolution begins.

The compiler selects the function that parameter types match the given parameters best. In other words, which requires the least type conversions to match. This selection decision can lead to surprising effects, especially when the function has a template overload. As template parameters are deduced and therefore always match perfectly, a non-template overload might not be selected if one type conversion is required [6, §17.8.3].

Non variadic templates (see section 3) are preferred if available.

```cpp
#include <iostream>
void overload(double t){
    std::cout << "Double: " << t <<
    ↪  "\n";
}

template<typename T>
void overload(T t){
    std::cout << "Template: " << t <<
    ↪  "\n";
}

int main(){
    overload(2.5f);
    // Expected: "Double: 2.5"
    // Output: "Template: 2.5"
}
```

Figure 2: When calling overloaded functions the template version is preferred to functions that require type conversion

The code in figure 2 shows this problem. We have a function that takes a `double` and a template that takes anything. When we call the function with a `float` parameter, the compiler selects the template version as no overload with a `float` parameter exists, and the `double` overload requires a type conversion.

3 Variadic Templates

3.1 Problems and solutions before C++11

Until C++11 the standard only allowed a finite amount of template parameters. That leads to problems when writing class or function templates that could take an arbitrary amount of arguments. The simplest solution is to add code for each number of arguments explicitly. For the example of the Cartesian product, this looks like figure 1. Such an implementation approach gets large and hard to maintain very fast.

Another possible solution is to use nested pairs like `pair<int, pair<bool, double> >` as the parameter, but they are difficult to iterate — especially without methods like std::get which were also added in C++11.

If all types are the same, then std::vector or other collections provide an appropriate structure for the parameters.

If no type safety is required, there is the ellipsis operator from the cstdarg header[6, §21.10.1].

3.2 Syntax and functionality

Since C++11 variadic templates are available as a new language extension. They provide a syntax for defining an arbitrary count of template parameters — a so-called *template parameter pack*. It is defined by placing the ellipsis operator `...` between the type and the name, like `template<typename ... T> class foo`. This does also work with non-type and template template parameters.

Another type of parameter pack is the *function parameter pack*. It is used additionally, when defining a template function, and provides the syntax for multiple function arguments like `template <typename ... T> void bar(T ... t)`.

Template parameter packs have exactly one use - they can be expanded. This requires a pattern and an ellipse. The compiler applies the pattern to each parameter of the pack and concatenates all those expressions with commas. If T is the pa-

rameter pack, then a pattern $P(T)$ is expanded to $P(T_1), P(T_2), \cdots, P(T_n)$. Simultaneous expansion of the same parameter pack in a pattern like $P(T, T)$ to $P(T_1, T_1), P(T_2, T_2) \cdots$ and nested expansions are possible. These expansions work for template and function parameter packs, but only in specific contexts [6, §17.5.3]. We focus on three common ones below.

3.3 Recursive applications

The simplest use of a parameter pack is to expand it into another template. The main reason to do this is the recursive handling of all parameters similar to functional programming - with a head element and a tail of arbitrary length.
The code in figure 3 recursively calculates the mean of any number of parameters by using function overloading to define the base case for just one element and expanding the function template pack into the recursive call. To get the length of the tail, we use the special `sizeof...` function on the template parameter pack (here: `T`) or function parameter pack (here: `tail`).

```
template <typename H>
double mean(H head){
    return head;
}

template <typename H, typename ... T>
double mean(H head, T ... tail){
    unsigned tailLength = sizeof...(T);
    double sum =  head + tailLength *
    ↪  mean(tail...);
    return sum / (tailLength + 1.0);
}
```

Figure 3: Recursive calculation of the average with variadic templates

3.4 Mixin classes

One feature that C++ lacks is inheriting constructors from base classes. When inheriting from a class, the programmer has to redefine all constructors of the parent class and possibly add some parameters. Then the additional parameters can be processed by the new class, and all the other ones get forwarded to the base class by explicit enumeration in the call to the base constructor [4].
When creating mixins, i.e. classes which base classes get defined by their user — in C++ by template classes, this inconvenience gets problematic. As we don't know the base class when writing the mixin class, we can only call the default constructor of the parent.
Variadic templates provide a solution for this problem. We have to do two things to define our mixin class:

- The class itself needs a template parameter that we will use as the base class.
- The constructor is a templated function that takes its own arguments first and then a constant reference to a parameter pack which we will forward to the base class by expanding it.

The code is shown in figure 4, but this solution has a little problem, as described by Douglas Gregor and Jaakko Järvi:

> "not all argument types can be forwarded cleanly through const references: non-constant temporaries of primitive type, for example, are rejected, and non-constant lvalues will be forwarded to the Base constructor as constant lvalues"[4]

The proposed solution is to use rvalue references `&&` and `std::forward`, that are new features in C++11 as well. The required changes only affect the constructor[2] that now looks like this:

```
template <typename ... Args>
Named(string name, Args&& ... args) :
↪  Name(name),
↪  Base(std::forward<Args>(args)...){};
```

[2]The utility header has to be included as well.

```cpp
#include <string>
#include <iostream>
using std::string;

struct Point{
    double X, Y;
    Point(double x, double y) : X(x),
    ↪  Y(y) {};
};

template <typename Base>
struct Named : Base{
    string Name;
    template <typename ... Args>
    Named(string name, const Args& ...
    ↪  args) : Name(name),
    ↪  Base(args...) {};
};

int main(){
    Named<Point> np("Name", 1.2, 4.4);
    std::cout << np.Name << " : " <<
    ↪  np.X << ", " << np.Y << "\n";
}
```

Figure 4: Mixin class 'Named' with variadic templates and parameter forwarding and some base class 'Point'

There are more possibilities to create mixins with multiple classes without having to nest them into each other like with this approach. One that should be noted here is to prepare our point class to take a template parameter pack and use all of them as base classes as C++ got multiple inheritance. Then forward instances of those classes to their copy constructor and the following gets possible [4].

```cpp
Point<Named, Colored>(2.0, 3.0,
↪  Named("Name"), Colored("green"));
```

3.5 Side effects with initializer lists

The last use we want to present is expanding the template parameter pack into an initializer list as another way to apply a function to every parameter. As the elements of an initializer list must not be `void` or have different types, comma expressions can be used to ensure that each entry in the initializer list is an integer and the program will always compile[8]. Such initializer list has the following form, given that `args` is the parameter pack and `(f(args), 0)` the pattern: `auto x = {(f(args), 0) ...};`
As the compiler correctly complains that x is unused, we tell the compiler that it can get rid of x by casting the initializer list to `void`. A slightly different syntax is required for this to work. Finally, we additionally cast the return argument of our function `f` to `void` to preclude an overloaded comma operator on the return type from breaking our comma expression[8] — and therefore maybe the integer type of the initializer list:

```cpp
(void) std::initializer_list<int>{
↪  ((void)f(args), 0) ... };
```

We can now look at an implementation of the Cartesian product iteration discussed earlier now with variadic templates. The code in figure 5 has a helper function that iterates all pairs for a fixed first element as the expanded initializer list executes like $f(t, args_0), f(t, args_1)$, etc.
The method `cartesian` itself contains two nested expansions. First the inner `args...` is expanded to a simple comma separated list of all arguments, to just pass them on to the helper function. The outer ellipse now only expands the pattern with the former `args` as the latter is already replaced. Therefore the helper is called once for each first argument.

4 Fold Expressions

4.1 Problems and solutions before C++17

As seen in the last chapter we can use either recursion 3.3 or tricks with initializer lists 3.5 to calculate something with the parameter pack of a variadic template or apply a function to all values of the pack.

```cpp
template<typename T, typename ... Args>
void helper(T t, Args... args){
    (void) std::initializer_list<int>{
        ( (void) f(t,args), 0 ) ...
    };
}

template<typename ... Args>
void cartesian(Args... args){
    (void) std::initializer_list<int>{
        ( helper(args, args...), 0 ) ...
    };
}
```

Figure 5: The Cartesian product for arbitrary parameter counts with variadic templates and initializer lists

Recursion can get quite complex with different specializations for unpacking packs and already a simple fold requires two methods. The initializer list tricks never produce nice code as we have to define the initializer because we can not expand a function parameter pack to a comma expression without a specific context. In the end, we even throw away the initializer list by casting it to void, so the compiler will never create it.

4.2 Syntax and functionality

With C++17 *fold expressions* were added to the language. They are used to *fold* parameter packs over an operator, for example, to calculate a sum of numbers by folding them over the + operator.
There are 32 supported operators for fold expressions as shown in this table taken from the C++ standard[6, §8.1.6]:

```
+   -   *   /   %   ^   &   |   <<  >>
+=  -=  *=  /=  %=  ^=  &=  |=  <<= >>= =
==  !=  <   >   <=  >=  &&  ||  ,   .*  ->*
```

Not all of them have an intuitive expansion. For example, comparison operators as < do not allow to compare multiple values, like `a < b < c` does not either because the boolean result of the first comparison gets promoted to an integer with value `0` for `false` or `1` for `true`. Therefore `1 < 1 < 1` results in `true`.
A *binary fold expression*[6, §8.1.6] can have two different forms, where `init` is some initial value, `op` is an operator, `...` is again the ellipsis operator and `args` is the parameter pack or any pattern including the pack. Both operators must be the same one.

- Binary left fold `(init op ... op args)`
- Binary right fold `(args op ... op init)`

When using the left fold, the pack expansion is done from first/left to last/right. On the contrary, the right fold expands from the last to the first element. If the parameter pack is empty the `init` value is returned. When for example, the expression `(v || ... || args)` is expanded it results in $(((v \mid\mid args_0) \mid\mid args_1) \mid\mid \cdots)$ and will even be short-circuit evaluated when `||` or `&&` are used as operator.
When no `init` value is required, it can be omitted, including the neighboring operator, to get an *unary fold expression*[6, §8.1.6].

- Unary left fold `(... op args)`
- Unary right fold `(args op ...)`

They expand as the binary ones do, except when the argument pack is empty. Then only some operators will provide a default value. These are `||` with a default of `false`, `&&` with `true` and the comma operator ',' with `void()` [6, §17.5.3]. Any fold expression with another operator will not compile with an empty parameter pack.

4.3 Removal of template recursion

In section 3.3 the mean of any number of parameters is calculated with variadic templates and recursive expansion. See figure 3 for the previous code example. With fold expressions, we can now remove the recursion and the template specialization.
In the code in figure 6 we use an unary fold expression over the plus operator to calculate the sum.

The `sizeof...` operator is still a valid option to get the number of parameters, but for demonstration purposes, another fold expression is included that uses a comma expression as the pattern instead of just `args`. More complicated scenarios could require the utilization of an inline-if as the pattern and so on.

```
template<typename ... Args>
double mean(Args ... args){
    double sum = (... + args);
    unsigned length = (... +
    ↪  ((void)args, 1));
    return sum / length;
}
```

Figure 6: Calculation of the average with variadic templates and fold expressions

4.4 Replacement for initializer list trickery

Fold expressions finally allow to expand parameter packs without a special environment by folding over the comma operator. This removes the need for such trickery as seen in section 3.5 for the calculation of the Cartesian product. We can keep the code structure from figure 5 nearly intact and just replace the initializer list by the mentioned fold expression which removes the unnecessary clutter (See figure 7).

```
template<typename T, typename ... Args>
void helper(T t, Args... args){
    (f(t, args),...);
}

template<typename ... Args>
void cartesian(Args... args){
    (helper(args, args...),...);
}
```

Figure 7: The Cartesian product for arbitrary parameter counts with variadic templates and fold expressions

The code is now much smaller and clearer, but it is a pity that we still need the helper method. With lambda functions it can be moved into the other function and forwarding the parameter pack is no longer necessary (See figure 8). The pack needs expansion into the capture of the lambda function tough, so the arguments can be accessed in it. If we shorten the code even more by removing the local variable (See Appendix A, figure 12) we get code that is not only hard to read but even only works in the current version of Clang and not in GCC[3].

```
template<typename ... Args>
void cartesian(Args... args){
    auto helper = [args...](auto t){
        (f(t,args),...);
    };
    (helper(args),...);
}
```

Figure 8: The Cartesian product for arbitrary parameter counts with variadic templates, fold expressions and lambda functions.

5 Constexpr-if

5.1 Problems and solutions before C++17

The compiler selects the function overload with the least amount of type conversions required for the given function parameters (See section 2.2). Therefore the overload that the compiler selects is not always the one the programmer expects or wants. One solution for this is to overload the templated function for every native type, etc. But this results in code duplication. We just need a way to select different overloads for groups of types, for example, one for all integral types and one for all floating point types. There are multiple solutions for selecting overloads based on traits of the template

[3] As of writing the newest version of gcc is 7.1. In this version the execution of cartesian(1,2) as defined in Appendix produces "(1,1) (1,2) (2,0) (2,0)" as output instead of the expected "(1,1) (1,2) (2,1) (2,2)".

parameter, like *tag dispatching* or *SFINAE* (Short for: "substitution failure is not an error"). We focus on the latter.
SFINAE uses the fact that the compiler does not add overloads to the set of candidate functions (See section 2.2 on overload resolution) that produce ill-formed code when the types are substituted into the function declaration[4], instead of producing an *hard-error* [1].
This behavior can be used to write template arguments which substitution only succeeds when the parameter type has a particular trait. Such traits are compile-time constants or type definitions, and many are available in the std namespace via the `type_traits`[6, §23.15.2] header. Also available is `enable_if` which uses SFINAE to enable overloads only for some traits.

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

template <typename T>
struct enable_if<true, T> {
  typedef T type;
};
```

Figure 9: The enable if struct as it is defined in the standard library[6, §23.15.7.6]

In figure 9 the code from `enable_if` can be found. The struct has two parameters, a boolean and a type. Through the specialization, the typedef for `type` is only available if the boolean parameter is true, else the non-specialized template is initialized and the member `type` is not available.
Instead of using a template parameter directly we can pass it through this struct only if a condition is met. This results in ill-formed code when no 'type' member exists — in the false case, and the overload is discarded. As nested types are in a non-deduced context[6, §17.8.2.5] type deduction does not work here, so it is preferable to use any non-generic type of the function declaration for the `enable_if` struct. For example the return type as in figure 10, where two overloads for integral and for floating point types are defined. Note that any other type like string produces a compile error as no matching function exists.

```
#include <iostream>
#include <type_traits>
using namespace std;

template <typename T>
typename
 ↪  enable_if<is_integral<T>::value,
 ↪  void>::type print(T t)
{
    std::cout << t << " is integral\n";
}

template <typename T>
typename
 ↪  enable_if<is_floating_point<T>::value,
 ↪  void>::type print(T t)
{
    std::cout << t << " is floating
     ↪  point\n";
}

int main(){
      print(2); // is integral
      print(4.4); // is floating point
      //print("Hello"); // 'No matching
       ↪  function can be found' error
}
```

Figure 10: Overloaded function for parameters of integral and floating point types

5.2 Syntax and functionality

With the release of the C++17 standard, C++ finally got its static if — called constexpr-if, fitting the naming of the constexpr feature added in C++11 that for example allows executing functions at com-

[4]The standard states that the error has to occur in the *immediate context* of the function or template types to be a substitution failure. This immediate context basically means that the error must not occur in the function body.

pile time. With the static if it is possible to select one of many branches at compile time and completely discard all other[6, §9.4.1], similar to the preprocessor directives `#if` or `#ifdef`.
To define an if as static, just the token `constexpr` has to be added between if and the conditional like `if constexpr (cond)`. The conditional naturally has to be constant at compile time for this expression to be valid. As another restriction even not taken paths have to be well-formed code as the compiler has to parse them to find the end of the if block[7]. Additionally, static asserts are still executed in all branches and must be true for a successful compilation[2].

5.3 Replacement for SFINAE

Instead of using cumbersome SFINAE code for enabling function overloads for some parameter types the constexpr-if provides the tools needed to create one method that is capable of handling all cases by using a straightforward and easy readable syntax.
We can simplify the SFINAE example from figure 10 to the code in figure 11 by using the static if to decide based on the type trait of the parameter. It demonstrates a disadvantage of the static if as well. We can't limit our function to particular types[2]. While the SFINAE print function only supports numbers — integral or floating point — the new print function can be called with arbitrary parameters as the printing is defined for all types. If we still want to limit the parameters we have to add a static assert that will fail with other types than the supported.

```cpp
#include <iostream>
#include <type_traits>
using namespace std;

template <typename T>
void print(T t){
    if constexpr(is_integral<T>::value){
        std::cout << t << " is integral\n";

    } else if constexpr (is_floating_point<T>::value){
        std::cout << t << " is floating point\n";
    }

    static_assert(is_integral<T>::value || is_floating_point<T>::value);
}

int main(){
      print(2);
      print(4.4);
      //print("Hello"); // Would be valid without the assert
}
```

Figure 11: Function with different paths for parameters of integral and floating point types by using constexpr-if

5.4 Replacement for template specialization as recursion base cases

Template recursions with variadic templates like discussed in section 3.3 need a template specialization to define the base case, requiring the programmer to split the recursion logic into two functions.
With the constexpr-if this logic can be bundled into the same method by choosing another branch for the base case [2]. This can not be done with a typical if as the template instantiation is done at compile time and recursive templates are instantiated even if the path is not taken at runtime which results in trying to instantiate non existing templates or a instantiation loop for templates without arguments.

In simple recursions which just reduces the number of parameters in each step, the base case can be defined by `if constexpr (sizeof...(Args) == 0)` or something similar.

6 Summary

As shown in this paper, variadic templates are a useful addition for template metaprogramming that allowed creating templates for arbitrary parameter amounts by defining parameter packs and expanding them into comma-separated lists in some contexts. This expansion can be used to do recursive calculations, mixin classes or with some tricks with initializer lists even non-recursive function application to each pack member.
With C++17 *fold expressions* and *constexpr-if* were added. The first allows calculations with the complete pack by folding it over an operator similar to folding in functional programming languages. This removes the need for template recursion in some cases. By folding over the comma operator, the trick with initializer lists is no longer necessary.
Constexpr-if removes the need for template specialization to define recursion base cases. Additionally, it replaces the need for complex code when restricting function overloads to specific type traits by allowing the programmer to match for characteristics without the usage of *enable if* and SFINAE.

7 Conclusion and future work

The additions to the language standard improved the usability of template metaprogramming. Pre C++11, template metaprogramming was only viable when using external libraries like the Boost MPL [5].
This changed with the addition of variadic templates, parameter forwarding, *type traits* and *enable if* in C++11. The syntax is quite cumbersome in some places, but the functionality was there. The addition of the *constexpr* keyword allowed to replace template metaprogramming for some scenarios altogether as functions declared with this keyword can be executed at compile time.
With C++17 and the addition of fold expression and the static if, template metaprogramming got far easier to use with readable and maintainable code that does not require specific knowledge about how the compiler deduces types or handles overload resolution.
Interesting topics for further reading include the usage of template metaprogramming for compile-time functional programming or type calculations. Another one is the usage of concepts to restrict the allowed types of template parameters — maybe as a replacement to SFINAE. Another question is if there are still uses for Boost MPL or if the recent additions to the standard already rendered it obsolete.

References

[1] Eli Bendersky. Sfinae and enable_if. `http://eli.thegreenplace.net/2014/sfinae-and-enable_if/`. Accessed: 2017-05-22.

[2] Simon Brand. Simplifying templates and #ifdefs with if constexpr. `https://blog.tartanllama.xyz/c++/2016/12/12/if-constexpr/`. Accessed: 2017-05-22.

[3] D Language Foundation. Variadic templates. `https://dlang.org/variadic-function-templates.html`. Accessed: 2017-05-22.

[4] Douglas Gregor and Jaakko Järvi. Variadic templates for c++0x. *Journal of Object Technology*, 7(2):31–51, February 2008. OOPS Track at the 22nd ACM Symposium on Applied Computing, SAC 2007.

[5] Aleksey Gurtovoy and David Abrahams. The boost c++ metaprogramming library. `http://www.boost.org/doc/libs/1_31_0/libs/mpl/doc/paper/mpl_paper.html`. Accessed: 2017-05-25.

[6] ISO/IEC. Standard for programming language c++ [working draft]. Document N4659, International Organization for Standardization, Geneva, Switzerland, 2017.

[7] Arne Mertz. Constexpr additions in c++17. `https://arne-mertz.de/2017/03/`

constexpr-additions-c17/#Constexpr_if. Accessed: 2017-05-22.

[8] Arne Mertz. More about variadic templates. https://arne-mertz.de/2016/11/more-variadic-templates/. Accessed: 2017-05-22.

A Appendix

```cpp
#include <iostream>

template <typename A, typename B>
void f(A a, B b){
    std::cout << "(" << a << "," << b << ")\n";
};

// This works in gcc7
template <typename... Args>
void print(Args const&... args){
    auto func = [args...](auto x) {(f(x, args),...);};
    (func(args),...);
};

// This doesn't work in gcc7 but it does in clang4
template <typename... Args>
void print2(Args const&... args){
    (([args...](auto x) {(f(x, args),...);})(args),...);
};

int main(){
    print(1,2,3);
    std::cout << "\n";
    print2(1,2,3);
}
```

Figure 12: The Cartesian product for arbitrary parameter counts with variadic templates, fold expressions and lambda functions. This code produces a bug in GCC 7.1.

```cpp
#include <iostream>

struct F{
    template<typename A, typename B>
    void operator()(A a, B b) const {
        // Do sth useful in f
        std::cout << "(" << a << ", " << b << ")" << std::endl;
    }
};

template<typename F, typename T>
void cartesianProduct(F f, T t){
    f(t,t);
}

template<typename F, typename T1, typename T2>
void cartesianProduct(F f, T1 t1, T2 t2){
    f(t1,t1);    f(t1,t2);
    f(t2,t1);    f(t2,t2);
}

template<typename F, typename T1, typename T2, typename T3>
void cartesianProduct(F f, T1 t1, T2 t2, T3 t3){
    f(t1,t1);    f(t1,t2);    f(t1,t3);
    f(t2,t1);    f(t2,t2);    f(t2,t3);
    f(t3,t1);    f(t3,t2);    f(t3,t3);
}

int main(){
    F f;
    cartesianProduct(f, "Hello");
    cartesianProduct(f, "Hello", "World");
    cartesianProduct(f, "Hello", "World", 42);
}
```

Figure 13: The Cartesian product for varying parameter counts without variadic templates. Instead of a fixed function f this version does use a functor as parameter.