TECHNISCHE UNIVERSITÄT MÜNCHEN

C++17: Variadic Templates and Fold Expressions

Andrei Cosmin Aprodu
May 27, 2020

ABSTRACT

In this report we discuss the fundamentals of Variadic Templates and of their followup, Fold Expressions. In the first half, we define Variadic Templates, enumerate their properties and study their syntax and underlying methods of implementation through some examples. In the second half of the report, we transition to Fold Expressions, study their differences with the previously discussed templates and go deeper into their means of implementation.

CONTENTS

1 INTRODUCTION

The possibility to implement a function with an arbitrary number of parameters has been known since 1989, when *C89* proposed the concept named *variadic function*. They were designed to utilize a special symbol, named ellipsis, that would house all the optional arguments of that function. These days, the ellipsis retains its importance, being also the foundation for newer concepts, like *Variadic Functions*, introduced in *C++11* and *Fold Expressions*, introduced in *C++17*.

1.1 C-STYLE VARIADIC FUNCTIONS

Before analyzing the aforementioned notions, let us discuss the original variadic functions, namely *C89 variadic functions*. A classical example, based on this concept, is the function `printf`, used in the C programming language to date, that can print as many arguments as specified.

```
int printf(const char *format, ...);   // (...) is called ellipsis
```

The implementation is built around macros, such as `va_start`, `va_arg` and `va_end`, being very low-level. Moreover, the function is entirely evaluated at runtime, although the details about the optional arguments are known at compile time. Consequently, there is little to no information about the parameters housed in the ellipsis, hence the developer needs to know beforehand their types, their number, and so on. In case of a type mismatch, a runtime error will occur, as these functions are also type-unsafe, statement voiced by Bjarne Stroustrup [4], the creator of the C++ programming language.

Consider the following example where we declare the function `sum_stdarg` that calculates the sum of its arguments:

```
1  int sum_stdarg(int n_args, ...) {
2      int s = 0;
3      va_list ap;
4      va_start(ap, n_args);
5      for (int i = 0; i < n_args; i++) {
6          s += va_arg(ap, int);   // arguments expected to be int
7      }
8      va_end(ap);
9      return s;
10 }
```

As one can see, the implementation is rather extensive for the reduced functionality of the method. In this case, calling it with a parameter of type different than `int` will produce an unexpected and wrong final result, as it differs from the type already specified in the macro `var_arg`. Although *C99* brought some improvements to this topic, they were minor and did not resolve any of the problems that came with the original variant. A major change emerged in 2011, when *Variadic Templates* were proposed [7].

2 Variadic Templates

2.1 Overview

Introduced with *C++11*, *Variadic Templates* are an extension of regular templates and an important addition to generic programming. They are heavily based on recursion and inherit the ellipsis concept from *C-style variadics*, named *parameter pack*. Furthermore, these templates provide a method to create not only template functions, but also template classes, the area of usability being much more extensive. Unlike in previous versions of the C programming language, the major let-downs of the older implementation are solved as well. Type-safety is now a feature, non-POD[1] types finally being supported. Different optimizations are also found in this newer version of the programming language. For example, the template generation and the unpacking of the parameter packs, both take place at compile time and the syntax is more robust and easier to understand.

As mentioned earlier, Variadic Templates are suited for both function and class declaration. Let us begin analyzing the basic structure of a function developed in this scope:

```
1 template<typename... Args>        // Args is a template parameter pack
2 void my_function(Args... args) {  // args is a function parameter pack
3     .....
4 }
```

We begin with the `template` keyword, just like declaring a regular template, however a difference arises when specifying the type. In order to declare that `Args` is a *template parameter pack*, we are required to add the ellipsis after the keyword `typename`[2]. Regarding the signature of the function, notice the presence of the ellipsis after the type name to indicate that `args` is not a single parameter, but rather a *function parameter pack* that houses an arbitrary number of arguments.

The class declaration is almost identical. Consider the following example:

```
1 template<typename... Bases>       // Bases is a template parameter pack
2 class Master: public Bases... {  // Master inherits: Base1, Base2, ...
3     .....
4 };
```

Differences appear when specifying the classes to be inherited. In this case, `public Bases...` would expand into every class included in the pack, that is `public Base1, public Base2, public Base3` and so forth.

[1] Plain-Old-Data types do not contain any constructors, destructors, or virtual member functions.
[2] In the context of templates, the keyword can be substituted with `class` at any time.

2.2 Parameter Packs

Until now, we have continuously specified the term *parameter pack*, without giving a proper definition. In what follows, we will define the term and underline its importance in the context of Variadic Templates. Then we will formally state what Variadic Templates are. First and foremost, this concept is split into different categories.

1. A *template parameter pack* is a template that accepts zero or more template arguments, that is: types, non-types or even other templates [3].
2. A *function parameter pack*, on the other hand, is a function parameter that accepts zero or more function arguments [3].

Definition *A template with at least one parameter pack is called a variadic template [8].*

From now on, unless otherwise stated, we will call a *function parameter pack* simply *parameter pack* or just *pack*. Since the evaluation of the parameter packs is bound to compile time, it is possible to find out at each moment during runtime, how many parameters there are in a specific pack. This can be achieved using a modified version of the well-known function `sizeof`.

```
sizeof...(args);        // args is a function parameter pack
```

Regarding the actual function call, the syntax is straightforward, the only difference from a regular call being the addition of the ellipsis, to the right of the parameter pack's name. Its role is to inform the compiler to expand the pack into single arguments. It is important to mention that, in the event of passing an empty parameter pack, the expansion will return nothing and the function will be called with no arguments. Consider the following:

```
my_function(args...);   // my_function(arg1, arg2, ...)
```

2.3 Examples

We present a first trivial, yet important example. Consider the function `show_params` that is able to print everything that can be printed with `std::cout`. Our function must work for an undetermined number of arguments, including 0 arguments. We begin by declaring the function, then study its means of implementation:

```
1 void show_params() {}             // recursion base case
2
3 template<typename T, typename... Args>
4 void show_params(T current, Args... rest) {
5     std::cout << current << ' ';  // print first argument
6     show_params(rest...);         // expand pack
7 }
```

Starting with the signature, we notice right away that there is a second type name `T` declared besides the template parameter pack. We will state its purpose shortly. The method in itself only admits two parameters, one of type `T` and a function parameter pack of type `Args...`. Moving on to the actual implementation, we observe that, being a recursive-based approach, a base case is required. For the sake of simplicity, we leave it empty. Regarding the recursion step, we first print the argument of type `T`, followed by a space character, then we call the function recursively, making sure to specify the ellipsis in order to expand the parameter pack. On the second function call, because the pack was expanded, the argument of type `T` will be the first parameter in the old pack, the new one having one element less. This idea of parameter extraction is essential for Variadic Template usage.

In the next example we declare a function `sum_templates` that calculates the sum of its arbitrarily many arguments. A comparison to the C-style variadic functions can also be made, as the functionality is identical. Let us first define the function:

```
1 int sum_templates() {
2     return 0;
3 }
4
5 template<typename T, typename... Ts>
6 int sum_templates(T arg1, Ts... args) {
7     return arg1 + sum_templates(args...);
8 }
```

The strategy is similar. As before, we notice the second type name `T` that plays the role of the current extracted argument. The base case now returns 0, as we are dealing with a method that returns a concrete value. In the recursion step, the addition operator is applied to the argument of type `T` and to the recursion call that passes on the remaining arguments by expanding the pack each time.

2.4 Further optimizations

Although Variadic Templates greatly improve the idea behind the ellipsis concept and introduce more flexibility regarding function declaration, they still lack optimizations from the compile time point of view. Additionally, recursion remains mandatory when working with the aforementioned expressions. For that matter, in 2017 further optimizations were established in the name of *Fold Expressions*.

3 Fold Expressions

3.1 Overview

Introduced in *C++17*, *Fold Expressions* were designed as an extension and an overall improvement to the *Variadic Templates*. On one hand, they are a textual simplification of the templates, conferring the same benefits with an even more robust syntax. On the other hand, they offer more flexibility by folding a binary operator over the parameters of a pack and come with new compile time optimizations, also retaining the advantages of the former-discussed concepts. A formal, but simplified definition is stated in the Fold Expressions proposal published in 2014:

Definition *A Fold Expression is an expression that allows parameter packs to be expanded as a sequence of operators [6].*

Before we dive into the theory of Fold Expressions, let us evaluate the following example of a function `sum_fold` that calculates the sum of its parameters.

```
1 template<typename... Ts>
2 int sum_fold(Ts... args) {
3     return (... + args);   // unary left fold
4 }
```

We can immediately observe the new and concise syntax and the lack of recursion. Since the compiler has knowledge over the means of unpacking a parameter pack and applying a binary operator to its arguments, a recursion step is no longer required. Note that the open and closing brackets are part of the Fold Expression [9].

3.2 Folding types

We have previously mentioned that Fold Expressions benefit from a flexible implementation by folding over a binary operator. In total, there are four folding types available, in other words, there are four possibilities of applying a binary operator to the parameters of a given pack. Let `args` be a function parameter pack with $N \in \mathbb{N}_0$ parameters and `op` be a valid binary operator.

3.2.1 Unary left fold

Unary left folds are implemented by specifying the ellipsis on the left, followed by the binary operator and the name of the parameter pack on the right.

```
(... op args);          // ((arg1 op arg2) op arg3) op ....
```

The expansion process begins with the first two parameters, to which the operator is applied. The result is taken alongside the third parameter and the operator is applied for a second time. This process continues until all parameters are evaluated. A visual representation can be found to the right of the expression mentioned above.

3.2.2 Unary right fold

Similarly to the syntax of the unary left fold, the *unary right fold* requires the same elements for its implementation. A difference arises in the order in which the parameters are evaluated, although the order relative to one another remains unchanged.

```
(args op ...);          // arg1 op (.... op (argN-1 op argN))
```

The compiler begins the process with the last two parameters, applies the operator, then goes backwards and applies the operator a second time to the result and to the $(N-2)$th argument. The execution continues in this manner and stops when the first parameter of the original pack is reached.

3.2.3 Binary left fold

Fold expressions also provide the possibility of inserting an initial value. Thus, a *binary left fold* expands identically to a unary left fold, but first evaluates the initial value `val`. Notice the order in which the initial value, the ellipsis and the name of the pack are specified.

```
(val op ... op args);   // (((val op arg1) op arg2) op arg3) op ....
```

3.2.4 Binary right fold

The last folding method is the *binary right fold*. As mentioned earlier, it is implemented with the help of an initial value `val` that is positioned, in this case, to the right of the expression. The expansion is identical to the one from the unary right fold, but the initial value is evaluated first.

```
(args op ... op val);   // arg1 op (.... op (argN-1 op (argN op val)))
```

Remark It is recommended to use *left folds* over the other two options when possible, as some compilers can further optimize the expansion process by calculating intermediate results while the pack is being decoded [2].

3.3 Compile time optimizations

When discussing about Fold Expressions, we stated that this concept further introduces compile time optimizations. Let us clarify this aspect by considering two past functions, `sum_templates` (see Section 2.3) and `sum_fold` (see Section 3.1). If we call `sum_templates` with N arguments using the *C++11* compiler, because it uses recursion as implementation base, there will be N functions generated from the template. If N is large enough, the process can easily get out of hand. As an example, for $N = 4$ we obtain the following signatures:

```
1 sum_templates<int>(int);
2 sum_templates<int, int>(int, int);
3 sum_templates<int, int, int>(int, int, int);
4 sum_templates<int, int, int, int>(int, int, int, int);
```

On the other hand, when using the *C++17* compiler for `sum_fold`, there is no need for recursion, as the parameter pack is automatically evaluated and unpacked with the given binary operator. It follows that only a single function is to be generated. Going back to the previous example, the only signature, when calling `sum_fold` with $N = 4$ arguments, is:

```
1 sum_fold<int, int, int, int>(int, int, int, int);
```

REMARK This result can be reproduced on a Linux machine by executing the following command: `nm <exec-name> | c++filt -t | less`.

3.4 FOLD OPERATORS

There are a number of binary operators for which folding can be defined. Besides `(.)`, `->` and `[]`, the remaining 32[3] are split into different categories. The most important ones will be discussed in detail below.

1. Logical operators: `&& ||`
   Example: `(val && ... && args);`

2. Arithmetic operators: `+ - * / %`
   Example: `(val * ... * args);`

3. Relational operators: `< <= > >= == !=`
   Example: `(val == ... == args);`

4. Bitwise operators: `<< >> ^ & |`
   Example: `(val << ... << args);`
   Another usage for *bitwise operators* is printing every parameter from a pack with the following syntax: `(std::cout << ... << args);`. Here, we initialize a binary left fold (see Subsection 3.2.3), where the initial value `val` is `std::cout` itself. The single flaw of this approach is that the parameters are printed without any separator in between.

   A *common mistake* is to make use of the same idea to insert a separator after every element, as follows: `std::cout << (args << ... << '\n');`. This implementation instantiates, nevertheless, a binary right fold (see Subsection 3.2.4) because the name of the parameter pack is now on the left. This means that the initial value is the numerical value of `'\n'`, that is 10. Instead of printing every argument on a new line, the last argument is shifted to the left by 10 positions, the next-to-last by the previous result and so on. We will shortly discover how to correct this mistake.

5. Pointer manipulation: `.* ->*`
   Example: `(base ->* ... ->* args);`

[3] `+ - * / \% ^ & | = < > << >> += -= *= /= \%= ^= &= |= <<= >>\= == != <= >= && || , .* ->*`

A notable use case for the `(->*)` operator is the traversal of a tree constructed using a structure similar to the following:

```
struct Node {
    int val;
    Node *left, *right;
};
auto left = &Node::left;            // left direction pointer
auto right = &Node::right;          // right direction pointer
```

We now define a function called `path` [2] that returns the fold expression mentioned above, then we discuss its function call and return value.

```
template<typename T, typename... Ts>
Node* path(T root, Ts... ps) {
    return (root ->* ... ->* ps);  //root ->* p1 ->* p2 ->* ....
}
```

For the template declaration we use a second type name besides the template parameter pack, in order to specify the initial value of the binary left fold. After a path is chosen, `path` takes as first argument the root node and then the pointers `left` and `right` as many times as needed. The return value is the destination node. For instance:

```
Node* destination = path(root, left, right, right, left);
```

6. The Comma operator: `(,)`
   Example: `(..., my_function(args));`
   Being one of the more powerful operators, the *Comma operator* has the ability to combine multiple statements into one. In the Fold Expression given as example above, the function `my_function` will be called for every parameter contained in `args`, or never, if the parameter pack is empty.

3.5 Examples

We continue the list of examples with an already implemented function and update its syntax to use Fold Expressions. Consider `show_params`, that can print every argument printable with `std::cout` and that was initially written using Variadic Templates. With the help of a unary left fold and a helper lambda function we obtain the same output from a method that now lacks recursion entirely and still makes use of the ellipsis. Notice that a single type name is required, since the compiler has knowledge over the means of interpreting and expanding the parameter pack.

```
template<typename... Args>
void show_params(Args... args) {
    (..., [](auto& current) {
        std::cout << current << ' ';
    }(args));   // args is unpacked, each parameter is passed as argument
}
```

Until now, we mentioned that Fold Expressions and Variadic Templates are also suited for class usage, but did not study an real example in detail. As a consequence, let us declare the following structures[4], that implement the same method `print_msg` and a master structure, that inherits the others with the help of a template parameter pack:

```
struct A {
    void print_msg() {std::cout << "A says hi!\n";}
};
struct B {
    void print_msg() {std::cout << "B says hi!\n";}
};
struct C {
    void print_msg() {std::cout << "C says hi!\n";}
};

template<typename... Bases>
struct Master: public Bases... {
    void print_msg() {(..., Bases::print_msg());}
};
```

As previously stated, the template declaration is identical among all use cases. Examining the `Master` structure itself, we first notice the ellipsis and template parameter pack replacing the name of the actual structures to be inherited. The keyword specified before the name of the pack, in this case `public`, will be applied to all, after the expansion. Moving on to the contents of `Master`, the idea is to define the means in which we can call the function `print_msg` for every inherited member. To this end, we declare, inside a function with the same name, a unary left fold based on the Comma operator. We utilize the name of the template parameter pack as if it was the name of a regular structure and we call the public function `print_msg`. During the expansion process, we obtain the desired result[5].

REMARK If one of the inherited structures does not contain the method specified in the Fold Expression, a runtime error will occur.

[4] Classes function identically, but, for the sake of simplicity, we use their counterparts that default to public.
[5] *Note:* The order of the function calls is dependent on the order in which we declare the inherited structures.

4 Concluding remarks

4.1 Problem - Simulating Fold Expressions in C++11

We ask the following question: is it possible to simulate Fold Expressions in *C++11?* We are not allowed to use neither recursion, nor a second type name for that matter. We aim to create a unary left fold and to apply the addition operator to the elements of a given parameter pack. To solve this problem, we need to devise an ad-hoc solution:

```
template<typename... Ts>
auto c11_fold(const Ts&... vs) -> typename std::common_type<Ts...>::type {
    typename std::common_type<Ts...>::type result;
    std::initializer_list<decltype(result)> {(result += vs)...};
    return result;
}
```

How and why does the aforementioned function work?

Solution. We give a broad explanation regarding the techniques used to achieve the desired result. The function first determines the most general type of all arguments in the pack and sets it as a return type. We are aware that an array, or every aggregate that behaves like one, can be initialized in *C++11* using a parameter pack. In this case, we choose `std::initializer_list` as our data type and pass the most general type available using `decltype`. Furthermore, we are able perform basic operations before the elements are inserted into the above-mentioned list. Alongside operations, variable assignments are also allowed, so we use the syntax `{(r = r + vs)...}` to initialize a list that has at the position K the sum of all elements from 0 to K. Using this knowledge, we infer that the last element of the list is the one containing the sum of all previous elements. After the list initialization is complete, this element remains stored in the variable `result`, so we just return that variable. □

4.2 Final conclusions

After studying the basics presented in this report, we realize how greatly the development of functions that accept arbitrarily many parameters has evolved over time. From compile time optimizations, to greater flexibility and an even more robust syntax, this idea has become a standard for everyone to use without undergoing the initial fear of the unknown that was once induced by C-style implementations.

REFERENCES

[1] D. Gregor, J. Järvi and G. Powell *Variadic Templates (Revision 3)*. N2080, 2006.

[2] N. M. Josuttis *C++17 The Complete Guide*. Leanpub, 2019.

[3] S. Prata *C++ Primer Plus, sixth edition*. Addison-Wesley, 2011.

[4] B. Stroustrup *The Design and Evolution of C++*. Addison-Wesley, 1994.

[5] B. Stroustrup *The C++ Programming Language, fourth edition*. Addison-Wesley, 2013.

[6] A. Sutton and R. Smith *Folding expressions*. N4191, 2014.

[7] C++ reference *Variadic functions*. `https://en.cppreference.com/w/c/variadic`.

[8] C++ reference *Parameter pack*. `https://en.cppreference.com/w/cpp/language/parameter_pack`.

[9] C++ reference *Fold expression*. `https://en.cppreference.com/w/cpp/language/fold`.

[10] C++ reference *Function template*. `https://en.cppreference.com/w/cpp/language/function_template`.