docnum ONGOING INCOMPLETE DRAFT
project Programming Language C++, EWG
date June 15, 2014 21:14 EEST
author George Makrydakis
reply-to irrequietus@gmail.com
revision 7d4fb49

# Annotated C++ template parameter packs

George Makrydakis
irrequietus@gmail.com
**UNCORRECTED INCOMPLETE DRAFT**
7d4fb49672bbacaa63684f7188bc281ca559296e

**June 15, 2014 21:14 EEST**

**Abstract**

Parametric polymorphism in C++ is implemented through class and function templates whose parameter types refer to three different declarative parameter abstractions: template non-type, template type-type and template template-type parameters. Since C++11[6], a fourth kind of template parameter abstraction has been introduced, the *template parameter pack*. Such an abstraction is used for expressing an ordered sequence of zero or more parameters of one of those three parameter types. 1

In their current specification, template parameter packs do not have any declarative features that are explicitly descriptive of their size and/or actual expansion intent. Resourceful combinations of parameter packs with constant expressions in class template partial specializations, function templates, SFINAE[11], ADL[8] and partial ordering are extensively deployed in several C++ template meta-programming techniques for code generation purposes [1]. 2

This proposal explores the effects of *optionally annotating C++ template parameter packs* in a backwards compatible manner with information about size, repetition and patterned expansion during their use in template parameter lists. Such annotation can be detrimental in reducing the cost of using templates as computational constructs for code generation within a given translation unit. Code generation techniques using template meta-programming at any level would require less source code boilerplate, as well as having reduced dependency on type-unsafe C++ preprocessor meta-programming for repetitive constructs - where current template parameter list semantics offer no other viable alternative. 3

**DISCLAIMER:** *The current document is a publically disclosed but uncorrected and incomplete draft for a developing C++ proposal as was announced in the ISO C++ Standard - Future Proposals mailing list [10].* ***It does not constitute a complete work yet.*** *At the current drafting stage, several errors, omissions, repetitions or inadequate development of the notions exposed are to be taken care of in newer revisions of this document. When referring to this particular draft, use its short revision identifier (e.g. 7d4fb49) slashed with the paragraph number as it appears in the side margin (e.g. 7d4fb49/4). The author will periodically and frequently release newer revisions of this document at http://atpp.irrequietus.eu in good will, in hope of them being useful in a fruitful discussion on the subjects exposed herein.* 4

## 1 Introduction

Template parameter packs can be viewed as product types, tuples with cohere but undecided size in both their declaration and expansion within a given template parameter list prior to instantiation. However, a valid template parameter list containing parameters and parameter packs in the declaration of a class or function template is itself a tuple of parameters and parameter packs. In C++ template meta-programming, recursive 5

template instantiations using such abstractions can be shown to play the role of algebraic values, with class template partial specializations [9] being one way of inference of their internal structure. This and other techniques based on exploiting SFINAE[11], ADL[8], partial ordering and function template overloading, are actually manipulating the effective template candidates set for instantiation within a resolution scope, using constant expressions and forcing specific type sequences within the candidates' template parameter lists. Such idioms are critical in giving C++ template meta-programming *pattern matching* features typically belonging to functional programming languages.

**6** While parameter packs greatly enhance the functional nature of C++ template meta-programming [3], they are just an omnicomprehensive abstraction of an n-ordered sequence of zero or more parameters without the ability of specifying *which constraints the included parameters and their patterns must satisfy*. Inferences on their internal structure (the parameters included in a parameter pack and eventual patterns) and therefore use of template meta-programming *pattern matching*, is relying on the manipulation of several declarations of partial / explicit class template specializations and / or multiple function template overloads that must be written, leading to increased source code boilerplate. *Template parameter pack annotation* is a potential solution to this problem.

**7** The section on *motivation* briefly analyzes how *pattern matching* is used at the template meta-programming level through short examples and the context-dependent deployment of parameter packs, including multiple parameter packs within template parameter lists in valid C++11 / C++14 code. It is followed by a section of a more rigorous exposition of what annotators are. Annotator semantics and their limit equivalence scenarios with syntactical shorthands for packs of predetermined size ("fixed size parameter packs" [4]) as well as currently valid C++ parameter packs are also examined. The last section of this draft deals with the benefit of adopting *optionally annotated C++ template parameter packs* as a feature for constructs like typelists [5, 2] and "block" switches in comparison to more laborious but currently valid C++11 / C++14 implementations. Future revisions of this draft will display similar effects of the same *pack annotation* semantics for non-type and template-type template parameters.

# 2 Motivation

**8** The explicit goal of *template parameter pack annotation* is to offer an optional descriptive "field" to the current parameter pack declaration and expansion semantics, through which constraints on parameter sequences and patterns contained within a given parameter pack may be specified. Such a feature can offer deterministic control over the actual contents of a parameter pack within the template parameter list declaration through constant expressions, *without requiring breaking changes to be introduced with past and ongoing C++ standards*.

**9** *Template parameter pack annotation factors* are constant expressions enclosed within *template parameter pack annotators* like **{}** and **[]** immediately following parameter pack declaration and under certain conditions in expansion. They aim for specifying whether the presence of a pack in a template parameter list represents a valid match for instantiation to occur when the size of the pack is specific or within a left-closed, right-open interval of values (i.e. **{}** *annotator*), whether it is comprised of a repeated pattern of parameters or not (i.e. **[]** *annotator*). Only constant expressions can be used within *annotators*, including expressions using pack identifiers that have already been declared in the template parameter list of the template involved. These semantics arise from the typical uses of class and function templates where parameter packs are involved for *pattern matching purposes*, examples of which are following.

## 2.1 Parameter lists and packs as input for pattern matching

**10** In the following example, template parameter lists containing packs are declared and expanded in class

template partial specializations and function template overloads in a variety of ways fully compatible with the C++11/C++14 standard, for pattern matching purposes. Once compiled, the resulting binary prints a series of strings on the terminal that are dependent upon the C++ templates and other features are used as computational devices during instantiation phase.

```
#include <cstdio>

template<typename... Types_T>
struct packed; /* class template declaration */

template<typename Head, typename... Tail>
struct packed<Head,Tail...> { /* class template partial specialization */
    typedef packed<Tail...> pop_front;
};

template<>
struct packed<> /* class template explicit specialization */
{ typedef packed<> pop_front; };

template<typename... T>
void function(packed<T...>) { /* general case */
    printf( "%lu types contained in the pack, pack has still %lu\n"
          , sizeof...(T)
          , sizeof...(T) );
    function(typename packed<T...>::pop_front());
}

template<typename T,typename...X>
void function(packed<T,T,X...>) { /* notice the repetition */
    printf( "2 identical types found, pack has still %lu \n"
          , sizeof...(X) + 2 );
    function(typename packed<T,T,X...>::pop_front());
}

template<typename X, typename Y, typename...T>
void function(packed<X,Y,X,Y,T...>) { /* notice a repeating pattern */
    printf("4 types are a pattern of 2, pack has still %lu\n", 4 + sizeof...(T));
    function(typename packed<X,Y,T...>::pop_front());
}

template<typename X, typename Y, typename...T>
void function(packed<X,Y,T...>) { /* notice the difference with T,T,X... */
    printf("2 different types found, pack has still %lu\n", 2 + sizeof...(T));
    function(typename packed<X,Y,T...>::pop_front());
}

void function(packed<>) { /* better match than packed<T...>, with T... empty */
    printf("no types contained anymore!\n");
}

int main() {
   function(packed< char, short, short, long , int
                  , char, int  , char , short, short
                  , int , int>());
   return {};
}
```

The previous example shows how several behaviors of C++ class and function template features are combined for the "best match" to prevail according to which parameter pattern is more specialized in size and repetition for the template instantiation we are dealing with. **11**

Currently, pack semantics do not allow us to look further into their structure unless more code is written for such an occasion. They can be expanded only when present in a context where all of the parameters in the template parameter list they appear can be deduced and thus why they must appear last in a declarative template parameter list. This context dependency of parameter packs is due to their omnicomprehensive character of abstracting zero or more parameters without any size or pattern constraints. **12**

## 2.2 Pack ordering is a naive implicit matching constraint

**13** In order to better illustrate the template parameter list context of current pack semantics, we begin with a very simple duo of variadic class templates in the following code segment:

```
template<typename...>
struct wrapper {};

template<typename... T>
struct some_template {
    static void hello()
    { printf("hello world!\n"); }
};
```

**14** In the specializations that follow, the template parameter pack is not expanded within the same template parameter list where it was declared and is present as a terminal parameter abstraction. This allows for the pack to be expanded within a context where the totality of parameters turned arguments can be deduced.

```
template<typename... X>
struct some_template<wrapper<X...>,int> { /* X is alone! */
    static void hello()
    { printf("hello world from wrapper!\n"); }
};

template<typename... X>
struct some_template<wrapper<int,X...>,int> { /* rightmost X! */
    static void hello()
    { printf("the first type in the wrapper is an int...\n"); }
};
```

**15** Unlike what happened in our previous example, due to the pack appearing before a non-pack parameter the compiler cannot deduce all the parameters involved when the pack is required to expand in the template argument list the following class template partial specialization:

```
template<typename... X>
struct some_template<wrapper<X...,int>,int> { /* cannot deduce parameters! */
    static void hello()
    { printf("this code is wrong!"); }
};

template<typename...X>
struct some_template<X...,int> { /* cannot deduce parameters! */
    static void hello()
    { printf("this code is also wrong!\n"); }
};
```

**16** This behaviour is consistent in both class template partial specializations and function template declarations even when *multiple template parameter* pack declarations are used. Notice the ordering of multiple packs required to expand within each instantiation of class template I, allowing for all parameters to be deduced:

```
/* multiple parameter packs in class template partial specialization */
template<typename... A, typename... B, template<typename...> class I>
struct classy<I<A...>,I<B...>> {
    static void deploy() {
        printf("multiple parameter packs in partial specializations!\n");
    }
};
/* multiple parameter packs in function templates */
template<typename... A, typename... B, template<typename...> class I>
void function(I<A...>,I<B...>)
{ printf("multiple parameter packs in function templates!\n"); }
```




irrequietus@gmail.com

Unsurprisingly, the following code works because the parameter pack is required to expand within class template scope, where the all parameters have already been deduced during instantiation and thus parameter and argument lists used may deploy packs at any position. **17**

```
template<typename... X>
struct working {
    typedef wrapper<X...,int> type;

    static void function(X...,int)
    { printf("surprised to see me?\n"); }
};
```

### 2.3 Pack size and patterns are implicitly used as explicit constraints

As a last reminder, partial ordering in function template overloads (and by analogy in class template partial specializations) ensures that for a specific patterned sequence the best match is selected, but deploying this as a feature requires a geometrically increasing number of function template overloads to be specified in a naive implementation or complex SFINAE constraints for where certain patterns can be clustered into a single overload. This is another reason why having syntax permitting pattern clustering into a single overload or partial specialization can significantly lower source code boilerplate and lessen SFINAE dependency. **18**

```
template<typename A, typename B, typename C, typename D, typename... T>
void function(A,B,C,D,T...) { printf("A,B,C,D,T...\n"); }

template<typename A, typename B, typename C, typename D>
void function(A,B,C,D) { printf("A,B,C,D\n"); }

template<typename A, typename B>
void function(A,A,B,B) { printf("A,A,B,B\n"); }

template<typename A>
void function(A,A,A,A) { printf("A,A,A,A\n"); }

template<typename A, typename B>
void function(A,B,A,B) { printf("A,B,A,B\n"); }

template<typename... T>
void function(T...) { printf("T...\n"); }
```

While template parameter pack ordering is an *implicit, terminal-bound constraint* in order for deducibility of all the parameters in a template parameter list to be guaranteed, multiple patterns using class template partial specializations and function template overloads provide *explicit constraints* for pattern matching according to size and pattern criteria. Template parameter pack annotation is about offering *a systematic way for clustering explicit size and pattern constraints within the template parameter list itself.* **19**

## 3 Analyzing template parameter pack annotation

For the purposes of conceptualization, we are going to borrow and admittedly abuse some of the mathematical notation available for formulations, mixed with the annotators **{}[]** that we are going to use when we translate this into the proposed C++ syntax for annotation. A C++ template parameter pack ...$T$ will be symbolized as a capital letter with an overline (i.e. $\overline{T}$), while the ***sizeof...(T)*** constant expression will be symbolized as $|\overline{T}|$. **20**

**21** Unless otherwise specified, all other notation used in the following formulation refers to its pristine mathematical meaning. At the current stage of this incomplete draft, there may be unintentional discrepancies that have to be corrected and will be in a newer version of this draft.

## 3.1 Parameter pack annotation semantics

**22** 1. An annotated template parameter pack is an n-ordered sequence of zero or more template parameters of the same type (non-type, type-type, template-type) whose identifier is followed by the *interval* and *pattern* annotators. The *interval annotator* must always precede the *pattern annotator* while both follow the identifier used for the *annotated template parameter pack*. The *interval annotator* serves as an explicit size constraint, while the *pattern annotator* serves as the explicit pattern constraint we discussed about before.

**23** 2. In the context of the *interval annotator*, when $\{N\}$ or $\{N, K\}$ are used, they refer to constant expressions $N$ and $K$ and are referred to as *interval factors* of an *interval annotation*. The first case specifies that a given pack is actually a shorthand for a parameter sequence consisting of exactly $N$ parameters of the same kind (non-type,type-type,template-type), but not necessarily resolving to the same type during parameter to argument conversion. When two comma-separated constant expressions are used, the *interval annotator* refers to a valid right-open interval of $[N, K) \in \mathbb{Z}_{\geq 1}$ within which the size of a pack matching the expression should be in order for the match to be valid. When no constant expressions are used, $\overline{T}\{\}$ is equivalent to $\overline{T}$, $\{|\overline{T}|\}$ and $\{0, |\overline{T}|\}$ which abstract a parameter pack of zero or more parameters of the same kind (non-type,type-type,template-type), but not necessarily resolving to the same type during parameter to argument conversion. This is how *interval annotation* resolves into parameter pack semantics as we know them in C++11 / C++14.

**24** 3. When the *interval* annotator is used with a single constant expression, it can be referred to as *anchoring annotator* and the annotation itself referred to as *anchoring annotation* regardless of whether a *pattern annotator* follows or not. *Anchoring annotation* is equivalent to having declared a template parameter list with a fixed number of parameters.

**25** 4. In the context of the *pattern annotator* **[]** or **[1]** or $[N]$ are used for declaring an annotated parameter pack made out of either a single (**[]** or **[1]**) or more ($[N]$) repetitions of the same parameter pattern. Thus, in *pattern annotation*, the size of the pack does not get altered by what was specified in its preceding *interval annotation*. In order for such a pack to be considered in a resolution set as a viable candidate, its size during instantiation must be an exact multiple of the *pattern annotation factor*, given that it must be comprised of an exact multiple of repetitions of a parameter sequence whose size ***is*** the *pattern annotation factor*.

**26** 5. Annotated parameter packs co-declared with regular parameter packs in the same template parameter list gain partial ordering precedence over the latter unless they constitute equivalent forms of regular packs themselves, in which case a match cannot be made because parameter deduction cannot be guaranteed. Multiple packs with *anchoring annotation* may be present with any order in the declaration of a template parameter list. Unlike their non-annotated and interval annotated counterparts, they are deterministically specified in a context where deducibility is guaranteed.

**27** 6. When *anchoring annotation* is used in an expansion context it must immediately follow the unpacking ellipsis. Using an *anchoring annotator* after the pack specifier without it being expressed as expanding, results to individually accessing a parameter with an order specified by the enclosed constant expression. The *anchoring annotator* is called *individual accessor* in this context. If such an expression is invalid, the code is ill-formed.

7. When the *pattern annotator* is used in an expansion context, it must follow the unpacking ellipsis or an eventual *anchoring annotator*; it always resolves to expanding the preceding pack as many times as the *pattern factor* dictates and is not indicative of any constraints upon an already declared parameter pack. **28**
8. *Interval annotation* can only be used in the declaration of a template parameter list. A template instantiation or argument list involving parameter expansion of pack declared with *interval annotation*, with an anchoring factor in expansion not within the interval specified by the *interval annotation* used in its declaration results in its removal from the resolution set within the involved translation unit. Invalid intervals, negative annotation factors of any kind, zero-valued anchoring and / or pattern factors have the same removal effect. This is practically equivalent to SFINAE removal of annotated packs. **29**
9. The template parameter pack specifier can be used in constant expressions of any of its annotation factors. **30**

## 3.2 Anchoring annotation, fixed size parameter packs and individual accessors

One of the current characteristics of C++ parameter packs is the inability to access individual parameters contained for a given index without relying on source code boilerplate using recursive template instantiations. Proposing "fixed sized parameter packs" [4] as convenient shorthand notation without individual access (*despite it lays the ground for it!*) does not yield any substantial benefit over parameter pack semantics for the *explicit size constraint* that was previously discussed in template mediated pattern matching or the use of explicitly specified parameter types in parameter lists, for a series of reasons. **31**

1. Adopting such syntax for just the sake of fixed size parameter packs does not reduce code boilerplate requirements when a specific pattern of parameters must be specified in a class template partial specialization or a function template overload, despite any perceived benefits in non-type parameter expansion within initializer lists. **32**
2. If the constant expression used for specifying size cannot make use of the already declared pack specifier the "fixed size pack" is abstracted with, such packs are not able to do any inference on the internal structure of the parameter pack they are abstracting, increasing SFINAE cost in subsequent class template partial specializations and function template overloads. **33**
3. Lacking individual access means inability to specify declarations using parameters specified in such a pack without additional boilerplate, which puts them in a disadvantageous position respect to forms explicitly specifying parameter sequence and reducing them to just a shorthand during parameter pack declaration in a given template parameter list. **34**

In the following example, we are using () enclosed constant expressions for referring to isolated "fixed size parameter packs" in order to separate the context of such a proposal[4] with the context of our current draft and annotator syntax. The lack of individual parameter accessors becomes immediately evident in such a context because it does not allow for C++ parametric polymorphism to work by making it impossible for individual parameters contained in such packs to be used in declarations within template definitions. This is a syntactical handicap over templates where parameter lists are explicitly specified in parameter sequence forms, which fixed size parameter packs aim to replace. **35**

```
/* code in the context of just proposing "fixed sized parameter packs" */
template<typename... Args_T(3)> /* = template<typename,typename,typename> */
struct class_template
{ /* must have template<typename,typename,typename> semantics, access ? */ };

template<typename... Args_T(3)> /* = template<typename,typename,typename> */
void function(Args_T... args)   /* "fixed size pack" expanding syntax */
{ /* must have template<typename,typename,typename> semantics, access ? */ }
```

**36** Even if such packs were to be introduced lacking individual accessors, it would be trivial enough for programmers of any level of C++ expertise to write a simple class template providing the same effect, to the point of becoming a nuissance not having it in the standard as a library feature.

```
template<std::size_t N, typename... Args_T>
struct fixed_atpos {
    public:
        struct access_error {};
    private:
        template<typename... Fixed_T(K), typename X, typename... Types_T>
        static typename std::enable_if<(K < sizeof...(Args_T)),X>::type
        impl(Fixed_T...,X,Types_T...); /* fixed (determined) pack = ok */

        static access_error impl(...);
    public:
        typedef decltype(impl(std::declval<Args_T>()...)) type;
};
```

**37** Such code easily prompts for simplification just as was the case with *std::enable_if* [7] - like templates using template aliases in C++14 or the abbreviated form of *for* loops, in a context where *fixed_atpos* is valid.

```
template<std::size_t N, typename... Args_T>
using fixed_atpos_t = typename fixed_atpos<N,Args_T...>::type;
```

**38** In the context of this draft, *anchoring annotation* in declarative and expansive context has the advantage of behaving like fixed size parameter packs [4] without their disadvantages in either non-type, type-type and template-type parameter form.

**39** *Anchoring annotation* is a limit case of template parameter pack annotation, which is more malleable to solving both *explicit size* and *patterned repetition* problems in pattern matching computational constructs often used in modern C++ code, without requiring introduction of backwards - breaking changes or additional boilerplate to compete with explicitly specified template parameter lists. The latter is also due to its use for individual access of parameters when a pack has been declared with *anchoring annotation*.

**40** The final effect is to significantly reduce repetitive source code boilerplate through complex but type-unsafe preprocessor meta-programming approaches for resolution set manipulation, while lessening the need for SFINAE constructs outside the declaration of a template parameter list. This leads to concise, error-free and more readable code, some of which is presented in later sections.

## 3.3 Understanding annotation equivalence through a tentative formulation

**41** Prior to reading this section, a fair warning. The meaning of this section is conceptualization of the rules laid out before for annotated parameter packs. There is some abuse of familiar symbols which happens after overlined identifiers like $\overline{A}$ for the sake of illustrating the concepts involved. The semantics of {} and [] obviously refer to annotators when following an overline identifier since these are the ones we are going to be using when translating these concepts into proposed C++ syntax.

**42** We already know that parameters pack are n-ordered sequences (n-tuples) of zero or more parameter types of the same kind (non-type, type-type, template-type), of not necessarily identical parameters. They can be viewed as product types and modelled using nested ordered pairs, in a way reminiscent of a left/right fold of a cons function over the n-tuple that they actually are. The following two equivalent formulations inductively define a template parameter pack of $t_0, t_1, t_2, \ldots t_{n-1}$ are known to be valid representational approaches. The

most basic annotator equivalence is added to both.

$$\overline{X} \equiv \overline{X}\{\}[] = \underbrace{(t_0,(t_1,(t_2,(\ldots,(t_{n-1},\emptyset)\ldots))))}_{n=|\overline{X}|=sizeof\ldots(X)} \tag{1}$$

$$\overline{Y} \equiv \overline{Y}\{\}[] = \underbrace{((\ldots(((\emptyset,t_0),t_1),t_2),\ldots),t_{n-1})}_{n=|\overline{Y}|=sizeof\ldots(Y)} \tag{2}$$

We will be using the $\overline{X}$ formulation from now on, with an *annotator* next to parameter pack declaration containing the size of the pack enclosed in curly braces as $\{j\}$, when the intention is to force the j-tuple of contained elements that the pack is into a finite and well defined equivalent of the non-variadic form of said pack, which would be preferentially matched over the variadic form should the two coexist. When either $\overline{T}\{sizeof\ldots(T)\}$ or $\overline{T}\{|\overline{T}|\}$ are used, these are completely equivalent to declaring $\overline{T}\{\}$ and $\overline{T}$ and therefore the "size" annotation may be ommited. **43**

Another important operation can also be modelled after valid C++11/C++14 code like the following (*wrapper* and *error_type* were defined previously) when it comes to the semantics of joining two parameter packs of the same kind: **44**

```
/* NOTE: valid C++11,C++14 code */
template<typename,typename>
struct join_packs
{ typedef error_type type; };

template<typename... A0, typename... A1>
struct join_packs<wrapper<A0...>,wrapper<A1...>>
{ typedef wrapper<A0...,A1...> type; };
```

This will be formulated through the following convention that can be generalized for an arbitrary amount of parameter packs, having any variation of size and ordered sequence of types abstracted in said packs:

$$\begin{aligned}
\langle\overline{A_0}\{j_0\}\rangle &= (\overline{A_0}\{j_0\},\emptyset) = \\
&= (a_{0_0},(a_{0_1},(a_{0_2},\ldots(a_{0_{j_0-1}},\emptyset)\ldots))) &(3)\\
\langle\overline{A_0}\{j_0\},\overline{A_1}\{j_1\}\rangle &= (\overline{A_0}\{j_0\},(\overline{A_1}\{j_1\},\emptyset)) = \\
&= (a_{0_0},(a_{0_1},(a_{0_2},\ldots(a_{0_{j_0-1}},\overline{A_1}\{j_1\})\ldots))) &(4)\\
&= (a_{0_0},(a_{0_1},(a_{0_2},\ldots(a_{0_{j_0-1}},(a_{1_0},(a_{1_1},(a_{1_2},\ldots(a_{1_{j_1-1}},\emptyset)\ldots))))\ldots))) \\
\langle\overline{A_0}\{j_0\},\overline{A_1}\{j_1\},\overline{A_2}\{j_2\}\rangle &= (\overline{A_0}\{j_0\},(\overline{A_1}\{j_1\},(\overline{A_2}\{j_2\},\emptyset))) = \\
&= (a_{0_0},(a_{0_1},(a_{0_2},\ldots(a_{0_{j_0-1}},(\overline{A_1}\{j_1\},(\overline{A_2}\{j_2\},\emptyset)))\ldots))) &(5)
\end{aligned}$$

It is conceptually easy to inductively define the new m-tuple of different parameter packs forming a parameter pack itself using the "size" annotator. **45**

$$\underset{\forall m,j_0,j_1,j_2,\ldots,j_{m-1}\in\mathbb{Z}_{\geq 1}}{\overline{A}\{\Sigma_{i=0}^{m-1}j_i\}} = \overbrace{\underbrace{\langle\overline{A_0}\{j_0\},\overline{A_1}\{j_1\},\overline{A_2}\{j_2\},\ldots\overline{A_{m-1}}\{j_{m-1}\}\rangle}_{\boldsymbol{m-tuple}\text{ of different parameter packs.}}}^{sizeof\ldots(A)=\Sigma_{i=0}^{m-1}j_i}$$

$$\Rightarrow$$

$$\underset{\forall m,j_0,j_1,j_2,\ldots,j_{m-1}\in\mathbb{Z}_{\geq 1}}{\overline{A}\{\Sigma_{i=0}^{m-1}j_i\}} = (\overline{A_0}\{j_0\},(\overline{A_1}\{j_1\},(\overline{A_2}\{j_2\},\ldots,(\overline{A_{m-1}}\{j_{m-1}\},\emptyset)\ldots))) \tag{6}$$

If multiple copies of this new pack were used in sequence to create a new pack through repetition, the **46**

following would hold for the case of a 2-tuple repetition:

$$\underset{\forall m,j_0,j_1,j_2,\ldots,j_{m-1}\in\mathbb{Z}_{\geq 1}}{\langle \overline{A}\{\Sigma_{i=0}^{m-1} j_i\}, \overline{A}\{\Sigma_{i=0}^{m-1} j_i\}\rangle} = \overbrace{\langle \overline{A_0}\{j_0\}, \overline{A_1}\{j_1\}, \overline{A_2}\{j_2\}, \ldots \overline{A_{m-1}}\{j_{m-1}\}, \overline{A_0}\{j_0\}, \overline{A_1}\{j_1\}, \overline{A_2}\{j_2\}, \ldots \overline{A_{m-1}}\{j_{m-1}\}\rangle}^{2\times\Sigma_{i=0}^{m-1} j_i}$$
$$\underbrace{\qquad\qquad}_{\mathbf{2\times m-tuple}\text{ of different parameter packs.}}$$

**47** Continuing this process by repeatedly expanding this new pack through multiple repeats of the original from which it started, we arrive at the general form for n repeats as is below by introducing a "pattern" annotator $[n]$ as a symbolic shorthand of the process.

$$\underset{\forall n,m,j_0,j_1,j_2,\ldots,j_{m-1}\in\mathbb{Z}_{\geq 1}}{\overline{A}\{n\times\Sigma_{i=0}^{m-1} j_i\}[n]} = \overbrace{\langle \overline{A_0}\{j_0\}, \overline{A_1}\{j_1\}, \overline{A_2}\{j_2\}, \ldots \overline{A_{m-1}}\{j_{m-1}\}, \ldots, \overline{A_0}\{j_0\}, \overline{A_1}\{j_1\}, \overline{A_2}\{j_2\}, \ldots \overline{A_{m-1}}\{j_{m-1}\}\rangle}^{sizeof\ldots(A)=n\times\Sigma_{i=0}^{m-1} j_i}$$
$$\underbrace{\qquad\qquad}_{\mathbf{n\times m-tuple}\text{ of different parameter packs.}} \tag{7}$$

**48** After formulating (7) as the "annotated" form of a template parameter pack, there is an interesting and intuitive observation of whether there can be an equivalence between the "annotated" form and template parameter packs so that any template parameter pack can be rewritten into an annotated form of another parameter pack for which explicit specification on its size and eventually occurring repetitive patterns may be specified.

$$\underset{\exists j\in\mathbb{Z}_{\geq 0}\wedge n\in\mathbb{Z}_{\geq 1}}{\overline{A'}\{j\}[n]} \equiv \overline{A} \tag{8}$$

**49** There are just three cases where (8) has to be proven valid: the single parameter and empty parameter pack, the non-empty parameter pack and the pack constructed through pattern repetition.

**50** 1. A single parameter may be viewed as the expansion of an annotated pack with a single parameter, therefore for a given parameter $A$, when $j = n = 1$, making $\overline{A}\{1\}[1]$ equivalent to $A$. For the empty parameter pack case, according to annotation semantics, we have $j = 0, n = 1$:

$$\underset{j=n=1}{\overline{A}\{j\}[n]} \equiv A \tag{9}$$

$$\underset{j=0,n=1}{\overline{A}\{j\}[n]} \equiv \underset{|A|=0}{A} \tag{10}$$

**51** 2. A finite sequence of parameters of the same kind (non-type,type-type,template-type) $A_0, A_1, A_1, \ldots, A_{m-1}$ constituting an m-tuple themselves, can make use of the result of (9) and be rewritten in the form of an annotated pack as follows:

$$\overline{A} = \overbrace{\langle A_0, A_1, A_2, \ldots, A_{m-1}\rangle}^{m\in\mathbb{Z}_{\geq 1}} = \overbrace{\langle \overline{A_0}\{1\}[1], \overline{A_1}\{1\}[1], \overline{A_2}\{1\}[1], \ldots, \overline{A_{m-1}}\{1\}[1]\rangle}^{m} =$$
$$= \overline{A}\overbrace{\{1+1+\ldots+1\}}^{m}[1] = \underset{j=m,n=1}{\overline{A}\{m\}[1]} =$$
$$= \overline{A}\{j\}[n] \tag{11}$$

**52** 3. For cases where the *pattern annotator* is to have values greater than 1, we are really declaring a

parameter pack $\overline{A}$ that is constructed by multiple repetitions of another pack $\overline{A'}$:

$$\begin{aligned}\overline{A} &= \langle \overbrace{A_0, A_1, A_2, \ldots, A_{m-1}}^{m}, \overbrace{A_0, A_1, A_2, \ldots, A_{m-1}}^{m}, \ldots, \overbrace{A_0, A_1, A_2, \ldots, A_{m-1}}^{m} \rangle = \\ &= \langle \underbrace{\overline{A'}\{m\}[1], \overline{A'}\{m\}[1], \ldots, \overline{A'}\{m\}[1]}_{n=|\overline{A}|, \overline{A'}=\langle A_0, A_1, A_2, \ldots, A_{m-1}\rangle} \rangle = \\ &= \overline{A'}\{\underbrace{m+m+m+\cdots+m}_{n}\}[n] = \underset{j=n\times m}{\overline{A'}\{n\times m\}[n]} = \\ &= \overline{A'}\{j\}[n] \end{aligned} \tag{12}$$

Even when a new parameter pack is constructed by mixing packs that are completely unrelated regarding number of parameter packs contained, provided they are of the same kind (non-type,type-type,template-type), annotated form can be used to describe them. **53**

$$\begin{aligned}\overline{A} &= \langle \underset{\overline{A_0}\neq\overline{A_1}\neq\overline{A_2}\neq\cdots\neq\overline{A_{m-1}}}{\overline{A_0}, \overline{A_1}, \overline{A_2}, \ldots, \overline{A_{m-1}}} \rangle = \\ &= \langle \overline{A'_0}\{j_0\}[n_0], \overline{A'_1}\{j_1\}[n_1], \overline{A'_2}\{j_2\}[n_2], \ldots, \overline{A'_{m-1}}\{j_{m-1}\}[n_{m-1}] \rangle = \\ &= \underset{j=j_0+j_1+j_2+\ldots+j_{m-1}, n=1}{\overline{A'}\{j_0+j_1+j_2+\ldots+j_{m-1}\}[1]} = \\ &= \overline{A'}\{j\}[n] \end{aligned} \tag{13}$$

The generalization of *anchoring annotation* as tentatively formulated before is the *interval annotation*, which actually refers to a set of valid anchored packs whose rank is that of the annotation interval. Given the claims of the section in annotation semantics and the formulations involving equivalences so far, template parameter pack annotation in its entirety can be be trivially deduced into a tentative series of formulations as follows. **54**

$$\underset{\exists j\in\mathbb{Z}_{\geq 0}\wedge n\in\mathbb{Z}_{\geq 1}}{\overline{A}} \equiv \overline{A'}\{j\}[n] \tag{14}$$

$$\underset{\exists i\in\mathbb{Z}_{\geq 0}\wedge j,n\in\mathbb{Z}_{\geq 1}}{\overline{A}\{i,j\}[n]} = \left\{\overline{A'} \mid \overline{A'}\{x\}[n] \wedge x \in [i,j) \neq \emptyset\right\} \tag{15}$$

From (14) and (15) we can derive the equivalences between annotated packs and parameter packs as we currently know them, including that of the size of the empty pack in annotated form. **55**

$$\overline{A} \equiv \overline{A}\{\} \equiv \overline{A}\{\}[] \equiv \overline{A}\{\}[1] \equiv \overline{A}\{|\overline{A}|\}[] \equiv \overline{A}\{|\overline{A}|\}[1] \tag{16}$$

$$\overline{A} \equiv \overline{A}\{0,|\overline{A}|\} \equiv \overline{A}\{0,|\overline{A}|\}[] \equiv \overline{A}\{0,|\overline{A}|\}[1] \tag{17}$$

As for the empty annotated pack, since we are to use the empty set symbol $\emptyset$ for defining where SFINAE is to play a role, the **sizeof...()** operation is important in defining it as zero-length. **56**

$$|\overline{A}\{0\}[1]| = 0 \tag{18}$$

In conclusion, any invalid constant expression used as an annotation factor invalidates the parameter list, invoking SFINAE and thus the $\emptyset$ symbol is used. **57**

$$\underset{j\in\mathbb{Z}_{\geq 0}}{\overline{A}\{j\}[0]} \equiv \underset{\forall[i,j)\subset\mathbb{Z}_{\geq 0}\vee[i,j)=\emptyset}{\overline{A}\{i,j\}[0]} \equiv \underset{[x,y)=\emptyset\wedge n\in\mathbb{Z}_{\geq 0}}{\overline{A}\{x,y\}[n]} \equiv \underset{n\in\mathbb{Z}_{\geq 0}}{\overline{A}\{0,0\}[n]} \equiv \emptyset \tag{19}$$

# 4 Deployment scenarios

## 4.1 Easier typelist implementation

**58** Annotated and non-annotated packs, can be combined for the explicit purpose of declarating constructs useful for manipulating typelists [5, 2, 1]. Class template partial specializations deploying substitutive ordering and type accessors can be mixed with a series of constant expressions in the annotated packs used that yield interesting results.

**59** The following examples illustrate one possible implementation of the typelist concept using annotated packs, with the typevector class being named such due to type accessors yielding $O(1)$ access during compilation.

**60** 1. Initially, the *typevector* class is defined, along with an *error_type* for convenience. Definitions for three different operations, namely *at_pos*, *alter_at* and *split_at* are made for individual type access at a given position, changing the type at a given position and splitting the typevector into two different ones at a given position. The default result is always the *error_type*.Take notice that partial specializations of the *at_pos*, *alter_at*, *split_at* templates having the annotation equivalent of a regular parameter pack, would cause ambiguity to ensue because of the equivalence between $...T\{0, sizeof...(T)\}$ and $...T$.

**61** 2. We begin with the bare fundamentals of *error_type* and *typevector*.

```
struct error_type {};
/* the typevector is actually just a holder for a pack */
template<typename... X> struct typevector {};
```

**62** 3. The interval annotator is used for declaring an annotated parameter pack in the following partial specializations of *at_pos*, where the constant expression of the right bound will resolve to $T\{0,0\}$ when access beyond the bounds is attempted. This is not a valid interval, therefore this specialization will be removed from the viable candidates set due to SFINAE, with the unspecialized class template definition providing a better match.

```
template<std::size_t N, typename>
struct at_pos { typedef error_type type; };

template< std::size_t N
        , typename... T{0,(N < sizeof...(T) ? sizeof...(T) : 0)}>
struct at_pos<N,typevector<T...>>
{ typedef T{N} type; };
```

**63** 4. The combination of annotated and non-annotated packs can also be used for quickly altering the type parameter present at a given position in the *typevector*. Again, the interval annotation is deployed for placing invalid access through a constant expression evaluating to 0, forcing $T\{0,0\}$ when that is attempted.

```
template<std::size_t N, typename>
struct alter_at { typedef error_type type; };

template< std::size_t N
        , typename X
        , typename Z
        , typename... T{0,(N < sizeof...(T) ? sizeof...(T) : 0)} /* T{0,0} ! */
        , typename... R >
struct alter_at<N,X,typevector<T...{N-1},Z,R...>>
{ typedef typevector<T...{N-1},X,R...> type; };
```

5. Splitting a *typevector* into two halves at a given position is reduced to a problem of specifying the interval bounds correctly, with two constant expressions providing the same kind of $T\{0,0\}$ mediated SFINAE safety. **64**

```
template<std::size_t N, typename>
struct split_at {
    typedef error_type first_half;
    typedef error_type secnd_half;
};

template< std::size_t N
        , typename... L{0,(N < sizeof...(L) ? sizeof...(L) : 0)}
        , typename... R{N,(N < sizeof...(R) ? sizeof...(R) : 0)} >
struct split_at<N,typevector<L...{N-1},R...{sizeof...(R)}>> { /* anchored ! */
    typedef typevector<L...> first_half; /* already anchored */
    typedef typevector<R...> secnd_half; /* already anchored */
};
```

6. Given that in the context of the proposal $\overline{T}\{0,|\overline{T}|\}[1] \equiv \overline{T}\{0,|\overline{T}|\}$ while $\overline{T}\{0,|\overline{T}|\}[0] \equiv \emptyset$ with the latter invoking SFINAE since it would resolve to an empty set (making the template parameter list invalid) and $\overline{T}\{0,|\overline{T}|\} \equiv \overline{T}\{|\overline{T}|\} \equiv \overline{T}\{\}$, we can exploit the pattern annotator in declaration and the *individual accessor* within template definition body for an interesting rewrite of at least one of the partial specializations defined before, e.g. of *at_pos*: **65**

```
template<std::size_t N, typename... T{}[N < sizeof...(T)]>
struct at_pos<N,typevector<T...>>
{ typedef T{N} type; };
```

## 4.2 Combining with different kinds of expansions

An interesting consequence of *full interval annotation* in function templates derives from the fact that although interval annotation refers to a set of instantiations for which said template is to provide a valid match, it is one instantiation at a time actually occuring that is constrained to be matched with said template. According to the annotation rules, an *anchored expansion* whose enclosed constant expression does not fall within the specified expansion interval gets removed from the candidate match set. **66**

However, at instantiation time there is a specific, non-alterable length to which the expansion is to occur. Therefore multiple well-formed anchored expansions in function calls can be used in combination with a no-op (e.g. **void gun(...){}**, notice the use of the ellipsis) to make the following function template definition provide the optimal locality of reference for a series of different expansion semantics depending on the length of the parameter list used. The following snippet is taken from a GoingNative 2012 talk[3] by Andrei Alexandrescu, to which annotation is bolted on for illustrative purposes. **67**

```
template<class...{4,7} Ts>          /* sizeof...(Ts)= 4, 5 or 6 during instantiation ! */
void fun(Ts... vs) {                /* we do not specify an anchored expansion yet */
    gun(A<Ts...{4}>::hun(vs)...); /* sizeof...(Ts) = 4, else no-op */
    gun(A<Ts...{5}>::hun(vs...)); /* sizeof...(Ts) = 5, else no-op */
    gun(A<Ts>::hun(vs)...{6});    /* sizeof...(Ts) = 6, else no-op */
}
```

While the same effect could be achieved through extensive use of SFINAE and/or tag dispatching, it would require a lot more boilerplate and direct intervention into the function signatures of some of the function templates involved (e.g. **gun**). Annotation has the advantage of leaving the constant expressions within annotators to either be explicitly specified or evaluated during instantiation based on even more complex constraints. **68**

## 4.3 The C++11/C++14 block switch effect, better

**69** Using function templates and lambdas, a peculiar kind of "switch" may be implemented in valid C++11 / C++14 as in the following code snippet. Specifically, lambdas and SFINAE are combined to provide a "block switch" effect within the body of the *func_templ* function template depending on the number of parameters present in the pack during instantiation. The *no-op* calls are optimized away.

```
template<std::size_t N, typename... X, typename F>
typename std::enable_if<(sizeof...(X) == N),void>::type sel(F&& f) { f(); }

template<std::size_t N, typename... X>
void sel(...) {} /* no-op */

template<typename... T>
typename std::enable_if<(sizeof...(T) >= 1) && (sizeof...(T) < 6),void>::type
function(T... t) {
    sel<1,T...>([&t...](){ printf("one\n"); });
    sel<2,T...>([&t...](){ printf("two\n"); });
    sel<3,T...>([&t...](){ printf("three\n"); });
    sel<4,T...>([&t...](){ printf("four\n"); });
    sel<5,T...>([&t...](){ printf("five\n"); });
}
```

**70** We can implement the previous example more easily with interval annotation. An invalid anchored pack expansion for a pack identifier declared with interval annotation removes the template from the resolution candidates. Thus, it becomes easier to use such idioms even without the *std::enable_if* C++11/C++14 template when defining the *func_tmpl* function template and its assistive functions.

```
template<typename...X, typename F>
void sel(F&& f) { f(); }

void sel(...) {} /* no-op */

template<typename... T{1,6}>
void func_tmpl(T... t) {
    sel<T...{1}>([&t...](){ printf("one\n"); });
    sel<T...{2}>([&t...](){ printf("two\n"); });
    sel<T...{3}>([&t...](){ printf("three\n"); });
    sel<T...{4}>([&t...](){ printf("four\n"); });
    sel<T...{5}>([&t...](){ printf("five\n"); });
}
```

## 4.4 Enhancing class template partial specializations

**71** In contrast to function templates, the ability of class templates to have partial specializations makes them more malleable to specifying patterned sequences of types upon which template meta-programming driven type pattern matching occurs. A naive implementation for specializing a class template definition according to the size of the parameter pack declared in the template parameter list would require as many partial specializations as the sizes of interest.

**72** The following is a non-naive valid C++11 / C++14 implementation where the partial specializations required for this kind of problem are delegated to a nested class template. The partial specializations of the nested template are subject to removal from the resolution set through *std::enable_if*. Unlike the approach we followed in function templates, partial specializations open the door for more complex type calculations for "block-switch" like problems.

```
template<typename... X>
struct classy_impl {
private:
    struct void_{};

    template<bool B, typename Z> /* C++14 has this as std::enable_if_t */
    using enabler = typename std::enable_if<B,Z>::type;

    template<typename, typename...> struct impl_{};

    template<typename... T>
    struct impl_<enabler<(sizeof...(T) >= 0 && sizeof...(T) < 3), void_>, T...>
    { static void deploy(){ printf("[0,3)\n"); } };

    template<typename... T>
    struct impl_<enabler<(sizeof...(T) >= 3 && sizeof...(T) < 6), void_>, T...>
    { static void deploy(){ printf("[3,6)\n"); } };

    template<typename... T>
    struct impl_<enabler<(sizeof...(T) >= 6), void_>, T...>
    { static void deploy(){ printf(">=6\n"); } };
public:
    typedef impl_<void_,X...> type;
};

template<typename... X>
using classy = typename classy_impl<X...>::type;
```

If we try to solve the same problem using *interval annotation*, there is not even need for a nested template to exist in order to handle the necessary partial specializations. The inherent SFINAE character of annotation semantics allows for a very succint formulation of programmer intent directly into C++ code. Additionally, there is no need to provide for a long series of specializations even in naive implementations, making constructs depending on pattern matching through long sets of partial specializations easier to deal with. **73**

```
template<typename...>
struct classy {
    static void deploy()
    { printf("sizeof...(X) >=6\n"); }
};

template<typename... X{0,3}>
struct classy<X...> {
    static void deploy()
    { printf("sizeof...(X) in [0,3)\n"); }
};

template<typename... X{3,6}>
struct classy<X...> {
    static void deploy()
    { printf("sizeof...(X) in [3,6)\n"); }
};
```

# 5 Technical specification

## 5.1 Grammar Additions

This section will be among the last ones to be added because it requires far more constrained yet precise wording as to what would need to be amended in the C++ standard for annotators to be used. **74**

## 6 Acknowledgements

The author would like to thank Ville Voutilainen, current secretary of the C++ standard committee for strongly motivating me to write this initial draft. Fixed size parameter packs were discussed and presented on the C++ std-proposals mailing[4] list, in a thread started by Maurice Bos.

## References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN: 0321227255.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70431-5.

[3] Andrei Alexandrescu. *GoingNative 2012 – Variadic Templates are Funadic*. 2012. URL: http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Variadic-Templates-are-Funadic (visited on 02/02/2012).

[4] Maurice Bos. *ISO C++ Standard - Future Proposals – thread: Fixed Size Parameter Packs*. 2014. URL: https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/igOzjW-5KLM (visited on 05/26/2014).

[5] Krzysztof Czarnecki and Ulrich W Eisenecker. "Generative programming" (2000). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.2101.

[6] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[7] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Concept-Controlled Polymorphism." *GPCE*. Ed. by Frank Pfenning and Yannis Smaragdakis. Vol. 2830. Lecture Notes in Computer Science. Springer, 2003, pp. 228–244. ISBN: 3-540-20102-5. URL: http://dblp.uni-trier.de/db/conf/gpce/gpce2003.html#JarviWL03.

[8] Andrew Koenig and Barbara Moo. *Ruminations on C++*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996. ISBN: 0-201-42339-1.

[9] Joaquín M López Muñoz. *Bannalia: trivial notes on themes diverse Functional characterization of C++ template metaprogramming*. 2008. URL: http://bannalia.blogspot.com/2008/05/functional-characterization-of-c.html (visited on 05/20/2008).

[10] George Makrydakis. *ISO C++ Standard - Future Proposals – thread: Fixed Size Parameter Packs*. 2014. URL: https://groups.google.com/a/isocpp.org/d/msg/std-proposals/igOzjW-5KLM/qhNVR3jHsk8J (visited on 05/30/2014).

[11] David Vandevoorde and Nicolai M. Josuttis. *C++ templates : the complete guide*. Boston (Mass.), San Francisco (Calif.), Paris: Addison-Wesley, 2003. ISBN: 978-0-201-73484-3.