Variadic Templates: Exploring the Design Space

Douglas Gregor Jaakko Järvi Gary Powell

Document number: N1704=04-0144
Revises document number: N1603=04-0043
Date: September 10, 2004
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <gregod@cs.rpi.edu>

1 Introduction

This proposal directly addresses two problems:

- The inability to instantiate class and function templates with an arbitrarily-long list of template parameters.
- The inability to pass an arbitrary number of arguments to a function in a type-safe manner.

The proposed resolution is to introduce a syntax and semantics for variable-length template argument lists (usable with function templates via explicit template argument specification and with class templates) along with a method of argument "packing" using the same mechanism to pass an arbitrary number of function call arguments to a function in a typesafe manner and "unpacking" to forward packed arguments to other functions.

2 Motivation

2.1 Variable-length template parameter lists

Variable-length template parameter lists (variadic templates) allow a class or function template to accept some number (possibly zero) of template arguments beyond the number of template parameters specified. This behavior can be simulated in C++ via a long list of defaulted template parameters, e.g., a typelist wrapper may appear as:

```
struct unused;
template<typename T1 = unused, typename T2 = unused,
         typename T3 = unused, typename T4 = unused,
         /* up to */ typename TN = unused> class list;
```

This technique is used by various C++ libraries [7, 8, 5]. Unfortunately, it leads to very long type names in error messages (compilers tend to print the defaulted arguments) and very long mangled names. It is also not scalable to additional arguments without resorting to preprocessor magic [11]. In all of these libraries (and presumably many more), an implementation based on variadic templates would be shorter and would not suffer the limitations of the aforementioned implementation. The declaration of the `list<>` template above may be:

```
template<... Elements> class list;
```

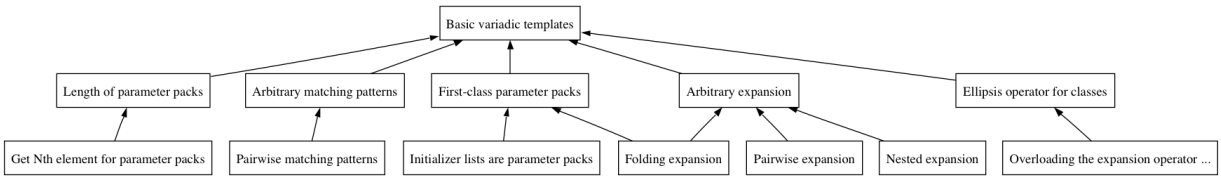

Figure 1: Feature dependency graph for variadic templates

2.2 Typesafe Variable-length Function Parameter Lists

Variable-length function parameter lists allow more arguments to be passed to a function than are declared by the function. This feature is rarely used in C++ code (except for compatibility with C libraries), because passing non-POD types through an ellipsis (`...`) invokes undefined behavior. However, a typesafe form of such a feature would be useful in many contexts, e.g., for implementing a typesafe C++ `printf` that works for non-POD types.

3 Feature menu

Prior versions of this proposal [4, 3] described various points in the design space of variadic templates. In this version of the proposal, we have chosen to enumerate the various potential design decisions (mainly features) and discuss the impact of each on users and compilers. Fig. 1 illustrates the dependencies between features, such that an edge $A \rightarrow B$ indicates that feature A requires acceptence of feature B.

The following sections describe each feature listed in Fig. 1. Sec. 4 then provides several "specials": prepacked sets of features that work well together and represent some overall goal for variadic templates. It is our hope that the committee will select a special or a set of features *a la carte* for which we can draft a more formal specification.

3.1 Basic variadic templates

The most basic form of variadic templates requires the ability to declare class/struct/union and function templates that accept an arbitrary number of template arguments, declare function templates that accept an arbitrary number of (function) arguments, and access the "extra" template and function arguments.

We adopt the use of the ellipsis operator "`...`" to represent variable-length argument lists. For variadic class templates we allow "`...`" as the "kind" of the last template parameter, which may optionally be followed by an identifier. The following class template accepts one type argument followed by zero or more other template arguments.

```
template<typename T, ... Tail> class tuple;

typedef tuple<int, float, double, std::string, 5, std::vector> t;
```

Variadic function templates are declared similarly, allowing one to explicitly specify any number of arguments:

```
template<... Args> void print_template_args();

print_template_args<int, 17, 42l, &X::m>();
```

We refer to `Tail` and `Args` as "template parameter packs", because they pack together a set of template arguments into a single parameter. In their most basic form, template parameter packs provide only a single operation: unpacking via the ellipsis (`...`) operator. Unpacking splits a template parameter pack into its

individual arguments, allowing each to be considered individually. For instance, the following code builds a recursive tuple from the types given:

```
template<typename Head, ... Tail> struct tuple { // #1
  Head head;
  tuple<Tail...> tail;
};

template<typename Head> struct tuple<Head> { // #2
  Head head;
};
```

The instantiation of `tuple<int, double, std::string>` uses the primary template (#1) with `Head=int` and `Tail` containing `double` and `std::string`, which we denote by `Tail=<double, std::string>`. The `tail` member is a tuple that will receive the template arguments `double` and `std::string`, in that order, due to the application of the `...` operator to the template parameter pack `Tail`.

The ellipsis operator represents both unpacking and packing arguments, depending on context. For instance, a partial specialization can fix some number of arguments and collect the rest in a template parameter pack:

```
// Searching for two adjacent types...
template<typename T, ... Elements>
struct adjacent_find : adjacent_find<Elements...> { };

// Found adjacent types!
template<typename T, ... Rest>
struct adjacent_find<T, T, Rest...> {
  static const bool value = true;
  typedef T type;
};

// List is empty. We found nothing.
template<> struct adjacent_find<> { static const bool value = false; };
```

Here, the partial specialization matches the case where two adjacent types are the same and then collects the rest of the arguments into the template parameter pack `Rest`. The ellipsis operator can be applied to a parameter pack wherever there is a list of parameters (for packing) or arguments (for unpacking), including:

- In a partial specialization (e.g., the `adjacent_find` example)
- In a *template-id* (e.g., `tuple<Elements...>`)
- In a function type (e.g., `void(int, ArgTypes...)`)

The ellipsis operator also applies to *value* parameter packs, which store run-time values for type-safe variadic function templates. For instance, a type-safe `printf` might be declared as:

```
template<... Args> void printf(const char* s, const Args& args...);
```

This declaration indicates that `printf` accepts a variable number of parameters (both template and function arguments): the `Args` template parameter pack contains the types of the arguments, whereas the `args` template parameter pack contains the values. The ellipsis operator following "`args`" indicates that extra arguments to `printf` should be bundled into the `args` parameter pack and their types deduced and placed into the `Args` template parameter pack. In this basic version, the template parameter pack containing

the types of the arguments may be a *cv*-qualified template parameter pack or a reference to a *cv*-qualified template parameter pack.

One can think of the typesafe variadic `printf` function as having an expansion to a family of overloads each having a different number of arguments:

```
void printf(const char* s);
template<typename T1> void printf(const char* s, const T1& a1);
template<typename T1, typename T2>
  void printf(const char* s, const T1& a1, const T2& a2);
/* up to some arbitrary N */
template<typename T1, typename T2, /* up to */, typename TN>
  void printf(const char* s, const T1& a1, const T2& a2, /* up to */, const TN& aN);
```

The most basic formulation of variadic templates therefore consists of:

- The ability to declare template and value parameter packs
- The ability to unpack and pack template and value parameter packs via the ellipsis operator.
- Partial ordering rules for class template partial specializations and function templates in the presence of variadic templates (discussed in a previous proposal [3]).

3.2 First-class parameter packs

In the basic formulation, template and value parameter packs are merely bundles of parameters that have no direct representation within the type system. However, we could make template parameter packs into actual types—albeit special ones that permit the use of the ellipsis operator—so that they could be passed to template arguments or compared against each other. Moreover, value parameter packs now have actual types based on the type of the associated template parameter pack, and may become first-class objects. Thus they can be default-constructed (if all types in them are default-constructible), copy-constructed or assigned (if all types of copy-constructible or assignable, respecitvely), and unpacked as needed.

First-class parameter packs have some advantages over the basic formulation of parameter packs, because it allows multiple arguments to be stored together without resorting to recursion as with the `tuple` template in Sec. 3.1:

```
template<... Elements>
class tuple
{
public:
  tuple(const Elements& elements...) : elements(elements) { }

private:
  Elements elements;
};
```

Here we store members of every type in `Elements` in a single parameter pack within the tuple, instead of creating a chain of `tuple` instantiations where each stores a single element plus the next `tuple` in the chain.

User impact : First-class parameter packs make writing some heterogenous data structures simpler because they can eliminate the need for recursive data structures. Performance (both compiler and generated code) will improve because the number of instantiations required to implement a heterogeneous data structure will be reduced from $\mathcal{O}(N)$ to $\mathcal{O}(1)$. This effect has already been seen when comparing the Boost.Tuples library [7], which uses a recursive data structure, to a preprocessor-based library, which enumerates all data members.

First-class parameter packs may be somewhat confusing to users, because the distinction between a "packed" parameter pack and an unpacked one becomes very important. For instance, if one forgets to apply the ellipsis operator the compiler will pass the parameter pack itself, e.g.,

```
template<... Args> void debugout(const std::string& s, const Args& args...)
{
  printf(("Debug: " + s).c_str(), args); // oops, should be "args..."
}
```

Compiler impact : First-class parameter packs introduce more complexity into the implementation of variadic templates, because the compiler must construct actual types and objects for each parameter pack. In addition, there are some interesting interactions between the object model used for parameter packs and the method by which arguments are passed to function objects that require further consideration; see [3].

Evaluation : The most important realization with first-class parameter packs is that they can be emulated via recursive data structures. However, this emulation places an extra burden on users and compilers, and is known to result in poorer performance.

3.3 Initializer lists are parameter packs

Initializer lists are typeless entities stored in the compiler that can only be used in very restrictive ways for initializing aggregates. We could lift some of the restrictions on initializer lists by making them (value) parameter packs. This would permit writing constructors and assignment operators that accept initializer lists, e.g.,

```
template<... Elements>
class tuple
{
public:
  // #1: Build from an initializer list
  tuple(Elements elements) : elements(elements) { }

  // #2: Build from an initializer list, converting as necessary
  template<... Args> tuple(Args args) : elements(args...) { }

private:
  Elements elements;
};

// Passes initializer list as parameter pack to constructor #1
tuple<int, double, float> t1 = { 17, 3.14159, 2.718f };

// Passes initializer list as parameter pack to constructor #2
tuple<int, double, std::string> t2 = { 17, 3.14f, "foo" };
```

Additionally, this change permits emulation of sequence constructors [12] via parameter packs, e.g.,

```
template<typename T>
class vector
{
public:
  template<... Elements> vector(Elements elements)
```

```
  {
    tr1::array<T, (pp_length<Elements>::value)> a = elements;
    assign(a.begin(), a.end());
  }
};
```

The construction of `a` actually performs aggregate initialization from the incoming parameter pack, which unfortunately requires one extra copy. This copy can be elided by implementing the constructor as a metaprogram[1]:

```
template<typename T>
class vector
{
public:
  template<... Elements> vector(const Elements& elements)
  {
    reserve(pp_length<Elements...>::value);
    do_push_back(elements...);
  }

  void push_back(const T&);

private:
  template<... Rest>
  void do_push_back(const T& x, Rest... rest) {
    push_back(x);
    do_push_back(rest...);
  }

  void do_push_back() {}
};
```

User impact : The ability to accept initializer lists in the constructors and assignment allows us to make initialization of standard library data structures (and other user types) more natural, and may improve readability of the language. It also subsumes several of the ideas in [12].

Compiler impact : Implementation of this feature is not expected to require a great deal of effort beyond that of implementing first-class parameter packs because the extension is small and pure.

3.4 Ellipsis operator for classes

One extension of the ellipsis operator would permit it to be applied to any object of class type, which would have the effect of unpacking all of the fields in that class type into separate arguments. This information could be used for compile-time reflective metaprogramming, e.g., to perform automatic marshalling or to construct property inspectors.

User impact : While we do not expect that many users would apply the ellipsis operator to classes, there are many secondary benefits to users when library developers are able to apply reflection to provide features that typically require preprocessing or user intervention.

[1] See Sec. 3.6 for a discussion of `pp_length`

Compiler impact : Implementation of this feature could require a moderate amount of work depending on the format used to describe the members of the class. We have not considered such a format in depth.

3.5 Overloading the expansion operator ...

This proposal introduces `...` as a full-fledged operator only usable for special, compiler-defined parameter packs. Some user-defined types (notable the library-defined `tuple` type) may wish to present a parameter-pack—like interface and support the ellipsis operator. `operator...` could be a nonstatic member function taking zero arguments and returning a parameter pack. For instance, within the `tuple` type:

```
template<... Elements>
  class tuple
  {
    // ...
  public:
    Elements operator...() const { return elements; }

  private:
    Elements elements;
  };
```

The ability to overload the ellipsis operator for any class type may require a new syntax for unpacking, because it will be unclear which objects are being unpacked. Consider:

```
template<typename T, typename U>
void foo(T t, U u)
{
  f(g(t, u), ...);
}
```

In this case, we do not know which object (`t` or `u`) will be unpacked because either (or both) may have an overloaded ellipsis operator. Having an overloadable ellipsis operator with this syntax can make the ellipsis operator unusable in generic code.

User impact : By allowing user data types to overload the ellipsis operator, users may achieve more natural syntax for certain types of data. On the other hand, the ambiguities introduced by the ability to overload this operator may outweigh its advantages.

Compiler impact : Implementation of this feature is expected to require moderate effort, because it introduces another overloadable operator and ambiguities with the expansion of parameter packs and parameter pack-like types.

3.6 Length of parameter packs

One can determine the number of elements in a template parameter pack with the following template:

```
template<...> struct pp_length;

template<typename T, ... Rest>
struct pp_length<T, Rest...> {
  static const int value = 1 + pp_length<Rest...>::value;
};
```

```
template<> struct pp_length<> {
  static const int value = 0;
};
```

However, this requires $\mathcal{O}(N)$ instantiations for a parameter pack of length N, which can be inefficient to compile. This option proposed that unpacking a template or value parameter pack in a `sizeof` expression will return its length, e.g.,

```
template<... Args>
  void f(Args args...)
  {
    const int num_args = sizeof(args...);
    // equivalent to
    const int num_args = sizeof(Args...);
  }
```

User impact : This feature gives users a simple, efficient way to determine the length of a parameter pack. The absence of this feature negatively affects both compile-time performance and usability.

Compiler impact : This feature should be trivial to implement.

3.7 Get Nth element for parameter pack

Parameter packs are only accessible linearly, by peeling off arguments at the front of the parameter pack. A general tuple, however, requires the ability to access the N^{th} element. For parameter packs, we would like to be able to access the Nth element. For this we propose to use the `[]` operator, but restrict the argument to integral constant expressions because the result type differs depending on the value of the argument. For instance,

```
template<... Args>
  void f(Args args...)
  {
    // Make copy of 2nd argument
    decltype(args[1]) second = args[1];
  }
```

We rely on `decltype` [10] to determine the type of the N^{th} argument. If `decltype` is not accepted into C++0x, we would require additional syntax to extract the N^{th} argument from a template parameter pack in $\mathcal{O}(1)$ instantiations.

User impact : This feature gives users a simple, efficient way to access a value in the parameter pack. The absence of this feature negatively affects both compile-time performance and usability.

Compiler impact : This feature should be trivial to implement.

3.8 Arbitrary expansion

With basic variadic templates, the ellipsis operator may only be applied to a parameter pack. A more powerful form of the ellipsis operator allows the parameter pack to occur within an expression of arbitrary complexity followed by an ellipsis; in this case, expansion is performed by replacing parameter pack with each of its values and duplicating the expression for each value. Fig. 2 illustrates the expansions of several

template and value parameter packs. In the figure, `X` is a template parameter pack containing template parameters `X1`, `X2`, ..., `XN` and, when all `Xi` are types, `x` is a value parameter pack of type `X` such that `x1`, `x2`, ..., `xN` are the contained values. Similarly, `Y` is a template parameter pack of size `M`, with associated `Yi`, `y`, and `yi`. Additionally, assume the following declarations:

```
template<...> class list;
template<...> class vector;
template<... Args> void f(Args args...);
template<... Args> void g(Args args...);
```

Types & Expressions	**Expansion**
`list<X...>`	`list<X1, X2, /* up to */, XN>`
`list<int, X..., float>`	`list<int, X1, X2, /* up to */, XN, float>`
`f(x...)`	`f(x1, x2, /* up to */, xN)`
`f(17, x..., 3.14159)`	`f(17, x1, x2, /* up to */, xN, 3.14159)`
`int (*)(X...)`	`int (*)(X1, X2, /* up to */, XN)`
`list<typename add_pointer<X>::type...>`	`list<typename add_pointer<X1>::type, typename add_pointer<X2>::type, /* up to */ typename add_pointer<XN>::type>`
`f(x*x)`	`f(x1*x1, x2*x2, /* up to */, xN*xN)`

Figure 2: Illustration of the expansions of types and expressions using template and value parameter packs.

User impact : Arbitrary expansions provide the equivalent of a metaprogramming `transform` operation, that allows one parameter pack to be transformed into another by applying some function to each value. We expect that many users that need variadic templates will also need arbitrary expansion.

Compiler impact : Arbitrary expansions are clearly more complicated to support than basic expansions, because they require the compiler to store the expression (or type) patterns and produce a sequence of expressions (or types) from those patterns. The implementation is expected to be very similar to that of template instantiation.

Evaluation : Arbitrary expansions are very useful for metaprogramming with parameter packs, but they are not absolutely essential: one can construct these operations given only the basic variadic templates, but they will require $\mathcal{O}(N)$ instantiations instead of $\mathcal{O}(1)$ instantiations.

3.9 Nested expansion

With this feature, ellipses operators may be nested, with each ellipsis binding to the argument text it follows. The same parameter pack may appear in multiple places within the argument text (and all will be replaced with the same type or argument from the parameter pack in each expansion).

The need for nested expansion is apparent when considering the construction of a facility such as Bind [1, §3.3]. With this facility, there is a binder function object consisting of a function object `f` and a set of bound arguments $a_1, a_2, \ldots, a_n$ that has been passed actual arguments $v_1, v_2, \ldots, v_m$. The bound arguments a_i consist of either (a) a value to be passed to `f`, (b) a placeholder referring to one of the values v_k, or (c) another binder function object. Thus, for each argument i to `f` we need to consider the bound argument a_i and potentiall all actual arguments v_k. We therefore define a function $\mu(a_i, v_1, v_2, \ldots, v_m)$ that performs the mapping for each a_i. If `a` is a parameter pack containing $a_1, a_2, \ldots a_n$ and `v` is a parameter pack containing $v_1, v_2, \ldots, v_m$, we can implement the binder function object's call to `f` via:

```
return f(mu(a, v...)...);
```

This line of code first expands the inner ellipsis for `v` in the call to `mu`, resulting in `mu(a, v1, v2, /* up to */, vm)`. Then the outer ellipsis is expanded for `a`, resulting in:

```
return f(mu(a1, v1, v2, /* up to */, vm),
         mu(a2, v1, v2, /* up to */, vm),
         /* up to */
         mu(an, v1, v2, /* up to */, vm));
```

User impact : This feature follows solidly in the category "When you need it, you *really* need it", but will likely not see widespread use. The lack of this feature will result in a great deal of template metaprogramming require $\mathcal{O}(m \cdot n)$ instantiations.

Compiler impact : Implementation of this feature is expected to be relatively simple, requiring only that expansions be a recursive operation.

3.10 Pairwise expansion

Arbitrary expansion requires that only a single parameter pack occur within the pattern. Nested expansion extends this notion to several parameter packs so long as the expansions are nested. Pairwise expansion, on the other hand, allows two or more parameter packs at the same nesting level so long as all parameter packs are of the same length N. In this case, the parameter packs are expanded together into N arguments, with the i^{th} argument having replaced each parameter pack with the i^{th} value in that parameter pack.

Pairwise expansion permits, among other things, "zipping" two parameter packs into a single parameter pack containing pairs[2]:

```
template<... Args> struct bundle_t { typedef Args type; };

template<... Args> Args bundle(const Args& args...) { return args; }

template<... Args1>
class zipper
{
public:
  zipper(const Args1& args1...) : args1(args1) {}

  template<... Args2>
  typename bundle_t<std::pair<Args1, Args2>...>::type
  operator()(const Args2& args2...)
  { return bundle(std::pair<Args1, Args2>(args1, args2)...); }

private:
  Args1 args1;
};
```

More importantly, pairwise expansion in conjunction with move semantics [6] permits perfect argument forwarding [2] for an arbitrary number of arguments[3]:

```
template<typename F, ... Args>
auto apply(F&& f, Args&& args...) -> decltype(f(static_cast<Args&&>(args)...))
{ return f(static_cast<Args&&>(args)...); }
```

[2] Note that we also take advantage of first-class parameter packs in this example

[3] Additionally, we use `decltype` to provide perfect forwarding of the return type

User impact : This feature is extremely important in conjunction with the move semantics proposal [6], providing perfect argument forwarding.

Compiler impact : Implementation of this feature is expected to be easy, requiring only that the compiler be able to substitute for several parameter packs at once.

3.11 Folding expansion

Arbitrary and pairwise expansions permit transformations on parameter packs, but require that all transformations accept parameter packs of length N and produce N arguments. The folding expansion allows the transformed arguments to be "folded" (or combined) into a single argument. Folding expansions occur when the ellipsis operator is applied within square brackets [], and operates by applying the comma operator. For instance, given a parameter pack `args` containing values `a1, a2, /* up to */, aN` and some object `foo` with a suitable `operator[]`, the expression

```
foo[sum, args...]
```

expands (syntactically) to

```
foo[sum, a1, a2, /* up to */, aN]
```

The comma separators invoke the comma operator, which may be overloaded to perform arbitrary sequence folding operations, such as reversing the sequence or, in this case, "summing" the elements of the sequence.

User impact : This feature allows users to apply metaprogramming in a very succinct way, using the compiler to generate multiple copies of the same code for each value in a parameter pack without resorting to recursion in template metaprograms. This feature opens up a variety of code-generation possibilities for metaprograms.

Compiler impact : Implementation of this feature is not expected to require much effort beyond support for arbitrary expansions. It differs from arbitrary expansions only in the way that multiple arguments are evaluated: instead of creating an argument list for a function call or template instantiation, the compiler must create a chain of comma-operator invocations.

3.12 Arbitrary matching patterns

With basic variadic templates, variadic class template partial specializations cannot restrict the arguments placed into their template parameter pack and typesafe variadic functions can only decide the *cv*-qualifiers of their parameters or whether the parameters are accepted by reference or by value. Supporting arbitrary matching patterns involves allowing compound types within the patterns, e.g.,

```
template<... PointeeTypes>
void accept_pointers(PointeeTypes>* args...);

template<... PointeeTypes>
void accept_auto_ptrs(std::auto_ptr<PointeeTypes> ptrs...);
```

This feature may also be useful to constrain the kinds of template arguments a partial specialization will accept. For instance, we may define a compile-time vector of integers as such:

```
template<...> class int_vector;

template<... Ints> class int_vector<int Ints...> {};
```

User impact : Users will benefit from the ability to more precisely specify the form of function parameters and variadic class template partial specialization parameters.

Compiler impact : Implementing this feature is expected to require a moderate amount of effort, because compilers will need to transform the patterns provided into parameter types (in a manner similar to arbitrary expansion from Sec. 3.8).

3.13 Pairwise matching patterns

Pairwise matching patterns allow multiple parameter packs to occur within a partial specialization or function overload, so that they will be matched pairwise. For instance, we might take key-value pairs as arguments and separate the key types from the value types[4]:

```
template<... Keys, ... Values>
void foo(const std::pair<Keys, Values>& args...);
```

Pairwise matching patterns give a much more literal meaning to the declaration of the parameter packs in typesafe variadic functions, e.g.,

```
template<... Args> void foo(const Args& args...);
```

Here, the pattern is merely expanded pairwise for the appropriate number of parameters.

User impact : Pairwise matching patterns allow users to decompose arguments into several parameter packs while enforcing a strict structure on the incoming arguments.

Compiler impact : This feature is not expected to require much effort to implement beyond arbitrary matching patterns, because one need only be able to perform substitutions for multiple parameter packs at the same time.

4 Specials

This section describes three prepackaged sets of features that the committee might consider, briefly describing the interactions among the features and characteristics of the sets.

4.1 Common cases

Features:

- Basic variadic templates
- Length of parameter packs
- Get Nth element for parameter pack

The "common cases" special provides the most basic building blocks on which variadic templates can be build, plus two easy-to-implement primitive operations that are expected to be very common. From these building blocks, we can construct a tuple type that implements nearly every one of the other features via template metaprogramming.[5]. This option eliminates much of the preprocessor metaprogramming required to build a library such a `function` [1, §3.4] or `bind` [1, §3.3], but instead requires a large degree of template metaprogramming that will negatively impact compiler performance. However, basic variadic templates with a size query and access to the N^{th} element should only be moderately difficult to implement.

[4] One can imagine implementing named parameters via a similar mechanism

[5] Initializer lists cannot become parameter packs as far as we know

4.2 Library improvements

- Basic variadic templates
- First-class parameter packs
- Initializer lists are parameter packs
- Length of parameter packs
- Get Nth element for parameter pack
- Arbitrary expansion
- Nested expansion
- Pairwise expansion
- Arbitrary matching patterns

The "library improvements" special provides features that can improve the usability, clarity of implementation, and compile-time performance of libraries in the C++ standard and upcoming Library TR, along with many third-party libraries. In particular, it permits succinct, efficient implementations of `reference_wrapper` [1, §2.1], `result_of` [1, §3.1], `mem_fn` [1, §3.1], `bind` [1, §3.3], `function` [1, §3.4], and `tuple` [1, §6.1], along with third-party libraries such as MPL [5] or Lamba [9] and improvements to the standard containers. These primitives can eliminate much of the need for template metaprogramming and preprocessor metaprogramming for these libraries, reducing the burden on compilers. However, the implementation effort required to support variadic templates will be considerably increased due to the need to support arbitrary expansion and matching patterns.

4.3 Full suite

- Basic variadic templates
- First-class parameter packs
- Initializer lists are parameter packs
- Ellipsis operator for classes
- Length of parameter packs
- Get Nth element for parameter pack
- Arbitrary expansion
- Nested expansion
- Pairwise expansion
- Folding expansion
- Arbitrary matching patterns
- Pairwise matching patterns

The "full suite" special builds on the "library improvements" special by introducing the ellipsis operator for classes and pairwise matching patterns. With these extra features, particularly compile-time reflection, it will become possible to build increasingly powerful libraries and applications. Many metaprogramming tasks involving transformation on heterogeneous or compile-time sequences and folding those sequences into alternative results will no longer require extensive template metaprogramming, allowing more rapid development of metaprogramming libraries and further reducing the burden that metaprogramming typically places on compilers.

5 Revision history

- **Since N1603=04-0043**:
  - Moved to feature-based decomposition of the variadic templates design space.
  - Added notion of "folding expansion".
- **Since N1483=03-0066:**
  - Variadic templates no longer solve the forwarding problem for arguments.
  - Parameter packs are no longer tuples; instead, they are a compiler-specific first-class entities.
  - Introduced the ability to deduce a template parameter pack from a type (or list of template parameters).
  - Eliminated the `apply`, `integral_constant`, and `template_template_arg` kludges.

6 Acknowledgements

Discussions among Daveed Vandevoorde, David Abrahams, Mat Marcus, John Spicer, and Jaakko Järvi resulted in the new syntax for unpacking arguments using "...". This proposal has been greatly improved with their feedback. Mat Marcus suggested that the ellipsis operator could be applied to class types for marshalling, reflection, etc. David Abrahams noticed that "..." could be used for metaprogramming if we supported arbitrary expansion patterns.

References

[1] M. Austern. (draft) technical report on standard library extensions. Number N1660=04-0100 in ANSI/ISO C++ Standard Committee 2004-07 mailing, 2004.

[2] P. Dimov, H. Hinnant, and D. Abrahams. The forwarding problem: Arguments. Number N1385=02-0043 in ANSI/ISO C++ Standard Committee Pre-Santa Cruz mailing, October 2002.

[3] D. Gregor, J. Järvi, and G. Powell. Variadic templates. Number N1603=04-0043 in ANSI/ISO C++ Standard Committee Pre-Sydney mailing, 2004.

[4] D. Gregor, G. Powell, and J. Järvi. Typesafe variable-length function and template argument lists. Number N1483=03-0066 in ANSI/ISO C++ Standard Committee Post-Oxford mailing, 2004.

[5] A. Gurtovoy. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, July 2002.

[6] H. E. Hinnant, P. Dimov, and D. Abrahams. A proposal to add move semantics support to the C++ language. Number N1377=02-0035 in ANSI/ISO C++ Standard Committee Pre-Santa Cruz mailing, October 2002.

[7] J. Järvi. The Boost Tuples library. `http://www.boost.org/libs/tuple/doc/tuple_users_guide.html`, June 2001.

[8] J. Järvi. Proposal for adding tuple types to the standard library. Number N1403=02-0061 in ANSI/ISO C++ Standard Committee Post-Santa Cruz mailing, October 2002.

[9] J. Järvi and G. Powell. The Boost Lambda library. `http://www.boost.org/libs/lambda/doc/index.html`, March 2002.

[10] J. Järvi and B. Stroustrup. Decltype and auto (revision 3). Number N1607=04-0047 in ANSI/ISO C++ Standard Committee Pre-Sydney mailing, 2004.

[11] V. Karvonen and P. Mensonides. The Boost Preprocessor library. `http://www.boost.org/libs/preprocessor/doc/index.html`, July 2001.

[12] G. D. Reis and B. Stroustrup. Generalized initializer lists. Number N1509=03-0092 in ANSI/ISO C++ Standard Committee Pre-Kona mailing, September 2003.