

Variadic Templates

Douglas Gregor

Jaakkо Järvi

Gary Powell

Document number: N1603=04-0043

Date: February 17, 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 “bundling” using the same mechanism to pass an arbitrary number of function call arguments to a function in a typesafe manner.

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 the Boost Tuples library [10], for the specification of class template `std::tuple<>` in the library TR [11], and in the Boost metaprogramming library [8]. Unfortunately, this method 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 [14]. In all of these libraries (and presumably many more), an implementation based on variadic template would be shorter and would not suffer the limitations of the aforementioned implementation. The declaration of the `list<>` template above may be:

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

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. The lack of such a facility has resulted in odd syntax for formatting in the Boost.Format library [15]:

```
format("writing %1%, x=%2%: %3%-th try") % "toto" % 40.23 % 50
```

3 Syntax and Semantics

3.1 Variadic templates

The template parameter list to a function or class template can be declared to accept an arbitrary number of extra template arguments by terminating it with “...” optionally followed by an identifier through which these extra arguments can be accessed. Any number of template parameters may precede the “...”. Thus we can define, for instance, a class `Foo` accepting a template type parameter followed by an arbitrary number of template arguments as:

```
template<typename T, ... Args> class Foo;
```

Here, `T` is the name of the template type parameter and `Args` is the name of a “template parameter pack” containing the (possibly empty) set of template arguments given to `Foo`. Template parameter packs may contain template type, non-type, and template template arguments and are represented as an unspecified class type when a type is required.

Function templates may also be variadic templates in the same manner as class templates. Arguments can be specified explicitly when calling the function template, e.g.:

```
template<... Args> void f();
// ...
f<int, double>(); // Args will be contain <int, double>
```

The next section describes when the types of the elements in a value parameter pack can be deduced from the argument types of the call.

3.2 Typesafe variadic functions

When a function template header contains a template parameter pack, that template parameter pack may be used in the “type” of the final function parameter, allowing the function to accept a variable number of function parameters in a “value parameter pack” (or just “parameter pack”). For instance, a typesafe C++ `printf` may be declared as:

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

The ellipsis indicates that the function “parameter” `args` actually represents the “unpacked” parameters, so for a call to `printf` with $N + 1$ arguments, the signature is equivalent to the pseudocode:

```
template<typename T1, typename T2, ..., typename TN>
void printf(const char* format, const T1&, const T2&, ..., const TN&);
```

Note that `Args` is used as a placeholder in the type of the `args` “parameter”, and `Args` is replaced by each implicitly-named template parameter while `args` is replaced by each implicitly-named function parameter. Deduction of template parameter packs from function call arguments proceeds as it would if the parameters were explicitly enumerated as in the pseudocode declaration of `printf`. For instance, calling `printf("%e %i", 3.14159, 17)` would deduce `Args = <double,int>`, with the parameter types `const double&` and `const int&`.

3.3 Unpacking template and value parameter packs

The ellipsis operator unpacks all of the values (or template arguments) within a value (or template) parameter pack into a list of arguments for use in function call argument lists, template argument lists, and function types. For instance, given a template parameter pack `Elements`, we can create a `tuple` storing elements of those types via `tuple<Elements...>`. Should we wish to append an `int` to the end of the tuple, we may write `tuple<Elements..., int>`.

Value parameter packs are similarly expanded within a function call, so that, for instance, a `printf`-like function may pass the remaining arguments on like this:

```
template<typename T, ... Args>
void printf(const char* s, const T& next, const Args&... args)
{
    // print characters in s until we hit a formatting command
    // format ‘‘next’’ by converting it as appropriate
    // move s after the formatting command
    printf(s, args...);
}

void printf(const char* s)
{
    // print characters in s, verifying that there is no
    // formatting command
}
```

Note that when the parameter pack is empty, it expands to zero arguments (in the function call or template argument list). Thus, the recursion terminates for `printf` when `args` is empty, calling the second overload of `printf`.

The ellipsis operator is valid following any of these constructs:

- An argument in a function call argument list.
- An argument in a template argument list.
- The type of a parameter in a parameter declaration

In each case, the text of the argument (or parameter) preceding the ellipsis is treated as a pattern to be repeated for each element in the parameter pack as individual arguments (or parameters) within the current argument (or parameter) list. In the pattern, the parameter pack is a placeholder for the positions where the elements in the parameter pack will be substituted. Fig. 1 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, ..., XN>
list<int, X..., float>	list<int, X1, X2, ..., XN, float>
f(x...)	f(x1, x2, ..., xN)
f(17, x..., 3.14159)	f(17, x1, x2, ..., xN, 3.14159)
int (*)(X...)	int (*)(X1, X2, ..., XN)
list<typename add_pointer<X>::type...>	list<typename add_pointer<X1>::type, typename add_pointer<X2>::type, typename add_pointer<XN>::type>
f(x*x)	f(x1*x1, x2*x2, ..., xN*xN)
f(g(x, y...)...)	f(g(x1, y1, y2, ..., yM), g(x2, y1, y2, ..., yM), . . . g(xN, y1, y2, ..., yM))

Figure 1: Illustration of the expansions of types and expressions using template and value parameter packs.

Ellipsis 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). However, different parameter packs must not be present within the same argument text unless the ellipsis operators are nested. A program that attempts to apply the ellipsis operator to a type or value that is not a parameter pack is ill-formed.

3.4 Deducing template parameter packs from types

Template parameter packs can be deduced from template argument lists and function parameter lists, by packing the list of integral constant expressions, templates, and types into the template parameter pack.

Example: we can define a template `function_traits` that extracts the result type and argument types of a function:

```
template<...> class tuple;

template<...> struct function_traits;

template<typename R, ... Args>
struct function_traits<R(Args...)>
{
    typedef R result_type;
    typedef tuple<Args...> argument_types;
};
```

Example: we can write a function accepting a tuple with an arbitrary number of elements in it:

```
template<...> class tuple;

template<... Elements>
void eat_tuple(tuple<Elements...>);
```

Example: We can also split a template class into its template name and template arguments, a task that could drastically reduce the amount of code required to implement the meta-lambda facility of the Boost Metaprogramming Library [8]. For instance:

```
template<typename T> struct split_template_class;

template<template<... Args> class T>
    struct split_template_class<T<Args...> >;
```

3.5 Explicit template argument specification

If template arguments are explicitly specified when naming a function template, the function template header is terminated with a template parameter pack, and there are at least as many template arguments as there are template parameters to the function (not including the template parameter pack), all template arguments beyond the last one required for the function's template parameters will comprise the template parameter pack.

Example:

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

f<int, double>(); // OK: Args is <int, double>
f<>();           // OK: Args is empty
f();              // error: Args cannot be deduced
```

3.6 Partial ordering of variadic class template partial specializations

The class template partial specialization partial ordering rules will need to be augmented to include partial ordering with variadic templates. This is the only deviation from the purely syntactic nature of variadic templates. Intuitively, a binding to a parameter that falls into a template's variable-length argument list is weaker than a binding to a specified template parameter. For instance, given:

```
template<...> struct foo;
template<typename T, ...> struct foo<T>; // #1
template<typename T, typename U> struct foo<T, U>; // #2
```

Partial specialization #2 is more specialized than partial specialization #1 because #2 requires that the second template argument by a type (and not a template or acceptable literal).

Formalizing this notion, we introduce a new type of template parameter we call a template *variant* parameter. Template variant parameters cannot be declared explicitly, but occur implicitly as parameters for variadic templates. However, any template argument (type, nontype, or template) can be passed to a template *variant* parameter.

Paragraph 3 of 14.5.5.2 [temp.func.order] describes the rules for transforming a template for the purpose of partial ordering. To support partial ordering with variadic templates, introduce two additional bullets:

- If after removing the variable-length template parameter list designator ... from both templates the template parameter lists of the templates are of different length, append unique template variant parameters to the shorter template parameter list until the template parameter lists are of equal length.
- A binding of an argument to a variant parameter is weaker than a binding of an argument to a parameter of the same kind.

3.7 Overloading

The overloading rules need two minor changes to accomodate variadic template:

- Template variadic functions follow the same overloading rules as non-template variadic functions.

- If two overloads differ only in that one is a variadic template function and the other is a non-template variadic function, the template variadic function is more specialized.

Thus we prefer the typesafe variadic functions to unsafe variadic functions, even though these rules conflict with 13.3.3p1, which prefers non-template functions to template functions when the conversion sequences are otherwise equal.

3.8 The types of variadic templates

A template parameter pack is an unspecified, compiler-specific class type; value parameter packs are instances of the corresponding template parameter pack. Any program attempting to instantiate a value parameter pack whose corresponding template parameter pack contains nontype template parameters or template template parameters is ill-formed.

The type of an instantiation of a typesafe variadic function is equivalent to the type of a nontemplate function with all template parameters substituted and both template and value parameter packs unpacked completely. Revisiting the `printf` definition

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

and given a call `printf("'%i:%f'", 5, 3.14f)`, the type of this `printf` instantiation will be `void(const char*, const int&, const float&)`. This type compatibility is logical within the context of templates (i.e., it follows the existing behavior of instantiations of function templates) and useful in the implementation of the polymorphic function adaptors [7] proposal, which has placed the most stress on the formulation of variadic templates thus far.

3.9 The types of parameter packs

Each template parameter pack `<T1, T2, ..., TN` has an associated class type that contains unnamed values here denoted `t1, t2, ..., tN` of types `T1, T2, ..., TN`, respectively, and must provide the following when all `Ti` are types:

- A default constructor that default-initializes `t1, t2, ..., tN`, if the types `<T1, T2, ..., TN` have default constructors.
- A copy constructor that copy-constructs `t1, t2, ..., tN`, if the types `<T1, T2, ..., TN` have copy constructors.
- A constructor that accepts *N* arguments, of types `cv1 T1&, cv2 t2&, ..., cvN tN&`, where `cvi` is equivalent to the cv-qualifiers in the copy constructor of type `Ti`, and copy-constructs each `ti` from the corresponding argument.
- A copy assignment operator that assigns to each `ti` from the corresponding `ti` of the parameter pack on the right-hand side.

Value parameter packs may (but are not required to) be implemented with the following pseudocode:

```
template<typename T1, typename T2, ..., typename TN>
struct __pp
{
    // implicit default constructor
    // implicit copy constructor
    // implicit copy assignment operator
```

```
--pp(cv1 T1& a1, cv2 T2& a2, ..., cvN TN& aN)
  : t1(a1), t2(a2), ..., tN(aN) {}}

T1 t1;
T2 t2;
.
.
.

TN tN;
};
```

Implementations are permitted to perform an extra copy construction to construct a parameter pack from the arguments passed to a function, which may be unavoidable when variadic template functions are invoked via a function pointer. However, since we make no restrictions on the layout of parameter packs, and do not confine them to the same layout as, e.g., a structure containing the same types as in the example above, implementations may elide these copy constructions by performing parameter pack layout in a manner that coincides with the activation record for a particular platform.

The types associated with template parameter packs *may not* be used in conjunction with the ellipsis operator. For instance:

```
template<typename T> void bar(const T& x);

template<typename T1, ... Args>
void foo(const T1&, const Args& args)
{
    foo(args...); // okay
    bar(args); // okay
}

template<typename T>
void bar(const T& x)
{
    T y = x; // okay for instantiation
    foo(x...); // not okay: x is not a parameter pack
}
```

4 Examples

To demonstrate the use of variadic templates, this section gives skeletal implementations of four library components using variadic templates,. All of these proposals were accepted by the library working group into the first library TR, and are implementable to some degree in C++03. However, all of them would benefit from variadic templates in that the implementations can be more complete, more useful, and more readable than their C++03 counterparts. The following libraries are implemented:

- Member pointer adaptors [3]
- Tuples [11]
- Function object wrappers [7]
- Function object binder [4]

The implementations are not generally complete, but omit only features that are irrelevant to the discussion of variadic template. The implementations use several class templates (metafunctions) from the type traits proposal [16]. Additionally, we use the keyword `auto` to make the examples cleaner and easier by omitting the messy—but expressible—return types; however, this proposal does not strictly depend on the `decltype` proposal [13].

4.1 Building Blocks

The class template `enable_if` is used liberally to help guide the overload process by eliminating candidates from the overload set that would otherwise cause ambiguities or be selected when they shouldn't be. It is defined as:

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

We can count the number of arguments in a parameter pack with this simple class template, using `integral_constant` from the type traits proposal [16] (part of the first library technical report [1]):

```
// Count the number of arguments in a template parameter pack
template<...> struct count_args;

template<> struct count_args<> : integral_constant<int, 0> {};

template<typename T1, ... Args> struct count_args<T1, Args...>
    : integral_constant<int, (1 + count_args<Args...>::value)> {};
```

4.2 Member pointer adaptor implementation

The following code implements the enhanced member pointer adaptor proposal [3]. This implementation is complete.

```
// Adaptor for member function pointers
template<typename Class, typename FunctionType>
struct mem_fn_adaptor
{
    typedef typename function_traits<FunctionType>::result_type result_type;

    mem_fn_adaptor(FunctionType Class::*pmf) : pmf(pmf) {}

    template<... Args>
    result_type operator()(Class& object, Args... args) const
        { return (object.*pmf)(args...); }

    template<... Args>
    result_type operator()(const Class& object, Args&... args) const
        { return (object.*pmf)(args...); }

    template<typename T, ... Args>
    typename enable_if<(!is_base_of<Class, T>::value), result_type>::type
    operator()(T& ptr, Args&... args) const
        { return ((*ptr).*pmf)(args...); }

    template<typename T, ... Args>
    typename enable_if<(!is_base_of<Class, T>::value), result_type>::type
```

```

operator()(const T& ptr, Args&... args) const
{ return ((*ptr).*pmf)(args...); }

private:
    FunctionType Class::*pmf;
};

// Adaptor for member data pointers (for completeness only)
template<typename Class, typename T>
struct mem_ptr_adaptor
{
    typedef const T& result_type;

    mem_ptr_adaptor(T Class::*pm) : pm(pm) {}

    T& operator()(Class& object) const { return object.*pm; }
    const T& operator()(const Class& object) const { return object.*pm; }

    template<typename Ptr>
    typename enable_if<(!is_base_of<Class, Ptr>::value), const T&>::type
    operator()(Ptr& ptr) const { return (*ptr).*pm; }

    template<typename Ptr>
    typename enable_if<(!is_base_of<Class, Ptr>::value), const T&>::type
    operator()(const Ptr& ptr) const { return (*ptr).*pm; }

private:
    T Class::*pm;
};

template<typename Class, typename FunctionType>
typename enable_if<(is_function<FunctionType>::value),
                   mem_fn_adaptor<Class, FunctionType> >::type
mem_fn(FunctionType Class::*pmf)
{ return mem_fn_adaptor<Class, FunctionType>(pmf); }

template<typename Class, typename T>
typename enable_if<(!is_function<T>::value),
                   mem_ptr_adaptor<Class, T> >::type
mem_fn(T Class::*pm)
{ return mem_ptr_adaptor<Class, T>(pm); }

```

4.3 Tuple implementation

The following code implements the interesting portions of the Tuple proposal [11] using variadic as proposed here.

```

// Derivation from tuple_base indicates that a type is a tuple
class tuple_base {};

// Determine if type T is a tuple
template<typename T>
struct is_tuple
{ static const bool value = is_base_of<tuple_base, T>::value; };

// A tuple of arbitrary length N

```

```

template<... Elements>
class tuple : public tuple_base
{
public:
    static const size_t size = count_args<Elements...>::value;

    tuple() {}

    // enable_if condition ensures that we only match if N arguments are given
    explicit tuple(const Elements&... elements) : elements(elements) {}

    // enable_if condition only allows us to attempt implicit conversions
    // from tuples of the appropriate length
    template<typename Tuple>
    tuple(const Tuple& other,
          typename enable_if<(is_tuple<Tuple>::value && size == Tuple::size),
                           void>::type* = 0)
        : elements(other.elements...) {}

    // enable_if condition only allows us to attempt assignment from tuples of
    // the appropriate length
    template<typename Tuple>
    typename enable_if<(is_tuple<Tuple>::value && size == Tuple::size),
                       tuple&>::type
    operator=(const Tuple& other)
    {
        *this = tuple(other.elements...);
        return *this;
    }

    Elements elements;
};

// tuple_element
template<int I, typename Head, typename Tail>
struct tuple_element<I, tuple<Head, Tail...> >
    : tuple_element<I-1, tuple<Tail...> > { };

template<typename Head, typename Tail>
struct tuple_element<0, tuple<Head, Tail...> >
{
    typedef Head type;
};

// make_tuple implementation
template<typename T>
struct make_tuple_element_type
{
    typedef T type;
};

template<typename T>
struct make_tuple_element_type<reference_wrapper<T> >
{
    typedef T& type;
};

```

```

};

template<... Elements>
auto make_tuple(const Elements&... elements)
{
    typedef tuple<typename make_tuple_element_type<Elements>::type...>
        result_type;
    return result_type(elements...);
}

// tie implementation
template<... Elements>
tuple<Elements> tie(Elements&... elements)
{ return tuple<Elements&...>(elements...); }

```

4.4 Function implementation

This section implements the core interesting portions of class template `function` from the function object wrapper proposal [7]. The implementation of this library is much more subtle than that of the prior libraries, particularly the use of template parameter deduction and explicit specification of function template arguments to resolve the address of an overloaded function within the constructor. The implementation is complete with one exception: it does not properly support function pointers. The modifications to account for function pointers are simple but are outside the scope of this proposal.

```

struct bad_function_call : public std::exception {};

template<typename> struct function;

template<typename R, ... ArgumentTypes>
struct function<R(ArgumentTypes...)>
{
    typedef R result_type;

    template<typename Functor>
    auto functor_invoker(ArgumentTypes&... args) const
    { return (*static_cast<Functor*>(functor))(args...); }

    template<typename Functor>
    static void* functor_manager(void* ptr, bool clone)
    {
        const Functor* functor = static_cast<Functor*>(ptr);
        if (clone) return new Functor(*functor);
        else { delete functor; return 0; }
    }

    static void* trivial_manager(const void* ptr, bool)
    { return const_cast<void*>(ptr); }

    Function function::* invoker;
    mutable void*         functor;
    void* (*manager)(void*, bool /* clone or delete */);

    struct clear_type;
};

public:

```

```

function() : invoker(0), functor(0), manager(0) {}

function(const function& other) : invoker(0), functor(0), manager(0)
{
    if (other.invoker) {
        invoker = other.invoker;
        functor = other.manager(other.functor, true);
        manager = other.manager;
    }
}

template<typename F>
function(F f,
         typename enable_if<(!is_integral<F>::value, void*>::type = 0)
         : invoker(0), functor(0), manager(0)
{
    invoker = &function::functor_invoker<F>;
    functor = new Functor(f);
    manager = &functor_manager<F>;
}

template<typename F>
function(reference_wrapper<F> f) : invoker(0), functor(0), manager(0)
{
    invoker = &function::functor_invoker<F>;
    functor = &f.get();
    manager = &trivial_manager;
}

template<typename T, typename C>
function(T C::*pm) : invoker(0), functor(0), manager(0)
{ *this = std::mem_fn(pm); }

function(clear_type*) : invoker(0), functor(0), manager(0) {}

~function() { if (invoker) manager(functor, false); }

function& operator=(const function& other)
{
    function tmp(other);
    tmp.swap(*this);
    return *this;
}

template<typename F>
typename enable_if<(!is_integral<F>::value, function&>::type
operator=(F f)
{
    function tmp(f);
    tmp.swap(*this);
    return *this;
}

function& operator=(clear_type*)
{

```

```

    if (invoker) {
        manager(functor, false);
        invoker = 0;
        functor = 0;
        manager = 0;
    }
}

operator bool() const { return invoker; }

result_type operator()(ArgumentTypes... args) const
{
    if (!invoker) throw bad_function_call;
    return (this->*invoker)(args...);
}

void swap(function& other)
{
    using std::swap;
    swap(invoker, other.invoker);
    swap(functor, functor);
    swap(manager, manager);
}
};


```

4.5 Bind implementation

The following code implements the enhanced binder proposal [4] using variadic templates and building on the existing member pointer adaptor code. This implementation is complete, with two exceptions (both of which require uninteresting but nontrivial code, expressible in C++03):

- `result_type` is not specified in `binder`
- arity checking is not performed at bind time

```

// Placeholders
template<int N> struct placeholder {};

namespace placeholders {
    typedef placeholder<1> _1;
    typedef placeholder<2> _2;
    typedef placeholder<3> _3;
}

template<typename T> struct is_placeholder : integral_constant<size_t, 0> {};

template<int N>
struct is_placeholder<placeholder<N>> : integral_constant<size_t, N> {};

// Lambda
template<typename X> X& lambda(const reference_wrapper<X>& x)
    { return x.get(); }
template<typename X> X& lambda(X& x)
    { return x; }

// Get placeholder argument

```

```

template<typename T1, ... Args>
T1& get_placeholder_arg(placeholder<1>, T1& t1, Args&...)
{ return t1; }

template<typename T1, typename T2, ... Args>
T2& get_placeholder_arg(placeholder<2>, T1&, T2& t2, Args&...)
{ return t2; }

template<typename T1, typename T2, typename T3, ... Args>
T3& get_placeholder_arg(placeholder<3>, T1&, T2&, T3& t3, Args&...)
{ return t3; }

template<int N, typename T1, typename T2, typename T3, ... Args>
auto get_placeholder_arg(placeholder<N>, T1&, T2&, T3&, Args&... args)
{ return get_placeholder_arg(placeholder<N-3>(), args...); }

// Mu
template<typename X, typename Args> X& mu(reference_wrapper<X>& x, Args&)
{ return x.get(); }

template<typename X, typename Args>
auto
mu(X&, Args& args,
    typename enable_if<(is_placeholder<X>::value), void*>::type=0)
{
    return get_placeholder_arg(placeholder<is_placeholder<X>::value>(), args...);
}

template<typename X, typename Args>
auto
mu(X& x, Args& args,
    typename enable_if<(is_bind_expression<X>::value), void*>::type = 0)
{ return x(args...); }

template<typename X, typename Args> X& mu(X& x, Args&)
{ return x; }

// Return type of bind(...)
template<typename F, ... BoundArgs>
struct binder
{
    binder(const F& f, const BoundArgs&... bound_args)
        : f(f), bound_args(bound_args...) {}

    template<... Args> auto operator()(Args&... args)
    { return lambda(f)(mu(bound_args, args...)...); }

    template<... Args> auto operator()(Args&... args) const
    { return lambda(f)(mu(bound_args, args...)...); }

private:
    F f;
    BoundArgs bound_args;
};

```

```

// Helper to construct binder objects
template<typename F, ... BoundArgs>
binder<F, BoundArgs...> make_binder(F f, const BoundArgs&... bound_args)
{ return binder<F, BoundArgs...>(f, bound_args...); }

// Determine if a type T is a binder type
template<typename T> struct is_bind_expression : integral_constant<bool, false> {};

template<typename F, ... BoundArgs>
struct is_bind_expression<binder<F, BoundArgs...>> : integral_constant<bool, true> {};

// Bind overload for function objects and function pointers
template<typename F, ... BoundArgs>
auto bind(F f, const BoundArgs&... bound_args)
{ return binder<F, BoundArgs...>(f, bound_args...); }

// Bind overload for member pointers (both function and data)
template<typename Class, typename T, ... BoundArgs>
auto bind(T Class::*pm, const BoundArgs&... bound_args)
{ return make_binder(mem_fn(f), bound_args...); }

```

5 Alternatives & Extensions

5.1 Ellipsis operator for ordinary class types

One extension of the ellipsis operator would permit it to be applied for any 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. There are several open questions to be considered for this extension:

1. How should base classes be handled? Base classes need to be forbidden (severely limiting), enumerated as arguments in some defined order, or have their fields enumerated. The middle option provides the most information, assuming that some method exists to distinguish a base class from a member.
2. How should protected/private members or base classes be handled? Enumerating protected and private members is essential for many applications of compile-time reflection, but through this operator one can easily (and even accidentally) break encapsulation.
3. Would it lead to confusing situations? The proposed ellipsis operator is very different from other C++ operators, because it essentially permits macro-like expansion via template parameters. If extended to support arbitrary class types, problems involving the use of the ellipsis operator might be harder to debug, because, e.g., one may accidentally use ... on a type that was never meant to be unbundled. Without this extension, diagnosing errors in the use of ... would be very easy and very effective.
4. What syntax should we use? If the ellipsis operator applies to any class type, it is not clear what types the ... binds to. We will require a new syntax that explicitly states which types to unpack.

The combination of these factors (especially #2) leads us to believe that reflection should be supported differently, although it is very likely that the underlying mechanism will take advantage of variadic templates.

5.2 Parameter pack Nth element and size operators

Access to the Nth element of a parameter pack requires a linear number of instantiation, as in the `tuple_element` definition in Sec. 4.3. A further extension may provide a new operator that accesses the Nth element of a

parameter pack without the linear number of instantiations. The suggested syntax for such an operator involves two new operators: `.[]` to access values and `.<>` to access types. For instance:

```
template<int N, typename Tuple> struct tuple_element;

template<int N, ... Elements>
struct tuple_element<tuple<Elements...> >
{
    typedef Elements.<N> type;
};

template<int N, ... Elements>
Elements.<N>& get(tuple<Elements...>& t)
{ return t.[N]; }

template<int N, ... Elements>
const Elements.<N>& get(const tuple<Elements...>& t)
{ return t.[N]; }
```

In addition, support for an operation returning the length (i.e., number of elements) in a parameter pack would permit the entirety of the tuple proposal [11] (plus other metaprogramming tasks) to be implemented without incurring a large number of instantiations. The `sizeof` operator, when provided with an argument followed by an ellipsis, could provide the number of elements in a parameter pack. For instance:

```
template<typename Tuple> struct tuple_size;

template<... Elements>
struct tuple_size<tuple<Elements...> >
: integral_constant<size_t, sizeof(Elements...)> { };
```

While these extensions may be useful, we believe that they introduce a large amount of syntactic sugar that will not be very important in practice. For the cases such as `tuple_element` and `tuple_size` that incur a linear number of instantiations without these extensions, the definitions can be manually “unrolled” to alleviate these problems, as is done for the `mu` function in Sec. 4.5.

5.3 Overloadable 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 down side to allowing overloading of `operator...` is that it will require a new syntax for the `...` operator. For instance, consider the following code:

```
template<...> struct bar;

template<typename T1, typename T2>
struct foo
{
    typedef foo<bar<T1, T2>...> foobar;
};
```

When neither or both of `T1` and `T2` have an `operator...`, the instantiation is ill-formed. This ambiguity delays checking of the use of the ellipsis operator in templates until instantiation time, so we have decided not to include it in this version of the proposal.

5.4 Extended template argument deduction

One potential extension to this proposal we have considered is to allow parameter packs to occur before the end of the template header (so that multiple template parameter packs may be declared in a single template header). This extension would then allow one to deduce multiple parameter packs for, e.g., a function call:

```
struct PivotType {};  
  
template<... Head, ... Tail>  
void split(Head... h, PivotType p, Tail... t);
```

Another example of this would be a “`pop_back`” operation on a parameter pack (here we place the result in a tuple):

```
template<... Elements, typename T>
tuple<Elements> pop_back(Elements... elements, const T&)
{ return tuple<Elements...>(elements); }
```

While we acknowledge that there may be some benefit to introducing these extensions, we feel that the benefits are not sufficient to outweigh the cost of introducing yet more overloading rules to account for this cases. We feel that the few operations that would benefit may better be accomplished by working with a true heterogeneous data structure such as a tuple.

6 Revision history

- 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.

7 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. David Abrahams noticed that “...” could be used for metaprogramming if it could be applied to constructs other than parameter packs.

References

- [1] M. Austern. (draft) technical report on standard library extensions. Number N1540=03-0123 in ANSI/ISO C++ Standard Committee Post-Kona mailing, November 2003.
- [2] P. Dimov. The Boost Bind library. <http://www.boost.org/libs/bind/bind.html>, August 2001.
- [3] P. Dimov. A proposal to add an enhanced member pointer adaptor to the library technical report. Number N1432=03-0014 in ANSI/ISO C++ Standard Committee Pre-Oxford mailing, March 2003.
- [4] P. Dimov, D. Gregor, J. Järvi, and G. Powell. A proposal to add an enhanced binder to the library technical report. Number N1455=03-0038 in ANSI/ISO C++ Standard Committee Post-Oxford mailing, April 2003.
- [5] 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.
- [6] D. Gregor. The Boost Function library. <http://www.boost.org/doc/html/function.html>, June 2001.
- [7] D. Gregor. A proposal to add a polymorphic function object wrapper to the standard library. Number N1402=02-0060 in ANSI/ISO C++ Standard Committee Post-Santa Cruz mailing, October 2002.
- [8] A. Gurtovoy. The Boost MPL library. <http://www.boost.org/libs/mpl/doc/index.html>, July 2002.
- [9] 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.
- [10] J. Järvi. The Boost Tuples library. http://www.boost.org/libs/tuple/doc/tuple_users_guide.html, June 2001.
- [11] 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.
- [12] J. Järvi and G. Powell. The Boost Lambda library. <http://www.boost.org/libs/lambda/doc/index.html>, March 2002.
- [13] J. Järvi and B. Stroustrup. Mechanisms for querying types of expressions: decltype and auto revisited. Number N1527=03-0110 in ANSI/ISO C++ Standard Committee Pre-Kona mailing, September 2003.
- [14] V. Karvonen and P. Mensonides. The Boost Preprocessor library. <http://www.boost.org/libs/preprocessor/doc/index.html>, July 2001.
- [15] S. Krempp. The Boost Format library. <http://www.boost.org/libs/format/index.htm>, January 2002.
- [16] J. Maddock. A proposal to add type traits to the standard library. Number N1424=03-0006 in ANSI/ISO C++ Standard Committee Pre-Oxford mailing, March 2003.