

Variadic Templates (Revision 3)

Authors: Douglas Gregor, Indiana University
Jaakko Järvi, Texas A&M University
Gary Powell, Amazon
Document number: N2080=06-0150
Revises document number: N1704=04-0144
Date: September 9, 2006
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Contents

1	Introduction	1
2	A Variadic Templates Primer	2
2.1	Variadic Class Templates	3
2.2	Packing and Unpacking Parameter Packs	3
2.3	Variadic Function Templates	4
2.4	Packing and Unpacking, Revisited	5
2.5	Is the Meta-Operator “...” Necessary?	7
2.6	Learning More	7
3	Interactions with Other C++0x Proposals	7
3.1	Rvalue References	7
3.2	Initializers	8
3.3	Concepts	9
A	Extended Examples	10
A.1	Tuples	10
A.1.1	Creation functions	11
A.1.2	Helper classes	11
A.1.3	Element access	12
A.1.4	Relational operators	12
A.2	Polymorphic Function Wrappers	13
A.3	Function Object Binders	15
B	Grammar	18
B.1	Expressions	18
B.2	Declarators	18
B.3	Templates	19
C	Revision history	19

1 Introduction

Many function and class templates are naturally expressed with a variable number of parameters. For instance, the `tuple` facility of TR1 [2, §6.1] is a generalized `pair` that can store any number of values:

`tuple<int, float>` stores two values, `tuple<int, float, string>` stores three values, etc. A C++-friendly, type-safe variant of C's `printf()` would naturally accept any number of arguments containing values to be printed.

We are unable to directly express the notion that a function or class template takes an arbitrary number of arguments in current C++. Instead, we use various emulations, including extra defaulted template parameters and sets of overloaded function templates (each with a different number of parameters). In the worst case, the amount of code repetition required by these emulations leads us to preprocessor metaprogramming [13], which has become increasingly prevalent in recent years, despite our best attempts to supercede the preprocessor. Preprocessor metaprogramming to emulate “variadic” templates is necessary to eliminate a large amount of code duplication, but comes with many problems: GCC's implementation of TR1's function objects chapter [2, §3] originally supported up to 20 parameters, but excessive compilation times forced a move back to 10 parameters (the minimum required by TR1). The implementation itself uses at least 27 different macros, most of which contain template parameters and template arguments in various forms, and includes the same “pattern” header 11 times with different definitions of these macros. The code still requires far too much time to preprocess and parse, and is nearly unintelligible.

Variadic templates provide a way to express function and class templates that take an arbitrary number of arguments. Variadic templates are a small, pure language extension that achieves the following goals:

- Enables the simple and direct expression of class and function templates that accept a variable number of parameters, such as the Library TR1 facilities `tuple`, `function`, `bind()`, and `mem_fn()`.
- Helps reverse the trend toward *more* preprocessor use, by eliminating the need for most uses of preprocessor metaprogramming.
- Enables the implementation of a type-safe, C++-friendly `printf()`, bringing the “March of Progress” full circle ¹.
- Improves argument forwarding [4] for a variable number of arguments, enabling the implementation of better factories and adaptors.
- Requires only minimal changes to the C++ template system and existing compilers. A complete implementation of variadic templates in the GNU C++ compiler required only a week to develop.

N2087 [5] provides a brief introduction to variadic templates, and is recommended for readers not familiar with the ideas of variadic templates. This document describes variadic templates in more depth, including their interaction with other C++0x features and proposals. Appendix A provides several extended examples drawn from Library TR1, illustrating how variadic templates can be used to drastically simplify the implementations of these library components. A complete implementation of variadic templates is provided as a patch against the GNU C++ compiler, available online at <http://www.generic-programming.org/~dgregor/cpp/variadic-templates.html>.

2 A Variadic Templates Primer

Variadic templates allow a class or function template to accept some number (possibly zero) of “extra” template arguments. This behavior can be simulated for class templates in C++ via a long list of defaulted template parameters, e.g., a typical implementation for `tuple` looks like this:

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

This `tuple` type can be used with anywhere from zero to N template arguments. Assuming that N is “large enough”, we could write:

```
typedef tuple<char, short, int, long, long long> integral_types;
```

¹<http://www.horstmann.com/>

Of course, this `tuple` instantiation actually contains $N - 5$ unused parameters at the end, but presumably the implementation of `tuple` will somehow ignore these extra parameters. In practice, this means changing the representation of the argument list or providing a host of partial specializations:

```
template<>
class tuple<> { /* handle zero-argument version. */ };

template<typename T1>
class tuple<T1> { /* handle one-argument version. */ };

template<typename T1, typename T2>
class tuple<T1, T2> { /* handle two-argument version. */ };
```

This technique is used by various C++ libraries [10,11,8,1]. Unfortunately, it leads to a huge amount of code repetition, very long type names in error messages (compilers tend to print the defaulted arguments) and very long mangled names. It also has a fixed upper limit on the number of arguments that can be provided, which is not extensible without resorting to preprocessor metaprogramming [13] (which has its own fixed upper limits).

2.1 Variadic Class Templates

Variadic templates let us explicitly state that `tuple` accepts one or more template type parameters: template above becomes:

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

The ellipsis to the left of the `Elements` identifier indicates that `Elements` is a *template type parameter pack*. A *parameter pack* is a new kind of entity introduced by variadic templates. Unlike a parameter, which can only bind to a single argument, multiple arguments can be “packed” into a single parameter pack. With the `integral_types typedef` above, the `Elements` parameter pack would get bound to a list of template arguments containing `char`, `short`, `int`, `long`, and `long long`.

Aside from template parameter packs for template type arguments, variadic class templates can also have parameter packs for non-type and template template arguments. For instance, we could declare an `array` class that supports an arbitrary number of dimensions:

```
template<typename T, unsigned PrimaryDimension, unsigned... Dimensions>
class array { /* implementation */ };

array<double, 3, 3> rotation_matrix; // 3x3 rotation matrix
```

For `rotation_matrix`, the template parameters for `array` are deduced as `T=double`, `PrimaryDimension=3`, and `Dimensions` contains 3. Note that the argument to `Dimensions` is actually a list of template non-type arguments, even though that list contains only a single element.

2.2 Packing and Unpacking Parameter Packs

It is easy to declare a variadic class template, but so far we haven’t done anything with the extra arguments that are passed to our template. To do anything with parameter packs, one must *unpack* (or *expand*) all of the arguments in the parameter pack into separate arguments, to be forwarded on to another template. To illustrate this process, we will define a template that counts the number of template type arguments it was given:

```
template<typename... Args> struct count;
```

Our first step is to define a basis case. If `count` is given zero arguments, this full specialization will set the value member to zero:

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

Next we want to define the recursive case. The idea is to peel off the first template type argument provided to `count`, then pack the rest of the arguments into a template parameter pack to be counted in the final sum:

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

Whenever the ellipsis occurs *to the right* of a template argument, it is acting as a meta-operator that unpacks the parameter pack into separate arguments. In the expression `count<Args...>::value`, this means that all of the arguments that were packed into `Args` will become separate arguments to the `count` template.

The use of the ellipsis for unpacking a parameter pack also occurs in the partial specialization, `count<T, Args...>`, where template argument deduction packs “extra” parameter passed to `count` into `Args`. For instance, the instantiation of `count<char, short, int>` would select this partial specialization, binding `T` to `char` and `Args` to a list containing `short` and `int`.

The `count` template is a recursive algorithm. The partial specialization pulls off one argument (`T`) from the template argument list, placing the remaining arguments in `Args`. It then computes the length of `Args` by instantiating `count<Args...>` and adding one to that value. When `Args` is eventually empty, the instantiation `count<Args...>` selects the full specialization `count<>` (which acts as the basis case), and the recursion terminates.

The use of the ellipsis operator for unpacking allows us to implement `tuple` with variadic templates:

```
template<typename... Elements> class tuple;
template<typename Head, typename... Tail>
class tuple<Head, Tail...> : private tuple<Tail...> {
    Head head;
public:
    /* implementation */
};
template<>
class tuple<> {
    /* zero-tuple implementation */
};
```

Here, we use recursive inheritance to store a member for each element of the parameter pack. Each instance of `tuple` peels off the first template argument, creates a member `head` of that type, and then derives from a `tuple` containing the remaining template argument.

2.3 Variadic Function Templates

C’s variable-length function parameter lists allow an arbitrary number of “extra” arguments to be passed to the function (sometimes called a “varargs function” or a “variadic 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. For instance, this simple code will likely crash at run-time:

```
const char* msg = "The value of %s is about %g (unless you live in %s)";
printf(msg, std::string("pi"), 3.14159, "Indiana");2
```

Variadic templates also allow an arbitrary number of “extra” arguments to be passed to a function, but retain complete type information. For instance, the following function accepts any number of arguments, each of which may have a different type:

```
template<typename... Args> void eat(Args... args);
```

The `eat()` function is a variadic template with template parameter pack `Args`. However, the function parameter list is more interesting: the ellipsis *to the left* of the identifier `args` indicates that `args` is a *function parameter pack*. Function parameter packs can accept any number of function arguments, the types of which

²See http://www.agecon.purdue.edu/crd/Localgov/Second%20Level%20pages/Indiana_Pi_Story.htm for an explanation.

are in the template type parameter pack `Args`. Consider, for instance, a call `eat(17, 3.14159, string("Hello!"))`: the three function call arguments will be packed, so `args` contains 17, 3.14159 and `string("Hello!")` while `Args` contains the types of those arguments: **int**, **double**, and **string**. The effect is precisely the same as if we had just written a three-argument version of `eat()` like the following, except without the limitation to three arguments:

```
template<typename Arg1, typename Arg2, typename Arg3>
void eat(Arg1 arg1, Arg2 arg2, Arg3 arg3);
```

Variadic function templates allow us to express a type-safe `printf()` that works for non-POD types. The following code uses variadic templates to implement a C++-friendly, type-safe `printf()`:

```
void printf(const char* s) {
    while (*s) {
        if (*s == '%' && *++s != '%')
            throw std::runtime_error("invalid format string: missing arguments")
        std::cout << *s++;
    }
}

template<typename T, typename... Args>
void printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *++s != '%') {
            std::cout << value;
            return printf(++s, args...);
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

There is only one innovation in this implementation of `printf()`, but we've seen it before: in the recursive call to `printf()`, we unpack the function parameter pack `args` with the syntax `args...`. Just like unpacking a template parameter pack (e.g., for `tuple` or `count`), the use of the ellipsis meta-operator *on the right* unpacks the parameter pack into separate arguments. We've again used recursion, peeling an argument off the front of the list, processing it, then recursing to handle the rest of the arguments.

2.4 Packing and Unpacking, Revisited

When using the ellipsis meta-operator on the right to unpack a template parameter pack, we can employ arbitrary *unpacking patterns* containing the parameter pack. These patterns will be expanded for each argument in the parameter pack, or used to direct how template argument deduction proceeds. For example, our `printf()` function above should really have taken its arguments be **const** reference, because copies are unnecessary and could be expensive. So, we change the declaration of `printf()` to:

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

In fact, when the ellipsis operator is used on the right, the argument can be arbitrarily complex and can even include multiple parameter packs. In conjunction with rvalue references [9], this allows us to implement perfect forwarding for any number of arguments [4], for instance allowing an allocator's `construct()` operation to work even when constructing a noncopyable type:

```
template<typename T>
struct allocator {
    template<typename... Args>
    void construct(T* ptr, Args&&... args) {
        new (ptr)(static_cast<Args&&>(args)...);
    }
};
```

In this example, we use placement new and call the constructor by **static_casting** each argument in **args** to an rvalue reference of its corresponding type in **Args**. That pattern will be expanded for all arguments, with both **Args** and **args** expanding concurrently. When used with two arguments, it essentially produces the same code as the following two-argument version of **construct()**:

```
template<typename T>
struct allocator {
    template<typename Arg1, typename Arg2>
    void construct(T* ptr, Arg1&& arg1, Arg2&& arg2) {
        new (ptr)(static_cast<Arg1&&>(arg1), static_cast<Arg2&&>(arg2));
    }
};
```

This same approach can be used to mimic inheritance of constructors, e.g.,

```
template<typename Base>
struct adaptor : public Base {
    template<typename... Args>
    adaptor(Args&&... args) : Base(static_cast<Args&&>(args)...) { }
};
```

An unpacking pattern can occur at the end of any template argument list. This also means that multiple parameter packs can occur in the same template. For instance, one can declare a **tuple** equality operator than can compare tuples that store different sets of elements:

```
template<typename... Elements1, typename... Elements2>
bool operator==(const tuple<Elements1...>& t1, const tuple<Elements2...>& t2);
```

This function is a variadic template (it contains template parameter packs), but the length of the function parameter list is fixed (it contains no function parameter packs). It uses two template parameter packs, one for the arguments to each **tuple**. If we wanted our equality comparison to only consider tuples with precisely the same element types, we would have written:

```
template<typename... Elements>
bool operator==(const tuple<Elements...>& t1, const tuple<Elements...>& t2);
```

Similarly, one can use a template type parameter pack at the end of a function parameter list in a function type, e.g.,

```
template<typename Signature> struct result_type;
// Matches any function pointer3
template<typename R, typename Args...>
struct result_type<R(*) (Args...)> {
    typedef R type;
};
// Matches a member function pointer with no cv-qualifiers
template<typename R, typename Class, typename Args...>
struct result_type<R(Class::*) (Args...)> {
    typedef R type;
};
```

Parameter packs can be unpacked in one additional context: the parenthesized argument to a **sizeof** expression. When used in this way, **sizeof** returns the length of the parameter pack, which may be zero:

```
template<typename... Args>
struct count {
    static const int value = sizeof(Args...);
};
```

Although not strictly necessary (we can implement **count** without this feature), checking the length of a parameter pack is a common operation that deserves a simple syntax. Moreover, this operation may become necessary for type-checking reasons when variadic templates are combined with concepts; see Section 3.3.

³Unless that function is uses a C-style variadic argument list. Then we'd need two ellipses!

2.5 Is the Meta-Operator “...” Necessary?

Parameter packs are a new kind of entity in C++ and can only be used in very limited ways, e.g., as arguments in a template argument list or function call or as parameter types in a function type. Moreover, the only way to actually use a parameter pack is to unpack it with an ellipsis on the right, e.g.,

```
return printf(++s, args...);
```

With so few operations on parameter packs, can we eliminate the need for the ellipsis on the right, e.g.,

```
return printf(++s, args); // okay??
```

In fact, we cannot eliminate the need for the ellipsis on the right, because that would introduce unresolvable ambiguities when more interesting patterns come to place. For instance, consider the following metafunction, `one_tuples`, that builds a tuple of 1-tuples from the arguments it is given, e.g., `one_tuples<int, float>::type` is `tuple<tuple<int>, tuple<float> >`:

```
template<typename... Args>
struct one_tuples {
    typedef tuple<tuple<Args>...> type;
};
```

Now, consider a similar metafunction, `nest_tuple`, that builds a tuple containing a single tuple, which itself has all of the arguments the original metafunction was given:

```
template<typename... Args>
struct nest_tuple {
    typedef tuple<tuple<Args>...> > type;
};
```

If we take away the ellipsis on the right from the definitions of each of the metafunctions, the `typedefs` for type become identical, and there would be no way to resolve the ambiguity:

```
typedef tuple<tuple<Args> > type;
```

2.6 Learning More

The best ways to understand a new language feature are to look at examples and to try it out for yourself. Appendix A contains extended examples illustrating how to use variadic templates to implement various libraries from TR1. Our patches to the GNU C++ compiler are available online at <http://www.generic-programming.org/~dgregor/cpp/variadic-templates.html>.

3 Interactions with Other C++0x Proposals

Variadic templates are a pure extension to the C++ template system, but they can be used in conjunction with other features that have been proposed for C++0x. In this section, we discuss some of these interactions.

3.1 Rvalue References

Rvalue references [9] prefer to bind to rvalues (including `const` rvalues) rather than lvalues. Among other things, rvalue references allow one to implement move semantics and perfect forwarding of function arguments [4]. The latter is crucial for user-friendly implementations of facilities such as `bind()` [2, 3.6]. Rvalue references and variadic templates work well together, enabling perfect argument forwarding for any number of function arguments:

```
template<typename F, typename T1, typename T2>
void forward(F&& f, T1&& a1, T2&& a2) {
    f(static_cast<T1&&>(a1), static_cast<T2&&>(a2));
}
```

3.2 Initializers

Variadic templates can interact with generalized initializer lists [15, 14] in several different ways. In some cases, variadic templates make it possible to use generalized initializer lists in new ways, for instance, by allowing a tuple to be initialized with an initializer list:

```
tuple<int, float, string> t = {42, 2.71828, "Hello, world!"};
```

This code “just works” with the tuple element presented in this proposal and the initializer list rules in [15, §1.2]. With variadic templates, one can even write constructors that take a completely arbitrary initializer list:

```
class any_list {
    template<typename... Values>
    any_list(const Values&... values);
};
```

```
any_list values = {42, 2.71828, "Hello, World!"};
```

The ability to accept an arbitrary initializer list gives an alternative formulation of *sequence constructors* [15, §1.3], using only variadic templates. For instance:

```
template<typename T>
class vector {
public:
    template<typename... Values>
    vector(const Values&... values) {
        reserve(sizeof(values...));
        push_back_all(values...);
    }
private:
    void push_back_all() { }
    template<typename... Values>
    void push_back_all(const T& value, const Values&... values) {
        push_back(value);
        push_back_all(values...);
    }
};

vector<int> v = { 17, 42, 3.1 }; // okay!
```

This emulation of sequence constructors is not perfect. The “sequence constructor” is really a constructor taking any number of arguments, and is invoked when an initializer list is broken down into separate arguments. The constructor needs to be a template, and needs to be implemented recursively, whereas a “real” sequence constructor is a non-template that receives a range of pointers. Thus, sequence constructors are easier to use when initializing storage when all of the parameters have the same type, whereas variadic templates are better when the stored values can have varying types.

We are considering whether the addition of function parameter packs with concrete types might offer an alternative to sequence constructors. One might imagine writing the following `vector` constructor, which accepts an arbitrary number of `T` arguments. In this case, the function parameter pack could just behave like an array:

```
template<typename T, typename Allocator>
class vector {
    vector(T... values) {
        reserve(sizeof(values...));
        for (std::size_t i = 0; i < sizeof(values...); ++i)
            push_back(values[i]);
    }
};
```



```
vector<int> v = { 17, 42, 3.1 }; // okay!
```

3.3 Concepts

Concepts [6] introduce improved separate type checking capabilities for templates, simplifying their use and implementation. Because concepts are a non-trivial change to the template system, they do interact with variadic templates. The biggest questions are how we can place concept constraints on variadic templates, how we can check that those constraints are met, and how we can type-check the definition of a variadic template.

Placing concept constraints on variadic templates is rather straightforward. We need only introduce a new context for using the ellipsis meta-operator on the right in the `where` clause, so that constrained templates can place constraints on all of the types in a template parameter pack, e.g.,

```
auto concept OutputStreamable<typename T> {
    ostream& operator<<(ostream&, const T&);
};

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

Likewise, we could use the “shortcut” syntax for concept constraints to declare template type parameter packs with concept requirements:

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

When invoking `printf()`, template argument deduction will determine which template arguments are packed into the template parameter pack. The compiler need only verify that each of these template arguments are `OutputStreamable`.

Variadic templates will solve some warts in the concept specification for the standard library. For instance, the `CallableN` concepts (where `N` goes from 0 up to some implementation-defined maximum) currently have to be separate concepts, each having `N+1` concept arguments [7]. We could replace this set of concepts with a single, variadic concept:

```
auto concept Callable<typename F, typename... Args> {
    typename result_type;
    result_type operator(F&, const Args&...);
};
```

The `Callable` concept is particularly useful when introducing concepts into the function object facilities of TR1 [2, §3].

There are some open questions regarding the type-checking of variadic template definitions, but we are confident that the two features can peacefully coexist in C++0x. In the future, we will address these questions and consider integrating our concept compiler (ConceptGCC) with our variadic templates implementation.

Acknowledgements

Discussions among Daveed Vandevor, 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 marshallng, reflection, etc. David Abrahams noticed that “...” could be used for metaprogramming if we supported arbitrary expansion patterns. The presentation of this proposal was greatly improved thanks to comments from Andrei Alexandrescu.

A Extended Examples

This section contains some extended examples that illustrate how several of the libraries in TR1 [2] can be implemented using variadic templates. The implementation of each of these libraries requires a large amount of redundant code, of generated via preprocessor metaprogramming or external scripts. We annotate these examples to illustrate

All of these examples have been verified to work using our modified GNU C++ compiler.

A.1 Tuples

Tuples [2, §6.1] are generalized pairs. They can hold zero or more values, the types of which are specified as template arguments.

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

Tuples with zero arguments are easy to handle, because an empty specialization suffices:

```
template<> class tuple<> { };
```

The most important part of the tuple implementation is the recursive case, where we peel off the first argument (the Head) to be stored in the `m_head` member, then derive from a tuple containing the remaining arguments (the Tail).

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
: private tuple<Tail...> {
    typedef tuple<Tail...> inherited;
public:
    tuple() { }

    // implicit copy-constructor is okay
    // Construct tuple from separate arguments.
    tuple(typename add_const_reference<Head>::type v,
          typename add_const_reference<Tail>::type... vtail)
        : m_head(v), inherited(vtail...) { }

    // Construct tuple from another tuple.
    template<typename... VValues>
    tuple(const tuple<VValues...>& other)
        : m_head(other.head()), inherited(other.tail()) { }

    template<typename... VValues>
    tuple& operator=(const tuple<VValues...>& other)
    {
        m_head = other.head();
        tail() = other.tail();
        return *this;
    }

    typename add_reference<Head>::type head() { return m_head; }
    typename add_reference<const Head>::type head() const { return m_head; }
    inherited& tail() { return *this; }
    const inherited& tail() const { return *this; }

protected:
    Head m_head;
};
```

Most of the operations are just recursive operations. The signature of the tuple constructor for separate arguments supports construction of tuples with, e.g., `tuple<int, float>(17, 3.14159)`. The `add_const_reference` trait turns a type `T` into `T const&` unless `T` is a reference, in which case it is left alone. Note that this constructor is not a member template, but it does have a function parameter pack at the end.

A.1.1 Creation functions

Tuples can be created with two helper functions, `make_tuple()` and `tie()`, which return new tuples from a set of arguments. `make_tuple()` creates a tuple that stores copies of its arguments. This behavior can be overridden using `reference_wrapper`, which indicates that a reference to the value should be stored. The complete code for `make_tuple()` follows.

```
template<typename T>
struct make_tuple_result {
    typedef T type;
};

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

template<typename... Values>
tuple<typename make_tuple_result<Values>::type...>
make_tuple(const Values&... values) {
    return tuple<typename make_tuple_result<Values>::type...>(values...);
}
```

Note that `make_tuple()`'s return type uses a non-trivial unpacking pattern, where we apply `make_tuple_result` to each of the types in the `Values` parameter pack before forming the resulting tuple. In contrast, `tie()` is much more simple and direct, because it does not need to perform any transformations on the types:

```
template<typename... Values>
tuple<Values&...> tie(Values&... values) {
    return tuple<Values&...>(values...);
}
```

A.1.2 Helper classes

Class template `tuple` has a few helper classes that determine various properties of tuples. The first computes the number of values stored in a tuple:

```
template<typename Tuple>
struct tuple_size;

template<typename... Values>
struct tuple_size<tuple<Values...> > {
    static const int value = sizeof(Values...);
};
```

The `tuple_element` class template retrieves the type of the l^{th} value. We again use a recursive algorithm, but this time we terminate when l hits zero:

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

template<int l, typename Head, typename... Tail>
struct tuple_element<l, tuple<Head, Tail...> > {
    typedef typename tuple_element<l-1, tuple<Tail...> >::type type;
};

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

A.1.3 Element access

The tuple `get<I>()` operation accesses the I^{th} value. Its recursive implementation is rather longer than we would like because we need to perform type computations along the way to be sure that we access the value as a **const** or non-**const** reference, dependent on its actual type and the **constness** of the tuple itself. The introduction of `decltype` and **auto** should simplify this code significantly [12].

```
template<int I, typename Tuple>
class get_impl;

template<int I, typename Head, typename... Values>
class get_impl<I, tuple<Head, Values...> > {
    typedef typename tuple_element<I-1, tuple<Values...> >::type Element;
    typedef typename add_reference<Element>::type RJ;
    typedef typename add_const_reference<Element>::type PJ;
    typedef get_impl<I-1, tuple<Values...> > Next;

public:
    static RJ get(tuple<Head, Values...> & t)
    { return Next::get(t.tail()); }

    static PJ get(const tuple<Head, Values...> & t)
    { return Next::get(t.tail()); }
};

template<typename Head, typename... Values>
class get_impl<0, tuple<Head, Values...> > {
    typedef typename add_reference<Head>::type RJ;
    typedef typename add_const_reference<Head>::type PJ;

public:
    static RJ get(tuple<Head, Values...> & t) { return t.head(); }
    static PJ get(const tuple<Head, Values...> & t) { return t.head(); }
};

template<int I, typename... Values>
typename add_reference<
    typename tuple_element<I, tuple<Values...> >::type
>::type
get(tuple<Values...> & t) {
    return get_impl<I, tuple<Values...> >::get(t);
}

template<int I, typename... Values>
typename add_const_reference<
    typename tuple_element<I, tuple<Values...> >::type
>::type
get(const tuple<Values...> & t) {
    return get_impl<I, tuple<Values...> >::get(t);
}
```

A.1.4 Relational operators

tuple provides a full set of relational operators that perform pairwise comparisons for the elements of two tuples. We illustrate the implementation of these relational operators without further comment, because all of them are simply recursive variadic templates.

```
inline bool operator==(const tuple<>&, const tuple<>&) { return true; }

template<typename T, typename... TTail, typename U, typename... UTail>
bool operator==(const tuple<T, TTail...> & t, const tuple<U, UTail...> & u) {
    return t.head() == u.head() && t.tail() == u.tail();
}
```

```

template<typename... TValues, typename... UValues>
bool operator!=(const tuple<TValues...>& t, const tuple<UValues...>& u) {
    return !(t == u);
}

inline bool operator<(const tuple<>&, const tuple<>&) { return false; }

template<typename T, typename... TTail, typename U, typename... UTail>
bool operator<(const tuple<T, TTail...>& t, const tuple<U, UTail...>& u) {
    return (t.head() < u.head() ||
            (!(t.head() < u.head()) && t.tail() < u.tail()));
}

template<typename... TValues, typename... UValues>
bool operator>(const tuple<TValues...>& t, const tuple<UValues...>& u) {
    return u < t;
}

template<typename... TValues, typename... UValues>
bool operator<=(const tuple<TValues...>& t, const tuple<UValues...>& u) {
    return !(u < t);
}

template<typename... TValues, typename... UValues>
bool operator>=(const tuple<TValues...>& t, const tuple<UValues...>& u) {
    return !(t < u);
}

```

A.2 Polymorphic Function Wrappers

The polymorphic function wrapper in TR1, called `function`, is essentially a generalized function pointer, which can target any compatible function object [2, §3.7]. `function` is typically used for callbacks, e.g.,

```

function<void(int x, int y, int button_mask)> on_click;

struct hello {
    void operator()(int x, int y, int button_mask) {
        std::cout << "Hello, world, from (x" << x << ", " << y << ")\n";
    }
};

on_click = hello();

on_click(17, 42, BUTTON_LEFT);

```

The class template `function` takes a single template argument, a function type, that states what function call signature it should have:

```

template<typename Signature>
class function;

```

The actual functionality of `function` is placed into a class template partial specialization, which picks up any function type:

```

template<typename R, typename... Args>
class function<R (Args...)> {
public:
    typedef R result_type;

    function() : invoker(0) { }

    function(const function& other) : invoker(0) {
        if (other.invoker)
            invoker = other.invoker->clone();
    }
}

```

```

template<typename F>
function(const F& f) : invoker(0) {
    invoker = new functor_invoker<F, R, Args...>(f);
}

~function() {
    if (invoker)
        delete invoker;
}

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

template<typename F>
function& operator=(const F& f) {
    function(f).swap(*this);
    return *this;
}

void swap(function& other) {
    invoker_base<R, Args...>* tmp = invoker;
    invoker = other.invoker;
    other.invoker = tmp;
}

result_type operator()(Args... args) const {
    assert(invoker);
    return invoker->invoke(args...);
}

private:
    invoker_base<R, Args...>* invoker;
};

```

We will come to the “invoker” member and its associated types momentarily. For now, however, note how we have used a template parameter pack `Args` to represent the function call arguments provided in the signature. `Args` is then reused to build the signature of `operator()`, so that function can pass the appropriate arguments along to its target.

The invoker member stores information about the target of function, so that we can call the target function (in `operator()`) or make a copy of it (copy constructor and assignment operator). The `invoker_base` class is an abstract base class that can perform these two operations:

```

template<typename R, typename... Args>
class invoker_base {
public:
    virtual ~invoker_base() { }
    virtual R invoke(Args...) = 0;
    virtual invoker_base* clone() = 0;
};

```

Each time we assign a new target for a function, we use a class derived from `invoker_base` that knows how to invoke a function object of that type. In this implementation of function, we use the following `functor_invoker` variadic template:

```

template<typename F, typename R, typename... Args>
class functor_invoker : public invoker_base<R, Args...> {
public:
    explicit functor_invoker(const F& f) : f(f) { }
    R invoke(Args... args) { return f(args...); }
    functor_invoker* clone() { return new functor_invoker(f); }
private:

```

```

    F f;
};

```

A.3 Function Object Binders

The `bind()` facility of TR1 provides a uniform mechanism for binding and reordering arguments of function objects [2, §3.6]. It is often used in conjunction with `function` to create callbacks to member functions, e.g.,

```

class Button {
public:
    function<void(int x, int y, int button_mask)> on_click;
};

class Dialog {
public:
    Dialog();
    void close(bool okay, int button_mask);

    Button okay;
    Button cancel;
};

Dialog::Dialog() {
    okay.on_click = bind(&Dialog::close, this, true, _1);
    cancel.on_click = bind(&Dialog::close, this, false, _1);
}

```

Readers unfamiliar with the use of `bind()`, from either Boost or TR1, are encouraged to read about its usage [3]. Even with a solid understanding of `bind()`'s usage, the implementation of `bind()`—with or without variadic templates—is nontrivial. We present a nearly-complete `bind()` implementation, because it is extremely important that a library such as `bind()` can be implemented without a large amount of code repetition (as is currently required). However, we note that there is inherent difficulty in implementing this kind of library, and this example is necessarily complex.

First, some preliminary type traits required for the `bind()` implementation:

```

template<typename T>
struct is_bind_expression {
    static const bool value = false;
};

template<typename T>
struct is_placeholder {
    static const int value = 0;
};

```

The core of `bind()`'s implementation is in the `bound_functor` class template. `bind()` is merely a generator function for instances of this class template:

```

template<typename F, typename... BoundArgs>
class bound_functor {
    typedef typename make_indexes<BoundArgs...>::type indexes;

public:
    typedef typename F::result_type result_type;

    explicit bound_functor(const F& f, const BoundArgs&... bound_args)
        : f(f), bound_args(bound_args...) { }

    template<typename... Args>
    typename F::result_type operator()(Args&... args);

private:
    F f;
    tuple<BoundArgs...> bound_args;
};

```

```
};
template<typename F, typename... BoundArgs>
inline bound_functor<F, BoundArgs...>
bind(const F& f, const BoundArgs&... bound_args) {
    return bound_functor<F, BoundArgs...>(f, bound_args...);
}
```

A bound function object stores all of the arguments bassed to `bind()` in a tuple called `bound_args`. These arguments will be consulted when the binder object is invoked. The function objects returned by `bind()` can be called with arguments of any type, so `bound_functor::operator()` is itself a variadic function template:

```
template<typename F, typename... BoundArgs>
template<typename... Args>
typename F::result_type bound_functor<F, BoundArgs...>::operator()(Args&... args) {
    return apply_functor(f, bound_args, indexes(), tie(args...));
}
```

The implementation of `operator()` redirects through a function called `apply_functor()`, which accepts the function object to call (`f`), the bound arguments, a set of `indexes`, and the arguments that were passed to `operator()`. These arguments to `operator()` are placed in a tuple using `tie()`.

The `indexes` type is actually a tuple of of indices `int_tuple<0, 1, 2, ..., N-1>`, where `N` is the number of bound arguments. To build this tuple of indices, we use the recursive algorithm implemented in the following `make_indexes` and `make_indexes_impl` templates:

```
template<int...> struct int_tuple {};
// make_indexes_impl is a helper for make_indexes
template<int I, typename IntTuple, typename... Types>
struct make_indexes_impl;

template<int I, int... Indexes, typename T, typename... Types>
struct make_indexes_impl<I, int_tuple<Indexes...>, T, Types...>
{
    typedef typename make_indexes_impl<I+1,
                                     int_tuple<Indexes..., I>,
                                     Types...>::type type;
};

template<int I, int... Indexes>
struct make_indexes_impl<I, int_tuple<Indexes...> > {
    typedef int_tuple<Indexes...> type;
};

template<typename... Types>
struct make_indexes : make_indexes_impl<0, int_tuple<>, Types...> { };
```

With all of the arguments to `apply_functor()` defined, we can delve into its implementation. It uses three separate template argument packs: `BoundArgs`, which contains the types of the arguments passed to `bind()`; `Indexes`, which contains the index values 0, 1, 2, ... `N-1`; and `Args`, which contains the types of the arguments passed to `bound_functor::operator()`:

```
template<typename F, typename... BoundArgs, int... Indexes, typename... Args>
typename F::result_type
apply_functor(F& f, tuple<BoundArgs...>& bound_args, int_tuple<Indexes...>,
              const tuple<Args&...>& args) {
    return f(mu(get<Indexes>(bound_args), args)...);
}
```

The actual implementation of `apply_functor()` performs a mapping `mu` that turns the bound arguments (passed to `bind()`) and the arguments passed to `operator()` into arguments that call be forwarded on to the function object `f`. We apply `mu` to each bound argument (in the tuple `bound_args`), passing along the tuple of arguments to `operator()` (`args`). To expand the arguments in the tuple, we apply unpacking operator ...

on the `mu` expression, unpacking `Indexes` to retrieve elements in the `bound_args` tuple. When there are three bound arguments and two arguments to `operator()`, `apply_functor()` essentially expands to:

```
template<typename F, typename BoundArg1, typename BoundArg2, typename BoundArg3,
        typename Arg1, typename Arg2>
typename F::result_type
apply_functor(F& f, tuple<BoundArg1, BoundArg2, BoundArg3>& bound_args, int_tuple<0, 1, 2>,
              const tuple<Arg1&, Arg2&>& args) {
    return f(mu(get<0>(bound_args), args),
             mu(get<1>(bound_args), args),
             mu(get<2>(bound_args), args)));
}
```

Note that we are replacing a quadratic number of overloaded function templates with a single variadic template, because `BoundArgs` and `Args` may have different lengths, and we need to support all combinations.

The mapping function `mu()` is implemented as a set of cunition templates, each corresponding to one of the following cases:

1. If the bound argument is the l^{th} placeholder (e.g., `_1`), `mu` returns the l^{th} argument that was passed to `operator()`.

```
template<int l, typename Tuple, typename = void>
struct safe_tuple_element { };

template<int l, typename... Values>
struct safe_tuple_element<l, tuple<Values...>,
                        typename enable_if<(l >= 0 &&
                                           l < tuple_size<tuple<Values...>>::value)
                                           >::type> {
    typedef typename tuple_element<l, tuple<Values...>>::type type;
};

template<typename Bound, typename... Args>
inline typename safe_tuple_element<is_placeholder<Bound>::value - 1,
                                tuple<Args...>>::type
mu(Bound& bound_arg, const tuple<Args&...>& args) {
    return get<is_placeholder<Bound>::value - 1>(args);
}
```

2. If the bound argument is a reference wrapper, return the stored reference:

```
template<typename T, typename... Args>
inline T& mu(reference_wrapper<T>& bound_arg, const tuple<Args&...>&) {
    return bound_arg.get();
}
```

3. If the bound argument is itself a `bind()` expression (meaning that we have a nested `bind()`), invoke that `bind` expression with the arguments passed to `operator()`:

```
template<typename F, int... Indexes, typename... Args>
inline typename F::result_type
unwrap_and_forward(F& f, int_tuple<Indexes...>, const tuple<Args&...>& args) {
    return f(get<Indexes>(args)...);
}

template<typename Bound, typename... Args>
inline typename enable_if<is_bind_expression<Bound>::value,
                        typename Bound::result_type>::type
mu(Bound& bound_arg, const tuple<Args&...>& args) {
    typedef typename make_indexes<Args...>::type Indexes;
    return unwrap_and_forward(bound_arg, Indexes(), args);
}
```

4. Otherwise, just return the bound argument:

```
template<typename Bound, typename... Args>
inline typename enable_if<(!is_bind_expression<Bound>::value
    && !is_placeholder<Bound>::value
    && !is_reference_wrapper<Bound>::value),
    Bound&>::type
mu(Bound& bound_arg, const tuple<Args&...>&) {
    return bound_arg;
}
```

B Grammar

Variadic templates do not introduce any new keywords or operators. The existing ellipsis operator (“...”), which is currently only used for C-style variadic functions, is overloaded to provide support for variadic templates.

B.1 Expressions

expression-list:
assignment-expression ..._{opt}
expression-list , *assignment-expression* ..._{opt}

An ellipsis can occur at the end of a parameter in an *expression-list*. This allows the use of variadic templates to call functions, construct objects, and initialize members and base classes.

B.2 Declarators

declarator-id:
..._{opt} *id-expression*
abstract-declarator:
...

An ellipsis can occur as an *abstract-declarator* (creating an unnamed parameter pack) or preceding the identifier in a *declarator-id* (creating a named parameter pack). Ellipses may only be used on declarators that are part of a *parameter-declaration*, and the parameter being declared must be the last parameter in the *parameter-declaration-list* (or any parameter in a *template-parameter-list*). Moreover, the type of a parameter containing an ellipsis shall contain at least one parameter pack, otherwise the code is ill-formed. Note that the use of the ellipsis as an abstract declarator introduces a syntactic ambiguity in code such as:

```
template<typename T> void foo(T...); // ambiguous?
```

Without variadic templates, `foo()` will be parsed as a (C-style) variadic function with a single template type parameter, `T`. With variadic templates, the ellipsis will be parsed as an *abstract-declarator*, which would make the parameter into a parameter pack. This second parse is ill-formed, because the type of the parameter (`T`) does not contain a parameter pack. We resolve the ambiguity semantically: if the abstract declarator is an ellipsis, the next token is a closing parenthesis, and the type of the parameter does not contain an unexpanded parameter pack, then the ellipsis is not parsed as part of the declarator. Instead, it is parsed as part of the *parameter-declaration-clause*, making the function a C-style variadic function. This semantic disambiguation is always performed when the template is initially declared, and retains the behavior of existing C++ code (because one can only get a parameter pack in the parameter list by declaring a variadic template).

In some cases one may want the disambiguation to make the opposite decision. For example, the following template is intended to create a tuple of function pointers, each of which takes a reference to one of the types

in `ArgTypes` and a (C-style) variable number of arguments. For instance, `tuple_fp<int, float>::type` would be `tuple<void(*)(&int&...), void(*)(&float&...)>`:

```
template<typename... ArgTypes>
struct tuple_fp {
    typedef tuple<void(*)(&ArgTypes&...)...> type; // error!
};
```

However, because the innermost abstract declarator is an ellipsis, and `ArgTypes&` contains a (template) parameter pack, `ArgTypes&` is expanded into separate function parameters. Then the second ellipsis, meant to expand the function pointer type into separate template arguments to `tuple`, causes an error: there are no parameter packs left to unpack for the outer ellipsis. To address this problem, we need to force the parser to consider the first (“inner”) ellipsis as an indicator of a C-style variable-length parameter list using a comma:⁴

```
template<typename... ArgTypes>
struct tuple_fp {
    typedef tuple<void(*)(&ArgTypes&, ...)...> type; // okay
};
```

B.3 Templates

```
type-parameter:
    class ...opt identifieropt
    typename ...opt identifieropt
    template < template-parameter-list > class ...opt identifieropt

template-argument-list:
    template-argument ...opt
    template-argument-list , template-argument ...opt
```

A template parameter pack can be declared by placing an ellipsis just prior to the (optional) identifier naming a template type parameter. Non-type template parameter packs can be declared in the same way as (function) parameter packs; see Section B.2. Template parameter packs must come after normal template parameters. Class template partial specializations and function templates may have multiple template parameter packs.

An ellipsis can occur after any template argument in a template argument list, indicating an expansion expression. When the template argument list is part of the *template-id* of a partial specialization, the ellipsis for the template pack/unpack expression can only occur at the end of the template argument list.

C Revision history

- **Since N1704=04-0144:**

- We selected the set of “core” features we need from the “menu” provided in N1704, focusing on the smallest set of features needed to express important ideas and libraries.
- We implemented all of the proposed features in the GNU C++ compiler, and verified that the implementations provided in this proposal work properly.

- **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.

⁴This syntax is not new; it is already part of C and C++. The comma preceeding an ellipsis is optional.

- 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.

References

- [1] A. Alexandrescu. Loki. <http://loki-lib.sourceforge.net/>, June 2006.
- [2] M. Austern, editor. *C++ Library Extensions*. Number TR19768. ISO/IEC, 2006.
- [3] P. Dimov. The Boost Bind library. <http://www.boost.org/libs/bind/bind.html>, August 2001.
- [4] 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.
- [5] D. Gregor. A brief introduction to variadic templates. Number N2087=06-0157 in ANSI/ISO C++ Standard Committee Pre-Portland mailing, 2006.
- [6] D. Gregor and B. Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.
- [7] D. Gregor, J. Willcock, and A. Lumsdaine. Concepts for the C++0x Standard Library: Utilities. Technical Report N2038=06-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.
- [8] A. Gurtovoy. The Boost MPL library. <http://www.boost.org/libs/boost/doc/index.html>, July 2002.
- [9] H. E. Hinnant, D. Abrahams, J. S. Adamczyk, P. Dimov, and A. Hommel. A proposal to add an rvalue reference to the C++ language. Number N1770=05-0030 in ANSI/ISO C++ Standard Committee Pre-Lillehammer mailing, March 2005.
- [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 B. Stroustrup. Decltype and auto (revision 3). Number N1607=04-0047 in ANSI/ISO C++ Standard Committee Pre-Sydney mailing, 2004.
- [13] V. Karvonen and P. Mensonides. The Boost Preprocessor library. <http://www.boost.org/libs/preprocessor/doc/index.html>, July 2001.
- [14] G. D. Reis and B. Stroustrup. Generalized initializer lists. Number N1509=03-0092 in ANSI/ISO C++ Standard Committee Pre-Kona mailing, September 2003.
- [15] B. Stroustrup and G. D. Reis. Initialization and initializers. Number N1890=05-0150 in ANSI/ISO C++ Standard Committee Post-Tremblant mailing, October 2005.