JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 2, Special Issue OOPS Track at SAC 2007, February 2008

Variadic Templates for C++0x

Douglas Gregor, Department of Computer Science, Indiana University, USA
Jaakko Järvi, Department of Computer Science, Texas A&M University, USA

Generic functions and classes typically accept a fixed number of type arguments. However, generic functions and classes that accept a variable number of type arguments have proven to be a very useful, even though there is no support for this feature in C++. Numerous foundational libraries rely on clever template and preprocessor tricks to emulate such variable-length templates. By several measures these emulations are inadequate. This paper describes *variadic templates*, an extension to the C++ language that significantly improves existing implementations of widely used C++ libraries in terms of code size, quality of error diagnostics, compilation speed, and generality. Furthermore, variadic templates enable new applications, such as type-safe implementations of functions like `printf`, and improved support for generic mixin classes. Variadic templates are part of the upcoming ISO C++ Standard, dubbed C++0x, and we have integrated variadic templates into the GNU C++ compiler.

1 INTRODUCTION

Many situations call for generic functions that accept an arbitrary number of parameters or generic classes that can be instantiated with any number of type arguments. An example of the former kind is a type-safe, secure version of the `printf` function in C. A parametrized class representing *tuple* objects is an example of the latter kind. Both of these above scenarios can be supported with *variadic templates*, an extension to C++ for types parametrized with a varying number of arguments.

The need for variadic templates in C++ is pronounced. Though not part of C++, variadic templates are crucial in implementations of many widely-used libraries. This is possible because of clever tricks with templates (such as the use of many template parameters with default arguments), often combined with more tricks with the preprocessor (to effect code repetition for templates with varying numbers of parameters), that enable a clumsy emulation of variadic templates in today's C++. Libraries relying on such emulation include the *bind* libraries [1, 2] that provide a form of partial function application for C++, *tuple* libraries [3, 4], libraries generalizing C++ function pointers [5], and the *Boost Metaprogramming Library* (MPL) that enables compile-time computation with types [6]. Apart from the last one, the functionality of these libraries is included in the draft specification of the next revision of the C++ standard library (see [7]). The above libraries rely on tricks that are (1) limited in expressiveness requiring, for example, a predefined upper limit for the length of the template parameter list; (2) expensive in terms of time and memory needed for compilation; (3) subtle and expert-friendly, difficult to master and use;

and (4) a nightmare to debug, exhibiting enormous compiler error messages from simple programming errors.

The built-in variadic templates proposed here can be instantiated with an unbounded number of template arguments. Both class and function templates can be variadic—the latter allowing the definition of functions that can accept any number of arguments in a type-safe manner. Variadic templates make obsolete the ugly hacks used in their emulation today. The libraries mentioned above can be implemented using variadic templates in a significantly more concise manner with no artificial limits on the number of arguments, placing less burden on the compiler and producing shorter, clearer error diagnostics. Variadic templates also enable new uses, such as a type-safe, secure implementation of `printf`. We show the `printf` implementation in Section 3; here we settle for an example of its use:

```
const char* msg = "%s can accept %i parameters (or %s).";
printf(msg, std::string("Variadic templates"), 100, "more");
```

The above code compiles and executes correctly with our extensions. In current C++ the `printf` call invokes undefined behavior because `printf` cannot handle `std::string`s or other user-defined types. As another example, variadic templates prove very useful in defining *mixin* classes, discussed in Section 3.

The compilation model of C++ templates, where a distinct piece of code is generated for each different instantiation of a template, matches well with variadic templates. Variadic templates essentially become program generators. Later sections show how manipulating the variadic template argument lists may require generating many functions, but C++'s function inlining combined with the instantiation model can typically coalesce such calls into a single, efficient block of code.

Variadic templates are part of the draft ISO C++ standard [7]. We have implemented variadic templates in the GNU C++ compiler [8]. This language feature is used within the GNU implementation of the C++ Standard Library and will be available to users in version 4.3 of the GNU compiler.

2 EMULATING VARIADICS

This section briefly explains the tricks used to emulate variadic templates, and why these emulations are inferior to a built-in language feature.

C++ class templates with variable-length argument lists can be emulated using a long list of "extra" template parameters that all have a special default value, e.g.,

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

This `tuple` template can be instantiated with anywhere from zero to `N` template arguments. Assuming that `N` is large enough, we could instantiate `tuple`, say, as:

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

Of course, this instantiation actually contains $N - 5$ `unused` parameters, but it is possible to "ignore" those extra parameters with tricks in the implementation of the `tuple` template, so that objects of the type `unused` are not stored in `tuple` objects.

Default template arguments are not allowed in function templates, so emulating variadic function templates requires a host of overloads. For example, the draft standard tuple library [7, §20.3] overloads the `make_tuple` helper functions for each different number of template arguments. Figure 1 shows three of these overloads.

```
tuple<> make_tuple() { return tuple<>(); };

template<typename T1>
tuple<T1> make_tuple(const T1& t)
{ return tuple<T1>(t1); };

template<typename T1, typename T2>
tuple<T1, T2> make_tuple(const T1& t1, const T2& t2)
{ return tuple<T1, T2>(t1, t2); };
```

Figure 1: Emulating variadic function templates with overloads.

As mentioned in Section 1, the above emulation is used by many widely-used foundational C++ libraries. Unfortunately, the emulation 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 sets an upper limit on the number of variadic arguments, which is not extensible without resorting to preprocessor metaprogramming [9] (which has its own fixed upper limits).

As an example of the severity of the problems with the emulations, consider the `tuple` and function object facilities in the current draft standard library specification [7, §20], implemented by many compiler vendors. In the GCC standard library implementation, excessive compilation times forced reverting the maximum number of supported parameters from the original twenty to ten—the smallest value allowed by the draft standard [10]. Still, the code is very slow to preprocess and parse, and is almost impenetrable to readers. Section 4 demonstrates how variadic templates drastically reduce compilation times for these libraries.

3 VARIADIC TEMPLATES

Variadic templates are a syntactically lightweight extension to the C++ template system, using only the existing ellipsis operator "`...`" and mixing well with existing template code. A template parameter declared with an ellipsis prior to its name signifies that zero or more template arguments can "fit" into that single parameter—such a parameter is referred to as a *template parameter pack*. For example, using a template parameter pack the `tuple` template shown in Section 2 becomes:

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

Instantiating `tuple` as, say, `tuple<char, int, string>` binds the template parameter pack `Elements` to a list of template arguments whose elements are the types `char`, `int`, and `string`.

The only operation one can do to a parameter pack is to *expand* it. To demonstrate, we add a static constant in the tuple class to tell the length of the tuple:

```
template<typename... Elements> class tuple {
  static const int length = count<Elements...>::value;
};
```

Here, the `Elements` parameter pack is expanded into normal template arguments, and the `count` template is instantiated as if the contents of `Elements` were written out explicitly. For example, `tuple<char, int, string>::length` is defined as `count<char, int, string>::value`.

Access to individual elements of a parameter pack is achieved via template specialization. The `count` template has a primary definition (which is never used), and two specializations:

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

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

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

This template computes, at compile-time, the number of arguments it was instantiated with. The idea is to "peel off" one argument at a time from a parameter pack, computing the length of the remaining parameter pack recursively, and adding one to that length. Consider the expression `count<A, B, C>::value`. Both the primary template and the first specialization match. In the primary template, `Args` would cover all arguments `A`, `B`, and `C`, whereas in the first specialization, `Args` only covers `B` and `C` as `A` is bound to the individual template parameter `T`. The first specialization is a better match. It computes the constant `value` by expanding `Args` to a new instantiation of `count`. The generated expression becomes `count<B, C>::value`, which again instantiates the first specialization, peeling off `B`, and generating the expression `count<C>::value`, and finally `count<>::value`, in which case the second specialization matches and ends the recursive chain of instantiations.

For this to work, we extend the C++'s template specialization rules that decide when one specialization is a better match than another as follows: *A non-variadic specialization is a better match than a variadic specialization. Of two variadic specializations, the one where fewer arguments are "covered" by a parameter pack is a better match.* Hence, in the `count` definition, the first specialization matches for instantiations with one or more arguments, and is a better match than the primary template in those cases. The second specialization only matches for instantiations with zero arguments, and is the best match in those cases.

Variadic function templates can be called with an arbitrary number of arguments. A variadic function template is defined by using a parameter pack in the type of the last parameter of a function template. Similarly to the syntax of template parameter packs, a trailing ellipsis is required. The definition of the `make_tuple` function serves as an example:

```
template <typename... Elements>
tuple<Elements...> make_tuple(const Elements&... elems) {
  return tuple<Elements...>(elems...);
}
```

The types of the arguments to this function are contained in the parameter pack `Elements`, and the values of the arguments in the *function parameter pack* `elems`. Note that what precedes the ellipsis in the function parameter list is not a plain parameter pack `Elements`, but rather a *pattern*, `const Elements&`, that will be repeated for each type in the parameter pack.

Similarly to template parameter packs, the only valid use for a function parameter pack is to expand it, now in an argument list of a function call. Again, we require that when expanding a function parameter pack it (or the pattern it is contained in) be followed by an ellipsis. The body of the `make_tuple` function shows an example: the `elems...` expression generates a list of actual arguments to the constructor of the `tuple<Elements...>` class. Calling `make_tuple` with, say, two arguments generates a function identical to the last `make_tuple` overload in Figure 1.

When expanding a template parameter pack or a function parameter pack, we can also use patterns. For example, the `tuple` library provides `tie` functions that can be used to assign the elements of a tuple into separate variables:

```
int a; string b;
tie(a, b) = make_tuple(100, string("42"));
// a has value 100, b has value "42"
```

The way this works is by having `tie` to construct a tuple that stores references to `tie`'s arguments, here `a` and `b`. Assignment between tuples is defined as element-wise assignment, so the desired result follows. Now, however, the element types of the tuple returned by `tie` are reference types. We can express this with the pattern `Elements&...`:

```
template <typename... Elements>
tuple<Elements&...> tie(Elements&... elems) {
  return tuple<Elements&...>(elems...);
}
```

We saw above that the access to the elements of template parameter packs is via recursive template instantiations where the recurring cases, as well as stopping conditions, are defined as template specializations. In a similar manner, we access values in function parameter packs with function template overloads. We thus need to extend the overload resolution rules analogously to the rules of ordering template specializations: a matching non-variadic function is a better match than a variadic function, and between two variadic functions, the one where fewer arguments are covered by the parameter pack is a better match.

We illustrate the use of function overloading with variadic templates by defining a simple `print` operation that accepts any number of function arguments and displays them using the C++ stream operations. As with the `count` template, we peel off one argument at a time and recursively process the remaining arguments; in this case, however, we print the argument ("`current`") at each stage.

```
void print() { }

template<typename T, typename... Rest>
void print(const T& current, const Rest&... rest) {
  std::cout << current;
  print(rest...);
}
```

The essential elements of variadic templates are: declaring template and function parameter packs with the ellipsis operator; expanding them, again with the ellipsis operator; using partial specialization to access elements of template parameter packs; and using function overloading to access elements of function parameter packs.

Tuple types

The current draft of the C++ standard library [7] specifies a `tuple` template, essentially a generalization of the `std::pair` to an arbitrary number of parameters. Implementing tuples in today's C++ relies on the hacks described in Section 2. With variadic templates, the implementation becomes fairly effortless.

The notable parts of the implementation, consisting of a primary template and two specializations, are shown in Figure 2. The primary `tuple` template is never instantiated; either the specialization for zero arguments (line 3) or for one argument and a parameter pack (line 5) is always a better match. The latter of these is the interesting case—it is the recursive definition that peels off the first argument (`Head`) to be stored in the `m_head` member and derives from a tuple containing the remaining arguments (`Tail`).

The constructor in line 15, with a function parameter pack as its last parameter, accepts one argument for each element in the tuple. It initializes the `m_head` data member and passes all the arguments of the function parameter pack `vtail` to the constructor of the base class, which is another tuple, containing all but the first element. Similar structure is visible in the templated constructor in line 19, that allows one tuple to be converted to another one, assuming their corresponding element types are convertible from one to another. The assignment operator (line 23) follows the same pattern.

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

template<> class tuple<> { };

template<typename Head, typename... Tail>
class tuple<Head, Tail...> : private tuple<Tail...>
{
public:

  tuple() { }

  // implicit copy-constructor is okay

  // Construct tuple from separate arguments.
  tuple(const Head& v, const Tail&... vtail)
    : m_head(v), tuple<Tail...>(vtail...) { }

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

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

  Head& head() { return m_head; }
  const Head& head() const { return m_head; }
  tuple<Tail...>& tail() { return *this; }
  const tuple<Tail...>& tail() const { return *this; }

 protected:
  Head m_head;
};
```

Figure 2: A tuple template.

The tuple template also provides four helper functions, starting from line 30, for accessing the head and the tail of a tuple. Note that only the head is stored as a data member. The elements in the tail of a tuple will each be stored as a head of some tuple. This may seem horribly inefficient, but C++ compilers routinely inline code that traverses such nested instantiations, making element access to tuple as fast as referring to any member variable of a class.

Of other functionality the draft C++ standard library requires of tuples, Section 3 showed the `make_tuple` and `tie` functions; we omit functions for accessing an element in a tuple based on an index, as the definitions of those are a bit tedious; and as a representative of all comparison operators, we show how the equality operator for tuples is defined:

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

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

Type-safe, secure printf

Many security problems stem from so-called "format string vulnerabilities", which exploit the inability of C's `printf` implementation to check its format string parameter against the actual types of the arguments. Here, we outline a type-safe, secure `printf` implemented with variadic templates.

The definition is shown in Figure 3. This `printf` function accepts, e.g., the call shown in Section 1. The first `printf` overload, accepting no arguments besides the format string, is the base case of the recursive definition. The second `printf` overload accepts two or more arguments, the formatting string and one or more values. In this overload, the function parameter pack `args` covers all but the first value. The implementation scans the format string, emitting characters until it reaches a format specifier. The format specifier is then processed and checked (by `check_specifier`, shown in Figure 4), which may result in an exception if the format specifier does not match the argument type. Once the format specifier is known to be correct, `printf` outputs the first value, then recursively calls `printf` again with what remains of the format string and with the rest of the values obtained by expanding `args`. Note that the recursion occurs at compile-time, each nested call will be to a different instance of the `printf` template, with one fewer argument on every round. Note also that we have extended `printf` with the 'Z' specifier, permitting the display of user-defined types.

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

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

Figure 3: A simple type-safe, secure `printf` function.

Mixins and inheriting constructors

Deriving a new class from an existing class is a convenient mechanism to adapt an existing type to manifest slightly different behaviour. For classes with many constructors, however, it may be a notable programming task—C++ classes do not inherit the constructors from their base classes. Often, however, inheriting constructors would be just what the programmer desires: the new derived class may be a small adaptation of the base class, where the ways of constructing objects remain unchanged. Alternatively, an adaptation of the construction scheme may be desired, e.g., adding a parameter or two to each constructor.

Not being able to inherit or adapt constructors from base classes becomes a more serious problem with *mixin composition* in C++. One way of implementing mixins is by inheriting from a template parameter (see, e.g., [11]). In such a design, the number and argument types of the base class constructors are not known. Consider for example the following class that can add a *location* information to any type:

```
template <typename Base>
class Location : public Base {
  int x, y;
public:
  Location() : x(0), y(0), Base() { }
};
```

```
template<typename T>
bool check_specifier(const char*& p) {
  // process padding, precision information...
  switch (*p) {
    case 'd': case 'i': return is_same<T, int>::value;
    case 'e': case 'E': return is_same<T, double>::value;
    case 's': return is_same<T, char*>::value;
    case 'Z': return true; // user-defined
    // ...
  }
  return false;
}
```

Figure 4: Subroutine of `printf` that checks the current formatting specifier against the current argument type, `T`. The (draft) standard `is_same` template determines whether two type expressions denote the same type [7, §20.4.5].

We could use it for example to give physical co-ordinates to a class representing graph vertices:

```
typedef Location<Vertex> PositionedVertex;
```

The new class, however, only supports default construction. Whatever other construction schemes the `Vertex` class may support are not accessible anymore. Variadic templates offer a simple solution for "inheriting" constructors. A single constructor definition suffices:

```
template <typename... Args>
Location(const Args&... args) : x(0), y(0), Base(args...) { }
```

Note that this solution has a minor deficiency: not all argument types can be forwarded cleanly through `const` references: non-constant temporaries of primitive type, for example, are rejected, and non-constant lvalues will be forwarded to the `Base` constructor as constant lvalues. In fact, today's C++ offers no generic way to define a parameter type that could accept an object of arbitrary type and forward it to another function. This problem is addressed with *rvalue references* [12], another new feature in the upcoming C++ standard [7]. With rvalue references and variadic templates, one can forward an arbitrary number of arguments to the base class as follows:

```
template <typename... Args>
Location(Args&&... args)
  : x(0), y(0), Base(std::forward<Args>(args)...) { }
```

In the parameter list, `&&` denotes an rvalue reference parameter, which can bind to either rvalues or lvalues. More interesting from the variadic templates perspective is the use of `std::forward` to forward arguments to the `Base` constructor. The pack expansion uses two parameter packs—`Args` and `args`—which will be expanded pairwise to produce an argument list `std::forward<Args1>(args1)`, `std::forward<Args2>(args2)`, . . ., `std::forward<Args`N`>(args`N`)`.

The constructor forwarding arguments to base class constructors is a specific case of a more general situation: a function forwarding its arguments to another function, possibly manipulating them in some ways, or adding other functionality. Using variadic templates with rvalue references, any number of arguments can be forwarded without introducing additional temporaries or losing the lvalue- or rvalueness of the original argument. The lack of this feature in today's C++ has long been the source of ugly hacks in several libraries [1, 2, 13].

Our `Location` class template permits extension by wrapping the class it is extending, e.g., `Vertex`. An alternative approach would be to prepare the `Vertex` class to accept a user-defined mixin (such as `Location`) from which it would derive. Given that C++ permits multiple inheritance, the `Vertex` class could even accept multiple mixins, e.g.,

```
typedef Vertex<Location, Population, Mayor> City;
```

We can prepare `Vertex` to accept many mixins by providing it with a `Mixins` template type parameter pack, which will contain all of the class types from which `Vertex` will derive. We can then expand `Mixins` into separate, public base classes:

```
template <typename... Mixins>
class Vertex : public Mixins... {
public:
  template<typename... Args>
  Vertex(Args&&... args)
    : Mixins(std::forward<Args>(args))... { }
};
```

In our extended `Vertex` class, we use pack expansions for the base classes and base class initializers of our mixins, the latter of which actually uses three parameter packs in a single pack expansion (all of which must have the same length). Thus, the i^{th} argument to the constructor is used to initialize the i^{th} base class. In general, parameter packs can be expanded anywhere the C++ grammar contains a comma-separated list. The `City` type defined above could be constructed as, e.g.,

```
City bloomington(Location(39.2,86.4), Population(69017),
                 Mayor("Mark Kruzan"));
```

Template Metaprogramming

The `tuple` type developed in Section 3 acts both as a container for (run-time) values and as a container for types. The latter usage makes a tuple useful in many of the same contexts as a *typelist*, used as the primary container for types within the world of template metaprogramming [14, 15]. A type list is typically constructed using a LISP-like `cons` template, e.g.,

```
struct null { };

template<typename Head, typename Tail = null> struct cons { };

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

LISP-like cons typelists are relatively easy to extend and manipulate with template metaprograms. For example, one can easily write algorithms to insert an element at the beginning of a list (producing a new list), map each element in a list via a given metafunction, or zip two sequences into a sequence of pairs. The variadic tuple can work in much the same way, with most of these algorithms requiring only a trivial amount of code, far less than for typelists. Moreover, the syntax of tuple is more concise (compare the cons-based definition of integral_types to the tuple-based formulation in Section 2) and, because variadic templates are directly supported by the language, more efficient to compile.

Inserting a new element at the beginning of a tuple can be achieved with a simple metafunction push_front:

```
template<typename Sequence, typename T> struct push_front;

template<typename... Elements, typename T>
struct push_front<tuple<Elements...>, T> {
  typedef tuple<T, Elements...> type;
};
```

The map operation on typelists (referred to as transform in [14]) requires a recursive template metaprogram to walk the list, map each value, and build the result. With variadic templates, this operation can be accomplished with a single non-recursive template specialization, shown below. The complex-looking instantiation of the apply template is an invocation of a template metafunction according to the conventions of the MPL library [14], which uses nested templates inside type parameters to express template metafunctions rather than the more obvious (but more limited) template template parameters.

```
template<typename Sequence, typename F> struct map;

template<typename... Elements, typename F>
struct map<tuple<Elements...>, F> {
  typedef tuple<typename F::template apply<Elements>::type...> type;
};
```

Finally, the zip operation takes two sequences of equal length and produces a sequence of pairs. Again, with a LISP-like typelist, one would write a recursive algorithm. With variadic templates, pairwise expansion allows for a concise definition:

```
template<typename Sequence1, typename Sequence2> struct zip;

template<typename... Elements1, typename... Elements2>
struct zip<tuple<Elements1...>, tuple<Elements2...> > {
  typedef tuple<std::pair<Elements1, Elements2>...> type;
};
```

We have shown how one can manipulate `tuple`s and typelists, efficiently and concisely, using variadic templates. Section 3 illustrated how one can create tuples from an arbitrary number of arguments via `tie` and `make_tuple`. What remains is to take an existing tuple object (containing run-time values of various types) and break it apart into separate arguments, one for each run-time value, allowing tuples to serve as the medium for storing arbitrary sets of statically-typed data. In particular, we need to define a function `apply(f, t)` that extracts the values `t0`, `t1`, . . ., `t`N from the tuple `t` and passes them to the function `f` as `f(t1, t2, ..., t`N`)`.

The `tuple` type defined in C++0x [7] provides a `get<`I`>(t)` operation that accesses the I^{th} run-time value in the tuple `t`. Thus, one could extract all of the run-time values from the tuple by producing arguments `get<0>(t)`, `get<1>(t)`, . . ., `get<`N`-1>(t)`. Thus, the problem reduces to the need to produce a parameter pack containing the integral values 0, 1, . . ., $N - 1$, which can be achieved via a *template non-type parameter pack*:

```
template<int... Values> struct int_tuple { };
```

Given a length N, it is possible to write a simple template metafunction (call it `indices`) that constructs a set of indices as `int_tuple<0, 1, ..., `N`-1>`. Then, this index tuple can be used to index into the actual tuple via `get`, providing access to all of the run-time values as arguments:

```
template<typename F, typename... Elements, int... Indices>
inline void
apply(const F& f, const tuple<Elements...>& t, int_tuple<Indices...>) {
  f(get<Indices>(t)...);
}
```

The template non-type parameter pack `Indices` dictates which elements are picked out the tuple, and in what order. When `Indices` corresponds to 0, 1, . . ., N-1, the tuple's run-time values will be passed to `f` in their natural order. However, one can permute the indices in any way, eliminate or repeat indices, etc., to provide a different set of arguments to `f`. In the common case, however, we wish to pass all of the values in order, which is accomplished by the following definition of `f`.

```
template<int N> struct indices;

template<typename F, typename... Elements>
inline void apply(const F& f, const tuple<Elements...>& t) {
  apply(f, t, typename indices<sizeof...(Elements)>::type());
}
```

We assume that `indices` is a template metafunction returning an `int_tuple` of indices of length N. The `sizeof...` operator is a compiler-supported shorthand that computes the length of a parameter pack. While not strictly necessary (we illustrated the equivalent `count` metafunction in Section 3), this operator reduces the burden of this common operation on both users and compilers.

Other applications

We mentioned a few applications of variadic templates in Section 1. These include the polymorphic function wrappers and the partial function application with `bind` functions, both in the draft standard library, currently implemented with the hacks described in Section 2. We have implemented these facilities in GCC 4.3's C++ standard library, where the use of variadic templates resulted in a code-size reduction of more than 50 kilobytes and led to a much more readable and maintainable formulation.

As can be seen from the kinds of uses we describe for variadic templates, the feature is a powerful tool for library writing, enabling applications that essentially extend the C++ language with new features. In day-to-day programming, variadic template uses would be frequent in using library components implemented in terms of variadic templates. Direct definitions of variadic templates would surface mostly in implementations of mixins and forwarding functions.

4 IMPLEMENTATION AND EVALUATION

Parameter packs are not first-class citizens in the C++'s type system: template parameter packs are not types and function parameter packs are not values. A single object representing a parameter pack does not exist in memory when a program is executing. Every time a parameter pack is referred to, it is at compile time expanded to individual arguments. This enables a lightweight specification and implementation on top of the existing template system of C++, integrating well with existing compiler technology.

All C++ compilers implement templates with a so called *instantiation model*, where template instantiation with a different set of arguments leads to generating a different piece of code. Variadic templates follow the same approach, and thus are essentially program generators. It is easy to see that this generation is programmable, and variadic templates are expressive enough so that arbitrary computations can be encoded with them. As a consequence, it is possible to write variadic templates for which instantiation with certain arguments leads to a non-terminating sequence of other instantiations. For example:

```
template <typename... Args>
void add_one(Args... args) { add_one(1, args...); }
```

The problem of determining whether a C++ template instantiation is terminating has been shown to be undecidable [16]. C++ deals with non-terminating (or terminating but very long) chains of instantiations by simply setting an upper limit on the length of an instantiation chain. A standard compliant compiler must allow at least 17 nested instantiations, but most compilers let the programmer increase this limit with a command line option.

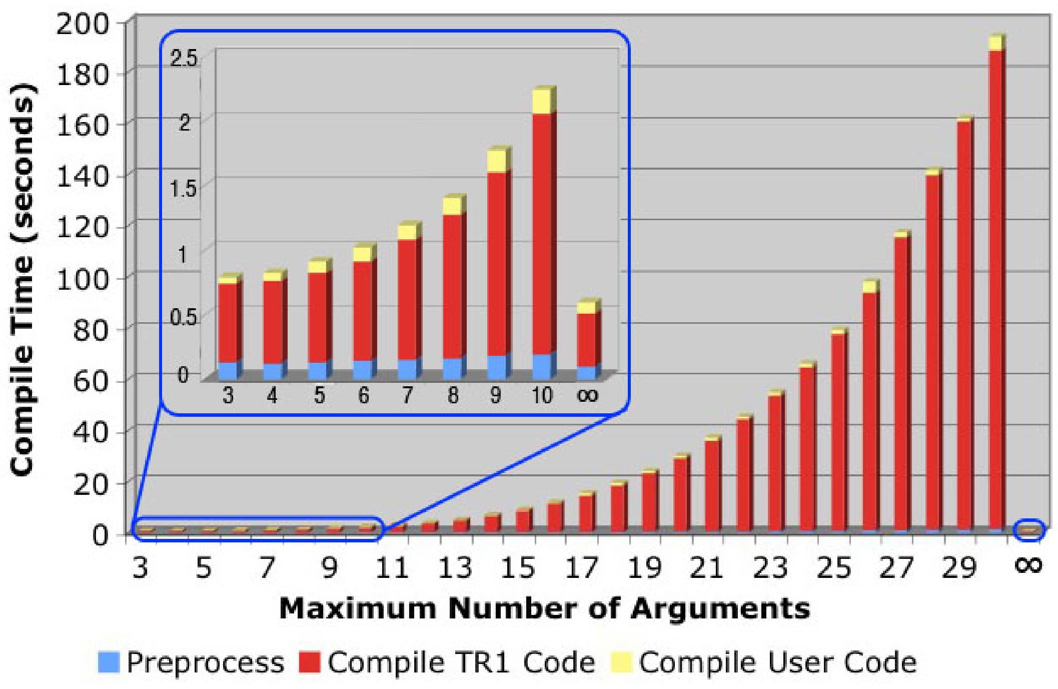

Figure 5: Compile-time performance for a small program with variadic templates (∞) and with emulation of variadic templates supporting up to a fixed number of arguments.

We have implemented the complete specification of variadic templates in the GNU C++ compiler [8], which are available in GCC 4.3. The implementation itself was relatively straightforward in GCC, implying that this feature can be implemented in other C++ compilers without architectural changes. Our basic implementation approach involved adding flags to each kind of template parameter and each function parameter stating whether these parameters are in fact parameter packs. Parameter packs are only treated differently from their non-pack equivalents at several key places in the compiler:

- When deducing template arguments from a function call or when matching a partial specialization of a class template, parameter packs can bind to many arguments rather than a single argument.
- When instantiating an expansion expression (e.g., `args...`), the instantiation is performed once for each argument in the parameter pack, and the results of each instantiation are placed in separate arguments.
- Uses of the ellipsis operator to expand expressions or types, which can be used in template argument deduction and instantiation. The compiler must also verify that parameter packs are not used outside of an ellipsis operator.

To determine the impact of variadic templates on compilation time, we updated the GNU implementation of the C++ Library Extensions Technical Report (TR1) [17] to compare variadic templates against their emulation. We opted to evaluate TR1 because it contains many template-heavy constructs whose implementation currently requires a great deal of repetition and can benefit greatly from variadic templates. Moreover, the components in TR1 have also been introduced into the C++0x working draft [7], where variadic templates are used within the specification itself, eliminating the pseudo-code description used in TR1.

We modified the GNU implementation of TR1 to completely emulate variadic templates via preprocessor metaprogramming [9], allowing us to vary the maximum number of arguments with a macro definition, and (alternatively) to use variadic templates as implemented in our compiler. We then compiled a simple, fixed user program that included the TR1 library headers and used their facilities in some simple ways. Figure 5 illustrates the compilation time of this simple program as we vary the maximum number of arguments supported in the library, divided into preprocessing time, TR1 header compilation time, and user code compilation time. We vary the maximum number of arguments from 3 to 30; the compilation time increases quickly, with the vast majority of time spent compiling the TR1 library headers themselves. This effort is entirely wasted for our example user program (and most user programs), which only uses up to 3 arguments. Beyond 15 arguments the compilation time becomes prohibitive for real-world use. The inset portion of Figure 5 illustrates compile-time performance for a subset of the data, and we can see that variadic templates require less compilation time than even the three-argument version of the emulated variadics while supporting an unbounded number of arguments.

5 RELATED WORK

Different forms of type-safe variadic functions are supported by different languages, and using notably different language constructs and implementations. If each parameter type in the variable argument list must be the same, we say that the variable argument list is *homogeneous*, otherwise we say it is *heterogeneous*. We use the same terminology for functions that use such argument lists.

Mainstream object-oriented languages, such as Java and C#, support type-safe homogeneous variadic functions. The *variable arity methods* [18] of Java are declared with an ellipsis that follows the last parameter type of a method. This construct is defined as syntactic sugar for declaring an array parameter; within the body of a variable arity method, the “vararg” parameter is treated as an array. Another dose of sugar turns a method argument list into an array at a call site of a variable arity method. C#’s *params* [19] are similar to Java’s variable arity methods. Heterogeneous variadic argument lists can be emulated with the parameter list `Object...` that accepts objects of any types, but obviously the exact types of the arguments are not known statically.

The idea of extending Java generics with heterogeneous variadic type parameter lists has been brought up [20], with related syntax and expansion behavior to that of ours. Java generics does not have a counterpart to that of C++'s template specialization—the expansion functionality would thus not enable recursive variadic definitions such as the tuple template presented in Section 3. Heterogeneous variadic forwarding functions would become possible, though.

The programming language D [21] provides two related features to our work: *variadic functions with type info* and *type-safe variadic functions*. The former is comparable to the `Object...` parameter list in Java: variadic arguments can be accessed through `void` pointers, but also their *typeid* is available. Thus, the programmer can safely cast the arguments to their expected types and deal with errors in a controlled manner. The latter feature is for initializing an array with a fixed element type, as in Java's variable arity methods. Neither of these features offer support for heterogeneous variadic argument lists where one would not need to resort to dynamic type information (*typeid*) to achieve type-safety. Note that D uses the ellipsis notation to request an automatic generation of a parameter list that consists of all the members of a particular class, and D's language reference classifies the feature under type-safe variadic functions. Regardless of the notations used and this classification, such functions are not variadic.

Functional languages can support a form of variadic polymorphic functions with combinators, with solutions of varying level of sophistication—`printf` has been the showcase example. In essence, a variable length argument list is emulated with a composition of calls to polymorphic unary functions. For example, a type-safe `printf`-like functionality can be provided in ML with such combinators comprising the formatting string [22]. E.g., our `printf` example from Section 3 becomes the following:

```
format (str oo lit " can accept " oo int oo
       lit "parameters (or " oo str oo lit ").")
  : string → int → string → string
```

The `format` function thus seemingly accepts a variadic number of parameters, the types of which determine the number and types of the parameters that the resulting function expects. In this solution, the formatting specifiers must be known statically.

Template Haskell [23] can evaluate and generate code at compile-time. Thus, in an analogous approach as the one above, the format specification can be represented by a string. This string is parsed at compile time, and the function with a type matching the format specification is generated. More general and systematic way of emulating polymorphic variadic functions is offered by the type-indexed functions of *data-type generic programming*—a generalization of the `printf` implementation of [22] is presented in [24].

Some C++ libraries use variations of the combinator functions to emulate type-safe variadic polymorphic functions. The underlying technique is known as *expression templates*. Staying loyal to the `printf` function, we demonstrate with the Boost.Format C++ library [25]. It provides analogous functionality with its `format` function that accepts a format specifier string and returns a type that overloads the modulus operator to accept additional parameters. For example:

```
format ("%s can accept %i parameters (or %s).")
  % std::string("Variadic templates") % 100 % "more";
```

The expansion mechanism of variadic templates borrows from Scheme's [26] *macros*. For each Scheme macro declaration, one defines a sequence of *pattern–output-expression* pairs; a call to a macro expands to the output-expression of the first matching pattern. Similarly to our variadic function templates, the last parameter of a Scheme macro can be declared to be of a variable length. To access the elements of such a variable length parameter, patterns can peel of arguments from the front. This is similar to accessing the elements of function parameter packs: the patterns are defined as the signatures of overloaded functions, output-expressions as the bodies of these functions.

Variadic templates have evolved through a series of proposals to the C++'s ISO standardization committee [27, 28, 29, 30, 31].

6 CONCLUSION

Variadic templates are a lightweight extension to the C++ template system. The new functionality proves useful in the implementation of numerous widely-used template libraries, allowing more concise implementation, faster compile times, and avoiding uses of ugly template and preprocessor trickery that is a cause of unwieldy error messages and "write-only" code. Variadic templates enable type-safe implementations of variadic functions, such as `printf`, and the definition of transparent forwarding functions. The latter provide significant improvements in expressing mixin classes in C++.

We have implemented variadic templates within the GNU C++ compiler and used variadic templates to implement several libraries [1, 3, 5] that use various forms of variadic template emulations. Variadic templates are included in GCC 4.3, both in its C++0x mode and in its implementation of the aforementioned TR1 components. Variadic templates are a part of C++0x, the upcoming revision of the ISO C++ standard, currently in draft form [7].

ACKNOWLEDGMENTS

The authors thank Gary Powell, who has been involved with variadic templates since their inception. David Abrahams and Daveed Vandevoorde have given valuable suggestions during the course of the project. Jens Maurer and Jason Merrill helped to craft the specification of variadic templates for the ISO standard. This work was supported by a grant from the Lilly Endowment.

REFERENCES

[1] Peter Dimov. The Boost Bind library. http://www.boost.org/libs/bind/bind.html, August 2001.

[2] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The Lambda Library: unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[3] Jaakko Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 19:24–35, August 2001.

[4] Dan Marsden Joel de Guzman. Boost Fusion. http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html, 2006.

[5] Douglas Gregor. *Boost.Function*. Boost. http://www.boost.org/doc/html/function.html.

[6] Aleksei Gurtovoy and David Abrahams. The Boost C++ metaprogramming library. www.boost.org/libs/mpl, 2002.

[7] Pete Becker. Working draft, standard for programming language C++. Technical Report N2284=07-0144, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2007.

[8] Douglas Gregor. Variadic templates for GCC. http://www.osl.iu.edu/~dgregor/cpp/variadic-templates.html, August 2006.

[9] Vesa Karvonen and Paul Mensonides. The Boost Preprocessor library. http://www.boost.org/libs/preprocessor/doc/index.html, July 2001.

[10] Douglas Gregor. [libstdc++ patch] tr1::bind, take 2. GNU libstdc++ mailing list, March 2005. http://gcc.gnu.org/ml/libstdc++/2005-03/msg00367.html.

[11] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in C++. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering*, pages 163–177, London, UK, 2001. Springer-Verlag.

[12] Howard E. Hinnant, Dave Abrahams, and Peter Dimov. A proposal to add an rvalue reference to the C++ language. Technical Report N1690=04-0130, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Pro gramming language C++, September 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html.

[13] Joel de Guzman. The Spirit C++ parser framework. http://spirit.sourceforge.net, 2006.

[14] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Technique s from Boost and Beyond*. Addison-Wesley, 2004.

[15] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[16] Todd L. Veldhuizen. C++ templates are Turing complete. `www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf`, 2003.

[17] Matt Austern. (Draft) Technical Report on Standard Library Extensions. Number N1660=04-0100 in ISO C++ Standard Committee 2004-07 mailing, 2004.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[19] ECMA. *C# Language Specification*, June 2005. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[20] David A. Hall. Variable number of generic params. Java bug report 6261297: `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6261279`, 2005.

[21] Walter Bright. D programming language. `http://www.digitalmars.com/d/`, 2006.

[22] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[23] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.

[24] Bruno C. d. S. Oliveira and Jeremy Gibbons. Typecase: a design pattern for type-indexed functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 98–109, New York, NY, USA, 2005. ACM Press.

[25] Samuel Krempp. The Boost Format library. `http://www.boost.org/libs/format`, 2002.

[26] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[27] Douglas Gregor, Gary Powell, and Jaakko Järvi. Typesafe variable-length function and template argument lists. Number N1483=03-0066 in ISO C++ Standard Committee Post-Oxford mailing, April 2003.

[28] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates. Number N1603=04-0043 in ISO C++ Standard Committee Pre-Sydney mailing, February 2004.

[29] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates: Exploring the design space. Number N1704=04-0144 in ISO C++ Standard Committee Pre-Redmond mailing, September 2004.

[30] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates (revision 3). Technical Report N2080=06-0150, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006.

[31] Douglas Gregor, Jaakko Järvi, Jens Maurer, and Jason Merrill. Proposed wording for variadic templates (revision 1). Technical Report N2191=07-0051, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2007.

ABOUT THE AUTHORS

Douglas Gregor is a post-doctoral researcher at Indiana University and a member of the ISO C++ standards committee. His research interests include generic programming, metaprogramming, parallel programming and experimental programming language design for all of the above. He can be reached at dgregor@osl.iu.edu.
See also http://www.generic-programming.org/∼dgregor.

Jaakko Järvi is an assistant professor at Texas A&M University and a member of the ISO C++ standards committee. His research interests include generic and generative programming, programming languages, type systems, and software construction in general. He can be reached at jarvi@cs.tamu.edu.
See also http://parasol.tamu.edu/∼jarvi/.