Proposed Wording for Variadic Templates
(Revision 1)

Authors: Douglas Gregor, Indiana University
Jaakko Järvi, Texas A&M University
Jens Maurer
Jason Merrill, Red Hat

Document number: N2191=07-0051
Revises document number: N2152=07-0012
Date: 2007-03-08
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <dgregor@osl.inu.edu>

Contents
1 Introduction .............................................. 1
2 Proposed Wording ...................................... 1
   2.1 Basic concepts [basic] ................................. 1
   2.2 Expressions [expr] ................................. 2
   2.3 Declarators [dcl] ................................. 3
   2.4 Derived classes [class.derived] .................. 4
   2.5 Special member functions [special] ........... 4
   2.6 Templates [temp] .................................. 4
      2.6.1 Variadic templates [temp.variadic] ....... 10
   2.7 Exception Handling [except] ................. 11

1 Introduction

This document provides proposed wording for variadic templates [3, 1]. Readers unfamiliar with variadic templates are encouraged to read the complete proposal [3]. This document provides only minor, editorial changes to the previous wording [2] for variadic templates. 1

2 Proposed Wording

2.1 Basic concepts [basic]

Modify paragraph 3 of [basic] as follows:

An entity is a value, object, subobject, base class subobject, array element, variable, function, instance of a function, enumerator, type, class member, template, or namespace, or parameter pack.

1Thanks to Daniel Krügler for his detailed suggestions.
2.2 Expressions [expr]

In [expr.post] paragraph 1, modify the grammar production of expression-list as follows:

```
assignment-expression ... opt
expression-list , assignment-expression ... opt
```

Add the following paragraph to [expr.post]:

- An assignment-expression followed by an ellipsis is a pack expansion [temp.variadic].

In [expr.unary] paragraph 1, modify the grammar production of unary-expression as follows:

```
unary-expression:
   postfix-expression
   ++ cast-expression
   -- cast-expression
   unary-operator cast-expression
   sizeof unary-expression
   sizeof ( type-id )
   sizeof ... ( identifier )
   new-expression
   delete-expression
```

Modify paragraph 6 of [expr.sizeof] as follows:

```
The result of sizeof and sizeof... is a constant of type std::size_t. [Note: std::size_t is defined in the standard header <cstdlib> (18.1). – end note]
```

Add the following paragraph to [expr.sizeof] prior to the existing paragraph 6:

- The identifier in a sizeof... expression shall name a parameter pack. The sizeof ... operator yields the number of arguments provided for the parameter pack identifier. The parameter pack is expanded [temp.variadic] by the sizeof ... operator.

Modify paragraph 1 of [expr.const] as follows:

```
In several places, C++ requires expressions that evaluate to an integral or enumeration constant: as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3).

constant-expression:
   conditional-expression
```

An integral constant-expression shall involve only literals of arithmetic types (2.13, 3.9.1), enumerators, non-volatile const variables and static data members of integral and enumeration types initialized with constant expressions (8.5), non-type template parameters of integral and enumeration types, and sizeof, sizeof expressions, and sizeof... expressions. Floating literals (2.13.3) shall appear only if they are cast to integral or enumeration types. Only type conversions to integral and enumeration types shall be used. In particular, except in sizeof expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function call (including new-expressions and delete-expressions), comma operators, and throw-expressions shall not be used.
2.3 Declarators [dcl]

In [dcl.decl] paragraph 4, modify the grammar production of declarator-id as follows:

```
declarator-id:
  ...opt id-expression
  ::opt nested-name-specifier_opt class-name
```

In [dcl.name], paragraph 1, modify the grammar production of abstract-declarator as follows:

```
abstract-declarator:
  ptr-operator abstract-declarator_opt
direct-abstract-declarator
...
```

Modify paragraph 2 of [dcl.fct] as follows:

```
The parameter-declaration-clause determines the arguments that can be specified, and their processing, when the function is called. [ Note: the parameter-declaration-clause is used to convert the arguments specified on the function call; see 5.2.2. – end note ] If the parameter-declaration-clause is empty, the function takes no arguments. The parameter list (void) is equivalent to the empty parameter list. Except for this special case, void shall not be a parameter type (though types derived from void, such as void*, can). If the parameter-declaration-clause terminates with an ellipsis or a function parameter pack [temp.variadic], the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where “...” is not part of an abstract-declarator, “,...” is synonymous with “...”. [ Example: the declaration
```
  int print(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.
```
  print("hello world");
  printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a const char* – end example ] [ Note: the standard header <cstdarg> contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.8). – end note ]
```

Add the following paragraphs to [dcl.fct]:

A parameter-declaration with an ellipsis in its declarator-id is a parameter pack [temp.variadic]. When the parameter-declaration is part of a parameter-declaration-clause, the parameter pack is a function parameter pack with type “pack expansion of T” [temp.variadic], where T is the type of its declarator-id. [ Note: Otherwise, the parameter-declaration is part of a template-parameter-list and the parameter pack is a template parameter pack; see [temp.param]. – end note ] A function parameter pack shall occur at the end of the parameter-declaration-list.

There is a syntactic ambiguity when an ellipsis occurs at the end of a parameter-declaration-clause without a preceding comma. In this case, the ellipsis is parsed as part of the abstract-declarator if the type of the parameter names template parameter packs that have not been expanded; otherwise, it is parsed as part of the parameter-declaration-clause.²

Modify paragraph 3 of [dcl.fct.default] as follows:

A default argument expression shall be specified only in the parameter-declaration-clause of a function declaration or in a template-parameter (14.1). It shall not be specified for a parameter pack. If it is specified in a parameter-declaration-clause, it shall not occur within a declarator or abstract-declarator of a parameter-declaration.

²One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the parameter-declaration-clause) or introducing a name for the parameter (so the ellipsis will be parsed as part of the declarator-id).
In [dcl.init], paragraph 1, modify the grammar production of \textit{initializer-list} as follows:

\begin{verbatim}
initializer-list:
  initializer-clause ... opt
  initializer-list , initializer-clause ... opt
\end{verbatim}

Add the following paragraph to [dcl.init):

\begin{quote}
An \textit{initializer-clause} followed by an ellipsis is a pack expansion \texttt{[temp.variadic]}. \end{quote}

\section*{2.4 Derived classes [class.derived]}

In [class.derived], paragraph 1, modify the grammar production of \textit{base-specifier-list} as follows:

\begin{verbatim}
base-specifier-list:
  base-specifier ... opt
  base-specifier-list , base-specifier ... opt
\end{verbatim}

Add the following paragraph to [class.derived]:

\begin{quote}
A \textit{base-specifier} followed by an ellipsis is a pack expansion \texttt{[temp.variadic]}. \end{quote}

\section*{2.5 Special member functions [special]}

In [class.base.init], paragraph 1, modify the grammar production of \textit{mem-initializer-list} as follows:

\begin{verbatim}
mem-initializer-list:
  mem-initializer ... opt
  mem-initializer ... opt , mem-initializer-list
\end{verbatim}

Add the following paragraph to [class.base.init]:

\begin{quote}
A \textit{mem-initializer} followed by an ellipsis is a pack expansion \texttt{[temp.variadic]} that initializes the base classes specified by a pack expansion in the \textit{base-specifier-list} for the class. \texttt{[Example:]}

\begin{verbatim}
template<typename... Mixins>
struct X
{
  X(const Mixins&... mixins) : Mixins(mixins)... \{ \}
};
\end{verbatim}

\texttt{-- end example } \end{quote}

\section*{2.6 Templates [temp]}

In [temp.param], paragraph 1, modify the grammar production of \textit{type-parameter} as follows:

\begin{verbatim}
type-parameter:
  class ... opt identifier_opt
  class identifier_opt = type-id
  typename ... opt identifier_opt
  typename identifier_opt = type-id
  template < template-parameter-list > class ... opt identifier_opt
  template < template-parameter-list > class identifier_opt = id-expression
\end{verbatim}

Modify paragraph 3 of [temp.param] as follows:
A type-parameter whose identifier does not follow an ellipsis defines its identifier to be a typedef-name (if declared with class or typename) or template-name (if declared with template) in the scope of the template declaration. [Note: because of the name lookup rules, a template-parameter that could be interpreted as either a non-type template-parameter or a type-parameter (because its identifier is the name of an already existing class) is taken as a type-parameter. For example,

class T { /* ... */ };
int i;

template<class T, T i> void f(T t)
{
    T t1 = i; // template-parameters T and i
    ::T t2 = ::i; // global namespace members T and i
}

Here, the template f has a type-parameter called T, rather than an unnamed non-type template-parameter of class T. — end note ]

Modify paragraph 9 of [temp.param] as follows:

A default template-argument is a template-argument (14.3) specified after = in a template-parameter. A default template-argument may be specified for any kind of template-parameter (type, non-type, template) that is not a template parameter pack. A default template-argument may be specified in a template declaration. A default template-argument shall not be specified in the template-parameter-list of the definition of a member of a class template that appears outside of the member’s class. A default template-argument shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default template-argument, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

Add the following paragraph to [temp.param]:

If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a parameter pack [decl], then the template-parameter is a template parameter pack [temp.variadic].

template<class... Types> class Tuple; // Types is a template type parameter pack
template<class T, int... Dims> struct multi_array; // Dims is a non-type template parameter pack

Modify paragraph 11 of [temp.param] as follows:

If a template-parameter of a class template has a default template-argument, all subsequent template-parameters shall either have a default template-argument supplied or be template-parameter packs. If a template-parameter of a class template is a template parameter pack, it must be the last template-parameter. [Note: This is not a requirement These are not requirements for function templates because template arguments might be deduced (14.8.2). ] Example:

template<class T1 = int, class T2> class B; // error

— end example ] — end note ]

In [temp.names], paragraph 1, modify the template-argument-list grammar production as follows:

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

Add the following paragraph to [temp.arg]:
A template-argument followed by an ellipsis is a pack expansion [temp.variadic]. A template-argument pack expansion shall not occur in a simple-template-id whose template-name refers to a class template, unless the template-parameter-list of that class template declares a template parameter pack.

Modify paragraph 4 of [temp.arg] as follows:

When template parameter packs or default template-arguments are used, a template-argument list can be empty. In that case the empty <> brackets shall still be used as the template-argument-list. [Example:

```cpp
template<class T = char> class String;
String<>* p; // OK: String<char>
String* q; // syntax error
```

doesn’t

```cpp

```template<typename ... Elements> class Tuple;
Tuple<>* t; // OK: Elements is empty
Tuple* u; // syntax error
```

– end example ]

Add the following paragraph to [temp.arg.template]:

[ Example:

```
template<class T> class A { /* ... */ };
template<class T, class U = T> class B { /* ... */ };
template<class... Types> class C { /* ... */ };
```

doesn’t

```

template<template<class> class P> class X { /* ... */ };
template<template<class...> class Q> class Y { /* ... */ };
```

– end example ]

Modify paragraph 3 of [temp.class] as follows:

When a member function, a member class, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the template-parameters are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Template parameter packs shall be expanded with an ellipsis in the template argument list. [Example:

```
template<class T1, class T2> struct A {
    void f1();
    void f2();
};
```

doesn’t

```
template<class T2, class T1> void A<T2,T1>::f1() { } // OK
```

doesn’t

```
template<class T2, class T1> void A<T1,T2>::f2() { } // error
```

doesn’t
template<class... Types> struct B {
    void f3();
    void f4();
};

template<class... Types> void B<Types...>::f3() { } // OK

template<class... Types> void B<Types...>::f4() { } // error

end example

In paragraph 9 of [temp.class.spec], add the following bullet:

- An argument shall not contain unexpanded parameter packs. If an argument is a pack expansion [temp.variadic], it shall be the last argument in the template argument list.

Modify paragraph 3 of [temp.func.order] as follows:

To produce the transformed template, for each type, non-type, or template template parameter (or template parameter pack) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template.

Modify the fourth bullet of paragraph 1 of [temp.dep.type] as follows:

- in the definition of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <>. If the nth template parameter is a parameter pack, the nth template argument shall be a pack expansion [temp.variadic] whose pattern is the name of the parameter pack.

Modify paragraph 2 of [temp.dep.type] as follows:

The template argument list of a primary template is a template argument list in which the nth template argument has the value of the nth template parameter of the class template. If the nth template parameter is a template parameter pack, the nth template argument shall be a pack expansion [temp.variadic] whose pattern is the name of the template parameter pack.

In paragraph 4 of [temp.dep.expr], add the following case:

sizeof ... ( identifier )

In [temp.dep.constexpr], add the following paragraph:

Expressions of the following form are value-dependent:

- sizeof ... ( identifier )

Modify paragraph 3 of [temp.arg.explicit] as follows:

Trailing template arguments that can be deduced (14.8.2) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. Trailing template parameter packs not otherwise deduced will be deduced to an empty sequence of template arguments. If all of the template arguments can be deduced, they may all be omitted; in this case, the empty template argument list <> itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the template-id is an lvalue for the function template specialization. [ Example:
template<class X, class Y, class Z> X f(Y);
void g()
{
    int i = f<int>(5.6); // Y is deduced to be double, Z is deduced to an empty sequence
    int j = f(5.6); // ill-formed: X cannot be deduced
    f<void>(f<int, bool>()); // Y for outer f deduced to be
    // int (*)(bool), Z is deduced to an empty sequence
    f<void>(f<int>()); // ill-formed: f<int> does not denote a
    // single function template specialization
}

– end example ]

Modify paragraph 5 of [temp.arg.explicit] as follows:

Template arguments that are present shall be specified in the declaration order of their corresponding template-parameters. The template argument list shall not specify more template-arguments than there are corresponding template-parameters unless one of the template-parameters is a template parameter pack. | Example:

template<class X, class Y, class Z> X f(Y,Z);
template<class... Args> void f2();

void g()
{
    f<int, char*, double>("aa",3.0);
    f<int, char*>("aa",3.0); // Z is deduced to be double
    f<int>("aa",3.0); // Y is deduced to be const char*, and
    // Z is deduced to be double
    f("aa",3.0); // error: X cannot be deduced
    f2<char, short, int, long>(); // okay
}

– end example ]

In [temp.arg.explicit], add the following paragraph:

Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly-specified template arguments. | Example:

template<typename... Types> void f(Types... values);

void g()
{
    f<int*, float*>(0, 0); // Types is deduced to the sequence int*, float*, int
}

– end example]

Modify the first bullet of paragraph 2 in [temp.deduct] as follows:

– The specified template arguments must match the template parameters in kind (i.e., type, non-type, template), and there. | There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack. | otherwise Otherwise, type deduction fails.

Modify paragraph 1 of [temp.deduct.call] as follows:
Template argument deduction is done by comparing each function template parameter type (call it P) with the type of the corresponding argument of the call (call it A) as described below. For each Ai corresponding to a pack expansion [temp.variadic] (that is, the type of a function parameter pack), Pi is the type that results from substituting the ith element of each expanded template parameter pack into the pattern of the expansion. [Example:

```cpp
template<class...> struct Tuple { };

template<class... Types> void f(Types&...);
template<class... Types1, class... Types2> void g(Tuple<Types1...>, Tuple<Types2...>);

void h(int x, float& y)
{
    const int z = x;
    f(x, y, z); // Types is deduced to int, float, const int
    g(Tuple<short, int, long>(), Tuple<float, double>()); // Types1 is deduced to short, int, long
    // Types2 is deduced to float, double
}
```

—end example—

In [temp.deduct.partial], add the following paragraph:

[Note: Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs.—end note] [Example:

```cpp
template<typename...> struct Tuple { };

template<typename... Types> void g(Tuple<Types...>); // #1

template<typename T1, typename... Types> void g(Tuple<T1, Types...>); // #2

template<typename T1, typename... Types> void g(Tuple<T1, Types...>); // #3

    g(Tuple<>()); // calls #1
    g(Tuple<int, float>()); // calls #2
    g(Tuple<int, float&>()); // calls #3
    g(Tuple<int>()); // calls #3
```

—end example—

Add the following paragraphs to [temp.deduct.type]:

For each Ai corresponding to a pack expansion [temp.variadic] (including the type of a function parameter pack), Pi is the type that results from substituting the ith element of each expanded template parameter pack into the pattern of the expansion. [Example:

```cpp
template<class> struct X { };

template<class R, class... ArgTypes> struct X<R(int, ArgTypes...)> { };

template<class... Types> struct Y { };

template<class T, class... Types> struct Y<T, Types...> { }

    template <class... Types> int f (void (*)(Types...));
    void g(int, float);

    X<int> x1; // uses primary template
    X<int(int, float, double)> x2; // uses partial specialization, ArgTypes contains float, double
    X<int(float, int)> x3; // uses primary template
    Y<> y1; // uses primary template, Types is empty
    Y<int&, float&, double&> y2; // uses partial specialization. T is int&, Types contains float, double
    Y<int, float, double> y3; // uses primary template, Types contains int, float, double
    int fv = f(g); // okay, Types contains int, float
– end example]

If the original function parameter associated with A is a function parameter pack, and the function parameter associated with P is not a function parameter pack, then template argument deduction fails. [ Example:

```cpp
template< typename... Args > void f(Args... args); // #1
template< typename T1, typename... Args > void f(T1 a1, Args... args); // #2
template< typename T1, typename T2 > void f(T1 a2, T2 a2); // #3
```

f(); // calls #1
f(1, 2, 3); // calls #2
f(1, 2); // calls #3; non-variadic template #3 is
// more specialized than the variadic templates #1 and #2
– end example]

Add a new subsection to [temp.decls] that contains:

### 2.6.1 Variadic templates [temp.variadic]

1. A *template parameter pack* is a template parameter that accepts zero or more template arguments. [ Example:

```cpp
template< typename... Types > struct Tuple { };
```

Tuple<> t0; // Types contains no arguments
Tuple< int > t1; // Types contains one argument: int
Tuple< int, float > t2; // Types contains two arguments: int and float
Tuple< 0 > error; // Error: 0 is not a type
– end example]

2. A *function parameter pack* is a function parameter that accepts zero or more function arguments. [ Note: The type of a function parameter pack is a pack expansion. – end note ] [ Example:

```cpp
template< typename... Types >
void f(Types... args);
```

f(); // okay: args contains no arguments
f(1); // okay: args contains one int argument
f(2, 1.0); // okay: args contains two arguments, an int and a double
– end example]

3. A *parameter pack* is either a template parameter pack or a function parameter pack.

4. A *pack expansion* is a sequence of tokens which names one or more parameter packs, followed by an ellipsis. The sequence of tokens is called the pattern of the expansion; its syntax depends on the context in which the expansion appears. Pack expansions can occur in the following contexts:

   – In an *expression-list* [expr.post]; the pattern is an *assignment-expression*
   – In an *initializer-list* [dcl.init]; the pattern is an *initializer-clause*
   – In a *base-specifier-list* [class.derived]; the pattern is a *base-specifier*
   – In a *mem-initializer-list* [class.base.init]; the pattern is a *mem-initializer*
   – In a *template-argument-list* [temp.arg]; the pattern is a *template-argument*
   – In an *exception-specification* [except.spec]; the pattern is a *type-id*

[ Example:
template< typename... Types >
  void f(Types... rest);

template< typename... Types >
  void g(Types... rest) {
    f(&rest...); // "&rest..." is a pack expansion, "&rest" is its pattern
  }

  end example ]

A parameter pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a parameter pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more parameter packs that are not expanded by a nested pack expansion. All of the parameter packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a parameter pack that is not expanded is ill-formed. [Example:

template< typename... > struct Tuple { };

template< typename T1, typename T2 > struct Pair { };

template< typename... Args1 >
  struct zip {
    template< typename... Args2 >
      struct with {
        typedef Tuple<Pair<Args1, Args2>...> type;
      };
  };

typedef zip< short, int >::with< unsigned short, unsigned >::type T1;
  // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned> >

typedef zip< short >::with< unsigned short, unsigned >::type T2; // error: different number of arguments specified
  // for Args1 and Args2

template< typename... Args >
  void g(Args... args)
  {
    f(const_cast< const Args >&args...); // okay: "Args" and "args" are expanded
    f(5 ...); // error: pattern does not contain any parameter packs
    f(args); // error: parameter pack "args" is not expanded
    f(h(args...) + args...); // okay: first "args" expanded within h, second "args" expanded within f.
  }

  end example ]

The instantiation of an expansion produces a comma-separated list E₁, E₂, ..., Eₙ, where N is the number of elements in the pack expansion parameters. Each Eᵢ is generated by instantiating the pattern and replacing each pack expansion parameter with its iᵗʰ element. All of the Eᵢ become elements in the enclosing list. [ Note: The variety of list varies with context: expression-list, base-specifier-list, template-argument-list, etc. – end note ]

2.7 Exception Handling [except]

In [except.spec], paragraph 1, modify the type-id-list grammar production as follows:

type-id-list:
  type-id . . . opt
  type-id-list , type-id . . . opt

Add the following paragraph to [except.spec]:

  In an exception-specification, a type-id followed by an ellipsis is a pack expansion [temp.variadic].
References

