Proposed Wording for Variadic Templates (Revision 2)

Authors: Douglas Gregor, Indiana University
Jaakko Järvi, Texas A&M University
Jens Maurer
Jason Merrill, Red Hat
Document number: N2242=07-0102
Revises document number: N2191=07-0051
Date: 2007-04-19
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Contents

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 revises the previous wording [2] to improve the specification of template argument deduction and several other details. [1]

2 Changes since N2191

- The notion of the "type of a function parameter pack" has been completely removed.
- Added change to [expr.call] paragraph 6 to state that either an ellipsis or a function parameter pack can make a function accept more parameters than it has been declared with.
- Added example to [expr.sizeof] paragraph 6 for sizeof....
- Noted that the changes to [expr.const] will disappear with the constant expressions proposal.
- In [dcl.fct] paragraph 3, made *parameter-type-list* include function parameter packs.

[1]Thanks to Daniel Krügler for his detailed suggestions, and feedback from the Core Working Group.

- Added an example with a more complex function parameter pack in the new paragraphs in [dcl.fct].
- In the new paragraphs for [dcl.fct], corrected grammar, spelling, and noted that an ellipsis in a *declarator-id* can only occur in a *parameter-declaration*.
- Corrected example in new paragraph for [class.base.init].
- Cleaned up grammar in [temp.param] paragraph 11, [temp.class] paragraph 3, [temp.class.spec] paragraph 9, [temp.func.order] paragraph 3, [temp.dep.type] paragraphs 1 and 2.
- Added a change to [temp.arg] paragraph 1 to state that template parameter packs correspond to zero or more *template-arguments*.
- In [temp.arg.explicit] paragraph 3, clean up grammar and example.
- Clarified template argument deduction from a call ([temp.deduct.call]) for function parameter packs.
- Clarified template argument deduction from a type ([temp.deduct.type]), and the recursive behavior of template argument deduction.
- Removed [temp.deduct.type] paragraph 19.
- Corrected description of `(T)` and `()` forms in paragraph 8 in [temp.deduct.type].

3 Proposed Wording

3.1 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.

3.2 Expressions [expr]

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

> *assignment-expression* $\ldots_{opt}$
> *expression-list* **,** *assignment-expression* $\ldots_{opt}$

Add the following paragraph to [expr.post]:

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

Modify [expr.call] paragraph 6 as follows:

> A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, ..., or a function parameter pack 8.3.5) than the number of parameters in the function definition (8.4). [*Note*: this implies that, except where the ellipsis (...) or a function parameter pack is used, a parameter is available for each argument. – *end note*]

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:

1 The result of sizeof and sizeof... is a constant of type std::size_t. [*Note*: std::size_t is defined in the standard header <cstddef> (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.

[*Example*:

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

– *end example*]

Modify paragraph 1 of [expr.const] as follows:
Editorial note: the constant expressions proposal removes this paragraph entirely, making the changes below unnecessary.

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.

3.3 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 printf(const char∗, ...);
> ```
>
> declares a function that can be called with varying numbers and types of arguments.
>
> ```
> printf("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*]

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

> A single name can be used for several different functions in a single scope; this is function overloading (clause 13). All declarations for a function shall agree exactly in both the return type and the *parameter-type-list*. The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively. After producing the list of parameter types, several transformations take place upon these types to determine the function type. Any *cv-qualifier* modifying a parameter type is deleted. [*Example*: the type void(∗)(const int) becomes void(∗)(int) – *end example*] Such *cv-qualifier*s affect only the definition of the parameter within the body of the function; they do not affect the function type. If a *storage-class-specifier* modifies a parameter type, the specifier is deleted. [*Example*: register char∗ becomes char∗ – *end example*] Such *storage-class-specifier*s affect only the definition of the parameter within the body of the function; they do not affect the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function's *parameter-type-list*.

Add the following paragraphs to [dcl.fct]:

> A *declarator-id* or *abstract-declarator* containing an ellipsis shall only be used in a *parameter-declaration*. Such a *parameter-declaration* is a parameter pack [temp.variadic]. When it is part of a *parameter-declaration-clause*, the parameter pack is a function parameter pack [temp.variadic]. [*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, if present, shall occur at the end of the *parameter-declaration-list*. The type T of the *declarator-id*

of the function parameter pack shall contain a template parameter pack; each template parameter pack in T is expanded by the function parameter pack.

[*Example*:

```
template<typename... T> void f(T (* ...t)(int, int));

int add(int, int);
float subtract(int, int);

void g() {
  f(add, subtract);
}
```

– *end example*]

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 a template parameter pack that has 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*.

In [dcl.init], paragraph 1, modify the grammar production of *initializer-list* as follows:

initializer-list:
  initializer-clause $\ldots_{opt}$
  initializer-list **,** *initializer-clause* $\ldots_{opt}$

Add the following paragraph to [dcl.init]:

> An *initializer-clause* followed by an ellipsis is a pack expansion [temp.variadic].

3.4 Derived classes [class.derived]

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

base-specifier-list:
  base-specifier $\ldots_{opt}$
  base-specifier-list **,** *base-specifier* $\ldots_{opt}$

Add the following paragraph to [class.derived]:

> A *base-specifier* followed by an ellipsis is a pack expansion [temp.variadic].

3.5 Special member functions [special]

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

mem-initializer-list:
  mem-initializer $\ldots_{opt}$
  mem-initializer $\ldots_{opt}$**,** *mem-initializer-list*

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

[2] One 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*).

A *mem-initializer* followed by an ellipsis is a pack expansion [temp.variadic] that initializes the base classes specified by a pack expansion in the *base-specifier-list* for the class. [*Example*:

```
template<typename... Mixins>
class X : public Mixins...
{
public:
  X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

– *end example*]

3.6 Templates [temp]

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

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
```

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-lists* 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 [dcl.fct], 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~~each subsequent *template-parameter~~s~~* shall either have a default *template-argument* supplied or be a template parameter pack. 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 $\ldots_{opt}$
  template-argument-list **,** *template-argument* $\ldots_{opt}$

Modify paragraph 1 of [temp.arg] as follows:

1 There are three forms of *template-argument*, corresponding to the three forms of *template-parameter*: type, non-type and template. The type and form of each *template-argument* specified in a *template-id* shall match the type and form specified for the corresponding parameter declared by the template in its *template-parameter-list*. When the parameter declared by the template is a template parameter pack, it will correspond to zero or more *template-arguments*. [*Example*:

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
  // ...
};

Array<int> v1(20);
typedef std::complex<double> dcomplex; // \tcode{std::complex}\ is a standard
          // library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
  v1[3] = 7;
  v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

– *end example*]

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-argument*s 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*:

```
template<class T = char> class String;
String<>∗ p; // OK: String<char>
String∗ q; // syntax error
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 { /∗ ... ∗/ };

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

X<A> xa; // okay
X<B> xb; // ill-formed: default arguments for the parameters of a template template argument are ignored
X<C> xc; // ill-formed: a template parameter pack does not match a template parameter

Y<A> ya; // ill-formed: a template parameter pack does not match a template parameter
Y<B> yb; // ill-formed: a template parameter pack does not match a template parameter
Y<C> yc; // okay
```

– *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-parameter*s 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. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [*Example*:

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

template<class T2, class T1> void A<T2,T1>::f1() { } // OK
template<class T2, class T1> void A<T1,T2>::f2() { } // error
```

```
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 an unexpanded parameter pack. 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 (including template parameter packs thereof) 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 is 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 is 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*. A template parameter pack 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> X f(Y);
template<class X, class Y, class... Z> X g(Y);
void ~~g~~h()
{
  int i = f<int>(5.6); // Y is deduced to be double
  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)
  f<void>(f<int>); // ill-formed: f<int> does not denote a
                         // single function template specialization

  int k = g<int>(5.6); // Y is deduced to be double, Z is deduced to an empty sequence
  f<void>(g<int, bool>); // Y for outer f deduced to be
                         // int (*)(bool), Z is deduced to an empty sequence
}
```

– *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-parameter*s. The template argument list shall not specify more *template-argument*s than there are corresponding *template-parameter*s unless one of the *template-parameter*s 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, 0); // Types is 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 a function parameter pack, the type A of each remaining argument of the call is compared with the type P of the parameter of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. [*Note*: A function parameter pack can only occur at the end of a *parameter-declaration-list* [dcl.fct]. – *end note*]
>
> [*Example*:
>
> ```
> template<class... Types> void f(Types&...);
> template<class T, class... Types> void g(T1, Types...);
>
> void h(int x, float& y)
> {
>   const int z = x;
>   f(x, y, z); // Types is deduced to int, float, const int
>   g(x, y, z); // T1 is deduced to int, Types is deduced to float, int
> }
> ```
>
> – *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*:
>
> ```
> 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*]

Modify paragraph 8 of [temp.deduct.type] as follows:

> A template type argument T, a template template argument TT or a template non-type argument i can be deduced if P and A have one of the following forms:
>
> T
> *cv-list* T
> T*
> T&
> T[*integer-constant*]
> *template-name*<T> (where *template-name* refers to a **class template**)
> *type*(T)
> T()
> T(T)
> T *type*::*
> *type* T::*
> T T::*
> T (*type*::*)()
> *type* (T::*)()

```
type (type::*)(T)
type (T::*)(T)
T (type::*)(T)
T (T::*)()
T (T::*)(T)
type[i]
template-name<i> (where template-name refers to a class template)
TT<T>
TT<i>
TT<>
```

where (T) represents ~~argument lists~~a *parameter-type-list* where at least one ~~argument~~parameter type contains a T, and () represents ~~argument lists~~a *parameter-type-list* where no parameter type contains a T. Similarly, <T> represents template argument lists where at least one argument contains a T, <i> represents template argument lists where at least one argument contains an i and <> represents template argument lists where no argument contains a T or an i.

Add the following new paragraphs after paragraph 8 of [temp.deduct.type]:

If P has a form that contains <T> or <i>, then each argument P_i of the respective template argument list of P is compared with the corresponding argument A_i of the corresponding template argument list of A. If the template argument list of P contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If P_i is a pack expansion, then the pattern of P_i is compared with each remaining argument in the template argument list of A. Each comparison deduces template arguments for the template parameters in P_i and for subsequent positions in the template parameter packs expanded by P_i.

Similarly, if P has a form that contains (T), then each parameter type P_i of the respective *parameter-type-list* of P is compared with the corresponding parameter type A_i of the corresponding *parameter-type-list* of A. If the *parameter-declaration* corresponding to P_i is a function parameter pack, then the type of its parameter is compared with each remaining parameter type in the *parameter-type-list* of A. Each comparison deduces template arguments for the template parameters in P_i and for subsequent positions in the template parameter packs expanded by the function parameter pack. [*Note*: A function parameter pack can only occur at the end of a *parameter-declaration-list* [dcl.fct]. – *end note*]

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

[*Note*: Template argument deduction involving parameter packs ([temp.variadic]) can deduce zero or more arguments for each parameter pack. – *end note*] [*Example*:

```
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*:

```
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*]

Remove paragraph 19 of [temp.deduct.type]:

~~[*Note*: a default *template-argument* cannot be specified in a function template declaration or definition; therefore default *template-arguments* cannot be used to influence template argument deduction. – *end note*]~~

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

3.6.1 Variadic templates [temp.variadic]

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

```
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. [*Example*:

```
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*]

5 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*]

6 The instantiation of an expansion produces a comma-separated list E_1, E_2, ..., E_N, where N is the number of elements in the pack expansion parameters. Each E_i is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element. All of the E_i 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*]

3.7 Exception Handling [except]

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

type-id-list:
  type-id $\ldots_{opt}$
  type-id-list **,** *type-id* $\ldots_{opt}$

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

[1] D. Gregor. A brief introduction to variadic templates. Number N2087=06-0157 in ANSI/ISO C++ Standard Committee Pre-Portland mailing, 2006.

[2] D. Gregor, J. Järvi, J. Maurer, and J. Merrill. Proposed wording for variadic tempplates (revision 1). Technical Report N2191=07-0051, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2007.

[3] D. Gregor, J. Järvi, and G. Powell. Variadic templates (revision 3). Number N2080=06-0150 in ANSI/ISO C++ Standard Committee Pre-Portland mailing, October 2006.