Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Variable-Argument-Count Templates

variadic-templates

By Andrei Alexandrescu

Variadic templates provide language-level support for specifying templates that accept an arbitrary number of template arguments.

Description

description-variadic

Experience with C++03 revealed a recurring need to specify a class or function that accepts an arbitrary number of arguments. The C++03 workarounds often require considerable boilerplate and hardcoded limitations that impede usability. Consider, for example, a function `concat` taking zero or more **`const`** `std::string&`, **`const`** **`char`**`*`, or **`char`** arguments and returning an `std::string` that is the concatenation of that argument sequence:

std::string

```
#include <string>  // std::string

std::string d = "d";
std::string str0 = concat();               // str0 == "" (by definition)
std::string str1 = concat("apple");        // str1 == "apple"
std::string str2 = concat('b', "ccd");     // str2 == "bccd"
std::string str3 = concat(d, 'e', "fg");   // str3 == "defg"
```

One advantage of using a variadic function, such as `concat`, instead of repeatedly using the `+` operator is that the `concat` function can build the destination string exactly once, whereas each invocation of `+` creates and returns a new `std::string` object.

A simpler example is a variadic function, `add`, that calculates the sum of zero or more integer values supplied to it:

```
int v0 = add();         // v0 ==  0 (by definition)
int v1 = add(3);        // v1 ==  3
int v2 = add(-6, 2);    // v2 == -4
int v4 = add(7, 1, 4);  // v3 == 12
// ...
```

Historically, variadic functions, such as `concat` and `add` (above), were implemented as a suite of related non-variadic functions accepting progressively more arguments, up to some arbitrary limit (e.g., 20) chosen by the implementer:

```
int add();
int add(int);
int add(int, int);
int add(int, int, int);
// ...
// ... declarations from 4 to 19 int parameters elided
// ...
int add(int, int, int, /* 16 more int parameters elided, */ int);
```

Given the existence of an identity value (zero for addition), an alternative approach would be to have a single `add` function but with each of the parameters defaulted:

```
int add(int=0, int=0, /* 17 more defaulted parameters omitted */ int=0);
```

However, `concat` cannot use the same approach because each of its arguments may be a **`char`**, a **`const char*`**, or an `std::string` (or types convertible thereto), and no single type accommodates all of these possibilities. To accept an arbitrary mix of arguments of the allowed types with maximal efficiency, the brute-force approach would require the definition of an exponential number of overloads taking any combination of **`char`**, **`const char*`**, and **`const std::string&`**, again up to some implementation-chosen maximum number of parameters, N.

With such an approach, the number of required overloads is $O(3^N)$, which means that accommodating a maximum of just 5 arguments would require 283 overloads and 10 arguments would require 88,573 overloads![1] Defining a suite of N function *templates* instead is one approach to avoiding this combinatorial explosion of overloads:

```
#include <string>  // std::string

std::string concat();                                       // 0 arguments

template <typename T1>
std::string concat(const T1&);                              // 1 argument

template <typename T1, typename T2>
std::string concat(const T1&, const T2&);                   // 2 arguments

template <typename T1, typename T2, typename T3>
std::string concat(const T1&, const T2&, const T3&);        // 3 arguments

// ...
// ... similar declarations taking up to, say, 20 parameters
// ...
```

Using conventional function templates, we can drastically reduce the volume of source code required, albeit with some manageable increase in implementation complexity.

Each of the $N+1$ templates can be written to accept any combination of its M arguments ($0 \leq M \leq N$) such that each parameter will independently bind to a **`const char*`**, an `std::string`, or a **`char`** with no unnecessary conversions or extra copies at run time. Of the exponentially many possible `concat` template instantiations, the compiler generates — on demand — only those overloads that are actually invoked.

With the introduction of variadic templates in C++11, we are now able to represent variadic functions such as `add` or `concat` with just a single template that expands automatically to accept any number of arguments of any appropriate types — all by, say, **`const`** *lvalue* reference:

[1] The `std::string_view` standard library type introduced with C++17 would help here because it accepts conversion from both **`const char*`** and `std::string` and incurs no significant overhead. However, `std::string_view` cannot be initialized from a single **`char`**, so we'd be looking at $O(2^N)$ instead of $O(3^N)$ — not a dramatic improvement.

```
    std::string

template <typename... Ts>
std::string concat(const Ts&...);
     // Return a string that is the concatenation of a sequence of zero or
     // more character or string arguments --- each of potentially distinct
     // C++ type and passed by const lvalue reference.
```

A variadic function template will typically be implemented with **recursion** to the same function with fewer parameters. Such function templates will typically be accompanied by an overload (templated or not) that implements the lower limit, in our case, the overload having exactly zero parameters:

```
std::string concat();
    // Return an empty string ("") of length 0.
```

The non-template overload above declares `concat` taking no parameters. Importantly, this overload will be preferred for calls of `concat` with no arguments because it's a better match than the variadic declaration, even though the variadic declaration would also accept zero arguments.

Having to write just two overloads to support any number of arguments has clear advantages over writing dozens of overloaded templates: (1) there is no hard-coded limit on argument count, and (2) the source is dramatically smaller, more regular, and easier to maintain and extend — e.g., it would be easy to add support for efficiently passing by forwarding reference (see Section 2.1."**??**" on page **??**). A second-order effect should be noted as well. The costs of defining variadic functions with C++03 technology are large enough to discourage such an approach in the first place, unless overwhelming efficiency motivation exists; with C++11, the low cost of defining variadics often makes them the simpler, better, and more efficient choice altogether.

Variadic *class* templates are another important motivating use case for this language feature.

A tuple is a generalization of `std::pair` that, instead of comprising just two objects, can store an arbitrary number of objects of heterogeneous types:

```
    std::string

Tuple<int, double, std::string> tup1(1, 2.0, "three");
    // tup1 holds an int, a double, and an std::string

Tuple<int, int> tup2(42, 69);
    // tup2 holds two ints
```

`Tuple` provides a container for a specified set of types, in a manner similar to a **`struct`**, but without the inconvenience of needing to introduce a new **`struct`** definition with its own name. As shown in the example above, the tuples are also initialized correctly according to the specified types (e.g., `tup2` contains two integers initialized, respectively, with `42` and `69`).

In C++03, an approximation to a tuple could be improvised by composing an `std::pair` with itself:

```
#include <utility>  // std::pair
```

```
std::pair<int, std::pair<double, long>> v;
    // Define a holder of an int, a double, and an long, accessed as
    // v.first, v.second.first, and v.second.second, respectively.
```

Composite use of `std:pair` types could, in theory, be scaled to arbitrary depth; defining, initializing, and using such types, however, is not always practical. Another approach commonly used in C++03 and similar to the one suggested for the `add` function template above is to define a template class, e.g., `Cpp03Tuple`, having many parameters (e.g, 9), each defaulted to a special marker type (e.g., `None`), indicating that the parameter is not used:

```
struct None { };  // empty "tag" used as a special "not used" marker in Tuple

template <typename T1 = None, typename T2 = None, typename T3 = None,
          typename T4 = None, typename T5 = None, typename T6 = None,
          typename T7 = None, typename T8 = None, typename T9 = None>
class Cpp03Tuple;
    // struct-like class containing up to 9 data members of arbitrary types
```

`Cpp03Tuple` may be used to store, access, and modify up to 9 values together, e.g., `Cpp03Tuple<int, int, std::string>` would consist of two `int`s and an `std::string`.

`Cpp03Tuple`'s implementation uses a variety of **metaprogramming** tricks to detect which of the 9 type slots are used. This is the approach taken by `boost::tuple`,[2] an industrial-strength tuple implemented using C++03-era technology. In contrast, the variadic-template-based declaration (and definition) of a modern C++ tuple is much simpler:

```
template <typename... Ts>
class Cpp11Tuple;  // class template storing an arbitrary sequence of objects
```

C++11 introduced the standard library class template `std::tuple`, declared in a manner similar to `Cpp11Tuple`.

An `std::tuple` can be used, for example, to return multiple values from a function. Suppose we want to define a function, `minAverageMax`, that — given a **range** of double values — returns, along with its cardinality, its minimum, average, and maximum values. The interface for such a function in C++03 might have involved multiple output parameters passed, e.g., by non`const` *lvalue* reference:

```
#include <cstddef>  // std::size_t

template <typename Iterator>
void minAverageMax(std::size_t& numValues,   // (out only) number of inputs
                   double& minimum,          // (out only) minimum value
                   double& average,          // (out only) average value
                   double& maximum,          // (out only) maximum value
                   Iterator b, Iterator e);  // input range
    // Load into the specified numValues, minimum, average, and maximum
    // the corresponding values extracted from the specified range [b, e).
```

Alternatively, one could have define a separate `struct` (e.g., `MinAverageMaxRes`) and incorporate that into the interface of the `minAverageMax` function:

[2] https://github.com/boostorg/tuple/blob/develop/include/boost/tuple/tuple.hpp

```
#include <cstddef>  // std::size_t

struct MinAverageMaxRes  // used in conjunction with minAverageMax (below)
{
    std::size_t count;  // number of input values
    double min;         // minimum value
    double average;     // average value
    double max;         // maximum value
};

template <typename Iterator>
MinAverageMaxRes minAverageMax(Iterator b, Iterator e);
```

Adding a helper **aggregate** such as `MinAverageMax` (above) works but demands a fair amount of boilerplate coding that might not be reusable or otherwise worth naming. The C++11 library abstraction `std::tuple` allows code to define such simple aggregates "on the fly" (as needed):

```
#include <tuple>    // std::tuple
#include <cstddef>  // std::size_t

typedef std::tuple<std::size_t, double, double, double> MinAverageMaxRes;
    // type alias for a standard tuple of four specific scalar values.

template <typename Iterator>
MinAverageMaxRes minAverageMax(Iterator b, Iterator e);
    // Return the cardinality, min, average, and max of the range [b, e].
```

We can now use our `minAverageMax` function to extract the relevant fields from a vector of `double` values:

```
#include <vector>  // std::vector

void test(const std::vector<double>& v)
{
    MinAverageMaxRes res = minAverageMax(v.begin(), v.end());  // calculate
    std::size_t num = std::get<0>(res);  // fetch slot 0, the number of values
    double      min = std::get<1>(res);  // fetch slot 1, the minimum value
    double      ave = std::get<2>(res);  // fetch slot 2, the average value
    double      max = std::get<3>(res);  // fetch slot 3, the maximum value
    // ...
    std::get<2>(res) = 0.0;              // store 0 in slot 2 (just FYI)
    // ...
}
```

Note that elements in a tuple are accessed in a numerically indexed manner (beginning with slot 0) using the standard function template `std::get` for both reading and writing.

There are other motivating uses of variadic templates, such as allowing generic code to forward arguments to other functions, notably constructors, without the need to know in advance the number of arguments required. Related artifacts added to the C++ standard library include `std::make_shared`, `std::make_unique`, and the `emplace_back` member

function of `std::vector` (see *Use Cases — Object factories* on page 53 **[AUs: There is no subsection called "Factory functions." Did you mean the Object factories section? Response: Yes, it should be object factories]**).

There is a lot to unpack here. We will begin our journey by understanding variadic *class* templates as **generic types** that provide a solid basis for understanding variadic *function* templates. In practice, however, variadic *function* templates are arguably more frequently applicable outside of advanced metaprogramming; see *Variadic function templates* on page 18.

ariadic-class-templates

Variadic class templates

Suppose we want to create a class template `C` that can take zero or more template type arguments. C++11 introduces new syntax — based on the ellipsis (`...`) token — that supports such variadic parameter lists:

```
template <typename... Ts> class C;
    // The class template C can be instantiated with a sequence of zero or
    // more template arguments of arbitrary type. Because this is only a
    // declaration, the parameter name, Ts (for "types"), isn't used.
```

The declaration above introduces a class template `C` that can accept an arbitrary sequence of type parameters optionally represented here (for documentation purposes) by the identifier `Ts` (for "Types").

First we point out that the ellipsis (`...`), just like `++` or `==`, is parsed as a separate token; hence, any whitespace around the ellipsis is optional. The common style, and the one used in the C++ Standard documents, follows the typographic convention in written prose: `...` abuts to the left and is followed by a single space as in the declaration of `C` above.

Whether we use **`typename`** (as we do throughout this book) or **`class`** to introduce a **template type parameter** is entirely a matter of style:

```
template <typename... Ts> class C;  // style used throughout this book
template <class... Ts> class C;     // style used in the C++ Standard
```

Note that, as with non-variadic template parameters, providing a name to represent the supplied template "parameters" (a.k.a. **template parameter pack**, see below) is optional:

```
template <typename... Ts> class C;  // with    name identifying parameter pack
template <typename...> class C;     // without name identifying parameter pack
```

When an ellipsis token (`...`) appears *after* **`typename`** or **`class`** and before the optional type-parameter name (e.g., `Ts`), it introduces a template parameter pack:

```
template <typename... Ts>  // Ts names a template parameter pack.
class C;
```

We call entities such as `Ts` template parameter packs (as opposed to parameter packs) to distinguish them from **function parameter packs** (see *Variadic function templates* on page 18), **non-type parameter packs** (see *Non-type template parameter packs* on page 31 **[AUs: There is no section called "Non-type parameter packs" Response: non-type template parameter packs is the correct intraref]**), and **template template parameter packs** (see *Template parameter packs* on page 32 The phrase **parameter pack**

is a conflation of the four and also a casual reference to either of them whenever there is no ambiguity.

This syntactic form of `...` is used primarily in **declarations** (including those associated with **definitions**):

```
template <typename... Ts>
class C { /*...*/ };  // definition of class C
```

The same `...` token, when it appears to the *right* of an existing template parameter pack (e.g., `Ts`), is used to *unpack* it:

```
#include <vector>  // std::vector

template <typename... Ts> class C  // definition of class C
{
    std::vector<Ts...> d_data;
        // using ... to unpack a template parameter pack
};
```

In the example above, the *unpacking* results in a comma-separated list of the types with which `C` was instantiated. The `...` token is used after the template parameter pack name `Ts` to recreate the sequence of arguments originally passed to the instantiation of `C`. Referring to our example above, we might choose to create an object, `x`, of type `C<int>`:

```
C<int> x;  // has data member, d_data, of type std::vector<int>
```

We might, instead, consider creating an object, `y`, that also passes `std::allocator<char>` to `C`'s instantiation:

```
C<char, std::allocator<char>> y;
    // y.d_data has type std::vector<char, std::allocator<char>>.
```

And so on. That is, `class C` can be instantiated with any sequence of types that is supported by the `d_data` member variable.

An instantiation such as `C<float, double>` would correspondingly attempt to instantiate `std::vector<float, double>`, which is in error and would cause the instantiation of `C` to fail. Several other cases and patterns of *unpacking* of template parameter packs exist and are described in detail in *Template parameter packs* on page 32.

Continuing our pedagogical discussion, let's **define** an empty variadic class template, `D`:

```
template <typename...> class D { };  // empty variadic-class-template definition
```

We can now create explicit instantiations of class template `D` by providing it with any number of *type* arguments:

```
D<>               d0;  // instantiation of D with no type arguments
D<int>            d1;  // instantiation of D with a single int argument
D<int, int>       d2;  // instantiation of D with two int arguments
D<int, const int> d3;  // Note that d3 is a distinct type from d2.
D<double, char>   d4;  // instantiation of D with a double and char
D<char, double>   d5;  // Note that d5 is a distinct type from d4.
D<D<>, D<int>>   d6;   // instantiation of D with two UDT arguments
```

The number and order of arguments are part of the instantiated type, so each of the objects `d0` through `d6` above has a distinct C++ type:

```
void f(const D<double, char>&);  // (1) overload of function f
void f(const D<char, double>&);  // (2) overload of function f

void test()
{
    f(d4);                       // invokes overload (1)
    f(d5);                       // invokes overload (2)
}
```

The sections that follow examine in full detail how to

- declare variadic *class* and *function* templates using parameter packs
- make use of variadic argument lists in the implementation of class definitions and function bodies

emplate-parameter-packs

Template parameter packs

A template parameter pack is the name representing a list of zero or more parameters following **`class...`** or **`typename...`** within a variadic template declaration.

Let's take a closer look at just the *declaration* of a variadic class template, `C`:

```
template <typename... Ts> class C { };
```

Here, the identifier `Ts` names a template parameter pack, which — as we saw previously — can bind to any sequence of explicitly supplied types including the empty sequence:

```
C<> c0;             // OK, instantiation of C with no type arguments
C<int> c1;          // OK, instantiation of C with a single int argument
C<float, bool> c2;  // OK, instantiation of C with two type arguments
```

Passing an argument to `C` that is not a type, however, is not permitted:

```
C<128> cx0;           // Error, expecting type template argument, 128 provided
C<std::vector> cx1;   // Error, expecting type argument, template name provided
C<int, 42, int> cx2;  // Error, expecting type argument in second position
```

Template parameter packs can appear together with simple template parameters, with one restriction: **Primary class template declarations** allow at most one variadic parameter pack at the end of the template parameter list. (Recall that in standard C++ terminology, the **primary declaration** of a class template is the first declaration introducing the template's name; all specializations and partial specializations of a class template require the presence of a primary declaration.)

```
template <typename... Ts>
class C0  { };  // OK

template <typename T, typename... Ts>
class C1  { };  // OK
```

```
template <typename T, typename U, typename... Ts>
class C2  { };  // OK

template <typename... Ts, typename... Us>
class Cx0 { };  // Error, more than one parameter pack

template <typename... Ts, typename T>
class Cx1 { };  // Error, parameter pack must be the last template parameter
```

There is no way to specify a default for a parameter pack; however a parameter pack can follow a defaulted parameter:

```
template <typename... Ts = int>
class Cx2 { };  // Error, a parameter pack cannot have a default.

template <typename T = int, typename... Ts = char>
class Cx3 { };  // Error, a parameter pack cannot have a default.

template <typename T = int, typename... Ts>
class C3  { };  // OK

C3<> c31;                   // OK, T=int, Ts=<>
C3<char> c32;               // OK, T=char, Ts=<>
C3<char, double, int> c33;  // OK, T=char, Ts=<double, int>
```

What can we do with a template parameter pack? Template parameter packs are of a so-called *kind* distinct from other C++ entities. They are not types, values, or anything else found in C++03. As such, parameter packs are not subject to any of the usual operations one might expect:

```
template <typename... Ts>
class C
{
    C<Ts>* next;        // Error, cannot use unexpanded parameter pack Ts
    Ts memberVariable;  // Error, cannot use unexpanded parameter pack Ts
    typedef Ts Ts1;     // Error, cannot use unexpanded parameter pack Ts
    using Ts2 = Ts;     // Error, cannot use unexpanded parameter pack Ts
};
```

There is no way to use a parameter pack in unexpanded form. Once introduced, the name of a parameter pack can occur only as part of a **pack expansion**. We've already encountered the simple pack expansion `Ts...`, which expands to the list of types to which `Ts` is bound. That expansion is only allowed in certain contexts where multiple values would be allowed:

```
template <typename... Ts>
class C
{
    C<Ts...>* d_next;        // OK, instantiate C in definition of member
    C<Ts...> f0();           // OK, instantiate C in member function signature
    void f1(C<Ts...>&);      // OK, instantiate C in member function signature
    Ts... d_memberVariable;  // Error, expansion as a member type
```

```
    typedef Ts... Ts1;       // Error, expansion as a typedef type
    using Ts2 = Ts...;       // Error, expansion as a using declaration argument
};
```

Note that the code above remains invalid even if `Ts` contains a single type, e.g., in the instantiation `C<int>`. Pack expansion is not textual and not allowed everywhere; it is allowed only in certain well-defined contexts. The first such context, as showcased with the member variable `next` in the code example above, is inside a **template argument list**. Note how the pattern may be surrounded by other parameters or expansions:

```
#include <map>  // std::map

template <typename... Ts>
class C
{
    void arbitraryMemberFunction()      // for illustration purposes
    {
        C<Ts...>                  v0;  // OK, same type as *this
        C<int, Ts...>             v1;  // OK, expand after another argument
        C<Ts..., char>            v2;  // OK, expand before another argument
        C<char, Ts..., int>       v3;  // OK, expand in between
        C<void, Ts..., Ts...>     v4;  // OK, two expansions
        C<char, Ts..., int, Ts...> v5;  // OK, no need for them to be adjacent
        C<void, C<Ts...>, int>    v6;  // OK, expansion nested within
        C<Ts..., C<Ts...>, Ts...> v7;  // OK, mix of expansions
        std::map<Ts...>           v8;  // OK, works with non-variadic template
    }
};
```

The examples above illustrate how pack expansion is not done textually, like a macro expanded by the C preprocessor. A simple textual expansion of `C<char, Ts..., int>` would be `C<char, , int>` if `Ts` were empty (i.e., inside the instantiation `C<>`). **Parameter pack expansion** is syntactic and "knows" to eliminate any spurious commas, caused by the expansion of empty parameter packs.

Within the context of a template argument list, `Ts...` is not the only pattern that can be expanded; any template instantiation using `Ts` (e.g. `C<Ts>...`) can be expanded as a unit. The result is a list of template instantiations using each of the types in `Ts`, in turn. Note that, in the example below, the expansions `C<Ts...>` and `C<Ts>...` are both valid but produce different results:

```
#include <vector>  // std::vector

template <typename... Ts> class D
{
    void memberFunction()
    {
        D<Ts...> v0;              // OK, same type as *this
        D<D<Ts...>> v1;           // OK, expand to D<D<T0, T1>, ...>
        D<D<Ts>...>  v2;          // OK, expand to D<D<T0>, D<T1>, ...>
        D<std::vector<Ts>...> v3; // OK, expand to C1<std::vector<T0>, ...>
```

```
    }
};
```

The second important **parameter pack expansion context** for type parameter packs is in a **base specifier list**. All patterns that form a valid base specifier are allowed:

```
template <typename... Ts>   // zero or more arguments
class D1 : public Ts...     // publicly inherit T0, T1, ...
{ /*...*/ };

template <typename... Ts>
class D2 : public D<Ts>...  // publicly inherit D<T0>, D<T1>, ...
{ /*...*/ };

template <typename... Ts>
class D3 : public D<Ts...>  // publicly inherit D<T0, T1, ...>
{ /*...*/ };
```

The access control specifiers — **public**, **protected**, and **private** — can be applied as usual, though, within a single expansion pattern, the access specifier must be the same for all expanded elements:

```
template <typename... Ts>
class D4 : private Ts...   // privately inherit T0, T1, ...
{ /*...*/ };

template <typename... Ts>
class D5 : public Ts..., private D<int, Ts>...
    // publicly inherit T0, T1, ...
    // and privately inherit D<int, T0>, D<int, T1>, ...
{ /*...*/ };
```

Pack expansions can be freely mixed with simple base specifiers:

```
class AClass1 { /*...*/ };  // arbitrary class definition
class AClass2 { /*...*/ };  // arbitrary class definition

template <typename... Ts>
class D6 : protected AClass1, public Ts...                    // OK
{ /*...*/ };

template <typename... Ts>
class D7 : protected AClass1, private Ts..., public AClass2  // OK
{ /*...*/ };
```

If the parameter pack being expanded (e.g., `Ts`, in the code snippet above) is empty, the expansion does not introduce any base class. Notice, again, how the expansion mechanism is semantic, not textual, e.g., in the instantiation `D7<>` the fragment **`private`** `Ts...,` disappears entirely, leaving `D7<>` with `AClass1` as a **`protected`** base and `AClass2` as a **`public`** base.

To recap, the two essential parameter expansion contexts for template parameter packs are inside a template argument list and in a base specifier list.

ariadic-class-templates

Specialization of variadic class templates

Recall from prior to C++11 that after a class template is introduced by a primary class template declaration, it is possible to create **specializations** and **partial specializations** of that class template. We can declare specializations of a variadic class template by supplying zero or more arguments to its parameter pack:

```
template <typename... Ts> class C0;  // primary class template declaration

template <> class C0<>;              // specialize C0 for Ts=<>
template <> class C0<int>;           // specialize C0 for Ts=<int>
template <> class C0<int, void>;     // specialize C0 for Ts=<int, void>
```

Similar specializations can be applied to class templates that have other template parameters preceding the template parameter pack. The nonpack template parameters must be matched exactly by the arguments, followed by zero or more arguments for the parameter pack:

```
template <typename T, typename... Ts>
class C1;                            // primary class template declaration

template <> class C1<int>;           // Specialize C1 for T=int, Ts=<>.
template <> class C1<int, void>;     // Specialize for T=int, Ts=<void>.
template <> class C1<int, void, int>; // Specialize for T=int, Ts=<void, int>.
template <> class C1<>;              // Error, too few template arguments
```

Partial specializations of a class template may take multiple parameter packs because some of the types involved in the partial specialization may themselves use parameter packs. Consider, for example, a variadic class template, `Tuple`, defined as having exactly one parameter pack:

```
template <typename... Ts> class Tuple  // variadic class template
{ /*...*/ };
```

Further assume a primary declaration of a variadic class template , `C2`, also having one type parameter pack. We also introduce definitions in addition to declarations so we can instantiate `C2` later:

```
template <typename... Ts> class C2
    // (0) primary declaration of variadic class template C2
{ /*...*/ };
```

This simple setup allows a variety of partial specializations. First, we can partially specialize `C2` for exactly two `Tuple`s instantiated with the same exact types:

```
template <typename... Ts>
class C2<Tuple<Ts...>, Tuple<Ts...>>  // (1) two identical Tuples
{ /*...*/ };
```

We can also partially specialize `C2` for exactly two `Tuple`s but with potentially different type arguments:

```
template <typename... Ts, typename... Us>
class C2<Tuple<Ts...>, Tuple<Us...>>  // (2) any two Tuples
{ /*...*/ };
```

Furthermore, we can partially specialize `C2` for any `Tuple` followed by zero or more types:

```
template <typename... Ts, typename... Us>
class C2<Tuple<Ts...>, Us...>  // (3) any Tuple followed by 0 or more types
{ /*...*/ };
```

The possibilities are endless; let us show one more partial specialization of `C2` with three template parameter packs that will match two arbitrary `Tuple`s followed by zero or more arguments:

```
template <typename... Ts, typename... Us, typename... Vs>
class C2<Tuple<Ts...>, Tuple<Us...>, Vs...>
    // (4) Specialize C2 for Tuple<Ts...> in the first position,
    // Tuple<Us...> in the second position, followed by zero or more
    // arguments.
{ /*...*/ };
```

Now that we have definitions for the primary template `C2` and four partial specializations of it, let's take a look at a few variable definitions that instantiate `C2`. **Partial ordering of class template specializations**[3] will decide the best match for each instantiation and also deduce the appropriate template parameters:

```
C2<int>                           c2a;  // uses (0), Ts=<int>
C2<Tuple<int>, Tuple<int>>        c2b;  // uses (1), Ts=<int>
C2<Tuple<int, char>, Tuple<char>> c2c;  // uses (2), Ts=<int,char>, Us=<char>
C2<Tuple<int, int>, char>         c2d;  // uses (3), Ts=<int,int>, Us=<char>
C2<Tuple<int>, Tuple<char>, void> c2e;  // uses (4), Ts=<int>, Us=<char>, Vs=<void>
```

Notice how partial ordering chooses (2) instead of (4) for the definition of `c2c`, although both (2) and (4) are a match; a matching non-variadic template is always a better match than one involving deduction of a parameter pack.

Even in a partial specialization, if a template argument of the specialization is a pack expansion, it must be in the last position:

```
template <typename... Ts>
class C2<Ts..., int>;
    // Error, template argument int can't follow pack expansion Ts....

template <typename... Ts>
class C2<Tuple<Ts>..., int>;
    // Error, template argument int can't follow pack expansion Tuple<Ts>....

template <typename... Ts>
class C2<Tuple<Ts...>, int>;
    // OK, pack expansion Ts... is inside another template.
```

Parameter packs are a terse and very flexible placeholder for defining partial specialization — a "zero or more types fit here" wildcard. The primary class template does not even need to be variadic. Consider, for example, a non-variadic class template, `Map`, fashioned after the `std::map` class template:

[3] **?**, section 14.5.5.2, "Partial ordering of class template specializations," [temp.class.order], pp. 339–340.

```
template <typename Key, typename Value> class Map;  // similar to std::map
```

We would then want to partially specialize another non-variadic class template, `C3`, for all maps, regardless of keys and values:

```
template <typename T> class C3;  // (1) primary declaration; C3 not variadic

template <typename K, typename V>
class C3<Map<K, V>>;
    // (2a) Specialize C3 for all Maps, C++03 style.
```

Variadics offer a terser, more flexible alternative:

```
template <typename... Ts>
class C3<Map<Ts...>>;
    // (2b) Specialize C3 for all Maps, variadic style.
    // Note: Map works with pack expansion even though it's not variadic.
```

The most important advantage of (2b) over (2a) is flexibility. In maintenance, `Map` may acquire additional template parameters, such as a predicate and an allocator. This requires surgery on `C3`'s specialization (2a), whereas (2b) will continue to work unchanged because `Map<Ts...>` will accommodate any additional template parameters `Map` may have.

An application must use either the non-variadic (2a) or the variadic (2b) partial specialization but not both. If both are present, (2a) is always preferred because an exact match is always more specialized than one that deduces a parameter pack.

ariadic-alias-templates

Variadic alias templates

Alias templates (since C++11) are a new way to associate a name with a family of types without needing to define forwarding glue code. For full details on the topic, see Section 1.1."**??**" on page **??**. Here, we focus on the applicability of template parameter packs to alias templates.

Consider, for example, the `Tuple` artifact — briefly discussed in Description — that can store an arbitrary number of objects of heterogeneous types:

```
template <typename... Ts> class Tuple;  // declare Tuple
```

Suppose we want to build a simple abstraction on top of `Tuple` — a "named tuple" that has an `std::string` as its first element, followed by anything a `Tuple` may store:

```
#include <string>  // std::string

template <typename... Ts>
using NamedTuple = Tuple<std::string, Ts...>;
    // introduce alias for Tuple of std::string and anything
```

In general, alias templates take template parameter packs following the same rules as primary class template declaration: An alias template may be defined to take at most one template parameter pack in the last position. Alias templates do not support specialization or partial specialization.

ic-function-templates

Variadic function templates

The simplest example of a variadic function template accepts a template parameter pack in its **template parameters list** but does not use any of its template parameters in its **parameter declaration**:

```
template <typename... Ts>
int f0a();     // does not use Ts in parameter list

template <typename... Ts>
int f0b(int);  // uses int but not Ts in parameter list

template <typename T, class... Ts>
int f0c();     // does not use T or Ts in parameter list
```

The only way to call the functions shown in the code snippet above is by passing them template argument lists explicitly:

```
int a1 = f0a<int, char, int>();  // Ts=<int, char, int>
int a2 = f0b<double, void>(42);  // Ts=<double, void>
int a3 = f0c<int, void, int>();  // T=int, Ts=<void, int>
int e1 = f0a();                  // Error, cannot deduce Ts
int e2 = f0b(42);                // Error, cannot deduce Ts
int e3 = f0c();                  // Error, cannot deduce T and Ts
```

The notation `f0a<int, char, int>()` means to explicitly instantiate template function `f0a` with the specified type arguments and to call that instantiation.

Invoking any of the instantiations shown above will, of course, require that the corresponding definition of the function template exists somewhere within the program:

```
template <typename...> void f0a() { /*...*/ }  // variadic template definition
```

This definition will typically be part of or reside alongside its declaration in the same header or source file, but see Section 2.1."**??**" on page **??**.

Such functions are rarely encountered in practice; most of the time a template function would use its template parameters in the function parameter list. They have only pedagogical value; as the joke goes, the keys were lost in the living room but we're looking for them in the hallway because the light is better. To take a look in the living room, let's now consider a different kind of variadic function template — one that not only accepts an arbitrary number of template arguments, but also accepts an arbitrary number of *function* arguments working in tandem with the template arguments.

ction-parameter-packs

Function parameter packs

The syntax for declaring a variadic function template that accepts an arbitrary number of function arguments makes two distinct uses of the ellipsis (`...`) token. The first use is to introduce the template parameter pack `Ts`, as already shown. Then, to make the argument list of a function template variadic, we introduce a function parameter pack by placing the `...` to the left of the function parameter name:

```
template <typename... Ts>  // template parameter pack Ts
```

```
int f1a(Ts... values);     // function parameter pack values
   // f1a is a variadic function template taking an arbitrary sequence of
   // function arguments by value (explanation follows), each independently of
   // arbitrary heterogeneous type.
```

A function parameter pack is a function parameter that accepts zero or more function arguments. Syntactically, a function parameter pack is similar to a regular function parameter declaration, with two distinctions:

- The type in the declaration contains at least one template parameter pack
- The ellipsis `...` is inserted right before the function parameter name, if present, or replaces the parameter name, if absent

Therefore, the declaration of `f1a` above has a template parameter pack `Ts` and a function parameter pack `values`. The function parameter declaration `Ts... values` indicates that `values` may accept zero or more arguments of various types, all by value. Replacing the parameter declaration with `const Ts&... values` would result in pass by `const` reference. Just as with non-variadic function templates, variadic template parameter lists are permitted to include any legal combination of **qualifiers** (`const` and `volatile`) and **declarator operators** (i.e. pointer `*`, reference `&`, forwarding reference `&&`, and array `[]`):

```
template <typename... Ts> void f1b(Ts&...);
   // accepts any number and types of arguments by reference

template <typename... Ts> void f1c(const Ts&...);
   // accepts any number and types of arguments by reference to const

template <typename... Ts> void f1d(Ts* const*...);
   // accepts any number and types of arguments by pointer to const
   // pointer to nonconst object

template <typename... Ts> void f1e(Ts&&...);
   // accepts any number and types of arguments by forwarding reference

template <typename... Ts> void f1f(const volatile Ts*&...);
   // accepts any number and types of arguments by reference to pointer to
   // const volatile objects
```

To best understand the syntax of variadic template declarations, it is important to distinguish the distinct — and indeed complementary — roles of the two occurrences of the `...` token. First, as discussed in *Template parameter packs* on page 32, `typename... Ts` introduces a template parameter pack called `Ts` that matches an arbitrary sequence of types. The second use of `...` is, instead, a pack expansion that transforms the pattern — whether it's `Ts...`, `const Ts&...`, `Ts* const*...` etc, — into a comma-separated parameter list for the function, where the C++ type of each successive parameter is determined by the corresponding type in `Ts`. The resulting construct is a function parameter pack.

Conceptually, a single variadic template function declaration can be thought of as a multitude of similar declarations with zero, one,… parameters fashioned after the variadic declaration:

```
template <typename... Ts> void f1c(const Ts&...);
    // variadic, any number of arguments, any types, all by const &

void f1c();
    // pseudo-equivalent for variadic f1c called with 0 arguments

template <typename T0> void f1c(const T0&);
    // pseudo-equivalent for variadic f1c called with 1 argument

template <typename T0, typename T1>
void f1c(const T0&, const T1&);
    // pseudo-equivalent for variadic f1c called with 2 arguments

template <typename T0, typename T1, typename T2>
void f1c(const T0&, const T1&, const T2&);
    // pseudo-equivalent for variadic f1c called with 3 arguments

/* ... and so on ad infinitum ... */
```

A good intuitive model would be that the pack expansion in the function parameter list is like an "elastic" list of parameter declarations that expands or shrinks appropriately. Note that the types in a parameter pack may all be different from one another, but their qualifiers (**const** and/or **volatile**) and declarator operators (i.e. pointer `*`, reference `&`, *rvalue* reference `&&`, and array `[]`) are the same.

The name of the pack may be missing from the declaration, which, combined with the parameter declaration syntax coming all the way from C, does allow for some obscure constructs: [**AUs: The below example is not valid C++, though clang will accept it with a warning when not being pedantic. I suggest finding an alternative example that actually works or excising these two paragraphs as fluff.**]

```
template<typename... Ts> void fpf(Ts (*...)(int));
    // mystery declaration
```

To read such a declaration, add a name, keeping in mind that the `...` in a function parameter pack always comes immediately to the left of where the name would be:

```
template <typename... Ts> void fpf(Ts (*...pFunctions)(int));
    // Aha, the function parameter pack is pFunctions!
    // The function fpf takes zero or more pointers to functions taking one
    // int, each returning some arbitrary type.
```

A variadic function template may take additional template parameters as well as additional function parameters:

```
template <typename... Ts> void f2a(int, Ts...);
    // one int followed by zero or more arbitrary arguments by const &

template <typename T, typename... Ts> void f2b(T, const Ts&...);
    // first by value, zero or more by const &

template <typename T, typename U, typename... Ts> void f2c(T, const U&, Ts...);
```

```
    // first by value, second by const &, zero or more by value
```

There are restrictions on such declarations; see *The Rule of Greedy Matching* on page 25 and *The Rule of Fair Matching* on page 28.

Note that inside the function parameters declaration, `...` can be used only in conjunction with a template parameter pack. Attempting to use `...` where there's no parameter pack to expand could lead to inadvertent use of the old C-style variadics:

```
template <typename T, typename... Ts>
void good(T, Ts...);                    // variadic template

template <typename T, typename... Ts>
void oops(T, T...);                     // old C-style variadic
```

Such a mistake may be caused by a simple typo in the declaration of `oops`, leading to a variety of puzzling compile-time or link-time errors; see *Potential Pitfalls — Accidental use of C-style ellipsis* on page 73.

It is possible to use the parameter pack name (e.g., `Ts`) as a parameter to a user-defined type:

```
template <typename> struct S1;  // declaration only

template <typename... Ts>       // parameter pack named Ts
int fs1(S1<Ts>...);
    // **Pack expansion** for explicit instantiation of S1 accepts any number of
    // independent explicit instantiations of S1 by value.
```

The parameter pack name `Ts` acts as though it were a separate type parameter for each function argument, thereby allowing a different instantiation for `S1` at each parameter position. To invoke `fs1` on arguments of user-defined type `S1`, however, `S1` will need to be a **complete type** — i.e., its definition must precede the point of invocation of the function in the current translation unit:

```
int s1a = fs1(S1<int>());             // Error, S1 declared but not defined

template <typename T>
struct S1 { /*...*/ };                // introduce definition for S1

int s1b = fs1();                       // Ts is the empty pack.
int s1c = fs1(S1<const char*>());      // Ts=<const char*>
int s1d = fs1(S1<int>(), S1<bool>());  // Ts=<int,bool>
```

More complex setups are possible as well. For example, we can write a variadic function template that operates on instantiations of user-defined-type templates that take two independent type parameters:

```
template <typename, typename>
struct S2;                  // two-parameter class template declaration

template <typename... Ts>   // parameter pack named Ts
int fs2(S2<Ts, Ts>...);
    // The function fs2 takes by value any number of explicit instantiations
```

```
    // of S2 as long as they use the same type in both positions.
```

Calls to `fs2` work only if we supply instantiations of `S2` having the same type for both template parameters:

```
template <typename, typename> struct S2 { /*...*/ };  // S2's definition

int s2a = fs2();                                      // OK
int s2b = fs2(S2<char, char>());                      // OK
int s2c = fs2(S2<int, int>(), S2<bool, bool>());      // OK
int s2d = fs2(S2<char, int>());                       // Error
int s2e = fs2(S2<char, const char>());                // Error
```

The problem with the last two calls above is that the instantiations `S2<char, int>` and `S2<char, const char>` violate the requirement that the two types in the instantiation of `S2` are identical, so there is no way to provide or deduce some `Ts` in a way that would make the call work.

adic-member-functions

Variadic member functions

Member functions may be variadic in two orthogonal ways: The class they are part of may be variadic, and they may be variadic themselves. The simplest case features a variadic member function of a non-template class:

```
struct S3                                  // non-variadic non-template class
{
    template <typename... Ts> int f(Ts...);  // OK
};

int s3 = S3().f(1, "abc");                 // Ts=<int, const char*>
```

Expectedly, a non-variadic **class template** may declare variadic member functions as well:

```
template <typename T>
struct S4                                          // class template
{
    template <typename... Ts> int f1(Ts...);          // OK
    template <typename... Ts> int f2(T, const Ts&...);  // OK
};

int s3b = S4<int>().f1(1, false, true);            // Ts=<int, bool, bool>
int s3c = S4<int>().f2(1, false, true);            // Ts=<bool, bool>
```

A **variadic class template** may define regular **member functions**, member functions that take their own template parameters, and **variadic member function templates**:

```
template <typename... Ts>
struct S5
{
    int f1();       // non-template member function of variadic class

    template <typename T>
```

```
    int f2(T);      // template member function of variadic class

    template <typename... Us>
    int f3(Us...);  // variadic template member function of variadic class
};

int s5a = S5<int, char>().f1();
    // Ts=<int, char>

int s5b = S5<char, int>().f2(2.2);
    // Ts=<int, char>, T=double

int s5c = S5<int, char>().f3(1, 2.2);
    // Ts=<int, char>, Us=<int, double>
```

Although in a sense all member functions of a variadic class template (e.g., `S5`, above) are variadic, the class's template parameter pack (e.g., `Ts`, above) is fixed at the time the class is instantiated. The only truly variadic function is `f3` because it takes its own template parameter pack, `Us`.

A member function of a variadic class may use the template parameter pack of the class in which it is defined:

```
template <typename... Ts>
struct S6                   // variadic class template
{
    int f1(const Ts&...);  // OK, not truly variadic; Ts is fixed

    template <typename T>
    int f2(T, Ts...);      // OK, also not truly variadic; Ts is fixed

    template <typename... Us>
    int f3(Ts..., Us...);  // OK, variadic template member function
};

int s6a = S6<int, char>().f1(1, 'a');
    // Ts=<int, char>

int s6b = S6<char, int>().f2(true, 'b', 2);
    // Ts=<int, char>, T=bool

int s6c = S6<int, char>().f3(1, 2.2, "asd", 123.456);
    // Ts=<int, char>, Us=<const char*, int, double>
```

Notice how, in the initialization of `s6c`, the type arguments `Ts` of class template `S6` must be chosen explicitly, whereas the variadic template `f3` need not have `Us` specified because they are deduced from the argument types. This takes us to the important topic of **template argument deduction**.

te-argument-deduction

Template argument deduction

A popular feature of C++ ever since templates were incorporated into the language pre-standardization has been the ability of function templates to determine their template arguments from the types of arguments provided. Consider, for example, designing and using a function, `print`, that outputs its argument to the console. In most cases, just letting `print` deduce its template argument from the function argument's type has brevity, correctness, and efficiency advantages:

```
#include <string>  // std::string

template <typename T> void print(const T& value);  // prints x to stdout

void testPrint0(const std::string& s)
{
    print<const char*>("Hi");
        // verbose:   specifies template argument *and* function argument
        // redundant: function argument's type *is* the template argument

    print<int>(3.14);
        // Error-prone: Mismatch may cause incorrect results (prints 3).

    print<std::string>("Oops");
        // inefficient: may incur additional expensive implicit conversions

    print("Hi");
    print(3.14);
    print(s);
        // all good: let print deduce template argument from function argument
}
```

What makes the shorter form of the call work is the magic of C++ template argument deduction, which, unsurprisingly, works with variadic template parameters as well. A detailed description of all rules for template argument deduction, including those inherited from C++03, is outside the scope of this book.[4] Here, we focus on the additions to the rules brought about by variadic function templates.

With that in mind, imagine we set out to redesign the API of `print` (in the code example above) to output any number of arguments to the console:

```
#include <string>  // std::string

template <typename... Ts>               // template parameter pack Ts
void print(const Ts&... values);        // prints each of values to stdout

void testPrint1(const std::string& s)  // arbitrary function
{
```

[4] Please insert a cite to Scott Meyers' "Effective Modern C++" (which is already in your bib file), with the addition: Chapter 1: "Deducing Types", pages 9–35. [**AUs: We have several editions of the Meyers book in our bib; which one did you intend? Response: meyers15b and meyers15 are the same book, which this should be referring to.**]

```
    print<const char*, int, std::string>("Hi", 3.14, "Oops");
        // verbose:     specifies template arguments *and* function arguments
        // Redundant:   Function arguments' types *are* the template arguments.
        // Error-prone: Mismatch may cause incorrect results (prints 3).
        // inefficient: may incur additional expensive implicit conversions

    print("Hi", 3.14, s);
        // All good: Deduce template arguments from function arguments.
}
```

The compiler will independently deduce each type in `Ts` from the respective types of the function's argument `values`:

```
void testPrint2()  // arbitrary function
{
    print();               // OK, Ts=<>
    print(42, true);       // OK, Ts=<int, bool>
    print(42.2, "hi", 5);  // OK, Ts=<double, const char*, int>
}
```

As shown in the first line of `test2` (above), a template parameter pack may be specified in its entirety when instantiating a function template:

```
void testPrint3()  // arbitrary function
{
    print<>();                    // OK, exact match
    print<int, bool>(42, true);   // OK, exact match
    print<int, bool>('a', 'b');   // OK, arguments convertible to parameters
}
```

Interestingly, we can specify only the first few types of the template parameter pack and let the others be deduced, mixing together **explicit template argument specification** and template argument deduction:

```
void testPrint4()  // arbitrary function
{
    print<>(5, 'a');                 // OK, Ts=<int, char>
    print<unsigned int>(42, true);   // OK, Ts=<unsigned int, bool>
    print<int, int>('a', "ab", 1);   // Error, cannot deduce Ts
}
```

Such a mix of explicit and implicit may be interesting but is not new; it has been the case for function templates since C++ was first standardized. The new element here is that we get to specify a fragment of a template parameter pack.

In the general case, a function may mix template parameter packs with other template parameters in a variety of ways. It may also mix function parameter packs with other function parameters in a variety of ways.

The Rule of Greedy Matching

rule-of-greedy-matching

The Rule of Greedy Matching for template parameter packs (but not function parameter packs) states that once a template parameter pack starts matching one explicitly specified

template argument, it also matches all template arguments following it. There's no way to tell a template parameter pack it matched enough. Template parameter packs are *greedy*.

Consequently, there is no syntactic way to explicitly specify any template argument following one that matches a template parameter pack; the first pack gobbles all remaining arguments up. Put succinctly: all **template parameters** following a template parameter pack can *never* be explicitly specified as template arguments.

Notice that the rule applies only once a pack starts matching, i.e., it has matched at least one item; indeed there are a few legitimate cases in which a parameter list makes no match at all, which we'll discuss soon.

Using the Rule of Greedy Matching allows us to navigate with relative ease a variety of combinations of pack and nonpack template parameters.

In the simplest and overwhelmingly most frequently encountered situation, the function takes one template parameter pack and one function parameter pack, both in the last position:

```
template <typename T1, typename T2, typename... Ts>  // ... in last position
int f1(T1, T2, const Ts&...);                        // ... in last position
```

In such cases, the Rule of Greedy Matching doesn't need to kick in because there are no parameters following a pack. Template argument deduction can be used for all of `T1`, `T2`, and `Ts` or for only for a subset. In the simplest case, no template parameters are specified and template argument deduction is used for all:

```
int x1a = f1(42, 2.2, 'a', true);
    // T1=int, T2=double, Ts=<char, bool> (all deduced)
```

Explicitly specified template arguments, if any, will match template parameters in the order in which they are declared. Therefore, if we specify one type, it binds to `T1`:

```
int x1b = f1<double>(42, 2.2, 'a', true);
    // T1=double (explicitly), T2=double (deduced), Ts=<char, bool> (deduced)
```

If we specify two types, they will bind to `T1` and `T2` in that order:

```
int x1c = f1<double, char>(42, 65, "abc", true);
    // T1=double (explicitly), T2=char (explicitly),
    // Ts=<const char*, bool> (deduced)
```

Note how, in the two examples above, we also have implicit conversions going on, i.e., `42` is converted to **`double`** and `65` is converted to **`char`**. Furthermore, a call may specify `T1`, `T2`, and the entirety of `Ts`:

```
int x1d = f1<const char*, char, bool, double>("abc", 'a', true, 42U);
    // T1=const char* (explicitly), T2=char (explicitly),
    // Ts=<bool, unsigned> (explicitly)
```

Last but not least, as mentioned, a call may specify `T1`, `T2`, and only the first few types in `Ts`:

```
int x1e = f1<const char*, char, bool, double>("abc", 'a', true, 42, 'a', 0);
    // T1=const char* (explicitly), T2=char (explicitly),
    // Ts=<bool, int, char, int> (first two explicit, others deduced)
```

Let us now look at a function having a template parameter pack not in the last position. However, in the argument list, the function parameter pack is still in the last position:

```
template <typename T, typename... Ts, typename U>  // ... not last
int f2(T, U, const Ts&...);
```

In such cases, by the Rule of Greedy Matching, there is no way to specify `U` explicitly, so the only way to call `f1` is to let `U` be deduced:

```
int x2a = f2(1, 2);
    // T=int (deduced), U=int (deduced), Ts=<> (by arity)

int x2b = f2<long>(1, 2, 3);
    // T=long (explicit), U=int (deduced), Ts=<int> (deduced)
```

The first template argument passed to `f2`, if any, is matched by `T`. Note that inferring the empty template parameter pack in the initialization of `x2a` does not involve deduction; the empty length is inferred by the arity of the call. In contrast, in the initialization of `x2b`, a pack of length 1 is deduced.

Any subsequent template arguments will be matched *en masse* by `Ts` in concordance with the Rule of Greedy Matching:

```
int x2c = f2<long, double, char>(1, 2, 3, 4);
    // T1=long (explicit), T2=int (deduced), Ts=<double, char> (explicit)

int x2d = f2<int, double>(1, 2, 3.0, "four");
    // T1=int (explicit), T2=int (deduced), Ts=<double, const char*> (partially specified)

int x2e = f2<int, char, double>(1, 3.0, 'a');
    // Error, no viable function for T1=int and Ts=<char, double>
```

In all cases, `T2` must be deduced for the call to `f2` to go through.

Another way to make a template parameter work even if it's positioned after a parameter pack is by assigning it a default argument:

```
template <typename... Ts, typename T = int>
T f3(Ts... values);
```

Due to the way `f3` is defined, there is no way to either deduce or specify `T`, so it will always be **`int`**:

```
int x3a = f3("one", 2);
    // Ts=<const char*, int> (deduced), T=int (default)

int x3b = f3<const char*>("one", 2);
    // Ts=<const char*, int> (partially deduced), T=int (default)

int x3c = f3<const char*, int>("one", 2);
    // Ts=<const char*, int> (explicit), T=int (default)
```

rule-of-fair-matching

The Rule of Fair Matching

To further explore the varied ways in which parameter packs may interact with the rest of C++, let's take a look at a function that has a type following a template parameter pack and also a value following a function parameter pack:

```
template <typename... Ts, typename T>
int f4(Ts... values, T value);
```

Here, a new rule applies, the Rule of Fair Matching, which is to a good extent the converse of the Rule of Greedy Matching. When a function parameter pack (e.g., `values` in the code snippet above) is *not* at the end of a function's parameter list, its corresponding type parameter pack (e.g., `Ts` above) *cannot be deduced* ever.

This rule makes the function parameter pack `values` fair because function parameters following a function parameter pack have a chance to match function arguments.

Let's take a look at how the rule applies to `f4`. In calls with one argument, `Ts` is not deduced, so it forcibly matches the empty list, and `T` matches the type of the argument:

```
int x4a = f4(123);     // Ts=<> (forced non-deduced), T=int
int x4b = f4('a');     // Ts=<> (forced non-deduced), T=char
int x4c = f4('a', 2);  // Error, cannot deduce Ts
```

Calls with more than one argument can be made if and only if we provide `Ts` to the compiler explicitly:

```
int x4d = f4<int, char>(1, '2', "three");
    // Ts=<int, char> (explicit), T=const char*
```

Incidentally, for `f4` there is no way to specify `T` explicitly because of the Rule of Greedy Matching:

```
int x4e = f4<int, char, const char*>(1, '2', "three");
    // Error, Ts=<int, char, const char*> (explicit), no argument for T value
```

Note that the two rules may work simultaneously on the same function call; they do not compete because they apply in different places; the Rule of Greedy Matching applies to template parameter packs, and the Rule of Fair Matching applies to function parameter packs.

Let's now consider declaring a function with two consecutive parameter packs and see how the rules work together in calls:

```
template <typename... Ts, typename... Us>  // two template parameter packs
int f5(Ts... ts, Us... us);                // two function parameter packs
```

By the Rule of Greedy Matching, we cannot specify `Us` explicitly so `us` will rely on deduction exclusively. By the Rule of Fair Matching, `Ts` cannot be deduced. First, let's analyze a call with no template arguments:

```
int x5a = f5(1);
    // Ts=<> (forcibly), Us=<int>

int x5b = f5(1, '2');
    // Ts=<> (forcibly), Us=<int, char>
```

```
int x5c = f5(1, '2', "three");
    // Ts=<> (forcibly), Us=<int, char, const char*>
```

Whenever a call specifies no **template arguments**, `Ts` cannot be deduced so it can at best match the empty list. That leaves all of the arguments to `us`, and deduction will work as expected for `Us`. This right-to-left matching may surprise at first and requires close reading of different parts of the C++ Standard but is easy to explain by using the two rules.

Let's now issue a call that does specify template arguments:

```
int x5d = f5<int, char>(1, '2');
    // Ts=<int, char> (explicitly), Us=<const char*>

int x5e = f5<int, char>(1, '2', "three");
    // Ts=<int, char> (explicitly), Us=<const char*>

int x5f = f5<int, char>(1, '2', "three", 4.0);
    // Ts=<int, char> (explicitly), Us=<const char*, double>
```

By the Rule of Greedy Matching, all explicit template arguments go to `Ts`, and, by the Rule of Fair Matching, there's no deduction for `Ts`, so, even before looking at the function arguments, we know that `Ts` is exactly `<int, char>`. From here on, it's easy: The first two arguments go to `ts`, and all others, if any, will go to `us`.

plate-argument-matching

Corner cases of function template argument matching

There are cases in which a template function could be written that can never be called, whether with explicit template parameters or by relying on template argument deduction:

```
template <typename... Ts, typename T>
void odd1(Ts... values);
```

By the Rule of Greedy Matching applied to `Ts`, `T` can never be specified explicitly. Moreover, `T` cannot be deduced either because it's not part of the function's parameter list. Hence, `odd1` is impossible to call. According to standard C++, such function declarations are **ill formed, no diagnostic required (IFNDR)**. Current compilers do allow such functions to be defined without a warning. However, any conforming compiler will disallow any attempt to call such an ill-fated function.

Another scenario is one whereby a variadic function can be instantiated, but one or more of its parameter packs must always have length zero:

```
template <typename... Ts, typename... Us, class T>
int odd2(Ts..., Us..., T);  // specious
```

Any attempt to call `odd2` by relying on only template argument deduction will force both `Ts` and `Us` to the empty list because, by the Rule of Fair Matching, neither `Ts` nor `Us` can benefit from template argument deduction. So calls with two, three, or more arguments fail:

```
int x2a = odd2(1, 2.5);           // Error, Ts=<>, Us=<>, too many arguments
int x2a = odd2(1, 2.5, "three");  // Error, Ts=<>, Us=<>, too many arguments
```

However, there seem to be ways to invoke `odd2`, at least on contemporary compilers. First, calls using deduction with exactly one argument will merrily go:

```
int x2c = odd2(42);  // Ts=<>, Us=<>, T=42
```

Better yet, functions that pass an explicit argument list for `Ts` also seem to work:

```
int x2d = odd2<int, double>(1, 2.0, "three");
    // Ts=<int, double>, Us=<>, T=const char*
```

The call above passes `Ts` explicitly as `<int, double>`. Then, as always, `Us` is forced to the empty list, and `T` is deduced as `const char*` for the last argument. That way, the call goes through!

Or does it? Alas, the declaration of `odd2` is **IFNDR**. By the C++ Standard, if all valid instantiations of a variadic function template require a specific template parameter pack argument to be empty, the declaration is IFNDR. Although such a rule sounds arbitrary, it does have a good motivation: If all possible calls to `odd2` require `Us` to be the empty list, why is `Us` present in the first place? Such code is more indicative of a bug than of a meaningful intent. Also, diagnosing such cases may be quite difficult in the general case, so no diagnostic is required. As it turns out, today's compilers do not issue such a diagnostic, so the onus is on the template writer to make sure the code does what it's supposed to do.

A simple fix for `odd2` is to just eliminate the `Us` template parameter, in which case `odd2` has the same signature as `f4` discussed in *The Rule of Fair Matching* on page 28. Another possibility to "legalize" `odd2` is to drop the nonpack parameter, in which case it has the same signature as `f5` in the same section.

A function that has three parameter packs in a row would also be IFNDR:

```
template <typename... Ts, typename... Us, typename... Vs>
void odd3(Ts..., Us..., Vs...);  // impossible to instantiate
```

The reason `odd3` cannot work is purely bureaucratic: Neither `Ts` nor `Us` may benefit from deduction, and then there is no way to specify `Us` explicitly because `Ts` is greedy. Consequently, `Us` is forced to be always of length zero.

It may seem there is no way to define a function template taking more than two parameter packs. However, recall that deducing variadic function template parameters from the (object) arguments passed to the function uses the full power of C++'s template argument deduction. Defining functions with any number of template parameter packs is entirely possible provided the parameter packs are themselves part of other template instantiations:

```
template <typename...> class Vct { };  // variadic class template definition

template <typename T, typename... Ts, typename... Us, typename... Vs>
int fvct(const T&, Vct<Ts...>, Vct<Us...>, Vct<Vs...>);
    // The first parameter matches any type by const&, followed by three
    // not necessarily related instantiations of Vct.
```

The function template `fvct` takes a fixed number of parameters (four), the last three of which are independent instantiations of a variadic class template `Vct`. For each of them, `fvct` takes a template parameter pack that it passes along to `Vct`. For each call to `fvct`, template argument deduction figures out whether the call is viable and also binds `Ts`, `Us`, and `Vs` to the packs that make the call work:

```
int x = fvct(5, Vct<>(), Vct<char, int, long>(), Vct<bool>());
    // OK, T=int, Ts=<>, Us=<char, int, long>, Vs=<bool>
```

For each argument in the call above, the compiler matches the type of the argument with the pattern required by the template parameter; the matching process deduces the types that would make the match work. The general algorithm for matching a concrete type against a type pattern is called **unification**.[5]

emplate-parameter-packs

Non-type template parameter packs

Defining a variadic template that take arguments other than types is also possible. Just as C++03 template parameters can be *types*, *values*, or *templates*, so can template parameter packs. We used **type template parameter packs** up until now to simplify exposition, but non-type template parameter packs apply to class templates and function templates as well.

To clarify terminology, the C++ Standard refers to template parameters that accept values as **non-type template parameters** and to template parameters that accept names of templates as **template template parameters** (to be discussed in subsection *Template parameter packs* on page 32).

Non-type template parameter packs are defined analogously with non-type template parameters:

```
template <int...>
class Ci { };           // variadic int-parameter class template

Ci<>              ci0;  // OK, zero ints given
Ci<1>             ci1;  // OK, one int argument
Ci<2, 3>          ci2;  // OK, two int arguments
Ci<true, 'a', 3u> ci3;  // OK, converts to three int arguments
Ci<4.0>           ci4;  // Error, floating-point literal 4.0 is ineligible.
```

The type specifier of the template non-type parameter pack need not only be **int**; the same rules as for **non-type parameters** apply, restricting the types of non-type parameters to

- integral
- enumerated
- pointer to function or object
- *lvalue* reference to function or object
- pointer to member
- `std::nullptr_t`

These types are the value types allowed for C++03 non-type template parameters. In short, value parameter packs obey the same restrictions as the C++03 non-type template parameters:

```
#include <cstddef>  // std::nullptr_t
#include <string>  // std::string
```

[5] ?

```
enum Enum { /*...*/ };                  // arbitrary enumerated type
class AClass { /*...*/ };               // arbitrary class type

template <long... ls>          class Cl;   // OK, integral
template <bool... bs>          class Cb;   // OK, integral
template <char... cs>          class Cc;   // OK, integral
template <Enum... es>          class Ce;   // OK, enumerated
template <int&... is>          class Cri;  // OK, reference
template <std::string*... ss>  class Csp;  // OK, pointer
template <void (*... fs)(int)> class Cf;   // OK, pointer to function
template <int AClass::*... >   class Cpm;  // OK, pointer to member
template <std::nullptr_t>     class Cnl;  // OK, std::nullptr_t

template <Ci<>... cis>         class Cu;   // Error, cannot be user-defined
template <double... ds>        class Cd;   // Error, cannot be floating-point
template <float... fs>         class Cf;   // Error, cannot be floating-point
```

In the example above, the declaration of class template `Cu` is not permitted because `Ci<>` is a *user-defined* type, whereas the declarations of both `Cd` and `Cf` are disallowed because **`double`** and **`float`** are floating-point types.[6]

Template parameter packs

plate-parameter-packs

Template parameter packs are the variadic generalization of C++03's template template parameters. Classes and functions may be designed to take template parameter packs as parameters in addition to **type parameter packs**.

A template *template* parameter is a template parameter that names an argument that is itself a *template*. Consider, for example, two arbitrary class templates, `A1` and `A2`:

```
template <typename> class A1 {/* ...*/ };  // some arbitrary class template
template <typename> class A2 {/* ...*/ };  // another arbitrary class template
```

Now suppose that we have a *class* template, (e.g., `C1`) that takes, as its template parameter, a *class template*:

```
template <template<typename> class X>  // X is a template template parameter.
struct C1 : X<int>, X<double>          // inherit X<int> and X<double>
{ };
```

We can now create instances of class `C1` where the bases are obtained by instantiating whatever argument we pass to `C1` with **`int`** and **`double`**, respectively:

```
C1<A1> c1a;  // inherits A1<int> and A1<double>
C1<A2> c1b;  // inherits A2<int> and A2<double>
```

In the code snippet above, `X` is a template *template* parameter that takes one *type* parameter. The template classes `A1` and `A2` match `X` because each of these templates, in turn, takes one *type* parameter as well.

[6] C++20 does allow *floating point* non-type template parameters, which enable the definition and use of non-type parameter packs using **`float`**, **`double`**, or **`long double`** (making, e.g., the final entry — `Ci<4.0> ci4;` — in the previous example above valid as well).

If an instantiation is attempted with a class template that does not take the same number of template parameters, the instantiation fails, even when there is no doubt as to what the intent might be:

```
template <typename, typename = char>
class A3 { /*...*/ };  // OK, two-parameter template, with second one defaulted

C1<A3> c1c;  // Error, parameters of A3 are different from parameters of X.
```

Although `A3` could be instantiated with a single template argument (due to its second template parameter having a default argument) and `A3<int>` is valid, `C1<A3>` will not compile. The compiler complains that `A3` has two parameters, whereas `X` has a single parameter.

The same limitation is at work when the argument to `C1` is a variadic template:

```
template <typename...>
class A4 { /*...*/ };    // OK, arbitrary variadic template

C1<A4> c1d;  // Error, parameters of A4 are different from parameters of X.
```

Let's define a class `C2` that is a variadic generalization of `C1` in such a way that it allows instantiation with `A3` and `A4`:

```
template <template<typename...> class X>  // template template parameter
struct C2 : X<int>, X<double>             // inherit X<int>, X<double>
{ };
```

Note that, although one may use **`typename`** and **`class`** interchangeably inside the template parameters of `X` above, one must always must use **`class`** for `X` itself.[7]

The difference between `C2` and `C1` is literally one token: `C2` adds one `...` after **`typename`** *inside* the parameter list of `X`. That one token makes all the difference: by using it, `C2` signals to the compiler that it accepts templates with any number of parameters, be they fixed in number, defaulted, or variadic. In particular, it works just fine with `A1` and `A2` as well as `A3` and `A4`:

```
C2<A1> c2a;
    // inherits A1<int> and A1<double> in that order
C2<A2> c2b;
    // inherits A2<int> and A2<double> in that order
C2<A3> c2c;
    // C2<A3> inherits A3<int, char> and A3<double, char> in that order.
    // char is the default argument for A3's second type parameter.
C2<A4> c2d;
    // inherits A4<int> and A4<double> in that order
```

When `C3` instantiates `A3<int>` or `A3<double>`, `A3`'s default argument for its second parameter kicks in. `A4` will be instantiated with the type parameter pack `<int>` and separately with the type parameter pack `<double>`.

In contrast with template template parameters that require exact matching with their arguments, variadic template template parameters specifying **`typename...`** work in a more

[7] Since C++17, code may use either **`typename`** or **`class`** for template template parameters and for template parameter packs.

"do what I mean" manner by matching templates with default arguments and variadic templates. Applications may need either style of matching depending on context.

There is an orthogonal other direction in which we can generalize `C2`: We can define a template, `C3`, that accepts zero or more template template arguments:

```
template <template<typename> class... Xs>  // Xs is a template parameter pack.
struct C3
    : Xs<int>...     // inherit X0<int>, X1<int>, ...
    , Xs<double>...  // inherit X0<double>, X1<double>, ...
{ };
```

We get to instantiate `C3` with zero or more template classes, each of which takes exactly one type parameter:

```
C3<>           c3a;
    // no base classes at all; Xs=<>, all base specifiers vanish

C3<A1>         c3b;
    // inherits A1<int> and A1<double> in that order

C3<A1, A2>     c3c;
    // inherits A1<int>, A2<int>, A1<double>, and A2<double> in that order

C3<A2, A1>     c3d;
    // inherits A2<int>, A1<int>, A2<double>, and A1<double> in that order

C3<A1, A2, A1> c3e;
    // Error, cannot inherit A1<int> and A1<double> twice
```

Note that `C3`, just like `C1`, cannot be instantiated with `A3`, again due to mismatched template parameters:

```
C3<A3> c3f;
    // Error, parameters of A3 are different from parameters of Xs.
C3<A4> c3g;
    // Error, parameters of A4 are different from parameters of Xs.
```

If we want to make instantiations with `A3` and `A4` possible, we can combine the two generalization directions by placing the `...` inside the `Xs` parameter declaration *and* on `Xs` as well:

```
template <template<typename...> class... Xs>  // two sets of ...
struct C4                                     // most flexible
    : Xs<int>...                              // X0<int>, X0<int>, ...
    , Xs<double>...                           // X1<double>, X1<double>, ...
{ };
```

`C4` combines the characteristics of `C2` and `C3`. It accepts zero or more arguments, which in turn are accepted in a "do what I mean" manner:

```
C4<>           c4a;
    // no base classes at all; Xs=<>, all base specifiers vanish
```

```
C4<A1>         c4b;
    // inherits A1<int> and A1<double> in that order

C4<A1, A2>     c4c;
    // inherits A1<int>, A2<int>, A1<double>, and A2<double> in that order

C4<A3, A1>     c4d;
    // inherits A3<int>, A1<int>, A3<double>, and A1<double> in that order

C4<A1, A4, A2> c4e;
    // may pass any subset of A1, A2, A3, A4, in any order
```

Quite a few other templates match `C4`'s template argument, even though they will fail to instantiate with a single parameter:

```
template <typename, typename> class A5 { /*...*/ };
template <typename, typename, typename...> class A6 { /*...*/ };

C4<A5> err1;  // Error, matches, but A5<int> and A5<double> are invalid.
C4<A6> err2;  // Error, matches, but A6<int> and A6<double> are invalid.
```

Templates that do not take type template parameters, however, will *not* match `C4` or, for that matter, any of `C1` through `C4`:

```
template <typename, int>
class A7 { /*...*/ };          // one type parameter and one non-type parameter
template <typename, int = 42>
class A8 { /*...*/ };          // same, but non-type parameter defaulted
template <template<typename> class>
class A9 { /*...*/ };          // template template parameter

C4<A7> err3;  // Error, second template argument of C4 has a different kind
C4<A8> err4;  // Error, second template argument of C4 has a different kind
C4<A9> err5;  // Error, first template argument of C4 has a different kind
```

In short, class templates that do not take specifically *types* as their template parameters cannot be used in instantiations of `C4`.

In the general case, template parameter packs may appear together with other template parameters and follow the rules and restrictions discussed so far in the context of type parameter packs. Any number of subtle but nevertheless perfectly meaningful matching cases may be defined involving combinations of fixed and variadic template parameters. Suppose, for example, we want to define a class `C5` that accepts a template that takes *at least* two parameters and possibly more:

```
template <template<typename, typename, typename...> class X>
class C5
    // class template definition having one template template parameter
    // for which the template template accepts two or more type
    // arguments
{ /*...*/ };
```

```
// A few templates that match C5.

template <typename, typename> class B1;
template <typename, typename, typename = int> class B2;
template <typename, typename = int, typename = int> class B3;
template <typename, typename, typename...> class B4;

C5<B1> c5a;  // OK
C5<B2> c5b;  // OK
C5<B3> c5c;  // OK
```

However, templates that don't have two fixed type parameters in the first two position will not match `C5`:

```
template <typename> class B5;
template <int, typename, typename...> class B6;
template <typename, typename, int, typename...> class B7;
template <typename, typename...> class B8;

C5<B5> c5d;  // Error, argument mismatch
C5<B6> c5e;  // Error, argument mismatch
C5<B7> c5f;  // Error, argument mismatch
C5<B8> c5g;  // Error, argument mismatch
```

To match `C5`'s template parameter, a template must take types for its first two parameters, followed by zero or more type parameters (with default values or not). `B5` does not match because it takes only one parameter. `B7` does not match because it takes an **`int`** in the first position, as opposed to a type, as required. `B7` fails to match because it takes an **`int`** in the third position instead of a type. Finally, `B7` is not a match because its second argument is not fixed.

To summarize our findings, template parameter packs generalize template template parameters in two distinct, orthogonal ways.

- The `...` at the template template parameter level allows zero or more template arguments to match.
- The `...` inside the parameter list of the template template parameter allows loose matching of templates with default arguments or variadic; however, type vs. value vs. template parameters are still checked, as in the example involving `A7`, `A68`, and `A9`.

`pack-expansion`

Pack expansion

Now that we have a solid command of using parameter packs in type and function declarations, it is time to explore how to use parameter packs in function implementations.

As briefly mentioned in *Variadic class templates* on page 9, parameter packs belong to a *kind* distinct from any other C++ entity; they are not types, values, template names, and so on. As far as learning parameter packs goes, they cannot be related to existing entities, so they may as well come from another language with its own syntax and semantics.

Literally the only way to use a parameter pack is to make it part of a so-called pack expansion. A pack expansion consists of a fragment of code (a "pattern") followed by `...`. The code fragment must contain at least a pack name; otherwise, it does not qualify as an expansion. Exactly what patterns are allowed depends on the place where the expansion occurs. Depending on context, the pattern is syntactically a simple identifier, a parameter declaration, an expression, or a type.

The dual use of `...` — to both introduce a pack and expand it — may seem confusing at first, but distinguishing between the two uses of `...` is easy: When the ellipsis occurs *before* a previously undefined identifier, it is meant to introduce it as the name of a parameter pack; in all other cases, the ellipsis is an expansion operator.

We've already seen pack expansion at work in *Variadic function templates* on page 18. In a variadic function template declaration, the function argument list (e.g., `Ts... values`) is an expansion resulting in zero or more by-value parameters.

To introduce a simple example of pack expansion in an actual computation, recall the `add` example in *Description* on page 4, in which a variadic function adds together an arbitrary number of integers. The full implementation shown below uses **`double`** instead of **`int`** for better usability. The important part is the expansion that keeps the computation going:

```
double add()                                    // base case, no arguments
{
    return 0;                                   // neutral element for addition
}

template <typename T, typename... Ts>           // recursive case
double add(const T& lhs, const Ts&... rest)     // accepts 1 or more arguments
{
    return lhs + add(rest...);                  // recurse expanding rest
}
```

The key to understanding how `add` works is to model the expansion `rest...` as a comma-separated list of **`double`**s, always invoking `add` with fewer arguments than it currently received; when `rest` becomes the empty pack, the expansion expands to nothing and `add()` is called, which terminates the recursion by providing the neutral value 0.

Consider the call:

```
int x1 = add(1.5, 2.5, 3.5);
```

Computation proceeds in a typical recursive manner.

1. The top-level call goes to the variadic function template `add`.
2. `add` binds `T` to **`double`** and `Ts` to `<double, double>`.
3. The expression `add(rest...)` expands into the recursive call `add(2.5, 3.5)`.
4. That call binds `T` to **`double`** and `Ts` to `<double>`.
5. The second expansion of `add(rest...)` leads to the recursive call `add(3.5)`.
6. Finally, the last expansion will recurse to `add()`, which returns 0.

The result is constructed as the recursion unwinds.[8]

From an efficiency perspective, it should be noted that `add` is not recursive in a traditional Computer Science sense. It does not call itself. Each seemingly recursive call is a call to an entirely new function with a different arity generated from the same template. After inlining and other common optimizations, the code is as efficient as writing the statements by hand.

A few rules apply to all parameter-pack expansions irrespective of their kind or the context in which they are used.

First, any pattern must contain at least one parameter pack. Therefore, expansions such as `C<int...>` or `f(5...)` are invalid. This requirement can be problematic in function declarations because a typo in a name may switch the meaning of `...` from parameter pack expansion to an old-style C variadic function declaration; see *Potential Pitfalls — Accidental use of C-style ellipsis* on page 73.

A single pack expansion may contain two or more parameter packs. In that case, expansion of multiple parameter packs within the same expansion is always carried *in lockstep*, i.e., all packs are expanded concomitantly. For example, consider a function modeled after a slightly modified `add`, and note how, instead of expanding just `rest`, we expand a slightly more complex pattern in which `rest` appears twice:

```
template <typename... Ts>
double add2(const Ts&... xs)   // accepts 0 or more arguments
{
    return add((xs * xs)...);  // expand xs * xs in call to add
}
```

This time `add2` calls `add` with the expansion `(xs * xs)...`, not just `xs...`, so we're looking at two parameter packs (the two instances of `xs`) expanded in lockstep. The call `add2(1.5, 2.5, 3.5)` will forward to `add(1.5 * 1.5, 2.5 * 2.5, 3.5 * 3.5)`, revealing that `add2` computes the sum of squares of its arguments.

The parentheses around the pattern in the expansion `(xs * xs)...` are not needed; the expansion comprehends the full expression to the left of `...`, so the call could have been written as `add(xs * xs...)`.

For expansion to work, all parameter packs in one expansion must be of the same length; otherwise, an error will be diagnosed at compile time. Suppose we define a function `f1` that takes two parameter packs and passes an expansion involving both to `add2`:

```
template <typename... Ts, typename... Us>
double f1(const Ts&... ts, const Us&... us)
{
    return add2((ts - us)...);
}
```

Here, the only valid invocations are those with equal number of arguments for `ts` and `us`:

```
double x1a = f1<double, double>(1, 2, 3, 4);
    // OK, Ts=<double, double> (explicit), Us=<int, int> (deduced)

double x1b = f1<double, double>(1, 2, 3, 4, 5);
```

[8] C++17 adds fold expressions — `return lhs + ... + rest;` — that allow a more succinct implementation of `add`.

```
// Error, parameter packs ts and us have different lengths.
// Ts=<double, double> (explicit), Us=<int, int, int> (deduced)
```

Every form of pack expansion produces a comma-separated list consisting of copies of the pattern with the parameter packs suitably expanded. Expansion is semantic, not textual — that is, the compiler is "smarter" than a typical text-oriented preprocessor. If the pack expansion is within a larger comma-separated list and the parameter packs being expanded have zero elements, the commas surrounding the expansion are adjusted appropriately to avoid syntax errors. For example, if `ts` is empty, the expansion `add(1, ts..., 2)` becomes `add(1, 2)`, not `add(1, , 2)`.

Expansion constructs may be nested. Each nesting level must operate on at least one parameter pack:

```
template <typename... Ts>
double f2(const Ts&... ts)
{
    return add2((add2(ts...) + ts)...);
}
```

Expansion always proceeds "inside out," with the innermost expansion carried first. In the call `f2(2, 3)`, the expression returned after the first, innermost `ts` is expanded is `add2((add2(2, 3) + ts)...)`. The second expansion results in the considerably heftier expression `add2(add2(2, 3) + 2, add2(2, 3) + 3)`. Nested expansions grow in a combinatorial manner, so care is to be exercised.

What forms of `pattern...` are allowed, and where in the source code is the construct allowed? Parameter pack expansions are defined for a variety of well-defined contexts, each of which is separately described in the following subsections. Not all potentially useful contexts are supported, however; see **??** — *Limitations on expansion contexts* on page 75[**AUs: There is no section called "Expansion at the statement level is not permitted." Response: limitations on expansions contexts is the appropriate reference**] Expansion is allowed in:

- a function parameter pack
- a function call argument list or a braced initializer list
- a template argument list
- a base specifier list
- a member initializer list
- a lambda capture list
- an alignment specifier
- an attribute list
- a **`sizeof...`** expression
- a template parameter pack that is a pack expansion

nction-parameter-pack

Expansion in a function parameter pack

An ellipsis in a function declaration's parameter list expands to a list of parameter declarations. The pattern being expanded is a parameter declaration.

We discussed this expansion at length in the preceding subsections, so let's quickly recap by means of a few examples:

```
template<typename... Ts>
void f1(Ts...);                // expands to T0, T1, T2,...

template<typename... Ts>
void f2(const Ts&...);         // const T0&, const T1&,...

template<typename... Ts>
void f3(const C<Ts>&...);      // Complex use, e.g., in templates, is allowed.

template<typename... Ts, typename... Us>
void f4(const C<Ts, Us>&...);  // const C<T0, U0>&, const C<T1, U1>&,...
```

aced-initializer-list

Expansion in a function call argument list or a braced initializer list

Expansion may occur in the argument list of a function call or in an initializer list — either parenthesized or *brace-enclosed* (see Section 2.1."??" on page ??). In these cases, the pattern is the *largest expression or braced initialization list* to the left of the ellipsis.

This expansion is the only one that expands into expressions (all others are declarative), so in a way it is the most important because it relates directly to runtime work getting done.

Let's look at a few examples of expansion. Suppose we have a library that comprises three variadic function templates, `f`, `g`, and `h`, and an ordinary class, `C`, having a variadic *value* constructor:

```
template <typename... Ts> int f(Ts...);  // variadic function template
template <typename... Ts> int g(Ts...);  //    "        "        "
template <typename... Ts> int h(Ts...);  //    "        "        "

struct C                                 // ordinary class
{
    template <typename... Ts> C(Ts...);  // variadic value constructor
};
```

Let's now suppose that we another variadic function template, `client1`, that intends to make use of this library by expanding its own parameter pack, `xs`, in various contexts:

```
template <typename... Ts>
void client1(Ts... xs)
{
    f(xs...);                  // (1) f(x0, x1, ...);
    f(C(xs...));               // (2) f(C(x0, x1, ...));
    f(C(xs)...);               // (3) f(C(x0), C(x1), ...);
    f(3.14, xs + 1 ...);       // (4) f(3.14, x0 + 1, x1 + 1, ...);
    f(3.14, xs * 2. ...);      // (5) f(3.14, x0 * 2., x1 * 2., ...);
}
```

In comments, we informally denote with `x0`, `x1`, `...`, the elements of the pack `xs`. The first call, `f(xs...)`, illustrates the simplest expansion; the pattern being expanded, `xs`, is simply the name of a function parameter pack. The expansion results in a comma-separated list of the arguments received by `client1`.

The other examples illustrate a few subtleties. Examples (2) and (3) show how the positioning of `...` determines how expansion unfolds. In (2), the expansion is carried inside the call to `C`'s constructor, so `f` is called with exactly one object of type `C`. In (3), the ellipsis occurs outside the constructor call, so `f` gets called with zero or more `C` objects, each constructed with exactly one argument.

Examples (4) and (5) show how whitespace may be important. If the space before `...` were missing, the C++ parser would encounter `1...` and `2....`, both of which are incorrect floating-point literals.

Let's now look at a few more complex examples in another function, `client2`:

```
template <typename... Ts>
void client2(Ts... xs)
{
    f("hi", xs + xs..., 3.14);  // (6) f(x0 + x0, x1 + x1, ..., 3.14);
    f(const_cast<Ts&>(xs)...);  // (7) f(const_cast<T1&>(x0), ...);
    f(g(xs)..., h(xs...));      // (8) f(g(x0), g(x1), ..., h(x0, x1, ...))
    C object1(*xs...);          // (9) C::C(*x0, *x1, ...)
    C object2{*xs...};          // (10) C::C(*x0, *x1, ...)
    int a[] = { xs..., 0 };     // (11) int[] a = { x0, x1, ..., 0 }
}
```

Examples (6) and (7) feature simultaneous expansion of two packs. In (6), `xs` is expanded twice. In (7), template parameter pack `Ts` and value parameter pack `xs` are both expanded. In all cases of simultaneous expansion, the two or more packs are expanded in lockstep, i.e., the first element in `Ts` together with the first element in `vs` form the first element in the expansion, and so on. Attempting to expand packs of unequal lengths results in a compile-time error. Example (6) also shows how an expansion may be somewhere in the middle of a function's argument list.

Example (8) shows two sequential expansions that look similar but are quite different. The two expansions are independent and can be analyzed separately. In `g(xs)...`, the pattern being expanded is `g(xs)` and results in the list `g(x0)`, `g(x1)`, `...`. In contrast, in the expansion `h(xs...)`, the expansion is carried inside the call to `h` — `h(x0, x1, ...)`.

Example (9) shows that expansion is allowed inside special functions as well. The expansion `C(*xs...)` results in `C`'s constructor called with `*x0`, `*x1`, and so on.

Last but not least, examples (10) and (11) illustrates expansion inside a braced-initialization list. Example (10) calls the same constructor as (9), and example (11) initializes an array of `int` with the content of `xs` followed by a `0`.

For each of these examples to compile, the code resulting from pack expansions needs to pass the usual semantic checks; for example, the `int` array initialization in (11) would fail to compile if `xs` contained a value with a type not convertible to `int`. As is always the case with templates, some instantiations work, whereas some don't.

In the examples above, the functions involved are all variadic. However, a function doesn't need to be variadic — or template for that matter — to be called with an expansion. The

expansion is carried in the function call expression, and then the usual lookup rules apply to decide whether the call is valid:

```
int f1(int a, double b);  // simple function with two parameters

int f2();                 // no parameters
int f2(int a, double b);  // overload with two parameters

template <typename... Ts>
void client3(const Ts&... xs)
{
    f1(xs...);  // Works if and only if xs has exactly two elements
                // convertible to int and double, respectively.

    f2(xs...);  // Works if and only if xs is empty or has exactly two elements
                // convertible to int and double, respectively.
}
```

emplate-argument-list

Expansion in a template argument list

We have already encountered pack expansion in the argument list of a template instantiation; if `C` is a class template and `Ts` is a template parameter pack, `C<Ts...>` instantiates `C` with the contents of `Ts`. There is no need for the template `C` to accept a variadic list of parameters. The resulting expansion must be appropriate for the template instantiated.

For example, suppose we define a variadic class `Lexicon` that uses its parameter pack in an instantiation of `std::map`:

```
#include <map>     // std::map
#include <string>  // std::string

template <typename... Ts>              // template parameter pack
class Lexicon                          // variadic class template
{
    std::map<Ts...> d_data;            // Use Ts to instantiate std::map.
    // ...
};

Lexicon<std::string, int> c1;          // (1) OK, std::map<std::string, int>
Lexicon<int> c2;                       // (2) Error, std::map<int> invalid
Lexicon<int, long, std::less<int>> c3; // (3) OK
Lexicon<long, int, 42> c4;             // (4) Error, 42 instead of functor
```

Given that `Lexicon` forwards all of its template arguments to `std::map`, the only viable template arguments for `Lexicon` are those that would be viable for `std::map` as well. Therefore, (1) is valid because it instantiates `std::map<std::string, int>`. As usual, `std::map`'s default arguments for its third and fourth parameters kick in; the pack expansion does not affect default template arguments. To wit, (3) passes three template arguments to `std::map` and leaves the last one (the allocator) to be filled with the default. Instantiations (2) and (4) of `Lexicon` are not valid because they would attempt to instantiate `std::map` with incompatible template arguments.

n-a-base-specifier-list

Expansion in a base specifier list

Suppose we set out to define a variadic template, `MB1`, that inherits all of its template arguments. This scheme is useful in applying design patterns such as Visitor[9] or Observer[10]. To enable such designs, expansion is allowed in a base specifier list. The pattern under expansion is a base specifier, which includes an optional protection specifier (**`public`**, **`protected`**, or **`private`**) and an optional **`virtual`** base specifier. Let us define `MB1` to inherit all of its template arguments using **`public`** inheritance:

```
template <typename... Ts>  // template parameter pack
class MB1 : public Ts...   // multibase class, publicly inherit each of Ts
{
    // ...
};
```

The pattern **`public`** `Ts...` expands into **`public`** `T0`, **`public`** `T1`, and so on for each type in `Ts`. All bases resulting from the expansion have the same protection level. If `Ts` is empty, `MB1<>` has no base class.

```
class S1 { /*...*/ };  // arbitrary class
class S2 { /*...*/ };  // arbitrary class

MB1<>           m1a;     // OK, no base class at all
MB1<S1>         m1b;     // OK, instantiate with S1 as only base
MB1<S1, S2>     m1c;     // OK, instantiate with S1 and S2 as bases.
MB1<S1, S2, S1> m1d;     // Error, cannot inherit S1 twice
MB1<S1, int>    m1e;     // Error, cannot inherit from scalar type int
```

After expansion, the usual rules and restrictions apply; a class cannot inherit another one twice and cannot inherit types such as **`int`**.

Other bases may be specified before and/or after the pack. The other bases may specify other protection levels (and if they don't, the default protection level applies):

```
template <typename... Ts>      // parameter pack Ts
class MB2
    : virtual private S1       // S1 virtual private base
    , public Ts...            // inherit each of Ts publicly
{ /*...*/ };

template <typename... Ts>      // parameter pack Ts
class MB3
    : public S1               // S1 public base
    , virtual protected Ts...  // each type in Ts a virtual protected base
    , S2                      // S2 private base (uses default protection)
{ /*...*/ };

MB2<>   m2a;                  // (1) virtual private base S1
MB2<S2> m2b;                  // (2) virtual private base S1, public base S2
MB3<>   m3a;                  // public base S1, private base S2
```

[9] **?**, Chapter 10, "Visitor," pp.!235–262
[10] **?**, Chapter 5, section "Observer," pp. 293–303

```
MB3<S2> m3b;                    // Error, cannot inherit S2 twice
```

Expansions are not limited to simple pack names. The general pattern allowed in a base specifier list is that of a full-fledged base specifier. For example, the parameter pack can be used to instantiate another template:

```
template <typename T>
class Act                       // arbitrary class template
{ /*...*/ };

template <typename... Ts>       // template parameter pack Ts
class MB4
    : public Act<Ts>...         // bases Act<T0>, Act<T1>, ...
{ /*...*/ };

MB4<>                     m4a;  // no base class
MB4<int, double>          m4b;  // bases Act<int>, Act<double>
MB4<MB4<int>, int>        m4c;  // bases Act<MB4<int>>, Act<int>
```

Arbitrarily complex instantiations can be specified in an base specifier pack expansion, which opens to opportunity for a variety of expansion patterns. Depending on where the ellipsis is placed, different expansion patterns can be created.

```
template <typename... T>
class Avct                  // arbitrary variadic class template
{ /*...*/ };

template <typename... Ts>  // template parameter pack Ts
class MB5                  // multibase class example
: public Avct<Ts>...       // zero or more: Avct<T0>, Avct<T1>, ...
, private Avct<Ts...>      // exactly one: Avct<T0, T1, ...>
{ /*...*/ };
```

Although the two expansions featured above are similar in syntax, they are semantically very different. First, **public** `Avct<Ts>...` expands into multiple bases for `MB5`: **public** `Avct<T0>`, **public** `Avct<T1>`, and so on. The second expansion is completely different; in fact, it's not even an expansion in a base specifier list. Its context is a template's argument list; see *Expansion in a template argument list* on page 42. The result of that expansion is a single class `Avct<T0, T1, ...>` that is an additional private base of `MB5`:

```
MB5<int, double> mb5a;
    // inherits publicly Avct<int>, Avct<double>
    // inherits privately Avct<int, double>

MB5<Avct<int, char>, double> mb5b;
    // inherits publicly Avct<Avct<int, char>>, Avct<double>,
    // inherits privately Avct<Avct<int, char>, double>

MB5<int> mb5c;
    // Error, cannot inherit Avct<int> twice
```

member-initializer-list

Expansion in a member initializer list

This feature is a "forced move" of sorts. Allowing variadic bases on a class naturally creates the necessity of being able to initialize those bases accordingly. Consequently, parameter pack expansion is allowed in a base initializer list. The pattern is a base initializer, i.e., the name of a base followed by a parenthesized list of arguments for its constructor:

```
template <typename... Ts>  // template parameter pack
struct S1 : Ts...          // publicly inherit every type in the pack
{
    S1() : Ts(0)...        // (1) call constructor with 0 for each base
    { /*...*/ }

    S1(int x) : Ts(x)...   // (2) call constructor with x for each base
    { /*...*/ }

    S1(const S1& rhs) : Ts(static_cast<const Ts&>(rhs))...
        // OK, call the copy constructor for each base
    { /*...*/ }
};
```

The default constructor (1) calls all bases' constructors, passing `0` to each. The second constructor (2) passes its one **`int`** argument to each base. The last constructor (3) of `S` implements the copy constructor and is rather interesting because it expands in turn to a call to the copy constructor for each member, passing it the result of a **`static_cast`** (which, in fact, is implicit) to the appropriate base type. Similar syntax can be used for defining the move constructor (see Section 2.1."**??**" on page **??**) and other constructors.

Let's embark on a more complex example. Suppose we want to define a class `S2` with a constructor that accepts any arguments and forwards them to all of its base classes. To do so, that constructor itself needs to be variadic with a distinct parameter pack:

```
template <typename... Ts>      // template parameter pack
struct S2 : Ts...              // publicly inherit every type in the pack
{
    template <typename... Us>  // variadic constructor
    S2(const Us&... xs)        // accepts any number of arguments by const &
    : Ts(xs...)...             // (!) forwards them to each base constructor
    { }
};
```

The code above has at least one ellipse on every significant line, and all are needed. Let's take a closer look.

First off, class `S2` inherits all of its template arguments. It has no exact knowledge about the types it would be instantiated with and, out of consideration for flexibility, defines a variadic constructor that forwards any number of arguments from the caller into each of its base classes. That constructor, therefore, is itself variadic with a separate template parameter pack, `Us`, and a corresponding argument pack, `xs`, that accepts arguments by reference to **`const`**. The key line, commented with `(!)` in the code, performs two expansions, which proceed inside out. First, `xs...` expands into the list of arguments passed to `S2`'s constructor, leading to the pattern `Ts(x0, x1, ...)`. In turn, that pattern itself gets expanded by

the outer `...` into a base initialization list: `T0(x0, x1, ...), T1(x0, x1, ...), ...`.

What if pass by reference to **`const`** is too constraining, and we would want to define a more general constructor that can forward modifiable values as well? In that case, we need to use forwarding references (see Section 2.1."??" on page ??) and the Standard Library function `std::forward`:

```
#include <utility>  // std::forward

template <typename... Ts>      // template parameter pack
struct S3 : Ts...              // publicly inherit every type in the pack
{
    template <typename... Us>  // variadic constructor
    S3(Us&&... xs)             // arguments by forwarding reference
    : Ts(std::forward<Us>(xs)...)...
                               // forwards them to each base constructor
    { }
};
```

`S3`'s variadic constructor makes use of **forwarding references**, which automatically adapt to the type of the arguments passed. The library function `std::forward` ensures that each argument is forwarded with the appropriate type, qualifier, and *lvalue*ness to the constructors of each base class. The expansion process is similar to that in `S2`'s constructor previously discussed, with the additional detail that `std::forward<Us>(xs)...` expands in lockstep `Us` and `xs`.

Expansion in a lambda capture list

a-lambda-capture-list

A C++ lambda expression, introduced with C++11 (see Section 2.1."??" on page ??, is an unnamed function object. A lambda can store internally some of the local variables present at the point of creation by a mechanism known as **lambda capture**. This important capability distinguishes lambdas from simple functions. Let's put it to use in defining a `tracer` lambda that is able to print a given variable to the console:

```
#include <iostream>  // std::cout, std::endl

template <typename T>  // single-parameter template
auto tracer(T& x)      // returns a lambda that, when invoked, prints x
{
    auto result = [&x]() { std::cout << x << std::endl; };
        // [&x] means the function object captures x by reference.
        // tracer must use auto to initialize the function object.
    return result;
}
```

Lambdas have a type chosen by the compiler, so we need **`auto`** to pass lambda objects around; see Section 2.1."??" on page ??.

This may seem like a lot if you're new to lambdas, in which case reading locationc."??" on page ?? along with Section 3.2."??" on page ?? before continuing may be in order. The underlying idea is simple: A call such as `tracer(x)` saves a reference to `x` in a function object, which it then returns. Subsequent calls to that function object output the current

value of `x`. It's important to save `x` by reference (hence the `&` in the capture), lest the lambda store `x` by value and uninterestingly print the same thing on each call.

Let's see `tracer` at work, tracing some variable in a function:

```
int process(int x)           // uses the trace facility
{
    auto trace = tracer(x);  // Initialize trace to follow x.
    trace();                 // prints current value of x
    ++x;                     // change the value of x
    trace();                 // prints current (changed) value of x
    return x;                // Return x back to the caller.
}

int x0 = process(42);         // prints 42, then 43, and initializes x0 to 43
```

What is the connection with variadics? Variadics are all about generalization, which applies here as well. Suppose we now set out to trace several variables at once. Instead of a one-argument function, `tracer` would then need to be variadic. Also, crucially, the lambda returned needs to store references to all arguments received in order to print them later. That means an expansion must be allowed inside a lambda capture list.

First, let's assume a function, `print`, exists that prints any number of arguments to the console (*Use Cases — Generic variadic functions* on page 52 features an implementation of `print`):

```
template <typename... Ts>     // variadic function
void print(const Ts&... xs);  // prints each argument to std::cout in turn
```

The definition of `multitracer` uses `print` in a lambda with variadic capture:

```
template <typename... Ts>    // variadic template
auto multitracer(Ts&... xs)  // returns a lambda that, when invoked, prints xs
{
    auto result = [&xs...]() { print(xs...); };
        // [&xs...] means capture all of xs by reference.
        // result stores one reference for each argument.
    return result;
}
```

The entire API is enabled by the ability to expand `xs` inside the capture list. Expanding with `[&xs...]` captures by reference, whereas `[xs...]` captures the pack by value. Inside the lambda, `print` expands the pack as usual with `x...`.

Expansions in captures can be combined with all other captures:

```
template <typename... Ts>    // variadic template
auto test(Ts&... xs)         // for illustration purposes
{
    int a = 0, b = 0;
    auto f1 = [&a, xs...]() { /*...*/ };
        // Capture a by reference and all of xs by value.

    auto f2 = [xs..., &a]() { /*...*/ };
```

```
        // same capture as f1

    auto f3 = [a, &xs..., &b]() { /*...*/ };
        // Capture a by value, all of xs by reference, and b by reference.

    auto f4 = [&, xs...]() { /*...*/ };
        // Capture all of xs by value and everything else by reference.

    auto f5 = [=, &xs..., &a]() { /*...*/ };
        // Capture a and all of xs by reference, and everything else by value.
}
```

The pattern must be that of a simple capture; complex capture patterns are not allowed as of C++14.[11]

n-alignment-specifier

Expansion in an alignment specifier

The **alignas** specifier is a feature new to C++11 that allows specifying the alignment requirement of a type or object; see Section 2.1."**??**" on page **??**:

```
alignas(8)      float x2;  // Align x1 at an address multiple of 8.
alignas(double) float x1;  // Align x2 with the same alignment as a double.
```

Pack expansion inside the **alignas** specifier is allowed. The meaning in the presence of the pack is to specify the largest alignment of all types in the pack:

```
template <typename... Ts>    // variadic template
int test1(Ts... xs)        // for illustration purposes
{
    struct alignas(Ts...) S { };
        // Align S at the largest alignment of all types in Ts.
        // If Ts is empty, the alignas directive ignored.

    alignas(Ts...) float x1;
        // Align x1 at the largest alignment of all types in Ts.
        // If Ts is empty, the alignas directive ignored.

    alignas(Ts...) alignas(float) float x2;
        // Align x2 at the largest alignment of double and all types in Ts.

    alignas(float) alignas(Ts...) float x3;
        // same alignment as x2; order does not matter

    return 0;
}
```

As always with **alignas**, requesting an alignment smaller than the minimum alignment required by the declaration is an error:

[11] C++20 introduces pack expansion in lambda initialization captures (see **?**) that allows capturing variadics with a syntax such as `[...us = vs]` or `[...us = std::move(vs)]`.

```
int a1 = test1();          // OK, Ts empty, all alignas(Ts...) ignored
int a2 = test1('a', 1.0);  // OK, align everything as a double.
int a3 = test1('a');       // Error (most systems), can't align x1 to 1 byte
```

An idiom that avoids such errors is to use two **alignas** for a given declaration, one of which is the natural alignment of the declaration. This is the idiom followed by the declarations of `x2` and `x3` in the definition of function template `test1`.

Using handwritten, comma-separated lists inside an **alignas** specifier is, however, not allowed. Expansion *outside* the specifier is also disallowed:

```
template <typename... Ts>    // variadic template
void test2(Ts... xs)          // for illustration purposes
{
    alignas(Ts)... float x4;
        // Error, cannot expand outside the alignas specifier

    alignas(double, Ts...) float x5; // Error, syntax not allowed
    alignas(Ts..., double) float x6; // Error, syntax not allowed
    alignas(long, double) float x7;  // Error, syntax not allowed
}
```

To conclude, pack expansion in an **alignas** specifier allows choosing without contortions the largest alignment of a parameter pack. Combining two or more **alignas** specifiers facilitates a simple idiom for avoiding errors and corner cases.

Expansion in an attribute list

on-in-an-attribute-list

Attributes, introduced with C++11, are a mechanism for adding built-in or user-defined information about declarations; see Section 1.1."**??**" on page **??**.

Attributes are added to declaration using the syntax `[[attribute]]`. For example, `[[noreturn]]` is a standard attribute indicating that a function will not return:

```
[[noreturn]] void abort();  // Once called, it won't return.
```

Two or more attributes can be applied to a declaration either independently or as a comma-separated list inside the square brackets:

```
[[noreturn]] [[deprecated]] void finish();  // won't return, also deprecated
[[deprecated, noreturn]]    void finish();  // same
```

For completeness and future extensibility, pack expansion is allowed inside an attribute specifier as in `[[attribute...]]`. However, this feature is not currently usable with any current attribute, standard or user-defined:

```
template <typename... Ts>
[[Ts()...]] void functionFromTheFuture();
    // NONWORKING CODE
    // Receive a number of types; instantiate them as attributes.
```

The ability to expand packs inside attribute specifiers is reserved for future use and good to keep in mind for future additions to the language.

-sizeof...-expression

Expansion in a `sizeof...` expression

The **`sizeof...`** expression is an oddity in three ways. First, it has nothing to do with classical **`sizeof`** in the sense that **`sizeof...`** does not yield the extent occupied in memory by an object. Second, it is the only parameter pack expansion that does *not* use the (by now familiar) `pack...` syntax. And third, although it is considered an expansion, it does not expand a pack into its constituents.

For any parameter pack `P`, **`sizeof...(P)`** yields to a compile-time constant of type `size_t` equal to the number of elements of `P`:

```
    std::size_t
template <typename... Ts>
std::size_t countArgs(Ts... xs)
{
    std::size_t x1 = sizeof...(Ts);        // x1 is the number of parameters
    std::size_t x2 = sizeof...(xs);        // same value as x1
    static_assert(sizeof...(Ts) >= 0,"");  // sizeof...(Ts) is a constant.
    static_assert(sizeof...(xs) >= 0,"");  // sizeof...(xs) is a constant.
    return sizeof ... (Ts);                // whitespace around ... allowed
}
```

Let's see `countArgs` in action:

```
std::size_t a0 = countArgs();           // initialized to 0
std::size_t a1 = countArgs(42);         // initialized to 1
std::size_t a2 = countArgs("ab", 'c');  // initialized to 2
```

Whitespace is allowed around the `...`, but parentheses are not optional. Also, expansion is disallowed inside **`sizeof`** and **`sizeof...`** alike:

```
template <typename... Ts>
std::size_t nogo(Ts... xs)
{
    std::size_t x1 = sizeof 42;         // OK, same as sizeof(int)
    std::size_t x2 = sizeof... Ts;      // Error, parens required around Ts
    std::size_t x3 = sizeof... xs;      // Error, parens required around xs
    std::size_t x4 = sizeof(Ts...);     // Error, cannot expand inside sizeof
    std::size_t x5 = sizeof...(Ts...);  // Error, cannot expand inside sizeof...
    std::size_t x6 = sizeof(xs...);     // Error, cannot expand inside sizeof
    std::size_t x7 = sizeof...(xs...);  // Error, cannot expand inside sizeof...
}
```

mplate-parameter-list

Expansion inside a template parameter list

Not to be confused with the case discussed in *Expansion in a template argument list* on page 42, pack expansion may occur in a template *parameter* (not argument) list. This is different from all other expansion cases because it involves two distinct parameter packs: a type parameter pack and a non-type parameter pack. To set things up, suppose we define a class template, `C1`, that has a type parameter pack, `Ts`. Inside, we define a secondary class template, `C2`, that does *not* take any type parameters. Instead, it takes a non-type template parameter pack with types derived from `Ts`:

```
template <typename... Ts>  // type parameter pack
struct C1                  // class template
{
    template <Ts... vs>    // non-type parameters (attention: no typename!)
    struct C2 { };         // expansion in C2's parameter list
};
```

Once `C1` is instantiated with some types, the inner class `C2` will accept *values* of the types used in the instantiation of `C1`. For example, if `C1` is instantiated with **`int`** and **`char`**, its inner class template `C2` will accept an **`int`** value and a **`char`** value:

```
C1<int, char>::C2<1, 'a'> x1;       // OK, C2 takes an int and a char.
C1<int, char>::C2<1> x2;            // Error, too few arguments for C2
C1<int, char>::C2<1, 'a', 'b'> x3;  // Error, too many arguments for C2
```

Only instantiations of `C1` that lead to valid declarations of `C2` are allowed. For example, user-defined types are not allowed as non-type template parameters, and consequently `C1` cannot be instantiated with a user-defined type:

```
class AClass { };   // simple user-defined class

C1<int, AClass>::C2<1, AClass()> x1;
    // Error, a non-type template parameter cannot have type AClass.
```

ther-expansion-contexts

No other expansion contexts

Note that what's missing is as important as what's present. Parameter pack expansion is explicitly disallowed in any other context, even if it would make sense syntactically and semantically:

```
template <typename... Ts>
void bumpAll(Ts&... xs)
{
    ++xs...;  // Error, cannot expand xs in an expression-statement context
}
```

Annoyances — Limitations on expansion contexts on page 75[**AUs: There is no subsection called "Expansion at the statement level is not permitted" Response: limitations on expansion contexts is the correct reference**] discusses this context further. Also recall that it is illegal to use pack names anywhere without expanding them, so they don't enjoy first-class status; see *Annoyances — Parameter packs cannot be used unexpanded* on page 76.

n-contexts-and-patterns

Summary of expansion contexts and patterns

To recap, expansion is allowed in only the following places:

Context	Pattern
function parameter pack	parameter declaration
function call argument list or a braced initializer list	function argument
template argument list	template argument
base specifier list	base specifier
member initializer list	base initializer
lambda capture list	capture
alignment specifier	alignment specifier
attribute list	attribute
`sizeof...` expression	identifier
template parameter pack that is a pack expansion	parameter declaration

use-cases-variadic

Use Cases

ic-variadic-functions

Generic variadic functions

A variety of functions of general utility are naturally variadic, either mathematically (`min`, `max`, `sum`) or as a programmer's convenience. Suppose, for example, we want to define a function, `print`, that writes its arguments to `std::cout` in turn followed by a newline[12]:

```
#include <iostream>  // std::cout, std::endl

std::ostream& print()                // parameterless overload
{
    return std::cout << std::endl;   // only advances to next line
}

template <typename T, typename... Ts>             // one or more types
std::ostream& print(const T& x, const Ts&... xs)  // one or more args
{
    std::cout << x;                               // output first argument
    return print(xs...);                          // recurse to print rest
}

void test()
{
    print("Pi is about ", 3.14159265);            // "Pi is about 3.14159"
}
```

The implementation follows a head-and-tail recursion that is typically used for C++ variadic function templates. The first overload of `print` has no parameters and simply outputs a newline to the console. The second overload does the bulk of the work. It takes one or more arguments, prints the first, and recursively calls `print` to print the rest. In the limit, `print` is called with no arguments, and the first definition kicks in, outputting the line terminator and also ending the recursion.

A variadic function's smallest number of allowed arguments does not have to be zero, and it is free to follow many other recursion patterns. For example, suppose we want to define a variadic function `isOneOf` that returns **`true`** if and only if its first argument is

[12] C++20 introduces `std::format`, a facility for general text formatting.

equal to one of the subsequent arguments. Calls to such a function are sensible for two or more arguments:

```
template <typename T1, typename T2>     // normal template function
bool isOneOf(const T1& a, const T2& b)  // two-parameter version
{
  return a == b;
}

template <typename T1, typename T2, typename... Ts>      // two or more
bool isOneOf(const T1& a, const T2& b, const Ts&... xs)  // all by const&
{
  return a == b || isOneOf(a, xs...);                    // compare, recurse
}
```

Again, the implementation uses two definitions in a pseudo-recursive setup but in a slightly different stance. The first definition handles two items and also stops recursion. The second version takes three or more arguments, handles the first two, and issues the recursive call only if the comparison yields **`false`**.

Let's take a look at a few uses of `isOneOf`:

```
#include <string>  // std::string

int a = 42;
bool b1 = isOneOf(a, 1, 42, 4);  // b1 is true.
bool b2 = isOneOf(a, 1, 2, 3);   // b2 is false.
bool b3 = isOneOf(a, 1, "two");  // Error, can't compare int with const char*
std::string s = "Hi";
bool b4 = isOneOf(s, "Hi", "a"); // b4 is true.
```

object-factories

Object factories

Suppose we want to define a generic **factory function** — a function able to create an instance of any given type by calling one of its constructors. Object factories[13,14] allow libraries and applications to centrally control object creation for a variety of reasons: using special memory allocation, tracing and logging, benchmarking, object pooling, late binding, deserialization, interning, and more.

The challenge in defining a generic object factory is that the type to be created (and therefore its constructors) is not known at the time of writing the factory. That's why C++03 object factories typically offer only default object constructions, forcing clients to awkwardly use two-phase initialization, first to create an empty object and then to put it in a meaningful state.

Writing a generic function that can transparently forward calls to another function ("perfect forwarding") has been a long-standing challenge in C++03. An important part of the puzzle is making the forwarding function generic in the number of arguments, which is where variadic templates help in conjunction with forwarding references (see Section 2.1."**??**" on page **??**):

[13] **?**, Chapter 1-5, section "Factory Method," pp. 107–115
[14] **?**, Chapter 8, "Object Factories," pp. 197–218

```
#include <utility>  // std::forward

void log(const char* message);                // logging function

template <typename Product, typename... Ts>   // type to be created and params
Product factory(Ts&&... xs)                   // call by forwarding reference
{
    log("factory(): Creating a new object");  // Do some logging.
    return Product(std::forward<Ts>(xs)...);  // Forward arguments to ctor.
}
```

`Ts&&... xs` introduces `xs`, a function parameter pack that represents zero or more forwarding references. As we know, the construct `std::forward<Ts>(xs)...` is a pack expansion that expands to a comma-separated list `std::forward<T0>(x0), std::forward<T1>(x1),` and so on. The Standard Library function template `std::forward` passes accurate type information from the forwarding references `x0`, `x1`, `...` to `Product`'s constructor.

To use the function, we must always provide the `Product` type explicitly; it is not a function parameter, so it cannot be deduced. The others are at best left to template argument deduction. In the simplest case, `factory` is usable with primitive types:

```
int i1 = factory<int>();              // Initialize i1 to 0.
int i2 = factory<int>(42);            // Initialize i2 to 42.
```

It also works correctly with overloaded constructors:

```
struct Widget
{
    Widget(double);        // constructor taking a double
    Widget(int&, double);  // constructor taking an int& and a double
};

int g = 0;
Widget w1 = factory<Widget>(g, 2.4);   // calls ctor with int& and double
Widget w2 = factory<Widget>(20);       // calls ctor with double
Widget w3 = factory<Widget>(20, 2.0);  // Error, cannot bind rvalue to int&
```

The last line introducing `w3` fails to compile because the *rvalue* `20` cannot convert to the non-`const int&` required by `Widget`'s constructor — an illustration of perfect forwarding doing its job.

Many variations of object factories (e.g., using dynamic allocation, custom memory allocators, and special exceptions treatment) can be built on the skeleton of `factory` shown. In fact, Standard Library factory functions, such as `std::make_shared` and `std::make_unique`, use variadics and perfect forwarding in this same manner.

ooking-function-calls

Hooking function calls

Forwarding is not limited to object construction. We can use it to intercept function calls in a generic manner and add processing such as tracing, logging, and so on. Suppose, for example, writing a function that calls another function and logs any exception it may throw:

```
#include <exception>  // std::exception
```

```
#include <utility>    // std::forward

void log(const char* msg);                    // Log a message.

template <typename Callable, typename... Ts>
auto logExceptions(Callable&& fun, Ts&&... xs)
    -> decltype(fun(std::forward<Ts>(xs)...))
{
    try
    {
        return fun(std::forward<Ts>(xs)...);  // perfect forwarding to fun
    }
    catch (const std::exception& e)
    {
        log(e.what());                        // log exception information
        throw;                                // Rethrow the same exception.
    }
    catch (...)
    {
        log("Nonstandard exception thrown."); // log exception information
        throw;                                // Rethrow the same exception.
    }
}
```

Here, we enlist not only the help of `std::forward` but also that of the **`auto -> decltype`** idiom; see Section 1.1."**??**" on page **??** and Section 1.1."**??**" on page **??**. By using **`auto`** instead of the return type of `logExceptions` and following with `->` and the trailing type **`decltype`**`(fun(std::forward<Ts>(xs)...))`, we state that the return type of `logExceptions` is the same as the type of the call `fun(std::forward<Ts>(xs)...)`, which matches perfectly the expression that the function will actually return.

In case the call to `fun` throws an exception, `logExceptions` catches, logs, and rethrows that exception. So `logExceptions` is entirely transparent other than for logging the passing exceptions. Let's see it in action. First, we define a function, `assumeIntegral`, that is likely to throw an exception:

```
#include <stdexcept>  // std::runtime_error

long assumeIntegral(double d)            // throws if d has a fractional part
{
    long result = static_cast<long>(d);  // Compute the returned value.
    if (result != d)                     // Verify.
        throw std::runtime_error("Integral expected");
    return result;
}
```

To call `assumeIntegral` via `logExceptions`, we just pass it along with its argument:

```
void test()
{
    long a = logExceptions(assumeIntegral, 4.0);  // Initialize a to 4.
```

```
    long b = logExceptions(assumeIntegral, 4.4);  // throws and logs
}
```

Tuples

tuples

A *tuple* or a record is a type that groups together a fixed number of values of unrelated types. The C++03 Standard Library template `std::pair` is a tuple with two elements. The standard library template `std::tuple`, introduced in C++11, implements a tuple with the help of variadic templates. For example, `std::tuple<int, int, float>` holds two `int`s and a `float`.

There are many possible ways to implement a tuple in C++. C++03 implementations typically define a hardcoded limit on the number of values the tuple can hold and use considerable amounts of scaffolding, as described in *Description* on page 4:

```
struct None { };  // empty "tag" used as a special "not used" marker

template <typename T1 = None, typename T2 = None, typename T3 = None,
          typename T4 = None, typename T5 = None, typename T6 = None,
          typename T7 = None, typename T8 = None, typename T9 = None>
class Cpp03Tuple;
    // tuple containing up to 9 data members of arbitrary types
```

Variadics are a key ingredient in a scalable, manageable tuple implementation. We discuss a few possibilities in approaching the core definition of a tuple, with an emphasis on data layout.

The definition of `Tuple1` (in the code snippet below) uses specialization and recursion to accommodate any number of types:

```
    std::size_tstd::stringassertstd::runtime_error
```

```
template <typename... Ts>
class Tuple1;                // (0) incomplete declaration

template <>
class Tuple1<>               // (1) specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple1<T, Ts...>       // (2) specialization for one or more elements
{
    T first;                 // first element
    Tuple1<Ts...> rest;      // all other elements
    // ...
};
```

`Tuple1` uses composition and recursion to create its data layout. As discussed in *Expansion in a base specifier list* on page 43, the expansion `Tuple1<Ts...>` results in `Tuple1<T0, T1, ..., Tn>`. The specialization `Tuple1<>` ends the recursion.

The only awkward detail is that `Tuple1<int>` has member `rest` of type `Tuple1<>`, which is empty but is required to have nonzero size, so it ends up occupying space in the

tuple.[15] This is an important issue if the tuple is to be used at scale.

A solution to avoid this issue is to partially specialize `Tuple1` for one element in addition to the two existing specializations:

```
template <typename T>
class Tuple1<T>                 // (3) specialization for one element
{
    T first;
    // ...
};
```

With this addition, `Tuple1<>` uses the total specialization (1), `Tuple1<int>` uses the partial specialization (3), and all instantiations with two or more types use the partial specialization (2). For example, `Tuple1<int, long, double>` instantiates specialization (2), which uses `Tuple1<long, double>` as a member, which in turn uses the partial specialization (3) for member `rest` of type `Tuple1<double>`.

The disadvantage of the design above is that it requires similar code in the `Tuple1<T>` partial specialization and the general definition, leading to a subtle form of code duplication. This may not seem very problematic, but a good tuple API has a considerable amount of scaffolding; for example, `std::tuple` has 25 member functions.

Let's address `Tuple1`'s problem by using inheritance instead of composition, thus benefitting from an old and well-implemented C++ layout optimization known as the **empty base optimization**. When a base of a class has no state, that base is allowed, under certain circumstances, to occupy no room at all in the derived class. Let's design a `Tuple2` variadic class template that takes advantage of the empty base optimization:

```
template <typename... Ts>
class Tuple2;                   // incomplete declaration

template <>
class Tuple2<>                  // specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple2<T, Ts...>          // specialization for one or more elements
    : public Tuple2<Ts...>      // recurses in inheritance
{
    T first;
    //...
};
```

If we assess the size of `Tuple2<int>` with virtually any contemporary compiler, it is the same as `sizeof(int)`, so the base does not, in fact, add to the size of the complete object. One awkwardness with `Tuple2` is that with most compilers the types specified appear in the memory layout in reverse order; for example, in an object of type `Tuple1<int, int, float, std::string>`, the string would be the first member in the layout, followed by the `float` and then by the two `int`s. (Compilers do have some freedom

[15] On Clang 11.0 and GCC 7.5, `sizeof(T1)` is `8`, twice the size of an `int`.

in defining layout, but most of today's compilers simply place bases first in their order, followed by members in the order of their declarations.)

To ensure a more intuitive layout, let's define `Tuple3` that uses an additional **`struct`** to hold individual elements, which `Tuple3` inherits (by means of the seldom-used **`protected`** inheritance) before recursing:

```
template <typename T>
struct Element3      // element holder
{
    T value;         // no other data or member functions
};

template <typename... Ts>
class Tuple3;        // declaration to introduce the class template

template <>
class Tuple3<>       // specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>  // one or more types
class Tuple3<T, Ts...>                 // one or more elements
    : public Element3<T>               // first in layout
    , public Tuple3<Ts...>             // recurse to complete layout
{ /*...*/ };
```

This is close to what we need, but there is one additional problem to address: The instantiation `Tuple3<int, int>` will attempt to inherit `Element<int>` twice, which is not allowed. One way to address this issue is by passing a so-called cookie to `Element`, an additional template parameter that uniquely tags each `Element` differently. We choose `size_t` for the cookie type:

```
template <typename T, std::size_t cookie>
struct Element4      // Element holder also takes a cookie so the same element
                     // is not inherited twice; cookie is actually not used.
{
    T value;
};
```

The `Tuple4` class instantiates `Element` with a decreasing value of `cookie` for each successive element:

```
template <typename... Ts>
class Tuple4;                          // (0) incomplete declaration

template <>
class Tuple4<>                         // (1) specialization for no elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple4<T, Ts...>                 // (2) one or more elements
    : public Element4<T, sizeof...(Ts)>  // first in layout, count is cookie
```

```
    , public Tuple4<Ts...>              // recurse to complete layout
{ /*...*/ };
```

To see how it all works, consider the instantiation `Tuple4<int, int, char>`, which matches specialization (2) with `T=int` and `Ts=<int, char>`. Consequently `sizeof...(Ts)` — the number of elements in `Ts` — is 2. The specialization first inherits `Element<2, int>` and then `Tuple4<int, char>`. The latter, in turn, also uses specialization (2) with `T=int` and `Ts=<char>`, which inherits `Element<int, 1>` and `Tuple4<char>`. Finally, `Tuple4<char>` inherits `Element<char, 0>` and `Tuple4<>`, which kicks the first specialization into gear to terminate the recursion.

It follows that `Tuple4<int, int, char>` ultimately inherits (in this order) `Element<int, 2>`, `Element<int, 1>`, and `Element<char, 0>`. Most implementations of `std::tuple` are variations of the patterns illustrated by `Tuple1` through `Tuple4`.[16]

If `Element` didn't take a distinct number for each member of the product type, `Tuple4<int, int>` would not work because it would inherit `Element<int>` twice, which is illegal. With `cookie`, the instantiation works because it inherits the distinct types `Element<int, 1>` and `Element<int, 2>`.

The expanded templates for the above code example might look like this invalid but illustrative code:

```
class Tuple4<>
{ /*...*/ };
class Tuple4<char> : public Element<char, 0>, public Tuple4<>
{ /*...*/ };
class Tuple4<int, char> : public Element<int, 1>, public Tuple4<char>
{ /*...*/ };
class Tuple4<int, int, char> : public Element<int, 2>, public Tuple4<int, char>
{ /*...*/ };
```

The complete implementation of a tuple type would contain the usual constructors and assignment operators as well as a *projection function* that takes an index `i` as a compile-time parameter and returns a reference to the `i`th element of the tuple. Let's see how to implement this rather subtle function. To get it done, we first need a helper template that returns the `n`th type in a template parameter pack.

```
template <std::size_t n, typename T, typename... Ts>
struct NthType              // yields nth type in the sequence <T, Ts...>
{
    typedef typename NthType<n - 1, Ts...>::type
        type;               // recurse to smaller n
};

template <typename T, typename... Ts>
struct NthType<0, T, Ts...>  // base case, 0th type in <T, Ts...> is T
```

[16] `libstdc++`, the GNU C++ Standard Library, uses an inheritance-based scheme with increasing indexes (as opposed to `Tuple4` in which `cookie` values are decreasing). `libc++`, the LLVM Standard Library that ships with Clang, does not use inheritance for `std::tuple`, but its state implementation uses inheritance in conjunction with an increasing integral sequence. Microsoft's open-source STL (`?`) uses, at time of this writing, the approach taken by `Tuple2`.

```
{
    typedef T type;
};
```

`NthType` follows the now familiar pattern of recursive parameter pack handling. The first declaration introduces the recursive case. The specialization that follows handles the limit case `n == 0` to stop the recursion. It is easy to follow that, for example, `NthType<1, short, int, long>::type` is **`int`**.

We are now ready to define the function `get` such that `get<0>(x)` returns a reference to the first element of a `Tuple4` object called `x`.

```
template <std::size_t n, typename... Ts>  // n is the index. Ts is the tuple pack.
auto& get(Tuple4<Ts...>& x)               // top-level function
{
    typedef typename NthType<n, Ts...>::type
        ResultType;                       // Reference to this type is returned.
    typedef Element4<ResultType, sizeof...(Ts) - n - 1>
        ElementType;                      // element holding the value returned
    ElementType& r = x;                   // implicit conversion to get element
    return r.value;                       // Access the value from the Element.
}
```

Calculating the cookie **`sizeof`**`...(Ts) - n - 1` requires some finesse. Recall that the elements in a tuple come with cookies in the reverse order of their natural order, so the first element in a `Tuple4<Ts...>` has cookie **`sizeof`**`(Ts...) - 1` and the last element has cookie `0`. Therefore, when we compute the cookie of the `n`th element in the cookie, we use elementary algebra to get to the expression shown.

After all typing has been sorted out, the implementation itself is trivial; it fetches the appropriate `Element` base by means of an implicit cast and then returns its `value`. Let's put the code to test:

```
void test()
{
    Tuple4<int, double, std::string> value;
    get<0>(value) = 3;
    get<1>(value) = 2.718;
    get<2>(value) = "hello";
    assert(get<2>(value) == "hello");
}
```

The example also illustrates why `get` is best defined as a nonmember function: A member function would have been forced to use the awkward syntax `value.`**`template`**` get<0>() = 3` to disambiguate the use of `<` as a template instantiation as opposed to the less-than operator.

Variant types

`variant-types`

A *variant* type, sometimes called a *discriminated union*, is similar to a C++ **`union`** that keeps track of which of its elements is currently active and protects client code against unsafe uses. Variadic templates support a natural interface to express this design as a generic

library feature.[17] For example, a variant type such as `Variant<int, float, std::string>` would be able to hold exactly one **`int`** or one **`float`** or one `std::string` value. Client code can change the current type by assigning to the variant object an **`int`**, a **`float`**, or an `std::string` respectively.

To define a variant type, we need it to have enough storage to keep any of its possible values, plus a *discriminator* — typically a small integral that keeps the index of the currently stored type. For `Variant<int, float, std::string>`, the discriminator could be by convention 0 for **`int`**, 1 for **`float`**, and 2 for `std::string`.

We saw in the previous sections how to define data structures recursively for parameter packs, so let's try our hand at a variant layout in the `Variant1` design:

```
template <typename... Ts>                    // parameter pack
class Variant1                               // can hold any in parameter pack
{
    template <typename...>                   // union of all types in Ts
    union Store {};

    template <typename U, typename... Us>    // Specialize for >=1 types.
    union Store<U, Us...>
    {
        U head;                              // Lay out a U object.
        Store<Us...> tail;                   // all others at same address
    };

    Store<Ts...> d_data;                     // Store for current datum.
    unsigned int d_active;                   // index of active type in Ts

public:
    // ... (API goes here)
};
```

C-style **`union`**s can be templates, too, and variadic ones at that. We take advantage of this feature in `Variant1` to recursively define `Store<Ts...>` that stores each of the types in the parameter pack `Ts` at the same address. One important C++11 feature that conveys flexibility to the design above is the relaxation on the restrictions on types that can be stored in **`union`**s. In C++03, **`union`** members were not allowed to have non-trivial constructors or destructors. Starting with C++11, any type can be a member in a **`union`**; see Section 3.1."**??**" on page **??**. Therefore, types such as `Variant1<std::string, std::map<int, int>>` work fine.

It is possible to define `d_data` in a more succinct manner as a fixed-size array of **`char`**. There are two challenges to address. First, the size of the array needs to be computed during compilation as the maximum of the sizes of all types in `Ts`. Second, the array needs to have sufficient alignment to store any of the types in `Ts`. Fortunately, both problems have simple solutions in idiomatic modern C++:

```
#include <algorithm>  // std::max
```

[17] C++17's Standard Library type `std::variant` provides a robust and comprehensive implementation of a variant type.

```
template <typename... Ts> class Variant2;  // introducing declaration

template <> class Variant2<>     // specialization for empty Ts
{ /*...*/ };

template <typename T, typename... Ts>
class Variant2<T,Ts...>          // specialization for one or more types
{
    enum : std::size_t { size = std::max({sizeof(T), sizeof(Ts)...}) };
        // std::max takes std::initializer_list and is constexpr in C++14

    alignas(T) alignas(Ts...)
    char d_data[size];           // payload
    unsigned int d_active;       // index of active type in Ts

public:
    // ... (API goes here)
};
```

The code above uses a new use of `std::max` — overload introduced in C++14 that takes an initializer list as parameter; see Section 2.1."**??**" on page **??**. Another novelty is the use of `std::max` during compilation; see Section 2.1."**??**" on page **??**. We apply `std::max` to **`sizeof`**`(T)` and the **`sizeof`**`(Ts)...` expansion, which results in a comma-separated list **`sizeof`**`(T)`, **`sizeof`**`(T0)`, **`sizeof`**`(T1)`, ... (Note it is not the same expansion as **`sizeof`**`...(Ts)`, which would just return the number of elements in `Ts`).

In brief, `d_data` is an array of **`char`** as large as the maximum of the sizes of all of the types passed to `Variant2`. In addition, the **`alignas`** directives instructs the compiler to align `d_data` at the largest alignment of all types among `T` and all of `Ts`; see *Description — Expansion in an alignment specifier* on page 48 and Section 2.1."**??**" on page **??**.

It is worth noting that both `Variant1` and `Variant2` are equally good from a layout perspective; in fact, even the implementation of their respective APIs are identical. The only use of `d_data` is to take its address and use it as a pointer to **`void`**, which the API casts appropriately.

The public interface ensures that when an element is stored in `d_data`, `d_active` will have the value of the index into the parameter pack `Ts...` that corresponds to that type. Hence, when a user attempts to retrieve a value from the variant, if the wrong type is requested, a runtime error will be reported.

Let's take a look at defining some relevant API functions — the default constructor, a value constructor, and the destructor:

```
    std::size_tstd::stringassertstd::runtime_error

#include <algorithm>  // std::max

template <typename... Ts>              // introducing declaration
class Variant;

template <> class Variant<>            // specialization for empty Ts
```

```
{ /*...*/ };

template <typename T, typename... Ts>  // specialization for 1 or more
class Variant<T,Ts...>
{
    enum : std::size_t { size = std::max({sizeof(T), sizeof(Ts)...}) };
        // compute payload size

    alignas(T) alignas(Ts...)
    char d_data[size];                 // approach in Variant1 fine too
    unsigned int d_active;             // index of active type in Ts

    template <typename U, typename... Us>
    friend U& get(Variant<Us...>& v);  // friend accessor

public:
    Variant();                         // default constructor

    template <typename U>              // value constructor
    Variant(U&&);                      // U must be among T, Ts.

    ~Variant();                        // destroy the current object

    // ...
};
```

The default constructor should put the object in a simple, meaningful state. A reasonable decision is to create (by default construction) the *first* object in `Ts`:

```
template <typename T, typename... Ts>
Variant<T, Ts...>::Variant()
{
    ::new(&d_data) T();  // default-constructed T at address of d_data
    d_active = 0;        // Set the active type to T, first in the list.
}
```

The default constructor uses **placement new** to create a default-constructed object at the address of `d_data`. The first element in the parameter pack is selected by means of partial specialization.

The value constructor is a bit more challenging because it needs to compute the appropriate index for `d_active` during compilation, for example as an **`enum`** value. To implement it, first we need a support metafunction that reports the index of a type in a template parameter pack. The first type is the sought type, followed by the types to be searched. If the first type is not among the others, an error is produced:

```
template <typename X, typename T, typename... Ts>  // Find X in T, Ts....
struct IndexOf                                     // primary definition
{
    enum : std::size_t { value = IndexOf<X, Ts...>::value + 1 };
};
```

```
template <typename X, typename... Ts>              // partial specialization 1
struct IndexOf<X, X, Ts...>                        // found X at front
{
    enum : std::size_t { value = 0 };              // found in position 0
};

template <typename X, typename... Ts>              // partial specialization 2
struct IndexOf<const X, X, Ts...>                  // found const X at front
{
    enum : std::size_t { value = 0 };              // also found in position 0
};
```

The `: size_t` syntax, new to C++11, specifies that the introduced anonymous **`enum`** has type `size_t` as a base; see Section 2.1."**??**" on page **??**. The class template `IndexOf` follows a simple recursive pattern. In the general case, the type `X` is different from the first type `T`, and `value` is computed recursively as a search through the tail of the list.

If the sought type is identical to the first in the list, partial specialization 1 kicks in; if the sought type is a **`const`** variant of the second, partial specialization 2 matches. (A complete implementation would also add a similar specialization for the **`volatile`** qualifier.) In either case, the recursion ends and the value `0` is popped up the compile-time recursion stack:

```
std::size_t i1 = IndexOf<int, int, long>::value;         // i1 is 0
std::size_t i2 = IndexOf<int, short, int, long>::value;  // i2 is 1
std::size_t i3 = IndexOf<const int, short, int>::value;  // i3 is 1
std::size_t i4 = IndexOf<int, float, double>::value;     // Error
```

If the type is not found in the pack at all, then the recursion will come to an end when `Ts` is empty and the recursion cannot find a specialization for only one type `T`, resulting in a compile-time error.

It is worth noting that `IndexOf` has an alternative implementation that uses `std::integral_constant`, a Standard Library facility introduced in C++11 that automates part of the `value` definition:

```
#include <type_traits>  // std::integral_constant
template <typename X, typename T, typename... Ts>  // general definition
struct IndexOf2
    : std::integral_constant<std::size_t, IndexOf<X, Ts...>::value + 1> {};

template <typename X, typename... Ts>
struct IndexOf2<X, X, Ts...>
    : std::integral_constant<std::size_t, 0u> {};  // partial specialization 1

template <typename X, typename... Ts>
struct IndexOf2<const X, X, Ts...>
    : std::integral_constant<std::size_t, 0u> {};  // partial specialization 2
```

The type `std::integral_constant<size_t, n>` defines a constant member named `value` of type `size_t` with value `n`, which simplifies to some extent the definition of `IndexOf` and clarifies its intent.

With this template in our toolbox, we are ready to implement `Variant`'s value constructor, using **perfect forwarding** to create a `Variant` holding an object of the given type, with a compile-time error if the specified type is not found in the parameter pack `Ts`.

There is one more detail to handle. By design, we require `Ts` to contain only unqualified (no **`const`** or **`volatile`**), nonreference types, but the value constructor may deduce its type parameter as a reference type (such as **`int&`**). In such cases, we need to extract **`int`** from **`int&`**. The Standard Library template `std::remove_reference_t` was designed exactly for this task — both `std::remove_reference_t<int>` and `std::remove_reference_t<int&>` are aliases for **`int`**:

```
template <typename T, typename... Ts>  // definition for partial specialization
template <typename UARG>               // value constructor
Variant<T, Ts...>::Variant(UARG&& xs)  // uses perfect forwarding
{
    typedef std::remove_reference_t<UARG> U;
        // Remove reference from UARG if any, e.g. transform int& into int.
    ::new(&d_data) U(std::forward<UARG>(xs));  // Construct object at address.
    d_active = IndexOf<U, T, Ts...>::value;    // This code fails if U not in Ts.
}
```

Now we get to construct a `Variant` given any value of one of its possible types:

```
Variant<float, double> v1(1.0F);  // v1 has type float and value 1.
Variant<float, double> v2(2.0);   // v2 has type double and value 2.
Variant<float, double> v3(1);     // Error, int is not among allowed types.
```

A more advanced `Variant` implementation could support implicit conversions during construction as long as there is no ambiguity. Such functionality is supported by `std::variant`.

Now that `Variant` knows the index of the active type in `Ts`, we can implement an accessor function that retrieves a reference to the active element, by a client that knows the type. For example, given a `Variant<short, int, long>` object named `v`, **`int&`** `get<int>(v)` should return a reference to the stored **`int`** if and only if the current value in `v` has indeed type **`int`**; otherwise, it throws an exception:

```
template <typename T, typename... Ts>            // T is the assumed type.
T& get(Variant<Ts...>& v)                        // Variant<Ts...> by ref
{
    if (v.d_active != IndexOf<T, Ts...>::value)  // Is the index correct?
        throw std::runtime_error("wrong type");  // If not, throw.
    void* p = &v.d_data;                         // If so, take store address
    return *static_cast<T*>(p);                  // and convert.
}
```

A overload of `get` that takes and returns **`const`** is defined analogously.

In spite of its simplicity, the `get` function (which needs to be a **`friend`** of `Variant` because it needs access to its **`private`** members) is safe and robust. If given a type not present in the `Variant`'s parameter pack, `IndexOf` fails to compile, and, consequently, `get` does not compile either. If the type is present in the pack but is not the current type stored in the `Variant`, an exception is thrown. If everything works well, the address of data is converted to a reference to the target type, in full confidence that the cast is safe to perform:

```
typedef Variant<long, double, std::string> Var;

Var x1(1L);                             // type long, value 1
Var x2(2.5);                            // type double, value 2.5
Var x3(std::string("hi"));              // type std::string, value "hi"

long y1(get<long>(x1));                 // OK, y1 is 1.
double y2(get<double>(x2));             // OK, y2 is 2.5.
std::string y3(get<std::string>(x3));   // OK, y3 contains "hi".
double y4(get<double>(x3));             // throws exception, wrong type
```

Writing `Variant`'s destructor is more difficult because, in that case, we need to produce the compile-time type of the active element from the *runtime* index, `d_active`. The language offers no built-in support for such an operation, so we must produce a library solution instead.

One idea is to use a linear search approach: Starting with the active index `d_active` and the entire parameter pack `Ts`, we reduce both successively until `d_active` becomes zero. At that point, we know that the dynamic type of the variant is the head of what's left of `Ts`, and we call the appropriate destructor. To implement such an algorithm, we define two overloaded functions, `destroyLinear`, that are friends of `Variant`:

```
template <unsigned int>
void destroyLinear(unsigned int, void*)      // terminates recursion
{ }

template <unsigned int i, typename T, typename... Ts>
void destroyLinear(unsigned int n, void* p)  // index and pointer to data
{
    if (n == i)
        static_cast<T*>(p)->~T();            // found, call destructor manually
    else
        destroyLinear<i + 1, Ts...>(n, p);   // "recurse" with list tail
}
```

The second overload employs the idiom (used in several places in this feature section) of stripping the first element from the parameter pack on each call using a named type parameter. If the runtime index is 0, then the destructor of `T`, the first type in the pack, is called against the pointer received. Otherwise, `destroyLinear` "recursively" calls a version of itself with the parameter pack reduced by 1 and the compile-time counter `i` correspondingly bumped. Note that "recursion" is not quite the correct term because the template instantiates a different function for each call.

The first overload simply terminates the recursion. It is never called in a correct program, but the compiler doesn't know that, so we need to provide a body for it.

`Variant`'s destructor ensures that the variant object is not corrupt and then calls `destroyLinear`, passing the entire pack `Ts...` as template argument and the current index and pointer to state as runtime arguments:

```
template <typename T, typename... Ts>  // definition for partial specialization
Variant<T, Ts...>::~Variant()          // linear lookup destructor
```

```
{
    assert((d_active < Variant<T,Ts...>::size));    // check invariant
    destroyLinear<0, T, Ts...>(d_active, &d_data);  // initiate destruction
}
```

When presented with a linear complexity algorithm, the natural reaction is to look for a similar solution with lower complexity, especially considering that the linear search will be performed at run time, not during compilation. In this case, we might try a *binary search* through the type parameter pack. In common usage, a variant does not have that many types, so this may be considered overkill, but there are two reasons why this problem deserves our attention. First, there are variant types in common use that have a large number of alternatives, such as the `VARIANT` type in the Windows operating system with around 50 options in its union, and some data exchange formats that can have hundreds of types. Second, destructors are of particular importance because they tend to be called intensively, so destructors' size and speed can often affect performance of programs using them.

Defining a binary search that is a mix of compile-time and runtime computations is challenging, particularly because parameter packs do not have indexed access. However, we can implement `destroyLog` with relative ease if we design the algorithm in a hybrid manner — linear search during compilation and binary search at runtime. We do so by defining `destroyLog` to take two integral template parameters, `skip` and `total`, that instruct the template which types in the pack it must look at. If `skip` is greater than 0, the algorithm does a linear search during compilation in the parameter pack. When `skip` is zero, we know we need to search `total` elements in the parameter pack, and we do so in a textbook binary search manner:

```
template <unsigned int, unsigned int>
void destroyLog(unsigned int n, void*) // no-op terminal function
{ }

template <unsigned int skip, unsigned int total, typename T, typename... Us>
void destroyLog(unsigned int n, void* p)
{
    enum : std::size_t { mid = total / 2 };
    if (skip > 0)
        destroyLog<skip - 1, total - 1, Us...>(n, p);
    else if (n == 0)
        static_cast<T*>(p)->~T();
    else if (n < mid)
        destroyLog<0, mid, T, Us...>(n, p);
    else
        destroyLog<mid, total, T, Us...>(n - mid, p);
}
```

There are quite a few moving parts in `destroyLog`, so let's take a look at each in turn. The first overload is simple: Its role, just like before, is to stop the recursion. It takes two **`unsigned int`** parameters, both of which it ignores. The first overload will never be called or even linked.

The bulk of the work is carried by the second overload. First, it computes half of the total in constant `mid`.

Now, to understand how the four branches of the **if**/**else** cascade work, it's important to distinguish the first condition from all others; the `skip > 0` test is essentially a compile-time test and does no computation at run time. All instantiations with a nonzero `skip` will simply recurse to a different instantiation and do nothing else. The compiler will in all likelihood inline the call and eliminate all other code in the function.

Why do we subtract `(skip > 0)` from `skip` and `total` instead of simply subtracting `1`? If we used `skip - 1`, when `skip` reaches `0`, the compiler would attempt to instantiate `destroyLog` with `-1` (even if it never calls that instantiation), which translates into a very large **unsigned**, causing a runaway of further instantiations. The way the condition is written avoids this situation.

All other tests are performed during run time. The second test `n == 0` checks for a match, and, if the test passes, the appropriate destructor is called manually. This is in keeping with how `destroyLinear` works.

The rest of the function is the engine that accelerates `destroyLog` in comparison to `destroyLinear`. If `n` is less than `mid`, then there's no need to consider the upper elements of the pack at all; therefore, `destroyLog` is instantiated with `mid` as the total elements to search. All other arguments — both of the template and runtime kind — are the same. Otherwise, on the contrary, there's no need to look at the first `mid` elements of the pack, so `destroyLog` is instantiated accordingly.

All in all, the `destroyLog` template compactly encodes a linear search during compilation and simultaneously a binary search during run time. As expected, contemporary compilers eliminate dead code entirely and generate code virtually identical to code competently handwritten in a hardcoded approach.[18]

Of course, once we have a logarithmic implementation, we immediately wonder if we can do better, perhaps even a constant time lookup. That brings us to an application of variadic templates in a braced initialization, as described in *Description — Expansion in a function call argument list or a braced initializer list* on page 40. We use braced initialization to generate a table of function pointers, each pointing to a different instantiation of the same function template. For the table to work, each function must have exactly the same signature. For the case of invoking a destructor, we will want to supply the object to be destroyed by using **void*** just like we did with `destroyLinear` and `destroyLog`. We can then write a function template taking a non-deduced type parameter to carry the necessary type information for the function implementation to cast the **void*** pointer to the necessary type:

```
template <typename T>
static void destroyElement(void *p)
{
    static_cast<T*>(p)->~T();
}
```

With this simple function template, we can populate a static array of function pointers by initializing an array of unknown bound (that will implicitly deduce the correct size) with a braced list produced by a pack expansion, taking the address of the `destroyElement` function template instantiated with the types in the pack. Once we have the array of de-

[18] The corresponding code has been tested with Clang 11.0.1 and GCC 10.2 at optimization level `-O3`.

structor functions, matching the expected order of the runtime index `d_active`, we can simply invoke the function pointer at the current index to invoke the correct destructor for the currently active element.

```
template <typename... Ts>
void destroyCtTime(unsigned int n, void *p)  // same signature
{
    typedef void(*destructor)(void *);       // simplify definition
    static const destructor dt[] =           // array of function pointers
        { &destroyElement<Ts>... };          // initialize with pack expansion
    dt[n](p);                                // call appropriate destructor
}
```

Note that this constant-time lookup is also the simplest of the three forms presented since it leans more heavily on integrating variadic pack expansion with other language features, in this case braced array initialization.

It may appear that `destroyCtTime` is the best of the lot: It runs in constant time, it's small, it's simple, and it's easy to understand. However, upon a closer look, `destroyCtTime` has serious performance disadvantages. First, each destructor call entails an indirect call, which is notoriously difficult to inline and optimize except for the most trivial cases.[19]

Second, it is often the case that many types involved in a `Variant` have trivial destructors that do not perform any work at all. The functions `destroyLinear` and `destroyLog` have a white-box approach that naturally leads to the inlining and subsequent elimination of such destructors, leading to a massive simplification of the ultimately generated code. In contrast, `destroyCtTime` cannot take advantage of such opportunities; even if some destructors do no work, they will still be hidden beyond an indirect call, which is paid in all cases.

There is, however, a way to combine the advantages of `destroyCtTime`, `destroyLinear`, and `destroyLog` by using a meta-algorithmic strategy called **algorithm selection**[20]: Choose the appropriate algorithm depending on the `Variant` instantiation. The characteristics of instantiation can be inferred by using **compile-time introspection**. The criteria for selecting the best of three algorithms can be fairly complex. For instantiations with only a few types, most of which have trivial destructors, `destroyLinear` is likely to work best; for moderately large parameter pack sizes, `destroyLog` will be the algorithm of choice; finally, for very large parameter pack sizes, `destroyCtTime` is best.

For a simple example, we can use the parameter pack size as a simple heuristic:

```
template <typename T, typename... Ts>  // definition for partial specialization
Variant<T, Ts...>::~Variant()          // improved destructor implementation
{
    assert((d_active < Variant<T,Ts...>::size));   // check invariant of variant
    void* p = &d_data;                             // d_data as void*
    if (Variant<T,Ts...>::size <= 4)
        destroyLinear<0u, T, Ts...>(d_active, p);  // choose linear algorithm
    else if (Variant<T,Ts...>::size <= 64)
```

[19] GCC 10.2 generates tables and indirect calls even for the most trivial uses. Clang 11.0.1 is able to optimize away indirect calls for locally defined `Variant` objects only if the function has no control flow affecting `Variant` values. Both compilers were tested at optimization level `-O3`.

[20] **?**

```
        destroyLog<0u, Variant<T,Ts...>::size,
                   Ts...>(d_active, p);            // choose log algorithm
    else
        destroyCtTime<T,Ts...>(d_active, p);       // choose O(1) algorithm
}
```

The constants 4 and 64 deciding the thresholds for choosing between algorithms are called **metaparameters** and are to be chosen experimentally. As mentioned, a more sophisticated implementation would eliminate from `Ts...` all types that have trivial destructors and focus only on the types that require destructor calls. Distinguishing between trivial and non-trivial destructors is possible with the help of the Standard Library introspection primitive `std::is_trivially_destructible` introduced in C++11.

advanced-traits

Advanced traits

The use of **`template template`** parameters with variadic arguments allows us to create partial template specializations that match template instances with an arbitrary number of type parameters. This allows the definition of traits that were not possible with prevariadics technology.

For example, consider the family of **smart pointer** templates. A smart pointer type virtually always is instantiated from a template having the pointed-to type as its first parameter:

```
template <typename T>
struct SmartPtr1
{
    typedef T value_type;
    T& operator*() const;
    T* operator->() const;
    // ...
};
```

A more sophisticated smart pointer might take one or more additional template parameters, such as a deletion policy:

```
template <typename T, typename Deleter>
struct SmartPtr2
{
    T& operator*() const;
    T* operator->() const;
    // ...
};
```

`SmartPtr2` still takes a value type as its first template parameter but also takes a `Deleter` functor for destroying the pointed-to object. (The Standard Library smart pointer `std::unique_ptr` added with C++11 also takes a deleter parameter). Note that the author of `SmartPtr2` did not add a nested `value_type`, yet the human reader can easily deduce that `SmartPtr2`'s value type is `T`.

Now, we aim to define a traits class template that, given an arbitrary pointer-like type such as `SmartPtr1<int>` or `SmartPtr2<double, MyDeleter>`, can deduce the value type

pointed-to by the pointer (in our examples, as **int** and **double**, respectively). Additionally, our traits class should allow us to "rebind" the pointer-like type, yielding a new pointer-like type with a different value type. Such an operation is useful, for example, when we want to use the same smart pointer facility as another library, but with our own types.

Suppose, for example, a library defines a type `Widget`:

```
class Widget           // third-party class definition
{ /*...*/ };
```

Furthermore, the same library defines type `WidgetPtr` that behaves like a pointer to a `Widget` type but could be (depending on the library version, debug versus release builds, and so on) either `SmartPtr1` or `SmartPtr2`:

```
class FastDeleter     // policy for performing minimal checking on SmartPtr2
{ /*...*/ };

class CheckedDeleter        // policy for performing maximal checking on SmartPtr2
{ /*...*/ };

#if !defined(DBG_LEVEL)
typedef SmartPtr1<Widget>
    WidgetPtr;                              // release mode, fastest
#elif DBG_LEVEL >= 2
typedef SmartPtr2<Widget, CheckedDeleter>
    WidgetPtr;                              // safe smart pointer for debugging
#else
typedef SmartPtr2<Widget, FastDeleter>
    WidgetPtr;                              // safety/speed compromise
#endif
```

In debug mode (`DBG_LEVEL` defined), the library uses a `SmartPtr2` smart pointer that does additional checking around, for example, dereference and deallocation. There are two possible debugging levels, one with stringent checks and one that cuts a trade-off between safety and speed. In release mode, the library wants to run at full speed so it uses `SmartPtr1`. User code simply uses `WidgetPtr` transparently because the interfaces of the types are very similar.

On the client side, we'd like to define a `GadgetPtr` type that behaves like a pointer to our own type `Gadget` but automatically adjusts to use the same smart pointer underpinnings, if any, that `WidgetPtr` is using. However, we don't have control over `DBG_LEVEL` or over the code introducing `WidgetPtr`. The strategies used by the definition of `WidgetPtr` may change across releases. How can we robustly figure out what kind of pointer — smart or not — `WidgetPtr` is representing?

Let's begin by declaring a primary class template with no body, and then specializing it for native pointer types:

```
template <typename Ptr>
struct PointerTraits;       // incomplete declaration

template <typename T>
struct PointerTraits<T*>   // partial specialization for all raw pointers
```

```
{
    typedef T value_type;  // normalized alias for T
    template <typename U>
    using rebind = U*;     // rebind<U> is an alias for U*
};
```

The new **using** syntax has been introduced in C++11 as a generalized **typedef**; see Section 1.1."**??**" on page **??**. `PointerTraits` provides a basic traits API. For any built-in pointer type, `P`, `PointerTraits<P>::value_type` resolves to whatever type `P` points to. Moreover, for some other type, `X`, `PointerTraits<P>::rebind<X>` is an alias for `X*`, i.e., it propagates the information that `P` is a built-in pointer to `X`:

```
#include <type_traits>  // std::is_same

static_assert(std::is_same<int, PointerTraits<int*>::value_type>::value, "");
static_assert(std::is_same<double*,
                           PointerTraits<int*>::rebind<double>>::value, "");
```

`PointerTraits` has a nested type, `value_type`, and a nested alias template, `rebind`. The first **static_assert** shows that, when `Ptr` is **int***, `value_type` is **int**. The second **static_assert** shows that, when `Ptr` is **int***, `rebind<double>` is **double***. In other words, `PointerTraits` can determine the pointed-to type for a raw pointer and provides a facility for generating a raw pointer type pointing to a *different* value type.

For now, however, `PointerTraits` is not defined for any type that is not a built-in pointer. For `PointerTraits` to work with an arbitrary smart pointer class (such as `SmartPtr1` and `SmartPtr2` above but also `std::shared_ptr`, `std::unique_ptr`, and `std::weak_ptr`), we must partially specialize it for a template instantiation, `PtrLike<T, X...>`, where `T` is assumed to be the value type and `X...` is a parameter pack of zero or more additional type parameters to `PtrLike`:

```
template <
    template <typename, typename...> class PtrLike,
    typename T, typename... X>
struct PointerTraits<PtrLike<T, X...>>  // partial specialization for template
{
    using value_type = T;               // extract pointee type
    template <typename U>
    using rebind = PtrLike<U, X...>;    // rebind to some other type U
};
```

This partial specialization will produce the correct result for any pointer-like class template that takes one or more type template parameters, where the first parameter is the value type of the pointer. First, it correctly deduces the `value_type` from the first argument to the template:

```
typedef SmartPtr2<Widget, CheckedDeleter>  WP1;  // fully checked
typedef SmartPtr2<Widget, FastDeleter>     WP2;  // minimally checked

static_assert(std::is_same<
    PointerTraits<WP1>::value_type,  // Fetch the pointee type of WP1.
```

```
    Widget>::value, "");            // should be Widget

static_assert(std::is_same<
    PointerTraits<WP1>::value_type,  // Fetch the pointee type of WP2.
    Widget>::value, "");             // should also be Widget
```

Second, `rebind` is able to reinstantiate the original template, `SmartPtr2`, with another first argument but the same second argument:

```
class Gadget { /*...*/ };

static_assert(std::is_same<
    PointerTraits<WP1>::rebind<Gadget>,  // Rebind WP1 to Gadget.
    SmartPtr2<Gadget, CheckedDeleter>>   // fully checked, just like WP1
::value, "");

static_assert(std::is_same<
    PointerTraits<WP2>::rebind<Gadget>,  // Rebind WP2 to Gadget.
    SmartPtr2<Gadget, FastDeleter>>      // minimally checked, like WP2
::value, "");
```

The Standard Library facility `std::pointer_traits`, introduced with C++11, is a superset of our `PointerTraits` example.

Potential Pitfalls

tial-pitfalls-variadic

Accidental use of C-style ellipsis

use-of-c-style-ellipsis

Inside the function parameters declaration, `...` can be used only in conjunction with a template parameter pack. However, there is an ancient use of `...` in conjunction with C-style variadic functions such as `printf`. That use can cause confusion. Say we set out to declare a simple variadic function, `process`, that takes any number of arguments by pointer:

```
class Widget;                   // declaration of some user-defined type

template <typename... Widgets>  // parameter pack named Widgets
int process(Widget*...);        // meant as a pack expansion, but is it?
```

The author meant to declare `process` as a variadic function taking any number of pointers to objects. However, instead of `Widgets*...`, the author mistakenly typed `Widget*...`. This typo took the declaration into a completely different place: It is now a C-style variadic function in the same category as `printf`. Recall the `printf` declaration in the C Standard Library:

```
int printf(const char* format, ...);
```

The comma and the parameter name are optional in C and C++, so omitting both leads to an equivalent declaration:

```
int printf(const char*...);
```

Comparing `process` (with the typo in tow) with `printf` makes it clear that `process` is a C-style variadic function. Runtime errors of any consequence are quite rare because the

expansion mechanisms are very different across the two kinds of variadics. However, the compile- and link-time diagnostics can be puzzling. In addition, if the variadic function ignores the arguments passed to it, calling it may even compile, but the call will use a different calling convention than that intended and assumed.

As an anecdote, a similar situation occurred during the review stage of this feature section. A simple misunderstanding caused a function to be declared inadvertently as a C-style variadic instead of C++ variadic template, leading to numerous indecipherable compile-time and link-time errors in testing that took many emails to figure out.

undiagnosed-errors

Undiagnosed errors

Description — Corner cases of function template argument matching on page 29 shows definitions of variadic template functions that are in error according to the C++ Standard yet pass compilation on contemporary compilers — that is, IFNDR. In certain cases, they can even be called. Such situations are most assuredly latent bugs:

```
template <typename... Ts, typename... Us, typename T>
int process(Ts..., Us..., T);
    // ill-formed declaration - Us must be empty in every possible call
int x = process<int, double>(1, 2.5, 3);
    // Ts=<int, double>, Us=<>, T=int
```

In virtually all cases, such code reflects a misplaced expectation; an always-empty parameter pack has no reason to exist in the first place.

e-number-of-arguments

Compiler limits on the number of arguments

The C++ Standard recommends that compilers support at least 1024 arguments in a variadic template instantiation. Although this limit seems very generous, real-world code may push against it, especially in conjunction with generated code or combinatorial uses.

This limit may lead to a lack of portability in production — for example, code that works with one compiler but fails with another. Suppose, for example, we define `Variant` that carries all possible types that can be serialized in a large application:

```
typedef Variant<
    char,
    signed char,
    unsigned char,
    short,
    unsigned short
    // ... (more built-in and user-defined types)
>
WireData;
```

We release this code to production and then, at some later date, clients find it fails to build on some platforms, leading to a need to reengineer the entire solution to provide full cross-platform support.

annoyances-variadic

Annoyances

unusable-functions

Unusable functions

Before variadics, any properly defined template function could be called by using explicit template argument specification, type deduction, or a combination thereof. Now it is possible to define variadic function templates that pass compilation but are impossible to call (either by using explicit instantiation, argument deduction, or both). This may cause confusion and frustration. Consider, for example, a few function templates, none of which take any function parameters:

```
template <typename T>                  // template with one parameter
int f1();

template <typename T, typename U>      // template with two parameters
int f2();

template <typename... Ts>              // pack followed by type parameter
int f3();

template <typename T, typename... Ts>  // parameter pack at the end
int f4();

template <typename... Ts, typename T>  // pack followed by type parameter
int f5();
```

The first four functions can be called by explicitly specifying their template arguments:

```
int a1 = f1<int>();              // T=int
int a2 = f2<int, long>();        // T=int, U=long
int a3 = f3<char, int, long>();  // Ts=<char, int, long>
int a4 = f4<char, int, long>();  // T=char, Ts=<int, long>
```

However, there is no way to call `f5` because there is no way to specify `T`:

```
int a5a = f5();             // Error, cannot infer type argument for T
int a5b = f5<int>();        // Error, cannot infer type argument for T
int a5c = f5<int, long>();  // Error, cannot infer type argument for T
```

Recall that by the Rule of Greedy Matching (see *Description — The Rule of Greedy Matching* on page 25), `Ts` will match all of the template arguments passed to `f5`, so `T` is starved. Such uncallable functions are IFNDR. There are several other variations that render variadic function templates uncallable and therefore IFNDR. However, most contemporary compilers do allow compilation of `f5`.[21] If other overloads of such an uncallable function are present, the matter may be interpreted as the wrong overload being called.

s-on-expansion-contexts

Limitations on expansion contexts

As discussed in *Description — Pack expansion* on page 36 and in the sections that follow it, expansion contexts are prescriptive. The standard enumerates all expansion contexts

[21] The corresponding code has been tested with Clang 11.0 and GCC 7.5.

exhaustively: There is no other place in a C++ program where a parameter pack is allowed to occur, even if it seems syntactically and semantically correct.

For example, consider a variadic function template, `bump1`, that attempts to expand and increment each of the arguments in its parameter pack:

```
template <typename... Ts>
void bump1(Ts&... vs)  // some variadic function template
{
    ++vs...;           // Error, can't expand parameter pack at statement level
}
```

Attempting to expand a parameter pack at the statement level is simply not among the allowed expansion contexts; hence, the example function body above fails to compile.

Such limitations can be worked around by artificially creating an expansion context. For example, we can achieve our goal by replacing the erroneous line in `bump1` (above) with one in `bump2` (below) that, say, creates a local class with a constructor that takes `Ts&...` as parameters:

```
template <typename... Ts>
void bump2(Ts&... vs)
{
    struct Local          // local struct
    {
        Local(Ts&...) {}  // constructor takes each of Ts by reference
    }                     // no semicolon here, will create an object
    local(++vs...);       // OK, expansion allowed in constructor call
}
```

The code above creates a local **`struct`** called `Local` with a constructor and immediately constructs an object of that type called `local`. Expansion is allowed inside the argument list for the constructor call, which makes the code work.

A possibility to achieve the same effect with terser code is to create a lambda expression, `[](Ts&...){}` and then immediately call it with the expansion `++vs...` as its arguments:

```
template <typename... Ts>
void bump3(Ts&... vs)          // some variadic function template
{
     ([](Ts&...){})(++vs...);  // OK, pack expansion allowed in lambda call
}
```

The function above works, albeit awkwardly, by making use of another feature of C++11 that essentially allows us to define an anonymous function *in situ* and then invoke it; see Section 2.1."**??**" on page **??** [22]

Parameter packs cannot be used unexpanded

ot-be-used-unexpanded

As discussed in *Description — Pack expansion* on page 36, the name of a parameter pack cannot appear on its own in a correct C++ program; the only way to use a parameter pack

[22] C++17 adds *fold expressions* that allow easy pack expansion at expression and statement level. The semantics desired for the example shown would be achieved with the syntax `(...++vs);`.

is as part of an expansion by using `...` or **`sizeof`**. Such behavior is unlike types, template names, or values.

It is impossible to pass parameter packs around or to give them alternative names (as is possible with types by means of **`typedef`** and **`using`** and with values by means of references). Consequently, it is also impossible to define them as "return" values for metafunctions following conventions such as `::type` and `::value` that are commonly used in the `<type_traits>` standard header.

Consider, for example, sorting a type parameter pack by size. This simple task is not possible without a few helper types because there is no way to return the sorted pack. One necessary helper would be a typelist:

```
template <typename...> struct Typelist { };
```

With this helper type in hand, it is possible to encapsulate parameter packs, give them alternate names, and so on — in short, give parameter packs the same maneuverability that C++ types have:

```
typedef Typelist<short, int, long, float, double, long double> Numbers;
    // can be used to give a pack an alternate name

template <typename L>
struct SortBySize
{
    using type = Typelist< /*...*/ >;  // computed sorted by size version of
                                       // the Typelist L
};

typedef SortBySize<Numbers>::type SortedNumbers;
    // can be used to "return" a pack from a metafunction
```

Currently no `Typelist` facility has been standardized. An active proposal[23] introduces `parameter_pack` along the same lines as `Typelist` above. Meanwhile compiler vendors have attempted to work around the problem in nonstandard ways.[24] A related proposal[25] defines `std::bases` and `std::direct_bases` but has, at the time of writing, been rejected.

Expansion is rigid and requires verbose support code

es-verbose-support-code

There are only two syntactic constructs that apply to parameter packs: **`sizeof...`** and expansion via `...`. The latter underlies virtually all treatment of variadics and, as discussed, requires handwritten support classes or functions as scaffolding toward building a somewhat involved recursion-based pattern.

There is no expansion in an expression context, so it is not possible to write functions such as `print` in a concise, single-definition manner; see *Use Cases — Generic variadic*

[23] **?**

[24] GNU defines the nonstandard primitives `std::tr2::__direct_bases` and `std::tr2::__bases`. The first yields a list of all direct bases of a given class, and the second yields the transitive closure of all bases of a class, including the indirect ones. To make these artifacts possible, GNU defines and uses a helper `__reflection_typelist` class template similar to `Typelist` above.

[25] **[AUs: url please, so I can cite this. Response: http://wg21.link/N2965**

functions on page 52. In particular, expressions are not expansion contexts so the following code would not work:

```
#include <iostream>  // std::cout, std::ostream

template <typename T, typename... Ts>
std::ostream& print(const T& v, const Ts&... vs)
{
    std::cout << vs...;             // Error, invalid expansion
    return std::cout << std::endl;
}
```

search-for-everything

Linear search for everything

One common issue with parameter packs is the difficulty of accessing elements in an indexed manner. Getting to the nth element of a pack is a linear search operation by necessity, which makes certain uses awkward and potentially time-consuming during compilation. Refer to the implementation of `destroyLog` in *Use Cases — Variant types* on page 60 as an example.

see-also

See Also

- "??" (§2.1, p. ??) ♦ illustrates one of the expansion contexts for function parameter packs.
- "??" (§2.1, p. ??) ♦ describes a feature used in conjunction with variadics to achieve perfect forwarding.
- "??" (§2.1, p. ??) ♦ describes a feature that allows expansion both in capture list and in the arguments list.

further-reading

Further Reading

- "F.21: To return multiple"out" values, prefer returning a struct or tuple," ?
- ?

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

sec-unsafe-cpp11 Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14