C++20 idioms for parameter packs

David Mazières

June, 2022

Introduction

C++11 introduced variadic templates, which permit type-safe functions to accept a variable number of arguments. They also permit template types such as `std::tuple` that can hold a variable number of elements. The main language mechanism enabling variadic templates is *parameter packs*, which hold an arbitrary number of values or types. Some things are easy to do with parameter packs—for instance, passing the values they comprise to a function. Other tasks are a bit trickier to accomplish, such as iterating over a parameter pack or extracting specific elements. However, these things can generally be accomplished through various idioms, some more unwieldy than others.

Between C++11 and C++20, the language gained several improvements to variadic templates. Improvements to other features, such as concepts and lambdas, have also created new options for manipulating parameter packs, thereby enabling new variadic template idioms. This post lays out a grab-bag of techniques for using parameter packs in C++20. Ideally, cataloging these tricks makes it easier for people to do what they need with variadic templates. My interest in producing a clean C++20-focused exposition stems from a conjecture that variadic templates are easier to learn to use without the baggage of how we used to do things in C++17 and earlier. Moreover, even if a lot of the idioms are obvious at a high level, they provide a good context in which to showcase some of the new features of C++20.

Overview of variadic templates

A variadic template is a template whose definition captures a *parameter pack* in its template arguments or function arguments. A parameter pack is captured by introducing an identifier prefixed by an ellipsis, as in `...X`. Once captured, a parameter pack can later be used in a *pattern* expanded by an ellipsis (generally, but not always, to the right of the pattern). Pack expansion is conceptually equivalent to having one copy of the pattern for each element of the parameter pack. Here's a silly example of a program that prints "`one two `":

```
void
print_strings(std::convertible_to<std::string_view> auto&& ...s)
{
```

```
4     for (auto v : std::initializer_list<std::string_view>{ s... })
5       std::cout << v << " ";
6     std::cout << std::endl;
7   }
8
9   int
10  main()
11  {
12    print_strings("one", std::string{"two"});
13  }
```

The `print_strings` function takes an arbitrary number of arguments, all of which are captured by the parameter pack `...s` in line 2. In line 4, this parameter pack is expanded as `s...` to specify the values from which to construct an `initializer_list`. We then iterate over the `initializer_list` to print the strings.

As a reminder, the appearance of the placholder `auto` in the arguments of `print_strings` makes `print_strings` an abbreviated function template, which introduces an implicit template type parameter for each occurrence of the placeholder. The use of `auto&&` as opposed to `auto` or `auto&` is known as a forwarding reference, which can accept both lvalue and rvalue references (a confusing syntax since in most contexts postfix `&&` matches only rvalue references).[1] We need the universality of a forwarding reference because `"one"` is an lvalue (of type `const char(&)[4]`) while `std::string{"two"}` is a prvalue—the former cannot be captured by rvalue reference and the latter cannot be captured by non-const lvalue reference.[2] Finally, note that the type-constraint `convertible_to` restricts the types that match the template argument; without this constraint, the program would still work, but invocations of `print_strings` with incompatible types would create less intuitive error messages and, worse, any overloads of the `print_strings` function would cause ambiguity errors.

Expanding parameter packs

A captured parameter pack must be used in a *pattern* that is *expanded* with an ellipsis (`...`). A pattern is a set of tokens containing the identifiers of one or more parameter packs. When a pattern contains more than one parameter pack, all packs must have the same length. This length determines the number of times the pattern is conceptually replicated in the expansion, once for each position in the expanded pack(s). Here's a simple example:

```
void dummy(auto&&...) {}

template<std::same_as<char> ...C>
void
expand(C...c)
```

[1]The fact that `auto&&` produces forwarding references is not mentioned in the definition of forwarding references, but follows from the fact that type deduction for `auto` follows the same rules as templates.

[2]See my previous blog post on value categories for an explanation of why string literals are lvalues and what a prvalue is.

```
6  {
7    std::tuple<C...> tpl(c...);
8
9    const char msg[] = { C(std::toupper(c))..., '\0' };
10   dummy(msg, c...);
11 }
```

In line 3, the function `expand` captures a template parameter pack `C` consisting of a sequence of zero or more types, all of which must be `char` because of our use of the `std::same_as` concept. In line 5, we capture a function parameter pack `c` consisting of a sequence of values c_i each of type C_i for the ith position in parameter pack `C`. (Of course, in this example every C_i is `char`.) We then see several contexts in which these packs are expanded:

- In line 7, `tuple<C...>` expands the pack `C` in a *template-argument-list*, while `tpl(c...)` expands `c` in an *initializer-list* (which, not to be confused with `std::initializer_list`, is the term in the C++ grammar for comma-separated lists of expressions passed as arguments to function calls and constructors).
- In line 9, we expand the pattern `C(std::toupper(c))` in another initializer list. This is an example of a pattern with two packs, `C` and `c`, both of which have the same length and are expanded in lockstep. (`std::toupper` returns `int` rather than `char`, so its result requires a cast, though we could equivalently have written `char(std::toupper(c))...` in this case.)
- In line 10, we again expand `c` in an initializer list.

In most cases, an expanded pattern is conceptually equivalent to a number of copies of the pattern equal to the size of the parameter pack. Unless otherwise noted, a pattern is expanded by appending an ellipsis (`...`). It is illegal to use a captured parameter pack except in a pattern expanded by an ellipsis. Here is the list of contexts in which a pattern can be expanded:

- In **initializer-lists** (as shown above), including pack expansion in the arguments to a function call. Conceptually, such a pack expansion is equivalent to a comma-separated list of instances of the pattern.
- In **base specifier lists**, to specify one base class for each member of a type parameter pack, e.g.:

  ```
  template<typename ...Base>
  struct MyStruct : Base... {
      MyStruct();
  };
  ```

- When initializing base classes in a **mem-initializer list** in a class constructor, the pack expansion initializes a list of base classes based on a type parameter pack:

  ```
  template<typename ...Base>
  MyStruct<Base...>::MyStruct() : Base()... {}
  ```

- In **template argument lists** as in `std::tuple<C...>`, the pack expands to the equivalent of a comma-separated list of template arguments.
- In lambda **capture lists**, the pattern expansion is equivalent to a comma-separated list of captures. E.g.,

```
void
f(auto...arg)
{
  auto with_copy = [arg...]{
    /* do something with arg... */
  };
  with_copy();

  auto with_reference = [&arg...]{
    /* do something with arg... */
  };
  with_reference();
}
```

- Inside **function parameters** and **template parameters**, a pack expansion behaves like a comma separated list of patterns. An example in function parameters is the expansion `C` in the definition of `expand(C...c)`, above. An example in template parameters is the expansion of `T` in `Inner`, here:

```
template<typename ...T> struct Outer {
  template<T...V> struct Inner {
  };
};
```

  Note how in these these cases, the ellipsis plays double-duty, serving at once to expand one parameter pack and capture another. The `...` must always immediately precede the identifier of the captured parameter pack. This means the ellipsis falls in the middle of the pattern for arrays and function types, rather than at the end, but the pattern is still expanded as usual. For example:

```
template<std::size_t ...N>
void process_strings(const char (&...s)[N]) { /* ... */ }
// conceptually like:
// process_strings(const char s1[N1], const char s2[N2], etc.)

template<typename ...T>
auto function_results(T (&...f)()) { return std::tuple(f()...); }
// conceptually like:
// function_results(T1(&f1)(), T2(&f2)(), etc.)
```

- In a **using declaration**, the pattern conceptually expands to a list of `using` declarations.

```
template<typename ...Base>
struct MyStruct :  Base... {
    MyStruct();
    using Base::f...;
    // Conceptually equivalent to:
    //   using Base_1::f;
    //   using Base_2::f;
    //   ...
};
```

Obviously a `using` pattern is most useful when the method `f` of each base class in the pack has a different type signature—otherwise invoking `f` would be ambiguous. See multilambda for a great use of `using` patterns.

- In an **alignment specifier**, the argument must be a single parameter pack and the ellipsis goes inside the `alignas` operator. The result is an alignment restriction compatible with all the types (if the pack expands to types) or all the powers of two (if the pack expands to integer powers of two). In the following example, the type `storage<int, void*>` would be aligned to an address compatible with both `int` and `void*`. (Note also the expansion `sizeof(T)...` inside braces to create a `std::initializer_list<std::size_t>` argument for `std::max`.)

```
template<typename ...T>
struct alignas(T...) storage {
  char contents[std::max({ sizeof(T)... })];
};
```

- The standard also allows for pack expansions inside **attribute lists**. However, this feature does not apply to any standard attributes, and must be intended for compiler-specific ones.

While a pack expansion mostly behaves like a series of copies of the pattern, it is okay to have a pack of size zero even when the program wouldn't otherwise be syntactically well formed. For example, while `f(x,)` and `struct MyStruct : {};` are not valid C++ syntax, `f(x, pack...)` and `struct MyStruct : Base... {};` are okay even with empty parameter packs for `pack` and `Base`.

A pattern may itself contain an expanded parameter pack, in which case there is no need for the inner and outer packs to contain the same number of elements. The expansion of the inner pack simply becomes part of the pattern around the outer pack. For example:

```
constexpr int
sum(std::convertible_to<int> auto ...il)
{
  int r = 0;
  for (int i : { int(il)... })
    r += i;
  return r;
```

```
}

template<int ...N>
struct Nested {
  static constexpr int nested_sum(auto ...v) {
    return sum(sum(N..., v)...);
  }
};

static_assert(Nested<1>::nested_sum(100, 200) == 302);
// Equivalent to:  sum(sum(1, 100),sum(1, 200)) == 302
```

It is worth noting that a pack expansion is *not* valid outside of the contexts listed above. In particular, you cannot expand a free-floating expression (though see folds below), and you cannot expand a `case` clause in a `switch` statement.

`sizeof...(pack)`

The `sizeof...` operator returns a `std::size_t` corresponding to the number of elements in a parameter pack. While technically considered a pack expansion, it only ever returns a single value. Unlike ordinary `sizeof`, the argument to `sizeof...` must always be parenthesized and consist of a single identifier naming a parameter pack.

Folds

Another special form of pack expansion is folds, introduced in C++17. Above, we showed a function `sum` that summed a set of integers. This function can be implemented far more concisely with a fold:

```
constexpr int
sum(std::convertible_to<int> auto ...i)
{
  return (0 + ... + i);
}
```

Folds are defined in terms of the grammar rule for a *cast-expression*, which is a C++ expression whose outer operators bind at least a tightly (i.e., have precedence at least as high as) the C-style cast operator `(Type) val`. As an example, `&p[5]` is a cast-expression, because the left-associative subscript (`[]`) operator binds more tightly than a cast, while the right associative address-of operator (`&`) has the same precedence as a cast. By contrast, the expression `3*i` is not a cast-expression, because binary `*` has lower precedence than a cast. Parenthesizing an expression with lower-precedence operators, such as `(3*i)`, makes it into a cast-expression.

There are four types of fold in C++. In these examples, let `pat` be a cast-expression containing one more more unexpanded parameter packs (i.e., a pattern). Let `e` be a normal cast-expression without any unexpanded parameter packs. Let $p_1, \ldots, p_n$ be the instances of `pat` corresponding to each element captured by `pat`'s unexpanded parameter packs. Let $\oplus$ stand for any binary operator in the C++ grammar (`.*`, `->*`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `<=>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||`, `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`, or the comma operator "`,`").

A **binary left fold** has the form $\texttt{(e}\oplus\texttt{...}\oplus\texttt{pat)}$ and is equivalent to $\texttt{(((e}\oplus p_1\texttt{)}\oplus p_2\texttt{)}\oplus\cdots\texttt{)}\oplus p_n$.

A **unary left fold** has the form $\texttt{(...}\oplus\texttt{pat)}$ and is equivalent to $\texttt{((}p_1 \oplus p_2\texttt{)}\oplus\cdots\texttt{)}\oplus p_n$.

A **binary right fold** has the form $\texttt{(pat}\oplus\texttt{...}\oplus\texttt{e)}$ and is equivalent to $p_1\oplus\texttt{(}p_2\oplus\texttt{(}\cdots\oplus\texttt{(}p_n\oplus\texttt{e)))}$.

A **unary right fold** has the form $\texttt{(pat}\oplus\texttt{...)}$ and is equivalent to $p_1\oplus\texttt{(}p_2\oplus\texttt{(}\cdots\oplus p_n\texttt{))}$.

Note that parentheses are always required around a fold, regardless of context.

When the parameter pack is empty (has size 0), a binary fold is equivalent to `e`. A unary fold over an empty parameter pack is only permitted for 3 specific binary operators:

- If $\oplus$ is `&&`, then an empty unary fold is equivalent to `true`.
- If $\oplus$ is `||`, then an empty unary fold is equivalent to `false`.
- If $\oplus$ is the comma operator `,`, then an empty unary fold is equivalent to `void()`.

For all other binary operators, a unary fold over an empty parameter pack results in an ill-formed program.

Capturing parameter packs

While parameter packs can be expanded in a variety of places, they can only be captured in a much more restricted set of contexts. There's an appealing proposal for allowing parameter packs in structured bindings, which would simplify a lot of idioms, but as of now there are only three contexts in which you can introduce a new pack: template parameter packs, function parameter packs, and init-capture packs. In all cases, the ellipsis must appear immediately to the left of the identifier capturing the parameter pack (`...X`).

Template parameter packs

Template parameter packs consist of types, templates, and values within the angle brackets of a template definition. Any normal template parameter can be turned into a pack by prefixing the identifier with an ellipsis. For example:

```
template<typename ...T> struct S1{};
template<int ...I> struct S2{};
template<template<typename> typename ...Tmpls> struct S3{};
```

With two exceptions, a template parameter pack must generally be the last entry in the template parameter list. The first exception is for function templates, where template arguments can be inferred from the function arguments. So long as every template parameter following a captured parameter pack can be inferred, it is okay for the pack not to be last. The second exception is in template specializations, where captured packs may be used in the specialization. For example:

```
// Illegal for pack not to be last
template<typename ...T1, typename ...T2> struct S{}; // error
S<int, int, bool> a;  // If this were legal, what would T1 and T2 be?


// Okay to put ...Tmpls first because T inferred from function argument
template<template<typename...> typename ...Tmpls, typename T>
auto
ptr_tuple(const T &v)
{
  // Exception-safe since C++17 (see P0145R3)
  return std::tuple(Tmpls<T>(new T(v))...);
}
auto ones = ptr_tuple<std::shared_ptr, std::unique_ptr>(1);


using std::tuple;

template<typename T1, typename T2, typename T3>
struct is_tuple_cat : std::false_type {};

// Okay for ...T1 not to be last in specialization
template<typename ...T1, typename ...T2>
struct is_tuple_cat<tuple<T1...>, tuple<T2...>, tuple<T1..., T2...>>
  : std::true_type {};

static_assert(is_tuple_cat<tuple<int>, tuple<char*>, tuple<int, char*>>{});
static_assert(!is_tuple_cat<tuple<int>, tuple<char*>, tuple<int, bool>>{});
```

A template parameter can also be an expansion of another parameter pack, as we saw in the definition of `Inner` above.

Function parameter packs

Function parameter packs consist of values in the argument list of a function. We've seen examples like `c` in `expand(C...c)` and `i` in `sum(std::convertible_to<int> auto ...i)`. There's a big restriction that a function parameter pack must either itself be a pack expansion (as in `expand`) or else contain the placeholder `auto` (as in `sum`). Otherwise, the program is ill-formed. This is why all the examples make heavy use of `std::convertible_to`. The C++ committee considered but rejected allowing *homogeneous variadic function parameters*, which would have permitted the following simpler code:

```
template<> constexpr int
sum(int ...i)  // illegal
{
  return (0 + ... + i);
```

```
}
```

Why not allow the above code? The problem lies in the fact that, for compatibility with really old C++, there are two ways of defining a C-style variadic function function:

```
int printf(const char *, ...);  // better way, required by C
int printf(const char *...);    // 1983 C++ way, before C had prototypes
```

If C++ allowed homogeneous variadic function parameters, there would be an ambiguity between the older style of varargs definition above and a template function `printf` accepting a homogeneous parameter pack of `const char *` values. The standard unfortunately requires that the ambiguity be resolved in favor of the C-style varargs interpretation. The proposal to fix this did require an extra `template<>` in front of our definition of `sum`, which avoided the ambiguity. Moreover, if you give the parameter pack `i` a name, that also avoids any ambiguity. But the proposal lost anyway, so you can't do that.

Is there a workaround for the lack of homogeneous variadic function parameters? The following two functions seem like reasonable substitutes for the illegal `f(int ...i)`:

```
int f1(std::same_as<int> auto ...i);
int f2(std::convertible_to<int> auto ...i);
```

Unfortunately, it is important to realize that neither is quite equivalent. When a parameter type is declared `int`, unlike `auto`, it triggers integer conversion. A concept such as `same_as` doesn't change the type inferred by `auto`, it only restricts permissible types. Hence, calling `f1(0, sizeof(int))` won't match the above function, because `sizeof(int)` is a `std::size_t`, so the second argument type is inferred as `std::size_t`, which fails the constraint test `std::same_as<int>`. The invocation `f2(0, sizeof(int))` is okay, but its arguments are of heterogeneous type, which means inside `f2` writing code such as `for (int n : { i... })` won't work.

A workaround for `f2` is to use a cast in the pattern for expanding `i`, as in `for (int n : { int(i)... })`. This solution is perfectly fine for `int`. Unfortunately, for types with non-trivial constructors, such as `std::string`, `f2` can lead to gratuitous copies. For example, suppose that instead of taking `int`, we have a type `Obj`:

```
struct Obj { void use() { /* ... */ } /* ... */ };

void good() {}
void good(Obj o1) { o1.use(); }
void good(Obj o1, Obj o2) { o1.use(); o2.use(); }
// ...

void
bad(std::convertible_to<Obj> auto&& ...o)
{
  // Unary fold over comma
  (Obj{std::forward<decltype(o)>(o)}.use(), ...);
}
```

If you call `good({})` or `good(Obj{})`, you will construct exactly one object of type `Obj`. If you call `bad({})`, the program is ill-formed (the compiler doesn't know what type `{}` should be). If you call `bad(Obj{})`, then you will construct *two* objects of type `Obj`. First, a temporary will be constructed to pass into `bad`. Next, the cast expression in the fold will move-construct a second `Obj` object from the first. See the homogeneous function parameter packs idiom for a way to work around this problem.

Init-capture packs

The simplest way to capture a parameter pack in a lambda expression is simply to expand it into a conceptual list of values to capture, as seen above. Sometimes, however, you want to capture variables with an explicit initializer. For example, if your function parameter pack contains rvalue references, it may be more efficient to move-initialize the lambda's captures than to copy-initialize them. You can capture a parameter pack in the capture clause of a lambda by prefixing an identifier with an ellipsis. In this case, the initializer must be a pattern containing one or more packs (of the same length). Here's an example that will avoid copying any temporary strings passed in as arguments:

```
template<std::convertible_to<std::string> ...T>
auto
make_prefixer(T&& ...args)
{
  using namespace std::string_literals;
  return [...p=std::string(std::forward<T>(args))](std::string msg) {
    // binary right fold over +
    return ((p + ": "s + msg + "\n"s) + ... + ""s);
  };
}

int
main()
{
  auto p = make_prefixer("BEGIN", "END");
  std::cout << p("message");
  // prints:
  // BEGIN: message
  // END: message
}
```

Idioms

Below is a collection of idioms for working with parameter packs. I place all the code in this blog post in the public domain, so feel free to cut and paste. To keep things concise, I've omitted include files, but the system library features used in the examples come from the following set of includes:

```
#include <algorithm>
#include <array>
#include <concepts>
#include <initializer_list>
#include <iostream>
#include <memory>
#include <string>
#include <tuple>
#include <type_traits>
#include <utility>
#include <variant>
```

Recursing over argument lists

The most basic variadic template idiom, probably already known to most people reading this blog post, is to iterate over the argument list recursively, using function overloading to differentiate the base case (no arguments) from the recursive case (one or more). A silly example:

```
inline void
printall()
{
}

void
printall(const auto &first, const auto &...rest)
{
  std::cout << first;
  printall(rest...);
}
```

Of course, since C++17, many uses of recursion are better accomplished with folds:

```
void
printall2(const auto &...args)
{
  // binary left fold
  (std::cout << ... << args);
}
```

Recursing over template parameters

Another common technique is to recurse over template parameters. This can be done to consume the elements of a parameter pack, produce elements of a parameter, or both. Here's a simple example in which we recursively produce the arguments of a parameter pack. The goal is, at compile time, to produce a string corresponding to a number (so that you can safely use the string even in global initializers). We recurse over the parameter `N` so long as it is greater than 10, producing one digit at a time. Finally, when `N` is less than 10, we return a `char[]` from inside the `string_holder` template.

```
template<char ...Cs>
struct string_holder {
  static constexpr std::size_t len = sizeof...(Cs);
  static constexpr char value[] = { Cs..., '\0' };
  constexpr operator const char *() const { return value; }
  constexpr operator std::string() const { return { value, len }; }
};

template<size_t N, char...Cs>
consteval auto
index_string()
{
  if constexpr (N < 10)
    return string_holder<N+'0', Cs...>{};
  else
    return index_string<N/10, (N%10)+'0', Cs...>();
}

// "10"
constinit const char *ten = index_string<10>();
```

If you want to consume and produce argument packs recursively, then you need to use some kind of holder type so as to accommodate multiple parameter packs simultaneously. The `string_holder` type above is an example of such a holder type. Suppose we want a function `add_commas` that adds a comma between every three characters of a string starting from the right. We can do this by consuming the argument pack of one `string_holder` while producing the argument pack of another:

```
template<char ...Out>
consteval auto
add_commas(string_holder<>, string_holder<Out...> out)
{
  return out;
}

template<char In0, char ...InRest, char ...Out>
consteval auto
add_commas(string_holder<In0, InRest...>, string_holder<Out...> = {})
{
  if constexpr (sizeof...(InRest) % 3 == 0 && sizeof...(InRest) > 0)
    return add_commas(string_holder<InRest...>{},
                      string_holder<Out..., In0, ','>{});
```

```
  else
    return add_commas(string_holder<InRest...>{},
                      string_holder<Out..., In0>{});
}

// "1,000,000"
constinit const char *million = add_commas(index_string<1'000'000>());
```

Comma fold

Often you want to do the same operation to every element in a parameter pack. While you can accomplish this by recursing over the parameter pack, it can be simpler to use a fold over the comma operator, which just sequences one expression after the other. To avoid any strange behavior in cases where the program overloads `operator,`, you can cast the expression to void. Here's a simple example of a function that inserts an arbitrary number of elements into a container supporting an `insert` method:

```
template<typename T, typename ...E>
void
multi_insert(T &t, E&&...e)
{
  // unary right fold over comma
  (void(t.insert(std::forward<E>(e))), ...);
}

int
main()
{
  std::set<int> s;
  multi_insert(s, 1, 4, 7, 10);
  for (auto i : s)
    std::cout << i << " ";
  std::cout << std::endl;
  // prints:
  // 1 4 7 10
}
```

As always, remember that folds must be parenthesized, and that a bare expression cannot be expanded as a pattern. Neither of the following alternate function bodies for `multi_insert` would compile:

```
void(t.insert(std::forward<E>(e))), ...;  // error: bad fold
t.insert(std::forward<E>(e))...;  // error: bad expansion context
```

Short-circuiting `&&` and `||` folds

Sometimes you want to iterate over a parameter pack until some condition holds. Of course, you can do this with recursion, simply ending the recursion when you hit the stop condition. However, the recursive approach can be cumbersome and require you to define several functions. As an alternative, you can fold over the logical `&&` and `||` operators, which *short-circuit* evaluation and stop doing anything the minute the condition is guaranteed to be true or false. Here's an example of a function that finds the index of the first item in a tuple to satisfy some arbitrary predicate functor `f`:

```
template<typename T, typename F>
std::size_t
tuple_find(const T &t, F &&f)
{
  return std::apply([&f](const auto &...e) {
    std::size_t r = 0;
    ((std::forward<F>(f)(e) || (++r, false)) || ...);
    return r;
  }, t);
}

int
main()
{
  std::tuple t(-2, -1, 0U, 1UL, 2ULL);
  std::cout << tuple_find(t, [](auto i) {
    return std::cmp_greater(i, -1);
  }) << std::endl;
  // prints:
  // 2
}
```

Incidentally, while this has nothing to do with parameter packs, let me gratuitously plug C++20's safe integer comparison functions. Had our lambda predicate read `return i > -1` instead of `std::cmp_greater(i, -1)`, the program would have printed 5, because `0U > -1` is false.

Often the comma operator is handy to execute some action before testing the stop condition in a fold over a logical operator. For another example of the technique, see this implementation of `operator<=>` on a tuple-like type.

Using lambda expressions to capture packs

While parameter packs can be expanded in many contexts, you sometimes need to deconstruct a template type to extract the parameter pack. This can be awkward because there are fewer contexts in which to capture a pack. Worst-case scenario, this can be done by defining a helper type or function, but this leads to a lot of code and exposes the private internals of your implementation. One way to keep things more self-contained is with a lambda expression.

Lambdas are particularly helpful in working with tuples. Combining a lambda with `std::apply` lets you capture a parameter pack corresponding to the contents of a tuple.

```
auto
tuple_mult(auto scalar, auto tpl)
{
  return apply([&scalar]<typename ...T>(T...t) {
      return std::tuple(T(scalar * t)...);
    }, tpl);
}

int
main()
{
  auto t = std::tuple(1, 2U, 4.0);
  t = tuple_mult(2, t);
  std::cout << get<0>(t) << " "
            << get<1>(t) << " "
            << get<2>(t) << std::endl;
  // prints:
  // 2 4 8
}
```

Using lambdas to capture parameter packs is especially useful with `std::integer_sequence<typename T, T...>`, a trivial type akin to `string_holder` above, but for holding an arbitrary integer type `T`. Since `std::size_t` is often what you want, the alias `std::index_sequence<T...>` is equivalent to `std::integer_sequence<std::size_t, T...>`. To create `std::integer_sequence` types, you can use the type alias `std::make_integer_sequence<T, N>` (or the `std::size_t`-specific `std::make_index_sequence<N>`) to make an integer sequence containing the numbers from `0` through `N-1`.

Suppose you want to add two tuples, element by element. You could use two nested lambdas to expand the two tuples in succession, but this would be a bit awkward. Instead, you can use `std::make_index_sequence` to get the tuple indices, then use the indices in a pattern that accesses the elements of both tuples element by element. Here is the code:

```
template<typename T>
auto
tuple_add(const T &a, const T&b)
{
  return [&a, &b]<std::size_t ...I>(std::index_sequence<I...>) {
    return std::tuple(get<I>(a) + get<I>(b)...);
  }(std::make_index_sequence<std::tuple_size_v<T>>{});
}
```

```
int
main()
{
  auto t = std::tuple(1, 2U, 4.0);
  t = tuple_add(t, tuple_mult(10, t));
  std::cout << get<0>(t) << " "
            << get<1>(t) << " "
            << get<2>(t) << std::endl;
  // prints:
  // 11 22 44
}
```

Once again, life is better in C++20 because a lambda expression can have explicit type parameters, allowing us to capture the template parameter pack `...I` from the inferred `std::index_sequence` type of the function argument.

Using lambda expressions to capture packs in requires clauses

You can also use lambdas in requires clauses. Suppose you want to define a user-defined literal `_hex` to create strings from a series of hexadecimal digits specifying bytes. For example, the constant `0x48656c6c6f21_hex` should be equivalent to `std::string{"Hello!"}`. The C++ standard says such an operator must be defined as exactly `template<char ...C> operator""_hex()`, where the template arguments are the characters of the literal. It might be nice to strip the first two characters (`'0'` and `'x'`) from the literal with different template parameters, but unfortunately C++ disallows alternate definitions such as `template<char Zero, char X, char ...C> operator""_hex()`.

Here's one possible implementation, where we simply create an array from the template arguments and iterate over the characters, skipping the first two:

```
constexpr int
hexdigit(char c)
{
  if (c >= '0' && c <= '9')
    return c - '0';
  c |= 0x20;              // convert upper- to lower-case
  if (c >= 'a' && c <= 'f')
    return c - ('a' - 10);
  return -1;              // invalid
}

template<char ...C>
requires (sizeof...(C)%2 == 0)
constexpr std::string
operator""_hex()
{
```

```
  constexpr std::array digits{ C... };
  std::string result{};
  for (std::size_t i = 2; i < digits.size(); i += 2)
    result += char(hexdigit(digits[i])<<4 | hexdigit(digits[i+1]));
  return result;
}

int
main()
{
  std::cout << 0x48656c6c6f21_hex << std::endl;
  // prints:
  // Hello!
}
```

Unfortunately, there are a few problems with this code. First, nothing requires the constant to start with `0x`. For example, you could type a decimal constant `1234_hex`, and the result would be nonsense (the first two digits skipped). We could also feed in a floating point number such as `0xa98.76p0_hex`, which is allowed by the language, and we would get nonsense. To prevent this, we could maybe sprinkle some `static_assert` statements in the code, but: A) We'd rather the `operator""_hex` function simply not match than return confusing errors from within the function, or more confusingly still, a helper function, and B) We want all *except* the second character (`x`) to be valid hex characters, so a simple `&&` fold over all the digits won't do the right thing.

Of course, this could be handled by defining custom helper types and maybe a dedicated concept, but a cleaner solution is just to unpack the parameter pack with a lambda right in the requires clause:

```
template<char ...C>
requires (sizeof...(C)%2 == 0 &&
          [](char zero, char x, auto ...rest) {
            return zero == '0' && x == 'x' && ((hexdigit(rest) != -1) && ...);
          }(C...))
constexpr std::string
operator""_hex()
{
  constexpr std::array digits{ C... };
  std::string result{};
  for (std::size_t i = 2; i < digits.size(); i += 2)
    result += char(hexdigit(digits[i])<<4 | hexdigit(digits[i+1]));
  return result;
}
```

Note how we take advantage of the fact that, as of C++17, lambdas are implicitly `constexpr` when possible. Note also that C++20 requires most string operations to be `constexpr`, in which case the above function could be `consteval` rather than merely `constexpr`. Unfortunately, as of this writing, `constexpr` strings aren't supported by the standard libraries available in common linux distributions.

By the way, here's an alternative way of implementing this user-defined literal with `string_holder` and `std::make_index_sequence`:

```
template<char ...C>
requires (sizeof...(C)%2 == 0 &&
          [](char zero, char x, auto ...rest) {
            return zero == '0' && x == 'x' && ((hexdigit(rest) != -1) && ...);
          }(C...))
consteval auto
operator""_hex()
{
  return []<std::size_t ...I>(std::index_sequence<I...>) {
    constexpr std::array digits{ C... };
    return string_holder<hexdigit(digits[2*I+2])<<4 |
                         hexdigit(digits[2*I+3])...>{};
  }(std::make_index_sequence<sizeof...(C)/2-1>{});
}
```

Using `decltype` on lambda expressions

Sometimes you want to modify the types in a parameter pack in a context where it is inconvenient to capture the parameter pack. For instance, suppose you want a way to take a `std::tuple` type and generate another type corresponding to pointers to the types in the tuple. The brute-force approach would be to introduce helper types to leverage partial specialization, but the result is rather unwieldy:

```
// Clunky idiom with helper types
namespace detail {

template<typename T> struct tuple_ptr_helper;

template<typename ...T>
struct tuple_ptr_helper<std::tuple<T...>> {
  using type = std::tuple<T*...>;
};

} // namespace detail

template<typename T> using tuple_ptrs =
  typename detail::tuple_ptr_helper<T>::type;

static_assert(std::is_same_v<tuple_ptrs<std::tuple<int, char>>,
                             std::tuple<int*, char*>>);
```

Fortunately, C++20 lets us use lambdas in unevaluated contexts, which means that instead of defining helper types, you can often use a lambda expression inside of `decltype` to achieve what you want. In this case, we can produce an entirely self-contained definition of `tuple_ptrs` as follows:

```
template<typename T> using tuple_ptrs =
  decltype(std::apply([](auto ...t) { return std::tuple(&t...); },
                      std::declval<T>()));
```

Multilambda

Thus far, we've been writing generic lambdas in functions like `tuple_add` that use overloaded syntax like the `+` operator to add numbers of different types. However, what if we want to write different (non-generic) lambdas for different types? We can implement a new variadic template type, `multilambda`, that constructs a function object comprising multiple lambdas with different type signatures. Here's the implementation:

```
template<typename ...L>
struct multilambda : L... {
  using L::operator()...;
  constexpr multilambda(L...lambda) : L(std::move(lambda))... {}
};

int
main()
{
  using namespace std::string_literals;
  std::tuple t (1, true, "hello"s, 3.0);
  constexpr multilambda action {
    [](int i) { std::cout << i << std::endl; },
    [](double d) { std::cout << d << std::endl; },
    [](bool b) { std::cout << (b ? "yes\n" : "no\n"); },
    [](std::string s) { std::cout << s.size() << " bytes\n"; },
  };
  apply([action](auto ...v) {
    (action(v), ...);  // unary right fold
  }, t);
  // prints:
  // 1
  // yes
  // 5 bytes
  // 3
}
```

`multilambda` takes a bunch of lambda expressions or other callable objects as template parameters and makes them base classes. Then, by expanding the pattern `using`

`L::operator()...;`, it brings all of the function call operators into scope, so that any of them can be called so long as there is no ambiguity. The final thing to note is that we are taking advantage of implicitly-generated class template deduction guides so as to construct a `multilambda` without having to supply explicit template parameters.

Note that you can use `decltype` on `multilambda` to do things concisely that would previously have required auxiliary structs for partial specialization. For example, here's a *concept* that checks whether a particular type is an instance of a particular template or a reference to an instance of that template:

```
template<typename T, template<typename...> typename Tmpl>
concept is_template = decltype(multilambda{
    []<typename ...U>(const Tmpl<U...> &) { return std::true_type{}; },
    [](const auto &) { return std::false_type{}; },
  }(std::declval<T>()))::value;

static_assert(is_template<std::tuple<int, long>, std::tuple>);
static_assert(is_template<const std::tuple<int, long> &, std::tuple>);
static_assert(!is_template<std::tuple<int, long>, std::variant>);
```

There's one small issue with multilambda, which is that the copy constructor might do a little work to move lambdas into the structure. You can avoid this work by eliminating the constructor and directly initializing all of the superclasses, using a deduction guide to specify that the template types should be taken from the arguments:

```
template<typename ...Lambdas>
struct multilambda : Lambdas... {
  using Lambdas::operator()...;
};
template<typename ...Lambdas>
multilambda(Lambdas...) -> multilambda<Lambdas...>;
```

Recursive types through inheritance

Sometimes you want to define a type that holds a variable number of arguments depending on a parameter pack. `std::tuple` is a good example of such a type. A good way to do this is through inheritance, using the derived class to hold one element, and the base class to hold the remaining elements. Here's an example of a "heterogeneous list" with head and tail operations:

```
template<typename ...T> struct HList;

template<>
struct HList<> {
  static constexpr std::size_t len = 0;
};

```

```
8  template<typename T0, typename ...TRest>
9  struct HList<T0, TRest...> : HList<TRest...> {
10   using head_type = T0;
11   using tail_type = HList<TRest...>;
12 
13   static constexpr std::size_t len = 1 + sizeof...(TRest);
14   [[no_unique_address]] head_type value_{};
15 
16   constexpr HList() = default;
17   template<typename U0, typename ...URest>
18   constexpr HList(U0 &&u0, URest &&...urest)
19     : tail_type(std::forward<URest>(urest)...),
20       value_(std::forward<U0>(u0)) {}
21 
22   head_type &head() & { return value_; }
23   const head_type &head() const& { return value_; }
24   head_type &&head() && { return value_; }
25 
26   tail_type &tail() & { return *this; }
27   const tail_type &tail() const& { return *this; }
28   tail_type &&tail() && { return *this; }
29 };
30 // User-defined class template argument deduction guide:
31 template<typename ...T> HList(T...) -> HList<T...>;
32 
33 template<std::size_t N> struct dummy{};
34 static_assert(sizeof(HList<dummy<0>, dummy<1>, dummy<2>>) == 1);
35 static_assert(sizeof(HList<dummy<0>, dummy<0>, dummy<0>>) == 3);
```

The basic idiom may already be familiar to the reader, but a few things are worth pointing out for people less familiar with C++20 and C++17. First, note the use of the attribute `[[no_unique_address]]` on line 14. Without this attribute, the size of `HList<dummy<0>, dummy<1>, dummy<2>>` would be 3 bytes instead of 1 on line 34. Why? Because C++ requires most objects (other than bit fields) to have a unique address. However, there's long been an exception called the empty base optimization (EBO). Roughly speaking, given the following types:

```
struct Base {}; // empty

struct Derived : Base {
  /* ... unspecified ... */
};
```

EBO states that `Base` need not increase the size of `Derived` so long as the first data member in `Derived` doesn't also start with `Base`. In other words, `Base` is allowed to share the same address as any other object not of type `Base`. Before C++20, implementers of types such as `std::tuple` jumped through quite a few hoops to exploit EBO to reduce tuple sizes. Now, however, we can use the `[[no_unique_address]]` attribute to apply the same logic as EBO to any structure field, not just the base class.

Note that despite this otimization, `HList<dummy<0>, dummy<0>, dummy<0>>` still has size 3 bytes, as seen on line 35, because each instance of the same type `dummy<0>` still needs a unique address. This makes a certain amount of sense. For example, the constructor of `dummy<0>` might decide to enter the object's address in some global hash table, so while `dummy<1>` and `dummy<0>` would presumably have different hash tables, entering the same address twice into `dummy<0>`'s hash table could lead to confusion.

Another detail worth noting is the use of a user-defined deduction guide on line 31. This allows us to construct an `HList` without explicit template parameters, as in `HList{1, 2, 3, "hello"}`. The deduction guide is required because `HList::HList` takes arguments by forwarding reference, yet we need to make sure to infer non-reference types for the template parameters.

`HList` lets us explore another good example application of `std::make_index_sequence`, as well as of our `is_template` concept above. Suppose we want to implement an `apply` function analogous to `std::apply`. We can do this by implementing a `get` function, then capturing an integer sequence of list indices to expand in `apply`:

```
template<std::size_t N, is_template<HList> HL>
requires (N <= std::remove_cvref_t<HL>::len)
inline decltype(auto)
drop(HL &&hl)
{
  if constexpr (N)
    return drop<N-1>(std::forward<HL>(hl).tail());
  else
    return std::forward<HL>(hl);
}

template<std::size_t N, is_template<HList> HL>
requires (N < std::remove_cvref_t<HL>::len)
inline decltype(auto)
get(HL &&hl)
{
  return drop<N>(std::forward<HL>(hl)).head();
}

template<typename F, is_template<HList> HL>
decltype(auto)
apply(F &&f, HL &&hl)
{
  [&f,&hl]<std::size_t ...I>(std::index_sequence<I...>) -> decltype(auto) {
    return std::forward<F>(f)(get<I>(std::forward<HL>(hl))...);
```

```
  }(std::make_index_sequence<std::remove_cvref_t<HL>::len>{});
}
```

Homogeneous function parameter packs

As previously mentioned, it is tricky to emulate a function taking a homogeneous variadic parameter pack without introducing extra copies of the argument. Suppose you want to implement a function equivalent to the (illegal) `good(Obj ...obj)` that accepts a variable number of arguments all of type `Obj` by value (meaning modification of the arguments in the function does not affect the calling context). We can of course write `good(std::convertible_to<Obj> auto&& ...t)`, but now we will capture a heterogeneous set of arguments. Some of these arguments may be values from which we need to construct an `Obj`, but others may already be an `Obj` constructed specifically for this function, as when someone calls `good(Obj{})`. In the latter case, we want to avoid constructing a *second* `Obj` from the one that was passed in as an argument. On the other hand, if the user called `good(obj)` where `obj` is a variable containing an existing `Obj`, then we must copy `obj`, since the call-by-value semantics we want require that modifications of argument variables inside `good` do not affect the variables passed in.

When a new temporary `Obj` has been constructed as a function call argument, the inferred type of the argument will be the rvalue reference `Obj&&`. When an existing `obj` is being passed in, the type will be the lvalue reference `Obj&` (or possibly `const Obj&`). The trick is to define a function `local_copy` that will generate a new `Obj` in the later case, but return a reference to an existing temporary `Obj` in the former case. Here is the code:

```
struct Obj { void use() { /* ... */ } /* ... */ };

template<typename Want, typename Have>
inline std::conditional_t<std::is_same_v<Want, Have>, Want &&, Want>
local_copy(Have &in)
{
  return static_cast<Have&&>(in);
}

template<std::convertible_to<Obj> ...T>
void
good1(T&&...t)
{
  // Unary fold over comma operator
  (local_copy<Obj, T>(t).use(), ...);
}

// Another way to do it
template<std::convertible_to<Obj> ...T>
void
good2(T&&...t)
```

```
{
  auto use = []<typename U>(U &&arg) {
    decltype(auto) o = local_copy<Obj, U>(arg);
    o.use();
  };
  (use(std::forward<T>(t)), ...);
}
```

To understand the implementation and use of `local_copy`, you need to know a bunch of things about C++ value categories:

- When a function argument is a **forwarding reference**, as in `template<typename T> void f(T &&t)`, the type `T` will be an lvalue reference (e.g., `Type&`) if `f` was invoked with an lvalue (e.g., some variable `obj`) and a non-reference (e.g., `Type`) if `f` was invoked with an rvalue (e.g., `Obj{}`).

- When you take a reference type `TR` and add an rvalue reference, you get back `TR`, meaning `TR&&` and `TR` are always the same type for a reference. When you add an lvalue reference to reference type `TR`, you always get back an lvalue reference, meaning `TR&` is always the same as `std::remove_reference_t<TR>&`. This is known as **reference collapsing**, and it explains why, when an lvalue is passed to forwarding reference `f(T&&t)`, `T` can be inferred as an lvalue reference.

- When treated as an expression, a variable `v` is always an lvalue, regardless of whether `v` was declared as a non-reference `T`, an lvalue-reference `T&`, or an rvalue-reference `T&&`. Hence, if you want a forwarding reference to be inferred as something other than an lvalue reference, you need to pass something other than a variable, such as a function call result (e.g., `std::forward<T>(t)`) or a cast (`static_cast<T&&>(t)`).

- An expression that is an invocation of a function returning a non-reference, non-`void` type is of a category known as a *prvalue*. Since C++17, you can think of a prvalue as a set of instructions for how to create an object that has not yet been created. Hence, invoking `use_object` below creates only one `Obj`, namely `o`, because what `make_object()` returns is conceptually just a recipe saying, “Please initialize some `Obj` with the default initializer `{}`.”

  ```
  // Doesn't copy or move an Obj, but requires the invoking
  // context to create a default-initialized Obj
  Obj make_object() { return {}; }

  // Only one Obj is ever created per invocation
  void use_object() { Obj o = make_object(); o.use(); }
  ```

  This behavior is sometimes known as *mandatory copy elision*. Another way to think about it is that even if the code for creating an `Obj` resides within the generated code for `make_object`, the compiler must arrange for the new `Obj` to reside in the stack frame that belongs to the calling function (`use_object`) so that the `Obj` isn't destroyed on return from `make_object`.

- When you declare a variable as `auto v = expression`, the type deduction rules are the same as for templates—meaning `v`'s type will be the non-reference type `T` inferred when invoking the template function `template<typename T> void f(T t)` with the same expression, i.e., `f(expression)`.
- When you declare a variable as `decltype(auto) v = expression`, the type of `v` will be exactly `decltype(expression)`. `decltype` is confusing for the uninitiated, as the single keyword does two totally different things, often depending on something as trivial as extra parentheses (details here). However, when `decltype` is applied to a function call expression, the type is always exactly the return type of the function, which can be a non-reference, lvalue-reference, or rvalue-reference.

In `good1`, we need to construct a new `Obj` from the reference function argument in all cases except when the argument was a temporary object (i.e., `good1(Obj{})`). When the argument was a temporary, the inferred type `T` is `Obj` (meaning argument `t` has type `Obj&&`). If the argument was a variable of type `Obj`, then `T` will be inferred as `Obj&`, and we need to copy it to avoid modifying it. If the variable was anything other than an `Obj`, we need to construct an `Obj` from it.

The goal of `local_copy` is to construct a new object of type `Want` from an argument of type `Have` unless the argument is already a temporary object of type `Want`, in which case it should just return a reference to the existing temporary object. The return type of the function uses `std::conditional_t` to distinguish these two cases—it returns a new `Want` (meaning a prvalue) unless `Want` and `Have` are the same type, in which case it returns an rvalue reference to its input object.

Notice that `local_copy` takes a `Have&` and not a `Have&&`, because it is intended to be used with variables, whose expression type will always be an lvalue and hence cannot be bound to an rvalue reference. (We explicitly specify the type of `Have` when invoking `local_copy`, so it can't be a forwarding reference.) `local_copy` casts its return value to `Have&&`, which (through reference collapsing) is the same as `Have` when `Have` is an lvalue reference. It's also possible that the argument was a temporary object of type other than `Obj`, in which case it is returned as an rvalue reference from which the return `Obj` is constructed.

There are still a few cases in which `good1` behaves differently from the hypothetical homogeneous `good(Obj ...arg)`. If you call `good1(std::move(obj))`, no new `Obj` will be constructed from the argument, whereas ordinarily you would expect to move-construct a new `Obj`. Arguably this is a feature; since a moved object is left in an unspecified state, the main differences in semantics will be one fewer invocation of the move constructor `Obj::Obj(Obj&&)` (or copy constructor if `Obj` lacks a move constructor). Another difference is that you can call `good({})` and the `{}` will construct an `Obj`, since the argument type is known, whereas `good1({})` is illegal because the compiler doesn't know the argument type.

By the way, if this all seems too complex, there is sometimes an alternative to variadic functions for homogenous argument lists—you can write functions to take an `initializer_list`:

```
inline void
almost_good3(std::initializer_list<Obj> args)
```

```
{
  for (const Obj &o : args)
    o.use(); // doesn't work with non-const method, though
}
```

The two disadvantages are that you'll have to call the function with an extra set of braces, as in `alsmost_good3({obj, Obj{}})`, and that you only have `const` access to members of `std::initializer_list`, so it won't work with our example `Obj` type, in which the `use()` method is not `const`.

Array of function pointers

Sometimes you need to convert a runtime constant into a compile-time constant without manually writing out every possible value in a `switch` statement. The best way to do this is to initialize an array of function pointers using a parameter pack expansion.

As an example, suppose you wish to serialize a `std::variant` `v`. You can serialize the current type of the variant by serializing the number `v.index()`. Then you can serialize the body of the variant with `std::visit`. To deserialize `v`, you must reverse the process. First, deserialize the index value `i`. Then set `v.index()` to `i` by setting `v` to contain its `i`th type. Finally, deserialize the contents with `std::visit`.

Unfortunately, the problem with this plan is that the inverse of `v.index()`—a method to *set* the index of the variant—does not exist. We could try to implement such a function by brute-force, but the result will be problematic:

```
// Painful to write, only works if variant has exactly 3 types
void
set_index(auto &v, std::size_t n)
{
  switch (n) {
  case 0:
    v.template emplace<0>();
    break;
  case 1:
    v.template emplace<1>();
    break;
  case 2:
    v.template emplace<2>();
    break;
  }
}
```

The less serious problem with this code is that it is very tedious to write. Even though `emplace` really is a different function for each index of the variant, writing it out this way is painful. The more serious problem is that the above function only works for variants with exactly 3 cases. A variant with only 2 cases doesn't have valid code for `emplace<2>`, so invoking `set_index` on such a type will fail to compile.

What we need is a way to take a runtime constant and convert it to a compile time constant we can supply as a template parameter to `v.emplace<>()`. We can represent compile-time constants with the template type `std::integral_constant`, which has a `constexpr` conversion operator returning the integral value represented by the type. Conceptually, we would like to do something like this:

```
// Parse a pack of char as a decimal number
constexpr std::size_t
chars2size(std::same_as<char> auto ...c)
{
  std::size_t r = 0;
  for (std::size_t i : { c... })
    r = r*10 + i - '0';
  return r;
}

// Define 0_const, 1_const, etc. as compile-time integral constants
// (Note: doesn't work with clang++ yet)
template<char ...C> requires ((C >= '0' && C <= '9') && ...)
consteval std::integral_constant<std::size_t, chars2size(C...)>
operator ""_const()
{
  return {};
}

// Illegal nonsense--a function can only return one type
auto
get_constant(std::size_t n)
{
  switch (n) {
  case 0:
    return 0_const;
  case 1:
    return 1_const;
  // ...
  }
}
```

Of course, that doesn't make sense because `0_const` and `1_const` are of different types. A function can only return one type (which obviously can't depend on a runtime argument). In order for a function to produce a type dependent on a runtime parameter, instead of returning the type, the function needs to call an overloaded function object. We therefore must implement something like the following:

```
template<typename R = void, typename F>
```

```
inline constexpr R
with_constant(std::size_t n, F &&f)
{
  switch (n) {
  case 0:
    return std::forward<F>(f)(0_const);
  case 1:
    return std::forward<F>(f)(1_const);
  // ...
  }
}

void
set_index(auto &v, std::size_t n)
{
  with_constant(n, [&v](auto i) { v.template emplace<i>(); });
}
```

Okay, so now we know what the function should look like, but we still have the problem that it only works for a fixed number of values of `n`. Not even a fixed maximum, but a fixed number. However, we can fix this by taking the maximum value as a parameter, then using `std::make_index_sequence` to generate a parameter pack that we expand into an array of function pointers.

```
namespace detail {

template<std::size_t I, typename R, typename F>
inline constexpr R with_integral_constant(F f)
{
  return static_cast<F>(f)(std::integral_constant<std::size_t, I>{});
}

} // namespace detail

template<std::size_t N, typename R = void, typename F>
inline constexpr R
with_n(int n, F &&f)
{
  constexpr auto invoke_array =
    []<std::size_t...I>(std::index_sequence<I...>) {
    return std::array{ detail::with_integral_constant<I, R, F&&>... };
  }(std::make_index_sequence<N>{});

  return invoke_array.at(n)(std::forward<F>(f));
}

```

```
template<typename T> requires requires {
  { std::variant_size_v<T>+0 } -> std::same_as<std::size_t>;
}
void
set_index(T &t, std::size_t n)
{
  with_n<std::variant_size_v<T>>(n, [&t](auto i) {
    t.template emplace<i>();
  });
}
```

Some notes on the above code. First, note that `invoke_array`, because it is an array, must have all its elements be of the same type. Hence, we really do need `detail::with_integral_constant` to be a function, rather than a lambda expression, to ensure all elements of the array have the same type. We could create an array of `std::function<R(F&&)>`, but `std::function::~function` is not a `constexpr` destructor, so then `invoke_array` could not be `constexpr`, which might inhibit some compiler optimizations. (Note in some cases, if you have non-generic lambdas with no variable capture, they can be converted to ordinary function pointers by prefixing the lambda with `+`, as in `+[]{}`.)

Next, note that we supply the number of possible values, `N`, as the first template argument to `with_n`, and this argument is mandatory. By using `std::array::at`, we ensure that an exception will be thrown for out-of-bounds values of `n`. Note also that if we want to use `with_n` with a return type other than `void`, it must be supplied as the second template argument `R`. Since the code is different for each value of `n`, we can't infer a return type. (Possibly we could do something fancy to compute a plausible return type using `std::common_type_t`, but it doesn't seem worth the complexity.)

As a reminder about constraints, the clause `{ std::variant_size_v<T>+0 } -> std::same_as<std::size_t>` is known as a *compound requirement*, and states that `decltype((○))` of the expression in braces must satisfy the constraint to the right of the arrows. Since the expression `(std::variant_size_v<T>)` is probably an lvalue of type `const std::size_t&`, we just add 0 to turn it into a prvalue of type `std::size_t`, avoiding any worries about the const or references. An alternative would be to use our `is_template` concept, as in:

```
template<is_template<std::variant> T>
void
set_index(T &t, std::size_t n)
{
  /* ... */
}
```

Finally, note the importance of our lambda in `set_index` accepting the parameter by value (`auto i`) and not by reference (`auto &&i` or `const auto &i`). Because of some quirkiness in how `constexpr` works, you can't call `constexpr` methods (such as `std::integral_constant::operator std::size_t`) on a reference at compile time, only on a value. Hence, line 30 will cause a compilation error if `i` is a reference.

Conclusion

As C++ evolves, many old, error-prone, and frankly disgusting idioms are no longer necessary and should be eliminated. For instance, you should never again rely on SFINAE now that we have concepts. Similarly, it's time to update our variadic function idioms to leverage the new language features. Concepts now allow us to restrict the types in a function parameter pack. `constexpr`-by-default lambdas help us capture parameter packs even in contexts such as requires clauses. The ability to use lambdas in unevaluated contexts lets us avoid helper classes for many purposes. Folds and `if constexpr` can save us from implementing multiple overloaded functions for recursive and base cases. Class template argument deduction makes it possible to introduce types such as `multilambda` without helper functions to create them.

Unfortunately, effective use of variadic templates still involves some unwieldy idioms, in large part because of the asymmetry between expanding patterns (which works in many contexts) and capturing parameter packs (which often requires introducing otherwise unnecessary lambda expressions). If C++ adopts parameter packs in structured bindings, this will go a long way towards further simplifying the use of parameter packs. Another nice feature would be homogeneous variadic function parameters, though unfortunately that was rejected by the committee, so won't be adopted in current form. Still, perhaps a future version of the standard could at least deprecate omitting the comma in declarations of old-style varargs functions, which is the main impediment.

I hope the idioms I presented are useful to you. If you are still writing C++17-compatible code, I hope this post provides further motivation for you to abandon legacy compatibility and embrace the significant improvements in C++20.