VARIADIC TEMPLATES

Plan for Today

- Variadic Function Templates
- Understanding `constexpr`
- Fold Expressions
- Variadic Class Templates

Variadic Templates: Introduction

- Having to deal with unknown number of elements in type-safe fashion is common problem
  - Error reporting function may take between zero and ten arguments of varying types
  - Logging function may take unknown number of arguments of unknown types
  - Matrix may have between one and ten dimensions
  - Tuple can have zero to ten elements of varying types
- Templates that can take varying number of arguments of possibly differing types

Templates: Basic Terminology

function parameter

template type parameter

function parameter type

```
template <typename T> void f(ParamType param);

// call f with some expression
f(expr);
```

function argument

type of function argument is template type argument

Variadic Template: Definition

- Template whose definition captures a *parameter pack* in its template and function parameters
  - *Parameter pack* means "zero or more elements of something" and is represented by token `...`
  - `typename ...` captures list of template type parameters in *template parameter pack* `Types`
  - In variadic function template, `Types ...` captures (type/value) list of function parameters in *function parameter pack* `params` with each value in list having type in corresponding template parameter pack

```
// Types is template parameter pack
// params is function parameter pack
template <typename ...Types>
void variadic_template(Types ...params) {
  // statements ...
}
```

Variadic Templates: 1st Example


```
template <typename ...Types>
void f(Types&& ...params) {
  // empty function
}

template <typename ...Types>
class C {
  // empty class
};
```

```
f();  // f<>();
f(1); // f<int>(1);
f(1, "hello", std::string{"world"}, 3.14); // ???

C<> c0;
C<int> c1;
C<int, char const*, std::string, double> c2;
```

Variadic Templates: 2nd Example

- We can specify type parameter to disallow zero template arguments in previous example

```
template <typename T, typename ...Types>
void f(T&& t, Types&& ...params) { /* empty function */ }

template <typename T, typename ...Types>
class C { /* empty class */ };
```

```
f();  // error!!!
f(1); // empty pack
f(1, "hello"); // one parameter in pack
f(1, "hello", std::string{"world"}); // two parameters in pack
f(1, "hello", std::string{"world"}, 3.14); // 3 parameters in pack

C<> c0;     // error!!!
C<int> c1; // empty pack
C<int, char const*, std::string, double> c2;
```

sizeof... Operator (1/2)

- Queries number of elements in parameter pack

```
template <typename ...Types>
void f(Types ...params) {
  std::cout << "Number of parameters: "
            << sizeof...(Types) << '\n';
}

f();  // params has zero parameters
f(1); // params has 1 parameter: int
f(2,1.0); // params has 2 parameters: int,double
f(2,1.0,"hello"); // params has 3 parameters:
                  // int, double, char const*
```

sizeof... Operator (2/2)


```
template <typename ...Types>
struct C {
  auto size() const {
    return sizeof...(Types);
  }
};

C<> c0;
std::cout << c0.size() << "\n"; // prints 0
C<int> c1;
std::cout << c1.size() << "\n"; // prints 1
C<int, double> c2;
std::cout << c2.size() << "\n"; // prints 2
C<int, double, char const*> c3;
std::cout << c3.size() << "\n"; // prints 3
```

Pack Expansion: Function Call (1/12)

- Only other thing we can do with parameter pack is to *unpack/expand* it using a *pattern*
  - Unfortunately, parameter packs can only be expanded in *function calls* and *initializer lists*
  - Expansion separates pack into its constituent parts, applying *pattern* to each unpacked element
  - Simplest pack expansion is triggered by putting pack to left of `...` in function call resulting in comma-separated list of pack elements

```
template <typename T, typename ...Types>
void print(T t, Types ...params) {
  std::cout << t << "\n"; // print first parameter
  print(params...); // call print for remaining parameters
}
```

Pack Expansion: Function Call (2/12)

- Print variable number of arguments of different types

```
void print() {} // base function to end recursive
                // function template instantiation ...
template <typename T, typename ...Types>
void print(T t, Types ...params) {
  std::cout << t << "\n"; // print first parameter
  print(params...); // call print for remaining parameters
}

print(7.5);
print(7.5, "hello");
print(7.5, "hello", 7, std::string{"world"});
print(7.5, "hello", 7);
```

1st argument is `t` and remaining arguments bundled into parameter pack `params` for later use

Pack Expansion: Function Call (3/12)


```
void print() {} // base function to end recursive
                // function template instantiation ...

template <typename T, typename ...Types>
void print(T t, Types ...params) {
  std::cout << t << "\n";
  print(params...);
}

print(7.5, "hello", 7);
```

Parameter pack `params` is *unpacked* or *expanded* so that 1st element of `params` is selected as `t` and `params` is one element shorter than in previous call.

This carries on until `params` is empty, so that we call `void print();`

Pack Expansion: Function Call (4/12)


```
void print() {}

template <typename T, typename ...Types>
void print(T t, Types ...params) {
  std::cout << t << "\n";
  print(params...);
}
```

Call	t	params...
`print(7.5, "hello", 7)`	`7.5`	`"hello", 7`
`print("hello", 7)`	`"hello"`	`7`
`print(7)`	`7`	Because of empty parameter pack, nonvariadic version of `print` is called

Pack Expansion: Function Call (5/12)

- Previous example can also be implemented as overload of variadic and nonvariadic templates:

```
template <typename T>
void print(T t) {
  std::cout << t << '\n';
}

template <typename T, typename ...Types>
void print(T t, Types ...params) {
  print(t);          // call print() for first parameter
  print(params...); // call print() for remaining params
}

print(7.5, "hello", 7);
```

If 2 function templates only differ by trailing parameter pack, function template without trailing parameter pack is preferred!!!

Pack Expansion: Function Call (6/12)


```
template <typename T>
void print(T t) {
  std::cout << t << '\n';
}

template <typename T, typename ...Types>
void print(T t, Types ...params) {
  print(t);          // call print() for first parameter
  print(params...); // call print() for remaining params
}
```

Call	t	params...
`print(7.5, "hello", 7)`	`7.5`	`"hello", 7`
`print("hello", 7)`	`"hello"`	`7`
`print(7)` [nonvariadic]		

Pack Expansion: Function Call (7/12)

- Previous example can be generalized to any output stream:

```
template <typename T>
std::ostream& print( std::ostream& os, T const& t) {
  return os << t << '\n';
}

template <typename T, typename ...Types>
std::ostream& // return type
print(std::ostream& os, T const& t, Types const& ...params) {
  os << t << ", ";  // print first argument in call
  return print(os, params...); // print remaining arguments in call
}

print(std::cout, 7.5, "hello", 7);
```

Pack Expansion: Function Call (8/12)

- Another example illustrating recursion pattern …

```
template <typename T> T sum(T t) { return t; }

template <typename T, typename ...Types>
T sum(T t, Types ...params) { return t + sum(params...); }

std::cout << sum(1, 2, 3, 4, 5) << '\n';
```

Call	t	params...
sum(1, 2, 3, 4, 5)	1	2, 3, 4, 5
sum(2, 3, 4, 5)	2	3, 4, 5
sum(3, 4, 5)	3	4, 5
sum(4, 5)	4	5
sum(5) [nonvariadic]		

Pack Expansion: Function Call (9/12)

- Helpful convention to avoid confusion with `...`
  - Put parameter into which you *pack* on *right side* of `...`
  - Put parameter that is to be *unpacked* on *left side* of `...`
- See how this convention applies to types and objects of `sum` function
  - `typename ...Types`: pack multiple template type arguments into template type parameter `Types`
  - `<Types ...>`: unpack `Types` when instantiating class or function template
  - `Types ...params`: Pack multiple function arguments into variable pack `params`
  - `sum(params...)`: Unpack variable pack `params` as comma-separated values and call `sum` with these values as multiple arguments

Pack Expansion: Function Call (10/12)

- More complicated patterns are possible when expanding function parameter pack
- When expansion pack appears inside function call operator, largest expression to left of ... is pattern that is applied!!!

Pack Expansion: Function Call (11/12)

- In function call operator, pattern that is expanded is largest expression or brace initialization list to left of ...

```
template <typename T>
T incr(T x) { return x+1; }

template <typename T>
T sum(T t) { return t; }

template <typename T, typename ...Types>
T sum(T t, Types ...params) {
  return t + sum(incr(params)...);
}

// equivalent to: sum(incr(1), incr(2), incr(3), incr(4))
sum(1, 2, 3, 4);
```

Pack Expansion: Function Call (12/12)

- Parameter packs can unfortunately only be expanded in function calls and initializer lists

```
// expansion of parameter pack Types on sizeof operator
// in an initializer list ...
template <typename ...Types>
auto create_array() {
  return std::array<std::size_t, sizeof...(Types)>
                                {sizeof(Types)...};
}

auto x = create_array<int, double, std::string, float>();
for (auto y : x) {
  std::cout << y << ' '; // ???
}
```

Variadic Function Templates: Recursion (1/6)

- Implementation of variadic templates is typically thro' recursive "first"/"last" manipulation

```
// do something to 1st parameter and then recurse
// with rest of parameters
template <typename T,           // type of 1st parameter
          typename ...Tail>     // types of the rest
void f(T const& head,           // 1st parameter
       Tail const& ...tail) { // rest of parameters
  g(head);    // do something to 1st parameter
  f(tail...); // repeat with rest of parameters
}
```

Variadic Function Templates: Recursion (2/6)

- Here, we do something with 1st parameter head by calling g():

```
// write parameter to output stream
template <typename T>
void g(T const& t) {
  std::cout << t << ' ';
}
```

Variadic Function Templates: Recursion (3/6)

- Then, `f()` is called recursively with rest of parameters in parameter pack `tail...`

```
// do something to 1st parameter and then recurse
// with rest of parameters
template <typename T,          // type of 1st parameter
          typename ...Tail>   // types of the rest
void f(T const& head,          // 1st parameter
       Tail const& ...tail) { // rest of parameters
  g(head);     // do something to 1st parameter
  f(tail...); // repeat with rest of parameters
}
```

Variadic Function Templates: Recursion (4/6)

- Eventually, `tail` parameter pack will become empty
- Need a separate function to deal with it:

```
void f() { } // do nothing
```

Variadic Function Templates: Recursion (5/6)


```
// nonvariadic function must be declared
// before variadic function
template <typename T>
void g(T const& t) {
  std::cout << t << ' ';
}

void f() { } // do nothing

template <typename T, typename ...Tail>
void f(T const& head, Tail const& ...tail) {
  g(head);     // do something to 1st parameter
  f(tail...); // repeat with tail
}
```

Variadic Function Templates: Recursion (6/6)

- In call `f(0.3,'c',1)` recursion will execute as follows:

Call	`head`	`tail`
`f<double,char,int>(0.3,'c',1)`	`0.3`	`'c',1`
`f<char,int>('c',1)`	`'c'`	`1`
`f<int>(1)`	`1`	empty

Variadic Function Templates: Summary

- Provide type-safety and *extensibility to user-defined types* in contrast to variadic functions implemented thro’ `<cstdarg>`
- Performance
  - No actual recursion at runtime
  - Instead, sequence of function calls are pre-generated at compile time
  - With aggressive inlining, compilers can remove runtime function calls
  - In contrast, variadic functions using `<cstdarg>` involve manipulation of runtime stack

Forwarding Parameter Packs


```
template <typename T, typename ...Types>
std::unique_ptr<T> factory(Types&& ...params) {
  return std::make_unique<T>(std::forward<Types>(params)...);
}

struct C {
  C(int&, double&, double&&) {}
  friend std::ostream& operator<< (std::ostream& os, C const&) {
    return << os << "C";
  }
};
```

```
std::unique_ptr<double> pd = factory<double>(11.89);
std::cout << "*pd: " << *pd << "\n";
std::unique_ptr<int> pi = factory<int>(17);
std::cout << "*pi: " << *pi << "\n";
std::unique_ptr<std::string> ps =
                        factory<std::string>("hello world");
std::cout << "*ps: "  << *ps << "\n";
std::unique_ptr<C> pc = factory<C>(i, d, 3.14);
std::cout << "*pc: "  << *pc << "\n";
```

From const to constexpr (1/3)

- Prior to C++11, const machinery was restricted to two things:
  - Qualifying a type as const, and thus any instance of that type is immutable
  - Qualifying a nonstatic member function so that *this is const in its body

```
class int_wrapper {
public:
  explicit int_wrapper(int);
  void mutate(int);
  int inspect() const;
private:
  int mi;
};
```

```
const int_wrapper iwc{7};
iwc.mutate(5);          // error
int i = iwc.inspect(); // ok
```

From `const` to `constexpr` (2/3)

- Values known during compilation are privileged especially *integral constant expressions*
  - Array sizes, integral template arguments, lengths of `std::array` objects, enumerator values, alignment specifiers, …
  - Mathematical constants …
- `constexpr` object is like `const` object that has values known at compile time

From `const` to `constexpr` (3/3)


```
int sz; // non-constexpr variable

// error: sz's value not known at compilation
constexpr auto arr_sz1 = sz; // ok???
// error: same problem
std::array<int, sz> a1; // ok???

// fine, 10 is a compile-time constant
constexpr auto arr_sz2 = 10; // ok???

// fine, arr_sz2 is constexpr
std::array<int, arr_sz2> a2; // ok???
```

Difference Between `const` and `constexpr` (1/2)

- `const` doesn't offer same guarantee as `constexpr`
  - `const` objects need not be initialized with values known during compilation

```
int sz; // non-constexpr variable

// fine: arr_sz is const copy of sz
const auto arr_sz = sz; // ok???

// error: arr_sz's value not known at compilation
std::array<int, arr_sz> data; // ok???
```

Difference Between `const` and `constexpr` (2/2)

- Simply put, all `constexpr` objects are `const`, but not all `const` objects are `constexpr`
- If you want compilers to guarantee that a variable has a value that can be used in contexts requiring compile-time constants, use `constexpr`, not `const`!!!

Header-Only Libraries: Inlined Variables (1/4)

- C++17 introduced `inline` variables to allow for header-only libraries with variable definitions in header file
  - ODR not invoked when header file is included by many source files or multiple times by same source file
  - Instead, all source files including that header file will have same address for `inline` variable

```
// possibly defined in multiple header files
inline long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double const& r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (2/4)

- `constexpr` [and `const`] objects can be defined in header files
  - By default, such objects have static or internal linkage

```
// possibly defined in multiple header files
constexpr
long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (3/4)

- If you require address of constant to be same everywhere, you mark it as `inline`

```
// possibly defined in multiple header files
inline constexpr
long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double const& r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (4/4)

- C++17 allows static data members to be defined and initialized in class

```
struct Counter {
  // static data member is now defined and initialized
  // in-class without the need to provide definition in
  // source file
  static inline int counter = 0;

  Counter()  { ++counter; }
  ~Counter() { --counter; }
};
```

Variable Templates (1/5)

- Since C++14, variables can be parameterized by specific type

```
// can be defined in a header file
template <typename T>
constexpr T pi{3.141'592'653'589'793'238'462'643'383'279L};

template <class T>
T circ_area(T const& r) {
  return pi<T>*r*r;
}

std::cout << pi<long double> << '\n';
std::cout << pi<double>      << '\n';
std::cout << pi<float>       << '\n';
```

Variable Templates (2/5)

- Variables templates can also have default template arguments

```
template <typename T = long double>
constexpr T pi = T{3.141'592'653'589'793’238L};

std::cout << pi<> << '\n'; // outputs a long double
std::cout << pi<float>  << '\n'; // outputs a float
```

Variable Templates (3/5)

- Variables templates can also be parameterized by nontype parameters

```
// array with N elements, zero-initialized
template <int N> std::array<int,N> arr{};

// nontype parameter used to parameterize initializer
template <auto N> constexpr decltype(N) dval = N;

// N has value 'c' of type char
std::cout << dval<'c'> << '\n';

// set first element of global object arr
arr<5>[0] = 42;
```

Variable Templates (4/5)

- Useful application of variable templates is to define variables that represent members of class templates

```
// given definition of class C
template <typename T>
class C {
public:
  static constexpr int max{100};
};

// you can define variable template my_max:
template <typename T> int my_max = C<T>::max;

// so that you can define different values for
// different specialization of C<>:
auto sc = my_max<std::string>; // instead of C<string>::max
```

Variable Templates (5/5)


```
// better example ...
// for standard class such as
namespace std {
  template <typename T> class numeric_limits {
  public:
    ...
    static constexpr bool is_signed = false;
    ...
  };
}

// you can define variable template
template <typename T>
constexpr bool is_signed = std::numeric_limits<T>::is_signed;

// to be able to write expression
is_signed<char>
// rather than lengthier expression
std::numeric_limits<char>::is_signed
```

`constexpr` Functions

- Functions that produce compile-time constants *when they're called with compile-time constants*
  - `constexpr` functions can be used in contexts that demand compile-time constants
  - Acts like normal function computing its result at runtime when called with one or more values that are not known during compilation

`constexpr` Functions

`constexpr` in front of `fibonacci` doesn't say that `fibonacci` returns a `const` value, it says that if `n` is compile-time constant, `fibonacci`'s result may be used as compile-time constant.
If `n` is not compile-time constant, `fibonacci`'s result will be computed at runtime.

```
constexpr long fibonacci(long n) {
  return n <= 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

constexpr Functions


```
template <typename T>
constexpr T square(T x) noexcept {
  return x*x;
}
```

```
constexpr
int pow(int base, int exp) noexcept {
  int result{1};
  for (int i{}; i < exp; ++i) result *= base;
  return result;
}
```

constexpr Functions

- constexpr functions can be used in places with compile-time contexts

```
constexpr
int pow(int base, int exp) noexcept {
  int result{1};
  for (int i{}; i < exp; ++i) result *= base;
  return result;
}

// 5 conditions each with 3 possible states
constexpr int conds {5}, states{3};
std::array<int, pow(states, conds)> results;
```

constexpr Functions

- Another example of constexpr functions used in places with compile-time contexts

```
template <typename T1, typename T2>
constexpr
auto Max(T1 a, T2 b) -> decltype(b<a?a:b) {
  return b < a ? a : b;
}

int ai[Max(sizeof(int), 10L)] {1,2,3};

std::array<std::string, Max(sizeof(int), 8L)>
                                 as{"a","b","c"};
```

constexpr Functions

- Another example …

```
constexpr bool is_prime(uint64_t p) {
  for (uint64_t d{2}; d <= p/2; ++d) {
    // found divisor without remainder
    if (p % d == 0) return false;
  }
  // no divisor without remainder found
  return p > 1;
}

bool found =
is_prime(std::numeric_limits<uint64_t>::max());
```

constexpr Functions

Compiles with g++ but not with clang++!!!

```
constexpr long floor_sqrt(long n) {
  return floor(sqrt(n));
}
```

`constexpr` Functions for User-Defined Types

- `constexpr` functions are limited to taking and returning literal types [types that can have values determined during compilation]
  - All built-in types except `void` qualify
- User-defined types can be literal too …

constexpr Functions for User-Defined Types


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}

  // other stuff ...
};
```

`constexpr` Functions for User-Defined Types


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}

  // other stuff ...
};

// compiler will run constexpr ctor
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);
```

constexpr Functions for User-Defined Types


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  // other stuff ...
};
```

constexpr Functions for User-Defined Types


```
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);
constexpr Point mid = midpt(p1, p2);
```

```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  // other stuff ...
};

constexpr
Point midpt(Point const& p1, Point const& p2) noexcept {
  return { (p1.X()+p2.X())/2.0, (p1.Y()+p2.Y())/2.0 };
}
```

constexpr Functions for User-Defined Types


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  constexpr void X(double dx) noexcept { x = dx; }
  constexpr void Y(double dy) noexcept { x = dy; }
};
```

constexpr Functions for User-Defined Types


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  constexpr void X(double dx) noexcept { x = dx; }
  constexpr void Y(double dy) noexcept { x = dy; }
};
```

```
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);
constexpr Point mid = midpt(p1, p2);
constexpr Point rmid = reflection(mid);
```

```
constexpr Point reflection(Point const& p) noexcept {
  Point result;
  result.X(-p.X());
  result.Y(-p.Y());
  return result;
}
```

Folding

- Higher-order function that abstract process of iterating over recursive structures such as vectors, lists, trees, ... and lets you gradually build required result
- Called `std::accumulate` in C++ standard library
- Picture required

Compile-Time `if`

- C++17 introduces compile-time `if` statement that allows us to enable or disable specific statement based on compile-time conditions

Compile-Time `if`


```cpp
template <typename T>
void print(T const& t) {
  std::cout << t << '\n';
}

template <typename T, typename... Params>
void print(T const& head, Params const&... tail) {
  print(head);
  print(tail...);
}
```

```cpp
template <typename T, typename... Params>
void print(T const& head, Params const&... tail) {
  std::cout << head << '\n';
  if constexpr(sizeof...(tail) > 0) {
    // code only available if sizeof...(args)>0
    print(tail...);
  }
}
```