VARIADIC TEMPLATES

Plan for Today

- Variadic templates
- Fold Expressions
- std::pair
- std::tuple
- Structured Binding

Templates: Terminology

function parameter

template parameter

function parameter type

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

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

function expression

type of function expression is template argument

Variadic Templates: Introduction

- Template function or class that can take varying number and types of (function and template) parameters
- Useful when we know neither number nor types of arguments to be processed in call to function

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

Example: Variadic Function Template


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

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

Example: Variadic Class Template


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

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

Parameter Packs (1/2)

- Varying parameters indicated by ... known as *parameter pack*

`Types` is template parameter pack representing zero or more parameters

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

`params` is function parameter pack representing zero or more parameters.
Type of each `params` function parameter corresponds to `Types` template parameter

Parameter Packs (2/2)


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

Tuple<> t0;     // Types contains no arguments
Tuple<int> t1; // Types contains one argument: int
Tuple<int, double> t2; // Types contains two arguments:
                       // int and double
Tuple<0> error; // ERROR: 0 is not a type
```

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

f();      // args contains no arguments
f(1);     // args contains one argument: int
f(2,3.4); // args has 2 arguments: int and double
```

Variadic Function Templates: Recursion (1/6)

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

```
// do something to 1st argument and then recurse
// with rest of arguments
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 argument (the head) by calling `g()`:

```
// write argument 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 arguments (the `tail`)

```
// do something to 1st argument and then recurse
// with rest of arguments
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)

- Eventually, `tail` parameter pack will become empty

```
// 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 argument
  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: Example


```
void print() {
  std::cout << "\n";
}

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

int main() {
  print(1);
  print(1, 1.1);
  print(1, 1.1, "Hello");
  print(1, 1.1, "Hello", std::string("world"));
}
```

Overloading Variadic and Nonvariadic Templates

- If two function templates only differ by a trailing parameter, function template without trailing parameter pack is preferred

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

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

Performance of Varadic Function Templates

- No actual recursion involved
- Instead, sequence of function calls pre-generated at compile time – sort of like unrolling loops
- In general, sequence is manageable if number of number of arguments is not large
- With aggressive inlining, compilers can remove runtime function calls
- In contrast, variadic functions using `<cstdarg>` involve manipulation of runtime stack

Variadic Class Template: Example


```cpp
template <typename... Types>
struct C {
  std::size_t size() const {
    return sizeof...(Types);
  }
};

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

Variadic Class Template: Another Example


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

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

Variadic Class Template: Perfect Forwarding


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

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

```
double d = createT<double>(11.89);
std::cout << "d: " << d << "\n";
int i = createT<int>(17);
std::cout << "i: " << i << "\n";
std::string s = createT<std::string>("hello world");
std::cout << "s: "  << s << "\n";
C c = createT<C>(i, d, 3.14);
std::cout << "c: "  << c << "\n";
```

Fold Expressions

- An expression involving parameter pack that *reduces* elements of pack over binary operator
- Provides ability to compute result of using binary operator over all arguments of parameter pack

Fold Expressions: Motivation (1/2)

- Before C++17, we compute sum of unknown number of arguments like this:

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

template <typename T, typename... Types>
T sum(T head, Types... tail) {
  return head + sum(tail...);
}
```

1) Cumbersome to write [for programmers]
2) Stresses out compilers

Fold Expressions: Motivation (2/2)

- Since C++17, effort reduces significantly for both programmer and compiler with fold expressions:

```
template <typename... Types>
auto sum(Types... params) {
  return (... + params);
}
```

unary left fold expands to

```
((params1 + params2) + params3) + ...
```

Fold Expressions: Unary Right Fold


```
template <typename... Types>
auto sum(Types... params) {
  return (params + ...);
}
```

unary right fold expands to

```
params1 + (params2 + ... (paramsN-1 + paramsN))
```