VARIADIC TEMPLATES

Plan for Today

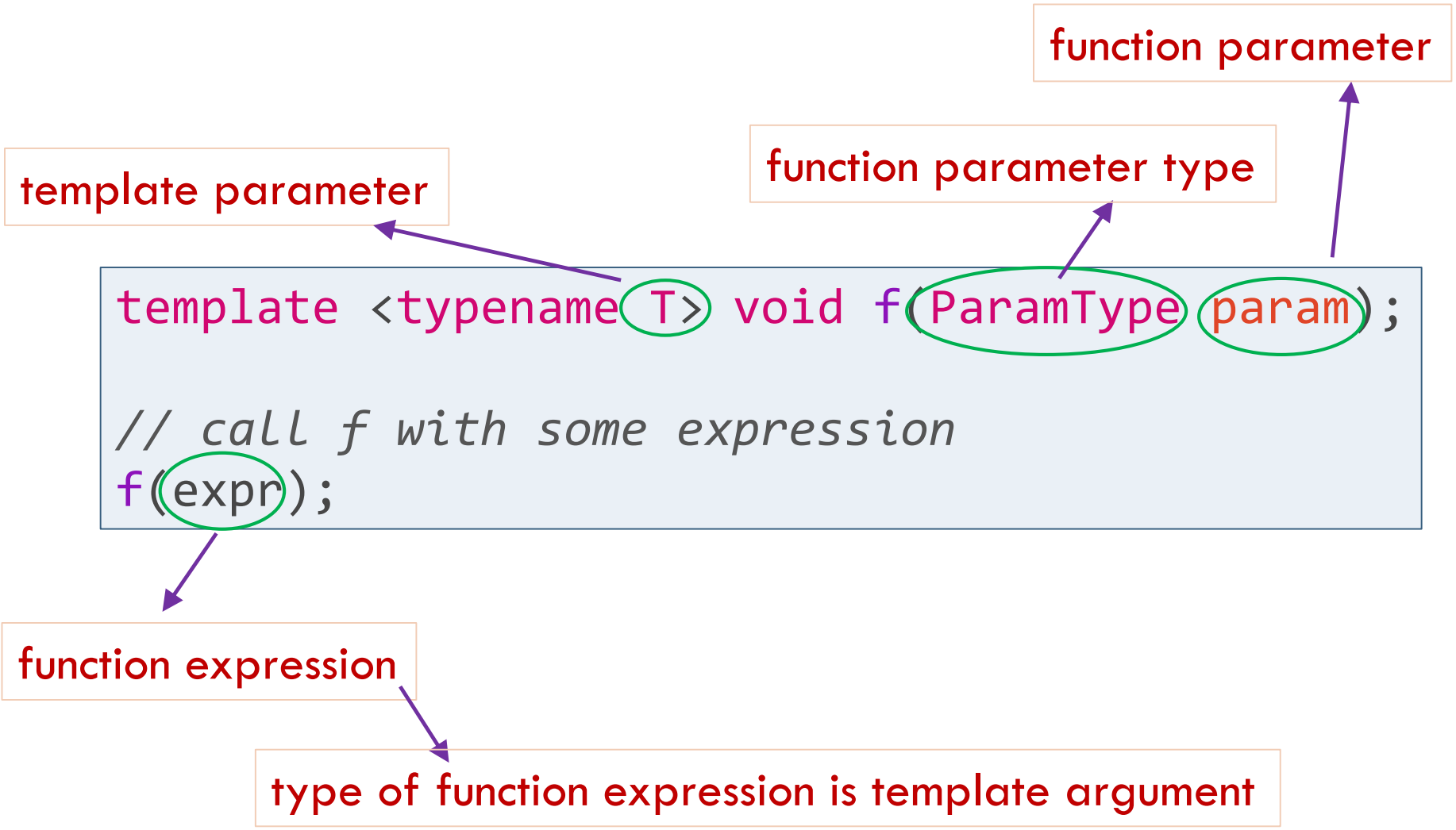
- Variadic templates
- Fold Expressions
- std::pair
- std::tuple
- Structured Binding

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 ...
}
```

Templates: Terminology



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

- 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 is corresponding `Types` template parameter

Parameter Packs


```
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
```

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
}
```

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 << ' ';
}
```

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
}
```

Function Templates: Recursion (4/6)

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

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

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
}
```

Function Templates: Recursion (6/6)

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

Call	Head	tail
`f(0.3,'c',1)`	`0.3`	`'c',1`
`f('c',1)`	`'c'`	`1`
`f(1)`	`1`	empty

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