Compose This!
Sean Parent | Senior Principal Scientist, Photoshop
Function Composition

“Function composition is an act or mechanism to combine simple functions to build more complicated ones.” – Wikipedia
Function Composition

“Function composition is an act or mechanism to combine simple functions to build more complicated ones.” – Wikipedia

\[ f(g(x)) \]
Category Theory

- The study of how objects with morphisms (functions) compose
- Category Theory ignores complexity
- There may be multiple ways to compose a function with different efficiency tradeoffs
STL Composition
STL Composition
STL Composition
STL Composition

stable_partition(p, l, s)
STL Composition

stable_partition(p, l, s)
STL Composition

```
stable_partition(f, p, not_fn(s))
```
STL Composition

\[
\text{stable}\_\text{partition}(f, p, \text{not}\_\text{fn}(s))
\]
STL Composition

```plaintext
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```
STL Composition

\[
\text{stable\_partition}(f, p, \text{not\_fn}(s))
\]
\[
\text{stable\_partition}(p, l, s)
\]
 STL Composition

```cpp
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```
```cpp
return { stable_partition(f, p, not_fn(s)),
        stable_partition(p, l, s) };
```
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
Lazy Gather (with Range v3)
Lazy Gather (with Range v3)
Lazy Gather (with Range v3)

```cpp
template <class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
```
Lazy Gather (with Range v3)

```cpp
// R models forward_range
// S models predicate
template<
class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
```
Lazy Gather (with Range v3)

```cpp
template <class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
auto [c, d] = partition(r | drop(p), s);
```
Lazy Gather (with Range v3)

```cpp
template <class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}
```

```cpp
auto [a, b] = partition(r | take(p), s);
auto [c, d] = partition(r | drop(p), s);
```
Lazy Gather (with Range v3)

```cpp
template <class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
auto [c, d] = partition(r | drop(p), s);
return make_tuple(b, concat(a, c), d);
```
Lazy Gather (with Range v3)

```cpp
template<
class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
auto [c, d] = partition(r | drop(p), s);
return make_tuple(b, concat(a, c), d);
```
Lazy Gather (with Range v3)

```cpp
template <class R, // R models forward_range
class S> // S models predicate
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

auto [a, b] = partition(r | take(p), s);
auto [c, d] = partition(r | drop(p), s);
return make_tuple(b, concat(a, c), d);
```
Lazy Gather (with Range v3)

```cpp
// R models forward_range
// S models predicate
template <class R, class S>
auto partition(R&& r, S s) {
    return make_tuple(r | filter(s), r | remove_if(s));
}

// R models forward_range
// S models predicate
template <class R, class S>
auto gather(R&& r, size_t p, S s) {
    auto [a, b] = partition(r | take(p), s);
    auto [c, d] = partition(r | drop(p), s);
    return make_tuple(b, concat(a, c), d);
}
```
“Surprisingly less bad than I expected!” – Me
“STL Doesn’t Compose”

- Can’t tie an output iterator to an input iterator
“STL Doesn’t Compose”

- Can’t tie an output iterator to an input iterator

```cpp
auto odd_even_n(size_t n) {
    vector<int> v;
    generate_n(back_inserter(v), n, [_n = 0](){ mutable { return _n++ ; } });
    vector<int> r1, r2;
    partition_copy(begin(v), end(v), back_inserter(r1), back_inserter(r2),
                   [] (const auto& e) { return e & 1 ; });
    return make_tuple(r1, r2);
}
```
Surprisingly less bad than you might expect!
Compositional Efficiency

- A theory of cost of composition categorized by
Compositional Efficiency

- A theory of cost of composition categorized by
  - object attributes
Compositional Efficiency

- A theory of cost of composition categorized by
  - object attributes
  - operation
Compositional Efficiency

- A theory of cost of composition categorized by
  - object attributes
  - operation
  - algorithms
Compositional Efficiency

- A theory of cost of composition categorized by
  - object attributes
  - operation
  - algorithms
  - result form (i.e. in-situ, lazy, copy)
Compositional Cost
Compositional Cost

- *Compositional Cost* of a function call is defined to be 1
Compositional Cost

- Compositional Cost of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
Compositional Cost

- Compositional Cost of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
- Cost may be parameterized
Compositional Cost

- Compositional Cost of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
- Cost may be parameterized
  - i.e. a virtual function call for every element might have a compositional cost of $1.2N$
Compositional Cost

- *Compositional Cost* of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
- Cost may be parameterized
  - i.e. a virtual function call for every element might have a compositional cost of \(1.2N\)

- Important for polymorphic values, asynchronous operations
Compositional Cost

- Compositional Cost of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
- Cost may be parameterized
  - i.e. a virtual function call for every element might have a compositional cost of $1.2N$

- Important for polymorphic values, asynchronous operations
- Cost is time, not number of operations, and considers cache effects and scale
Compositional Cost

- Compositional Cost of a function call is defined to be 1
- Compositional Cost of operation written in the most efficient form as a single function is 0
- Cost may be parameterized
  - i.e. a virtual function call for every element might have a compositional cost of $1.2N$

- Important for polymorphic values, asynchronous operations
- Cost is time, not number of operations, and considers cache effects and scale
- Goal is to be able to predict the most efficient approach to solve a given problem