Surfacing Composition

Toby Allsopp

WhereScape Software

toby@mi6.gen.nz
Twitter: @toby_allsopp

Composition

Expressions

Primary Expressions

Primary Expressions

literal 3

Primary Expressions

literal	`3`
id-expression	`x`

Primary Expressions

literal	`3`
id-expression	`x`
lambda	`[]{}`

Primary Expressions

literal	`3`
id-expression	`x`
lambda	`[]{}`
fold	`(xs + ... + 0)`

Operations on Expressions

Operations on Expressions

parenthesise `(x)`

Operations on Expressions

parenthesise	`(x)`
unary operator	`-x`

Operations on Expressions

parenthesise	`(x)`
unary operator	`-x`
binary operator	`x + 4`

Operations on Expressions

parenthesise	`(x)`
unary operator	`-x`
binary operator	`x + 4`
ternary operator	`x ? 7 : y`

Operations on Expressions

parenthesise	`(x)`
unary operator	`-x`
binary operator	`x + 4`
ternary operator	`x ? 7 : y`
function call	`f(x, y)`

Restrictions

`"Hello"s + 7`	(no matching overload)
`&7`	(can't take the address of a prvalue)
`7(1, 2)`	(not a function)

Not every composition yields an expression, but none yield anything *other* than an expression.

Statements

Primitive Statements

Primitive Statements

expression statement `x + 4;`

Primitive Statements

expression statement `x + 4;`

jump statement

```
goto fail;
return 7;
```

Primitive Statements

expression statement	`x + 4;`
jump statement	`goto fail;` `return 7;`
declaration statement	`auto x = 5;`

Composed Statements

Composed Statements

compound statement { *statement*$_1$ … *statement*$_n$ }

Composed Statements

compound statement { *statement*$_1$ … *statement*$_n$ }

selection statement `if` (*condition*) *statement*$_1$ `else` *statement*$_2$

`switch` (*condition*) *statement*

Composed Statements

compound statement { *statement*$_1$ … *statement*$_n$ }

selection statement `if` (*condition*) *statement*$_1$ `else` *statement*$_2$

`switch` (*condition*) *statement*

iteration statement `do` *statement* `while` (*condition*) `;`

`while` (*condition*) *statement*

`for` (*for-stuff*) *statement*

Types

Fundamental Types

- `void`
- `std::nullptr_t`
- `bool`
- `char`, `wchar_t`, `char16_t`, `char32_t`
- `int`, `unsigned`, `short`, `long`, …
- `float`, `double`, `long double`

Compound Types

Compound Types

reference `int&`

Compound Types

reference	`int&`
pointer	`int*`

Compound Types

reference	`int&`
pointer	`int*`
pointer to member	`int C::*`

Compound Types

reference	`int&`
pointer	`int*`
pointer to member	`int C::*`
array	`int[3]`

Compound Types

reference	`int&`
pointer	`int*`
pointer to member	`int C::*`
array	`int[3]`
function	`int(int, int)`

Compound Types

reference	`int&`
pointer	`int*`
pointer to member	`int C::*`
array	`int[3]`
function	`int(int, int)`
enumeration	`enum class E : int;`

Compound Types

reference	`int&`
pointer	`int*`
pointer to member	`int C::*`
array	`int[3]`
function	`int(int, int)`
enumeration	`enum class E : int;`
class	`struct { int x; int y; }`

Functions

Functions?

Functions?

Anything that works with `std::invoke` (C++17)

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions
- pointers to functions

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions
- pointers to functions
- objects with `operator()`

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions
- pointers to functions
- objects with `operator()`
- objects with implicit conversion to function pointer

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions
- pointers to functions
- objects with `operator()`
- objects with implicit conversion to function pointer
- pointers to member functions

Functions?

Anything that works with `std::invoke` (C++17)

Corresponds to the *Callable* named requirement

- actual functions
- references to functions
- pointers to functions
- objects with `operator()`
- objects with implicit conversion to function pointer
- pointers to member functions
- pointers to data members

Manual Function Composition

```
struct person {
  year_month_day dob;
  string name;
};

int dob_to_age(year_month_day dob);

int person_age(person p) {
  return dob_to_age(p.dob);
}
```

Manual Function Composition

```
struct person {
  year_month_day dob;
  string name;
};

int dob_to_age(year_month_day dob);

int person_age(person p) {
  return dob_to_age(p.dob);
}
```

See the function composition?

Manual Function Composition

```
struct person {
  year_month_day dob;
  string name;
};

int dob_to_age(year_month_day dob);

int person_age(person p) {
  return dob_to_age(p.dob);
}
```

See the function composition?

We call `dob_to_age` with the result of getting the dob member from p.

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Before

```
int person_age(person p) {
  return dob_to_age(p.dob);
}
```

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

```cpp
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Before

```cpp
int person_age(person p) {
  return dob_to_age(p.dob);
}
```

After

```cpp
int person_age(person p) {
  return compose(dob_to_age, &person::dob)(p);
}
```

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \rightarrow f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Before

```
int person_age(person p) {
  return dob_to_age(p.dob);
}
```

After

```
int person_age(person p) {
  return compose(dob_to_age, &person::dob)(p);
}
```

```
inline constexpr auto person_age = compose(dob_to_age, &person::dob);
```

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \rightarrow f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Before

```
int person_age(person p) {
  return dob_to_age(p.dob);
}
```

After

```
int person_age(person p) {
  return compose(dob_to_age, &person::dob)(p);
}
```

~~`inline constexpr auto person_age = compose(dob_to_age, &person::dob);`~~

Surfaced Function Composition

$$\texttt{compose}(f, g) = \lambda(x) \to f(g(x))$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Before

```
int person_age(person p) {
  return dob_to_age(p.dob);
}
```

After

```
int person_age(person p) {
  return compose(dob_to_age, &person::dob)(p);
}
```

~~`inline constexpr auto person_age = compose(dob_to_age, &person::dob);`~~

But is that actually useful?

Manual Function Composition

```
struct person {
  year_month_day dob;
  string name;
};
int person_age(person p);
```

```
bool any_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
          [](person p) {
            return person_age(p) > 40;
          });
}
```

Spot the function composition?

Manual Function Composition

```
struct person {
  year_month_day dob;
  string name;
};
int person_age(person p);
```

```
bool any_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
          [](person p) {
            return person_age(p) > 40;
          });
}
```

Spot the function composition?

There's lots, but let's focus on the lambda.

Surfaced Function Composition

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

Surfaced Function Composition

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

Before

```
bool any_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
          [](person p) {
            return person_age(p) > 40;
          });
}
```

Surfaced Function Composition

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

Before

```
bool any_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
          [](person p) {
            return person_age(p) > 40;
          });
}
```

After

```
bool any_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
          compose(greater_than(40), person_age));
}
```

Partial Function Application

$$\texttt{partial}(f, x) = \lambda(x_1, \ldots) \to f(x, x_1, \ldots)$$

Partial Function Application

$$\texttt{partial}(f, x) = \lambda(x_1, \ldots) \rightarrow f(x, x_1, \ldots)$$

`std::bind`

Partial Function Application

$$\texttt{\textbf{partial}}(f, x) = \lambda(x_1, \ldots) \to f(x, x_1, \ldots)$$

`std::bind`

Before

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

Partial Function Application

$$\texttt{partial}(f, x) = \lambda(x_1, \ldots) \to f(x, x_1, \ldots)$$

`std::bind`

Before

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

After

```
auto greater_than(int x) {
  return std::bind(std::greater(), _1, x);
}
```

Partial Function Application

$$\texttt{partial}(f, x) = \lambda(x_1, \ldots) \rightarrow f(x, x_1, \ldots)$$

`std::bind`

Before

```
auto greater_than(int x) {
  return [=](auto y) { return y > x; };
}
```

After

```
auto greater_than(int x) {
  return std::bind(std::greater(), _1, x);
}
```

OK, but is it *really* useful?

More Complicated Function Composition

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return person_age(p1) < person_age(p2);
            });
}
```

More Complicated Function Composition

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return person_age(p1) < person_age(p2);
            });
}

void sort_by_name(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return p1.name < p2.name;
            });
}
```

More Complicated Function Composition

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return person_age(p1) < person_age(p2);
            });
}

void sort_by_name(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return p1.name < p2.name;
            });
}
```

What kind of composition is this?

Surfaced Function Composition

$$\texttt{on}(f, g) = \lambda(x_1, x_2, \ldots) \rightarrow f(g(x_1), g(x_2), \ldots)$$

Surfaced Function Composition

$$\mathtt{on}(f, g) = \lambda(x_1, x_2, \ldots) \rightarrow f(g(x_1), g(x_2), \ldots)$$

```
template <typename F, typename G>
auto on(F f, G g) {
  return [=](auto... x) {
    return std::invoke(f, std::invoke(g, x)...);
  };
}
```

Surfaced Function Composition

$$\texttt{on}(f, g) = \lambda(x_1, x_2, \ldots) \to f(g(x_1), g(x_2), \ldots)$$

```
template <typename F, typename G>
auto on(F f, G g) {
  return [=](auto... x) {
    return std::invoke(f, std::invoke(g, x)...);
  };
}
```

Before

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return person_age(p1) < person_age(p2);
            });
}
void sort_by_name(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return p1.name < p2.name;
            });
}
```

Surfaced Function Composition

$$\texttt{on}(f, g) = \lambda(x_1, x_2, \ldots) \to f(g(x_1), g(x_2), \ldots)$$

```
template <typename F, typename G>
auto on(F f, G g) {
  return [=](auto... x) {
    return std::invoke(f, std::invoke(g, x)...);
  };
}
```

Before

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return person_age(p1) < person_age(p2);
            });
}
void sort_by_name(vector<person>& people) {
  std::sort(begin(people), end(people),
            [](person p1, person p2) {
              return p1.name < p2.name;
            });
}
```

After

```
void sort_by_age(vector<person>& people) {
  std::sort(begin(people), end(people),
            on(std::less(), person_age));
}


void sort_by_name(vector<person>& people) {
  std::sort(begin(people), end(people),
            on(std::less(), &person::name));
}
```

`compose` and `on`

$$\begin{aligned} \texttt{compose}(f, g) &= \lambda(x) &&\to f(g(x)) \\ \texttt{on}(f, g) &= \lambda(x_1, x_2, \ldots) &&\to f(g(x_1), g(x_2), \ldots) \end{aligned}$$

`compose` and `on`

$$\texttt{compose}(f, g) = \lambda(x) \rightarrow f(g(x))$$
$$\texttt{on}(f, g) = \lambda(x_1, x_2, \ldots) \rightarrow f(g(x_1), g(x_2), \ldots)$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

compose and on

$$\texttt{compose}(f, g) = \lambda(x) \rightarrow f(g(x))$$

$$\texttt{on}(f, g) = \lambda(x_1, x_2, \ldots) \rightarrow f(g(x_1), g(x_2), \ldots)$$

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

```
template <typename F, typename G>
auto on(F f, G g) {
  return [=](auto... x) {
    return std::invoke(f, std::invoke(g, x)...);
  };
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

More Difficult Composition

```
struct person {
  optional<year_month_day> dob;
  string name;
};

// same as before
int dob_to_age(year_month_day dob);

auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}
```

```
bool is_age_wise(int age) { return age > 40; }

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}

bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

Notice anything?

Composing with `std::optional`

Composing with `std::optional`

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

Composing with `std::optional`

```cpp
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

```cpp
template <typename F, typename G>
auto compose_optional(F f, G g) {
  return [=](auto x)
      -> optional<decltype(
           std::invoke(f, *std::invoke(g, x)))> {
    const auto o = std::invoke(g, x);
    if (!o) return nullopt;
    return std::invoke(f, *o);
  };
}
```

Composing with `std::optional`

```cpp
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

```cpp
template <typename F, typename G>
auto compose_optional(F f, G g) {
  return [=](auto x)
      -> optional<decltype(
            std::invoke(f, *std::invoke(g, x)))> {
    const auto o = std::invoke(g, x);
    if (!o) return nullopt;
    return std::invoke(f, *o);
  };
}
```

Before

```cpp
auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}
```

Composing with `std::optional`

```
template <typename F, typename G>
auto compose(F f, G g) {
  return [=](auto x) {
    return std::invoke(f, std::invoke(g, x));
  };
};
```

```
template <typename F, typename G>
auto compose_optional(F f, G g) {
  return [=](auto x)
      -> optional<decltype(
           std::invoke(f, *std::invoke(g, x)))> {
    const auto o = std::invoke(g, x);
    if (!o) return nullopt;
    return std::invoke(f, *o);
  };
}
```

Before

```
auto person_age(person p) -> optional<int> {
  if (!p.dob) return nullopt;
  return dob_to_age(*p.dob);
}

auto is_person_wise(person p) -> optional<bool> {
  const auto age = person_age(p);
  if (!age) return nullopt;
  return is_age_wise(*age);
}
```

After

```
auto person_age(person p) -> optional<int> {
  return compose_optional(dob_to_age,
                          &person::dob)(p);
}

auto is_person_wise(person p) -> optional<bool> {
  return compose_optional(is_age_wise,
                          person_age)(p);
}
```

Composing with `std::optional`

```
template <typename T>
auto value_or(T x) {
  return [=](auto o) { return o.value_or(x); };
};
```

Before

```
bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

Composing with `std::optional`

```
template <typename T>
auto value_or(T x) {
  return [=](auto o) { return o.value_or(x); };
};
```

Before

```
bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    [](person p) {
      return is_person_wise(p).value_or(false);
    });
}
```

After

```
bool any_known_wise_ones(vector<person> people) {
  return std::any_of(
    begin(people), end(people),
    compose(value_or(false), is_person_wise));
}
```

Badly Behaved Functions

inputs -> outputs = easy to compose

Badly Behaved Functions

inputs -> outputs = easy to compose

What about functions that return an error code and have output parameters?

Badly Behaved Functions

inputs -> outputs = easy to compose

What about functions that return an error code and have output parameters?

```
bool read_file(path in, string& out);
bool parse_person(string in, person& out);

bool read_person(path in, person& out) {
  string s;
  if (!read_file(in, s)) return false;
  return parse_person(s, out);
}
```

Badly Behaved Functions

inputs -> outputs = easy to compose

What about functions that return an error code and have output parameters?

```
bool read_file(path in, string& out);
bool parse_person(string in, person& out);

bool read_person(path in, person& out) {
  string s;
  if (!read_file(in, s)) return false;
  return parse_person(s, out);
}
```

```
template <typename F, typename G>
auto compose_out(F f, G g) {
  return [=](auto in, auto& out) {
    second_param_t<G>  tmp; // left as an exercise
    if (!g(in, tmp)) return false;
    return f(tmp, out);
  };
}

bool read_person(path in, person& out) {
  return compose_out(read_file,
                     parse_person)(in, out);
}
```

Algorithms

Transforming and Filtering

Say we want to extract the names of wise people…

Transforming and Filtering

Say we want to extract the names of wise people…

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

Transforming and Filtering

Say we want to extract the names of wise people...

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

But, algorithms!?!?

Standard Algorithms

```
template <typename InputIt, typename OutputIt,
          typename UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 UnaryPredicate pred);
```

```
template <typename InputIt, typename OutputIt,
          typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

Standard Algorithms

```
template <typename InputIt, typename OutputIt,
          typename UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 UnaryPredicate pred);
```

```
template <typename InputIt, typename OutputIt,
          typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

Before

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

Standard Algorithms

```
template <typename InputIt, typename OutputIt,
          typename UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 UnaryPredicate pred);
```

```
template <typename InputIt, typename OutputIt,
          typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

Before

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

After

```
auto names_of_wise_ones(vector<person> people) {
  vector<person> wise_ones;
  std::copy_if(begin(people), end(people),
               std::back_inserter(wise_ones),
               compose(value_or(false),
                       is_person_wise));
  vector<string> names;
  std::transform(begin(wise_ones), end(wise_ones),
                 std::back_inserter(names),
                 &person::name);
  return names;
}
```

Standard Algorithms

```
template <typename InputIt, typename OutputIt,
          typename UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 UnaryPredicate pred);
```

```
template <typename InputIt, typename OutputIt,
          typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

Before

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

After

```
auto names_of_wise_ones(vector<person> people) {
  vector<person> wise_ones;
  std::copy_if(begin(people), end(people),
               std::back_inserter(wise_ones),
               compose(value_or(false),
                       is_person_wise));
  vector<string> names;
  std::transform(begin(wise_ones), end(wise_ones),
                 std::back_inserter(names),
                 &person::name);
  return names;
}
```

Great, right?

Standard Algorithms

```
template <typename InputIt, typename OutputIt,
          typename UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 UnaryPredicate pred);
```

```
template <typename InputIt, typename OutputIt,
          typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

Before

```
auto names_of_wise_ones(vector<person> people) {
  vector<string> names;
  for (const auto& person : people) {
    if (is_person_wise(person).value_or(false)) {
      names.push_back(person.name);
    }
  }
  return names;
}
```

After

```
auto names_of_wise_ones(vector<person> people) {
  vector<person> wise_ones;
  std::copy_if(begin(people), end(people),
               std::back_inserter(wise_ones),
               compose(value_or(false),
                       is_person_wise));
  vector<string> names;
  std::transform(begin(wise_ones), end(wise_ones),
                 std::back_inserter(names),
                 &person::name);
  return names;
}
```

Great, right? We've made a whole extra copy!

Enter Range Adaptors (and Views)

```
auto names_of_wise_ones(vector<person> people) {
  std::vector<string> names =
    people | view::filter(compose(value_or(false), is_person_wise))
           | view::transform(&person::name);
  return names;
}
```

Enter Range Adaptors (and Views)

```
auto names_of_wise_ones(vector<person> people) {
  std::vector<string> names =
    people | view::filter(compose(value_or(false), is_person_wise))
           | view::transform(&person::name);
  return names;
}
```

Available now in range-v3

Enter Range Adaptors (and Views)

```
auto names_of_wise_ones(vector<person> people) {
  std::vector<string> names =
    people | view::filter(compose(value_or(false), is_person_wise))
           | view::transform(&person::name);
  return names;
}
```

Available now in range-v3

Proposed for C++20

- [P0896] The One Ranges Proposal
- [P0789] Range Adaptors and Utilities
- [P1206] Range constructors for standard containers and views

Summary

Summary

Summary

- Composition is everywhere

Summary

- Composition is everywhere
- Composition can be hidden or explicit

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions
- Output parameters hurt composability

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions
- Output parameters hurt composability
- Standard algorithms do not compose well

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions
- Output parameters hurt composability
- Standard algorithms do not compose well
- Ranges will fix this

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions
- Output parameters hurt composability
- Standard algorithms do not compose well
- Ranges will fix this

But most of all:

Summary

- Composition is everywhere
- Composition can be hidden or explicit
- Functions can be composed by functions
- Output parameters hurt composability
- Standard algorithms do not compose well
- Ranges will fix this

But most of all:

Think compositionally!

Thank you!

Questions?

Email	toby@mi6.gen.nz
Twitter	@toby_allsopp
#include<C++> Discord	Toby Allsopp#8409
Cpplang Slack	toby_allsopp