Example - array

```
void print_coordinates(const std::array<int, 3>& data)
{
    const auto& [x, y, z] = data;
    std::printf("x=%d, y=%d, z=%d", x, y, z);
}
```

…is roughly equivalent to…

```
void print_coordinates(const std::array<int, 3>& data)
{
    const auto& arr = data;
    std::printf("x=%d, y=%d, z=%d", arr[0], arr[1], arr[2]);
}
```

Example - struct

```
struct person { std::string _name; int _age; };

[[nodiscard]] std::string to_json(const person& p)
{
    const auto& [name, age] = p;
    return concat("{'name':", name, ",'age':", age, '}');
}
```

Example - struct

```
[[nodiscard]] std::string to_json(const person& p)
{
    const auto& [name, age] = p;
    return concat("{'name':", name, ",'age':", age, '}');
}
```

…is roughly equivalent to…

```
struct person { std::string _name; int _age; };

[[nodiscard]] std::string to_json(const person& p)
{
    const auto& e = p;
    return concat("{'name':", e._name, ",'age':", e._age, '}');
}
```

Semantics - tuple/pair/custom

```
auto [a, b] = expr;
```

- Let `E` be the type of `expr`
- Applies only when `std::tuple_size<E>` is defined
- `a` is a reference to a `std::tuple_element_t<0, E>`, initialized with `get<0>(expr)`
- `b` is a reference to a `std::tuple_element_t<1, E>`, initialized with `get<1>(expr)`
- The traits can be specialized for custom types

Example - tuple/pair

```
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& [p, a] : addresses)
    {
        std::cout << "Person " << p << " lives at " << a << '\n';
    }
}
```

- `p` is a `const person&` initialized from `std::get<0>(/* element */)`
- `a` is a `const address&` initialized from `std::get<1>(/* element */)`

Example - tuple/pair

```
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& [p, a] : addresses)
    {
        std::cout << "Person " << p << " lives at " << a << '\n';
    }
}
```

…is roughly equivalent to…

```
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& element : addresses)
    {
        const person& p = std::get<0>(element);
        const address& a = std::get<1>(element);
        std::cout << "Person " << p << " lives at " << a << '\n';
    }
}
```

Customizing structured bindings

» Example custom type

```
struct io_channel
{
    struct sender   { /* ... */ };
    struct receiver { /* ... */ };

    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

Example custom type

```
struct io_channel
{
    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

Instead of…

```
io_channel channel;
auto sender = channel.make_sender();
auto receiver = channel.make_receiver();
```

…we would like the following:

```
io_channel channel;
auto& [sender, receiver] = channel;
```

Customization points

- Either must be valid:
  - `template <std::size_t I> T::get();`
  - `template <std::size_t I> get(T);`
- Both must be specialized:
  - `template <> struct std::tuple_size<T>;`
  - `template <std::size_t I> struct std::tuple_element<I, T>;`

Defining get

```
struct io_channel
{
    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

```
template <std::size_t I>
auto get(io_channel& c)
{
    if constexpr (I == 0) { return c.make_sender(); }
    else                  { return c.make_receiver(); }
}
```

Specializing `std::tuple_size`

```
struct io_channel
{
    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

```
namespace std
{
    template <>
    struct tuple_size<io_channel>
        : std::integral_constant<std::size_t, 2> { };
}
```

Specializing `std::tuple_element`

```
struct io_channel
{
    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

```
namespace std
{
    template <>
    struct tuple_element<0, io_channel>
    { using type = io_channel::sender; };

    template <>
    struct tuple_element<1, io_channel>
    { using type = io_channel::receiver; };
}
```

Final usage example

```
struct io_channel
{
    auto make_sender()   { return sender{/*... */}; }
    auto make_receiver() { return receiver{/*... */}; }
};
```

```
int main()
{
    io_channel channel;
    auto& [sender, receiver] = channel;
}
```

Pitfalls

- Bindings for class types are reference variables
- However, bindings for arrays/structs are **not** references
  - The identifiers introduced in the square brackets do **not** define new variables of reference type
  - They merely are alternative names for existing entities
- Due to this fact, type deduction can sometimes be surprising

Array/struct bindings are not references

```
struct coordinate { int _x; int _y; };

coordinate c{0, 0};
auto& [x, y] = c;

x = 42;
assert(c._x == 42);
    // Reference semantics...

static_assert(std::is_same_v<decltype(x), int>);
    // ...but `x` is not a reference!
```

Array/struct bindings are not references

C++17

» Remember that `x` is just another name for `c._x`

```
coordinate c{0, 0};
auto& [x, y] = c;
x = 42;
assert(c._x == 42);
static_assert(std::is_same_v<decltype(x), int>);
```

…is equivalent to…

```
coordinate c{0, 0};
auto& c_ref = c;
c_ref._x = 42;
assert(c._x == 42);
static_assert(std::is_same_v<decltype(c_ref._x), int>);
```

Array/struct bindings are not references

C++17

» This behavior is also noticeable when using `decltype(auto)`

```
coordinate c{0, 0};

auto& [x, y] = c;

decltype(auto) test = x;
static_assert(std::is_same_v<decltype(test), int>);
    // `test` is a copy of `c._x`!

test = 42;
assert(c._x == 0);
```

Bindings are not implicitly moved

C++17

» Another consequence is that RVO/implicit move will not apply

```
struct person { std::string _name; int _age; };

std::string name_of_nth_person(const std::size_t idx)
{
    auto [name, age] = global_person_registry[idx];
    return name;
}

auto example_name = name_of_nth_person(0);
    // The string is copied (!) out of `name_of_nth_person`.
```

Bindings are not implicitly moved

C++11/14

» The previous code is equivalent to

```
std::string name_of_nth_person(const std::size_t idx)
{
    auto p_copy = global_person_registry[idx];
    return p_copy._name;
}
```

C++11/14

» A move can be forced with `std::move`

```
std::string name_of_nth_person(const std::size_t idx)
{
    auto p_copy = global_person_registry[idx];
    return std::move(p_copy._name);
}
```

Qualifiers apply to the hidden object

C++17

» Any *cv-qualifier* applies to the hidden object, not to the bindings themselves

```
std::tuple<char, int&> get_data();

const auto [c, i] = get_data();

c = 'a';
    // Compile-time error. `decltype(c)` is `const char`.

i = 42;
    // OK. `decltype(i)` is `int&`.
```

Qualifiers apply to the hidden object

```
std::tuple<char, int&> get_data();
const auto [c, i] = get_data();
```

» The types are retrieved via `std::tuple_element`

- `c` ⟶ `std::tuple_element_t<0, const std::tuple<char, int&>>`
- `i` ⟶ `std::tuple_element_t<1, const std::tuple<char, int&>>`

```
std::tuple_element_t<1, const std::tuple<char, int&>>
// ...is...
int& const
// ...which is...
int&
```

Limitations

- Structured bindings cannot be captured in lambdas
  - This was addressed in C++20, see P1091R3 and P1381R1
- Structured bindings cannot be nested
- No way to ignore a particular binding (or to apply an *attribute* to it)

Section recap

- Structured bindings allow data destructuring
  - Useful for readability and conciseness
- They support structs, arrays, and user-defined types
  - Provide `std::tuple_size`, `std::tuple_element`, and `get` for custom types
- Syntax: `cv-qualifiers auto [id0, id1] = expr`
  - The *cv-qualifiers* apply to `expr`
- Bindings are not references, they are name aliases

Discussion

- How C++ is becoming more and more terse

Exercise

- Create structured bindings for a custom class
  - `exercise3.cpp`
    - on Wandbox
    - on Godbolt

Class Template Argument Deduction

Class Template Argument Deduction

In this section

- Function template argument deduction
- Situation in C++11/14
- Class template argument deduction
  - Basics
  - Use cases
  - Pitfalls
- Deduction guides

Function template argument deduction - full deduction

» Function templates can often be invoked without explicitly specifying template arguments

- The compiler can *deduce* template arguments from the given function arguments

```
template <typename T>
void print_triple(T x)
{
    std::cout << x * x * x << '\n';
}
```

```
print_triple<int>(10); // OK, explicitly provided `T = int`
print_triple(10);      // OK, `T` is deduced as `int`
print_triple(10.f);    // OK, `T` is deduced as `float`
```

Function template argument deduction - partial deduction

» Function template arguments can also be *partially* deduced

```
template <typename A, typename B>
void print_sum(A a, B b)
{
    std::cout << a + b << '\n';
}
```

```
print_sum<int, int>(1, 2); // OK, everything is explicitly provided
print_sum<int>(1, 2)       // OK, `A` is explicitly provided, `B` is deduced
print_sum(1, 2);           // OK, both `A` and `B` are deduced as `int`
```

Function template argument deduction - no function arguments

» Finally, there are cases where deduction cannot take place

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

```
print_default_constructed<int>(); // OK, explicitly provided `T = int`
print_default_constructed();      // Error, `T` could not be deduced
```

Function template argument deduction - “Non-deduced context”

» A more interesting case occurs in the presence of a “non-deduced context”

```
template <typename T>
struct identity { using type = T; };

template <typename T>
void print_wrapped(typename identity<T>::type x)
{
    std::cout << x << '\n';
}
```

```
print_wrapped<int>(0) // OK, explicitly provided `T = int`
print_wrapped(0);     // Error, `T` could not be deduced
```

» The compiler isn’t able to figure out a reverse mapping

- E.g. `identity` could be explicitly specialized

C++11/14 - Inconsistencies between function and class templates

» Function and class templates sometimes behave differently

- Template argument deduction is an example of that

```
// C++11/14
template <typename T> void foo(T) { }

foo<int>(100); // OK
foo(100);      // OK
```

```
// C++11/14
template <typename T> struct bar { bar(T) { } };

bar<int>(100); // OK
bar(100);      // Error, `T` could not be deduced
```

C++11/14 - The need for `make_xxx` functions

» Inability to deduce class template arguments is why `make_xxx` functions exists

```
// C++11/14
std::tuple<int, char, float> t0{0, 'a', 1.f}; // OK, everything is explicitly provided
std::tuple t1{0, 'a', 1.f};                   // Error, cannot deduce template arguments
auto t2 = std::make_tuple(0, 'a', 1.f);       // OK, deduction is performed by FTAD
```

» This is a burden on library developers

- They need to provide a `make_xxx` for their types

```
// C++11/14
template <typename T> struct bar { bar(T) { } };

template <typename T>
bar<T> make_bar(T x) { return bar<T>(x); }
    // Just boilerplate!

bar<int>(100);           // OK
bar(100);                // Error, `T` could not be deduced
auto b = make_bar(100); // OK, deduction is performed by FTAD
```

C++11/14 - `make_xxx` functions in the Standard Library

- The Standard Library provides:
  - `std::make_any`
  - `std::make_pair`
  - `std::make_tuple`
  - `std::make_optional`
  - `std::make_move_iterator`
  - `std::make_reverse_iterator`
- All of these are superfluous, except in rare cases

Class Template Argument Deduction - example

» Given…

```
template <typename T>
struct bar
{
    bar(T) { }
};
```

» …in C++17, the following code is valid:

```
bar b1{100};
```

» **Q:** What is the type of `b1`?

Class Template Argument Deduction - example

» Given…

```
template <typename T>
struct bar
{
    bar(T) { }
};
```

» …in C++17, the following code is valid:

```
bar b1{100};
```

» **Q:** What is the type of `b1`?

» The type of `b1` is `bar<int>`

Class Template Argument Deduction - introduction

- C++17 introduces *Class template argument deduction*, a new core language feature
  - Colloquially called “CTAD”
- It allows class template arguments to be deduced from construction of an object

```
template <typename T>
struct bar
{
    bar(T) { }
};

bar<int> b0{100}; // OK in C++11/14/17
bar b1{100};      // Error in C++11/14, OK in C++17
```

- `bar::bar(T)` is used by the compiler to deduce the type of `T`

Class Template Argument Deduction - deduction guides #0

» Conceptually, a *deduction guide* is created for every constructor of a class

- We will explore user-defined deduction guides later

```
template <typename T>
struct bar
{
    bar(T) { }
};

template <typename T>
bar(T) -> bar<T>;
    // Deduction guide, implicitly generated by the compiler
    // Looks a lot like a function, doesn't it? :)

bar<int> b0{100}; // OK in C++11/14/17
bar b1{100};      // Error in C++11/14, OK in C++17
```

Class Template Argument Deduction - deduction guides #1

```
template <typename T>
struct baz
{
    baz(std::vector<T>) { } // (0)
    baz(std::list<T>)   { } // (1)
};

baz<int> b0{std::list<int>{}}; // OK, calls (1)
baz b1{std::list<int>{}};      // OK, calls (1)
baz b2{std::vector<int>{}};    // OK, calls (0)
```

» Implicitly-generated deduction guides:

```
template <typename T> baz(std::vector<T>) -> baz<T>;
template <typename T> baz(std::list<T>) -> baz<T>;
```

» Key point: **CTAD leverages the existing FTAD rules**

Class Template Argument Deduction - inconsistencies #0

» Unfortunately, we're not fully consistent yet…

- FTAD supports partial deduction, CTAD does *not*!
- CTAD supports explicit deduction guides, FTAD does *not*!
- CTAD has *special* behavior for copies and moves (!)

Class Template Argument Deduction - inconsistencies #1

» Let's see an example of partial deduction:

```
template <typename A, typename B>
void f(A, B)
{
}

f<int, char>(0, 'a'); // OK
f<int>(0, 'a');       // OK
f(0, 'a');            // OK
```

```
template <typename A, typename B>
struct s
{
    s(A, B) { }
};

s<int, char>(0, 'a'); // OK
s<int>(0, 'a');       // Error - inconsistency with FTAD!
s(0, 'a');            // OK
```

Class Template Argument Deduction - inconsistencies #2

» What's special about copies and moves? Consider this class:

```
template <typename T>
struct wrapper
{
    wrapper(T) { }
};

wrapper<int> w0{0};              // OK
wrapper<wrapper<int>> w1{w0}; // OK
wrapper w2{w0};                  // ?
wrapper w3{w1};                  // ?
```

» **Q:** What are the types of `w2` and `w3`?

Class Template Argument Deduction - inconsistencies #2

» What's special about copies and moves? Consider this class:

```
template <typename T>
struct wrapper
{
    wrapper(T) { }
};

wrapper<int> w0{0};              // OK
wrapper<wrapper<int>> w1{w0}; // OK
wrapper w2{w0};                  // ?
wrapper w3{w1};                  // ?
```

» **Q:** What are the types of `w2` and `w3`?

» `w2` is a `wrapper<int>`

» `w3` is a `wrapper<wrapper<int>>`

» …surprising?

Class Template Argument Deduction - inconsistencies #3

» Recall that implicit deduction guides exist for *every* constructor

- Including copy and move constructors

```
template <typename T>
struct wrapper
{
    wrapper(T) { }
};

template <typename T> wrapper(T)                  -> wrapper<T>;
template <typename T> wrapper(const wrapper<T>&) -> wrapper<T>;
template <typename T> wrapper(wrapper<T>&&)       -> wrapper<T>;
```

» Note that the copy and move deduction guide do not double-wrap

- They also have higher priority over the user-defined constructor (!)

Class Template Argument Deduction - inconsistencies #4

» This can lead to unfortunate inconsistencies with `make_xxx` functions

```
std::tuple<int> t0;
std::tuple t1{t0};
auto t2 = std::make_tuple(t0);

static_assert(std::is_same_v<decltype(t1), std::tuple<int>>);
static_assert(std::is_same_v<decltype(t2), std::tuple<std::tuple<int>>>);
```

» The types of `t1` and `t2` are *not* the same!

- If you want double-wrapping, do not use CTAD
- This is particularly dangerous in template functions

```
template <typename T>
auto wrap_in_tuple(T x)
{
    return std::tuple{x};
}
```

» Terribly misleading name – it doesn't always wrap!

Sandbox

A

Wrap lines

Select all

```
A
S
M


g
e
n
e
r
a
t
i
o
```

Edit on Compiler Explorer

Class Template Argument Deduction - use cases #0

» There are many places where CTAD shines

- One example is classes like `std::lock_guard`

```cpp
// C++11/14
void do_something()
{
    static std::mutex my_mutex;
    std::lock_guard<std::mutex> guard{my_mutex};
    // ...
}
```

```cpp
// C++17
void do_something()
{
    static std::mutex my_mutex;
    std::lock_guard guard{my_mutex};
    // ...
}
```

Class Template Argument Deduction - use cases #1

» `std::array` is another great use case

```
// C++11/14
constexpr const char* error_to_string(ErrorEnum e)
{
    constexpr std::array<const char*, 3> strings {
        "OK",
        "Invalid operation",
        "Timeout"
    };
    return strings[static_cast<int>(e)];
}
```

```
// C++17
constexpr const char* error_to_string(ErrorEnum e)
{
    constexpr std::array strings {
        "OK",
        "Invalid operation",
        "Timeout"
    };
    return strings[static_cast<int>(e)];
}
```

» Partial deduction would make it even better…

Class Template Argument Deduction - use cases #2

» `scope_guard` is a killer example

- Wrapper for an arbitrary function to be invoked at the end of the scope

```
template <typename F>
struct scope_guard : F
{
    scope_guard(F&& f) : F{std::move(f)} { }
    ~scope_guard() { static_cast<F&>(*this)(); }
};
```

```
int main()
{
    scope_guard sg0{[]{ std::printf("world!"); }};
    scope_guard sg1{[]{ std::printf("hello "); }};
}
```

» Prints `“hello world!”

- Very useful for “cleanup” operations

Class Template Argument Deduction - use cases #3

» Here's an example with a stack:

```
int my_ranking::get_top_element_and_pop()
{
    scope_guard sg{[this]{ my_stack.pop_back(); }};
    return my_stack.back();
}
```

» This guarantees that the data will always be popped

- More elegant/efficient than using a `temp` variable

Class Template Argument Deduction - use cases #4

» Sometimes, it can be useful to create object instances

- But be aware of the priority given to copy/move operations!

```
std::tuple params0{5.f, -22.4f, -1.f};              // OK
auto params1 = std::make_tuple(5.f, -22.4f, -1.f); // Equivalent, but longer

std::tuple nested0{params0};                 // Not nested! Makes a copy of `params0`
auto nested1 = std::make_tuple(params0); // `std::tuple<std::tuple<...>>`
```

» Doesn’t happen that much in practice

- Prefer named structs to `std::tuple` and `std::pair`
- Prefer emplacement operations to `insert` or `push_back`
- Refrain from using CTAD in template function bodies

CTAD explicit deduction guides - motivation

```
std::list<int> src{0, 1, 2, 3};
std::vector dst(src.begin(), src.end());
```

» `dst` is correctly deduced as a `std::vector<int>`

- …how is that possible?

CTAD explicit deduction guides - example #0

» Developers can provide *explicit deduction guides* to control CTAD

- This allows arbitrary types to be deduced

» `std::vector` provides a deduction guide for the range constructor

```
template <typename InputIterator>
vector(InputIterator first, InputIterator last);
    // Range constructor

template <typename InputIterator>
vector(InputIterator first, InputIterator last)
    -> vector<std::iterator_traits_t<InputIterator>::value_type>;
    // Range constructor explicit deduction guide
```

» Thanks to the deduction guide, the `value_type` of the iterator is obtained

CTAD explicit deduction guides - example #1

» Deduction guides can be completely arbitrary

```
template <typename T>
struct s
{
    s(T) { }
};

template <typename T>
s(T) -> s<char>;
```

```
s s0{0};        // OK, deduced as `s<char>`
s s1{'a'};      // OK, deduced as `s<char>`
s s2{nullptr}; // Error: cannot convert `std::nullptr_t` to `char`
```

CTAD explicit deduction guides - example #2

» Deduction guides help with perfect forwarding

```
template <typename T>
struct wrapper
{
    T data;

    template <typename U>
    wrapper(U&& x) : data{std::forward<U>(x)} { }
};

template <typename U>
wrapper(U x) -> wrapper<U>;
    // Required: this is a "decaying" deduction guide
```

```
wrapper w0{100}; // OK, deduced as `wrapper<int>`

int i;
wrapper w1{i};   // OK, deduced as `wrapper<int>`
```

Section recap

- CTAD deduces class template parameters during construction of objects
  - Roughly same rules as good old FTAD, but there are some inconsistencies
- CTAD aims to remove the need for `make_xxx` functions
- CTAD helps with classes accepting closures or weird types during construction
- CTAD prioritizes copy/move constructors over wrapping ones (!)
  - Be careful with `std::tuple`, `std::optional`, and similar “nestable” types
  - Refrain from using CTAD in templates

Discussion

- Can you think of any use case where CTAD would help in any of your projects?

Exercise

- Implement a utility class to overload two lambdas leveraging CTAD
  - `exercise5.cpp`
    - on Wandbox
    - on Godbolt

Code Generation With Fold Expressions

Code Generation With Fold Expressions

In the past…

- Dealing with variadic *parameter packs* prior to C++17 is not easy
  - How to generate code for each element in the pack?
  - How to collapse the pack into a final single result?
- Some techniques can be used
  - Recursion
  - Arbitrary expansion context *(e.g.* `std::initializer_list`*)*

Adding elements together - C++11 with recursion

```
template <typename T>
auto add(const T& x)
{
    return x;
}

template <typename T, typename... Ts>
auto add(const T& x, const Ts&... xs)
{
    return x + add(xs...);
}
```

[live on Compiler Explorer]

- Requires two overloads
- Slow to compile

Adding elements together - C++11 with `std::initializer_list`

```
template <typename T, typename... Ts>
auto add(const T& x, const Ts&... xs)
{
    std::common_type_t<T, Ts...> acc{x};
    (void) std::initializer_list<bool>{
        ((acc += xs), true)...
    };

    return acc;
}
```

[live on Compiler Explorer]

- Arcane technique, hard to read and to explain
- Requires state and mutability
- Faster to compile

In C++17…

» C++17 introduces *fold expressions*

- They “collapse” a *parameter pack* into a single result, using a specified binary operator

```
template <typename... Ts>
auto add(const Ts&... xs)
{
    return (xs + ...);
}
```

[live on Compiler Explorer]

In C++17…

fold expression(since C++17)

Reduces (folds) a parameter pack over a binary operator.

Syntax

(*pack op* ...)	(1)
(... *op pack*)	(2)
(*pack op* ... *op init*)	(3)
(*init op* ... *op pack*)	(4)

1) unary right fold
2) unary left fold
3) binary right fold
4) binary left fold

op - any of the following 32 *binary* operators: `+` `-` `*` `/` `%` `^` `&` `|` `=` `<` `>` `<<` `>>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<=` `>>=` `==` `!=` `<=` `>=` `&&` `||` `,` `.*` `->*`. In a binary fold, both *op*s must be the same.

pack - an expression that contains an unexpanded parameter pack and does not contain an operator with precedence lower than cast at the top level (formally, a *cast-expression*)

init - an expression that does not contain an unexpanded parameter pack and does not contain an operator with precedence lower than cast at the top level (formally, a *cast-expression*)

Note that the open and closing parentheses are part of the fold expression.

Explanation

The instantiation of a *fold expression* expands the expression e as follows:

1) Unary right fold $(E \ op \ ...)$ becomes $(E_1 \ op \ (... \ op \ (E_{N-1} \ op \ E_N)))$
2) Unary left fold $(... \ op \ E)$ becomes $(((E_1 \ op \ E_2) \ op \ ...) \ op \ E_N)$
3) Binary right fold $(E \ op \ ... \ op \ I)$ becomes $(E_1 \ op \ (... \ op \ (E_{N-1} \ op \ (E_N \ op \ I))))$
4) Binary left fold $(I \ op \ ... \ op \ E)$ becomes $((((I \ op \ E_1) \ op \ E_2) \ op \ ...) \ op \ E_N)$

(where N is the number of elements in the pack expansion)

Fold expressions - example

```
template <typename... Xs>
void print(const Xs&... xs)
{
//              op         pack
//              v~         v~
    (std::cout << ... << xs);
//   ^~~~~~~~~~          ^~
//   init                op
}
```

» The above is a **binary left fold**

Explanation

The instantiation of a *fold expression* expands the expression e as follows:

1) Unary right fold $(E\ op\ ...)$ becomes $E_1\ op\ (...\ op\ (E_{N-1}\ op\ E_N))$

2) Unary left fold $(...\ op\ E)$ becomes $((E_1\ op\ E_2)\ op\ ...)\ op\ E_N$

3) Binary right fold $(E\ op\ ...\ op\ I)$ becomes $E_1\ op\ (...\ op\ (E_{N-1}\ op\ (E_N\ op\ I)))$

4) Binary left fold $(I\ op\ ...\ op\ E)$ becomes $(((I\ op\ E_1)\ op\ E_2)\ op\ ...)\ op\ E_N$

(where N is the number of elements in the pack expansion)

Fold expressions - example

```
template <typename... Xs>
void print(const Xs&... xs)
{
    (std::cout << ... << xs);
}
```

```
print(1, 'a', 2);
```

[live on Compiler Explorer]

```
((std::cout << 1) << 'a') << 2
```

Fold expressions - ordering

» Precedence, associativity, and sequencing order are given by the chosen **operator**, not by the parentheses

C++ Operator Precedence

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	`type``()` *`type`*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
...	...	...	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

9) Every value computation and side effect of the first (left) argument of the built-in comma operator `,` is *sequenced before* every value computation and side effect of the second (right) argument.

Fold expression - comma operator example

```
template <typename Vector, typename... Ts>
void push_back_all(Vector& vec, Ts&&... xs)
{
    (vec.push_back(std::forward<Ts>(xs)), ...);
}
```

```
push_back_all(vec, 1, 2, 3);
```

```
vec.push_back(1), (vec.push_back(2), vec.push_back(3));
// {1, 2, 3}
```

Fold expression - comma operator example

```
template <typename Vector, typename... Ts>
void push_back_all(Vector& vec, Ts&&... xs)
{
    (..., vec.push_back(std::forward<Ts>(xs)));
}
```

```
push_back_all(vec, 1, 2, 3);
```

```
(vec.push_back(1), vec.push_back(2)), vec.push_back(3);
// {1, 2, 3}
```

Fold expression - `print_with_spaces`

» **Q:** How would you implement the following?

```
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    // ?
}

print_with_spaces(1, 'a', 2, 'b');  // prints "1 a 2 b\n"
```

Fold expression - print_with_spaces

```
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    std::cout << x;
    ((std::cout << ' ' << xs), ...);
    std::cout << '\n';
}
```

- No recursion
  - Simpler
  - Faster to compile
  - Easier to read

Fold expression - use cases

- Fold expressions are useful whenever you need to…
  - …generate code for each element in a parameter pack;
  - …collapse a parameter pack into a single result.
- In practice, this translates to:
  - Helper functions that reduce boilerplate
  - Avoidance of recursion and speeding up compilation time

Fold expression - concatenation

```
template <typename... Ts>
std::string cat(Ts&&... xs)
{
    std::ostringstream oss;
    (oss << ... << xs);
    return oss.str();
}
```

```
std::cout << cat("meow", "purr") << '\n';
```

meowpurr

Fold expression - repeated comparisons

```
if(foo == 'a' || foo == 'c' || foo == 'e')
{
    // ...do something...
}
```

» `foo ==` is repeated multiple times

Fold expression - repeated comparisons

```
template <typename T, typename... Ts>
constexpr bool is_any_of(const T& x, const Ts&... xs)
{
    return ((x == xs) || ...);
}
```

```
if(is_any_of(foo, 'a', 'c', 'e'))
{
    // ...do something...
}
```