Fold expression - repeated comparisons

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

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

Fold expression - repeated comparisons

» Syntax can be improved with a helper class

```
if(any_of('a', 'b', 'c').is(foo))
{
    // ...do something...
}
```

Fold expression - compile-time unrolling

```
repeat<32>([](auto i)
{
    std::array<int, i> arr;
    // ...use `arr`...
});
```

- `i` is an `std::integral_constant`
- The closure is invoked 32 times

Fold expression - compile-time unrolling

```
template <auto N, typename F>
void repeat(F&& f)
{
    repeat_impl(f, std::make_index_sequence<N>{});
}
```

- `N` is explicitly provided by the user
- `F` is deduced
- `std::make_index_sequence` creates a compile-time integer sequence from 0 to *N (non-inclusive)*

Fold expression - compile-time unrolling

```
template <typename F, auto... Is>
void repeat_impl(F&& f, std::index_sequence<Is...>)
{
    (f(std::integral_constant<std::size_t, Is>{}), ...);
}
```

- “Match” the generated sequence into `Is...`
- Invoke `f` *N* times using a *fold expression* over the *comma operator*

Fold expression - compile-time unrolling

```
template <typename F, auto... Is>
void repeat_impl(F&& f, std::index_sequence<Is...>)
{
    (f(std::integral_constant<std::size_t, Is>{}), ...);
}

template <auto N, typename F>
void repeat(F&& f)
{
    repeat_impl(f, std::make_index_sequence<N>{});
}
```

https://wandbox.org/permlink/0WNWLn2s7Q6xtPdW

Fold expression - iteration over `std::tuple`

```
template <typename F, typename Tuple>
void for_tuple(F&& f, Tuple&& tuple)
{
    std::apply([&f](auto&&... xs)
    {
        (f(std::forward<decltype(xs)>(xs)), ...);
    }, std::forward<Tuple>(tuple));
}
```

- `std::apply` invokes a function by “unpacking” all the elements of a tuple as arguments
- The provided function uses a *fold expression* over the *comma operator* to invoke `f` for each tuple element

Fold expression - iteration over `std::tuple`

```
for_tuple([](const auto& x)
{
    std::cout << x;
}, std::tuple{1, 2, 'a', 'b'});
```

https://wandbox.org/permlink/3kRtPfP8TiM0PPGb

12ab

Fold expression - check typelist uniqueness

```
template <typename...>
inline constexpr auto is_unique = std::true_type{};

template <typename T, typename... Rest>
inline constexpr auto is_unique<T, Rest...> =
    std::bool_constant<(!std::is_same_v<T, Rest> && ...)
                        && is_unique<Rest...>>{};
```

- C++14 *variable templates* can be specialized
- Variables can be `inline` since C++17
- `std::bool_constant<X>` was introduced in C++17 - it’s an alias for `std::integral_constant<bool, X>`

Fold expression - check typelist uniqueness

BASE CASE

```
template <typename...>
inline constexpr auto is_unique = std::true_type{};
```

» An empty type list is unique

Fold expression - check typelist uniqueness

RECURSIVE CASE

```
template <typename T, typename... Rest>
inline constexpr auto is_unique<T, Rest...> =
    std::bool_constant<(!std::is_same_v<T, Rest> && ...)
                        && is_unique<Rest...>>{};
```

» `<T, Rest...>` type is unique if:

- `Rest...` does **not** contain `T`
- `<Rest...>` is an unique type list

» The “contains” check uses a *fold expression* over the `&&` operator

Fold expression - check typelist uniqueness

```
static_assert(is_unique<>);
static_assert(is_unique<int>);
static_assert(is_unique<int, float, double>);
static_assert(!is_unique<int, float, double, int>);
static_assert(!is_unique<int, float, double, int,
                         char, char>);
static_assert(is_unique<int, float, double, char>);
```

https://wandbox.org/permlink/tygPdyWf05xMjvTI

Section recap

- Fold expressions provide a clean way of reducing parameter packs
  - Useful to repeat an action for every element of a pack…
  - …or to collapse a pack into a single result

Discussion

- Use cases for metaprogramming in your projects

Exercise

- Implement compile-time loops with fold expressions
  - `exercise4.cpp`
    - on Wandbox
    - on Godbolt

Fold expression - iteration over a set of types (solution)

```
for_types<int, float, char>([](auto t)
{
    using type = typename decltype(t)::type;
    // ...use `type`...
});
```

- The passed closure is invoked for each type
- `t` is an empty object carrying information about the current type

Fold expression - iteration over a set of types (solution)

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

- `type_wrapper` stores information about a type inside an empty object that can be used like a value
- It will be passed to the user-provided lambda
- “Type-value encoding” idiom

Fold expression - iteration over a set of types (solution)

```
template <typename... Ts, typename F>
void for_types(F&& f)
{
    (f(type_wrapper<Ts>{}), ...);
}
```

- `Ts...` are explicitly provided by the user
- `F` is deduced
- A *fold expression* over the *comma operator* invokes `f` with every type

Fold expression - iteration over a set of types (solution)

```
struct A { void foo() { std::cout << "A\n"; } };
struct B { void foo() { std::cout << "B\n"; } };
struct C { void foo() { std::cout << "C\n"; } };

for_types<A, B, C>([](auto t)
{
    using type = typename decltype(t)::type;
    type{}.foo();
});
```

https://wandbox.org/permlink/8qtaDGbyL8gpKHAn

A
B
C

Performance Boost Via Copy Elision

Performance Boost Via Copy Elision

In this section

- Copy elision before C++17
- Guaranteed copy elision in C++17
- Value categories in C++17

Copy elision before C++17

- Prior to C++17, compilers were *allowed* (but not required) to elide copies/moves in some cases:
  - Return value optimization (RVO)
  - Named return value optimization (NRVO)
  - Passing a temporary by value to a function
  - Throwing and catching a temporary by value
  - Initialization of a variable from a temporary
- These “optimizations” can be disabled via `-fno-elide-constructors`

Sandbox

```cpp
#include <iostream>
#include <cstdio>
#include <type_traits>

struct s
{
    s()         { std::printf("s()\n"); }
    ~s()        { std::printf("~s()\n"); }
    s(const s&) { std::printf("s(const s&)\n"); }
    s(s&&)      { std::printf("s(s&&)\n"); }
};

s get_s_rvo()         { return s{}; }
s get_s_nrvo()        { s obj; return obj; }
void take_by_value(s) { }

int main()
{
    std::cout << "RVO ---------------------------\n";
    {
        s s0 = get_s_rvo();
    }
    std::cout << "-------------------------------\n\n";

    std::cout << "NRVO ---------------------------\n";
    {
        s s0 = get_s_nrvo();
    }
    std::cout << "-------------------------------\n\n";
```

x86-64 gcc (trunk) -std=c++11

Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 RVO ---------------------------
 -
 s()
```

x86-64 gcc (trunk) -std=c++11

Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 RVO ---------------------------
 -
 s()
```

Edit on Compiler Explorer

Limitations

- Not 100% reliable
- Even if operations are elided, they must still be available

```
struct fixed
{
    fixed() = default;

    fixed(const fixed&) = delete;
    fixed(fixed&&)      = delete;
};

fixed f0;             // OK
fixed f1 = fixed{}; // Error in C++11/14, even if copy elision occurs
```

[live on Compiler Explorer]

Guaranteed copy elision in C++17

- C++17 makes some instances of copy elision *mandatory*
  - Returning a *prvalue* from a function by value (RVO)
  - Initialization of a variable from a temporary (or nested chain)

Sandbox

```
#include <iostream>
#include <cstdio>
#include <type_traits>

struct s
{
    s()         { std::printf("s()\n"); }
    ~s()        { std::printf("~s()\n"); }
    s(const s&) { std::printf("s(const s&)\n"); }
    s(s&&)      { std::printf("s(s&&)\n"); }
};

s get_s_rvo()         { return s{}; }
s get_s_nrvo()        { s obj; return obj; }
void take_by_value(s) { }

int main()
{
    std::cout << "RVO ---------------------------\n";
    {
        s s0 = get_s_rvo(); // C++17 mandatory copy eli
    }
    std::cout << "-------------------------------\n\n";

    std::cout << "NRVO --------------------------\n";
    {
        s s0 = get_s_nrvo(); // No changes here
    }
    std::cout << "-------------------------------\n\n";
```

x86-64 gcc (trunk) -std=c++17

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 RVO ---------------------------
 -
 s()
```

x86-64 gcc (trunk) -std=c++17

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 RVO ---------------------------
 -
 s()
```

Edit on Compiler Explorer

Non-copyable/non-movable types

» Contrary to C++03/11/14, a copy/move constructor is *not* required if copy elision takes place

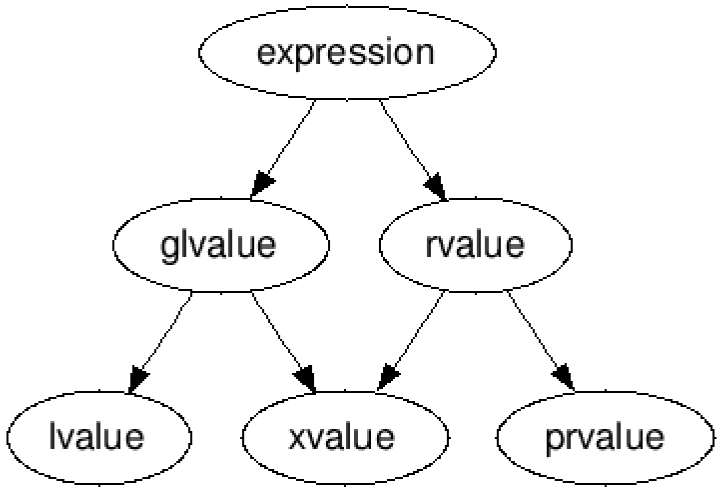
```
struct fixed
{
    fixed() = default;

    fixed(const fixed&) = delete;
    fixed(fixed&&)      = delete;
};

void foo(fixed) { }
```

```
fixed f0;             // OK
fixed f1 = fixed{}; // Error in C++11/14, OK in C++17
foo(fixed{});         // Error in C++11/14, OK in C++17
```

[live on Compiler Explorer]

Value categories in C++17

- » Mandatory copy elision relies on the redefinition of *value categories* in C++17
  - They have been simplified
- » The usual hierarchy still applies:



New meaning of *prvalue* and *glvalue*

- In C++17, *prvalue* and *glvalue* have a new meaning
  - A *glvalue* (generalized lvalue) is now defined as the “location of an object”
  - A *prvalue* is now defined as the “initializer of the object”
  - An *xvalue* is a *glvalue* that denotes an object whose resources can be reused
    - Usually because it is near the end of its lifetime (eXpiring)

Temporary materialization

- A *prvalue* is not an actual object anymore
- A *prvalue* of type `T` can be converted to an *xvalue* of type `T`
  - This process is called “temporary materialization conversion”

Whenever a prvalue appears in a context where an xvalue or glvalue is expected, the prvalue is converted to an xvalue

```
struct X { int n; }
int k = X().n; // OK, `X()` prvalue is converted to xvalue
```

Temporary materialization

- Materialization is delayed as long as possible in order to avoid creating unnecessary temporary objects
- It happens in the certain cases, among which:
  - When binding a reference to a *prvalue*
  - When performing member access on a class *prvalue*
- For cases like variable initialization, the wording is in terms of *direct-initialization*
  - E.g. *“…if the expression is a prvalue, the object is direct-initialized…”*

Section recap

- Compiler had opportunities to elide copies/moves in the past
  - But they were not required to do so
  - E.g., RVO, NRVO, initialization from temporary
- C++17 adds “mandatory copy elision” for RVO and temporary initialization
  - An available copy/move constructor is not required
  - Based on simplified value categories, temporary materialization, and direct-initialization

Discussion

- Have you ever been bitten by RVO?
- Opportunities for copy elision in your projects

Compile Time Branching With Constexpr If

Compile Time Branching With Constexpr If

In this section

» `if constexpr (...)`

In the past…

- Branching at compile-time was often cumbersome
- Regular `if` statements are not powerful enough

```
template <typename Component>
void registry::add_component(Component& component)
{
    if (has_initialize_v<Component>)
    {
        component.initialize();
    }

    track_component(component);
}
```

In the past…

```
template <typename Component>
void registry::add_component(Component& component)
{
    if (has_initialize_v<Component>) { component.initialize(); }
    track_component(component);
}
```

```
struct test_component { };

test_component tc;
registry r;
r.add_component(tc); // Compile-time error
```

```
error: 'test_component' has no member named 'initialize'
```

In the past…

» Even if the condition given to a regular `if` is a *constant expression*, all branches are instantiated

» A possible workaround is using *tag dispatch*

- Verbose and cumbersome

```
template <typename Component>
void registry::try_to_initialize(std::true_type, Component&);

template <typename Component>
void registry::try_to_initialize(std::false_type, Component&);
```

```
try_to_initialize(has_initialize<Component>{}, component);
```

In C++17…

- C++17 introduces a new construct, `if constexpr`

```
if constexpr (/* condition */)
{
    // `true` branch
}
else
{
    // `false` branch
}
```

- The provided `condition` **must** be a *constant expression*
- Only the taken branch is instantiated, the other one isn't

In C++17...

```
template <typename Component>
void registry::add_component(Component& component)
{
    if constexpr (has_initialize_v<Component>) { component.initialize(); }
//      ^~~~~~~~~
    track_component(component);
}
```

```
struct test_component { };
struct init_component { void initialize(); };

test_component tc;
init_component ic;

registry r;
r.add_component(tc); // OK
r.add_component(ic); // OK, calls `ic.initialize()`
```

if constexpr - recursive variadic function templates

```
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    std::cout << x;
    if constexpr (sizeof...(Ts) == 0)
    {
        std::cout << '\n';
    }
    else
    {
        std::cout << ' ';
        print_with_spaces(xs...);
    }
}
```

- Less efficient than a fold expression
- More flexible (e.g. graph navigation)

https://gcc.godbolt.org/z/S5-D9g

if constexpr - type traits

```
template <typename T>
constexpr bool fuzzy_equality(const T& x, const T& y)
{
    if constexpr (std::is_floating_point_v<T>)
    {
        return std::abs(x - y) < T(0.0001);
    }
    else
    {
        return a == b;
    }
}
```

Closing Thoughts

- `if constexpr` greatly simplifies branching at compile-time
  - Supersedes template trickery in most cases
  - Not powerful enough in others (e.g. generating data members)
- Compared to *overloading* or *template specialization*, `if constexpr`
  - …is more readable and requires less boilerplate;
  - …is faster at compile-time;
  - …is more “closed”, users cannot add new branches.
- There is no `constexpr` *ternary operator*

Algebraic Data Types:

`std::variant`

Algebraic Data Types:

`std::variant`

In this section

- What a “variant” is
- `std::variant`
- Variant visitation
- Use cases for variants

Understanding variants

- `struct` → `enum class` → `variant`
- *Product types* and *sum types*
- *Variants* vs *unions*

What is a `struct`?

A `struct` models **aggregation of types**.

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

A `point` is an `int` **AND** an `int`.

What is an `enum class`?

An `enum class` models **a choice between values**.

```
enum class traffic_light
{
    red,
    yellow,
    green
};
```

A `traffic_light` is **EITHER** `red` **OR** `yellow` **OR** `green`.

What is a `variant`?

A `variant` models **a choice between types**.

```
struct on  { int _temperature; };
struct off { };

using oven_state = std::variant<on, off>;
```

- The oven is `off`.

  …or…

- The oven is `on`, with a certain `_temperature` value.

From `struct` to `variant`

.	`struct`	`enum class`	`variant`
model	*aggregation:* types	*choice:* values	*choice:* types
class	product type	sum type	sum type

Product types

- `struct` is an example of a *product type*.
- The **total number of its possible states** is equal to the **product** of the number of possible states of its members.

```
struct foo
{
    int _a;
    bool _b;
};
```

states(*foo*) = *states*(*int*) * *states*(*bool*)

Sum types

- `variant` is an example of a *sum type*.
- The **total number of its possible states** is equal to the **sum** of the number of possible states of its alternatives.

```
using foo = std::variant<int, bool>;
```

states(*foo*) = *states*(*int*) + *states*(*bool*)

Variants vs unions

» Variant types can be thought of as **type-safe tagged unions** that:

- Require significantly less boilerplate
- Automatically deal with constructors/destructors and assignment
- Immensely increase **safety**

» Similarly to unions, `std::variant` requires no *dynamic allocation*

- The size of a `std::variant<Ts...>` is the `max(sizeof(Ts)...)`

`std::variant` - Basic interface

std::variant

Defined in header `<variant>`

```
template <class... Types>
class variant;
```

(since C++17)

- `std::variant` is a *variadic template class*
- The passed `Types...` are commonly called “alternatives”

```
using v0 = std::variant<int, float>;
using v1 = std::variant<std::string, bool, char>;
```

`std::variant` - Default constructor

- The *default constructor* of `std::variant` will create a variant with its **first alternative**, value-initialized.

```
std::variant<int, bool> v0;
// `v0` contains an `int` with value `0`

std::variant<bool, int> v1;
// `v1` contains a `bool` with value `false`
```

`std::variant` - `T` constructor

» `std::variant<Ts...>` can be constructed with an instance of any of its alternatives.

```
std::variant<int, bool, char> v0{42};
// `v0` contains an `int` with value `42`

std::variant<int, bool, char> v1{true};
// `v1` contains a `bool` with value `true`

std::variant<int, bool, char> v2{'a'};
// `v2` contains an `char` with value `'a'`
```

`std::variant` - `T` constructor

- Be careful with *implicit conversions*

```
std::variant<std::string> v0("hello");
// OK

std::variant<std::string, std::string> v1("hello");
// Compilation error due to ambiguity

std::variant<std::string, bool> v2("hello");
// OK, chooses `bool` (!) (Fixed by P0608)
```

`std::variant` - Copy/move constructors

- Variants of the same type can be copy/move-constructed
- The copy/move constructor of the *active alternative* will be invoked

```
std::variant<bool, int> v0{42};

std::variant<bool, int> v1{v0};
// copy-construction

std::variant<bool, int> v2{std::move(v1)};
// move-construction
```

`std::variant` - In-place constructors

```
template< class T, class... Args >
constexpr explicit variant(std::in_place_type_t<T>, Args&&... args);
```

```
template< class T, class U, class... Args >
constexpr explicit variant(std::in_place_type_t<T>,
                           std::initializer_list<U> il, Args&&... args);
```

```
template< std::size_t I, class... Args >
constexpr explicit variant(std::in_place_index_t<I>, Args&&... args);
```

```
template< std::size_t I, class U, class... Args >
constexpr explicit variant(std::in_place_index_t<I>,
                           std::initializer_list<U> il, Args&&... args);
```

- `args...` are perfectly-forwarded to construct the desired alternative in-place *(i.e. no unnecessary temporaries are created)*

std::variant - In-place constructors

```
struct A { A(int) { } };
struct B { B(int) { } };
```

```
std::variant<A, B> v0{std::in_place_type<A>, 42};
// `v0` contains `A`, initialized with `42`

std::variant<A, B> v1{std::in_place_type<B>, 1234};
// `v1` contains `B`, initialized with `1234`

std::variant<A, B> v2{std::in_place_index<0>, 999};
// `v2` contains `A`, initialized with `999`
```

std::variant - Assignment

- Variants support copy/move assignment and assignment from any of their alternative types

```
std::variant<int, char> v0;
v0 = 'a';

std::variant<int, char> v1;
v1 = v0;
```

`std::variant` - Checking active alternative

- The currently active alternative of a variant can be checked with:
  - `std::holds_alternative<T>`
  - `variant::index()`

```
std::variant<int, char> v0{'a'};

assert(std::holds_alternative<char>(v0));
assert(v0.index() == 1);
```

`std::variant` - Accessing active alternative

- The active alternative in an `std::variant` instance can be accessed with any of the following:
  - `std::get<T>`
  - `std::get_if<T>`

`std::variant` - Accessing active alternative

```
std::variant<int, std::string> v0{1};

assert(std::holds_alternative<int>(v0));
assert(std::get<int>(v0) == 1);
```

- `get<T>` requires the user to be aware of the currently active alternative of the variant. In case of error, an *exception* will be thrown.

`std::variant` - Accessing active alternative

```
std::variant<int, std::string> v0{1};

auto* s = std::get_if<std::string>(&v0);
if(s != nullptr)
{
    // ...
}
```

- `get_if<T>` returns a pointer to the object if the *active alternative* is `T`, otherwise `nullptr`.

std::variant - Usage example

```
std::variant<admin, moderator, guest> level
    = read_level(current_user);

if(auto* l = std::get_if<admin>(&level))
{
    l->grant_admin_permissions();
}
else if(auto* l = std::get_if<moderator>(&level))
{
    l->grant_moderator_permissions();
}

// ...
```

`std::variant` - Visitation

- What is *"visitation"*?
- Shortcomings of `get` and `get_if`
- `std::visit`

Variant Visitation

- Visitation can be defined as an **abstraction** over accessing the currently active variant *alternative* in an **exhaustive** and **expressive** manner
- Think about “unpacking” the object inside a `variant`, and dispatching to an handler depending on its type

`std::visit`

- `std::visit` requires a `Callable` object which can be invoked with every possible variant alternative.
- The “traditional” way of creating such as object is defining a `struct`.

std::visit - Single variant

```
struct printer
{
    void operator()(int x)    { cout << x << "i\n"; }
    void operator()(float x)  { cout << x << "f\n"; }
    void operator()(double x) { cout << x << "d\n"; }
};
```

```
using my_variant = std::variant<int, float, double>;
my_variant v0{20.f};

// Prints "20f".
std::visit(printer{}, v0);
```

[live on Compiler Explorer]

`std::visit` - Single variant

- `printer` is a “visitor” - it must be invocable with **every** alternative type of the variant being visited
- `std::visit` invokes the correct overload of `printer`’s `operator()` by passing the variant’s currently active alternative

`std::visit` - Multiple variants

```cpp
struct collision_detector
{
    void operator()(circle, circle) { /* ... */ }
    void operator()(circle, rect)   { /* ... */ }
    void operator()(rect,   circle) { /* ... */ }
    void operator()(rect,   rect)   { /* ... */ }
};
```

```cpp
using my_variant = std::variant<circle, rect>;
my_variant v0{circle{}};
my_variant v1{rect{}};

std::visit(collision_detector{}, v0, v1);
```

[live on Compiler Explorer]

`std::visit` - Multiple variants

- `std::visit` can take any number of variants as arguments: this results in **multiple dispatch**
- The passed visitor must be invocable with **every combination** of alternative types of the variants being visited

`std::visit` - With generic lambda

```
std::variant<int, float, char> v0{20.f};

std::visit([](auto x) {
    if constexpr(std::is_same_v<decltype(x), int>) {
        cout << x << "i\n";
    }
    else if constexpr(std::is_same_v<decltype(x), float>) {
        cout << x << "f\n";
    }
    else if constexpr(std::is_same_v<decltype(x), char>) {
        cout << x << "c\n";
    }
}, v0);
```