

Implementing variant visitation using lambdas

Bloomberg

Vittorio Romeo

<https://vittorioromeo.info>
vittorio.romeo@outlook.com

ACCU 2017

29/04/2017

About me

- Developer at **Bloomberg L.P.**
- Modern C++ enthusiast
 - Conference talks
 - Video tutorials & articles
 - Open-source projects

Overview

1. What is a *variant* and why is it useful?
2. Variant visitation
3. Recursive variants

part 1

What is a *variant* and why is it useful?

- Variant types and use cases
- Variant implementations
- `std::variant`

To understand variants, let's begin by understanding `struct` and `enum class`.

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 `_temperature` = 200°C.

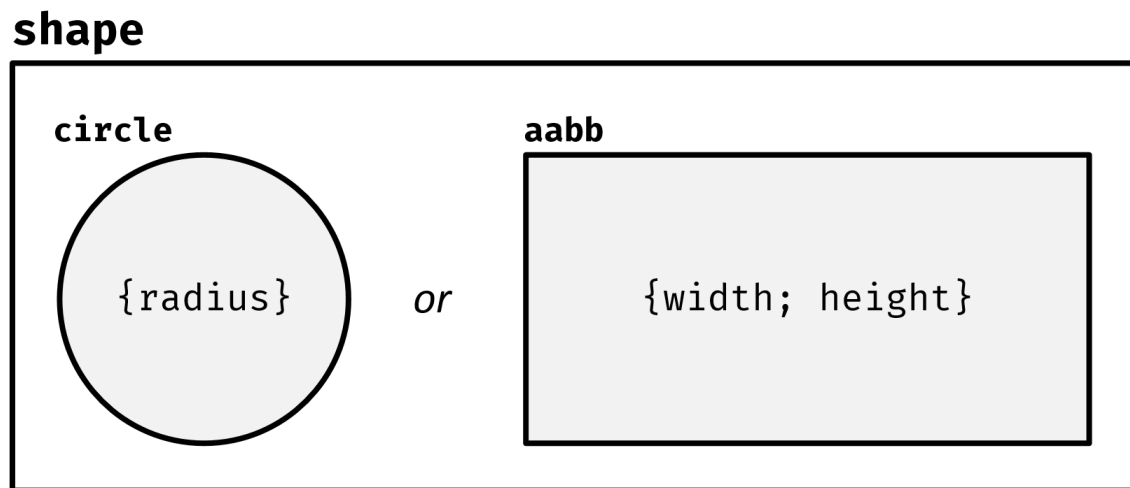
| | struct | enum class | variant |
|--------------|---------------------------|-----------------------|----------------------|
| model | <i>aggregation: types</i> | <i>choice: values</i> | <i>choice: types</i> |
| class | product type | sum type | sum type |

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**.

Those claims are easy to verify through an *example*. Let's...

- Define a `shape` **sum type** which can be a **circle** or an **axis-aligned bounding box**.
- Define an `area` function that, given a `shape` instance, calculates and returns its **surface area**.



We will start with a "*traditional*" *tagged union* first.

```
enum class shape_type{circle, aabb};

struct shape
{
    shape_type _type;

    union
    {
        struct { float _radius;          } circle_data;
        struct { float _width, _height; } aabb_data;
    } _u;
};
```


When defining a "*traditional*" tagged union type, we need:

- An `enum class` with a value for every *alternative*.
- A `struct` for every alternative that defines its *state*.
- An `union` that wraps all the state `struct` types.
- An interface `struct` that wraps the `union` and `enum class`.

```

auto area(const shape& s)
{
    switch(s._type)
    {
        case shape_type::circle:
        {
            const auto& r = s._u.circle_data._radius;
            return pi * r * r;
        }
        case shape_type::aabb:
        {
            const auto& x = s._u.aabb_data;
            return x._width * x._height;
        }
    };

    assert(false);
    __builtin_unreachable();
}

```

[on [gcc.godbolt.org](https://gcc.gnu.org/godbolt.org)]

When visiting it, we need:

- A `switch` with a `case` for every alternative.
- Every `case` needs to access the relevant state by going through the `union`.
- An *unreachable* guard at the end of the visitation.

Let's now see how `std::variant` compares.

```
struct circle { float _radius; };  
struct aabb { float _width, _height; };  
using shape = std::variant<circle, aabb>;
```

```
auto area(const shape& s)  
{  
    struct {  
        auto operator()(const circle& x) const  
        {  
            return pi * x._radius * x._radius;  
        }  
        auto operator()(const aabb& x) const  
        {  
            return x._width * x._height;  
        }  
    } visitor;  
  
    return std::visit(visitor, s);  
}
```

[on gcc.godbolt.org]

- Defining a variant type is trivial: all that's required is listing the alternative types.
- Visitation allows easy access to the members of currently active variant alternative.
- `visitor` is checked to be *exhaustive* at compile-time.

Examples of structures elegantly modeled by variants are:

- JSON
- ASTs (*abstract syntax trees*)
- State machines
- Error handling

```
using json_value = std::variant  
<  
    json_object,  
    json_array,  
    json_string,  
    json_number,  
    json_boolean,  
    json_null  
>;
```

```
template <typename T>  
json_value serialize(const T&);  
  
template <typename T>  
T deserialize(const json_value&);
```



```
struct success          { data _contents; }  
struct file_not_found  { };  
struct invalid_permissions { };  
  
using file_open_result = std::variant  
<  
    success,  
    file_not_found,  
    invalid_permissions  
>;
```

```
file_open_result open_file(const std::string& path);
```

The examples used `std::variant`, standardized in C++17.

Several other implementations are available today:

- `boost::variant`
- `eggs::variant`
- `type_safe::variant`
- `bdlb::Variant`

They differ in their:

- Interface
- Default initialization semantics
- Existence of an "empty" state
- Strategy of dealing with exceptions
- Rules for duplicate types

The visitation techniques that will be covered in this talk will be applicable to **any variant implementation**.

For simplicity, we're going to keep using `std::variant` for the rest of the talk.

`std::variant<Ts ... >` can be constructed/assigned with any instance of `Ts ...` or with any other `variant` instance.

```
std::variant<int, std::string> v0{1};  
v0 = "hello!"s;  
  
std::variant<int, std::string> v1{"bye!"s};  
v0 = std::move(v1);
```

The active alternative in an `std::variant` instance can be accessed with any of the following:

- `std::variant::get`
- `std::variant::get_if`

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

assert(v0.holds_alternative<int>());
assert(v0.get<int>() == 1);

auto v0_str = v0.get_if<std::string>();
if(v0_str != nullptr)
{
    // ...
}
```

- `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.
- `get_if<T>` requires syntactical overhead that does not scale well with a large number of types.

That's why the standard library provides `std::visit`, which allows **variant visitation**. We'll take a look at it in **part 2**.

part 1 - recap

- Variants are **sum types** that model a **choice between types**.
- Variants are **safer and more expressive** than "*traditional tagged unions*".
- *Recursive node-based structures, error handling, state machines, etc...* are examples of concepts that can be **elegantly** modeled with variants.
- `std::variant` is a **vocabulary type** introduced in C++17.

part 2

Variant visitation

- What is "*visitation*"?
- "Traditional" visitation
- "Lambda-based" visitation
 - Arbitrary *function object* overloading

What is "*visitation*"?

Visitation can be defined as an **abstraction** over accessing the currently active variant *alternative* in an **exhaustive** and **expressive** manner.

"Traditional" visitation

overview

- Visitation 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`.

"Traditional" visitation

example - one 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);
```

[on [gcc.godbolt.org](https://gcc.gnu.org/godbolt.org)]

"Traditional" visitation

example - two variants

```
struct collision_resolver
{
    void operator()(circle, circle) { /* ... */ }
    void operator()(circle, aabb)   { /* ... */ }
    void operator()(aabb, circle)  { /* ... */ }
    void operator()(aabb, aabb)    { /* ... */ }
};
```

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

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

[on gcc.godbolt.org]

"Traditional" visitation

shortcomings

- **Syntactical overhead:** a `struct` with multiple `operator()` overloads must be defined.
- **Lack of locality:** sometimes the `struct` cannot be defined locally (*e.g. contains template methods*).
- **Readability impact:** the visitation logic is defined far away from the visitation site.

| Can we do better?

Let's take some inspiration from *Rust*.

Rust *variants*: `enum`

example

```
enum MyVariant {  
    IntTag(i32),  
    FloatTag(f32),  
    DoubleTag(f64)  
}
```

```
let v0 = FloatTag(2.0);  
match v0 {  
    IntTag(x)    => println!("{}", x),  
    FloatTag(x) => println!("{}", x),  
    DoubleTag(x) => println!("{}", x)  
}
```

[on rust.godbolt.org]

Rust has *language-level variants* and **pattern-matching**. This allows expressive and safe local visitation of variants.

```
match v0 {  
    // IntTag(x)      ⇒ println!("{}", x),  
    //   ^^^^^^^^^  
    //   pattern  
  
    // FloatTag(x)   ⇒ println!("{}", x),  
    //   ^^^  
    //   decomposition  
  
    // DoubleTag(x) ⇒ println!("{}", x)  
    //   ^^^^^^^^^^^^^^^^^^^^^^^^^  
    //   expression  
}
```

C++

```
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"; }
};

my_variant v0{20.f};
std::visit(printer{}, v0);
```

Rust

```
let v0 = FloatTag(2.0);
match v0 {
    IntTag(x)      => println!("{}i", x),
    FloatTag(x)    => println!("{}f", x),
    DoubleTag(x)   => println!("{}d", x)
}
```

"Lambda-based" visitation

overview

- Visitation is done **locally** by using a set of exhaustive *lambda expressions*.
- "*Pattern matching*"-like syntax.
- Minimal syntactical boilerplate.
- No additional run-time overhead.

"Lambda-based" visitation

example

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

// Prints "20f".
match(v0)([](int x) { cout << x << "i\n"; },
          [](float x) { cout << x << "f\n"; },
          [](double x) { cout << x << "d\n"; });
```

Almost no syntactical overhead!

Let's compare again to Rust's `match` ...

C++

```
my_variant v0{20.f};

// Prints "20f".
match(v0)([](int x)    { cout << x << "i\n"; },
          [](float x)  { cout << x << "f\n"; },
          [](double x) { cout << x << "d\n"; });
```

Rust

```
let v0 = FloatTag(2.0);
match v0 {
    IntTag(x)    => println!("{}", x),
    FloatTag(x) => println!("{}", x),
    DoubleTag(x) => println!("{}", x)
}
```

(Note: Rust's `match` can be way more powerful than shown here.)

Before diving into `match`'s implementation, let's look at an additional example.

"Lambda-based" visitation

example

```
struct payload { int _data; };  
struct error   { std::string _what; };  
  
using request = std::variant<payload, error>;
```

```
match(make_request("legitwebsite.com"))(  
    [](const payload& p)  
    {  
        match(make_request("abc.com", p._data))(  
            [](const payload& p){ /* ... */ },  
            [](const error& e)  { /* ... */ });  
    },  
    [](const error& e){ /* ... */ });
```

"Lambda-based" visitation

implementation - overview

- `match` will be a function that takes N_v *variants* and returns a function that takes N_f *function objects*.

```
match(v0, v1, ... , vN_v)(f0, f1, ... , fN_f);
```

- In order to create a *visitor* from the passed function objects, an *overload set* must be built out of them.
- Internally, `std::visit` will be called with the *variants* and the newly-built *overload set*.

Let's begin with the creation of an **overload set**.

Let's start by looking at *non-member functions*.

```
int foo(float) { return 0; }  
int foo(char)  { return 1; }
```

`foo` is an *overloaded function*.

```
auto x0 = foo(0.f); // `x0` is `0`.  
auto x1 = foo('a'); // `x1` is `1`.
```

Can we "*generate*" an overload set out of them?

There is no way of defining *overloaded functions* locally:

```
void f()  
{  
    int foo(float) { return 0; }  
    int foo(char)  { return 1; }  
    // Nope. COMPILER ERROR!  
}
```

...this doesn't help.

| What about *function objects*?

```
struct foo
{
    int operator()(float) { return 0; }
    int operator()(char)  { return 1; }
};
```

`foo` is a *function object*.

```
auto x0 = foo{}(0.f); // `x0` is `0`.
auto x1 = foo{}('a'); // `x1` is `1`.
```

Can we "*generate*" an overload set out of them?

We can define them locally:

```
void f()  
{  
    struct foo  
    {  
        int operator()(float) { return 0; }  
        int operator()(char) { return 1; }  
    };  
}
```


We can also compose them via inheritance!

```
struct foo_float { int operator()(float) { return 0; } };  
struct foo_char { int operator()(char) { return 1; } };
```

```
struct foo : foo_float, foo_char  
{  
    using foo_float::operator();  
    using foo_char::operator();  
};
```

```
auto x0 = foo{}(0.f); // `x0` is `0`.  
auto x1 = foo{}('a'); // `x1` is `1`.
```

| "Why do we need the `using` declarations?"

```
struct foo : foo_float, foo_char
{
    // using foo_float::operator();
    // using foo_char::operator();
};

foo{}(0.f); // Nope. COMPILER ERROR!
```

[on gcc.godbolt.org]

Without the `using-declarations` the code would generate a compiler error.

The reason is that the call to `foo::operator()` would be **ambiguous** because *name resolution* is performed before *overload resolution*. See [§13.2 "Member name lookup"](#).

We're on the right track - all that's needed is some *generalization* via `template` metaprogramming.

First attempt:

```
template <typename ... TFs>
struct overload_set : TFs ...
{
    using TFs :: operator() ... ;
};
```

```
using foo = overload_set<foo_float, foo_char>;
auto x0 = foo{}(0.f);
auto x1 = foo{}('a');
```

...will it work?

The following code...

```
template <typename ... TFs>
struct overload_set : TFs ...
{
    using TFs :: operator() ... ;
};
```

...works perfectly in C++17, thanks to [P0195](#) ("*Pack expansions in using-declarations*").

It unfortunately causes a compiler error in C++14:

```
error: parameter packs not expanded with '...':
struct overload_set : TFs ...
{ using TFs::operator() ... ; };
//                ^^^^
```

This happens because C++14 `using`-*declarations* don't support multiple comma-separated items. E.g.

```
struct a : b, c
{
    using b::foo, c::bar; // Error before C++17.
};
```

The problems therefore are:

- Only one of the base classes' `operator()` can be introduced in the scope of `overload_set` with the `using` keyword.
- There's no way of generating all the `using`-*declarations* at once.

...how can we work around these limitations?

The first one gives us an hint: let's try to separately match the first base class.


```
template <typename TF, typename ... TFs>
struct overload_set : TF, TFs ...
{
    using TF::operator();
    // ... what to do with `TFs ...`?
};
```

Can you see the pattern? This looks like a job for **recursion!**

Recursive case:

```
template <typename TF, typename ... TFs>
struct overload_set : TF, overload_set<TFs ... >
{
    using TF :: operator();
    using overload_set<TFs ... > :: operator();
};
```

Base case (*specialization*):

```
template <typename TF>
struct overload_set<TF> : TF
{
    using TF :: operator();
};
```

Great! Let's try out our new `overload_set` :

```
struct foo_float
{
    constexpr int operator((float) { return 0; }
};

struct foo_char
{
    constexpr int operator((char) { return 1; }
};
```

```
overload_set<foo_float, foo_char> o;

static_assert(o(0.f) == 0);
static_assert(o('a') == 1);
```

[\[on gcc.godbolt.org\]](https://gcc.gnu.org/godbolt.org)

In order to support *lambda expressions* and automatically deduce the passed *function object* types, we need to make two improvements:

- Have a *perfect-forwarding* constructor inside `overload_set`.
 - Lambdas are **not** `DefaultConstructible` !
- Provide an `overload` interface *variadic function template* that deduces the types of *function objects* and returns an `overload_set`.

The *base case's perfect-forwarding* constructor is trivial to implement:

```
template <typename TF>
struct overload_set<TF> : TF
{
    using TF::operator();

    template <typename TFFwd>
    overload_set(TFFwd&& f)
        : TF{std::forward<TFFwd>(f)}
    {
    }
};
```

Note that we're not using `TF` as the type of the `f` argument because we want a *forwarding reference*, not an *rvalue reference*.

The *recursive case's* constructor is slightly more complicated:

```
template <typename TF, typename ... TFs>
struct overload_set : TF, overload_set<TFs ... >
{
    using TF::operator();
    using overload_set<TFs ... >::operator();

    template <typename TFFwd, typename ... TRest>
    overload_set(TFFwd&& f, TRest&& ... rest) :

        // Construct the current function.
        TF{std::forward<TFFwd>(f)},

        // Construct the base class.
        // Pay attention to the ellipses' positions.
        overload_set<TFs ... >{
            std::forward<TRest>(rest) ... }
    { }
};
```

The last piece of the puzzle is the aforementioned `overload` *variadic function template* interface:

```
template <typename ... TFs>
auto overload(TFs&& ... fs)
{
    return
        overload_set<std::remove_reference_t<TFs> ... >(
            std::forward<TFs>(fs) ... );
}
```

`std::remove_reference_t` is used to convert `T&` to `T` for the special *lvalue forwarding-reference* template deduction rules.

With everything in place, we can finally write the code below:

```
auto o = overload([](float){ return 0; },
                 [](char) { return 1; });

static_assert(o(0.f) == 0);
static_assert(o('a') == 1);
```

[on gcc.godbolt.org]

Note that lambdas in C++17 are *implicitly* `constexpr` if possible - `static_assert` therefore works with them. (See [N4487](#) and [P0170](#).)

Now that we have a working `overload_set`, let's go back to the original task: making the snippet below compile and work.

```
using my_variant = std::variant<int, float, double>;  
my_variant v0{20.f};  
  
// Prints "20f".  
match(v0)(  
    [](int x) { cout << x << "i\n"; },  
    [](float x) { cout << x << "f\n"; },  
    [](double x) { cout << x << "d\n"; });
```

Let's implement `match` !

Remember:

`match` will be a function that takes N_v *variants* and returns a function that takes N_f *function objects*.

```
match(v0, v1, ..., vN_v)(f0, f1, ..., fN_f);
```

Therefore it will roughly look like this:

```
template <typename ... TVariants>
auto match(TVariants&& ... vs)
{
    return [](auto&& ... fs){ /* ... */ };
}
```

Thanks to `overload_set` and `std::visit`, the real implementation is *trivial*.

```
template <typename ... TVariants>
constexpr auto match(TVariants&& ... vs)
{
    return [&vs ... ](auto&& ... fs) → decltype(auto)
    {
        auto visitor =
            overload(std::forward<decltype(fs)>(fs) ... );

        return std::visit(visitor,
            std::forward<TVariants>(vs) ... );
    };
}
```

Finally...

```
using my_variant = std::variant<int, float, double>;  
my_variant v0{20.f};  
  
// Prints "20f".  
match(v0)([](int x) { cout << x << "i\n"; },  
          [](float x) { cout << x << "f\n"; },  
          [](double x) { cout << x << "d\n"; });
```

[on gcc.godbolt.org] | [on melpon.org/wandbox]

...compiles and prints "20f" !

Here's an example with *multiple variants*:

```
struct circle { /* ... */ };
struct aabb   { /* ... */ };

using shape = std::variant<circle, aabb>;
shape s0{circle{}};
shape s1{aabb{}};

match(s0, s1)(
    [](circle, circle){ cout << "circle vs circle\n"; },
    [](circle, aabb)  { cout << "circle vs aabb\n"; },
    [](aabb,  circle){ cout << "aabb vs circle\n"; },
    [](aabb,  aabb)  { cout << "aabb vs aabb\n"; });
```

[on gcc.godbolt.org] | [on melpon.org/wandbox]

One more, with *nested variants*:

```
struct format { /* ... */ };  
struct timeout { /* ... */ };  
using error = std::variant<format, timeout>;  
  
struct accept { /* ... */ };  
struct reject { /* ... */ };  
using ok = std::variant<accept, reject>;
```

```
using response = std::variant<error, ok>;  
response r{error{timeout{}}};  
  
match(r)(  
    [](ok x) { match(x)([](accept) { /* ... */ },  
                        [](reject) { /* ... */ }); },  
  
    [](error x){ match(x)([](format) { /* ... */ },  
                          [](timeout){ /* ... */ }); });
```

[on gcc.godbolt.org] | [on melpon.org/wandbox]

part 2 - recap

- Variant visitation works by invoking the *overload* matching the active *alternative* on an overloaded visitor *function object*.
- `using`-*declarations* and *inheritance* can be used to **create overloads** of arbitrary *function objects*.
- Lambdas are just syntactic sugar for *function objects*.
- Overloads of lambdas built *on-the-spot* **reduce boilerplate** and **increase locality** when visiting variants.

part 2 - recap

- Note that `struct`-based "traditional" visitation is **still useful!**
- `struct` visitors can...
 - Contain internal state and a richer interface.
 - Be more easily shared/reused in a large codebase.
 - Provide *customization points* (e.g. via a `template` parameter).
- If your *visitation logic* is simple and local, prefer "*lambda-based*" visitation.

match can be implemented in less than
40 *well-formatted* lines of C++14 code

"...what's the catch?"

part 3

Recursive variants

- What is a "*recursive variant*"?
- "Traditional" recursive visitation
- "Lambda-based" recursive visitation
 - Recursive lambdas via **Y combinator**

A "*recursive*" variant is a variant which can **contain itself**, and can be used to represent **recursive structures** (*e.g. JSON objects*).

Let's assume we want to define a *recursive variant* that models simple arithmetical expressions.

| | |
|---------------|---|
| 5 | <code>expr<number></code> |
| $9 + 3$ | <code>expr<number + expr<number>></code> |
| $1 - (3 + 7)$ | <code>expr<number - expr<number + expr<number>>></code> |

Here's a possible grammar:

```
<number> ::= `int`  
<op>      ::= `plus` | `minus`  
<expr>    ::= <number> | <number> <op> <expr>
```

```
<number> ::= `int`  
<op>      ::= `plus` | `minus`  
<expr>    ::= <number> | <number> <op> <expr>
```

```
using number = int;  
  
struct plus { };  
struct minus { };  
using op = std::variant<plus, minus>;  
  
using expr = std::variant<number,  
                                     std::tuple<number, op, expr>>;
```

[\[on gcc.godbolt.org\]](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86441)

error: 'expr' was not declared in this scope

The previous error makes sense, because we're basically trying to define:

```
using r_expr = std::tuple<number, op, expr>;  
using expr = std::variant<number, r_expr>;
```

Could a *forward-declaration* help?

```
struct expr;

using r_expr = std::tuple<number, op, expr>;

struct expr
{
    std::variant<number, r_expr> _data;
};
```

[on [gcc.godbolt.org](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86312)]

error: invalid use of incomplete type 'struct expr'

Again, the error makes sense:

sizeof(expr) = ∞

Introducing a *layer of indirection* solves the issue...

```
struct expr;

using r_expr = std::tuple<number, op, expr>;

struct expr
{
    std::variant<number, std::unique_ptr<r_expr>> _data;
};
```

[on gcc.godbolt.org]

- The size of any instantiation of `std::variant` must be fixed.
- Recursive variants can be defined by using *forward-declarations* and *indirection*.

Let's also add a *perfect-forwarding* constructor to simplify usage:

```
struct expr
{
    std::variant<number, std::unique_ptr<r_expr>> _data;

    template <typename ... Ts>
    expr(Ts&& ... xs) : _data(std::forward<Ts>(xs) ... )
    {
    }
};
```

```
expr e{5};
```

| | |
|----|-------------|
| e0 | 5 |
| e1 | 9 + 3 |
| e2 | 1 - (3 + 7) |

```
expr e0{5};  
  
expr e1{make_unique<r_expr>(9, plus{}, 3)};  
  
expr e2{make_unique<r_expr>(1, minus{},  
                           make_unique<r_expr>(3, plus{}, 7))};
```

[on [gcc.godbolt.org](https://gcc.gnu.org/godbolt.org)]

|"How can we visit recursive variants?"

Recursive "traditional" visitation

overview

- A `struct` with overloaded `operator()` will be used.
- One or more `operator()` overloads will *recursively visit* the variant by invoking `std::visit` on the parent `struct`.

Recursive "traditional" visitation example

```
struct evaluator
{
    auto operator()(number x) { return x; }
    auto operator()(const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        const auto rest = std::visit(*this, rhs._data);

        return match(op)(
            [&](plus) { return lhs + rest; },
            [&](minus){ return lhs - rest; });
    }
};
```

```
// 5
expr e0{5};

// 9 + 3
expr e1{make_unique<r_expr>(9, plus{}, 3)};

// 1 - (3 + 7)
expr e2{make_unique<r_expr>(1, minus{},
    make_unique<r_expr>(3, plus{}, 7))};
```

```
cout << std::visit(evaluator{}, e0._data); // "5"
cout << std::visit(evaluator{}, e1._data); // "12"
cout << std::visit(evaluator{}, e2._data); // "-9"
```

[on [gcc.godbolt.org](https://gcc.gnu.org/godbolt.org)] | [on melpon.org/wandbox]

Recursive "traditional" visitation

shortcomings

- **Syntactical overhead, lack of locality, and readability impact.**
- **Explicit nested visitation:** recursing requires an explicit call to `std::visit` inside the visitation logic.

We can make a small improvement here.

Did you notice that we're forced to specify the `_data` member while using `std::visit` ?

Let's create a `visit_recursively` wrapper that does that for us.

```
template <typename TVisitor, typename ... TVariants>
constexpr decltype(auto) visit_recurisively(
    TVisitor&& visitor, TVariants&& ... variants)
{
    return std::visit(
        std::forward<TVisitor>(visitor),
        std::forward<TVariants>(variants)._data ...
    );
}
```

This very simple change makes the code much cleaner.

```

struct evaluator
{
    int operator()(int x) { return x; }
    int operator()(const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        const auto rest = visit_recursively(*this, rhs);

        return match(op)(
            [&](plus) { return lhs + rest; },
            [&](minus){ return lhs - rest; });
    }
};

```

```
// 5
expr e0{5};

// 9 + 3
expr e1{make_unique<r_expr>(9, plus{}, 3)};

// 1 - (3 + 7)
expr e2{make_unique<r_expr>(1, minus{},
                           make_unique<r_expr>(3, plus{}, 7))};
```

```
cout << visit_recursively(evaluator{}, e0);
cout << visit_recursively(evaluator{}, e1);
cout << visit_recursively(evaluator{}, e2);
```

[\[on gcc.godbolt.org\]](https://gcc.gnu.org/godbolt.org) | [\[on melpon.org/wandbox\]](https://melpon.org/wandbox)

Our small improvement does not however address the primary shortcomings of this technique.

Let's try to apply our "**lambda-based**" visitation here.

Remember our recursive variant type?

```
using r_expr = std::tuple<number, op, expr>;  
struct expr  
{  
    std::variant<number, std::unique_ptr<r_expr>> _data;  
};
```

Intuitively, the following "lambda-based" visitor should work:

```
auto vis = overload(  
    [](int x) { return x; },  
    [](const std::unique_ptr<r_expr>& x)  
    {  
        const auto& [lhs, op, rhs] = *x;  
        const auto rest = visit_recursively(vis, rhs);  
  
        return match(op)(  
            [&](plus) { return lhs + rest; },  
            [&](minus){ return lhs - rest; });  
    });
```


That unfortunately results in a *compile-time* error.

```
error: use of 'vis' before deduction of 'auto'  
  const auto rest = visit_recursively(vis, rhs);  
                                     ^~
```

[on gcc.godbolt.org]

The issue is that the language does not provide a way of referring to the current *closure* in a *lambda expression*.

A lambda-expression is just syntactic sugar for a function object:

```
[ ]{ return 0; }
```

...is roughly equivalent to...

```
struct ??  
{  
    int operator() const { return 0; }  
};
```

Inside the definition of *member functions*, we can refer to the current instance of the parent object with the `this` keyword.

```
struct foo
{
    int operator()(int x)
    {
        if(x == 0) return 0;
        return (*this)(x - 1);
    }
};
```

Inside a *lambda-expression*, `this` does not refer to the current instance of the generated *closure* - it instead refers to the outer context!

```
auto foo = [](int x) → int
{
    if(x == 0) return 0;
    return (*this)(x - 1);
};
```

error: `this` was not captured for this lambda function

Here's an example on how the language could allow this in the future, taken from proposal [P0200](#):

```
auto foo = [] self (int x) → int
{
    if(x == 0) return 0;
    return self(x - 1);
};
```

Let's try to find a solution that doesn't require *run-time overhead*.

```
auto fact = [](int n)
{
    if(n == 0)
    {
        return 1;
    }

    return n * ??(n - 1);
};
```

...what if we bind `fact` to itself and pass it as an extra argument?

```
auto fact = [](auto self, int n) → int
{
    if(n == 0)
    {
        return 1;
    }

    return n * self(self, n - 1);
};
```

```
static_assert(fact(fact, 3) == 3 * 2 * 1);
```

[on gcc.godbolt.org]

It works! (Note the *mandatory* trailing return type.)

There is some syntactical overhead though...

```
auto fact = [](auto self, int n) → int
{
    if(n == 0)
    {
        return 1;
    }

    return n * self(self, n - 1);
//          ^^^^^^^^^^^
};

static_assert(fact(fact, 3) == 3 * 2 * 1);
//          ^^^^^^^^^^^
```

...can we do anything about it?

Yes! We can use the **Y combinator**.

The **Y combinator** is commonly found in *functional programming* languages. It's an *higher-order function* that can be used to **implement recursion**.

This will be our final result:

```
auto fact = y_combinator([](auto self, int n) → int
{
    if(n == 0)
    {
        return 1;
    }

    return n * self(n - 1);
});
```

```
static_assert(fact(3) == 3 * 2 * 1);
```

You can think of `y_combinator` as a *function template* which:

- Accepts any *function object* f of arity N .
- Returns a wrapper f' around f of arity $N - 1$.

The arity gets decremented because one of the arguments is bound to f' itself.

Here's a possible implementation:

```
template <typename TF>
struct y_combinator_wrapper
{
    TF _f;

    template <typename TFFwd>
    y_combinator_wrapper(TFFwd&& f)
        : _f{std::forward<TFFwd>(f)}
    {
    }

    template <typename ... Ts>
    decltype(auto) operator()(Ts&& ... xs)
    {
        return _f(std::ref(*this),
                  std::forward<Ts>(xs) ... );
    }
};
```

That's it! An helper `y_combinator` function will return `y_combinator_wrapper` :

```
template <typename TF>
auto y_combinator(TF&& f)
{
    return y_combinator_wrapper<std::decay_t<TF>>(
        std::forward<TF>(f));
}
```

```
auto fact = y_combinator([](auto self, int n) → int
{
    if(n == 0)
    {
        return 1;
    }

    return n * self(n - 1);
});
```

```
static_assert(fact(3) == 3 * 2 * 1);
```

[\[on gcc.godbolt.org\]](https://gcc.gnu.org/)

The code above now compiles and works. We now know have all the tools to build recursive "lambda-based" visitation!

Recursive "lambda-based" visitation

overview

- Local "pattern-matching"-like visitation.
- Minimal syntactical boilerplate, no run-time overhead.
- Nested visitation abstracted behind a `recurse` call.

Recursive "lambda-based" visitation

example - in-place visitation

```
expr e{make_unique<r_expr>(1, minus{},
    make_unique<r_expr>(3, plus{}, 7))};
```

```
match_recursively<int>(e)(
    [](auto, number x) { return x; },
    [](auto recurse, const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        return match(op)(
            [&](plus) { return lhs + recurse(rhs); },
            [&](minus){ return lhs - recurse(rhs); });
    });
```

Recursive "lambda-based" visitation

example - visitor creation

```
auto evaluate = make_recursive_visitor<int>(
    [](auto, number x) { return x; },
    [](auto recurse, const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        return match(op)(
            [&](plus) { return lhs + recurse(rhs); },
            [&](minus){ return lhs - recurse(rhs); });
    }
);
```

```
cout << visit_recursively(evaluate, e0); // "5"
cout << visit_recursively(evaluate, e1); // "12"
cout << visit_recursively(evaluate, e2); // "-9"
```

Let's implement `make_recursive_visitor`.

- It will accept a *return type* `T` as a template parameter. Deducing the *return type* is **non-trivial**.
- It will take any number of *function objects* `fs ...` as input. These will be the "visitor branches".

```
template <typename TReturn, typename ... TFs>
auto make_recursive_visitor(TFs&& ... fs)
{
    // ... ?
}
```

The signatures of the `fs ...` visitors have arity $N + 1$, where:

- N is the number of variants.
- The extra $+1$ is the `recurse` *function object*.

Example:

```
[ ](auto recurse, const std::unique_ptr<r_expr>& x)
{
    // ...
}
```

The visitor is going to be used via `visit_recursively`. Example:

```
visit_recursively(evaluator, e0)
```

`visit_recursively` accesses the `._data` member of the variants.

```
template <typename TVisitor, typename ... TVariants>
decltype(auto) visit_recursively(
    TVisitor&& visitor, TVariants&& ... variants)
{
    return std::visit(std::forward<TVisitor>(visitor),
        std::forward<TVariants>(variants)._data ... );
}
```

Therefore, the *function object* passed to `visit_recursively` must have arity N , which is the number of variants.

We can use `y_combinator` to reduce the arity from $N + 1$ to N and to implement recursion!


```
template <typename TReturn, typename ... TFs>
auto make_recursive_visitor(TFs&& ... fs)
{
    return y_combinator([](auto self, auto&& ... xs)
        → TReturn
    {
        // ... ?
    });
}
```

To create a visitor out of `fs ...`, we can safely reuse our `overload` function.

Overload resolution will work as expected even with the extra `auto` parameter used for recursion.

Example:

```
auto v = make_recursive_visitor<bool>(
    [](auto, int) { return false; },
    [](auto, char) { return true; });

v([], 'a'); // Will return `true`.
```

```

template <typename TReturn, typename ... TFs>
auto make_recursive_visitor(TFs&& ... fs)
{
    return y_combinator(
        [o = overload(std::forward<TFs>(fs) ... )]
        (auto self, auto&& ... xs) → TReturn
        {
            // ... ?
        }));
}

```

Our goal is to call `o` with the currently active variant *alternatives*.

In the context of the lambda passed to `y_combinator`, the argument pack `xs ...` represents exactly that, as the *recursive visitor* is going to be invoked through `visit_recursively`.

```

template <typename TReturn, typename ... TFs>
auto make_recursive_visitor(TFs&& ... fs)
{
    return y_combinator(
        [o = overload(std::forward<TFs>(fs) ... )]
        (auto self, auto&& ... xs) → TReturn
        {
            return o(/* ?? */,
                std::forward<decltype(xs)>(xs) ... );
        });
}

```

The last missing piece is the `recurse` argument that will allow recursion directly through the visitor's lambdas. It is called with *variants*. Example:

```
[](auto recurse, const std::unique_ptr<r_expr>& x)
{
    const auto& [lhs, op, rhs] = *x;
    return match(op)(
        [&](plus) { return lhs + recurse(rhs); },
        [&](minus){ return lhs - recurse(rhs); });
}
```

`rhs` is a *variant* of type `expr`.

...what can we use to invoke *recursive visitors* with *variants*?

Yes, `visit_recursively` !

```

template <typename TReturn, typename ... TFs>
auto make_recursive_visitor(TFs&& ... fs)
{
    return y_combinator(
        [o = overload(std::forward<TFs>(fs) ... )]
        (auto self, auto&& ... xs) → TReturn
    {
        return o([&self](auto&& ... vs)
        {
            return visit_recursively(
                self,
                std::forward<decltype(vs)>(vs) ... );
        },
        std::forward<decltype(xs)>(xs) ... );
    });
}

```



```
[&self](auto&& ... vs)
{
    return visit_recursively(
        self,
        std :: forward<decltype(vs)>(vs) ... );
}
```

- We're binding `self` as the first argument for the next recursive step.
- The variants `vs ...` obtained from a `recurse(vs ...)` call are then being forwarded to `visit_recursively`.

That's it! Let's try `make_recursive_visitor` out.

```
auto evaluate = make_recursive_visitor<int>(
    [](auto, number x) { return x; },
    [](auto recurse, const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        return match(op)(
            [&](plus) { return lhs + recurse(rhs); },
            [&](minus){ return lhs - recurse(rhs); });
    }
);
```

```
cout << visit_recursively(evaluate, e0); // "5"
cout << visit_recursively(evaluate, e1); // "12"
cout << visit_recursively(evaluate, e2); // "-9"
```

[\[on gcc.godbolt.org\]](https://gcc.godbolt.org) | [\[on melpon.org/wandbox\]](https://melpon.org/wandbox)

The last missing facility is `match_recursively`, which performs in-place recursive visitation. Example:

```
match_recursively<int>(e)(
    [](auto, number x) { return x; },
    [](auto recurse, const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        return match(op)(
            [&](plus) { return lhs + recurse(rhs); },
            [&](minus){ return lhs - recurse(rhs); });
    });
```

It's trivial to implement it by slightly changing `match`'s implementation.

```
template <typename ... TVariants>
constexpr auto match(TVariants&& ... vs)
{
    return [&vs ... ](auto&& ... fs) → decltype(auto)
    {
        auto visitor = overload(std::forward<decltype(fs)>(fs) ... );
        return std::visit(visitor, std::forward<TVariants>(vs) ... );
    };
}
```

- An additional return type *template parameter* will be added.
- `overload` will be replaced by `make_recursive_visitor`.
- `std::visit` will be replaced by `visit_recursively`.

```

template <typename TReturn, typename ... TVariants>
constexpr auto match_recursively(TVariants&& ... vs)
{
    return [&vs ... ](auto&& ... fs) → decltype(auto)
    {
        auto visitor = make_recursive_visitor<TReturn>(
            std::forward<decltype(fs)>(fs) ... );

        return visit_recursively(visitor,
            std::forward<TVariants>(vs) ... );
    };
}

```

Example:

```
expr e{make_unique<r_expr>(1, minus{},
    make_unique<r_expr>(3, plus{}, 7))};

std::cout << match_recursively<int>(e)(
    [](auto, number x) { return x; },
    [](auto recurse, const std::unique_ptr<r_expr>& x)
    {
        const auto& [lhs, op, rhs] = *x;
        return match(op)(
            [&](plus) { return lhs + recurse(rhs); },
            [&](minus){ return lhs - recurse(rhs); });
    });
```

[\[on gcc.godbolt.org\]](https://gcc.godbolt.org) | [\[on melpon.org/wandbox\]](https://melpon.org/wandbox)

part 3 - recap

- Variants can be used to elegantly model **recursive structures** (*e.g. JSON, ASTs, lists, ...*).
- A **layer of indirection** must be used to define recursive variants and keep their size fixed.
- "Traditional" recursive variant visitation using a `struct` is trivial, but suffers from the usual issues.

part 3 - recap

- Lambdas cannot recursively call themselves, as it's **impossible to refer to the *current* closure from the closure itself**.
- Recursion can be implemented by **passing the lambda as one of its own arguments**. This doesn't introduce any unnecessary run-time or memory overhead.
- The **Y combinator** higher-order function generalizes that solution and minimizes boilerplate.

part 3 - recap

- When visiting recursive variants with *simple* or *one-off* logic, prefer the "lambda-based" approach.
- When the logic is *complex* and *reusable*, prefer the "traditional" approach.
 - Note that all the utilities covered in this talk can be *harmoniously* used together.
 - ...and can be adapted to `optional` as well!

Thanks!

- vromeo5@bloomberg.net
- <https://vittorioromeo.info>
- vittorio.romeo@outlook.com
- [@supahvee1234](#)