Appendix: implementing variant pattern matching

Section 7

In this section

- The problem with `std::visit`
- `match` syntax
- Implementing `match` from scratch
- Future improvements and considerations

The problem with `std::visit`

Part 2.1

In this part

- Pattern matching
- The reasons why `std::visit` is not "good enough"

Pattern matching

- Common feature of *functional programming* languages
- Form of dispatch that looks at the "shape" of the given value
- We can *pattern match* on `std::variant`'s alternatives

```
std::variant<int, float, char> v{/* ... */};

match([](int x)   { foo(x); },
      [](float x) { bar(x); },
      [](char x)  { baz(x); })(v);
```

Problems with `std::visit`

- `std::visit` is inherently **verbose** and cumbersome
- It discourages "*pattern matching*" on variants, even though it's a powerful pattern
- Visitation can be done through:
  - A `struct` with multiple `operator()` overloads
  - A generic lambda with an `if constexpr` chain
- Both harm readability and increase boilerplate

Problems with `std::visit` - comparison

```cpp
std::variant<int, float, char> v{/* ... */};

struct visitor
{
    void operator()(int x)   { foo(x); }
    void operator()(float x) { bar(x); }
    void operator()(char x)  { baz(x); }
};

std::visit(visitor{}, v);
```

Problems with `std::visit` - comparison

```
std::variant<int, float, char> v{/* ... */};
std::visit([](auto x)
{
    if constexpr(std::is_same_v<decltype(x), int>) {
        foo(x);
    }
    else if constexpr(std::is_same_v<decltype(x), float>) {
        bar(x);
    }
    else if constexpr(std::is_same_v<decltype(x), char>) {
        baz(x);
    }
}, v);
```

Problems with `std::visit` - comparison

```
std::variant<int, float, char> v{/* ... */};

match([](int x)   { foo(x); },
      [](float x) { bar(x); },
      [](char x)  { baz(x); })(v);
```

- Minimal boilerplate
- Short and readable
- Resembles "*pattern matching*"

`match` syntax

```
match(/* branches... */)(/* variants... */);
```

- `branches...` must be an exhaustive set of *function objects* that can be invoked with all the combination of `variants...`'s alternatives
- Two invocations: the first one returns an object that, when invoked with `variants...`, performs visitation

`match` syntax

```cpp
auto vis0 = match([](int x)   { foo(x); },
                  [](float x) { bar(x); },
                  [](char x)  { baz(x); });

vis0(variant0);
vis0(variant1);
```

- The double invocation allows reuse of the generated visitor

`match` syntax

```
std::variant<int, char> v0{/* ... */};
std::variant<int, char> v1{/* ... */};

match([](int, int)   {  },
      [](int, char)  {  },
      [](char, int)  {  },
      [](char, char) {  })(v0, v1);
```

- Example with two variants

`match` syntax

```
std::variant<int, float, char> v{/* ... */};

auto str = match([](int)   { return "integer"; },
                 [](float) { return "float";   },
                 [](char)  { return "char";    })(v);
```

- `match` can return values
- All branches must return the same type

Creating an overload set

Part 2.2

In this part

- Implemeting generic `overload( ... )` function

`match` - implementation overview

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

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

- 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*.

Building an overload set

- Given any number of generic *function objects*, how can we build an overload set out of them?

Building an overload set

Intuition: `struct` with multiple `operator()` overloads:

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

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

Building an overload set

- `foo` can be composed through *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();
};
```

Building an overload set

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

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

- Behaves exactly like before

Building an overload set

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

- Without the `using`*-declarations* the previous example code would result a compiler error.
- The reason is that the call to `foo::operator()` would be **ambiguous** because *name resolution* is performed before *overload resolution*.

Building an overload set

- We can generalize the pattern by templating over the base classes

```cpp
template <typename A, typename B>
struct overload_set : A, B
{
    using A::operator();
    using B::operator();
};
```

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

Building an overload set

- To support any number of functions, we can use a *variadic template*

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

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

(on *wandbox.org*)

  - Variadic `using` directives were introduced in C++17

Building an overload set

- Using `overload_set` with lambdas is cumbersome, as their type cannot be easily deduced, and they are not *default-constructible*

```
auto l0 = [](float){ return 0; };
auto l1 = [](char) { return 1; };
using foo = overload_set<decltype(l0), decltype(l1)>;
auto x0 = foo{l0, l1}(0.f);
```

(on *wandbox.org*)

Building an overload set

- We can solve both problems by introducing a *perfectly-forwarding constructor* and a *deduction guide* to our `overload_set` class

```cpp
template <typename... Fs>
struct overload_set : Fs...
{
    template <typename... Xs>
    constexpr overload_set(Xs&&... xs)
        : Fs{std::forward<Xs>(xs)}... { }

    using Fs::operator()...;
};
```

Building an overload set

```
template <typename... Xs>
overload_set(Xs&&... xs)
    -> overload_set<std::decay_t<Xs>...>;
```

- Deduction guides were introduce in C++17 and allow users to customize the behavior of *class template argument deduction*
- In this case, we are telling the compiler to deduce the type of `overload_set` by decaying the type of every *function object* passed to its constructor
- `decay_t` removes *cv-qualifiers* and *references*

Building an overload set

Example:

```
auto l = [](int){ };
overload_set o0{l};
overload_set o1{std::move(l)};
```

- Both deduced as `overload_set<std::decay_t<decltype(l)>>`
- `o0` copies the lambda into the wrapper
- `o1` moves the lambda into the wrapper

Building an overload set

- Before C++17, a `make_overload_set` function could have been provided:

```
template <typename ... Fs>
auto make_overload_set(Fs&& ... fs)
{
    return overload_set<std::decay_t<Fs> ... >(
        std::forward<Fs>(fs) ...
    );
}
```

  - This has the same purpose as the *deduction guide*

Building an overload set

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

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

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

(on *wandbox.org*)

- Lambdas in C++17 are *implicitly* `constexpr` if possible - `static_assert` therefore works with them.

`match` - implementation

Part 2.3

In this part

- Implementing `match( ... )( ... )`

`match` - implementation overview

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

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

1. Build an `overload_set` out of the f_x...
2. Invoke `std::visit` on the new overload set

`match` - implementation

> `match` will be a function that takes N_f *function objects* and returns a function that takes N_v *variants*.
>
> ```
> match(f0, f1, ... , fN_f)(v0, v1, ... , vN_v);
> ```

↓

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

`match` - implementation

> 1. Build an `overload_set` out of the f_x...

```
template <typename ... Fs>
auto match(Fs&& ... fs)
{
    return [
        visitor = overload_set{std::forward<Fs>(fs) ... }
    ](auto&& ... vs)
    {
        /* ... */
    };
}
```

`match` - implementation

> 2. Invoke `std::visit` on the new overload set

```
template <typename... Fs>
auto match(Fs&&... fs)
{
    return [
        visitor = overload_set{std::forward<Fs>(fs)...}
    ](auto&&... vs) -> decltype(auto)
    {
        return std::visit(visitor,
            std::forward<decltype(vs)>(vs)...);
    };
}
```

(on *wandbox.org*)

`match` - examples

- Complete usage example: https://wandbox.org/permlink/u9B1KQEOiUQ5WD3n
- Generated assembly: https://godbolt.org/g/BtY7dC

`match` - recap

Part 2.4

In this part

- Section recap

`std::visit` vs `match`

- `std::visit` is overly verbose and requires the definition of either:
  - A `struct` with multiple `operator()` overloads
  - A *generic lambda* with an `if constexpr` chain
- `match` has minimal boilerplate and resembles *pattern matching*
  - Its "double invocation" syntax *(currying)* allows easy reuse of the generated visitor
  - Much more readable than a traditional `std::visit` call

`overload_set`

- Public inheritance allows us to create *overload sets* from arbitrary *function objects*
    - `using` directives are required to expose all base classes' `operator()` overloads in the same scope
- C++17 *class template argument deduction* and *deduction guides* allow us to easily create `overload_set` instances from *lambda expressions*
- *Perfect forwarding* and `std::decay` are used to store the lambdas

`match`

- When invoked with `fs...`, produces a visitor by overloading `fs...` together and returning a *variadic generic lambda*
- The returned lambda accepts any amount of *variants* and internally calls `std::visit` with the newly-created visitor