`std::visit` - Benefits over `get`/`get_if`

- **Exhaustive**: compilation will fail if any of the alternatives cannot be handled by the visitor
- **Future-proof**: compilation will fail if new alternatives are added to the variant
- **Flexible**: supports multiple dispatch, can be used with stateful visitors

`std::visit` - With generic lambda

- A *generic lambda expression* produces a **closure** with a `template operator()` - this is a suitable visitor
- Using `if constexpr` inside the body of the lambda allows us to dispatch depending on the type of the active alternative

`std::visit` - `struct` 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.

`std::visit` - *Generic lambda* shortcomings

- **Boilerplate code**: verbose boilerplate is required to dispatch depending on the type of the argument
- **Imperative control flow**: variants lend themselves well with declarative control flow *(e.g. pattern matching)* - exhaustiveness is lost

`std::visit` - A better solution?

- It is possible to implement a wrapper over `std::visit` which:
  - Has terser syntax
  - Is easier to use
  - Roughly resembles pattern matching
- The implementation is available as an *appendix*
- If there's enough time at the end after the course, we'll go through it

`std::variant` - Use cases

- Representing choices between types
- Type-safe error handling
- State machines
- Recursive variants

Choices between types

- Whenever you need **any** type that **matches an interface**, using traditional *polymorphism* or *type erasure* is often a good idea
- Whenever you have a **closed set of types** with potentially different interfaces, `std::variant` is almost always the best choice

Choices between types - polymorphism example

```
struct key_value_store
{
    virtual void put(K, V) = 0;
    virtual V get(K) = 0;
};
```

```
struct redis : key_value_store { /* ... */ };
struct mock_database : key_value_store { /* ... */ };
struct on_hdd : key_value_store { /* ... */ };
```

```
void consume_data(key_value_store&);
```

Choices between types - Polymorphism example

- Requires a base class with `virtual` member functions
- Usually requires *dynamic allocation* and *indirection*
- All types must conform to the same interface
- Additional types can be created and used even after compilation

Choices between types - Closed set example

```
using chat_packet = std::variant<
    connection,
    disconnection,
    text_message,
    image_message,
    file_attachment
>;
```

```
void send(chat_packet);
chat_packet receive();
```

Choices between types - Closed set example

```
struct connection       { int _user_id; };
struct disconnection    { int _user_id; reason _reason; };
struct text_message     { std::string _content; };
struct image_message    { blob _content; format _format; };
```

- No inheritance required
- Types can have different *data members* and *interfaces*
- No dynamic allocation required
- All possible alternatives must be known at compile-time
- Compiler can usually optimize more aggressively

Type-safe error handling

- Often functions can **fail**, and need to return some sort of error code to the user. Common techniques include:
    1. Returning an error code and taking an output parameter
    2. Returning a pair containing a possible error code
    3. Throwing an exception
- All of these have shortcomings

Type-safe error handling - Error code + output parameter

```
int get_hostname(std::string& s)
{
    if(/* connected successfully */)
    {
        s = /* host name */;
        return 0;
    }

    return /* some non-zero error code */;
};
```

Type-safe error handling - Error code + output parameter

- `get_hostname` does not take advantage of C++'s type system and is error prone.

```
std::string out;
get_hostname(out);

// whoops, forgot to check the return code!
consume(out);
```

- The problem is that we can use `out` even though `get_hostname` failed

Type-safe error handling - Output + error pair

```
std::pair<std::string, int> out = get_hostname();

// whoops, forgot to check the return code!
consume(out.first);
```

- It is more obvious that there is an additional `int` here, but it is still possible to make a mistake
- Unnecessary memory is also being used, as the `int` will always take space even if useless *(i.e. on success)*
- Still not taking advantage of the type system

Type-safe error handling - Throwing an exception

```
std::string out = get_hostname();
consume(out);
```

- If `get_hostname` throws, we do not incorrectly call `consume`
- It is unclear how the function can fail - the signature does not provide any information anymore
- Failure is *implicit* - desirable for “exceptional” errors, but undesirable for logic/business errors
- Not taking advantage of the type system

Type-safe error handling - `std::variant`

```
struct success    { std::string _hostname; };
struct io_failure { int _system_code; };
struct timed_out  { };

using get_hostname_result = std::variant<
    success, io_failure, timed_out
>;
```

```
get_hostname_result get_hostname();
```

- All success/failure cases are exposed and part of the type system
- Misuse is almost impossible

Type-safe error handling - `std::variant`

```
struct visitor {
    void operator()(success x)     { consume(x._hostname);    }
    void operator()(io_failure x) { report(x._system_code); }
    void operator()(timed_out)     { report("timed out");     }
};

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

- All possible return states must be handled **explicitly** by the caller
- The type system prevents misuse - cannot invoke `consume` without first matching the `success` case

State machines

- Different states have different *data members* and *member functions*
- `std::variant` can guarantee that only the currently active state is accessible, avoiding mistakes

State machines

```
struct patrolling { direction _dir; timer _timer; }
struct chasing    { };
struct fighting   { int _cooldown; };
```

```
struct enemy
{
    target _target;
    std::variant<patrolling, chasing, fighting> _state;
};
```

State machines

```
struct visitor {
    target _target;
    void operator()(patrolling& x){ move(x._dir, x._timer); }
    void operator()(chasing& x)   { move_towards(_target);  }
    void operator()(fighting& x)  { attack(x._cooldown);    }
};

void process(enemy& e) {
    std::visit(visitor{e._target}, e);
}
```

- Data members of a state are only accessible if that state is active
- State transitions can be achieved by simply assigning a new state to the variant

Recap - What is a *variant*?

- A variant represents a “choice between types”
- Can be used to model “closed set polymorphism”
- Type-safe tagged `union`
- Sum type
- No dynamic allocation, value semantics

Recap - `std::variant`

- Variadic template class
- Supports all copy/move operations

```
using my_variant = std::variant<int, float>;

my_variant v0{10};  // contains an `int`
my_variant v1{5.f}; // contains a `float'

v0 = v1; // `v0` now contains `5.f`
```

Recap - `std::variant` - Manual access

- `std::holds_alternative<T>` can be used to check the active alternative
- `std::get<T>` can be used to access the active alternative - throws in case of error
- `std::get_if<T>` returns a valid pointer if `T` is the active alternative, `nullptr` otherwise

Recap - `std::variant` - Manual access

```
std::variant<int, float> v0{5.f};
assert(std::holds_alternative<float>(v0));

std::get<int>(v0); // will throw

if(auto* p = std::get_if<float>(&v0))
{
    // ...
}
else
{
    // ...
}
```

Recap - `std::variant` - Visitation

- Given a **visitor** that can be invoked with all alternatives of a variant, `std::visit` will automatically invoke the correct overload

```
struct visitor {
    void operator()(int)   { } // (0)
    void operator()(float) { } // (1)
};

std::variant<int, float> v{42};
std::visit(visitor{}, v); // invokes (0)

v = 123.4f;
std::visit(visitor{}, v); // invokes (1)
```

Recap - `std::variant` - Use cases

- **Type-safe error handling**
  - Superior alternative to error codes and often exceptions
- **Representing choices between types**
  - *E.g.* instructions in a virtual machine
- **State machines**
  - *E.g.* connection to a server; character in a video game
- **Recursive data structures**
  - *E.g.* JSON, XML, abstract syntax trees, mathematical expressions
  - Covered in an *appendix*
- …

Discussion

- Polymorphism versus variants

Algebraic Data Types:

`std::optional`

Algebraic Data Types:

`std::optional`

In this section

- What is an `optional<T>`?
- `std::optional<T>`’s interface and semantics
- Example use cases

What is an optional?

- What an optional represents
- Possible implementation
- *Optional* vs *pointers*

Meaning of `optional<T>`

- An `optional<T>` can fundamentally be in two states
  - **Set**: contains an instance of `T`
  - **Unset**: does not contain any instance of `T`
- It represents a *"value that may or may not be present"*
- Has *value semantics* and doesn't use dynamic allocations
- `sizeof(optional<T>)` is slightly bigger than `sizeof(T)`

Example: `std::optional` usage

```
std::optional<int> o; // <== initially unset
assert(o.has_value() == false);

o = 15; // <== `o` is now set
assert(o.has_value() == true);
assert(o.value() == 15);

o = std::nullopt; // <== `o` is now unset
assert(o.has_value() == false);
```

Meaning of `optional<T>`

- *Intuition:* `optional<T>` is similar to `variant<T, nothing>`
- `optional<T>` can properly model:
  - Functions that can fail, where no extra error information is required
  - Absence of a result *(e.g. searching in a container)*
  - Non-mandatory *data members* and *arguments*
  - Deferred construction of an object

Meaning of `optional<T>`

```
optional<int> parse_string_to_int(const string&);
```

» If the string cannot be parsed, an *unset optional* is returned

```
optional<int> find_substr(const string&, const string&);
```

» If a substring match is found, its index is returned

» If there is no such match, an *unset optional* is returned

» Similarly to `variant`, the caller is forced to check the status of the returned `optional`

Meaning of `optional<T>`

```
struct person
{
    std::string _name;
    std::optional<employer> _employer;
};
```

» A `person` may or may not have an employer

Meaning of `optional<T>`

```
struct service
{
    std::optional<request_processor> _rp;

    service()
    {
        // ... initialization steps ...
        _rp.emplace(); // <== construct instance in `_rp`
    }
};
```

- Complicated classes can have their construction deferred, while still retaining safety and convenience of **RAII**

Memory layout of `optional<T>`

» Conceptually, `optional<T>` is just:

- Storage for a `T` instance
- `bool` that keeps track of whether or not the optional is set

```
template <typename T>
class optional
{
    std::aligned_storage_t<sizeof(T), alignof(T)> _data;
    bool is_set = false;
    // ...
};
```

Memory layout of `optional<T>`

- This means that **no dynamic allocation** is required, and that `optional<T>` **has value semantics**
- Inexpensive abstraction compared to a *smart pointer*, potentially cache-friendly
- `optional<T>` should be your **first choice** when you want to:
  - Represent possible absence of a value/parameter
  - Model a function that can fail/return nothing
  - Manually control the lifetime of an object

`optional<T>` vs *smart pointers*

- Both `optional<T>` and `std::unique_ptr<T>` can be used to control the lifetime of an object or represent absence of a `T` instance
- Using a *smart pointer* has several drawbacks:
  - **Loss of value semantics**
  - Overhead due to **dynamic allocation**
  - Overhead due to **indirection**

`optional<T>` vs *raw pointers*

- `T*` can be used to represent a *"potentially-null reference"* to an existing `T` instance
- Not all optional implementations support `optional<T&>`
- It is therefore *idiomatic* and *recommended* to use `T*` to:
  - Return/accept a reference that might be null
- `T*` should **never** own the memory it points to
  - Manual memory management is **unsafe** and superseded by *smart pointers*

`optional<T>` vs *raw pointers*

» If your implementation supports `optional<T&>`, use it to represent **non-owning nullable** references to `T`

- It can be "more type-safe" than `T*`, as long as proper abstractions are used
- Otherwise, `T*` is fine - but you must remember to explicitly check for `nullptr`

`std::optional` - Basic interface

std::optional

Defined in header <optional>

```
template< class T >
class optional;
```
(since C++17)

- `std::optional<T>` is a *template class* that may or may not contain an instance of `T`
- The only requirement for `T` is `Destructible`

```
using maybe_int = std::optional<int>;
using maybe_str = std::optional<std::string>;
```

`std::optional` - Default constructor

- The *default constructor* of `std::optional` will create an **unset** optional

```
std::optional<int> o0;
// `o0` does not contain an instance of `int`

std::optional<float> o1;
// `o1` does not contain an instance of `float`
```

`std::optional` - `nullopt`

The Standard Library provides:

```
namespace std
{
    struct nullopt_t;
    inline constexpr nullopt_t nullopt{};
}
```

- `nullopt` can be used during `optional` *construction* or *assignment* to conveniently represent the “unset state”

`std::optional` - `nullopt` constructor

- Identical behavior to the *default constructor*, but takes a `std::nullopt_t` argument
- Useful in generic contexts and/or when we want to be *explicit* to the reader

```cpp
using foo = std::optional<int>;

// ...

foo f{std::nullopt}; // unset `optional<int>`
```

`std::optional` - `U&&` constructor

```
template <typename T>
template <typename U = T>
std::optional<T>::optional(U&& value);
```

- » Initializes a **set** optional by **perfectly-forwarding** `value` in the optional’s data storage
- » `T` must be *constructible* from `U&&`

```
std::optional<int> o0{10};
std::optional<float> o1{42};
```

`std::optional` - Copy/move constructors

» Optionals of the same type can be copy/move-constructed

- The *target optional* will be in the same state as the *source optional*
- If the *source optional* contained a value, it will be copied/moved

```
std::optional<int> s0;
std::optional<int> s1{42};

auto d0 = s0; // <== unset `optional<int>`
auto d1 = s1; // <== set `optional<int>`, value `42`
```

std::optional - in-place constructor

```
template< class... Args >
constexpr explicit optional( std::in_place_t, Args&&... args );
```

```
template< class U, class... Args >
constexpr explicit optional( std::in_place_t,
                             std::initializer_list<U> ilist,
                             Args&&... args );
```

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

```
std::optional<std::string> o5(std::in_place, 100, 'a');
// contains string with 100 'a' characters
```

`std::optional` assignment

» `std::optional<T>` supports:

- Copy/move assignment
- Assigment from any `U` or `optional<U>`, where `U` can be used to construct `T`

```
std::optional<int> o0;
o0 = 42;

std::optional<int> o1;
o1 = o0;
```

`std::optional` - Checking status

» The current status of an `optional` can be checked with:

- `optional<T>::has_value()`
- `optional<T>::operator bool()`

```
std::optional<int> o0{42};
assert(o0.has_value());
assert(o0);

o0 = std::nullopt;
assert(o0.has_value() == false);
assert(!o0);
```

`std::optional` - Accessing contained value

» The value in a set `std::optional` can be accessed with:

- Unchecked access:
  - `std::optional<T>::operator*`
  - `std::optional<T>::operator->`
- Checked access:
  - `std::optional<T>::value`
  - `std::optional<T>::value_or`

`std::optional` - Unchecked access

- **Undefined behavior** if `has_value() == false`
- Pointer-like interface
- Useful when you are sure the optional contains a value

```
std::optional<std::string> o0{"hello"};
assert(*o0 == "hello");
assert(o0->size() == 5);

std::optional<int> o1;
foo(*o1); // Undefined behavior!
```

`std::optional` - Checked access

- `.value()` returns a reference to the stored object, if any
- Otherwise, `std::bad_optional_access` is thrown

```
std::optional<int> o0{42};
assert(o0.value() == 42);

std::optional<int> o1;
foo(o1.value()); // Throws `std::bad_optional_access`
```

std::optional - value_or

- Returns the contained value, if any
- Otherwise, returns the passed default value
- Useful to model choices with a predefined value

```
std::optional<int> o0{42};
assert(o0.value_or(1000) == 42);

o0 = std::nullopt;
assert(o0.value_or(1000) == 1000);
```

`std::optional` - `reset` and `emplace`

- `.reset()` destroys the contained value, if any - otherwise it has no effects
- `.emplace(...)` takes any number of arguments constructs an object in-place by perfectly-forwarding them

```
std::optional<std::vector<char>> o;
o.emplace(20, 'a'); // `o` is a vector containing 20
                    // characters

o.reset();
assert(o.has_value() == false);
```

`std::optional` - Use cases

- When to use `std::optional`
- Simple failure cases
- Modeling optional data
- Controlling construction/destruction

When to use `std::optional`

- `optional<T>` is recommended for situation where there is a **single** and **obvious** reason to model the absence of a `T` value
- If there can be multiple reasons for the absence of a `T` value, choices such as `std::variant` or *exceptions* might be more appropriate

When to use `std::optional`

```
std::optional<double> safe_sqrt(double x);
```

» One failure case: $x < 0$

```
using connection_result =
    std::variant<success, timeout, invalid_address>;

connection_result connect_to(ip_address x);
```

» Multiple failure cases, with possible additional state

Example: parsing a string to `int`

```
std::optional<int> parse_to_int(const std::string& s);
```

- Good use case for `optional`, unless more information about the failure is required
- Signature makes it clear that `nullopt` will be returned if `s` cannot be parsed as a valid `int`
- No need for special values / extra booleans / output parameters

Example: modeling optional data - `person`

```
struct person
{
    std::string _name;
    int _age;
    std::optional<phone_number> _home_number;
    std::optional<phone_number> _work_number;
};
```

- Some data might be inherently “optional”
- Instead of implementing “empty value” semantics for types like `phone_number`, `std::optional` is the appropriate choice

Example: modeling optional data - Reading configuration

```
std::optional<int> get_config_arg(const std::string& key);
```

- Possible scenario: reading from a `.ini` configuration file
- Some key-value pairs are not mandatory (or might have been forgotten)

```
const auto speed = get_config_arg("speed").value_or(5);
```

- Elegant way of falling back to a *default value*

Controlling construction/destruction

- `std::optional<T>` can be useful to provide enough storage for a `T` instance, which has not yet been constructed
- Construction/destruction can be controlled manually with `.emplace(...)` and `.reset()`
- Useful for controlling the lifetime of *active* objects

Example: delaying construction of an active object

- `async_port_listener` creates a new thread that listens to a port on construction, and joins the thread on destruction
- The user must have control over `async_port_listener`

Example: delaying construction of an active object

```
struct state
{
    std::optional<async_port_listener> _listener;
    // ...
};
```

```
state app_state;
// ...

button_start.on_click([&app_state] {
    app_state._listener.emplace(port_entry.value());
});

button_stop.on_click([&app_state] {
    app_state._listener.reset();
});
```

Section recap

- `std::optional<T>` can be in two states: **set** or **unset**
- When *set*, it contains an instance of `T` in its internal storage - no indirection or dynamic allocation is used, has **value semantics**
- Usual implementation: `storage_for<T>` + `bool`
- `std::optional<T>` supports construction/assignment from types convertible to `T` and `std::nullopt`

Section recap

» `std::optional<T>` exposes a pointer-like interface (*`operator*` and `operator->`*) to access the internal value

- The **behavior is undefined** if `.has_value() == false`

» It also exposes `.value()`, which **throws** if the optional is unset

» `o.value_or(default)` will return `o`'s value if set, `default` otherwise

Section recap

- `std::optional<T>` can be used to model:
  - Simple failure cases
  - Optional data or data that might not exist
  - Delayed construction of a `T` instance
- It is superior to alternatives such as *return codes/output parameters* or *dynamic allocation* both in terms of performance and type-safety
- When more information is needed, consider using `std::variant` instead

Exercise

- Implement a message protocol using algebraic data types
  - `exercise2.cpp`
    - on Wandbox
    - on Godbolt

Appendix: Implementing variant pattern matching

Appendix: Implementing variant pattern matching

In this section

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

The problem with `std::visit`

- 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

```
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

```cpp
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

```
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

- Implementing 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

```
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`.
```

- 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

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

```
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

```
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

» `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

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

```
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` - 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

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

Recap - `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

Recap - `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

Appendix: Recursive Variants

Appendix: Recursive Variants

In this section

- Definition of recursive variants
- Visitation of recursive variants

Recursive data structures

- Variants can be defined in a recursive manner in order to model recursive data structures. E.g.
  - JSON
  - Abstract syntax trees
  - Arithmetical expressions
- Some sort of indirection is required, as the size of the variant type must be fixed and known at compile-time

Recursive data structures - *Arithmetical expression* example

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

```
using number = int;

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

Recursive data structures - *Arithmetical expression* example

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

```
struct expr;

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

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

on godbolt.org

Recursive data structures - *Arithmetical expression* example

`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))};
```

Recursive data structures - Visitation

- 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 data structures - Visitation

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

Recursive data structures - Visitation

```
// 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 wandbox.org

Appendix: C++11/14 Refresher

Appendix: C++11/14 Refresher

nullptr

- Keyword that unambiguously represents the null pointer literal
- Its type is `std::nullptr_t`

```
void f(int);      // (0)
void f(float*);   // (1)

f(nullptr);  // invokes (1)
```

```
void g(int);              // (0)
void g(float*);           // (1)
void g(std::nullptr_t);   // (2)

g(nullptr);  // invokes (2)
```

Type aliases and alias templates

- Modern and more flexible version of `typedef`

```
using Precision       = float;
using UserId          = std::uint16_t;
using BinaryPredicate = bool(*)(int, int);

template <typename T>
using StackVector = std::vector<T, StackAllocator<256>>;
```

Type inference

```
      auto  i    = 0;  // `int`
      auto& iref = i;  // `int&`
const auto  j    = i;  // `const int`
const auto& jref = j;  // `const int&`
```

```
decltype(0) k = 0;  // `int`
decltype(k) l = 1;  // `int`

int& f();
decltype(f()) lref = l;  // `int&`
```

```
auto           f();  // deduces return type by value
auto&          g();  // deduces return type by reference
decltype(auto) h();  // deduces return type exactly from `return` expression
```

Trailing return types

```
auto main() -> int { }  // equivalent to `int main() { }`
```

```
auto f(int& i)                  { return i; }  // returns `int`
auto f(int& i) -> decltype(i) { return i; }  // returns `int&`
```

```
template <typename F, typename Arg>
auto call(const F& f, const Arg& arg) -> decltype(f(arg));
```

Range-based `for` loop

- Desugars to traditional `for` loop
- Range expression is “cached” in a forwarding reference variable

```
std::vector<int> v;
for (int x : v) { f(x); }  // OK
```

```
std::vector<int> makeVector();
for (int x : makeVector()) { f(x); }  // OK
```

```
class VectorHolder
{
    std::vector<int> d_vec;

public:
    const std::vector<int>& getVec() const { return d_vec; }
};

for (int x : VectorHolder{}.getVec()) { f(x); }  // UB (!)
```

Other minor features

» `enum class`

- Enumerators are scoped
- Usage be qualified
- No implicit conversions from/to integers

» `enum class MyEnum : int { /* ... */ };`

- Specification of the enum’s underlying type

» Unicode literals: `u8"foo"`, `u"foo"`, `U"foo"`

» Raw string literals: `R"(hello\nworld\n\n)"`

» Binary literals: `0b10101010`

» Digit Separators: `1'000'000'000`

Uniform initialization

- Can use curly braces to initialize everything
- Prevents narrowing implicit conversions
- Does not suffer from “most vexing parse”
- Invokes `std::initializer_list` constructors

```
POD pod{1, 2, 3};  // performs aggregate initialization
UDT udt{1, 2, 3};  // invokes `UDT(int, int, int)` constructor

int i{10.5f};  // compile-time error

int j();  // function declaration
int k{};  // value-initialized integer (zero)

std::vector<int> v{1, 2, 3};  // invokes initializer list constructor
std::vector<int> v{1, 2};     // invokes initializer list constructor
std::vector<int> v(1, 2);     // invokes value-duplicating constructor
```

Move semantics

```
void f(const int& );  // takes anything as a `const` lvalue reference
void f(      int&&);  // only takes integer rvalues (rvalue reference)
```

```
std::vector<int> src{1, 2, 3};

std::vector<int> dst0 = src;  // invokes `vector(const vector&)`, copies
                              // `src` is unchanged

std::vector<int> dst1 = std::move(src);  // invokes `vector(vector&&)`, moves
                                         // `src` is in an unspecified state
```

```
std::vector<int> src{1, 2, 3};

std::move(src);                           // no-op
static_cast<std::vector<int>&&>(src);    // exactly the same as above

void g(std::vector<int> v);  // takes by value, can either copy or move into `v`
g(src);              // caller copies `src` into `v`
g(std::move(src)); // caller moves `src` into `v`
```

```
void h(std::unique_ptr<int>&& u)  // only takes rvalues
{
    sink(u);             // compile-time error, `u` is an lvalue (!)
    sink(std::move(u));  // OK, must move again
}
```

Perfect forwarding

```
void f(int&&);   // rvalue reference
template <typename T> void g(T&&);   // forwarding reference

template <typename T> struct S {
    void h(T&&);   // rvalue reference
};
```

» Forwarding references accept anything

- When bound to an lvalue, `T` is deduced as an lvalue reference

```
template <typename T>
void pipe(T&& x)
{
    if (std::is_lvalue_reference<T>) { sink(x); }
    else                              { sink(std::move(x)); }
}
```

» `std::forward` is equivalent to the above branch

```
template <typename T>
void pipe(T&& x) { sink(std::forward<T>(x)); }
```

Pass by value and move versus perfect forwarding

```
void setName(std::string s) { name = std::move(s); }

setName(std::string{});  // move into `s`, move-assign into `name`

std::string lvalue;
setName(lvalue);  // copy into `s`, move-assign into `name`
```

```
template <typename T>
void setName(T&& s) { name = std::forward<T>(s); }

setName(std::string{});  // bind to `s`, move-assign into `name`

std::string lvalue;
setName(lvalue);  // bind to `s`, copy-assign into `name`
```

- Perfect forwarding version
  - Must be a template
  - Is not constrained (does not only accept `std::string`)
  - Is optimal in terms of performance

Variadic templates

```
template <typename... Ts> struct TypeList { };

TypeList<>                tl0;
TypeList<int>             tl1;
TypeList<int, float, int> tl2;
```

```
template <typename... Ts> void pipe(const Ts&... args) { sink(args...); }
    //                                                      ^ pack expansion
```

```
template <typename... Ts>
std::string concat(const Ts&... args);

template <typename F, typename... Ts>
decltype(auto) callMemoize(F&& f, Ts&&... args);
```

Lambda expressions

```
int i = 0;

auto l = [i, j = 10](int x, auto y) mutable
{
    f(x, y, i, j);
};
```

```
struct Anonymous
{
    int i;
    int j = 10;

    template <typename T>
    auto operator()(int x, T y) /* mutable */
    {
        f(x, y, i, j);
    }
};
```

» Some use cases

- Invoking algorithms and higher-order functions
- Binding arguments
- Asynchronous interfaces, tasks

Other stuff

» Language

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- Reference Qualifiers
- `constexpr` variables
- `constexpr` functions
- SFINAE
- `alignas`, `alignof`
- Rule of Zero/Five

» Library

- `std::array<T, N>`
- `std::tuple<Ts...>`
- `std::function<R(Args...)>`
- `<thread>`, `<mutex>`, `<future>`, `<atomic>`
- `<chrono>`
- `<random>`

Reading material

- Lifetime Extension
  - https://en.cppreference.com/w/cpp/language/reference_initialization#Lifetime_of_a_tempor
  - https://abseil.io/tips/107