Algebraic Data Types: Variant

Section 3

In this section

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

Understanding variants

Part 3.1

In this part

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

What is a `struct`?

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

```cpp
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` $= 200$.

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

Part 3.2

In this part

- `std::variant`'s basic interface

std::variant

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> v0{'a'};
// `v2` contains an `char` with value `'a'`
```

std::variant - T constructor

- Be careful with *implicit conversions*

```cpp
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

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

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

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

```cpp
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

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

```cpp
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 assigment 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>() == 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_perimissions();
}

// ...
```

`std::variant` - visitation

Part 3.3

In this part

- 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);
```

(on godbolt.org)

`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

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

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

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

(on *godbolt.org*)

`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);
```

`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)*.
- **Readabily 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

Part 3.4

In this part

- 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

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

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

```cpp
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:
    i. Returning an error code and taking an output parameter
    ii. Returning a pair containing a possible error code
    iii. 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.

```cpp
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

```cpp
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

```cpp
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

`std::variant` - recap

Part 1.5

In this part

- What we learned about variants

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

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`
```

`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

std::variant - manual access

```cpp
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
{
    // ...
}
```

`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)
```

`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

Q&A

Break

5 minutes