Back to Basics: Algebraic Data Types

I also do C++ training!

`arthur.j.odwyer@gmail.com`

Arthur O’Dwyer
2020-09-16

Outline

- Why the name “algebraic data types”? [3–18]
  - Memory layout diagrams. Why not `std::any`?
- Quick motivation for each type [19–27]
- Shared terminology [28–41] *Questions?*
- More about `optional` [42–49] *Questions?*
- More about `variant` [50–56] *Questions?*
- More about `pair` and `tuple` [57–69] *Questions?*

What do I mean by algebraic types?

`pair`

C++98. The original algebraic data type.

`tuple`

C++11.

`optional`

C++17.

`variant`

C++17, with minor tweaks to its constructors in C++20.

Why do we say “algebraic”?

It’s about the type’s number of possible values, a.k.a. the size of its *domain*. How many possible states might the object take on?

`char`	**256** possible values
`bool`	**2** possible values (`true` and `false`)
`pair<char,bool>`	256 × 2 = **512** possible values
`tuple<char,char,bool>`	256 × 256 × 2 = **131072** possible values

Pair and tuple are product types

To find the size of the domain of a pair or tuple type, we take the *product* of the sizes of its element types.

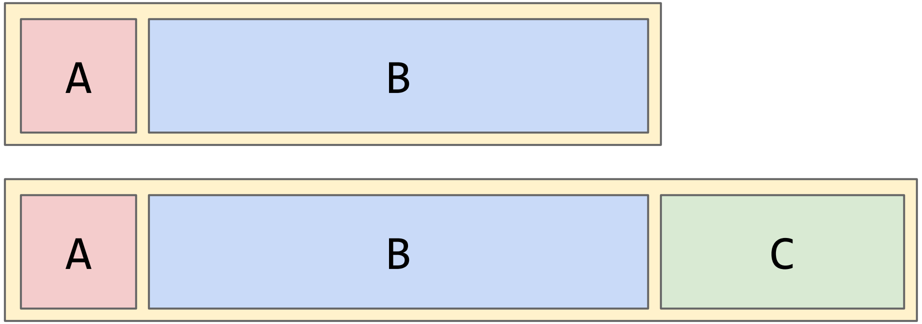
`A`	**A** possible values
`pair<A,B>`	**A×B** possible values
`tuple<A,B,C,...>`	**A×B×C×...** possible values

Therefore `pair` and `tuple` are known as ***product types***.

Memory layout of a product type

The memory layout of pair or tuple is going to be pretty much the same as the layout of a plain old data struct.

```
pair<A,B>

tuple<A,B,C>
```

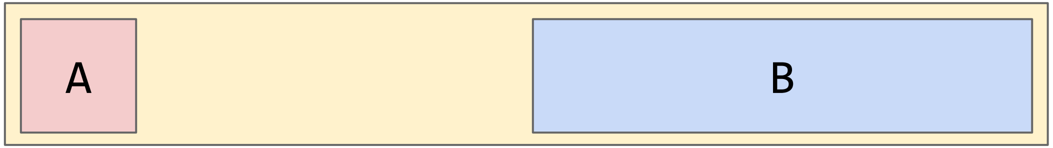


The compiler will do some **padding for alignment**, and may swap the order of the fields...

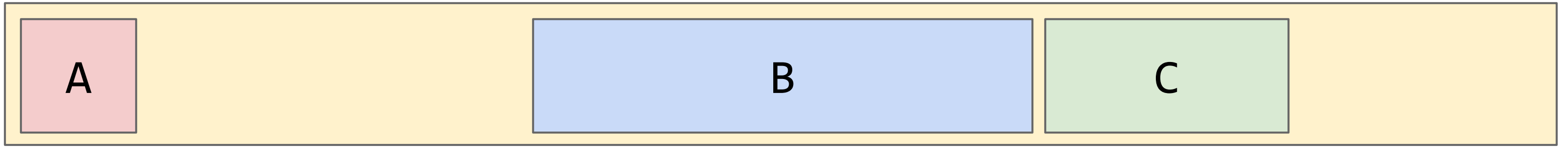
Memory layout of a product type

Here’s the same two types as they would actually be laid out in memory by libc++...

`pair<A,B>`



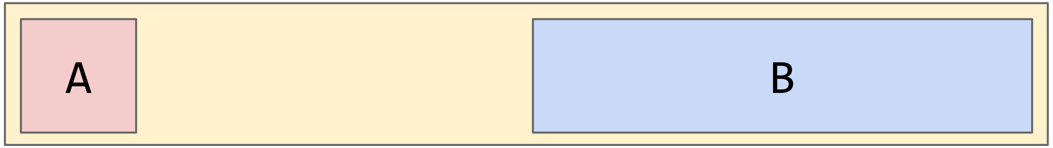
`tuple<A,B,C>`



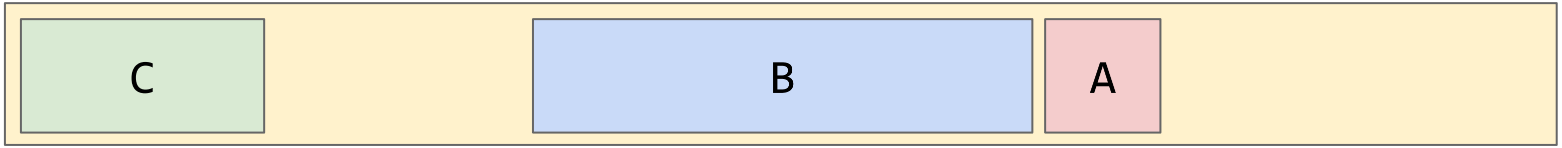
Memory layout of a product type

Here’s the same two types as they would actually be laid out in memory by both libstdc++ and MSVC...

```
pair<A,B>
```



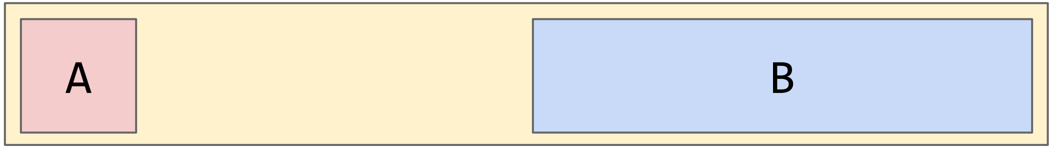

```
tuple<A,B,C>
```



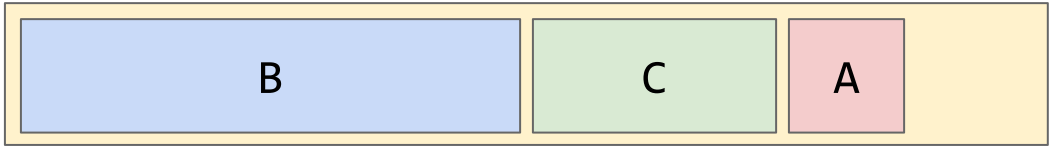
Memory layout of a product type

Here's the same two types as they would be laid out in memory by a hypothetical library ABI. As far as I know, no vendor does this.

`pair<A,B>`

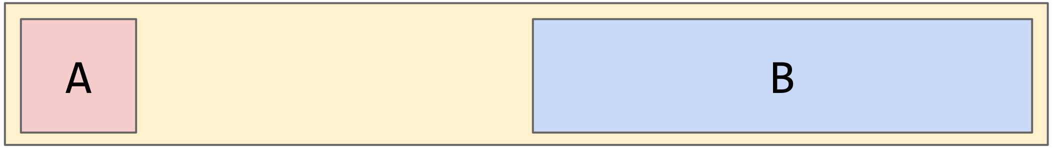


`tuple<A,B,C>`



`pair` appears as "basically" a struct

`pair<A,B>`



Notice how `pair`'s layout never changed.

The library specifies that `pair` is *basically* a simple struct:

```
template<class A, class B>
struct pair {
    A first;
    B second;
};
```

`pair` has two public data members, and `first` must be located at offset zero.

`tuple` has more freedom.

Why do we say “algebraic”?

Okay, that was product types. But there’s another mathematical operation we can use!

`char`	**256** possible values
`bool`	**2** possible values (`true` and `false`)
`variant<char,bool>`	256 + 2 = **258** possible values

`tuple<char,bool>` holds a `char` value ***and*** a `bool` value.

`variant<char,bool>` holds a `char` value ***or*** a `bool` value.

Variant is a sum type

To find the size of the domain of a variant type, we take the *sum* of the sizes of its alternative types.

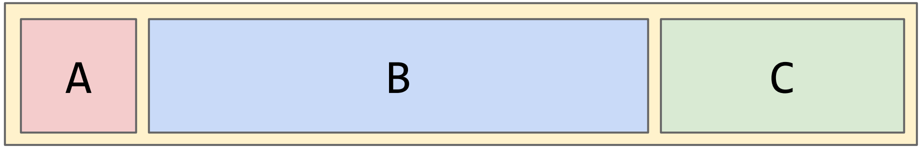
`A`	**A** possible values
`variant<A,B>`	**A+B** possible values
`variant<A,B,C,...>`	**A+B+C+...** possible values

Therefore `variant` is known as a ***sum type***.

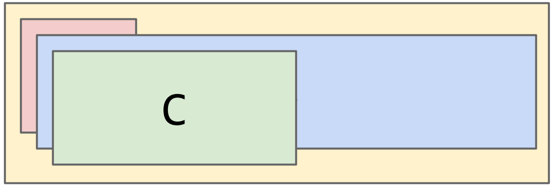
Memory layout of a sum type

Here's our ***conceptual*** picture of `variant` as opposed to `tuple`.

tuple<A,B,C>



variant<A,B,C>



Since we only need to store one alternative at a time, all three can be laid out at the *same* offset, just like in a union.

But this picture misses something!

Our ***conceptual*** picture isn't quite right. The STL's `variant` is designed to be type-safe.

A union doesn't know what type it holds. A `variant` does.

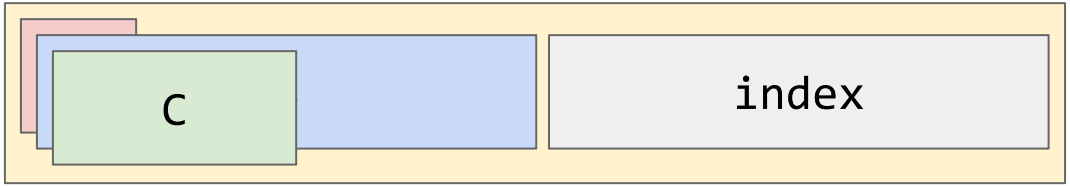
```
union U {
  int i; float f;
} u;
u.f = 3.14f;
int i = u.i;
  // i gets 1078523331
```

```
std::variant<int, float> v;

v = 3.14f;

int i = std::get<int>(v);
  // throws an exception!
```

Memory layout of a sum type

So, ***actually***, `variant` also stores an “index” field.

```
variant<A,B,C>
```



The index field tells us which of the variant’s “alternatives” is currently active.

```
size_t which = v.index();
```

libc++, libstdc++, and MSVC all use this same layout, with a 4-byte index field.

Finally, optional is another sum type

`optional` is the other ***sum type***. Like `pair`, `optional` has been optimized for certain common use-cases.

An `optional<A>` can store any value of type `A`, or `nullopt`.

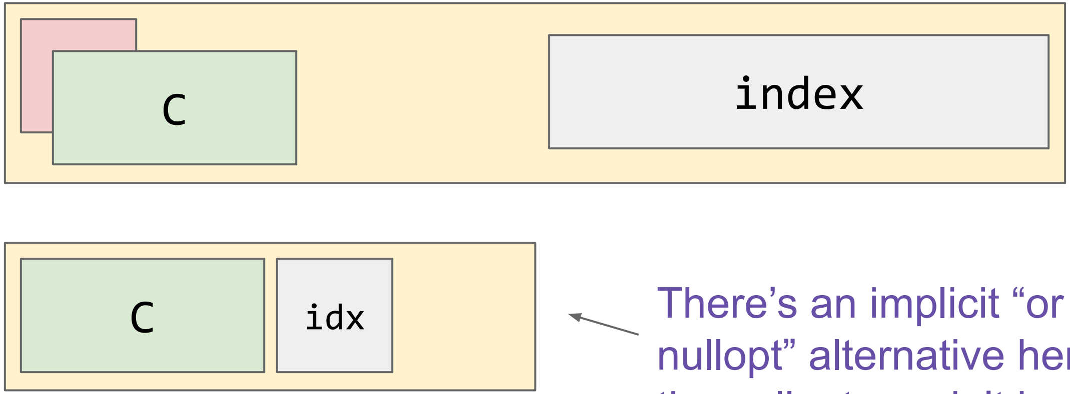
`A`	**A** possible values
`variant<A,B>`	**A+B** possible values
`optional<A>`	**A+1** possible values

Memory layout of optional

`optional`, like `variant`, must store an “index” field. But its index field can be smaller — just a `bool` — and in practice that’s what we see.

`variant<A,C>`

`optional<C>`



There’s an implicit “or nullopt” alternative here; but the nullopt needn’t be stored physically in memory.

What about `std::any`?

- `std::any` also arrived in C++17, alongside `optional<T>` and `variant<Ts...>`.
- It is ***not*** an algebraic type. We can't use math to say anything interesting about its domain, which is "union of all copyable types."
- `std::any` is a type-erasure type, similar to `std::function`.
- See my CppCon 2019 talk "Back to Basics: Type Erasure."

Still, I'll try to mention when there are commonalities between `std::any` and the algebraic types.

Quick motivation

Why use pair? (brief version)

`pair` is used many places in the classic STL.

```
using M = std::map<int, int>;
M myMap = {{1,2}, {2,4}, {3,6}};
std::pair<int, int> item = *myMap.begin();
```

Element type of `map`.

```
std::pair<M::iterator, bool> ii = myMap.insert({2,3});
```

Returned from `insert`.

```
std::vector<int> a = {1,2,3,5,7};
std::vector<long> b = {1,2,3,4,5};
std::pair<decltype(a)::iterator, decltype(b)::iterator> mm =
    std::mismatch(a.begin(), a.end(), b.begin(), b.end());
```

Returned from some algorithms, like `mismatch`, `equal_range`, and `uninitialized_move_n`.

But `std::ranges::mismatch` returns `ranges::mismatch_result`, not `pair`.

Why use tuple? (brief version)

Since the STL uses `pair` to simulate “returning multiple results,” you might imagine returning `tuple` when you need more than 2 results.

```
std::pair<int, int> minmax(int a, int b);
std::tuple<int, int, int> minmidmax(int a, int b, int c);
```

You might also imagine using a tuple to simulate a “pack” of data members:

```
template<class R, class... Args>
struct DeferredCall {
    std::tuple<Args...> arguments_;
    R operator()() const { ~~~ }
};
```

However, C++20 moved toward returning dedicated class types, e.g.

```
struct minmax_result {
  T min;
  T max;
};
```

and I recommend you do the same in your code.

Why use tuple? (brief version)

`tuple` is used (arcanely) to forward sets of arguments to `pair`'s constructor. More on this later.

```
std::pair<std::string, std::string> myPair(
    std::piecewise_construct,
    std::make_tuple("abc", 3),
    std::make_tuple(100, '*')
);
```

Construct `myPair.first` as `string("abc", 3)` and `myPair.second` as `string(100, '*')`.

We'll also see a couple of useful idioms with `std::tie`.

Why use optional? (brief version)

`optional` is never used in the STL, but it's the most useful for your own code.

In a return type, it represents "no answer."

```
std::optional<std::string> oGetenv(const char *name) {
    if (name is present) {
        return std::string(variable's value);
    } else {
        return std::nullopt;
    }
}
```

Why use optional? (brief version)

In a variable, it represents “no setting” (if that’s distinct from “set to empty”).

```
bool hasIncludePath_ = false;
std::string includePath_;
std::string getIncludePath() const {
    return hasIncludePath_ ? includePath_ : "/usr/include";
}
```

```
std::optional<std::string> includePath_;
std::string getIncludePath() const {
    return includePath_.value_or("/usr/include");
}
```

Why use optional? (brief version)

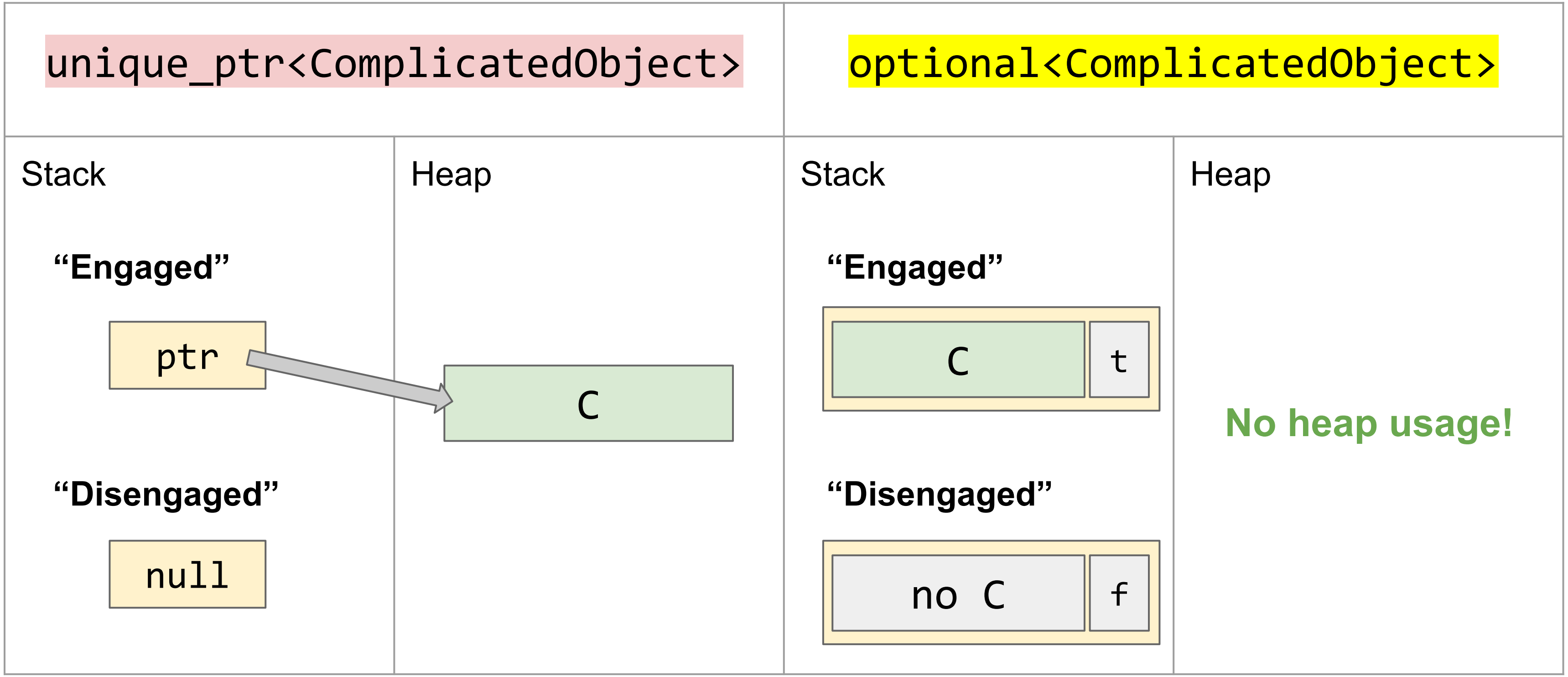
Or, it represents “not initialized yet,” for types with no default constructor.
This gives us the benefits of ***dynamic lifetime***, without any ***heap usage***.

```
std::unique_ptr<ComplicatedObject> obj_ = nullptr;
void setComplicated(int a, int b) {
    obj_ = std::make_unique<ComplicatedObject>(a, b);
}
```

```
std::optional<ComplicatedObject> obj_ = std::nullopt;
void setComplicated(int a, int b) {
    obj_.emplace(a, b);
}
```

The next slide draws this out in a diagram.

Why use optional? (brief version)



Why use variant? (brief version)

When you need a “sum type,” you might directly use `variant`.
Or you might use it as an implementation detail in writing your own sum type.

```
struct Simple { int fromx, fromy, tox, toy; };
struct Capture { int fromx, fromy, tox, toy; bool enpassant; };
struct Castle { bool kingside; };

class ChessMove {
    std::variant<Simple, Capture, Castle> details_;

    bool isSimple() const { return details_.index() == 0; }
    bool isCapturing() const { return details_.index() == 1; }
    bool isCastling() const { return details_.index() == 2; }
};
```

Common features of algebraic types

Now that you know how they look in memory, and have some general motivation for each of them, let’s talk about the C++ side of things.

All four algebraic types — `pair`, `tuple`, `optional`, `variant` — use some common concepts and vocabulary.

Let’s start with the new vocabulary.

“Engaged” and “disengaged”

- An `optional` which holds a value is often said to be *engaged*.
- An empty `optional` is said to be *disengaged*.
- These terms aren’t relevant to `pair`, `tuple`, or `variant`, but I explain them because they’re useful.
- Test whether an `optional` is engaged with `o.has_value()`, or via (explicit or contextual) conversion to bool.

```
if (auto o = oGetenv("foo")) ...
if (o) ...
```

“Engaged” and “disengaged”

By the way... Types which own a unique resource without being semantically “pointer-like” (and thus “nullable”) tend to have some “disengaged” state. This includes type-erased owners:

- `std::any` — `a.has_value()`
- `std::function` — conversion to bool

`optional` supports *both* of these methods!

And other movable RAII types:

- `std::future` — `fut.valid()`
- `std::unique_lock` — `(lk.mutex() != nullptr)`
- `std::promise, std::thread` — no explicit way to test

“Emplace” and in-place construction

- A variant or optional can be thought of as a “buffer” that *may or may not* hold a value of some given type.
- We can explicitly ***emplace*** an object into that buffer, in the same way as we might emplace a new element into a `vector` or `set`.
- The old value is destroyed before the new value is constructed.
- The arguments are perfectly forwarded to `T`’s constructor.

```
vectorOfString.emplace_back("abc", 3);

optionalString.emplace("abc", 3);

variantOfStringAndInt.emplace<std::string>("abc", 3);
```

“Emplace” and in-place construction

To construct a variant or optional with an object already emplaced, you have two options:

- The “basic” option: `variant` and `optional` are implicitly convertible from their alternative type(s).

```
optional<string> o1 = "foo";

variant<int, string> v1 = "foo";

  // Generally speaking, these will work fine.
  // But maybe not if the stored type is non-movable!
```

“Emplace” and in-place construction

- The “advanced” option: `variant` and `optional` have “in-place constructor” overloads to construct their contents directly in-place.

```
optional<string> o2(std::in_place, "abc", 3);

variant<int, string> v2(
    std::in_place_type<std::string>, "abc", 3);

variant<int, string> v3(
    std::in_place_index<1>, "abc", 3);
```

Also: `std::any a(std::in_place_type<std::string>, "abc", 3);`
`a.emplace<std::string>("abc", 3);`

`std::get` maps an index to a value

- You can think of a `pair`, `tuple`, or `array` as a ***mapping*** from an index in the range [0..n) to a value.
- Use `std::get<`*`index`*`>(obj)` to extract the *`index`*'th value.

```
std::pair<int, bool> p = {42, true};
std::cout << std::get<0>(p);  // synonym for p.first
std::get<1>(p) = false;       // synonym for p.second

std::array<int, 10> arr;
std::get<4>(arr) = 42;        // synonym for arr[4]

std::tuple<int, bool, double> t;
std::cout << std::get<1>(t);  // the bool element of t
```

`std::get` maps an index to a value

- `variant` is similar, but it's only a partial mapping.
- `std::get<I>` for any `I` other than the active index will throw an exception of type `bad_variant_access`. `variant` remains type-safe.

```
variant<int, string> v = 42;
std::get<0>(v) = 43;       // OK, it's the active alternative
std::get<1>(v) = "def";    // Throws std::bad_variant_access
std::get<2>(v) = 1;        // Error: get<2>(v) is just ill-formed
```

`std::get` can also map *type* to value

For `pair`, `tuple`, and `variant`, `std::get` also works with a type parameter — as long as it's unique and unambiguous.

```
pair<int, string> p = {42, "abc"};
std::cout << std::get<int>(p);  // synonym for p.first
std::get<string>(p) = "def";    // synonym for p.second
```

The implementation, simply, is that `get` has two overloads:

```
template<size_t I, class... Ts> T& get(tuple<Ts...>&);
template<class T, class... Ts> T& get(tuple<Ts...>&);
```

Easy to #include and declare

Or, using CTAD and/or direct-initialization, but personally I don’t recommend either...

```
#include <utility>
std::pair<int, char> p = {9, 'W'};          std::pair p{9, 'W'};

#include <tuple>
std::tuple<int, char> t = {9, 'W'};         std::tuple t{9, 'W'};

#include <optional>
std::optional<int> o = 9;                   std::optional o{9};
std::optional<int> no = std::nullopt;       std::optional<int> o{};

#include <variant>
std::variant<int, char> v = 'W';            variant<int, char> v{'W'};
```

Recursive default-construction

```
std::pair<std::string, int> p1{}, p2;
```

Both construct the pair with `{"", 0}`, as if by

```
std::string first{};
int second{};
```

```
std::tuple<int, char> t1{}, t2;
```

Likewise, constructs the tuple with `{0, 0}` in both cases.

```
std::optional<int> o1{}, o2;
```

In both cases, equivalent to `nullopt`: optionals are default-constructed into their disengaged state.

```
std::variant<int, char> v1{}, v2;
```

A default-constructed variant value-initializes its 0'th alternative. In this case the `int` gets `0`.

Special member functions

`pair`, `tuple`, `variant`, and `optional` all inherit their special members from their constituent types in the "natural" way.

- `foo<Ts...>` is copy-constructible if all `Ts...` are copy-constructible.
- `foo<Ts...>` is copy-assignable if all `Ts...` are copy-assignable.
  - variant also requires that all `Ts...` be copy-constructible, since it might need to change the active member of the LHS.
- `foo<Ts...>` is default-constructible if all `Ts...` are default-constructible.
  - Exception: Default-constructing a `variant<A,B,C>` will default-construct only its `A` alternative; so only `A` needs to be default-constructible.
  - And `optional` defaults to disengaged, so it's always default-constructible.

Comparison operators

Finally, `pair`, `tuple`, `variant`, and `optional` all have “natural” comparison operators.

- `pair<Ts...>` and `tuple<Ts...>` are ordered sequences, so they’re compared lexicographically.
- `optional<T>` is compared just like `T` would be, except that `nullopt` compares less-than any value from `T`’s domain.
- `variant<Ts...>` can be thought of as an ordered sequence `{index(),` *`value`*`}`, compared lexicographically.
  - Or, equivalently, `variant<Ts...>` can be thought of as `tuple<optional<Ts>...>` and compared lexicographically on *that* basis.

High-level questions?

FYI, we're about to discuss idioms around `optional`, `variant`, and `pair`/`tuple`, in that order.

About `optional`

```
#include <optional>

std::optional<int> o = std::nullopt;
```

`optional` for “not (yet) set”

Recall that `optional<T>` means “either a `T`, or nothing.”

It’s commonly used in business logic:

```
struct NetworkConnection {
    std::optional<std::string> password_;
    std::optional<Certificate> cert_;
};
```

`optional<string>` gives us distinct states for “no password” and “password is the empty string.”

`optional<Certificate>` gives us a state for “no certificate provided.”

Using optional

```
Certificate NetworkConnection::getCert() const {
    if (cert_.has_value()) {
        return cert_.value();
    } else {
        return getDefaultCertificate();
    }
}
```

`optional::value()` retrieves the held value (propagating value category appropriately).

If disengaged, `value()` will throw `bad_optional_access`.

Technically this means it does a redundant check in this case. But inlining will save us.

To eliminate that check...

Using optional

```
Certificate NetworkConnection::getCert() const {
    if (cert_) {
        return *cert_;
    } else {
        return getDefaultCertificate();
    }
}
```

`optional` has an explicit conversion to `bool`, synonymous with `has_value()`.

`operator*` is like `.value()`, except that it has UB instead of throwing, so it can skip the check.

Mnemonic: Just like `vector`'s `operator[]` and `.at()`, the terse punctuation has UB; the named method throws.

For clarity, I would always write `.has_value()`!

The one thing `operator bool` lets you do is write code that works generically with both `T*` and `optional<T>`, as part of a gradual upgrade strategy.

Using `optional::value_or`

```
Certificate NetworkConnection::getCert() const {
    return cert_.value_or(
        getDefaultCertificate()
    );
}
```

You can hide the test inside this library-provided convenience method.

Watch out for side effects, though. In the previous version, we never called `getDefaultCertificate` unless the optional was disengaged. In this version, we call `getDefaultCertificate` *before* testing the optional.

Setters for `optional` fields

```
class NetworkConnection {
    std::optional<Cert> cert_;

    void setCert(???);
};
```

For setters, do this!

- `void setCert(std::optional<Cert>)`
  - Usually what you want.
- `void setCert(Cert&&) + void setCert(std::optional<Cert>)`
  - Might save you a move, when called as `o.setCert(Cert("foo"))`.
- `void setCert(const std::optional<Cert>&)`
  - Basically never what you want. The caller usually can't give you a reference to "their" optional object because they haven't got one, so you'll get a temporary.

`optional` for “optional parameters”?

```
void openConnection(std::string_view host,
                    std::optional<Cert> cert = std::nullopt);
```

```
openConnection("example.com", Cert("foo"));  // OK, works
openConnection("example.com");               // OK, works
openConnection("example.com", std::nullopt); // OK, works
```

This may be exactly what you’re looking for. Personally, I wouldn’t use default function arguments for this (or anything ever). I’d either expect the caller to be passing along an `optional<Cert>` they got from somewhere else—

```
void openConnection(std::string_view, const std::optional<Cert>&);
```

or I’d expect the two different signatures to be used in different situations—

```
void openConnectionWithCert(std::string_view, const Cert&);
void openConnectionWithoutCert(std::string_view);
```

Questions on `optional`?

About `variant`

```
#include <variant>

std::variant<int, double> v = 3.14;
```

Detect the active alternative

There are several ways to figure out which alternative of a variant is active.

- `v.index()` — returns the active index as a `size_t`
- `std::holds_alternative<int>(v)` — I'd never recommend this
- `std::get_if<0>(&v)` — returns a pointer to the specified alternative, if it's active; otherwise returns `nullptr`
- `std::get_if<int>(&v)` — same but with types

Visitation

```
std::variant<int, double, std::string> v;

if (v.index() == 0) {
     std::cout << std::get<int>(v) << "\n";
} else if (v.index() == 1) {
     std::cout << std::get<double>(v) << "\n";
} else if (v.index() == 2) {
    std::cout << std::get<std::string>(v) << "\n";
}
```

This code works, but it’s ugly and error-prone.

We could have used `std::get<0>` etc. I’m foreshadowing the next slide.

Visitation

```
std::variant<int, double, std::string> v;

if (int *pi = std::get_if<int>(&v)) {
    std::cout << *pi << "\n";
} else if (double *pd = std::get_if<double>(&v)) {
    std::cout << *pd << "\n";
} else if (auto *ps = std::get_if<std::string>(&v)) {
    std::cout << *ps << "\n";
}
```

`std::get_if` is like `std::get`, but
- it takes a pointer
- it returns a pointer
- it returns `nullptr` if the specified alternative isn't active.

Compare it with `std::any_cast`.

Visitation with `std::visit`

```
std::variant<int, double, std::string> v;

auto printme = [](const auto& x) {
    std::cout << x << "\n";
};

std::visit(printme, v);
```

When every branch of the visitation does “the same thing” (syntactically speaking), you can use `std::visit`.

Internally, it does ***exactly the same thing*** as on the previous slide (modulo some clever optimizations). It branches on `v.index()`, and it instantiates a call to `printme(a)` for each alternative a that might be held by the variant.

Standard library functions usually take the lambda as their *last* argument, but visit takes it *first*, to leave room for a variadic number of `variants`.
With a two-parameter lambda `printus`, you could call `std::visit(printus, v1, v2)`.

https://godbolt.org/z/j7q7an

valueless_by_exception

```
std::variant<std::unique_ptr<int>, std::string> v;

v = std::make_unique<int>(127);
std::string x = "long enough to require heap allocation";
try {
    v.emplace<1>(x);
} catch (const std::bad_alloc&) {
    assert(v.valueless_by_exception());
    assert(v.index() == size_t(-1));
}
```

`v.emplace<1>(x)` destroys the `unique_ptr` to make room for the `string`. Contrariwise, v=x will effectively “copy-and-swap,” leaving v with its old value, as long as x is copyable and/or nothrow-movable.

This state basically never happens unless you use `emplace` in an unwise manner; and even if it does, you should keep it confined to `catch` blocks.

Questions on `variant`?

About `pair` and `tuple`

```
#include <utility>    // for pair
#include <tuple>      // for tuple
#include <functional> // for reference_wrapper

std::pair<int, int> p = {1,2};
std::tuple<int, double, int> t = {1,2,3};
```

Making pairs and tuples

Pairs and tuples represent “sequences” and so it makes sense to initialize them with braced initializer lists where convenient:

```
std::pair<int, bool> foo() {
    std::tuple<int, int, int> t = {1,2,3};
    return {42, false};
}
```

Or, use these helper functions:

```
auto foo() {
    std::tuple t = std::make_tuple(1, 2, 3);
    return std::make_pair(42, false);
}
```

C++17 CTAD alert!

Function templates `make_pair` and `make_tuple` will deduce their argument types.

Distinguish from `make_optional<T>`, `make_any<T>`, `make_shared<T>`, `make_unique<T>`.

pairs and tuples of references

You can't make an `optional<T&>` or `variant<T&, U&>`, but you ***can*** make a `tuple<T&, U&, ...>`, which will have "assign-through" behavior:

```
int a=1, b=2, c=3, d=4;
std::tuple<int&, int&> ab {a, b};
std::tuple<int&, int&> cd {c, d};

ab = cd;  // assign cd's values "through" ab

assert(a == 3 && b == 4);
```

Making `tuples` of references

You can implicitly create a tuple of references in at least three ways:

```
int a=1, b=2;
auto ab1 = std::make_tuple(std::ref(a), std::ref(b));
auto ab2 = std::tie(a, b);
auto ab3 = std::forward_as_tuple(a, b);
static_assert(std::is_same_v<decltype(ab1), std::tuple<int&, int&>>);
```

- `make_tuple` always captures values, except that it “decays” `reference_wrapper<T>` into `T&`.
- `tie` purposely works only on lvalues, and captures lvalue references.
- `forward_as_tuple` always captures references (to either lvalues or rvalues).

`forward_as_tuple` for argument lists

Remember our discussion of in-place construction?

```
std::optional<std::string> o(std::in_place, otherString, len);
```

For `pair` and `tuple`, we have to give ***two*** constructor argument lists.
So instead of an “in-place” constructor, we have a “piecewise” constructor.

```
std::pair<std::string, std::string> p(
    std::piecewise_construct,
    std::forward_as_tuple(otherString, len),
    std::forward_as_tuple(std::move(thirdString))
);
```

This is the primary use-case for `forward_as_tuple`. Very niche.

Multiple assignment with `tie`

We can use “assign-through” to simulate multiple assignment.

```
using Set = std::set<int>;
Set numbers;

Set::iterator it;
bool inserted;

std::tie(it, inserted) = numbers.insert(n);
```

`tie` returns `tuple<iterator&, bool&>`

`insert` returns `pair<iterator, bool>`

The assignment operator for `tuple<iterator&, bool&>` “assigns-through,” updating our named variables on the LHS with the values of the pair on the RHS.

Multiple assignment with `tie`

The STL provides a special tag type for “ignoring” one field during assign-through.

```
std::tuple<int, int, int> getXYZ();
int x, y;
std::tie(x, y, std::ignore) = getXYZ();
```

tie returns `tuple<int&, int&, const Magic&>`

`std::ignore` is a global variable of some magic type whose `operator=` accepts any type on the RHS and swallows it with no effect.

Multiple assignment with `tie`

Warning! Assign-through is not the same as ***simultaneous*** multiple assignment, as found in languages such as Python and Perl.

```
int x, y;

// Exchange the values of x and y
std::tie(x, y) = std::make_tuple(y, x);

// Exchange the values of x and y? No!
std::tie(x, y) = std::tie(y, x);
std::tie(x, y) = {y, x};
```

Don't try to be "clever" by over-using `std::tie`, and you'll never care about this particular pitfall.

Write one assignment per line. In this specific example, we should have used `std::swap`!

Using `tie` for comparison

The other useful idiom is “compare-through.”

Remember, `tuple` (and `pair`) do lexicographical comparison.

I wrote out this return type for clarity.
In real life you’d just say `auto`.

```
class Name {
  std::string firstname, lastname;

  std::tuple<const std::string&, const std::string&> asKey() const {
    return std::tie(lastname, firstname);
  }

  friend bool operator<(const Name& a, const Name& b) {
    return a.asKey() < b.asKey();
  }
};
```

tuple is not great for public APIs

Let's say I have a function that returns a hostname, a cert, and a time-to-live.

```
auto generateDefaultCert()
    -> std::tuple<std::string, Cert, double>;

// C++ before '17
auto hct = generateDefaultCert();
std::cout << "Made cert for host " << std::get<0>(hct) << "\n";

// C++17 structured binding
auto [host, cert, ttl] = generateDefaultCert();
std::cout << "Made cert for host " << host << "\n";
```

`tuple` is not great for public APIs

Using a named class type, with named fields, is ***much*** friendlier.

```
struct CertInfo { std::string host; Cert cert; double ttl; };
CertInfo generateDefaultCert();

// C++ before ’17
auto info = generateDefaultCert();
std::cout << "Made cert for host " << info.host << "\n";

// C++17 structured binding still permits this
auto [host, cert, ttl] = generateDefaultCert();
std::cout << "Made cert for host " << host << "\n";
```

In conclusion

- Use `std::optional` for “maybe a `T`” or “not a `T` ***yet***”
- Use `std::pair` and `std::tuple` for implementation details
  - Such as `std::tie` for “multiple assignment”
  - But prefer named classes in public interfaces
- Remember `std::in_place` and `std::piecewise_construct`
- Remember `std::visit` for variants
- Forget `std::variant::valueless_by_exception`

Questions?

Bonus Slides

Why use variant? (brief version)

`variant` can also be used as a poor man's `Expected<T>`.
I don't recommend this, but it shows basically what `variant` is capable of.

```
std::variant<std::string, std::errc> vGetenv(const char *name);

if (auto v = vGetenv("foo"); std::get_if<std::string>(&v)) {
    const auto& value = std::get<std::string>(v);
    std::cout << "Value is: " << value << "\n";
} else {
    std::error_condition error = std::get<std::errc>(v);
    std::cout << "Error was: " << error.message() << "\n";
}
```

https://godbolt.org/z/hqoP1W

I associate value_or with this pitfall

```
class NetworkConnection {
    static const int defaultIdleTimeout = 1000;
    ~~~
};

int NetworkConnection::getTimeout() const {
    return idleTimeout_.value_or(defaultIdleTimeout);
}
```

Can you spot the bug?

Solution on next slide.

```
test.o: In function `NetworkConnection::getTimeout() const':
test.cpp:10: undefined reference to `NetworkConnection::defaultIdleTimeout'
```

https://godbolt.org/z/hqoP1W

I associate value_or with this pitfall

```
class NetworkConnection {
    static constexpr int defaultIdleTimeout = 1000;
    ~~~
};

int NetworkConnection::getTimeout() const {
    return idleTimeout_.value_or(defaultIdleTimeout);
}
```

Static const members must still have an out-of-line definition!

Static constexpr members, or C++17 static *inline* const members, are freed of that restriction.

`value_or` takes its parameter by forwarding reference, which means it wants `defaultIdleTimeout`'s address. Often `value_or` is the only place in the program that asks for the *address*, rather than the *value*, of a static const data member.