Structured Bindings

Section 5

In this section

- "Destructuring" data
- *Stuctured bindings*
- Custom *structured bindings*

Destructuring data: pre-C++17

Part 5.1

In the past...

- Functions returning multiple values required effort on the caller side to use/inspect them
- Boilerplate was required when using output parameters, `std::pair` / `std::tuple`, or structs

In the past...

```
pair<iterator, bool> std::set::insert(const value_type&)
```

```
std::set<ip_address> online_machines;

void on_machine_startup(const ip_address& addr)
{
    const auto res = online_machines.insert(addr);

    if (res.second) {
        std::cout << "Machine " << addr << " now online";
    }
    else {
        std::cerr << "Machine " << *res.first << " seen before";
    }
}
```

In the past...

- `std::tie` used to provide a rudimentary way of destructuring data
  - It required objects to be *mutable* and *default-constructible*, and was verbose
  - Doesn't deduce types
  - Only works with `std::tuple` and `std::pair`

```
bool success;
std::set<ip_address>::iterator it;

std::tie(success, it) = online_machines.insert(addr);
```

Shortcomings

- Functions returning multiple values were discouraged due to verbosity
- `std::tie` is coupled to the Standard Library and not general
- `std::tie` and "output parameters"
  - Require the caller to prepare some "targets"
  - Prevents `const` from being used
  - Still verbose

Sneak peek - structured bindings

```
bool success;
std::set<ip_address>::iterator it;

std::tie(success, it) = online_machines.insert(addr);
```

↓

```
const auto [success, it] = online_machines.insert(addr);
```

Structured bindings

Part 5.2

Example - map insertion

```cpp
void on_machine_startup(const ip_address& addr)
{
    const auto [success, it] = online_machines.insert(addr);

    if (success)
    {
        std::cout << "Machine " << addr << " now online\n";
    }
    else
    {
        std::cerr << "Machine " << *it << " registered twice\n";
    }
}
```

Overview

```
const auto [success, it] = online_machines.insert(addr);
```

- The above is a *structured binding declaration*
- It introduces *names* for all the elements of the returned pair
- Deduces the types, allows usage of `const`
- Supports *structs*, *arrays*, and *custom types*
- Fully customizable

Syntax

```
/* qualifiers */ auto [/* identifier list */] = /* expr */;
```

- `auto` can be *cv-qualified* or be a reference
- At least one *identifier* must be provided
- The expression must be of *array* or *class* type

Semantics - array/struct

```
auto [a, b] = expr;
```

- `auto` applies to `expr` itself - not to `a` and `b`
- `a` and `b` are only *names* that can be used in the current scope
- `a` and `b` are not copied/moved (!)

Semantics - array/struct

```
const auto& [a, b] = expr;
```

- `const auto&` applies to `expr` itself
- `a` and `b` are not taken by reference (!)

Example - array

```
void print_coordinates(const std::array<int, 3>& data)
{
    const auto& [x, y, z] = data;
    std::printf("x=%d, y=%d, z=%d", x, y, z);
}
```

- `data` is referenced, not copied (due to `const auto&`)
- `x` is an alias for the 1st element of the array
- `y` is an alias for the 2nd element of the array
- `z` is an alias for the 3rd element of the array

Example - array

```cpp
void print_coordinates(const std::array<int, 3>& data)
{
    const auto& [x, y, z] = data;
    std::printf("x=%d, y=%d, z=%d", x, y, z);
}
```

...is roughly equivalent to...

```cpp
void print_coordinates(const std::array<int, 3>& data)
{
    const auto& arr = data;
    std::printf("x=%d, y=%d, z=%d", arr[0], arr[1], arr[2]);
}
```

Example - struct

```cpp
struct person { std::string _name; int _age; };

[[nodiscard]] std::string to_json(const person& p)
{
    const auto& [name, age] = p;
    return concat("{'name':", name, ",'age':", age, '}');
}
```

Example - struct

```cpp
[[nodiscard]] std::string to_json(const person& p)
{
    const auto& [name, age] = p;
    return concat("{'name':", name, ",'age':", age, '}');
}
```

...is roughly equivalent to...

```cpp
struct person { std::string _name; int _age; };

[[nodiscard]] std::string to_json(const person& p)
{
    const auto& e = p;
    return concat("{'name':", e.name, ",'age':", e.age, '}');
}
```

Semantics - tuple/pair/custom

```
auto [a, b] = expr;
```

- Let `E` be the type of `expr`
- Applies only when `std::tuple_size<E>` is defined
- `a` refers to an object of type `std::tuple_element_t<0, E>`, initialized with `get<0>(expr)`
- `b` refers to an object of type `std::tuple_element_t<1, E>`, initialized with `get<1>(expr)`

s

- The traits can be specialized for custom types

Example - tuple/pair

```
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& [p, a] : addresses)
        std::cout << "Person " << p << " lives at " << a << '\n';
}
```

- `p` is a `const person&` initialized from `std::get<0>(/* element */)`
- `a` is a `address&` initialized from `std::get<1>(/* element */)`

Example - tuple/pair

```cpp
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& [p, a] : addresses)
        std::cout << "Person " << p << " lives at " << a << '\n';
}
```

...is roughly equivalent to...

```cpp
void print_addresses(const std::map<person, address>& addresses)
{
    for (const auto& element : addresses)
    {
        const person& p = std::get<0>(element);
        address& a = std::get<1>(element);
        std::cout << "Person " << p << " lives at " << a << '\n';
    }
}
```

Customizing structured bindings

Part 5.3

Example custom type

```
struct io_channel
{
    struct sender   { /* ... */ };
    struct receiver { /* ... */ };

    auto make_sender()   { return sender{/* ... */}; }
    auto make_receiver() { return receiver{/* ... */}; }
};
```

Example custom type

```
struct io_channel
{
    auto make_sender()   { return sender{/* ...  */}; }
    auto make_receiver() { return receiver{/* ...  */}; }
};
```

Instead of...

```
io_channel channel;
auto sender = channel.make_sender();
auto receiver = channel.make_receiver();
```

...we would like the following:

```
io_channel channel;
auto& [sender, receiver] = channel;
```

Customization points

- Either must be valid:
  - `template <std::size_t I> T::get();`
  - `template <std::size_t I> get(T);`
- Both must be specialized:
  - `template <> struct std::tuple_size<T>;`
  - `template <std::size_t I> struct std::tuple_element<I, T>;`

Defining get

```
struct io_channel
{
    auto make_sender()   { return sender{/* ... */}; }
    auto make_receiver() { return receiver{/* ... */}; }
};
```

```
template <std::size_t I>
auto get(io_channel& c)
{
    if constexpr (I == 0) { return c.make_sender(); }
    else                  { return c.make_receiver(); }
}
```

Specializing std::tuple_size

```cpp
struct io_channel
{
    auto make_sender()   { return sender{/* ... */}; }
    auto make_receiver() { return receiver{/* ... */}; }
};
```

```cpp
namespace std
{
    template <>
    struct tuple_size<io_channel>
        : std::integral_constant<std::size_t, 2> { };
}
```

Specializing `std::tuple_element`

```
struct io_channel
{
    auto make_sender()   { return sender{/* ... */}; }
    auto make_receiver() { return receiver{/* ... */}; }
};
```

```
namespace std
{
    template <>
    struct tuple_element<0, io_channel>
    { using type = io_channel::sender; };

    template <>
    struct tuple_element<1, io_channel>
    { using type = io_channel::receiver; };
}
```

Final usage example

```cpp
struct io_channel
{
    auto make_sender()   { return sender{/* ...  */}; }
    auto make_receiver() { return receiver{/* ...  */}; }
};
```

```cpp
int main()
{
    io_channel channel;
    auto& [sender, receiver] = channel;
}
```

Pitfalls

Part 5.4

Bindings are not references

- The indentifiers introduced in the square brackets do **not** define new variables of reference type
- They merely are alternative names for existing entities
- Due to this fact, type deduction can sometimes be suprising

Bindings are not references

```
struct coordinate { int _x; int _y; };

coordinate c{0, 0};
auto& [x, y] = c;

x = 42;
assert(c._x == 42);
    // Reference semantics...

static_assert(std::is_same_v<decltype(x), int>);
    // ...but `x` is not a reference!
```

Bindings are not references

- Remember that `x` is just another name for `c._x`

```
coordinate c{0, 0};
auto& [x, y] = c;
x = 42;
assert(c._x == 42);
static_assert(std::is_same_v<decltype(x), int>);
```

…is equivalent to …

```
coordinate c{0, 0};
auto& c_ref = c;
c_ref._x = 42;
assert(c._x == 42);
static_assert(std::is_same_v<decltype(c_ref._x), int>);
```

Bindings are not references

- This behavior is also noticeable when using `decltype(auto)`

```
coordinate c{0, 0};

auto& [x, y] = c;

decltype(auto) test = x;
static_assert(std::is_same_v<decltype(test), int>);
    // `test` is a copy of `c._x`!

test = 42;
assert(c._x == 0);
```

Bindings are not implicitly moved

- Another consequence is that RVO/implicit move will not apply

```
struct person { std::string _name; int _age; };

std::string name_of_nth_person(const std::size_t idx)
{
    auto [name, age] = global_person_registry[idx];
    return name;
}

auto example_name = name_of_nth_person(0);
    // The string is copied (!) out of `name_of_nth_person`.
```

Bindings are not implicitly moved

- The previous code is equivalent to

```
std::string name_of_nth_person(const std::size_t idx)
{
    auto p_copy = global_person_registry[idx];
    return p_copy._name;
}
```

  - A move can be forced with `std::move`

```
std::string name_of_nth_person(const std::size_t idx)
{
    auto p_copy = global_person_registry[idx];
    return std::move(p_copy._name);
}
```

Qualifiers apply to the hidden object

- Any *cv-qualifier* applies to the hidden object, not to the bindings themselves

```
std::tuple<char, int&> get_data();

const auto [c, i] = get_data();

c = 'a';
    // Compile-time error. `decltype(c)` is `const char`.

i = 42;
    // OK. `decltype(i)` is `int&`.
```

Qualifiers apply to the hidden object

```
std::tuple<char, int&> get_data();
const auto [c, i] = get_data();
```

- The types are retrieved via `std::tuple_element`
  - `c` → `std::tuple_element_t<0, const std::tuple<char, int&>>`
  - `i` → `std::tuple_element_t<1, const std::tuple<char, int&>>`

```
std::tuple_element_t<1, const std::tuple<char, int&>>
// ... is ...
int& const
// ... which is ...
int&
```

Limitations

- Structured bindings cannot be captured in lambdas
  - This is being addressed for C++20
- Structured binding cannot be nested
- No way to ignore a particular binding (or to apply an *attribute* to it)

Section recap

- Structured bindings allow data destructuring
  - Useful for readability and conciseness
- They support structs, arrays, and user-defined types
  - Provide `std::tuple_size`, `std::tuple_element`, and `get` for custom types
- Syntax: `cv-qualifiers auto [id0, id1] = expr`
  - The *cv-qualifiers* apply to `expr`
- Bindings are not references, they are name aliases

Discussion

How C++ is becoming more and more terse

Exercise

- Create structured bindings for a custom class
  - `exercise3.cpp`
    - on Wandbox
    - on Godbolt

Q&A

Break

5 minutes