Quality of Life Improvements

Quality of Life Improvements

In this section

» Minor language features that can be immediately applied to improve readability and quality

Nested namespace definitions

C++11/14

- Nesting namespaces is useful for code organization
- Prior to C++17, it required multiple `namespace` definitions

```
namespace smartcars
{
    namespace lib
    {
        namespace ai
        {
            path calculate_path(const std::vector<node>& nodes);
        }
    }
}
```

In the past…

C++11/14

- With proper formatting, deep indentation can be avoided
- The solution is still suboptimal and not visually pleasing

```
namespace smartcars {
namespace lib {
namespace ai {

path calculate_path(const std::vector<node>& nodes);

}  // close namespace ai
}  // close namespace lib
}  // close namespace smartcars
```

In C++17…

C++17

- C++17 introduces *nested namespace definitions*
- Multiple namespaces can be defined with a single line of code
- The `::` token is used as a separator
- Anonymous namespaces are not supported

```
namespace smartcars::lib::ai
{
    path calculate_path(const std::vector<node>& nodes);
}
```

In C++17…

```
namespace smartcars::lib::ai { ... }
```

» …is exactly equivalent to…

```
namespace smartcars { namespace lib { namespace ai { ... } } }
```

Recommendations:

» Always use *nested namespace definitions* whenever possible

Optional message in `static_assert`

C++11/14

» Static assertions required a user-defined error message string

```
namespace smartcars::lib::util
{
    template <typename T>
    auto linear_interpolation(T a, T b, T value)
    {
        static_assert(std::is_floating_point<T>::value,
                      "`T` must be a floating point type");

        return a + value * (b - a);
    }
}
```

In the past…

C++11/14

- When the message is redundant, there is no way to avoid providing it
- A common workaround was to provide an empty message

```
namespace smartcars::lib::util
{
    template <typename T>
    auto linear_interpolation(T a, T b, T value)
    {
        static_assert(std::is_floating_point<T>::value, "");
        return a + value * (b - a);
    }
}
```

In C++17…

C++17

» The message can be omitted

```
namespace smartcars::lib::util
{
    template <typename T>
    auto linear_interpolation(T a, T b, T value)
    {
        static_assert(std::is_floating_point<T>::value);
        return a + value * (b - a);
    }
}
```

In C++17…

C++17

» Bonus: the Standard Library also provides `_v` shortcuts for type traits

```
namespace smartcars::lib::util
{
    template <typename T>
    auto linear_interpolation(T a, T b, T value)
    {
        static_assert(std::is_floating_point_v<T>);
        return a + value * (b - a);
    }
}
```

Recommendations:

» Omit `static_assert` messages if they do not add any value

» Use `_v` shortcuts for type traits whenever possible

Allow `typename` instead of `class` in template template parameters

C++11/14

» There was an inconsistency between `class` and `typename` in templates

```
namespace smartcars::lib::util
{
    template <template <typename...> class Container,
              typename... Ts>
    void serialize(const Container<Ts...>&);
}
```

» `template <typename...> class` was allowed

» `template <typename...> typename` was not

In C++17…

C++17

» `typename` can be used everywhere in template declarations/definitions

```
namespace smartcars::lib::util
{
    template <template <typename...> typename Container,
              typename... Ts>
    void serialize(const Container<Ts...>&);
}
```

Recommendations:

» Be consistent with the rest of your codebase

New rules for `auto` deduction with curly braces

C++11/14

» *List-initialization* of `auto` variables always deduced `std::initializer_list`

```
auto a = 0;    // `int`                          (copy-initialization)
auto b(0);     // `int`                          (direct-initialization)
auto c{0};     // `std::initializer_list<int>`   (direct-list-initialization)
auto d = {0}; // `std::initializer_list<int>`   (copy-list-initialization)
```

» The `auto c{0}` case has been deemed surprising by most developers

- It is also inconsistent with the other initialization syntaxes

In C++17…

C++17

» *Copy-list-initialization* will always deduce `std::initializer_list`

» *Direct-list-initialization* with one element will deduce from that element

- With multiple elements, the code is *ill-formed*

```
auto a = 0;    // `int`                          (copy-initialization)
auto b(0);     // `int`                          (direct-initialization)
auto c{0};     // `int`                          (direct-list-initialization)
auto d = {0}; // `std::initializer_list<int>` (copy-list-initialization)
auto e{0, 1}; // ill-formed, compilation error
```

In C++17…

» This *breaking* change aims to make usage of *direct-list-initialization* more uniform

Recommendations:

» Be consistent with the rest of your codebase

» In a new codebase, consider using curly braces as much as possible

- Be careful in templates

Allow attributes on namespaces and enumerators

C++11/14

- It was not possible to attach *attributes* to namespaces or enumerations
- This led to code repetition (in the case of namespaces)…

```
namespace smartcars::lib::protocol::v0
{
    [[deprecated("please use protocol v1")]]
    void send_message_to_car(message);

    [[deprecated("please use protocol v1")]]
    message get_message_from_car();
}
```

In the past…

C++11/14

» …and to not having a standard way to attach attributes to enumerators

```
namespace smartcars::lib::data
{
    enum class car_cpu_model
    {
        v1592     [[deprecated("discontinued cpu model")]],
        v1593, // ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        v1594, // Compilation error before C++17
    };
}
```

In C++17…

C++17

» Attaching *attributes* to namespaces or enumerators is now allowed

```
namespace smartcars::lib::protocol
{
    namespace [[deprecated("please use protocol v1")]] v0
    {
        void send_message_to_car(message);
        message get_message_from_car();
    }
}
```

» Notably, this cannot be used on a *nested namespace declaration*

In C++17…

```
namespace smartcars::lib::data
{
    enum class car_cpu_model
    {
        v1592 [[deprecated("discontinued")]],
        v1593,
        v1594,
    };
}
```

Recommendations:

- Avoid repetition of attributes by attaching them to namespaces
- Use attributes to deprecate entities in your code (and more…)

Initializers in `if` and `switch` statements

C++11/14

» Common pattern with return values that must be verified

- Declare a variable and check its value

```
int initialize_logger();

const int rc = initialize_logger();
if (rc == 0)
{
    log("logger initialization successful");
}
else
{
    std::cerr << "logger initialization error:" << rc;
}
```

In the past…

» This situation also happens when using containers

```
std::map<int, std::string> id_to_name{/* ... */};
const auto res = id_to_name.emplace(10, "Bjarne");

if (!res.second)
{
    std::cerr << "Name already exists\n";
}
```

In C++17…

C++17

» The syntax for `if` and `switch` is extended to allow variable declarations

```
std::map<int, std::string> id_to_name{/* ... */};
if (const auto res = id_to_name.emplace(10, "Bjarne");
    !res.second)
{
    std::cerr << "Name already exists\n";
}
```

In C++17…

» The syntax for `if` and `switch` is extended to allow variable declarations

```
if (/* init-statement */; /* condition */) { /* ... */ }
```

» …is equivalent to…

```
{
    /* init-statement */;
    if (/* condition */) { /* ... */ }
}
```

In C++17…

```
switch (/* init-statement */; /* condition */)
{
    case /* a */:
        /* ... */
        break;

    case /* b */:
        /* ... */
        break;
}
```

In C++17…

```
status_code get_machine_status(int node_id);

if(const status_code sc = get_machine_status(51284);
   sc == status_code::healthy)
{
    process_payload_from(51284);
}
else
{
    std::cerr << "Error: 51284 status code is " << sc << '\n';
}
```

Recommendations:

- Always try to reduce the scope of variables as much as possible
  - This feature can help with `if` and `switch` statements

`auto` non-type template parameters

C++11/14

» Taking non-type template parameters required a concrete type

```
template <typename T, T Value>
constexpr const char* as_string();

constexpr auto s = as_string<MyEnum, MyEnum::Enumerator0>();
```

```
std::integral_constant<int, 42>{};
std::integral_constant<long, 19481>{};
```

In the past…

- This results in unnecessary verbosity
- There was no “placeholder” for arbitrary non-type parameters

```
template <???>
constexpr const char* as_string();
```

In C++17…

C++17

» `auto` can be used to designate arbitrary non-type parameters

```
template <auto Value>
constexpr const char* as_string();

constexpr auto s = as_string<MyEnum::Enumerator0>();
// `decltype(Value)` is `MyEnum`
```

In C++17…

C++17

» `std::integral_constant` can be redefined as follows

```
template <auto X> struct constant { };

constant<42>{};   // `decltype(X)` is `int`
constant<'a'>{};  // `decltype(X)` is `char`
constant<50ul>{}; // `decltype(X)` is `unsigned long`
```

In C++17…

C++17

» Allows heterogeneous compile-time value lists

```
template <auto... Xs> struct values { };

values<4, 'b', 99ul>{};
// contains `int`, `char`, `unsigned long`
```

In C++17…

C++17

» Useful when “extracting” parameters from template classes

```
template <template <auto> typename Wrapper, auto X>
constexpr auto extractFirst(Wrapper<X>) { return X; }

static_assert(extractFirst(Foo<5>) == 5);
static_assert(extractFirst(Bar<'a'>) == 'a');
static_assert(extractFirst(Baz<50ul>) == 50ul);
```

In C++17…

Recommendations:

- Use non-type `auto` template parameters to:
  - Avoid repetition (e.g. `enum` or `constant`)
  - Make your code more generic
- Do not use `auto` if you need a particular type

Section recap

```
namespace smartcars::lib::array_util
{
    template <template <typename, auto> typename Container,
              typename T,
              auto Size>
    void foo(const Container<T, Size>& c)
    {
        static_assert(std::is_integral_v<T>);
        if (auto copy{c}; copy.empty())
        {
            // ...
        }
    }
}
```

Discussion

» How could the shown features improve your current projects?

Exercise

- Reduce boilerplate and improve readability in an existing code snippet
  - `exercise0.cpp`
    - on Wandbox
    - on Godbolt *(no `stdin` support)*

Bug Prevention With New Attributes

Bug Prevention With New Attributes

In this section

- Enforcing use of return values: `[[nodiscard]]`
- Being explicit in code: `[[maybe_unused]]` and `[[fallthrough]]`

[[nodiscard]]

- Removing all elements of a `std::vector` can be done with `.clear()`
- Beginners often use `.empty()` by mistake

```
void reload_addresses(std::vector<address>& addresses)
{
    addresses.empty(); // <== bug
    for (const auto& a : global_addresses())
    {
        addresses.emplace_back(a);
    }
}
```

A common mistake

- No compiler used to complain about this mistake

```
addresses.empty(); // ...?
```

- Even though the signature of `std::vector::empty` is as follows

```
bool std::vector<T, Allocator>::empty() const noexcept;
```

A common mistake

- By default, C++ assumes that not using a return value is *not a bug*
- This is true only when a function has *side effects*
  - Which is the minority of cases
- C++17 adds an *attribute* to warn if a return value is not used
  - `[[nodiscard]]`

A common mistake

```
addresses.empty();
```

```
warning: ignoring return value of function declared with
         'nodiscard' attribute [-Wunused-result]
    addresses.empty();
    ^~~~~~~~~~~~~~~
```

Usage

- `[[nodiscard]]` can be placed either on functions or types
- A warning will be issued if:
  - The result of a `[[nodiscard]]` function is unused
  - The result of a function returning a `[[nodiscard]]` type is unused
- The warning can be suppressed by casting to `void`

Example - marking a function

```
[[nodiscard]] port_status inspect_tcp_port(std::uint16_t port);
```

```
const port_status ps = inspect_tcp_port(27015); // OK

do_something(inspect_tcp_port(27015));          // OK

(void) inspect_tcp_port(27015);                 // OK

inspect_tcp_port(27015);                        // Warning (!)
```

Example - marking a type

```
struct [[nodiscard]] error_code { int value; };
error_code initialize_peripherals();
```

```
if (initialize_peripherals() == 0) { /* ... */ } // OK

const error_code ec = initialize_peripherals();  // OK

do_something(initialize_peripherals());          // OK

(void) initialize_peripherals();                 // OK

initialize_peripherals();                        // Warning (!)
```

Use cases

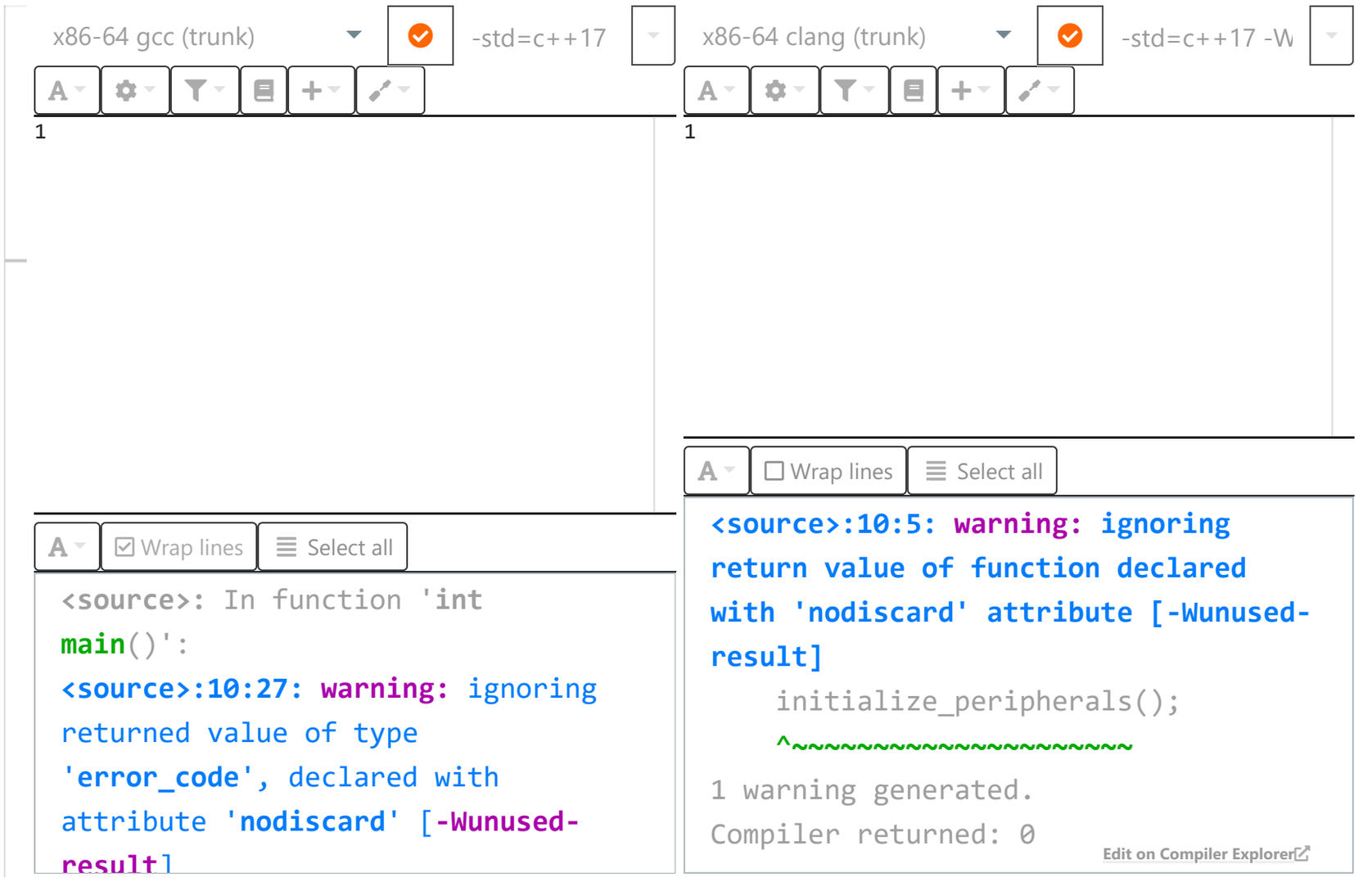
- Error codes or statuses
- Factory functions
- Resource handles
- Functions without side-effects (?)

In the C++17 Standard Library

- The following are marked `[[nodiscard]]`
  - All `.empty()` accessors
  - `operator ::new` and `std::allocator::allocate`
  - `std::async`
  - `std::launder` and `std::assume_aligned`

Sandbox

```
1   struct [[nodiscard]] error_code { int value; };
2
3   error_code initialize_peripherals()
4   {
5       return {};
6   }
7
8   int main()
9   {
10      initialize_peripherals();
11  }
12
13
```



Closing thoughts

- **Recommendations:**
  - Mark functions whose return value shouldn’t be ignored as `[[nodiscard]]`
  - Types that should never be ignored when returned should be `[[nodiscard]]`
- Food for thought:
  - Verbosity is a price to pay for compile-time safety
    - `[[nodiscard]]` should have been the default

`[[maybe_unused]]`

- The `[[maybe_unused]]` attribute is used to inform the compiler and humans that an entity might not be used
- Can be applied to most C++ entities: *classes*, *type aliases*, *data members*, *variables*, *functions*, and *enumerations*

Use cases

- An entity is only used in a particular build mode (e.g. debug)
- Marking unused parameters in functions
- Modern replacement for `(void)` cast

Example - assertions

```
void order_manager::send(const order& o)
{
    [[maybe_unused]] const bool valid_order =
        (o.id().size() > 0 && o.id().size() < 10)
     && (o.price() > 0)
     && (o.state() = order::state::unfulfilled);

    assert(valid_order);
    _socket.send(serialize(o));
}
```

Example - function parameters

```
struct message_listener
{
    virtual void on_received(const std::string& msg);
};

struct noop_message_listener : message_listener
{
    void on_received(
        [[maybe_unused]] const std::string& msg) override
    {
    }
};
```

Sandbox

```
1   #define NDEBUG 1
2
3   #include <string>
4   #include <cassert>
5
6   struct order { std::string id; };
7
8   void send_order(const order& o)
9   {
10      const bool valid_order =
11          (o.id.size() > 0 && o.id.size() < 10);
12
13      assert(valid_order);
14  }
15
16  int main()
17  {
18      send_order({"hello"});
19  }
```

x86-64 gcc (trunk) -std=c++17

```
1
```

Wrap lines | Select all

```
<source>: In function 'void send_order(const order&)':
<source>:10:16: warning: unused variable 'valid_order' [-Wunused-variable]
   10 |     const bool valid_order =
      |                ^
```

x86-64 clang (trunk) -std=c++17 -W

```
1
```

Wrap lines | Select all

```
<source>:10:16: warning: unused variable 'valid_order' [-Wunused-variable]
    const bool valid_order =
               ^
1 warning generated.
Compiler returned: 0
```

Edit on Compiler Explorer

Closing thoughts

» **Recommendations:**

- Use `[[maybe_unused]]` to mark entities that are only used in some build modes
- Use `[[maybe_unused]]` to mark intentionally unused parameters
  - Better readability compared to eliding them

`[[fallthrough]]`

- The `[[fallthrough]]` attribute is used to inform the compiler and humans that a `switch` case intentionally continues execution to the following one
- Can only be applied to *null statements* inside a `switch`
  - A *null statement* is a lonely `;`

Example

```
[[nodiscard]] config config_from_enum(option selected_option)
{
    bool enable_colors{false}, enable_formatting{false};

    switch (selected_option)
    {
        case option::colors_and_formatting:
            enable_colors = true;
            [[fallthrough]];
        case option::only_formatting:
            enable_formatting = true;
    }

    return {enable_colors, enable_formatting};
}
```

Sandbox

```
1   struct config { bool colors; bool formatting; };
2
3   enum class option
4   {
5       colors_and_formatting,
6       only_formatting
7   };
8
9   [[nodiscard]]
10  config config_from_enum(option selected_option)
11  {
12      bool enable_colors{false}, enable_formatting{fal
13
14      switch (selected_option)
15      {
16          case option::colors_and_formatting:
17              enable_colors = true;
18          case option::only_formatting:
19              enable_formatting = true;
20      }
21
22      return {enable_colors, enable_formatting};
23  }
```

x86-64 gcc (trunk) -std=c++17

x86-64 clang (trunk) -std=c++17 -W

A Wrap lines Select all

```
<source>: In function 'config config_from_enum(option)':
<source>:17:27: warning: this statement may fall through [-Wimplicit-fallthrough=]
   17 |             enable_colors = true;
```

A Wrap lines Select all

```
<source>:18:9: warning: unannotated fall-through between switch labels [-Wimplicit-fallthrough]
        case option::only_formatting:
        ^
<source>:18:9: note: insert '[[fallthrough]];' to silence this
```

Edit on Compiler Explorer

Closing thoughts

» **Recommendations:**

- Always mark `switch` cases that intentionally continue with `[[fallthrough]]`

Discussion

- Have you encountered any of these bugs?

Exercise

- Spot the bugs in an existing code snippet and apply attributes
  - `exercise1.cpp`
    - on Wandbox
    - on Godbolt

Destructuring Data With Structured Bindings

Destructuring Data With Structured Bindings

In this section

- “Destructuring” data
- *Structured bindings*
- Custom *structured bindings*

Destructuring data: pre-C++17

- 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…

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

```cpp
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…

C++11/14

» `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`

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

std::tie(it, success) = 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

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

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

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

Structured bindings

C++17

» Example - map insertion

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

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

Overview

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

- The above is a *structured binding declaration*
- It introduces *reference variables* or *names* for the elements of the destructured expression
- Deduces the types, allows usage of `const`
- Supports *structs*, *arrays*, and *custom types*
- Fully customizable

Syntax

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

- `auto` can be *cv-qualified* and/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

```
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…

```
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]);
}
```