"Quality of life" improvements

Section 1

In this section

- Nested `namespace` definitions
- Optional message in `static_assert`
- Allow `typename` instead of `class` in template parameters
- New rules for `auto` deduction with curly braces
- Allow attributes on `namespace` and `enum`
- Initializers in `if` and `switch` statements
- `auto` non-type template parameters

Nested namespace definitions

Part 1.1

In the past...

- 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...

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

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

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

}
}
}
```

In 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

```
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* when necessary

Optional message in static_assert

Part 1.2

In the past...

- 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...

- When the message is redundant, there is no way to avoid providing it
- A common workaround is 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...

- 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...

- 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);
    }
}
```

In C++17...

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 parameters

Part 1.3

In the past...

- 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...

- `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 ... >&);
}
```

Recommendation:

- Be consistent with the rest of your codebase

New rules for auto deduction with curly braces

Part 1.4

In the past...

- *List-initialization* of `auto` variables always deduced `initializer_list`

```
int main()
{
    auto a = 0;    // int
    auto b(0);     // int
    auto c{0};     // std::initializer_list<int>
    auto d = {0}; // std::initializer_list<int>
}
```

  - The `auto c{0}` case has been deemed surprising by most developers
    - It is also inconsistent with the other initialization syntaxes

In C++17...

- *Copy-list-initialization* will always deduce `initializer_list`
- *Direct-list-initialization* with one element will deduce from that element
  - With multiple elements, the code is *ill-formed*

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

In C++17…

- This 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 namespace and enum

Part 1.5

In the past...

- 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...

- ...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…

- Attaching *attributes* to namespaces or enumerations 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

Part 1.6

In the past...

- Common pattern with return values that must be verified
  - Declare a variable and check its value

```
int initalize_logger();

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

- 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';
}
```

In C++17...

Recommendations:

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

`auto` non-type template parameters

Part 1.7

In the past...

- 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, 1948l>{};
```

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...

- `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...

- `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...

- Allows heterogeneous compile-time value lists

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

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

In 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

- Nested `namespace` definitions
- Optional message in `static_assert`
- Allow `typename` instead of `class` in template parameters
- `auto` non-type template parameters
- Allow attributes on `namespace` and `enum`
- New rules for `auto` deduction with curly braces
- Initializers in `if` and `switch` statements

Section recap

```
namespace [[deprecated]] 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)*

Q&A

Break

5 minutes