Bug prevention with C++17 attributes

Section 2

In this section

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

[[nodiscard]]

Part 2.1

A common mistake

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

```cpp
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

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

↓

```cpp
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`

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]]

Part 2.2

Overview

- 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

```cpp
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

```cpp
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
    {
    }
};
```

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]]

Part 2.3

Overview

- 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

```cpp
[[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::formatting:
            _enable_formatting = true;
    }

    return {enable_colors, enable_formatting};
}
```

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

Q&A

Break

5 minutes