C++20 - the small things

Part 2 of 4: initializing structured spaceships

Overview

- Structured bindings & lambdas
- Initializers for range-based `for` loops
- Default / 3-way comparison

Structured bindings

- "Decomposing" aggregates
- C++11: `tie`

```
bool result;
std::tie(std::ignore, result) = set.insert(value);

int a,c; std::string b;
std::tie(a,b,c) = std::make_tuple(1, "Hello, world"s, 3);
```

Structured bindings

- "Decomposing" aggregates
- C++17: structured bindings

```
auto [it, result] = set.insert(value);
auto [a,b,c] = std::make_tuple(1, "Hello, world"s, 3);

// note: reference lifetime extension
const auto& [x,y] = std::make_tuple("Hello"s, "world"s);
```

Structured bindings

- You can bind to:
  - Arrays
  - Data members (non-static, public, direct)
  - Any type implementing the tuple protocol
- Introduces an alias to the existing subobjects, but not necessarily a reference

Structured bindings

- New in C++20: structured bindings can be captured in lambdas

```
const auto& [x,y] = std::make_tuple("Hello"s, "world"s);
auto l = [x, y] { return x + " " + y; };
assert(l() == "Hello world"s);
```

Structured bindings - outlook

- C++26 will support:
  - `constexpr` / `constinit`
  - Attributes
  - Packs

Initializers in control statements

- Introduced in C++17:

```
switch (http_error err = getHttpErr(); err.code()) {
    // handle err value cases
} // err goes out of scope

std::mutex m;
if (std::unique_lock<std::mutex> l(m, std::try_to_lock);
    l.owns_lock())
{
    std::cout << "mutex locked\n";
    // ... use the resource ...
} // lock released here through RAII
```

Initializers in control statements

- Benefits:
  - Simplifies limiting variable scope
  - Avoids name clashes
  - Better RAII usage
  - Better resource usage

Initializers in control statements

- C++20 allows this in range-based `for` loops, too

```
// if foo() returns by value
for (auto& x : foo().items()) { /* ... */ } // UB until C++23

// solution with C++20: init-statement
for (auto thing = foo(); auto& x : thing.items()) { /* ... */ }
```

Default & 3-way comparison

- Why do we have to write sooo much code?
- What if the class has 100 members?!

```
struct Comparable {
    bool operator<(const Comparable& other) { /*...*/ }
    bool operator<=(const Comparable& other) { /*...*/ }
    bool operator>(const Comparable& other) { /*...*/ }
    bool operator>=(const Comparable& other) { /*...*/ }
    bool operator==(const Comparable& other) { /*...*/ }
    bool operator!=(const Comparable& other) { /*...*/ }
};
```

We need a spaceship!

```cpp
int i = 2, j = 3;
auto res = i <=> j;

// compare the result against 0
if (res == 0) std::cout << "i == j";
if (res  < 0) std::cout << "i < j";
if (res  > 0) std::cout << "i > j";
```

spaceship 3-way comparison operator

- `operator<=>()` can return:
  - `std::strong_ordering`: no incomparable values, implies substitutability of equivalent values (think: integers)
  - `std::weak_ordering`: no incomparable values, no substitutability (think: case-insensitive strings)
  - `std::partial_ordering`: incomparable values, no substitutability (think: floats, NaN, 0/-0)
  - `auto` (when defaulted)

Spaceship operator - example

```cpp
struct MyError {
    int code;
    std::string details;

    friend std::weak_ordering operator<=>(const MyError& lhs, const MyError& rhs)
    {
        if (lhs.code == rhs.code) return std::weak_ordering::equivalent;
        if (lhs.code > rhs.code) return std::weak_ordering::greater;
        else return std::weak_ordering::less;
    }
    friend bool operator==(const MyError&, const MyError&) = default;
};

assert(MyError{404, "/foo not found"} == MyError{404, "/bar not found"});
```

Default all comparisons

```
struct Point {
    int x;
    int y;
    auto operator<=>(const Point&) const = default;
    // or better:
    // friend auto operator<=>(const Point&, const Point&) = default;
};
```

- Synthesizes all 6 comparisons (==, !=, <, <=, >, >=)
- May not be optimal for (in)equality - we want to short-circuit

Default all comparisons

```
struct Point {
    int x;
    int y;
    friend std::strong_ordering operator<=>(const Point&,
                                            const Point&) = default;
    friend bool operator==(const Point&, const Point&) = default;
};
```

Explicit is better than implicit

Short-circuited comparison

- You can default or implement each comparison individually (==, !=, <, <=, >, >=)

Comparing apples & oranges

- Cannot be defaulted :-(
- But C++20 *rewrites operators* during lookup
  - For `x < y`, `x <= y`, `x > y`, `x >= y` add all `operator<=>`
  - For `x < y`, `x <= y`, `x > y`, `x >= y`, `x <=> y` also add `y op x`
  - For `x != y`, add all `operator==` unless there's a matching `operator!=`
  - For `x == y` and `x != y`, add all `y == x` unless there's a matching `operator!=`