P.3 Express Intent (2 of N)
Easy-to-use-correctly implementations!

C++23 & Modern C++ enablers

Ankur M. Satle

https://ankursatle.wordpress.com

https://www.linkedin.com/in/ankursatle

Previously On

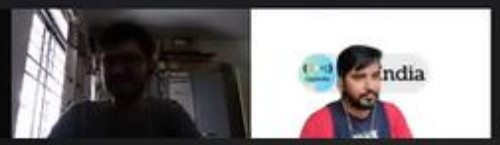
P.3 Express intent

Ankur M. Satle

https://ankursatle.wordpress.com

https://www.linkedin.com/in/ankursatle

+17 S MK KR VS



MG Mahendra Garodi (Guest)

AS Ameya Vikram Singh (Guest)

Hemil Ruparel

[CppIndia] P.3 Express Intent (1 of N) by Ankur Satle

Indispensable qualities

- Clean & direct implementations
- Clarity of intent
- Resulting in correct use
- Safety
- Enabling language & abstractions

- Maintainability
- Speed
- Debuggability

$whoami

- Architect at EXFO
- Polyglot when it comes to Natural languages
- But my mother-tongue is C++
- High-Performance Products
- Cloud Native products
- CppIndia

- https://ankursatle.wordpress.com
- ankursatle@gmail.com
- https://www.linkedin.com/in/ankursatle/
- https://twitter.com/AnkurSatle
- https://github.com/sankurm

Head spinning due to un-understandable code?

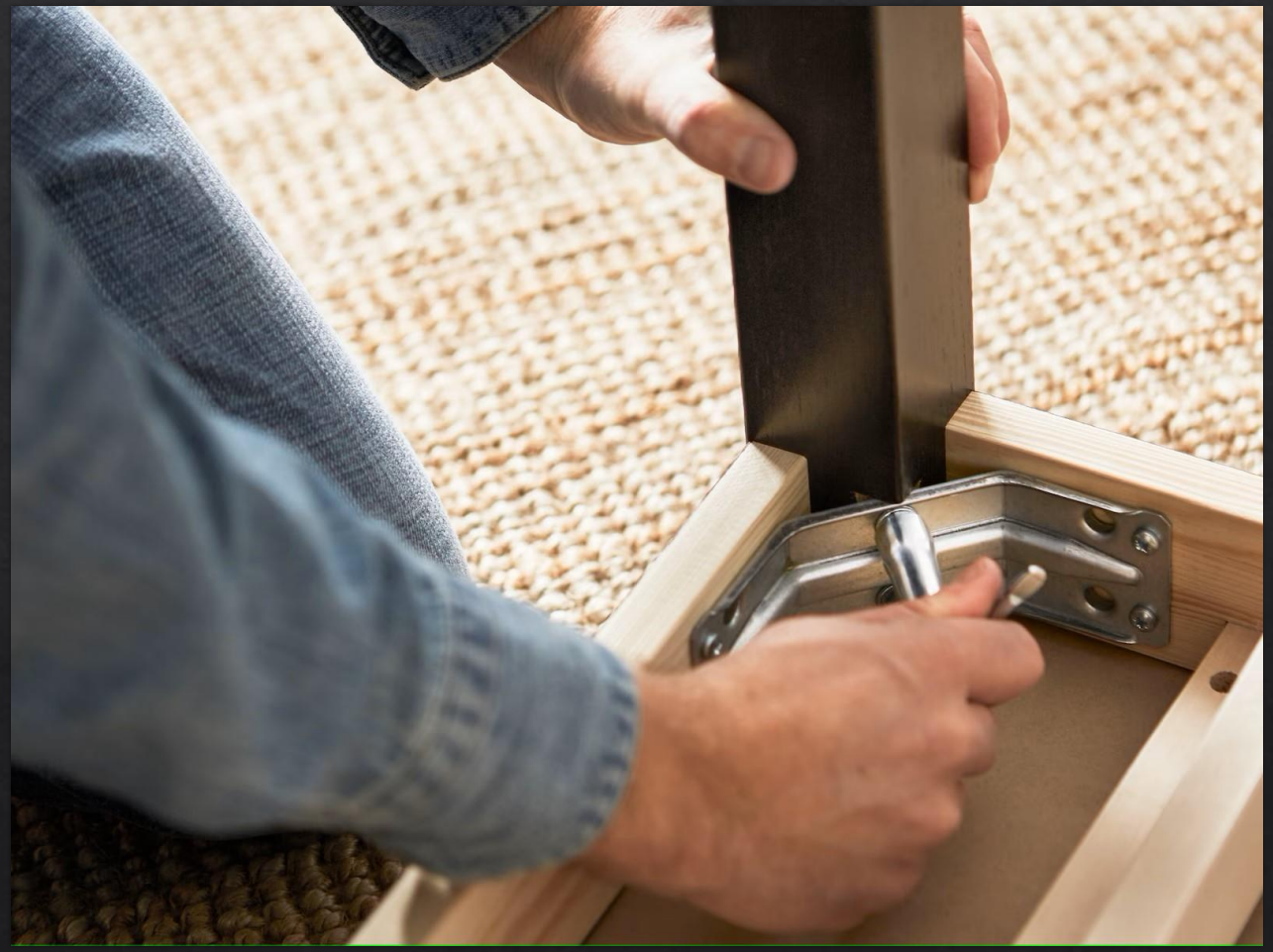
Must have the expert/author to guide?

Need usage instructions?

Image source: https://www.rockler.com/furniture-making-guide

Even a newbie should be able to find the way

The right way or nothing



Want to fall in the pit of success

P.3: Express intent

Reason Unless the intent of some code is stated (e.g., in names or comments), it is impossible to tell whether the code does what it is supposed to do.

Example

```
gsl::index i = 0;
while (i < v.size()) {
    // ... do something with v[i] ...
}
```

The intent of "just" looping over the elements of `v` is not expressed here. The implementation detail of an index is exposed (so that it might be misused), and `i` outlives the scope of the loop, which might or might not be intended. The reader cannot know from just this section of code.

Better:

```
for (const auto& x : v) { /* do something with the value of x */ }
```

Easy to use correctly, difficult to use incorrectly

Image source: https://www.rockler.com/furniture-making-guide

C++23

Consider

```
void send(const std::ve  r<  cd::fi esyst m  pat  >&  les,
          const std::v  to r<std::string>& metadata) {
    for (auto i = 0U  i < files.  ()  ) {
```

z or Z	• the sig std::si t t (since C++23)
both z/Z and u/U	• std::s ++23

```
}

int
```

```
#if __cpp_size_t_s  ix >= 202011L
    static_assert(  ::is_same_v<decltype(0UZ), std::size_t>);
    static_assert(st  is_same_v<decltype(0Z), std::make_sig  d_t<std::size_t>>);
#endif
}
```

```
    return 0;
}
```

REJECTED

REJECTED

REJECTED

In lock-step

By Les Chatfield from Brighton, England - A4 Union of South Africa motion and wheels, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=9645744

Pair them up!

```
void send(const vector<pair<path, string>>& file_detail);

vector<pair<path, string>> make_pairs(const vector<path>& files,
                                      const vector<string>& metadata);

int main(int, char**) {
    vector<path> files = get_files();
    vector<string> metadata = get_metadata();
    send(make_pairs(files, metadata));
    return 0;
}
```

Pair them up!

```
void send(const vector<pair<path, string>>& file_detail);

vector<pair<path, string>> make_pairs(const vector<path>& files,
                                      const vector<string>& metadata) {
    vector<pair<path, string>> pairs;
    transform(begin(files), end(files), begin(metadata), back_inserter(pairs),
        [](const auto& a, const auto& b) { return make_pair(a, b); });
    return pairs;
}

int main(int, char**) {
    vector<path> files = get_files();
    vector<string> metadata = get_metadata();
    send(make_pairs(files, metadata));
    return 0;
}
```

Pair them up!

```
void send(const vector<pair<path, string>>& file_detail) {
    for (const auto& info : file_detail) {
        send(info.first, info.second);
    }
}

vector<pair<path, string>> make_pairs(const vector<path>& files,
                                      const vector<string>& metadata);

int main(int, char**) {
    vector<path> files = get_files();
    vector<string> metadata = get_metadata();
    send(make_pairs(files, metadata));
    return 0;
}
```

Pair them up!

```
void send(const vector<pair<path, string>>& file_detail) {
    for (const auto& [file, meta] : file_detail) {
        send(file, meta);
    }
}

vector<pair<path, string>> make_pairs(const vector<path>& files,
                                      const vector<string>& metadata);

int main(int, char**) {
    vector<path> files = get_files();
    vector<string> metadata = get_metadata();
    send(make_pairs(files, metadata));
    return 0;
}
```

We just zipped them up together!

Zipping things together is common!

- `flat_map` with keys & values stored separately
- {A set of Data to update} in a database + {fetched rows}
- {List of students} zipped with a {list of results}
- Matching {expected} vs {actuals}

New in C++ fashion

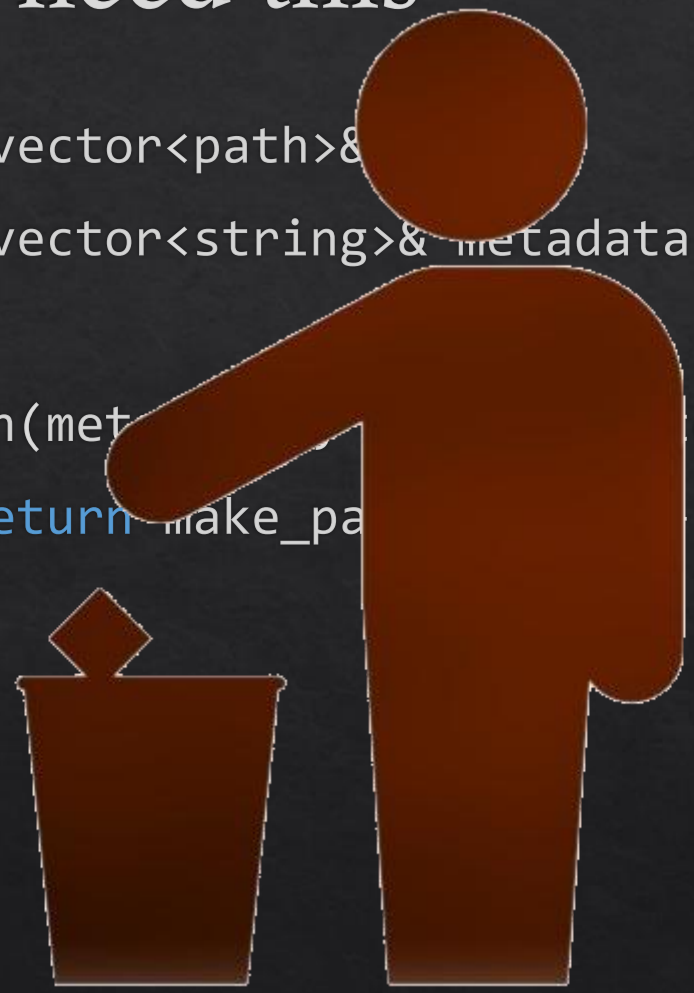
Using ranges

```
void send(const vector<path>& files,
          const vector<string>& metadata) {
    for (auto i = 0uz; i < files.size(); ++i) {
        send(files[i], metadata[i]);
    }
}
```

```
void send(const vector<path>& files,
          const vector<string>& metadata) {
    for (const auto& [file, meta] : std::views::zip(files, metadata)) {
        send(file, meta);
    }
}
```

Now, we do not need this

```
vector<pair<path, string>> make_pairs(const vector<path>&
                                      const vector<string>& metadata) {
    vector<pair<path, string>> pairs;
    transform(begin(files), end(files), begin(met        er(pairs),
        [](const auto& a, const auto& b) { return make_pa        );
    return pairs;
}
```



What if the ranges have different sizes

```
auto x = std::vector{1, 2, 3, 4};
auto y = std::list<std::string>{"α", "β", "γ", "δ", "ε"};
auto z = std::array{'A', 'B', 'C', 'D', 'E', 'F'};

print("\nzip(x,y,z):", "");
for (std::tuple<int&, std::string&, char&> elem :
                                    std::views::zip(x, y, z)) {
    std::cout << std::get<0>(elem) << ' '
              << std::get<1>(elem) << ' '
              << std::get<2>(elem) << '\n';

    std::get<char&>(elem) += ('a' - 'A'); // modifies the element of z
}
print("\nAfter modification, z: ", z);
```

```
Source view
x: 1 2 3 4
y: α β γ δ ε
z: A B C D E F

zip(x,y,z):
1 α A
2 β B
3 γ C
4 δ D

After modification, z:
a b c d E F
```

zip_view

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Ranges library / std::ranges::zip_view

std::ranges::views::zip, std::ranges::zip_view

Defined in header <ranges>

```
template< ranges::input_range... Views >
    requires (ranges::view<Views> && ...) && (sizeof...(Views) > 0)     (1)   (since C++23)
class zip_view : public ranges::view_interface<zip_view<Views...>>
```

```
namespace views {
    inline constexpr /*unspecified*/ zip = /*unspecified*/;              (2)   (since C++23)
}
```

Call signature

```
template< ranges::viewable_range... Rs >
    requires /* see below */                                                  (since C++23)
constexpr auto zip( Rs&&... rs );
```

1) zip_view is a range adaptor that takes one or more views, and produces a view whose *i*th element is a tuple-like value consisting of the *i*th elements of all views. The size of produced view is the minimum of sizes of all adapted views.

2) views::zip is a customization point object.

When calling with no argument, `views::zip()` is *expression-equivalent* to `auto(views::empty<std::tuple<>>)`.

Otherwise, `views::zip(rs...)` is *expression-equivalent* to `ranges::zip_view<views::all_t<decltype((rs))>...>(rs...)`.

zip_transform_view – it gets sweeter!

```
auto v1 = std::vector<float>{1, 2, 3};
auto v2 = std::list<short>{1, 2, 3, 4};
auto v3 = std::to_array({1, 2, 3, 4, 5});

auto add = [](auto a, auto b, auto c) { return a + b + c; };

auto sum = std::views::zip_transform(add, v1, v2, v3);

print("v1:  ", v1);
print("v2:  ", v2);
print("v3:  ", v3);
print("sum: ", sum);
```

```
v1:  1 2 3
v2:  1 2 3 4
v3:  1 2 3 4 5
sum: 3 6 9
```

[[assume!]]

C++ attribute: assume (since C++23)

Specifies that an expression will always evaluate to `true` at a given point.

Syntax

[[assume(*expression* **)]]**

expression - expression that must evaluate to `true`

Explanation

Can only be applied to a null statement, as in `[[assume(x > 0)]];`. This statement is called an *assumption*. If the expression (contextually converted to `bool`) would not evaluate to `true` at the place the assumption occurs, the behavior is undefined. Otherwise, the statement does nothing. In particular, the expression is not evaluated (but it is still potentially evaluated).

The purpose of an assumption is to allow compiler optimizations based on the information given.

The expression may not be a comma operator expression, but enclosing the expression in parentheses will allow the comma operator to be used.

Notes

If the expression would have undefined behavior, or if it would cause an exception to be thrown, then it does not evaluate to `true`.

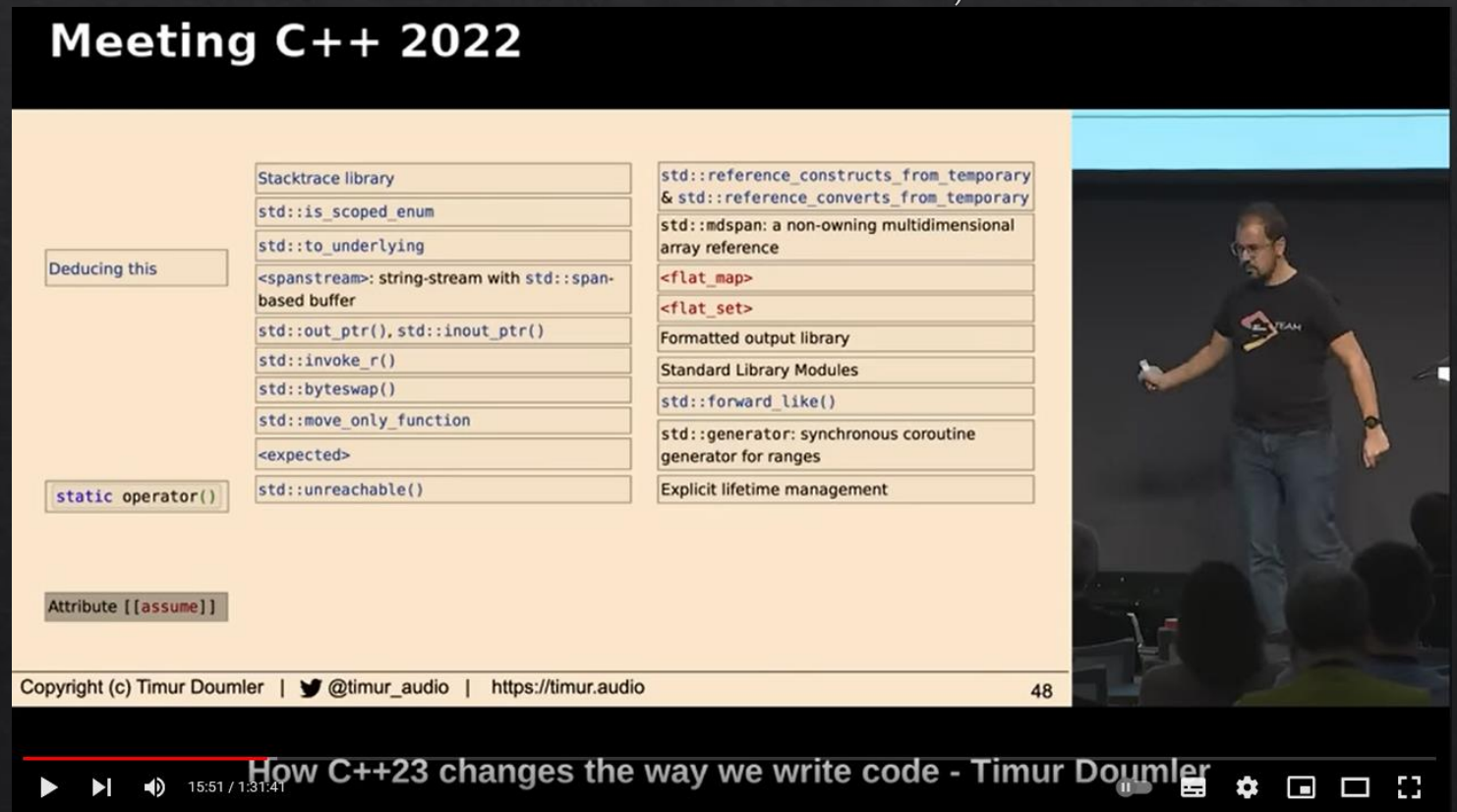
Since assumptions cause undefined behavior if they do not hold, they should be used sparingly. They are not intended as a mechanism to document the preconditions of a function or to diagnose violations of preconditions. Also, it should not be presumed, without checking, that the compiler actually makes use of any particular assumption.

2 vectors with [[assume(…)]]

```
void send(const std::vector<std::filesystem::path>& files,
          const std::vector<std::string>& metadata) {
    [[assume(files.size() == metadata.size())]];
    for (auto i = 0uz; i < files.size(); ++i) {
        send(files[i], metadata[i]);
    }
}
```

assume

- https://en.cppreference.com/w/cpp/language/attributes/assume
- Potentially evaluated
- If false or results in undefined behaviour or throws, behaviour is undefined



Some useful optimizations assume

- Assume that the size of an array is a power of 2
- Vectorization – alignment of memory is key
- Alternative for the restrict keyword in C?

- Beware of undefined behaviour
- Do not confuse this with [[likely]] & [[unlikely]]

C language – `restrict`

```c
void f(int n, int * restrict p, int * restrict q) {
    *p += n;
    *q += n;
}
void g(void) {
    extern int a, b;
    f(50, &a, &b);  // OK
    f(50, &a, &a);  // Undefined behaviour: `a` accessed through both `p` & `q` in `f`
}
```

- See https://en.cppreference.com/w/c/language/restrict

C++ – assume

```
void f(int n, int * p, int * q) {
    [[assume(p != q)]];
    *p += n;
    *q += n;
}
void g(void) {
    extern int a, b;
    f(50, &a, &b); // OK
    f(50, &a, &a); // Undefined behavior: a is accessed through both p and
q in f
}
```

C++ attribute: likely, unlikely (since C++20)

Allow the compiler to optimize for the case where paths of execution including that statement are more or less likely than any alternative path of execution that does not include such a statement

Syntax

`[[likely]]`	(1)
`[[unlikely]]`	(2)

Explanation

These attributes may be applied to labels and statements (other than declaration-statements). They may not be simultaneously applied to the same label or statement.

1) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are more likely than any alternative path of execution that does not include such a statement.
2) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are less likely than any alternative path of execution that does not include such a statement.

A path of execution is deemed to include a label if and only if it contains a jump to that label:

```
int f(int i)
{
    switch(i)
    {
        case 1: [[fallthrough]];
        [[likely]] case 2: return 1;
    }
    return 2;
}
```

`i == 2` is considered more likely than any other value of `i`, but the `[[likely]]` has no effect on the `i == 1` case even though it falls through the `case 2:` label.

Part 2 of N
Thank you!

Ankur M. Satle

https://ankursatle.wordpress.com

https://www.linkedin.com/in/ankursatle