COMP6771
Advanced C++ Programming

Week 2
Libraries

What's a library?

Why do we want to use libraries?

```cpp
auto what_am_i(std::vector<int> const& v,
               int const x) -> int {
    for (auto i = 0; i <= ranges::distance(v); ++i) {
        if (v[i] == x) {
            return i;
        }
    }
    return ranges::distance(v);
}
```

Why do we want to use libraries?

```cpp
auto find(std::vector<int> const& v,
          int const x) -> int {
    for (auto i = 0; i <= ranges::distance(v); ++i) {
        if (v[i] == x) {
            return i;
        }
    }
    return ranges::distance(v);
}
```

Why do we want to use libraries?

```cpp
auto find(std::vector<int> const& v,
          int const x) -> int {
    for (auto i = 0; i <= ranges::distance(v); ++i) {
        if (v[i] == x) {
            return i;
        }
    }
    return ranges::distance(v);
}
```

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Code in a popular library is often:

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Code in a popular library is often:

- Well-documented

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Code in a popular library is often:

- Well-documented
- Well-tested

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Code in a popular library is often:

- Well-documented
- Well-tested
- Well-reviewed

Why do we want to use libraries?

"Every line of code you don't write is bug-free!"

Code in a popular library is often:

- Well-documented
- Well-tested
- Well-reviewed
- Has lots of feedback

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library
- Abseil

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library
- Abseil
- Catch2 (test framework)

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library
- Abseil
- Catch2 (test framework)
- {fmt}

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library
- Abseil
- Catch2 (test framework)
- {fmt}
- gsl-lite

Libraries COMP6771 uses

We use the following extremely popular libraries

- C++ standard library
- Abseil
- Catch2 (test framework)
- {fmt}
- gsl-lite
- range-v3

You are not expected to learn everything in all of these libraries.

We will instead cherry-pick certain useful components from each.

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {      // Opens up a context for testing



























}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {      // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.
























}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
                                                          // scope run for *each* SECTION.





















    }
}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                   // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);
















    }
}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {      // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                   // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);

        SECTION("check we can insert elements to the front") {









        }
        SECTION("check we can remove elements from the back") {


        }
    }
}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                   // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);

        SECTION("check we can insert elements to the front") {
            auto const result = v.insert(v.begin(), -1);
            REQUIRE(ranges::distance(v) == 2);
            CHECK(result == v.begin());
            CHECK(v[0] == -1);
            CHECK(v.front() == v[0]);

            CHECK(v[1] == 0);
            CHECK(v.back() == v[1]);
        }

        SECTION("check we can remove elements from the back") {


        }
    }
}
```

Catch2 recap

```
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                   // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);

        SECTION("check we can insert elements to the front") {
            auto const result = v.insert(v.begin(), -1);
            REQUIRE(ranges::distance(v) == 2);
            CHECK(result == v.begin());
            CHECK(v[0] == -1);
            CHECK(v.front() == v[0]);

            CHECK(v[1] == 0);
            CHECK(v.back() == v[1]);
        }

        SECTION("check we can remove elements from the back") {


        }
    }
}
```

Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                    // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);

        SECTION("check we can insert elements to the front") {
            auto const result = v.insert(v.begin(), -1);
            REQUIRE(ranges::distance(v) == 2);
            CHECK(result == v.begin());
            CHECK(v[0] == -1);
            CHECK(v.front() == v[0]);

            CHECK(v[1] == 0);
            CHECK(v.back() == v[1]);
        }

        SECTION("check we can remove elements from the back") {


        }
    }
}
```

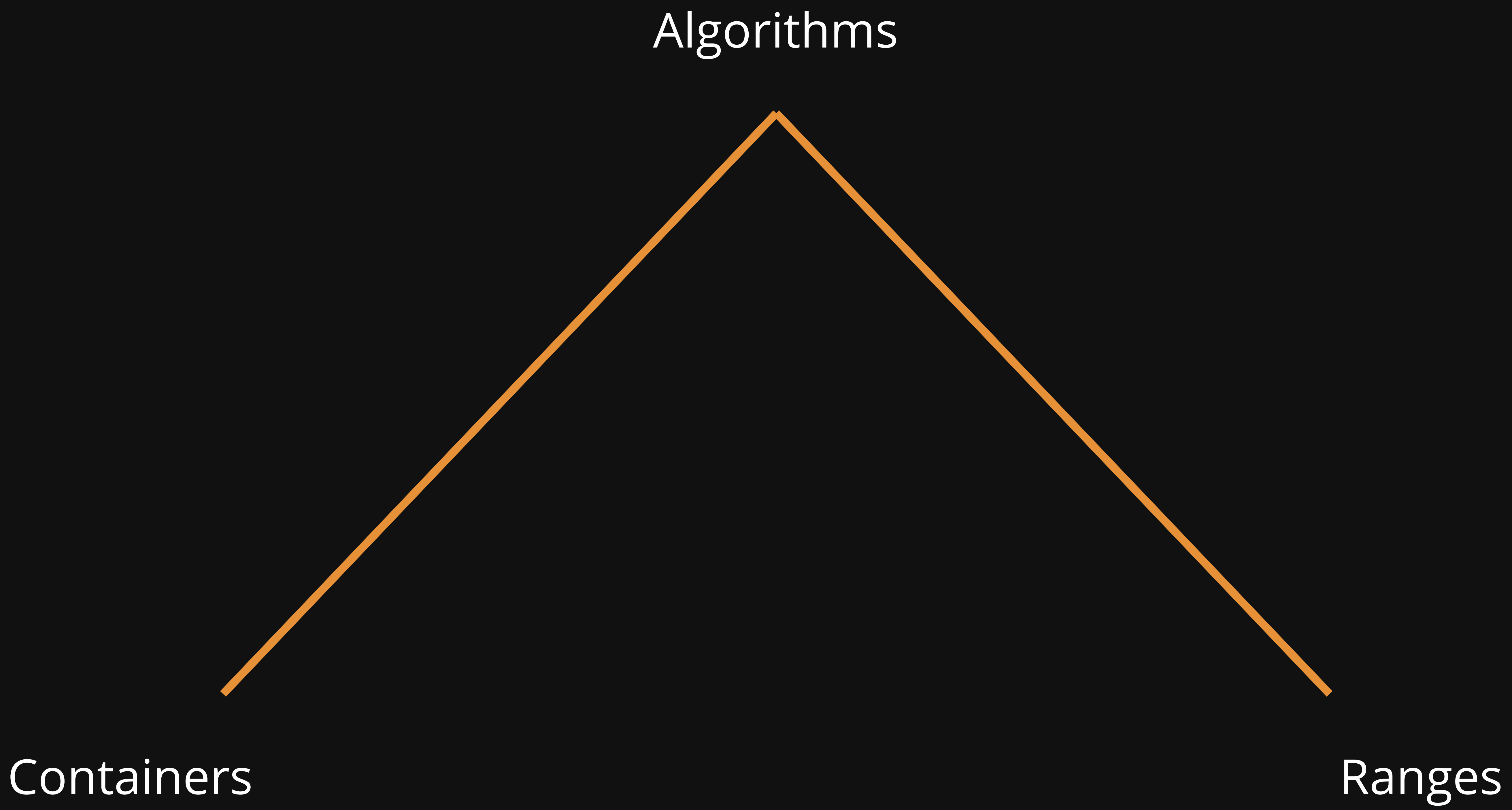
Catch2 recap

```cpp
#include <catch2/catch.hpp>

TEST_CASE("empty vectors") {     // Opens up a context for testing
    auto v = std::vector<int>();

    REQUIRE(v.empty()); // Aborts the test case (not the program) on failure.

    SECTION("check we can insert elements to the back") { // Opens a sub-context, where everything in the outer
        v.push_back(5);                                   // scope run for *each* SECTION.
        REQUIRE(ranges::distance(v) == 1);

        CHECK(v[0] == 1); // Gives a meaningful message on failure, but doesn't abort.
        CHECK(v.back() == v[0]);

        SECTION("check we can insert elements to the front") {
            auto const result = v.insert(v.begin(), -1);
            REQUIRE(ranges::distance(v) == 2);
            CHECK(result == v.begin());
            CHECK(v[0] == -1);
            CHECK(v.front() == v[0]);

            CHECK(v[1] == 0);
            CHECK(v.back() == v[1]);
        }

        SECTION("check we can remove elements from the back") {
            v.pop_back();
            CHECK(v.empty()); // remember that each section inherits an independent context from its parent scope
        }
    }
}
```

Themes

Algorithms

Themes

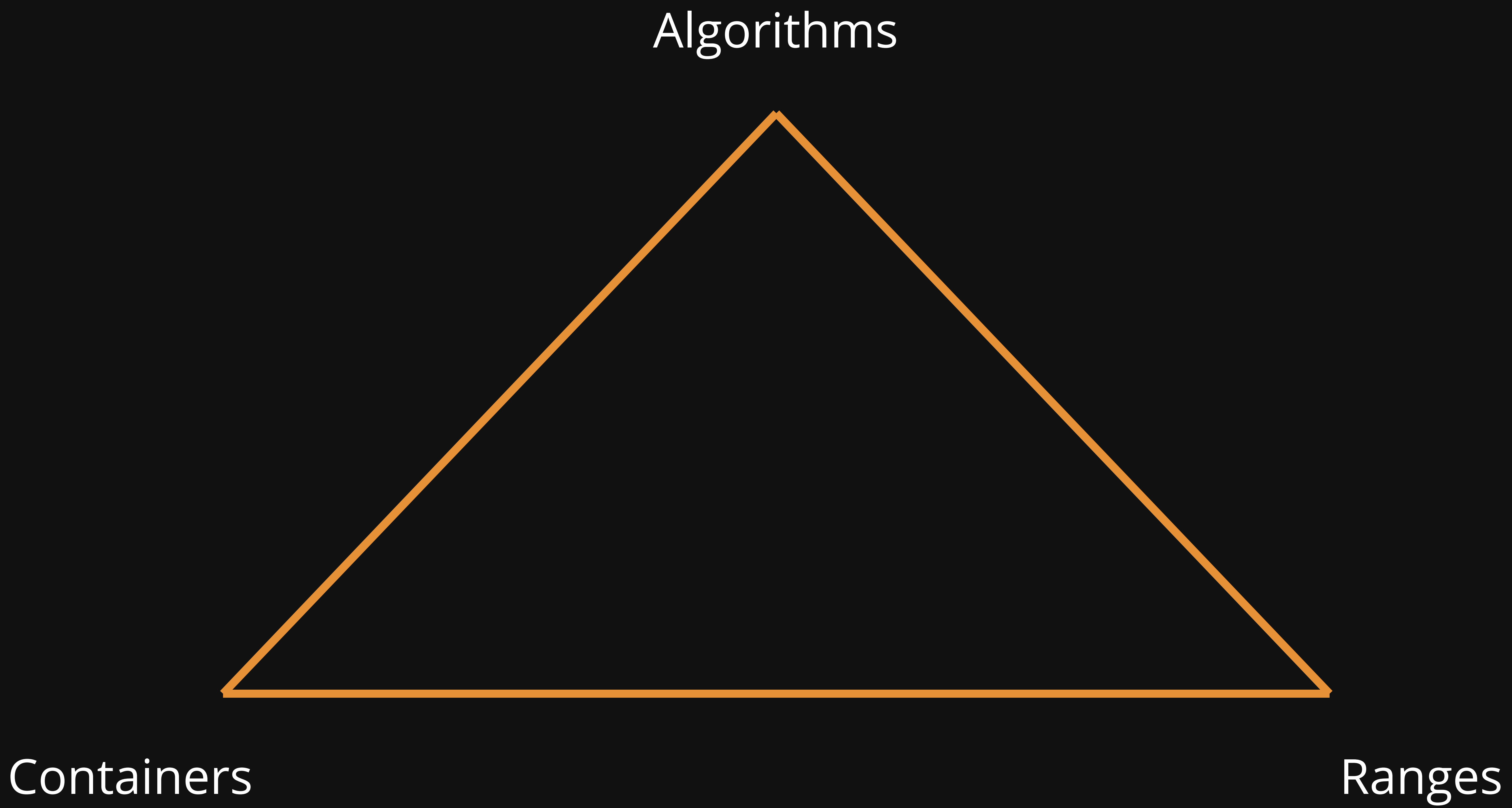


Algorithms

Containers

Ranges

Themes

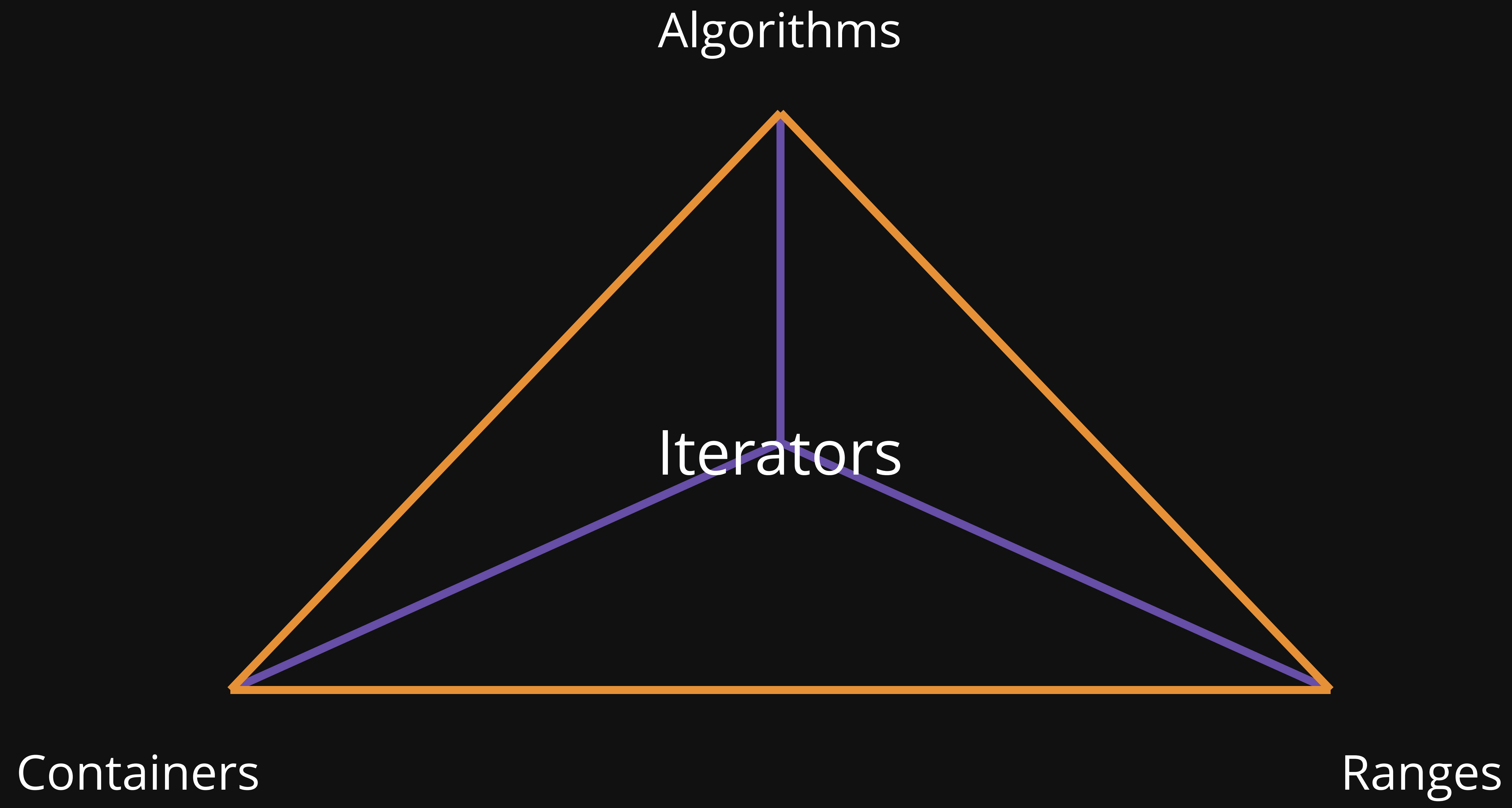


Algorithms

Containers

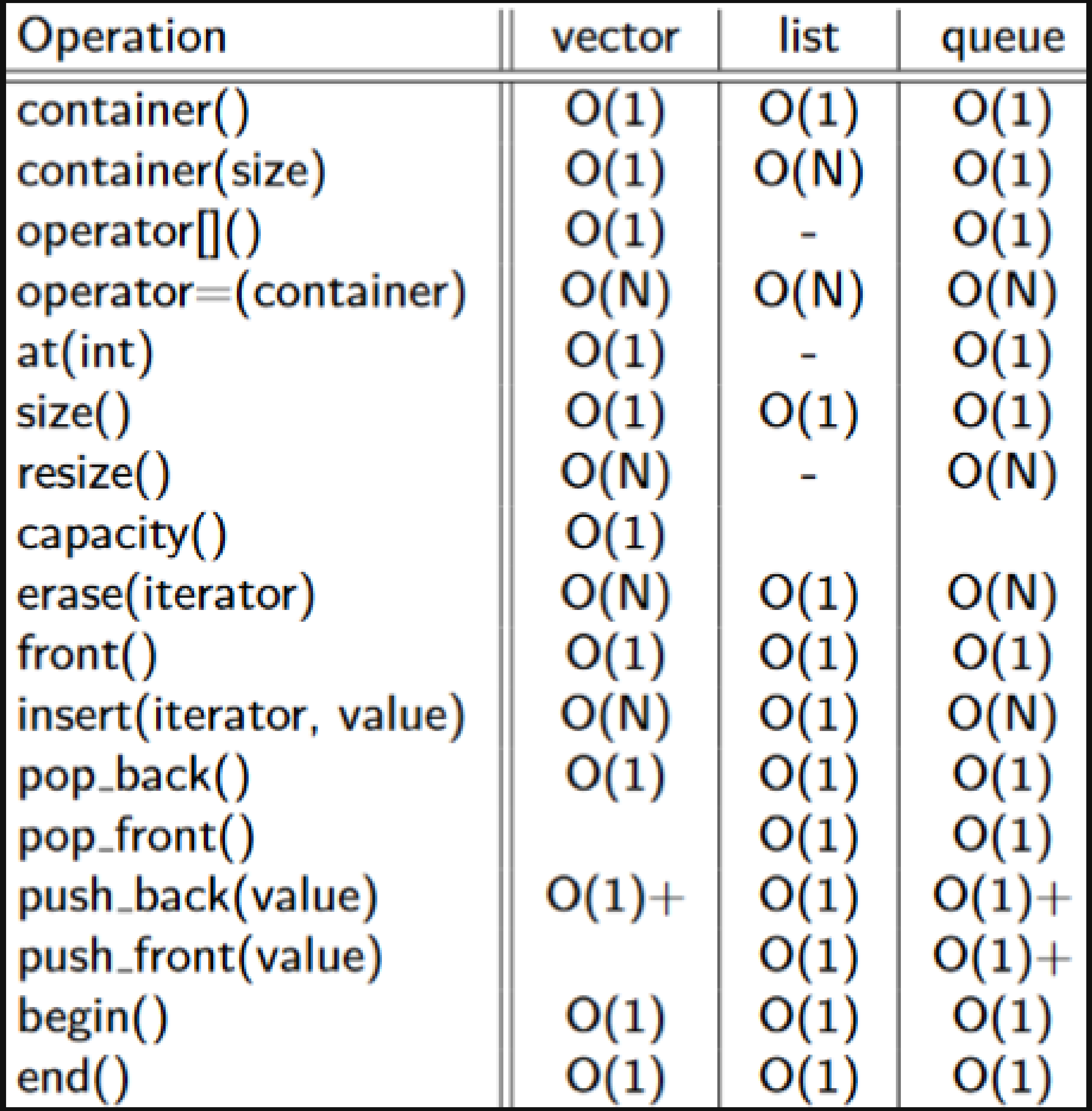
Ranges

Themes



Containers

Abstractions of common data structures.



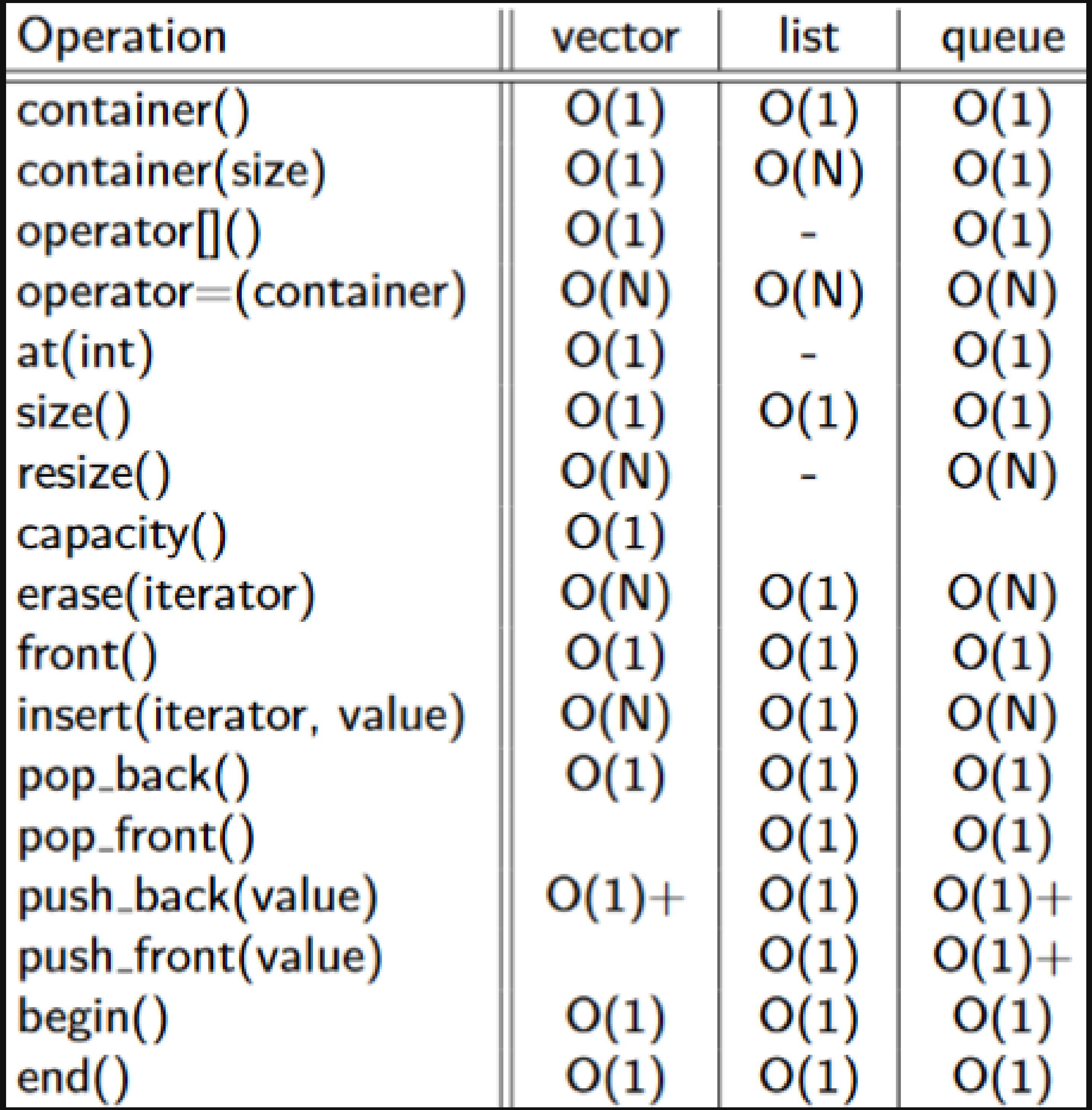
Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

https://en.cppreference.com/w/cpp/container

Containers

Abstractions of common data structures.

Are objects that you can "put" other objects "into".



Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

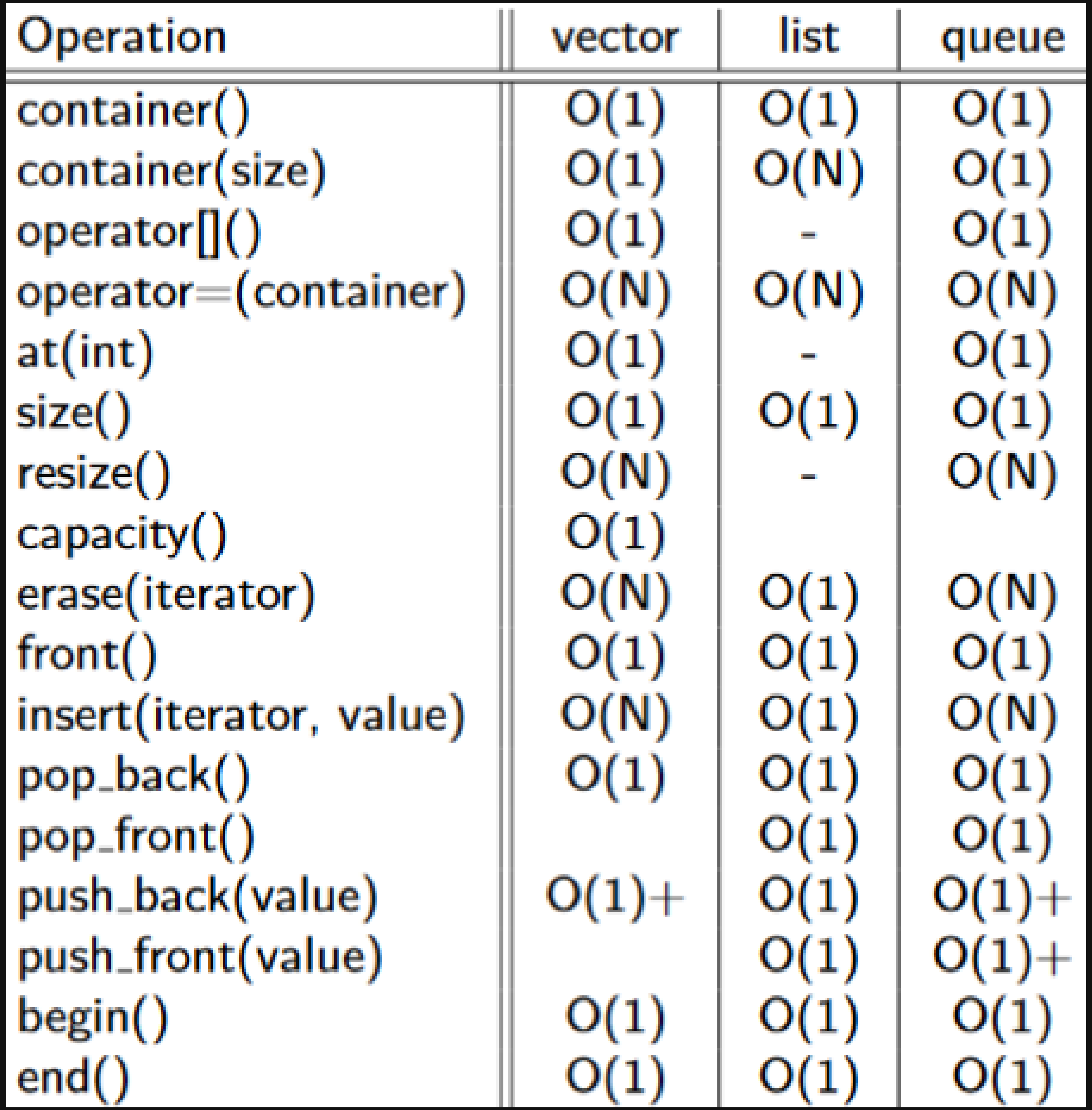
https://en.cppreference.com/w/cpp/container

Containers

Abstractions of common data structures.

Are objects that you can "put" other objects "into".

Container operations vary in time and space complexity.



Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

https://en.cppreference.com/w/cpp/container

Containers

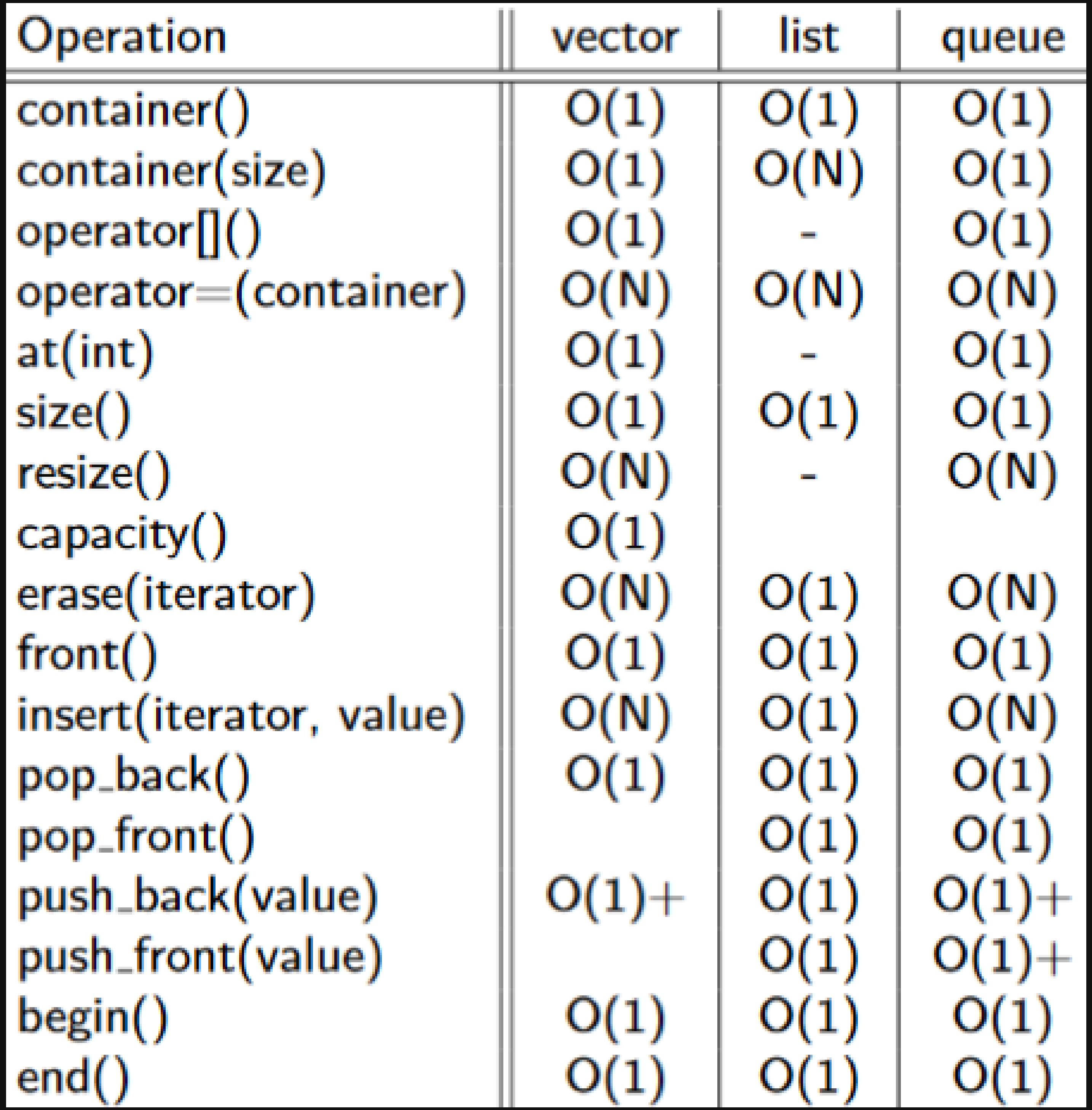
Abstractions of common data structures.

Are objects that you can "put" other objects "into".

Container operations vary in time and space complexity.

Performance has a basis in physics (see Week 10).

std::vector is always the default container (see Week 10).



Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

https://en.cppreference.com/w/cpp/container

Sequence containers

Organises a finite set of objects into a strict linear arrangement.

`std::vector`	Dynamically-sized array.
`std::array`	Fixed-sized array.
`std::deque`	Double-ended queue.
`std::forward_list`	Singly-linked list.
`std::list`	Doubly-linked list.

We will explore these in greater detail in Week 10.

It won't be necessary to use anything other than std::vector in COMP6771.

Unordered associative containers

Provide fast retrieval of data based on keys. The keys are hashed.

`std::unordered_set`

`absl::flat_hash_set`

A collection of unique keys.

`std::unordered_map`

`absl::flat_hash_map`

Associative array that map unique keys to a values.

We may explore these in greater detail in Week 10.

The Abseil flat-hash containers offer significant performance benefits over the std:: containers, which is why we use them in COMP6771.

For the purposes of COMP6771, they are interface-compatible.

Associative containers

Provide fast retrieval of data based on keys. The keys are sorted.

`std::set`	A collection of unique keys.
`std::multiset`	A collection of keys.
`std::map`	Associative array that map a unique keys to values.
`std::multimap`	Associative array where one key may map to many values.

We may explore these in greater detail in Week 10.

They are mostly interface-compatible with the unordered associative containers.

String processing

```cpp
// #include <string>

auto const greeting = std::string("hello, world!");
```

User-defined literals (UDLs)

```
// #include <string>

using namespace std::string_literals;
auto const greeting = "hello, world"s;
```

Put *using-directives* in the smallest scope possible

```cpp
// #include <string>

auto main() -> int {
    using namespace std::string_literals;
    auto const greeting = "hello, world"s;
}
```

String concatenation

```cpp
// #include <absl/strings/str_cat.h>
// #include <string>

auto const greeting = absl::StrCat("hello", "world", "!");
```

String formatting

```cpp
// #include <fmt/format.h>
// #include <iostream>
// #include <string>

auto const message = fmt::format("The meaning of life is {}", 42);
std::cout << message << '\n';
```

String formatting

```cpp
// #include <fmt/format.h>
// #include <iostream>
// #include <string>

auto const message = fmt::format("pi has the value {}", 3.1415);
std::cout << message << '\n';
```

String formatting

```cpp
// #include <fmt/format.h>
// #include <iostream>
// #include <string>

auto const message = fmt::format("life={}, pi={}", 42, 3.1415);
std::cout << message << '\n';
```

String formatting

```cpp
// #include <fmt/format.h>
// #include <iostream>
// #include <string>

auto const message = fmt::format("life={}, pi={}", 3.1415, 42);
std::cout << message << '\n';
```

Positional "named" args

```cpp
// #include <fmt/format.h>
// #include <iostream>
// #include <string>

auto const message = fmt::format("life={life}, pi={pi}",
                                 fmt::arg("pi", 3.1415),
                                 fmt::arg("life", 42));
std::cout << message << '\n';
```

std::vector revisited

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

// Querying a vector
CHECK(some_ints[0] == 0);
CHECK(some_ints[1] == 1);
CHECK(some_ints[2] == 2);
CHECK(some_ints[3] == 3);
CHECK(some_ints[4] == 2);
CHECK(some_ints[5] == 5);
```

std::vector grows as we add elements

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.push_back(42);
REQUIRE(ranges::distance(some_ints) == 7);

CHECK(some_ints[6] == 42);
```

std::vector grows as we add elements

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.push_back(42);
REQUIRE(ranges::distance(some_ints) == 7);

CHECK(some_ints[6] == 42);
```

std::vector shrinks as we remove elements

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.pop_back();
REQUIRE(ranges::distance(some_ints) == 5);
CHECK(some_ints.back() == 2);
```

std::vector shrinks as we remove elements

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.pop_back();
REQUIRE(ranges::distance(some_ints) == 5);
CHECK(some_ints.back() == 2);

// erases any occurrence of 2
std::erase(some_ints, 2);
REQUIRE(ranges::distance(some_ints) == 4);
CHECK(some_ints[2] == 3);
```

std::vector shrinks as we remove elements

```
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.clear(); // removes *all* the elements
CHECK(some_ints.empty());

auto const no_elements = std::vector<int>{};
REQUIRE(no_elements.empty());

CHECK(some_elements == no_elements);
```

std::vector shrinks as we remove elements

```cpp
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.clear(); // removes *all* the elements
CHECK(some_ints.empty());

auto const no_elements = std::vector<int>{};
REQUIRE(no_elements.empty());

CHECK(some_elements == no_elements);
```

std::vector shrinks as we remove elements

```
auto some_ints = std::vector<int>{0, 1, 2, 3, 2, 5};
REQUIRE(ranges::distance(some_ints) == 6);

some_ints.clear(); // removes *all* the elements
CHECK(some_ints.empty());

auto const no_elements = std::vector<int>{};
REQUIRE(no_elements.empty());

CHECK(some_elements == no_elements);
```

I want a vector with five zeroes

```
auto all_default = std::vector<double>(5);
REQUIRE(ranges::distance(all_default) == 5);

CHECK(all_default[0] == 0.0);
CHECK(all_default[1] == 0.0);
CHECK(all_default[2] == 0.0);
CHECK(all_default[3] == 0.0);
CHECK(all_default[4] == 0.0);
```

I want a vector with five zeroes

```
auto all_default = std::vector<double>(5);
REQUIRE(ranges::distance(all_default) == 5);

CHECK(all_default[0] == 0.0);
CHECK(all_default[1] == 0.0);
CHECK(all_default[2] == 0.0);
CHECK(all_default[3] == 0.0);
CHECK(all_default[4] == 0.0);
```

I want a vector with three identical values

```cpp
auto const initial_value = std::string("some words go here!");
auto all_same = std::vector<std::string>(3, initial_value);
REQUIRE(ranges::distance(all_same) == 3);

CHECK(all_same[0] == initial_value);
CHECK(all_same[1] == initial_value);
CHECK(all_same[2] == initial_value);

all_same[1] = "other words";
CHECK(all_same[0] != all_same[1]);
CHECK(all_same.front() == all_same.back());
```

I want a vector with three identical values

```
auto const initial_value = std::string("some words go here!");
auto all_same = std::vector<std::string>(3, initial_value);
REQUIRE(ranges::distance(all_same) == 3);

CHECK(all_same[0] == initial_value);
CHECK(all_same[1] == initial_value);
CHECK(all_same[2] == initial_value);

all_same[1] = "other words";
CHECK(all_same[0] != all_same[1]);
CHECK(all_same.front() == all_same.back());
```

A card game

```cpp
enum class colour { red, green, blue, yellow };
enum class value { number, draw_two, draw_four, reverse, skip };

struct card {
  colour colour;
  value value;

  friend auto operator==(card, card) -> bool = default;
};
```

A card game

```
enum class colour { red, green, blue, yellow };
enum class value { number, draw_two, draw_four, reverse, skip };

struct card {
  colour colour;
  value value;

  friend auto operator==(card, card) -> bool = default;
};
```

A card game

```
auto const red_number = card{colour::red, value::number};
auto const blue_number = card{colour::blue, value::number};
auto const green_draw_two = card{colour::green, value::draw_two};
auto const blue_skip = card{colour::blue, value::skip};
auto const yellow_draw_four = card{colour::yellow, value::draw_four};
```

Stacks

```
// #include <stack>

auto deck = std::stack<card>();
REQUIRE(deck.empty());

deck.push(red_number);
deck.push(green_draw_two);
deck.push(green_draw_two);
deck.push(yellow_draw_four);
deck.push(blue_number);

REQUIRE(deck.size() == 5);
```

Stacks

```cpp
// #include <stack>

auto deck = std::stack<card>();
REQUIRE(deck.empty());

deck.push(red_number);
deck.push(green_draw_two);
deck.push(green_draw_two);
deck.push(yellow_draw_four);
deck.push(blue_number);

REQUIRE(deck.size() == 5);
```

Stacks

```cpp
// #include <stack>

auto deck = std::stack<card>();
REQUIRE(deck.empty());

deck.push(red_number);
deck.push(green_draw_two);
deck.push(green_draw_two);
deck.push(yellow_draw_four);
deck.push(blue_number);

REQUIRE(deck.size() == 5);
```

Removing elements from a stack

```
// #include <stack>

CHECK(deck.top() == blue_number);
deck.pop();

CHECK(deck.top() == yellow_draw_four);
deck.pop();

// ...
```

Removing elements from a stack

```cpp
// #include <stack>

CHECK(deck.top() == blue_number);
deck.pop();

CHECK(deck.top() == yellow_draw_four);
deck.pop();

// ...
```

Removing elements from a stack

```
// #include <stack>

CHECK(deck.top() == blue_number);
deck.pop();

CHECK(deck.top() == yellow_draw_four);
deck.pop();

// ...
```

Comparing two stacks

```cpp
auto const more_cards = deck;
REQUIRE(more_cards == deck);

deck.pop();
CHECK(more_cards != deck);
```

Comparing two stacks

```cpp
auto const more_cards = deck;
REQUIRE(more_cards == deck);

deck.pop();
CHECK(more_cards != deck);
```

Queues

```cpp
// #include <queue>
auto deck = std::queue<card>();
REQUIRE(deck.empty());

deck.push(red_number);
deck.push(green_draw_two);
deck.push(green_draw_two);
deck.push(yellow_draw_four);
deck.push(blue_number);
```

Queues

```
// #include <queue>
auto deck = std::queue<card>();
REQUIRE(deck.empty());

deck.push(red_number);
deck.push(green_draw_two);
deck.push(green_draw_two);
deck.push(yellow_draw_four);
deck.push(blue_number);
```

Removing elements from a queue

```
// #include <stack>

CHECK(deck.front() == red_number);
deck.pop();

CHECK(deck.front() == green_draw_two);
deck.pop();

// ...
```

Removing elements from a queue

```
// #include <stack>

CHECK(deck.front() == red_number);
deck.pop();

CHECK(deck.front() == green_draw_two);
deck.pop();

// ...
```

Comparing two queues

```
auto const more_cards = deck;
REQUIRE(more_cards == deck);

deck.pop();
CHECK(more_cards != deck);
```

Comparing two queues

```
auto const more_cards = deck;
REQUIRE(more_cards == deck);

deck.pop();
CHECK(more_cards != deck);
```

A range is an ordered sequence of elements with a designated start and rule for finishing

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

```
std::vector<std::string>{"Hello", "world!"}
```

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

```
std::vector<std::string>{"Hello", "world!"}
```

$\mathbb{N}$ $\mathbb{Z}^+$ $\mathbb{Q}^+$ $\mathbb{R}^+$

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

```
std::vector<std::string>{"Hello", "world!"}
```

$\mathbb{N}$ $\mathbb{Z}^+$ $\mathbb{Q}^+$ $\mathbb{R}^+$

Exercise: how can $\mathbb{C}$ be made into a range?

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

```
std::vector<std::string>{"Hello", "world!"}
```

$\mathbb{N}$ $\mathbb{Z}^+$ $\mathbb{Q}^+$ $\mathbb{R}^+$

Exercise: how can $\mathbb{C}$ be made into a range?

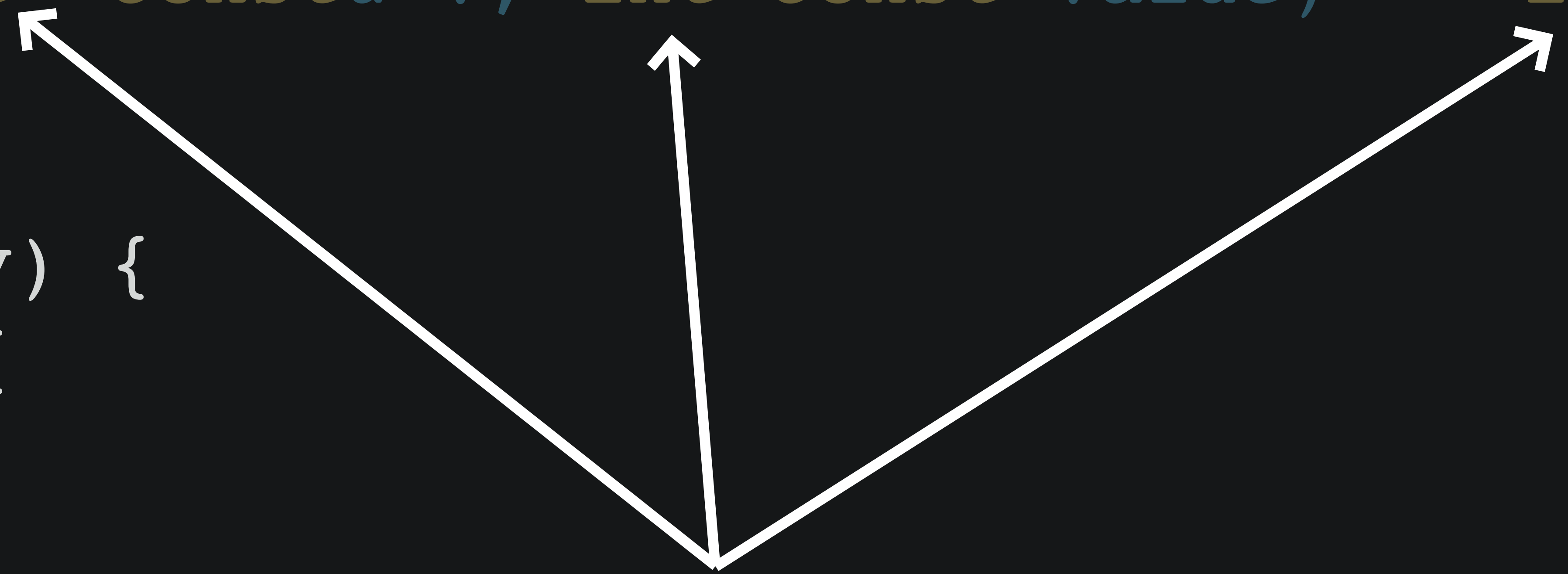
```
for (auto i = 0; std::cin >> i;) { ... }
```

Find revisited

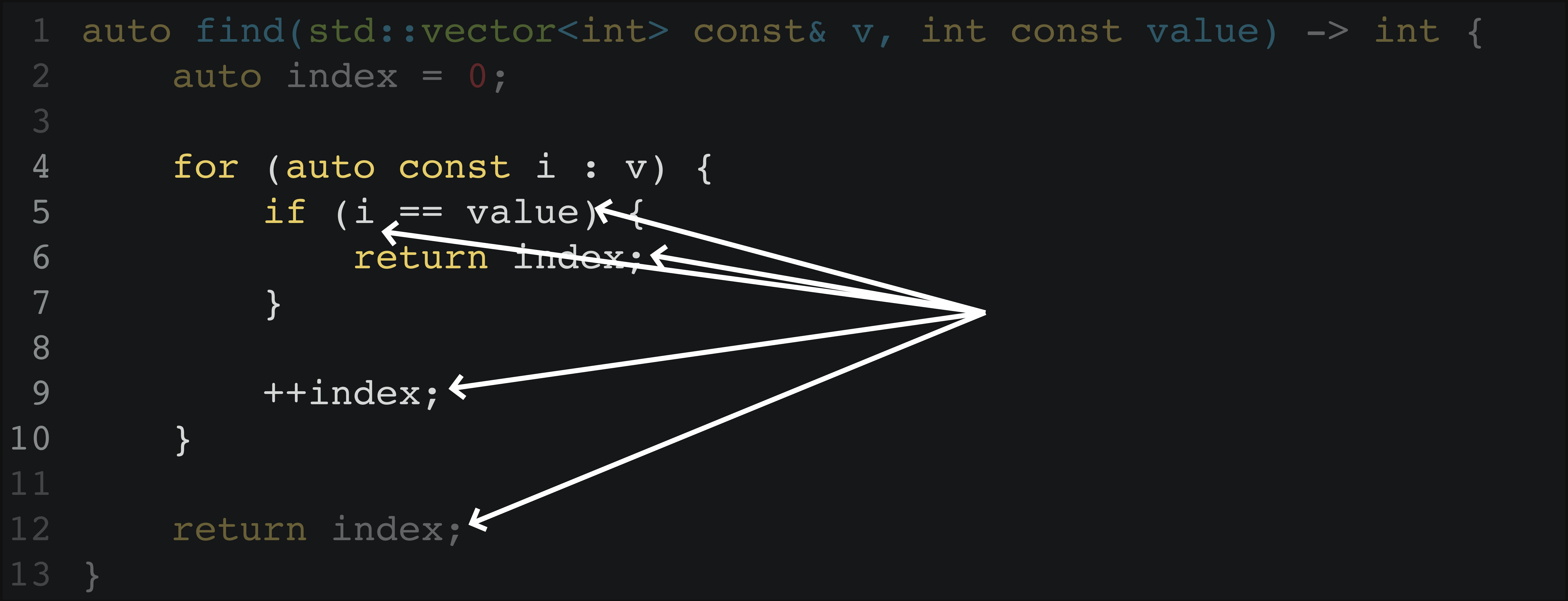
```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;

    for (auto const i : v) {
        if (i == value) {
            return index;
        }

        ++index;
    }

    return index;
}
```

Find revisited

```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;

    for (auto const i : v) {
        if (i == value) {
            return index;
        }

        ++index;
    }

    return index;
}
```



Find revisited



```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;

    for (auto const i : v) {
        if (i == value) {
            return index;
        }

        ++index;
    }

    return index;
}
```

What type should the index become?

```cpp
// Note: ??? is not a valid C++ symbol
auto find(std::vector<int> const& v, int const value) -> ??? {
    auto index = ???;

    for (auto const i : v) {
        if (i == value) {
            return index;
        }

        ++index;
    }

    return index;
}
```

A Java linked list

```java
class Node {
    Node next;
    public int value;
}

public class LinkedList {
    private Node head;

    public Node find(final int value) {
        Node i = head;
        while (i != null) {
            if (i.value == value) {
                return i;
            }
            i = i.next;
        }

        return null;
    }
}
```

Compare the pair

```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;
    for (auto const i : v) {
        if (i == value) {
            return index;
        }
        ++index;
    }
    return index;
}
```

```java
public Node find(final int value) {
    Node i = head;
    while (i != null) {
        if (i.value == value) {
            return i;
        }
        i = i.next;
    }
    return null;
}
```

Compare the pair

```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;
    for (auto const i : v) {
        if (i == value) {
            return index;
        }
        ++index;
    }
    return index;
}
```

```java
public Node find(final int value) {
    Node i = head;
    while (i != null) {
        if (i.value == value) {
            return i;
        }
        i = i.next;
    }
    return null;
}
```

Compare the pair

```
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;
    for (auto const i : v) {
        if (i == value) {
            return index;
        }
        ++index;
    }
    return index;
}
```

```
public Node find(final int value) {
    Node i = head;
    while (i != null) {
        if (i.value == value) {
            return i;
        }
        i = i.next;
    }
    return null;
}
```

Compare the pair

```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;
    for (auto const i : v) {
        if (i == value) {
            return index;
        }
        ++index;
    }
    return index;
}
```

```java
public Node find(final int value) {
    Node i = head;
    while (i != null) {
        if (i.value == value) {
            return i;
        }
        i = i.next;
    }
    return null;
}
```

Compare the pair

```cpp
auto find(std::vector<int> const& v, int const value) -> int {
    auto index = 0;
    for (auto const i : v) {
        if (i == value) {
            return index;
        }
        ++index;
    }
    return index;
}
```

```java
public Node find(final int value) {
    Node i = head;
    while (i != null) {
        if (i.value == value) {
            return i;
        }
        i = i.next;
    }
    return null;
}
```

Could you imagine having to do that for...

...std::string?

...a doubly-linked list?

...a double-ended queue?

...a static vector?

...a skip list?

...a cord?

x sequence containers

×

y sequence algorithms (e.g. find)

≈

xy algorithm implementations

7 sequence containers

×

≈

7 sequence containers

×

110 sequence algorithms (e.g. find)

≈

7 sequence containers

×

110 sequence algorithms (e.g. find)

≈

770 algorithm implementations

7 sequence containers

×

110 sequence algorithms (e.g. find)

≈

770 algorithm implementations

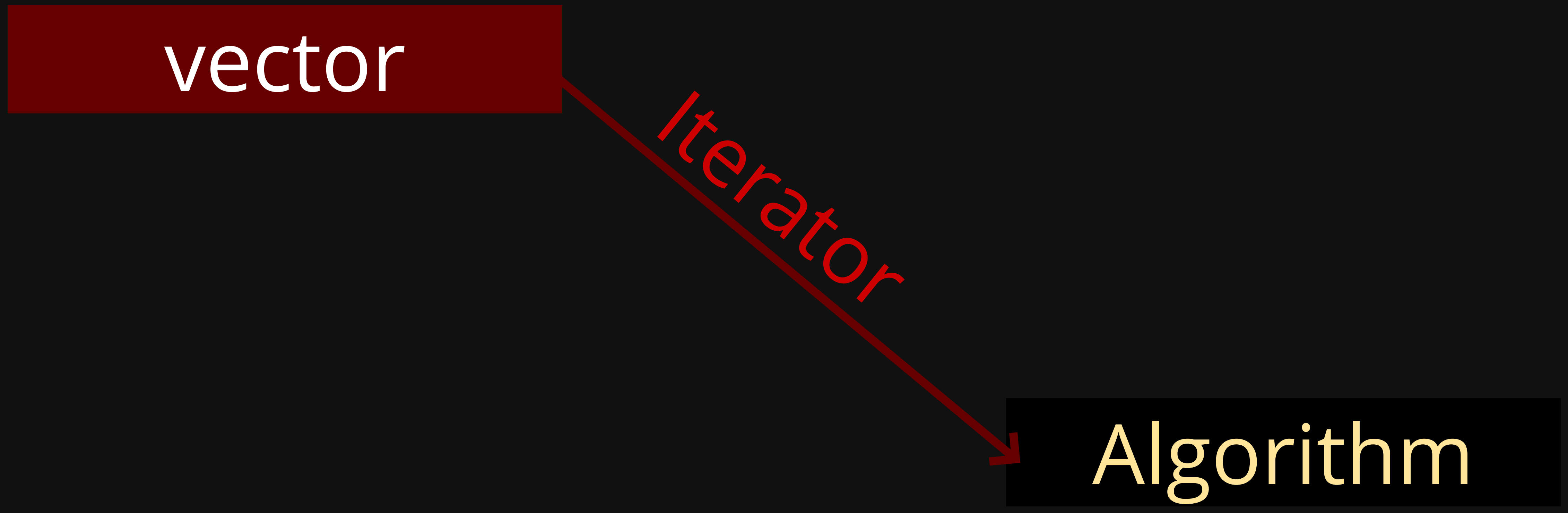
and we haven't come close to exhausting either group

We need an intermediate representation

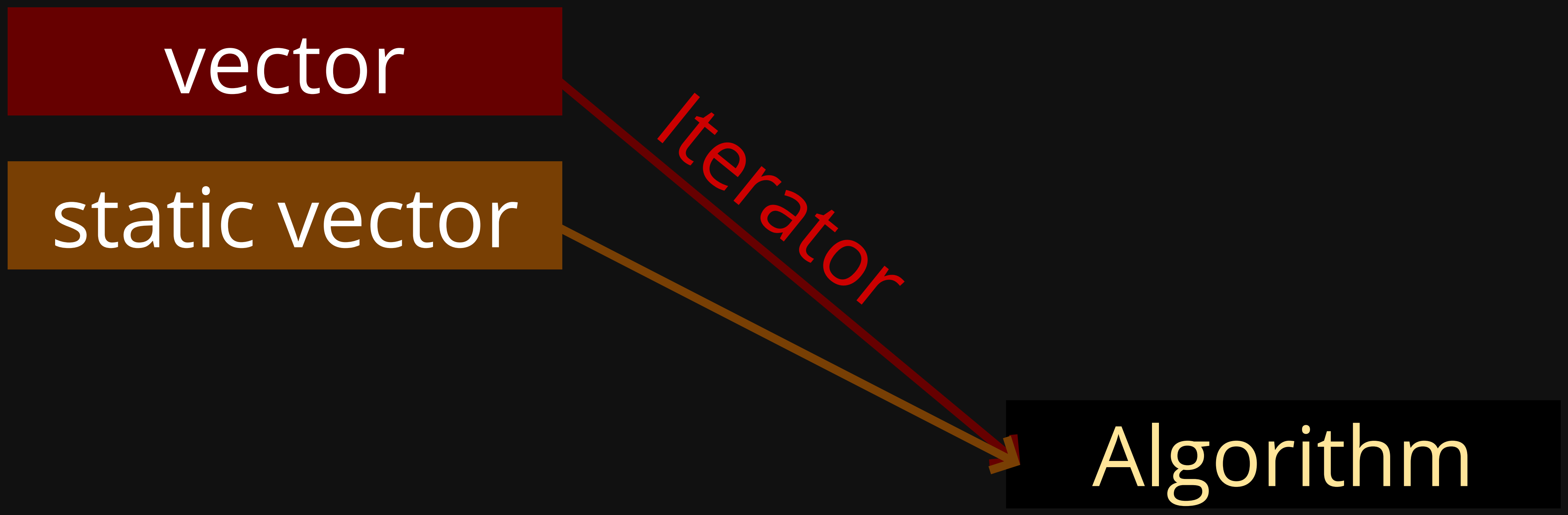
We need an intermediate representation

Algorithm

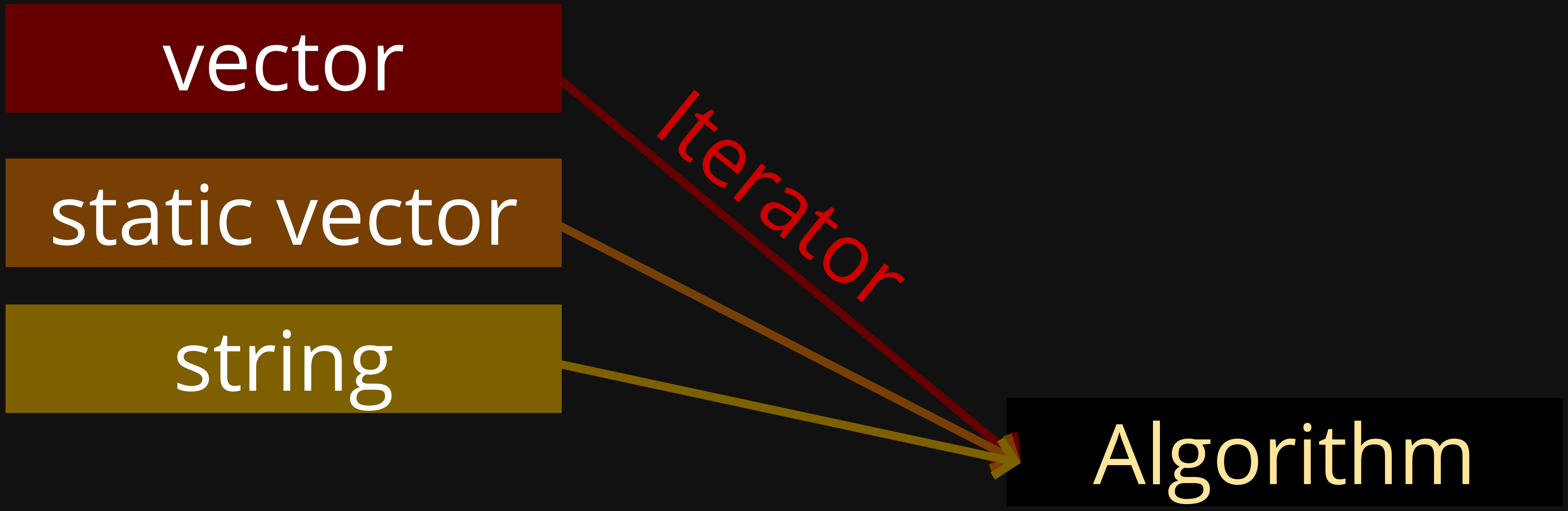
We need an intermediate representation



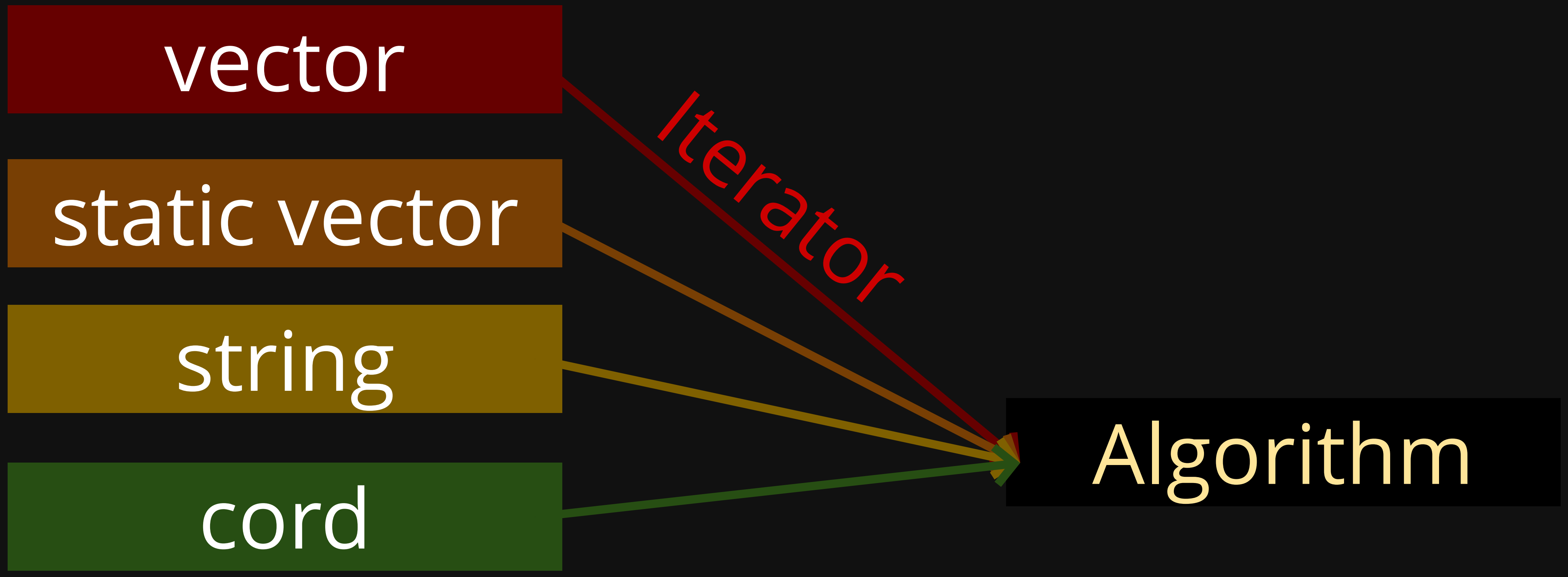
We need an intermediate representation



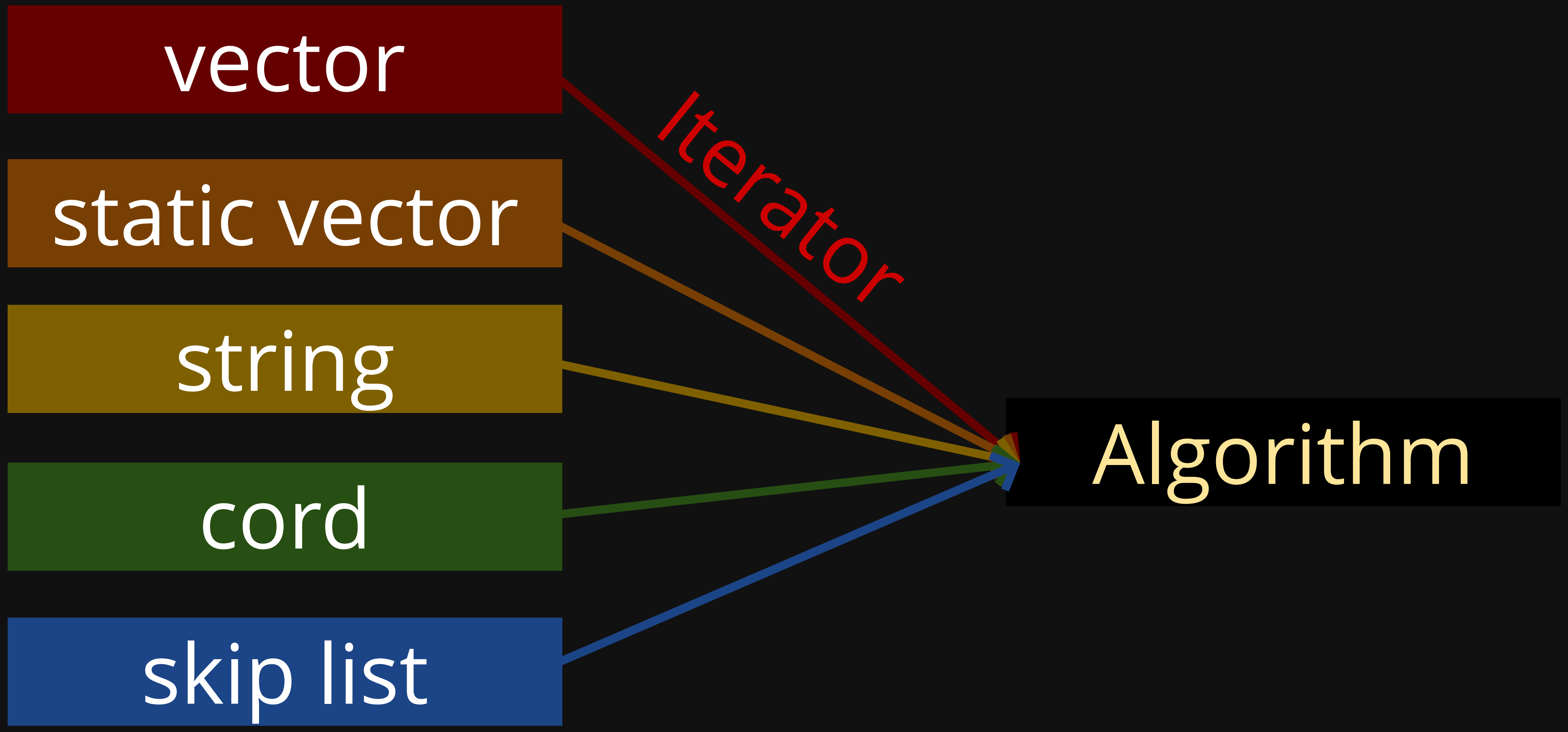
We need an intermediate representation



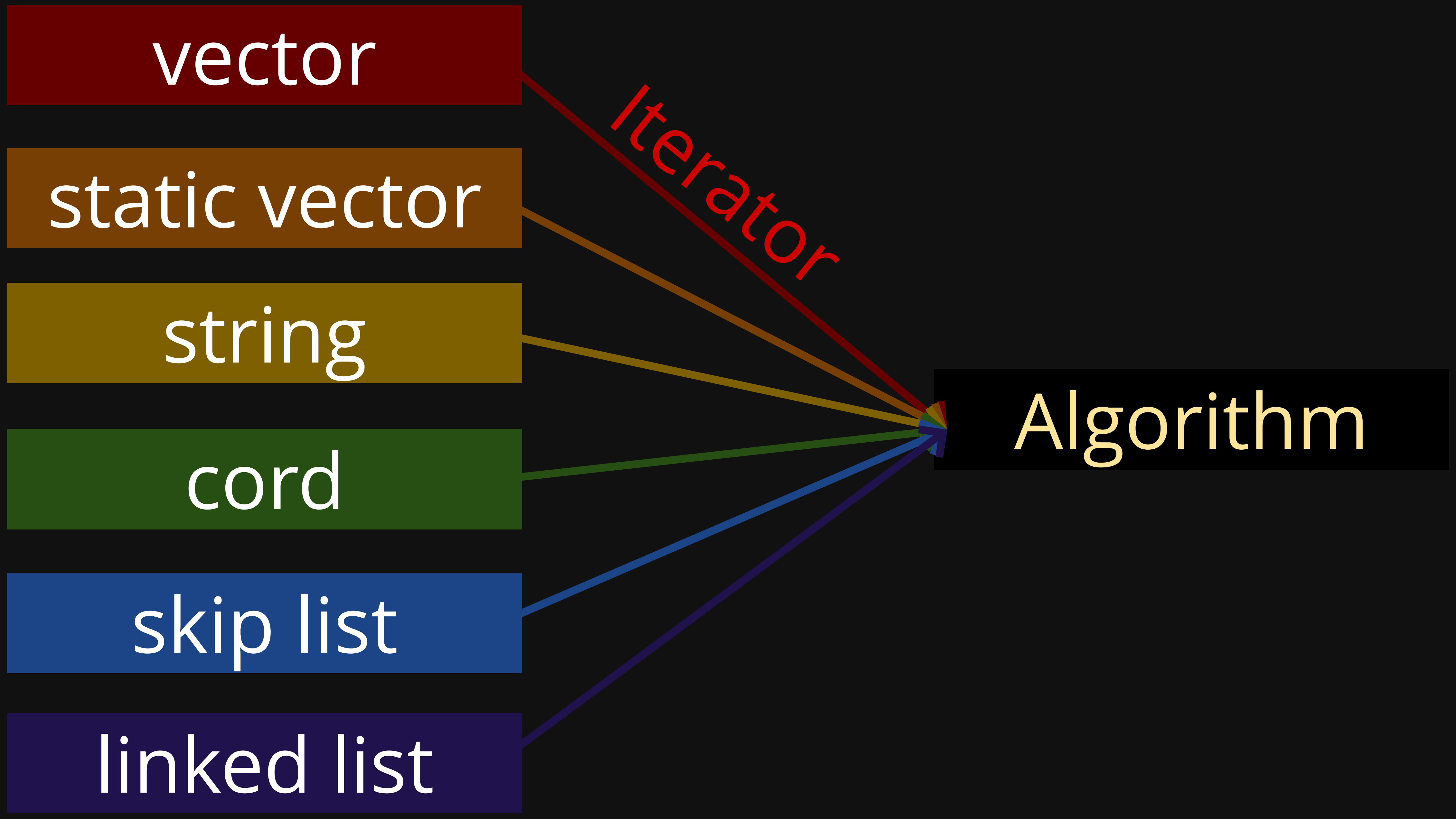
We need an intermediate representation



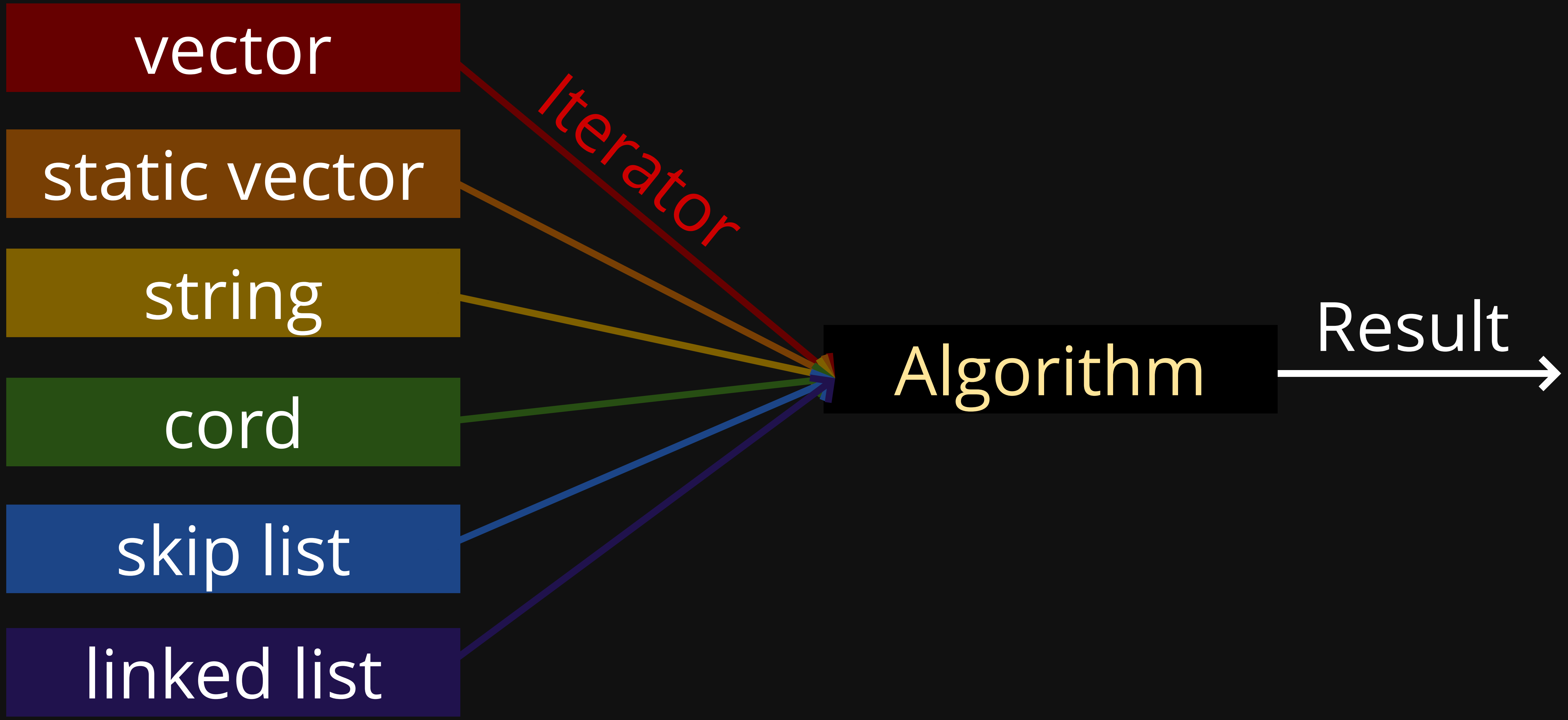
We need an intermediate representation



We need an intermediate representation

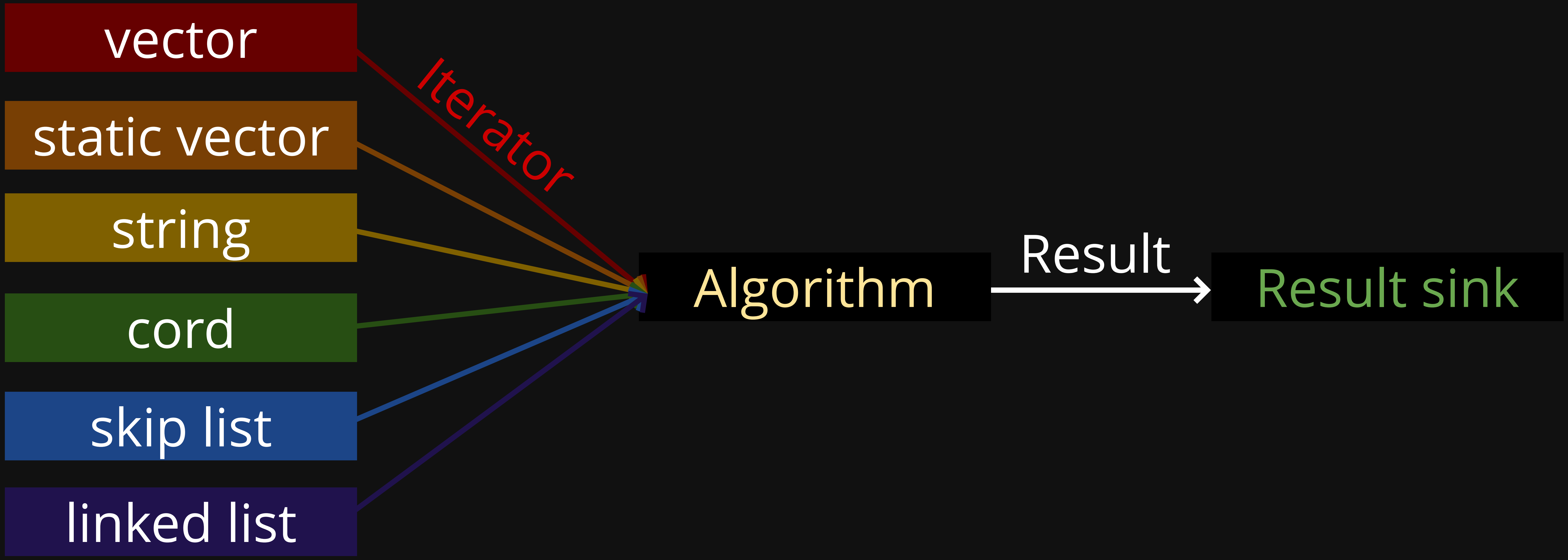


We need an intermediate representation



The result may be a one or more iterators, a scalar value, or some combination of both.

We need an intermediate representation



The result may be a one or more iterators, a scalar value, or some combination of both.

x sequence containers

+

y sequence algorithms (e.g. find)

≈

$x + y$ total implementations

7 sequence containers

+

≈

7 sequence containers

+

110 sequence algorithms (e.g. find)

≈

7 sequence containers

+

110 sequence algorithms (e.g. find)

≈

117 total implementations

7 sequence containers

+

110 sequence algorithms (e.g. find)

≈

117 total implementations

and we haven't come close to exhausting either group

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*unspecified*

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`

Successor	`j = i + n < ranges::distance(v)`
	`  ? i + n`
	`  : ranges::distance(v);`
		`j = i->successor(n)`	`ranges::next(i, s, n)`

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	gsl_lite::index	node*	*unspecified*
Read element	v[i]	i->value	*i

Successor	j = i + n < ranges::distance(v)
	? i + n
	: ranges::distance(v);
		j = i->successor(n)	ranges::next(i, s, n)

Advance fwd	++i	i = i->next	++i

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`

Successor	`j = i + n < ranges::distance(v)`
	`? i + n`
	`: ranges::distance(v);`
		`j = i->successor(n)`	`ranges::next(i, s, n)`

Predecessor	`j = i - n < 0 ? 0 : i - n`	`j = i->predecessor(n)`	`ranges::prev(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*unspecified*
Read element	`v[i]`	`i->value`	`*i`

Successor	`j = i + n < ranges::distance(v)`
	`? i + n`
	`: ranges::distance(v);`
		`j = i->successor(n)`	`ranges::next(i, s, n)`

Predecessor	`j = i - n < 0 ? 0 : i - n`	`j = i->predecessor(n)`	`ranges::prev(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`
Advance back	`--i`	`i = i->prev`	`--i`

Iterators: our intermediate representation

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*unspecified*
Read element	`v[i]`	`i->value`	`*i`

Successor	`j = i + n < ranges::distance(v)`
	`? i + n`
	`: ranges::distance(v);`
		`j = i->successor(n)`	`ranges::next(i, s, n)`

Predecessor	`j = i - n < 0 ? 0 : i - n`	`j = i->predecessor(n)`	`ranges::prev(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`
Advance back	`--i`	`i = i->prev`	`--i`
Comparison	`i < ranges::distance(v)`	`i != nullptr`	`i != s`

Generating a hand of cards

```cpp
auto hand = std::vector<card>{
    red_number,
    blue_number,
    green_draw_two,
    blue_number,
    blue_skip,
    yellow_draw_four,
    blue_number,
    blue_number,
    blue_skip,
};
```

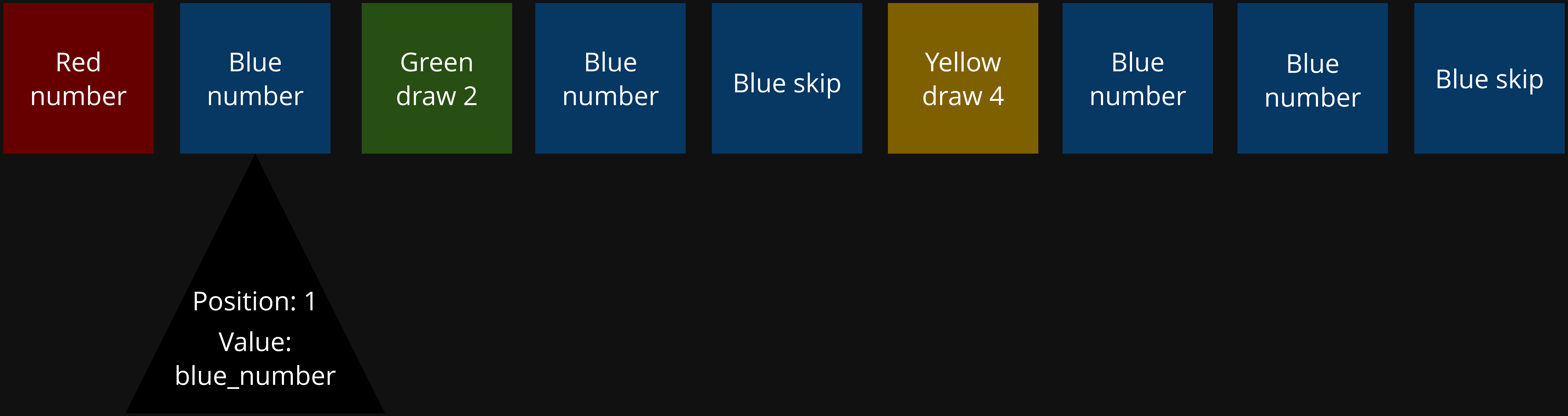
Counting cards

```cpp
// #include <range/v3/algorithm.hpp>

CHECK(ranges::count(hand, red_number) == 1);
CHECK(ranges::count(hand, blue_number) == 4);
CHECK(ranges::count(hand, blue_skip) == 2);
```

Finding a card

```cpp
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);
```

What is card_to_play?



Red number	Blue number	Green draw 2	Blue number	Blue skip	Yellow draw 4	Blue number	Blue number	Blue skip

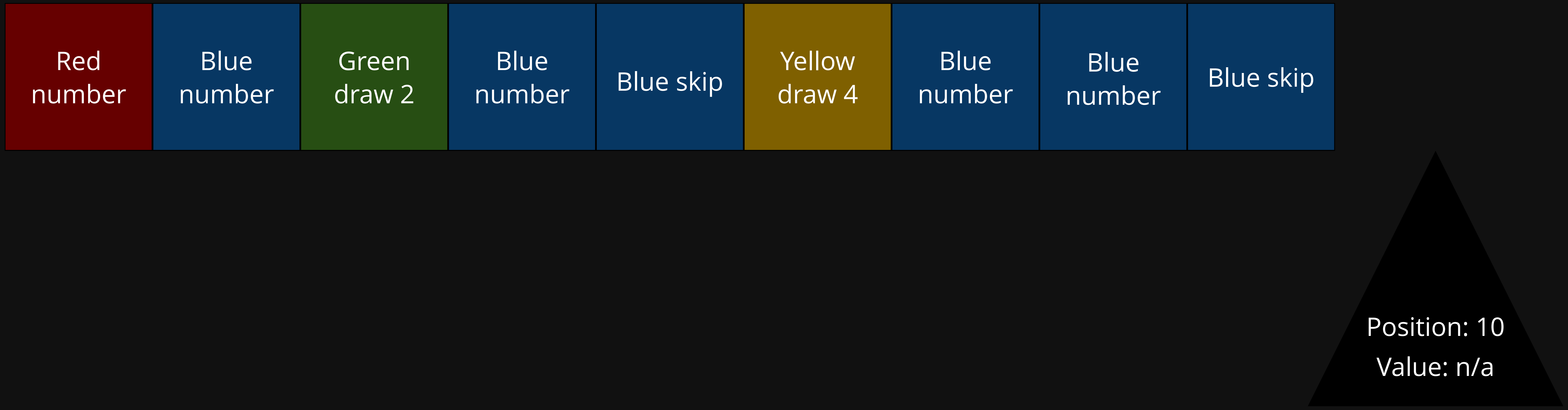
Position: 1

Value: blue_number

Finding a card

```cpp
// #include <range/v3/algorithm.hpp>

auto const green_draw_four = card{colour::green, value::draw_four};
auto card_to_play = ranges::find(hand, green_draw_four);

REQUIRE(card_to_play == hand.cend());
```

What is card_to_play?



Red number	Blue number	Green draw 2	Blue number	Blue skip	Yellow draw 4	Blue number	Blue number	Blue skip

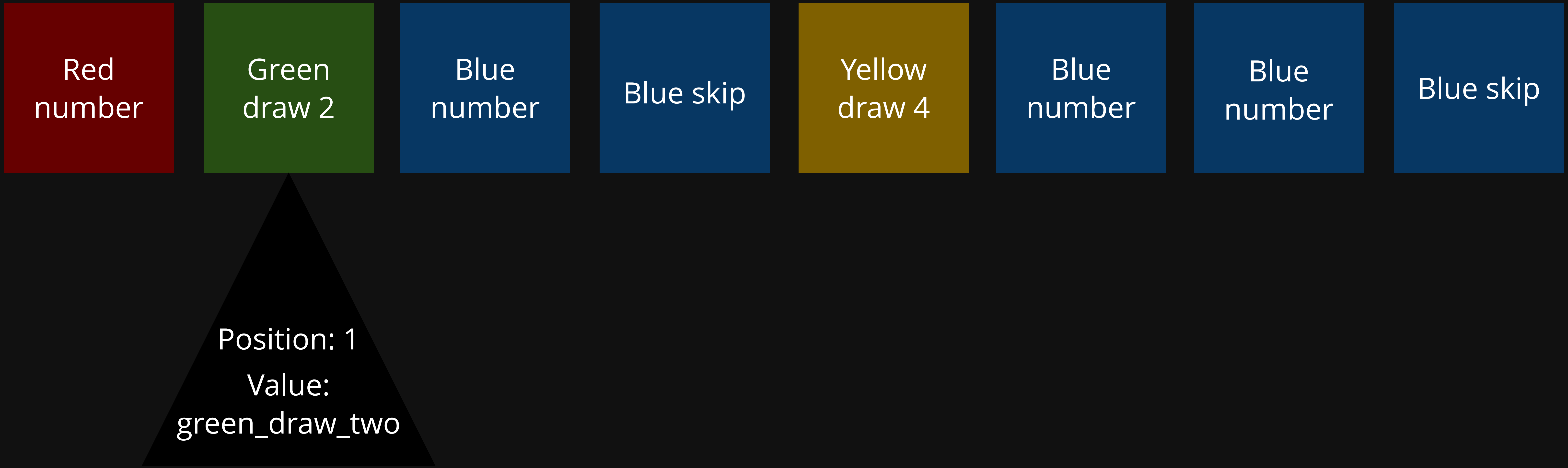
Position: 10

Value: n/a

Erasing a single, specific, card

```
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);

card_to_play = hand.erase(card_to_play);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play = green_draw_two);
```

What is card_to_play?



Red number

Green draw 2

Blue number

Blue skip

Yellow draw 4

Blue number

Blue number

Blue skip

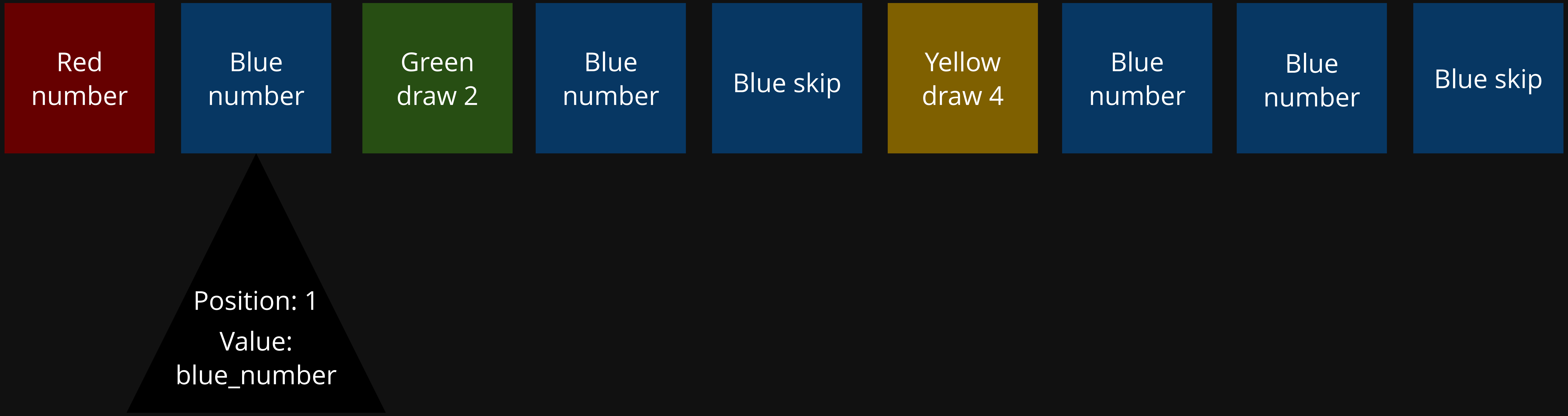
Position: 1

Value: green_draw_two

Adding elements and iterators

```
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);

hand.push_back(green_draw_two);
```

What is card_to_play?



Red number

Blue number

Green draw 2

Blue number

Blue skip

Yellow draw 4

Blue number

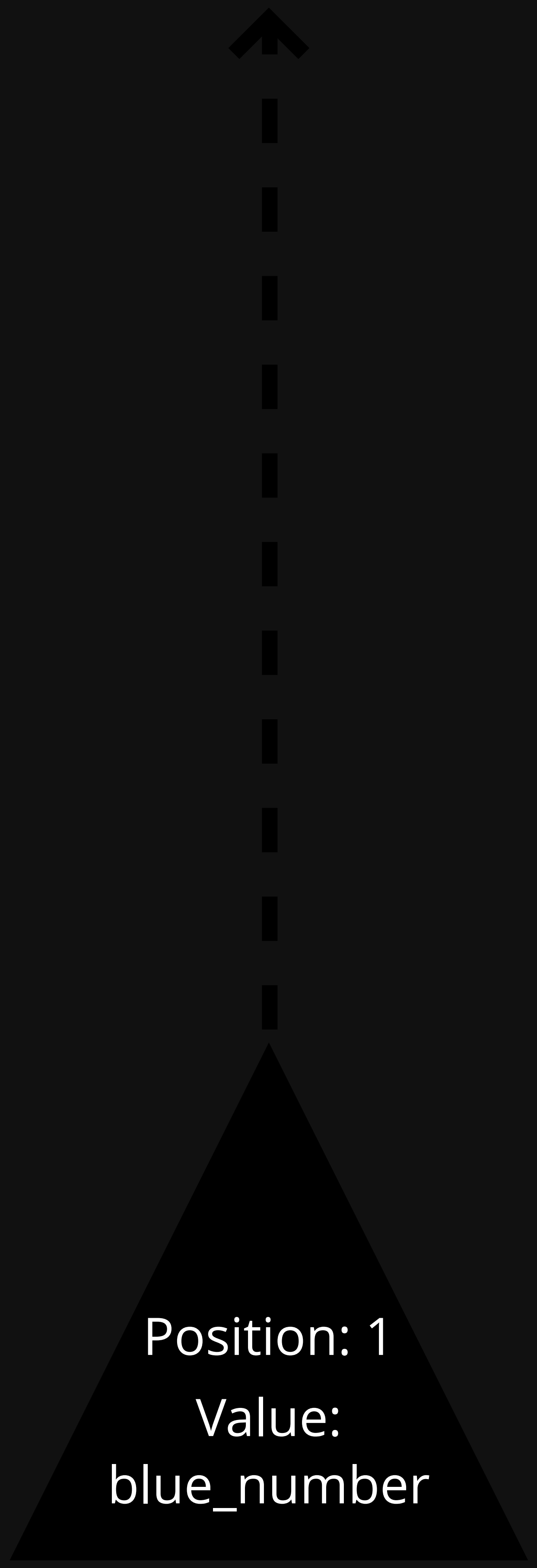
Blue number

Blue skip

Position: 1

Value: blue_number

What is card_to_play?



Position: 1

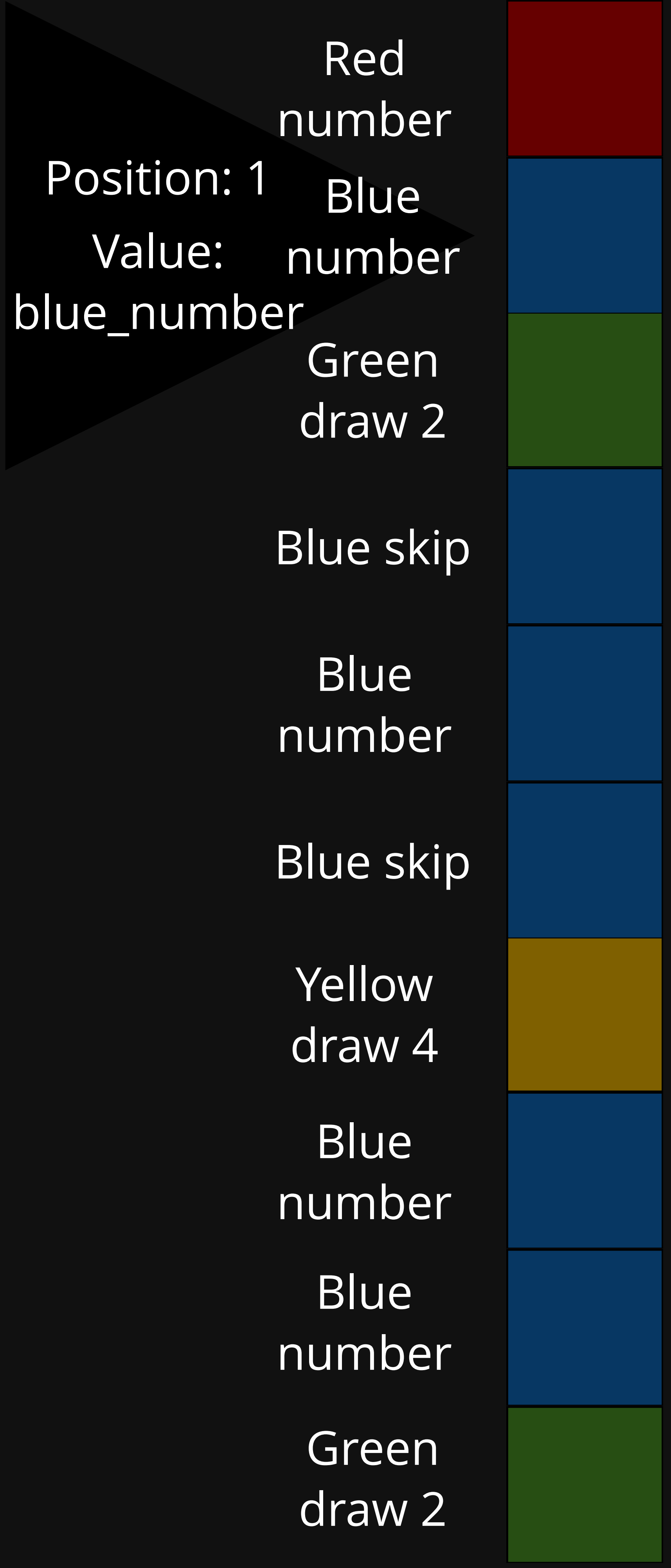
Value: blue_number

Red number	🟥
Blue number	🟦
Green draw 2	🟩
Blue skip	🟦
Blue number	🟦
Blue skip	🟦
Yellow draw 4	🟨
Blue number	🟦
Blue number	🟦
Green draw 2	🟩

What is card_to_play?

```
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);

hand.push_back(green_draw_two);

card_to_play = ranges::find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);
```

What is card_to_play?



Position: 1

Value: blue_number

Red number

Blue number

Green draw 2

Blue skip

Blue number

Blue skip

Yellow draw 4

Blue number

Blue number

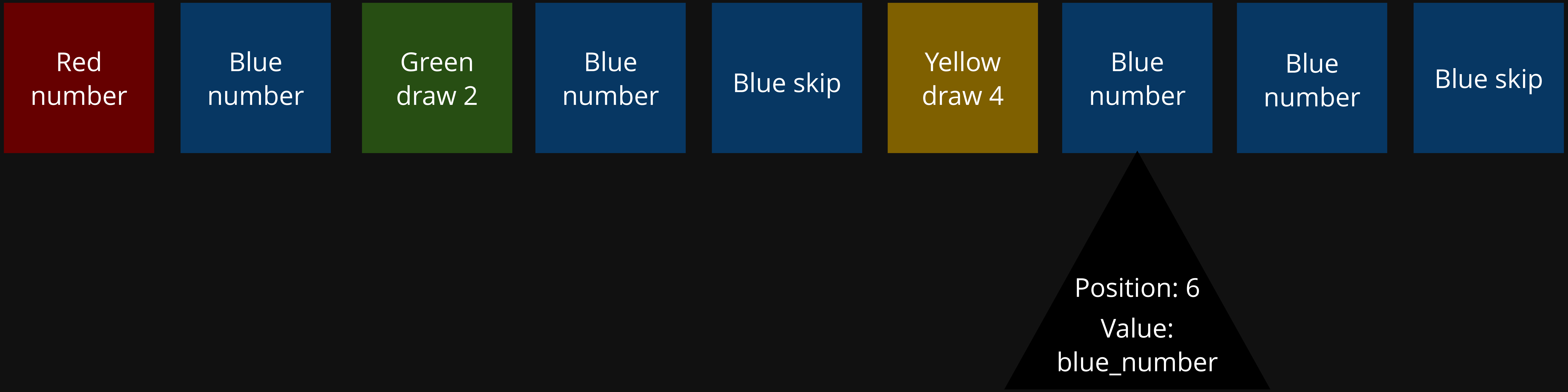
Green draw 2

Finding two adjacent cards that are the same

```
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::adjacent_find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);
```

Finding two adjacent cards that are the same

```
// #include <range/v3/algorithm.hpp>

auto card_to_play = ranges::adjacent_find(hand, blue_number);
REQUIRE(card_to_play != hand.cend());
CHECK(*card_to_play == blue_number);
```



Red number

Blue number

Green draw 2

Blue number

Blue skip

Yellow draw 4

Blue number

Blue number

Blue skip

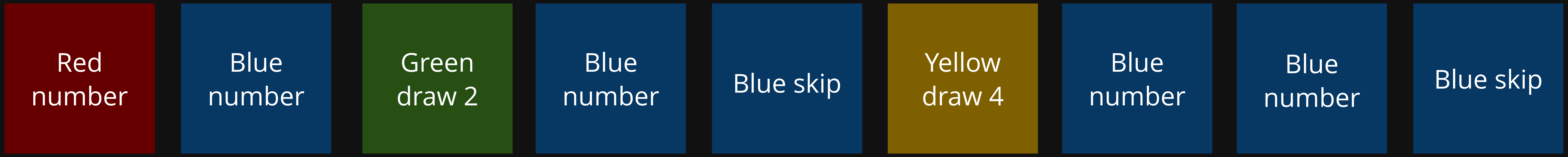
Position: 6

Value: blue_number

Lambda expressions

How many blue cards are there?

```cpp
// #include <range/v3/algorithm.hpp>

auto const blue_cards = ranges::count_if(hand, [](card const c) {
    return c.colour == colour::blue;
});

auto const expected_blue_cards = 6;
CHECK(blue_cards == expected_blue_cards);
```



Red number

Blue number

Green draw 2

Blue number

Blue skip

Yellow draw 4

Blue number

Blue number

Blue skip

Lambda unary predicate

```
[](card const c) {
    return c.colour == colour::blue;
}
```

Explicit return type

```
[](card const c) -> bool {
    return c.colour == colour::blue;
}
```

We'll need this to do a binary search

```cpp
#include <compare>

enum class colour { red, green, blue, yellow };
enum class value { number, draw_two, draw_four, reverse, skip };

struct card {
  colour colour;
  value value;

  friend auto operator==(card, card) -> bool = default;
  friend auto operator<=>(card, card) = default;
};
```

Surprise binary search

```cpp
// #include <range/v3/algorithm.hpp>

ranges::sort(hand);
REQUIRE(ranges::is_sorted(hand));

auto [first, last] = ranges::equal_range(hand, blue_number);
REQUIRE(first != last);
CHECK(ranges::distance(first, last) == 4);

CHECK(ranges::all_of(first, last, [blue_number](card const x) {
    return x == blue_number;
}));
```

Surprise binary search

```cpp
// #include <range/v3/algorithm.hpp>

ranges::sort(hand);
REQUIRE(ranges::is_sorted(hand));

auto [first, last] = ranges::equal_range(hand, blue_number);
REQUIRE(first != last);
CHECK(ranges::distance(first, last) == 4);

CHECK(ranges::all_of(first, last, [blue_number](card const x) {
    return x == blue_number;
}));
```

Surprise binary search

```cpp
// #include <range/v3/algorithm.hpp>

ranges::sort(hand);
REQUIRE(ranges::is_sorted(hand));

auto [first, last] = ranges::equal_range(hand, blue_number);
REQUIRE(first != last);
CHECK(ranges::distance(first, last) == 4);

CHECK(ranges::all_of(first, last, [blue_number](card const x) {
    return x == blue_number;
}));
```

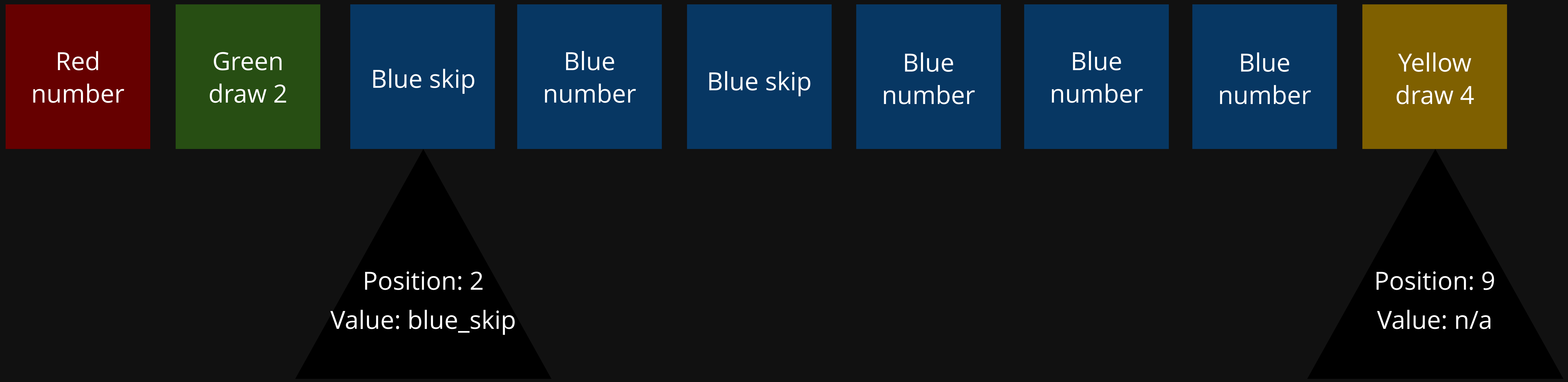
Surprise binary search

```
// #include <range/v3/algorithm.hpp>

ranges::sort(hand);
REQUIRE(ranges::is_sorted(hand));

auto [first, last] = ranges::equal_range(hand, blue_number);
REQUIRE(first != last);
CHECK(ranges::distance(first, last) == 4);

CHECK(ranges::all_of(first, last, [blue_number](card const x) {
    return x == blue_number;
}));
```

Binary search

Red number	Green draw 2	Blue skip	Blue number	Blue skip	Blue number	Blue number	Blue number	Yellow draw 4

Binary search



Red number

Green draw 2

Blue skip

Blue number

Blue skip

Blue number

Blue number

Blue number

Yellow draw 4

Position: 2
Value: blue_skip

Position: 9
Value: n/a

Lambda with value capture

```
[blue_number](card const x) {
    return x == blue_number;
}
```

Closures

```cpp
auto const blue_then_yellow = [](card const x, card const y) {
    return x.colour == colour::blue and y.colour == colour::yellow;
};

auto const blue_card = ranges::adjacent_find(hand, blue_then_yellow);
REQUIRE(blue_card != hand.end());
CHECK(*blue_card == blue_skip);

auto const yellow_card = ranges::next(blue_card);
CHECK(*yellow_card == yellow_draw_four);
```

Closures

```
auto const blue_then_yellow = [](card const x, card const y) {
    return x.colour == colour::blue and y.colour == colour::yellow;
};

auto const blue_card = ranges::adjacent_find(hand, blue_then_yellow);
REQUIRE(blue_card != hand.end());
CHECK(*blue_card == blue_skip);

auto const yellow_card = ranges::next(blue_card);
CHECK(*yellow_card == yellow_draw_four);
```

Closures

```
auto const blue_then_yellow = [](card const x, card const y) {
    return x.colour == colour::blue and y.colour == colour::yellow;
};

auto const blue_card = ranges::adjacent_find(hand, blue_then_yellow);
REQUIRE(blue_card != hand.end());
CHECK(*blue_card == blue_skip);

auto const yellow_card = ranges::next(blue_card);
CHECK(*yellow_card == yellow_draw_four);
```

Let's take a note of how many swaps we do

```cpp
auto note_swaps(std::map<card, int>& cards_swapped,
                card const c) -> void {
        auto result = cards_swapped.find(c);
        if (result == cards_swapped.end()) {
                cards_swapped.emplace(c, 1);
                return;
        }

        ++result->second;
}
```

With house rules

```
// #include <range/v3/algorithm.hpp>

// house rule: two players can swap a card of the same value
// (but for a different colour)
auto cards_swapped = std::map<card, int>{};
ranges::transform(hand, hand.begin(), [&cards_swapped](card const c) {
    if (c.colour != colour::blue) {
        return c;
    }

    note_swaps(cards_swapped, c);
    return card{colour::green, c.value};
});

CHECK(ranges::none_of(hand, [](card const c) {
    return c.colour == colour::blue;
}));
```

With house rules

```cpp
// #include <range/v3/algorithm.hpp>

// house rule: two players can swap a card of the same value
// (but for a different colour)
auto cards_swapped = std::map<card, int>{};
ranges::transform(hand, hand.begin(), [&cards_swapped](card const c) {
    if (c.colour != colour::blue) {
        return c;
    }

    note_swaps(cards_swapped, c);
    return card{colour::green, c.value};
});

CHECK(ranges::none_of(hand, [](card const c) {
    return c.colour == colour::blue;
}));
```

Capturing by reference

```
[&cards_swapped](card const c) {
    // ...
}
```

Finishing off the example

```
{
    REQUIRE(cards_swapped.contains(blue_number));
    CHECK(cards_swapped.at(blue_number) == 4);
    auto const green_number = card{colour::green, value::number};
    CHECK(ranges::count(hand, green_number) == 4);
}
{
    REQUIRE(cards_swapped.contains(blue_skip));
    CHECK(cards_swapped.at(blue_skip) == 2);
    auto const green_skip = card{colour::green, value::skip};
    CHECK(ranges::count(hand, green_skip) == 2);
}
```

Finishing off the example

```
{
    REQUIRE(cards_swapped.contains(blue_number));
    CHECK(cards_swapped.at(blue_number) == 4);
    auto const green_number = card{colour::green, value::number};
    CHECK(ranges::count(hand, green_number) == 4);
}
{
    REQUIRE(cards_swapped.contains(blue_skip));
    CHECK(cards_swapped.at(blue_skip) == 2);
    auto const green_skip = card{colour::green, value::skip};
    CHECK(ranges::count(hand, green_skip) == 2);
}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::equal_to{}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::equal_to{}
```

is roughly equivalent to

```
[](auto const& x, auto const& y) {
    return x == y;
}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::not_equal_to{}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::not_equal_to{}
```

is roughly equivalent to

```
[](auto const& x, auto const& y) {
    return x != y;
}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::plus{}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::plus{}
```

is roughly equivalent to

```
[](auto const& x, auto const& y) {
    return x + y;
}
```

Library function objects

```
// #include <range/v3/functional.hpp>
ranges::multiplies{}
```

Library function objects

```cpp
// #include <range/v3/functional.hpp>
ranges::multiplies{}
```

is roughly equivalent to

```cpp
[](auto const& x, auto const& y) {
    return x * y;
}
```

ranges::distance vs vector::size

We usually want to use ranges::distance because its return type is implicitly compatible with int.

The vector/string interface uses a different type with different characteristics, and we don't want to mix them up. The compiler helps us with this.

You can use size for those parts of the interface, if you keep the scopes small.

```cpp
// E.g. 1
auto v = std::vector<int>(other.size());

// E.g. 2 (yuck, but best option till you get more experience)
for (auto i = 0; i < ranges::distance(v); ++i) {
    using size_type = std::vector<int>::size_type; // C++ typedef
    v[gsl_lite::narrow_cast<size_type>(i)];
}

// E.g. 3 i should not leave the scope of the loop
for (auto i = std::vector<int>::size_type{0}; i < v.size(); ++i) {
    v[i];
}
```

Constructing a vector of one type from a vector of another type

```
auto standard_deviation_distribution() -> std::vector<double>;
static_cast<std::vector<int>>(standard_deviation_distribution());
```

Compile-time error: can't construct a vector<int> from a vector<double>

Constructing a vector of one type from a vector of another type

```cpp
auto standard_deviation_distribution() -> std::vector<double>;
auto const intermediate = standard_deviation_distribution();

std::vector<int>(intermediate.begin(), intermediate.end());
```

Captures vs parameters

```
[blue_number](card const x) {
    return x == blue_number;
}
```

Generating a sequence of integers on demand

```cpp
auto const first_ten = std::vector<int>{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
auto const first_hundred = std::vector<int>{
    0, 1, 2, 3, /* ... */, 99,
};
auto const first_thousand = std::vector<int>{
    0, 1, 2, 3, /* ... */, 999,
};
```

Generating a sequence of integers on demand

```cpp
auto const first_ten = std::vector<int>{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
auto const first_hundred = std::vector<int>{
    0, 1, 2, 3, /* ... */, 99,
};
auto const first_thousand = std::vector<int>{
    0, 1, 2, 3, /* ... */, 999,
};
```

Generating a sequence of integers on demand

```
auto const first_ten = std::vector<int>{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
auto const first_hundred = std::vector<int>{
    0, 1, 2, 3, /* ... */, 99,
};
auto const first_thousand = std::vector<int>{
    0, 1, 2, 3, /* ... */, 999,
};
```

Generating a sequence of integers on demand

```cpp
// #include <range/v3/numeric.hpp>
auto first_ten_thousand = std::vector<int>(10'000);

// populates vector with values [0, 10'000)
ranges::iota(first_ten_thousand, 0);
```

Generating a sequence of integers on demand

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>

auto first_hundred = views::iota(0, 100);
auto const all_at_once = first_hundred
                       | ranges::to<std::vector>;

CHECK(ranges::equal(first_hundred, all_at_once));
```

Filters ("keep if")

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto is_blue = [](card const c) { return c.colour == colour::blue; };
auto all_blue = hand | views::filter(is_blue);

auto const expected = std::vector<card>{
    blue_number,
    blue_number,
    blue_skip,
    blue_number,
    blue_number,
    blue_skip,
};

auto const actual = all_blue | ranges::to<std::vector>;
CHECK(expected == actual);
```

Filters ("remove if")

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto is_blue = [](card const c) { return c.colour == colour::blue; };
auto no_blue = hand | views::remove_if(is_blue);

auto const expected = std::vector<card>{
    red_number,
    green_draw_two,
    yellow_draw_four,
};

auto const actual = no_blue | ranges::to<std::vector>;
CHECK(expected == actual);
```

Reversing

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const is_blue_card = [](card const c) { return c.colour == colour::blue; };
{
    auto const result = ranges::find_if(hand, is_blue_card);
    REQUIRE(result != hand.end());
    CHECK(*result == blue_number);
}
```

Reversing

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const is_blue_card = [](card const c) { return c.colour == colour::blue; };
{
    auto const result = ranges::find_if(hand, is_blue_card);
    REQUIRE(result != hand.end());
    CHECK(*result == blue_number);
}
{
    auto back_to_front = hand | views::reverse;

    auto const result = ranges::find_if(back_to_front, is_blue_card);
    REQUIRE(result != back_to_front.end())
    CHECK(*result == blue_skip);
}
```

In-place transform

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto swap_blue = [](card const c) {
    return c.colour != colour::blue ? c : card{colour::green, c.value};
};

auto const expected = std::vector<card>{
    red_number,
    green_number,
    green_draw_two,
    green_number,
    green_skip,
    yellow_draw_four,
    green_number,
    green_number,
    green_skip,
};

auto const actual = hand | views::transform(swap_blue);
CHECK(expected == actual);
```

Splitting strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;
using namespace std::string_literals;

auto const sentence = "the quick brown fox jumps over the lazy dog"s;
auto to_string = [](auto x) { return x | ranges::to<std::string>; };
auto const individual_words = sentence
                            | views::split(' ')
                            | views::transform(to_string)
                            | ranges::to<std::vector>;

auto const expected = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

CHECK(individual_words == expected);
```

Splitting strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;
using namespace std::string_literals;

auto const sentence = "the quick brown fox jumps over the lazy dog"s;
auto to_string = [](auto x) { return x | ranges::to<std::string>; };
auto const individual_words = sentence
                            | views::split(' ')
                            | views::transform(to_string)
                            | ranges::to<std::vector>;

auto const expected = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

CHECK(individual_words == expected);
```

Splitting strings

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;
using namespace std::string_literals;

auto const sentence = "the quick brown fox jumps over the lazy dog"s;
auto to_string = [](auto x) { return x | ranges::to<std::string>; };
auto const individual_words = sentence
                            | views::split(' ')
                            | views::transform(to_string)
                            | ranges::to<std::vector>;

auto const expected = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

CHECK(individual_words == expected);
```

Splitting strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;
using namespace std::string_literals;

auto const sentence = "the quick brown fox jumps over the lazy dog"s;
auto to_string = [](auto x) { return x | ranges::to<std::string>; };
auto const individual_words = sentence
                            | views::split(' ')
                            | views::transform(to_string)
                            | ranges::to<std::vector>;

auto const expected = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

CHECK(individual_words == expected);
```

Joining strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "thequickbrownfoxjumpsoverthelazydog"s;
CHECK(sentence == expected);
```

Joining strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "thequickbrownfoxjumpsoverthelazydog"s;
CHECK(sentence == expected);
```

Joining strings

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "thequickbrownfoxjumpsoverthelazydog"s;
CHECK(sentence == expected);
```

Joining strings

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join(' ') | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Joining strings

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join(' ') | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Joining strings

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const individual_words = std::vector<std::string>{
    "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
};

auto const sentence = words | views::join(' ') | ranges::to<std::string>;

using namespace std::string_literals;
auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Concatenating ranges

```
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

using namespace std::string_literals;
auto const first = "the quick brown "s;
auto const second = "fox jumps over"s;
auto const thrid = std::vector<std::string>{" the", "lazy", "dog"};

auto const sentence = views::concat(first, second, third | views::join(' '))
                    | ranges::to<std::string>;

auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Concatenating ranges

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

using namespace std::string_literals;
auto const first = "the quick brown "s;
auto const second = "fox jumps over"s;
auto const thrid = std::vector<std::string>{" the", "lazy", "dog"};

auto const sentence = views::concat(first, second, third | views::join(' '))
                    | ranges::to<std::string>;

auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Concatenating ranges

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

using namespace std::string_literals;
auto const first = "the quick brown "s;
auto const second = "fox jumps over"s;
auto const thrid = std::vector<std::string>{" the", "lazy", "dog"};

auto const sentence = views::concat(first, second, third | views::join(' '))
                    | ranges::to<std::string>;

auto const expected = "the quick brown fox jumps over the lazy dog"s;
CHECK(sentence == expected);
```

Use only the first *n* elements

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const front3 = hand | views::take(3) | ranges::to<std::vector>;
auto const expeceted std::vector<card>{
    red_number,
    blue_number,
    green_draw_two,
};
CHECK(front3 == expected);
```

Use all but the first *n* elements

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const back6 = hand | views::drop(3) | ranges::to<std::vector>;
auto const expected = std::vector<card>{
    blue_number,
    blue_skip,
    yellow_draw_four,
    blue_number,
    blue_number,
    blue_skip,
};

CHECK(back6 == expected);
```

Use only the last *n* elements

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const back2 = hand | views::take_last(2) | ranges::to<std::vector>;
auto const expeceted std::vector<card>{
    blue_number,
    blue_skip,
};
CHECK(back2 == expected);
```

Don't use the last *n* elements

```cpp
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto const front6 = hand | views::drop_last(2) | ranges::to<std::vector>;
auto const expeceted std::vector<card>{
    red_number,
    blue_number,
    green_draw_two,
    blue_number,
    blue_skip,
    yellow_draw_four,
    blue_number,
};
CHECK(front6 == expected);
```

Iterating over multiple ranges at once

```cpp
// #include <range/v3/numeric.hpp>
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto hamming_distance(std::string const& s1,
                      std::string const& s2) -> int {
    auto different = views::zip_with(ranges::not_equal_to{}, s1, s2);
    return ranges::accumulate(different, 0);
}

CHECK(hamming_distance("chew", "chop") == 2);
CHECK(hamming_distance("hello", "world") == 4);
```

Iterating over multiple ranges at once

```cpp
// #include <range/v3/numeric.hpp>
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto hamming_distance(std::string const& s1,
                      std::string const& s2) -> int {
    auto different = views::zip_with(ranges::not_equal_to{}, s1, s2);
    return ranges::accumulate(different, 0);
}

CHECK(hamming_distance("chew", "chop") == 2);
CHECK(hamming_distance("hello", "world") == 4);
```

Iterating over multiple ranges at once

```cpp
// #include <range/v3/numeric.hpp>
// #include <range/v3/range.hpp>
// #include <range/v3/view.hpp>
namespace views = ranges::views;

auto hamming_distance(std::string const& s1,
                      std::string const& s2) -> int {
    auto different = views::zip_with(ranges::not_equal_to{}, s1, s2);
    return ranges::accumulate(different, 0);
}

CHECK(hamming_distance("chew", "chop") == 2);
CHECK(hamming_distance("hello", "world") == 4);
```

Writeable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Write	`v[i]`	`i->value`	`*i`

Successor	`j = i + n < ranges::distance(v)`
	`? i + n`
	`: ranges::distance(v);`
		`j = i->successor(n)`	`ranges::next(i, s, n)`

Advance	`++i`	`i = i->next`	`++i`
Comparison	`i < ranges::distance(v)`	`i != nullptr`	`i != s`

Populating an existing vector with a single value

```
auto reset_scores(std::vector<int>& scores) -> void {
    ranges::fill(scores, 0);
}
```

Copying values from one range to another, existing range

```cpp
auto chars_to_words(std::vector<char> const& from,
                    std::string& to) {
    return ranges::copy(from, to.begin());
}
```

Copying values from one range to another, existing range

```
auto chars_to_words(std::vector<char> const& from,
                    std::string& to) {
    return ranges::copy(from, to.begin());
}
```

What happens when ranges::distance(from) > ranges::distance(to)?

Copying values from one range to another, existing range

From:

H	e	l	l	o		t	h	e	r	e

To:

Insert iterators

```
// #include <range/v3/iterator.hpp>

auto to = std::vector<char>();
REQUIRE(to.empty());

ranges::copy(from, ranges::back_inserter(to));
CHECK(to == expected);
```

Works on containers with a push_back
member function like vector's

Insert iterators

```cpp
auto to = std::vector<char>(5);
REQUIRE(ranges::distance(from) > ranges::distance(to));
REQUIRE(not to.empty());

to.assign(from.begin(), from.end());
CHECK(to == expected);
```

Insert iterators

```
ranges::back_inserter
```

Works on containers that have push_back
(e.g. std::vector, std::string)

```
ranges::front_inserter
```

Works on containers that have push_front
(e.g. std::deque, std::list)

```
ranges::inserter
```

Works on containers that have insert
(e.g. all the above, absl::flat_hash_set/map)

I want to insert elements into my vector!

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | ranges::to<std::vector>;

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

I want to insert elements into my vector!

```
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | ranges::to<std::vector>;

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

I want to insert elements into my vector!

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | ranges::to<std::vector>;

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

I want to insert elements into my vector!

```
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | ranges::to<std::vector>;

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

I want to insert elements into my vector!

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | ranges::to<std::vector>;

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

...but we didn't really need two vectors?

```
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75) | views::transform(square);

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

...but we didn't really need two vectors?

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75) | views::transform(square);

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

...but we didn't really need two vectors?

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75) | views::transform(square);

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

...but we didn't really need two vectors?

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75) | views::transform(square);

auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

std::vector::insert wants a "common range"

```cpp
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75) | views::transform(square);

// This won't work because vector::insert expects begin and end to have
// the same type, but more_numbers' begin and end are different types.
auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

views::common gives us a "common range"

```
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | views::common;

// views::common will adapt the previous slide's more_numbers'
// begin and end into a type that has a _common_ begin and end
// type (hence the name views::common).
auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

views::common gives us a "common range"

```
auto some_numbers = views::concat(views::iota(0, 50), views::iota(75, 100))
                  | ranges::to<std::vector>;

auto square = [](int const x) { return x * x; };
auto more_numbers = views::iota(50, 75)
                  | views::transform(square)
                  | views::common;

// views::common will adapt the previous slide's more_numbers'
// begin and end into a type that has a _common_ begin and end
// type (hence the name views::common).
auto non_uniform_gap = ranges::adjacent_find(some_numbers,
    [](int const x, int const y) { return y - x != 1; });
some_numbers.insert(non_uniform_gap,
                    more_numbers.begin(), more_numbers.end());
```

Mutable iterators are iterators with both a read operation and a write operation.

What's going on here?

```cpp
auto from = std::vector<int>(10);
auto to = std::vector<int>(10);

// ...

// We would use ranges::copy IRL
for (auto i = from.begin(), j = to.begin();
     i != from.end() and j != to.end(); ++i, ++j)
{
    *i = *j;
}
```

What's going on here?

```
auto from = std::vector<int>(10);
auto to = std::vector<int>(10);

// ...

// We would use ranges::copy IRL
for (auto i = from.begin(), j = to.begin();
     i != from.end() and j != to.end(); ++i, ++j)
{
    *i = *j;
}
```

i is the read iterator, not the write one!

Iterator kinds

Let T be the placeholder for any type.

```
// mutable iterator       (similar to `T&`)
std::vector<T>::iterator

// read-only iterator     (similar to `T const&`)
std::vector<T>::const_iterator
```

Constant containers only have const_iterator

Getting a const_iterator from a mutable vector

```
auto from = std::vector<int>(10);
auto to = std::vector<int>(10);

// ...

// We would use ranges::copy IRL
for (auto i = from.cbegin(), j = to.begin();
     i != from.cend() and j != to.end(); ++i, ++j)
    *i = *j; // compile-time error: can't write to a const_iterator
}
```