Custom Views for the Rest of Us

JACOB RICE

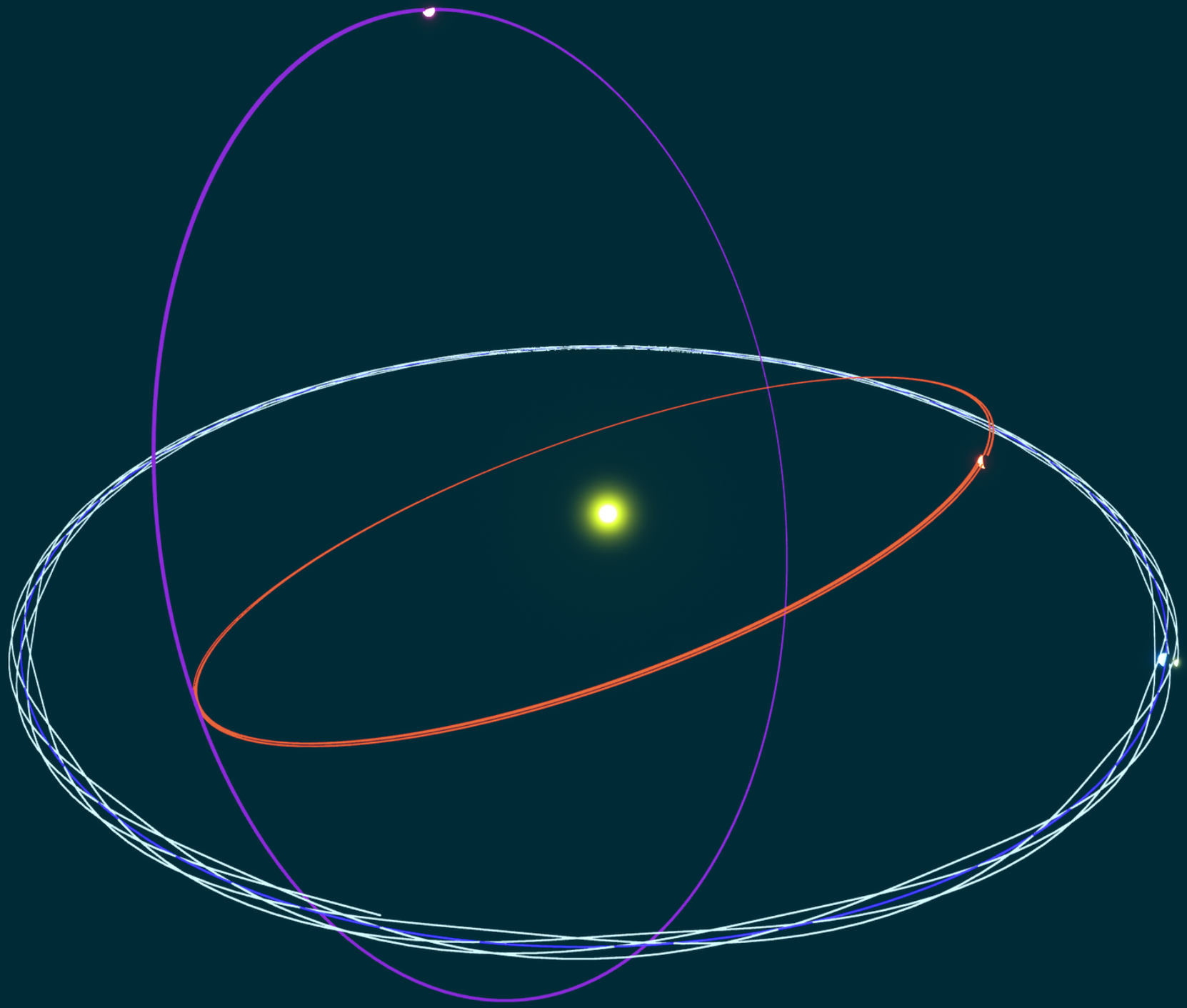
Custom Views for the Rest of Us

—

Jacob Rice

jacob.rice.cpp@gmail.com

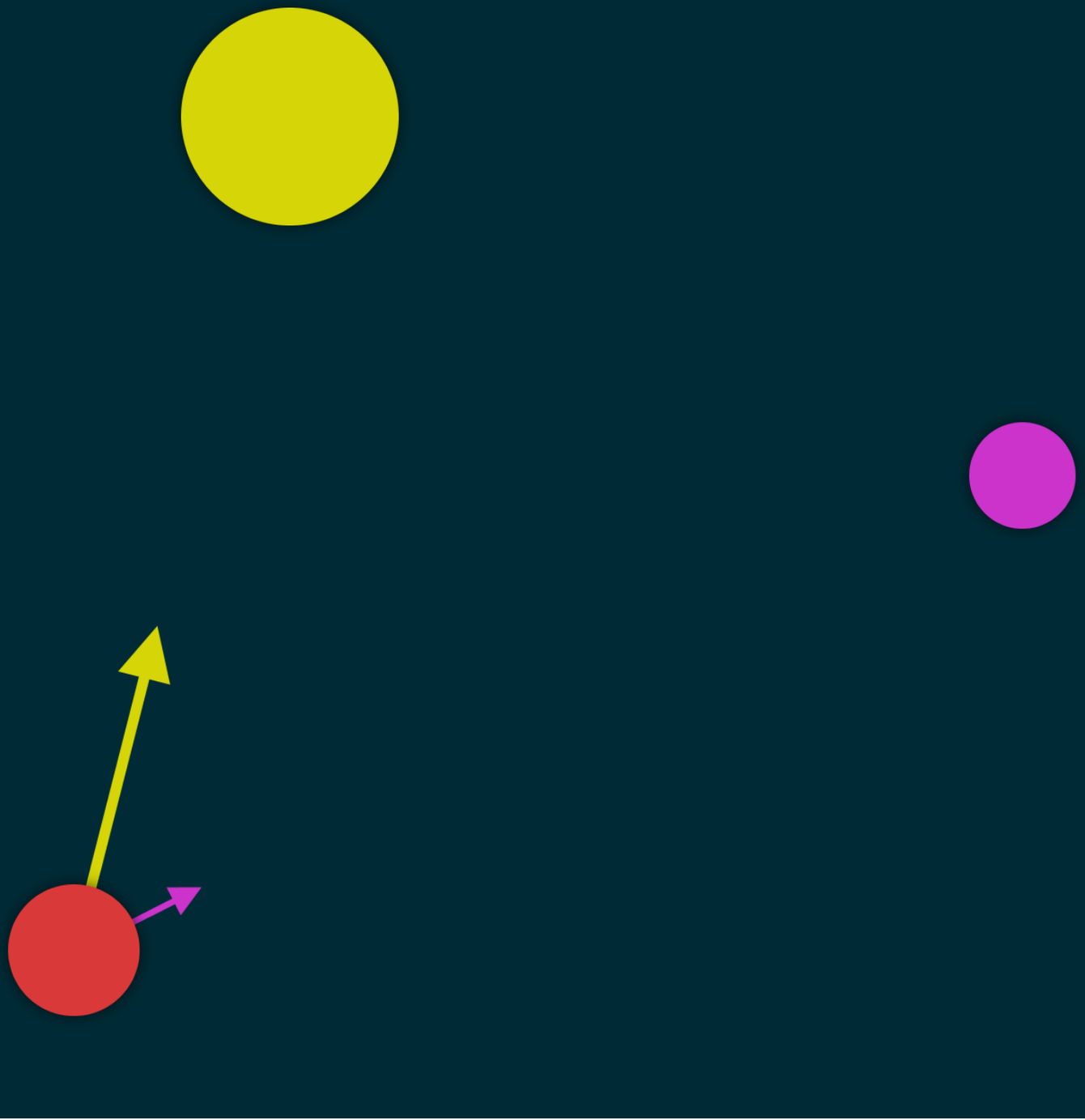
```
1  using namespace std::views;
2  for (int i : iota(2) | filter(even) | transform(square))
3      std::cout << i << ' ';
```

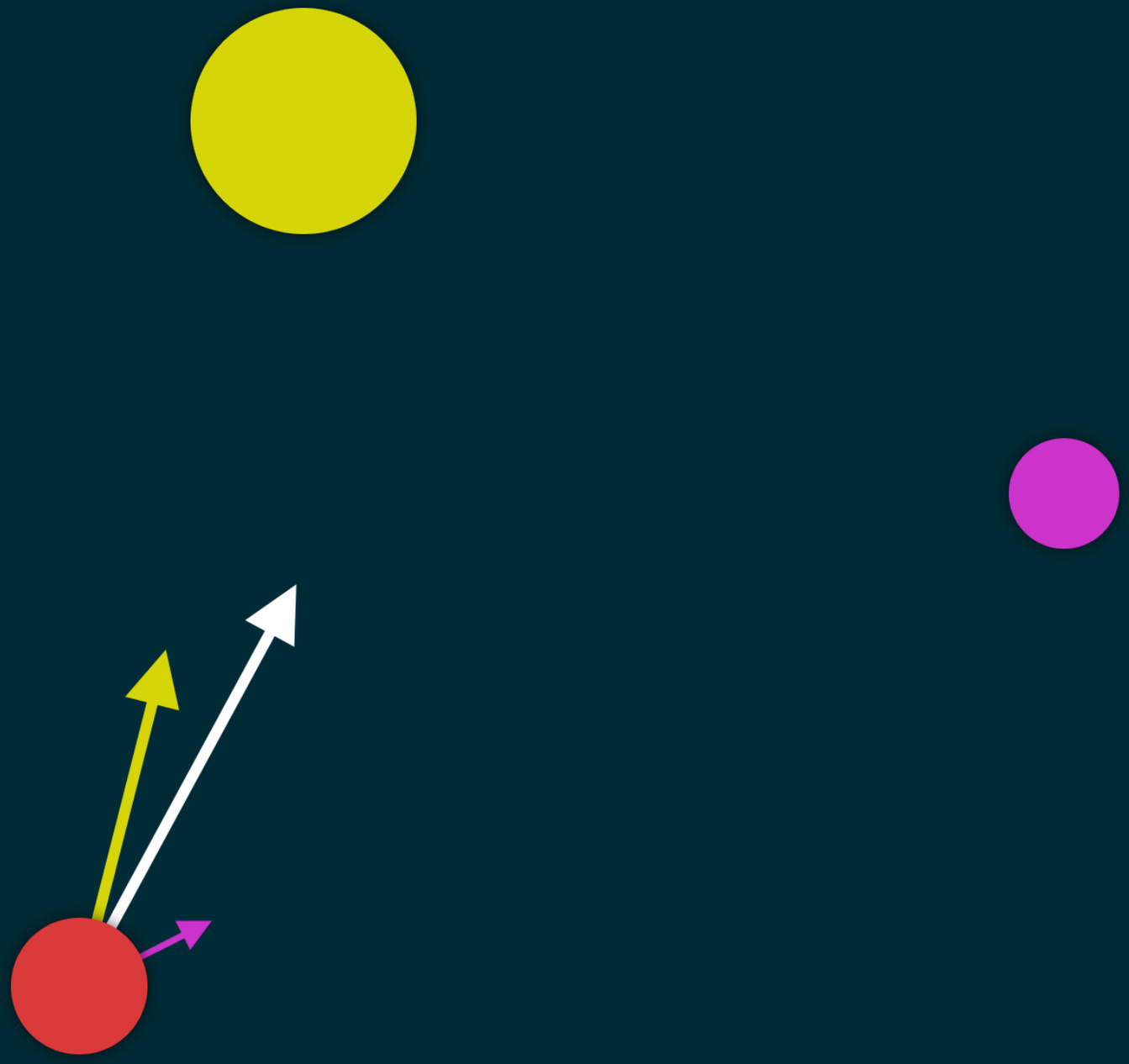




```
1  template <class T>
2  concept GravitationalBody = requires(T const t)
3  {
4      { t.GetMass() } -> std::floating_point;
5      { t.GetLocation() } -> std::same_as<FVector>;
6      { t.GetVelocity() } -> std::same_as<FVector>;
7  };
8
9  template <class T>
10 concept GravitationalRange = GravitationalBody<std::ranges::range_value_t<T>>;
```







```
def GetForceOnBody(this_body, all_bodies):
    all_forces = list()
    for other_body in all_bodies:
        all_forces.append(SingleForceOnThisBody(this_body, other_body))
    return sum(all_forces)
```

This can be done with a `transform_reduce`:

```
def GetForceOnBody(this_body, all_bodies):
    all_forces = list()
    for other_body in all_bodies:
        all_forces.append(SingleForceOnThisBody(this_body, other_body))
    return sum(all_forces)
```

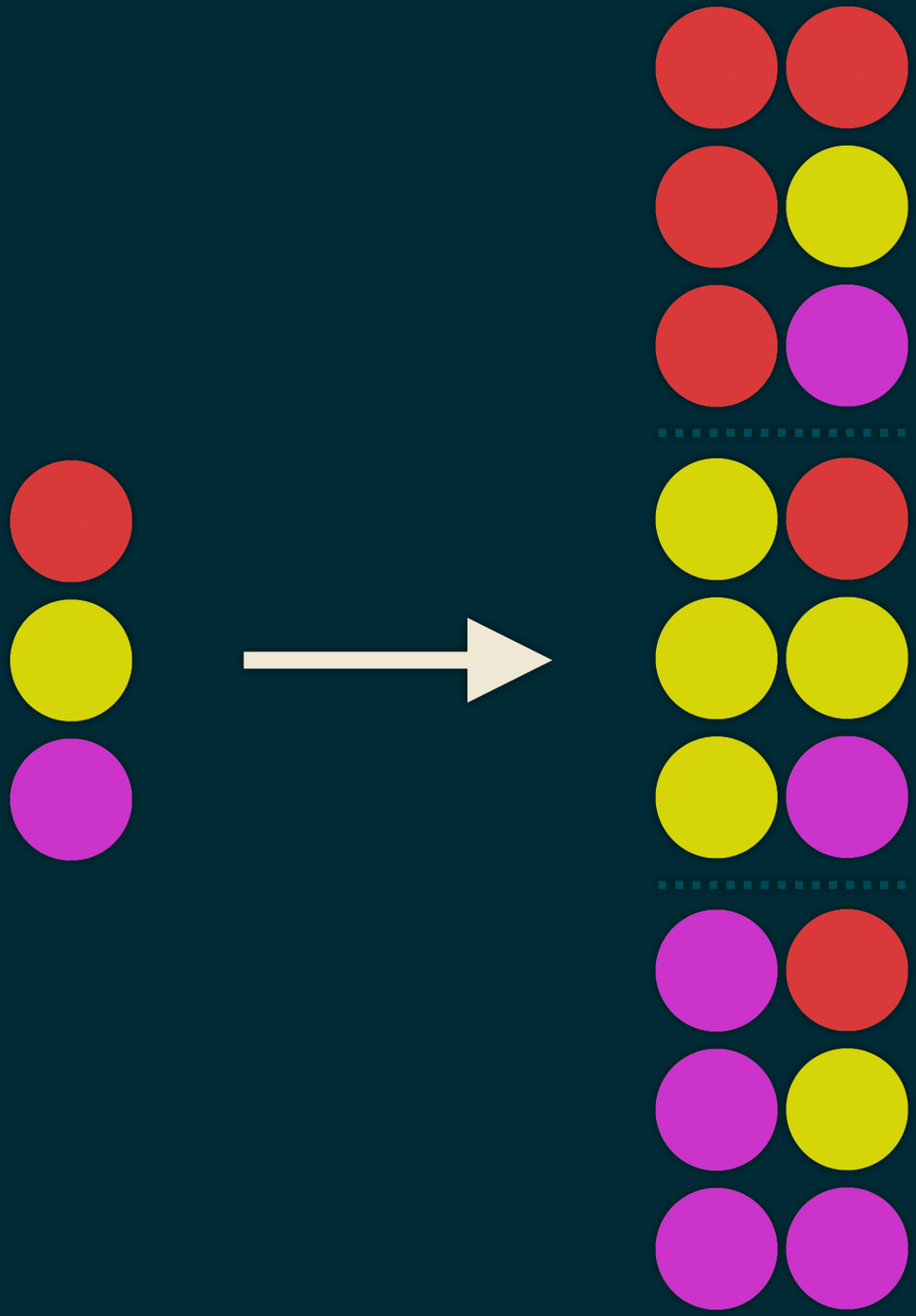
```cpp
template <GravitationalBody TThisBody, GravitationalRange TAllBodies>
FVector GetForceOnBody(TThisBody const& thisBody,
                       TAllBodies const& allBodies)
{
    auto const singleForceOnThisBody = [&](auto const& otherBody) -> FVector
    {
        if (otherBody == thisBody) return FVector{ 0 };

        auto const forceSize = CalculateForce(thisBody, otherBody);
        auto const forceDirection = ForceDirection(thisBody, otherBody);
        return forceSize * forceDirection;
    };

    return jtl::transform_reduce(allBodies, FVector{ 0 },
                                 std::plus<>{}, singleForceOnThisBody);
}
```

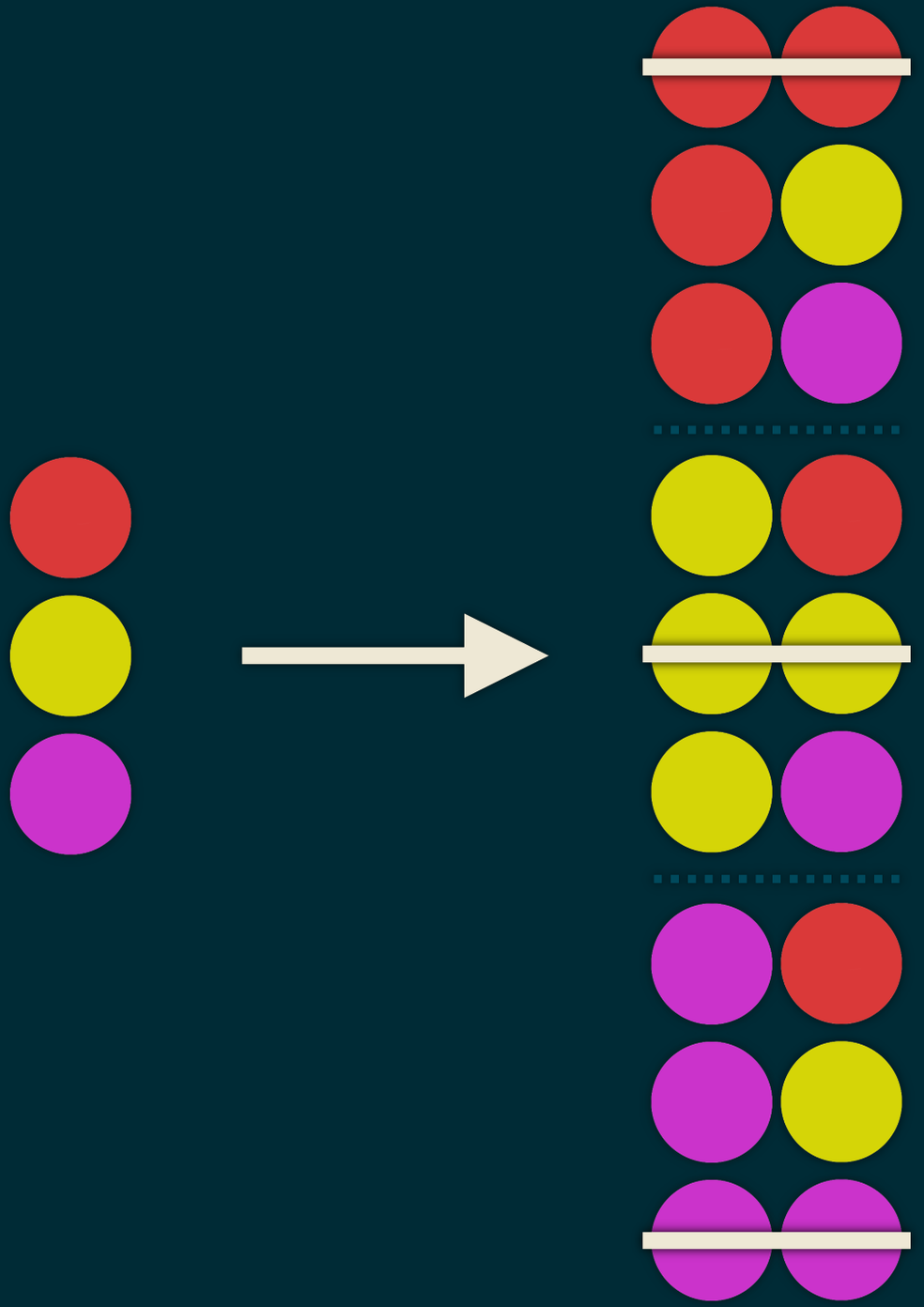
```cpp
template <GravitationalBody TThisBody, GravitationalRange TAllBodies>
FVector GetForceOnBody(TThisBody const& thisBody,
                       TAllBodies const& allBodies)
{
    auto const singleForceOnThisBody = [&](auto const& otherBody) -> FVector
    {
        if (otherBody == thisBody) return FVector{ 0 };

        auto const forceSize = CalculateForce(thisBody, otherBody);
        auto const forceDirection = ForceDirection(thisBody, otherBody);
        return forceSize * forceDirection;
    };

    return jtl::transform_reduce(allBodies, FVector{ 0 },
                                 std::plus<>{}, singleForceOnThisBody);
}
```

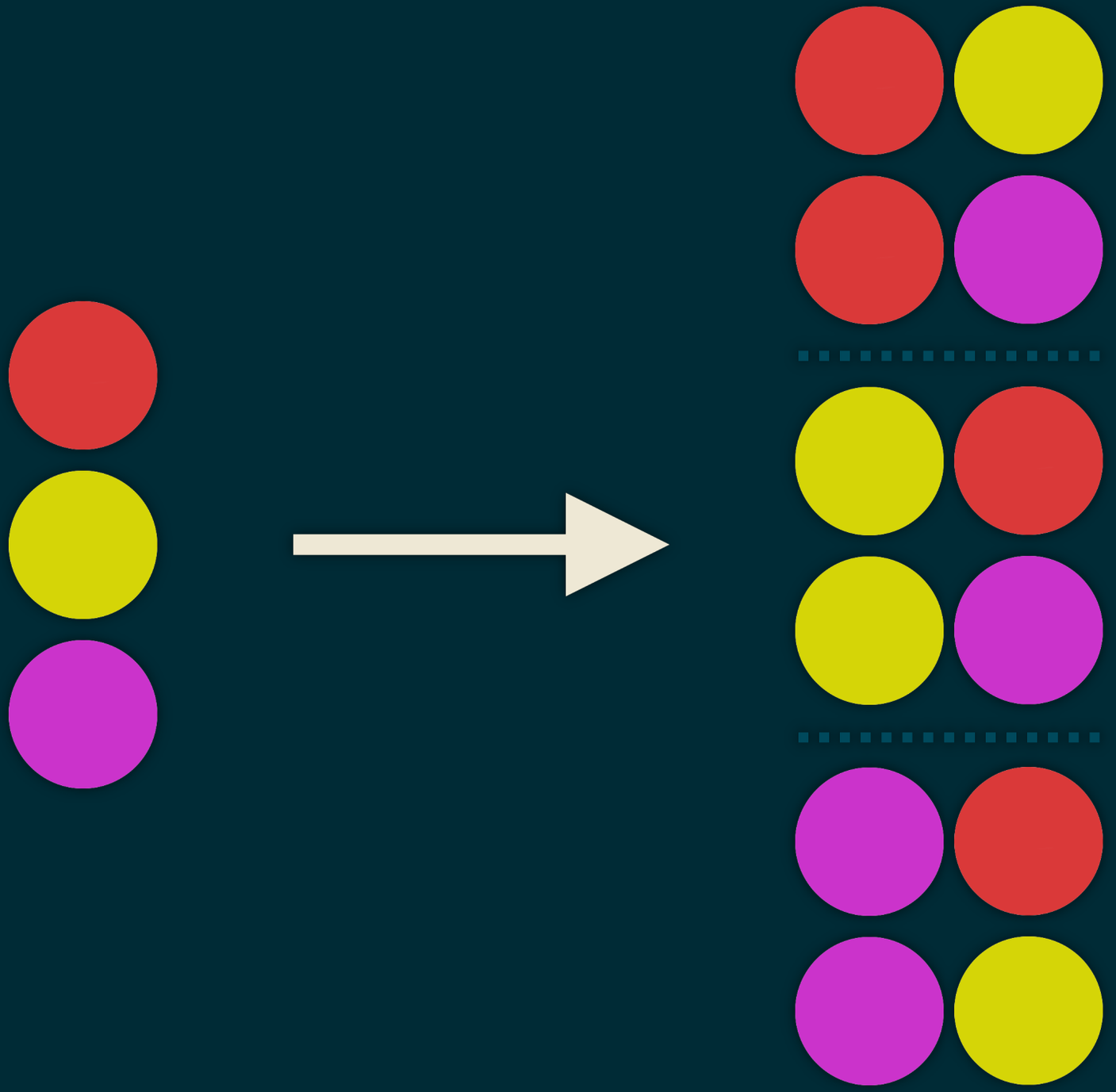
```cpp
template <GravitationalBody TThisBody, GravitationalRange TAllBodies>
FVector GetForceOnBody(TThisBody const& thisBody,
                       TAllBodies const& allBodies)
{
    auto const singleForceOnThisBody = [&](auto const& otherBody) -> FVector
    {
        if (otherBody == thisBody) return FVector{ 0 };

        auto const forceSize = CalculateForce(thisBody, otherBody);
        auto const forceDirection = ForceDirection(thisBody, otherBody);
        return forceSize * forceDirection;
    };

    return jtl::transform_reduce(allBodies, FVector{ 0 },
                                 std::plus<>{}, singleForceOnThisBody);
}
```

```cpp
template <GravitationalBody TThisBody, GravitationalRange TAllBodies>
FVector GetForceOnBody(TThisBody const& thisBody,
                       TAllBodies const& allBodies)
{
    auto const singleForceOnThisBody = [&](auto const& otherBody) -> FVector
    {
        if (otherBody == thisBody) return FVector{ 0 };

        auto const forceSize = CalculateForce(thisBody, otherBody);
        auto const forceDirection = ForceDirection(thisBody, otherBody);
        return forceSize * forceDirection;
    };

    return jtl::transform_reduce(allBodies, FVector{ 0 },
                                 std::plus<>{}, singleForceOnThisBody);
}
```

```cpp
template <GravitationalBody TThisBody, GravitationalRange TAllBodies>
FVector GetForceOnBody(TThisBody const& thisBody,
                       TAllBodies const& allBodies)
{
    auto const singleForceOnThisBody = [&](auto const& otherBody) -> FVector
    {
        if (otherBody == thisBody) return FVector{ 0 };

        auto const forceSize = CalculateForce(thisBody, otherBody);
        auto const forceDirection = ForceDirection(thisBody, otherBody);
        return forceSize * forceDirection;
    };

    return jtl::transform_reduce(allBodies, FVector{ 0 },
                                 std::plus<>{}, singleForceOnThisBody);
}
```







What is a `view`?

Easy to access:

```
1  namespace std {
2      namespace views = ranges::views;
3  }
```

Range

- Semantically: a collection of "things".
- Has `begin` and `end` functions.

That's it.

```
template <class T>
concept range = requires(T& t)
{
    ranges::begin(t);
    ranges::end(t);
};
```

View

- A specific type of range.
- Can be used in range adaptor pipelines.
- O(1) move (and maybe copy) construction.
- `ranges::enable_view<T>` is true.

```
template <class T>
concept view = ranges::range<T> &&
               movable<T> &&
               ranges::enable_view<T>;
```

range adaptor pipeline

noun

Composed range transformations that evaluate lazily as the resulting view is iterated.

Pipelines are cheap to create; “real” work is done during iteration.

```
1  using namespace std::views;
2  for (int i : iota(2) | filter(even) | transform(square))
3      std::cout << i << ' ';
```

Range Adaptor

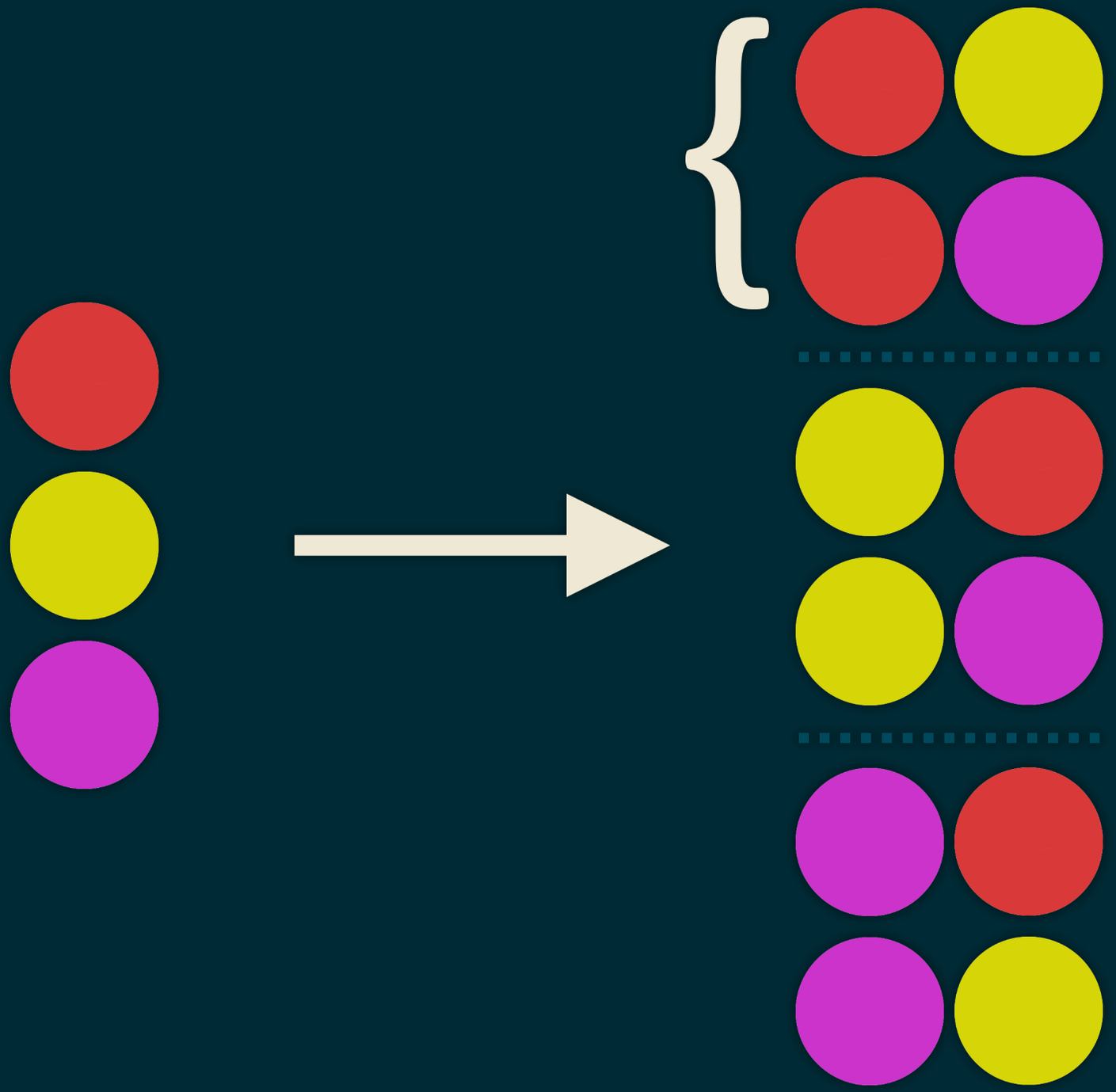
- A view that takes a range as input.
- Represents the range in a different way.
- Most views are adaptors.

```
using namespace std::ranges;
auto a = ref_view{ some_vect };
auto b = filter_view{ a, even };
```

Range Factory

- A view that generates elements on demand.
- Does not have any "backing" range.
- Only 4 in the standard.

```
using namespace std::ranges;
auto a = iota_view{ 1, 10 };
auto b = istream_view<std::string>{ words };
```



```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>> // CRTP!
{
private:
    TBase _vw;
public:
    class iterator { /* ... */ };

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>> // CRTP!
{
private:
    TBase _vw;
public:
    class iterator { /* ... */ };

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>> // CRTP!
{
private:
    TBase _vw;
public:
    class iterator { /* ... */ };

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>> // CRTP!
{
private:
    TBase _vw;
public:
    class iterator { /* ... */ };

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

A view is usually just a supplier
of a fancy iterator.

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;

public:
    constexpr all_pairs_view(TBase vw) : _vw{ std::move(vw) } { }
};

template <class Rng>
all_pairs_view(Rng&&) -> all_pairs_view<std::views::all_t<Rng>>;
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;

public:
    constexpr all_pairs_view(TBase vw) : _vw{ std::move(vw) } { }
};

template <class Rng>
all_pairs_view(Rng&&) -> all_pairs_view<std::views::all_t<Rng>>;
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
class inner_iterator
{
private:
    friend class inner_sentinel;

    using base_iterator = std::ranges::iterator_t<TBase>;
    using base_sentinel = std::ranges::sentinel_t<TBase>;
    using base_value_type = std::ranges::range_value_t<TBase>;
    using base_reference = std::ranges::range_reference_t<TBase>;
    /* ... */

public:
    using value_type = std::pair<base_value_type, base_value_type>;
    using reference = std::pair<base_reference, base_reference>;
    using difference_type = std::ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
    /* ... */
};
```

```
 1  class inner_iterator
 2  {
 3  private:
 4      friend class inner_sentinel;
 5
 6      using base_iterator = std::ranges::iterator_t<TBase>;
 7      using base_sentinel = std::ranges::sentinel_t<TBase>;
 8      using base_value_type = std::ranges::range_value_t<TBase>;
 9      using base_reference = std::ranges::range_reference_t<TBase>;
10      /* ... */
11
12  public:
13      using value_type = std::pair<base_value_type, base_value_type>;
14      using reference = std::pair<base_reference, base_reference>;
15      using difference_type = std::ptrdiff_t;
16      using iterator_category = std::forward_iterator_tag;
17      /* ... */
18  };
```

```
 1  class inner_iterator
 2  {
 3  private:
 4      friend class inner_sentinel;
 5
 6      using base_iterator = std::ranges::iterator_t<TBase>;
 7      using base_sentinel = std::ranges::sentinel_t<TBase>;
 8      using base_value_type = std::ranges::range_value_t<TBase>;
 9      using base_reference = std::ranges::range_reference_t<TBase>;
10      /* ... */
11
12  public:
13      using value_type = std::pair<base_value_type, base_value_type>;
14      using reference = std::pair<base_reference, base_reference>;
15      using difference_type = std::ptrdiff_t;
16      using iterator_category = std::forward_iterator_tag;
17      /* ... */
18  };
```

```
 1  class inner_iterator
 2  {
 3  private:
 4      friend class inner_sentinel;
 5
 6      using base_iterator = std::ranges::iterator_t<TBase>;
 7      using base_sentinel = std::ranges::sentinel_t<TBase>;
 8      using base_value_type = std::ranges::range_value_t<TBase>;
 9      using base_reference = std::ranges::range_reference_t<TBase>;
10      /* ... */
11
12  public:
13      using value_type = std::pair<base_value_type, base_value_type>;
14      using reference = std::pair<base_reference, base_reference>;
15      using difference_type = std::ptrdiff_t;
16      using iterator_category = std::forward_iterator_tag;
17      /* ... */
18  };
```

```cpp
class inner_iterator
{
private:
    friend class inner_sentinel;

    using base_iterator = std::ranges::iterator_t<TBase>;
    using base_sentinel = std::ranges::sentinel_t<TBase>;
    using base_value_type = std::ranges::range_value_t<TBase>;
    using base_reference = std::ranges::range_reference_t<TBase>;
    /* ... */

public:
    using value_type = std::pair<base_value_type, base_value_type>;
    using reference = std::pair<base_reference, base_reference>;
    using difference_type = std::ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
    /* ... */
};
```

```
class inner_iterator
{
private:
    /* ... */
    base_iterator _current_outer{};
    base_iterator _current_inner{};
    TBase const* _base{};

    void correct_inner_if_needed();

public:
    /* ... */
    inner_iterator() = default;
    inner_iterator(TBase const& base, base_iterator current_outer);

    [[nodiscard]] reference operator*() const;
    inner_iterator& operator++();
    [[nodiscard]] inner_iterator operator++(int)
    [[nodiscard]] bool operator==(inner_iterator const& rhs) const;
};
```

```
 1  class inner_iterator
 2  {
 3  private:
 4      /* ... */
 5      base_iterator _current_outer{};
 6      base_iterator _current_inner{};
 7      TBase const* _base{};
 8
 9      void correct_inner_if_needed();
10
11  public:
12      /* ... */
13      inner_iterator() = default;
14      inner_iterator(TBase const& base, base_iterator current_outer);
15
16      [[nodiscard]] reference operator*() const;
17      inner_iterator& operator++();
18      [[nodiscard]] inner_iterator operator++(int)
19      [[nodiscard]] bool operator==(inner_iterator const& rhs) const;
20  };
```

```
 1  class inner_iterator
 2  {
 3  private:
 4      friend class inner_sentinel;
 5
 6      using base_iterator = std::ranges::iterator_t<TBase>;
 7      using base_sentinel = std::ranges::sentinel_t<TBase>;
 8      using base_value_type = std::ranges::range_value_t<TBase>;
 9      using base_reference = std::ranges::range_reference_t<TBase>;
10
11      base_iterator _current_outer{};
12      base_iterator _current_inner{};
13      TBase const* _base{};
14
15      void correct_inner_if_needed()
16      {
17          if (_current_inner == _current_outer)
18          {
19              std::advance(_current_inner, 1);
20          }
21      }
22
```

```
43
44      inner_iterator& operator++()
45      {
46          ++_current_inner;
47          correct_inner_if_needed();
48          return *this;
49      }
50
51      [[nodiscard]] inner_iterator operator++(int)
52      {
53          auto tmp = *this;
54          ++(*this);
55          return tmp;
56      }
57
58      [[nodiscard]] bool operator==(inner_iterator const& rhs) const
59      {
60          return _current_outer == rhs._current_outer &&
61                 _current_inner == rhs._current_inner;
62      }
63
64      /* As of C++20, operator!= is auto-generated as !(operator==) */
```

```
45          {
46              ++_current_inner;
47              correct_inner_if_needed();
48              return *this;
49          }
50
51          [[nodiscard]] inner_iterator operator++(int)
52          {
53              auto tmp = *this;
54              ++(*this);
55              return tmp;
56          }
57
58          [[nodiscard]] bool operator==(inner_iterator const& rhs) const
59          {
60              return _current_outer == rhs._current_outer &&
61                     _current_inner == rhs._current_inner;
62          }
63
64          /* As of C++20, operator!= is auto-generated as !(operator==) */
65      };
```

```
45          {
46              ++_current_inner;
47              correct_inner_if_needed();
48              return *this;
49          }
50
51          [[nodiscard]] inner_iterator operator++(int)
52          {
53              auto tmp = *this;
54              ++(*this);
55              return tmp;
56          }
57
58          [[nodiscard]] bool operator==(inner_iterator const& rhs) const
59          {
60              return _current_outer == rhs._current_outer &&
61                     _current_inner == rhs._current_inner;
62          }
63
64          /* As of C++20, operator!= is auto-generated as !(operator==) */
65      };
```

```
45          {
46              ++_current_inner;
47              correct_inner_if_needed();
48              return *this;
49          }
50
51          [[nodiscard]] inner_iterator operator++(int)
52          {
53              auto tmp = *this;
54              ++(*this);
55              return tmp;
56          }
57
58          [[nodiscard]] bool operator==(inner_iterator const& rhs) const
59          {
60              return _current_outer == rhs._current_outer &&
61                     _current_inner == rhs._current_inner;
62          }
63
64          /* As of C++20, operator!= is auto-generated as !(operator==) */
65      };
```

```
45          {
46              ++_current_inner;
47              correct_inner_if_needed();
48              return *this;
49          }
50
51          [[nodiscard]] inner_iterator operator++(int)
52          {
53              auto tmp = *this;
54              ++(*this);
55              return tmp;
56          }
57
58          [[nodiscard]] bool operator==(inner_iterator const& rhs) const
59          {
60              return _current_outer == rhs._current_outer &&
61                     _current_inner == rhs._current_inner;
62          }
63
64          /* As of C++20, operator!= is auto-generated as !(operator==) */
65      };
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

What is a sentinel?

Value

- Special value, same type.
- Indicates end of a range.
- Most famous: `'\0'`

Type

- "New" to `std::ranges`.
- Class returned by `end()` calls.
- Checks `==` with `iterator` type.

Sentinel semantic requirements:

Let `s` and `i` be values of type `S` and `I`, respectively, such that `[i, s)` denotes a range. `sentinel_for<S, I>` is modeled only if:

- `i == s` is well-defined.
- If `bool(i != s)` then `i` is dereferenceable and `[++i, s)` denotes a range.
- `std::assignable_from<I&, S>` is either modeled or not satisfied.

Sentinel semantic requirements:

Let `s` and `i` be values of type `S` and `I`, respectively, such that `[i, s)` denotes a range. `sentinel_for<S, I>` is modeled only if:

- `i == s` is well-defined.
- If `bool(i != s)` then `i` is dereferenceable and `[++i, s)` denotes a range.
- `std::assignable_from<I&, S>` is either modeled or not satisfied.

```
1  template <class S, class I>
2  concept sentinel_for =
3      std::semiregular<S> &&
4      std::input_or_output_iterator<I> &&
5      __WeaklyEqualityComparableWith<S, I>;
```

```
1  template <class S, class I>
2  concept sentinel_for =
3      std::semiregular<S> &&
4      std::input_or_output_iterator<I> &&
5      __WeaklyEqualityComparableWith<S, I>;
```

```
1  template <class S, class I>
2  concept sentinel_for =
3      std::semiregular<S> &&
4      std::input_or_output_iterator<I> &&
5      __WeaklyEqualityComparableWith<S, I>;
```

```
1  template <class S, class I>
2  concept sentinel_for =
3      std::semiregular<S> &&
4      std::input_or_output_iterator<I> &&
5      __WeaklyEqualityComparableWith<S, I>;
```

```
template <class S, class I>
concept sentinel_for =
    std::semiregular<S> &&
    std::input_or_output_iterator<I> &&
    __WeaklyEqualityComparableWith<S, I>;
```

```cpp
class inner_sentinel
{
public:
    [[nodiscard]] bool operator==(inner_sentinel const&) const
    {
        return true;
    }

    [[nodiscard]] bool operator==(inner_iterator const& rhs) const
    {
        return rhs._current_inner == std::ranges::end(*rhs._base);
    }

    /* As of C++20, the compiler uses this^ to generate these:
     *    sentinel != iterator
     *    iterator == sentinel
     *    iterator != sentinel
     */
};
```

Why use a sentinel instead of an `end` iterator?

```
class inner_iterator
{
    /* ... */
    inner_iterator(TBase const& base, base_iterator current_outer)
        : _current_outer{ std::move(current_outer) }
        , _current_inner{ std::ranges::begin(base) }
        , _base{ std::addressof(base) }
    {
        correct_inner_if_needed();
    }
    /* ... */
};
```

```
1  class inner_iterator
2  {
3      /* ... */
4      inner_iterator(TBase const& base, base_iterator current_outer)
5          : _current_outer{ std::move(current_outer) }
6          , _current_inner{ std::ranges::begin(base) }
7          , _base{ std::addressof(base) }
8      {
9          correct_inner_if_needed();
10     }
11     /* ... */
12 };
```

```
1  class inner_iterator
2  {
3      /* ... */
4      inner_iterator(TBase const& base, base_iterator current_outer,
5                     base_iterator current_inner)
6          : _current_outer{ std::move(current_outer) }
7          , _current_inner{ std::move(current_inner) }
8          , _base{ std::addressof(base) }
9      {
10         correct_inner_if_needed(); // Should I even do this..?
11     }
12     /* ... */
13 };
```

```
1  class inner_iterator
2  {
3      /* ... */
4      inner_iterator(TBase const& base, base_iterator current_outer,
5                     base_iterator current_inner)
6          : _current_outer{ std::move(current_outer) }
7          , _current_inner{ std::move(current_inner) }
8          , _base{ std::addressof(base) }
9      {
10         correct_inner_if_needed(); // Should I even do this..?
11     }
12     /* ... */
13 };
```

This constructor is only for `end()` to use, but nothing actually tells us that.

Sentinels separate concerns.

The definition of “end” is in its own class.

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
class inner_view : public std::ranges::view_interface<inner_view>
{
private:
    using base_iterator = std::ranges::iterator_t<TBase>;

    all_pairs_view const* _parent{};
    base_iterator _current_outer{};

public:
    using iterator = inner_iterator;
    using sentinel = inner_sentinel;

    inner_view() = default;

    inner_view(all_pairs_view const& parent, base_iterator current)
        : _parent{ std::addressof(parent) }
        , _current_outer{ std::move(current) } { }

    [[nodiscard]] iterator begin() const { return { *_parent, _current_outer }; }
    [[nodiscard]] sentinel end() const { return { }; }
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```
 1  class outer_iterator
 2  {
 3  private:
 4      using base_iterator = std::ranges::iterator_t<TBase>;
 5
 6      all_pairs_view const* _parent{};
 7      base_iterator _current{};
 8
 9  public:
10      using value_type = inner_view;
11      using reference = value_type; // I'll come back to this
12      using difference_type = std::ptrdiff_t;
13      using iterator_category = std::forward_iterator_tag;
14
15      outer_iterator() = default;
16
17      outer_iterator(all_pairs_view const& parent, base_iterator current)
18          : _parent{ std::addressof(parent) }
19          , _current{ std::move(current) } { }
20
21      [[nodiscard]] reference operator*() const
22      {
```

```
15      outer_iterator() = default;
16
17      outer_iterator(all_pairs_view const& parent, base_iterator current)
18          : _parent{ std::addressof(parent) }
19          , _current{ std::move(current) } { }
20
21      [[nodiscard]] reference operator*() const
22      {
23          return inner_view{ *_parent, _current };
24      }
25
26      outer_iterator& operator++()  // Postfix necessary but omitted
27      {
28          ++_current;
29          return *this;
30      }
31
32      [[nodiscard]] bool operator==(outer_iterator const& rhs) const
33      {
34          return _parent == rhs._parent && _current == rhs._current;
35      }
36  };
```

```cpp
class outer_iterator
{
    /* ... */
public:
    using value_type = inner_view;
    using reference = value_type; // Why isn't this "value_type&"?
    using difference_type = std::ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
    /* ... */
};
```

```
1  class outer_iterator
2  {
3      /* ... */
4  public:
5      using value_type = inner_view;
6      using reference = value_type; // Why isn't this "value_type&"?
7      using difference_type = std::ptrdiff_t;
8      using iterator_category = std::forward_iterator_tag;
9      /* ... */
10 };
```

```
1  class outer_iterator
2  {
3      using reference = value_type;
4
5      [[nodiscard]] reference operator*() const
6      {
7          return inner_view{ *_parent, _current };
8      }
9  };
```

```
 1  class outer_iterator
 2  {
 3      using value_type = inner_view;
 4      using reference = value_type;
 5  };
 6
 7  class inner_iterator
 8  {
 9      using value_type = std::pair<base_value_type, base_value_type>;
10      using reference = std::pair<base_reference, base_reference>;
11  };
```

```
1  class outer_iterator
2  {
3      using value_type = inner_view;
4      using reference = value_type;
5  };
6
7  class inner_iterator
8  {
9      using value_type = std::pair<base_value_type, base_value_type>;
10     using reference = std::pair<base_reference, base_reference>;
11 };
```

```
vector<bool>
```

cppreference:

`std::vector<bool>` behaves similarly to `std::vector`, but in order to be space efficient, it:

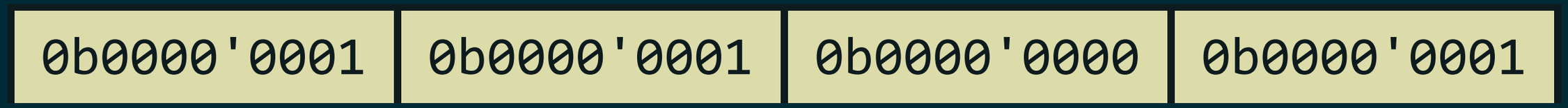
- Does not necessarily store its elements as a contiguous array.
- Exposes class `std::vector<bool>::reference` as a method of accessing individual bits. In particular, objects of this class are returned by `operator[]` by value.
- Does not use `std::allocator_traits::construct` to construct bit values.
- Does not guarantee that different elements in the same container can be modified concurrently by different threads.

cppreference:

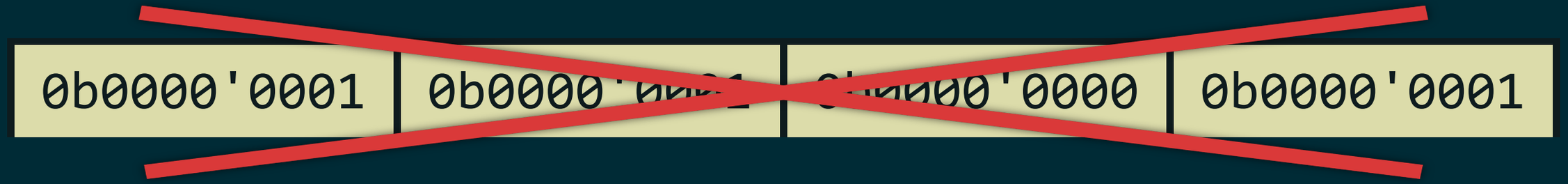
`std::vector<bool>` behaves similarly to `std::vector`, but in order to be space efficient, it:

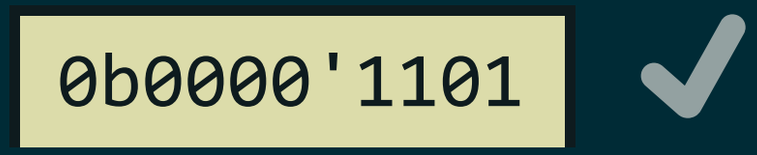
- Does not necessarily store its elements as a contiguous array.
- Exposes class `std::vector<bool>::reference` as a method of accessing individual bits. In particular, objects of this class are returned by `operator[]` by value.
- Does not use `std::allocator_traits::construct` to construct bit values.
- Does not guarantee that different elements in the same container can be modified concurrently by different threads.

```
std::vector<bool> v{ true, true, false, true };
```



```
std::vector<bool> v{ true, true, false, true };
```





Member type	Definition
`value_type`	`T`
`allocator_type`	`Allocator`
`size_type`	Unsigned integer type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`reference`	`Allocator::reference` (until C++11) `value_type&` (since C++11)
`const_reference`	`Allocator::const_reference` (until C++11) `const value_type&` (since C++11)
`pointer`	`Allocator::pointer` (until C++11) `std::allocator_traits<Allocator>::pointer` (since C++11)
`const_pointer`	`Allocator::const_pointer` (until C++11) `std::allocator_traits<Allocator>::const_pointer` (since C++11)
`iterator`	*LegacyRandomAccessIterator* to `value_type`
`const_iterator`	*LegacyRandomAccessIterator* to `const value_type`
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member type	Definition
`value_type`	`bool`
`allocator_type`	`Allocator`
`size_type`	implementation-defined
`difference_type`	implementation-defined
`reference`	proxy class representing a reference to a single bool (class)
`const_reference`	`bool`
`pointer`	implementation-defined
`const_pointer`	implementation-defined
`iterator`	implementation-defined
`const_iterator`	implementation-defined
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member type	Definition
`value_type`	`bool`
`allocator_type`	`Allocator`
`size_type`	implementation-defined
`difference_type`	implementation-defined
`reference`	proxy class representing a reference to a single bool (class)
`const_reference`	`bool`
`pointer`	implementation-defined
`const_pointer`	implementation-defined
`iterator`	implementation-defined
`const_iterator`	implementation-defined
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member type	Definition
`value_type`	`bool`
`allocator_type`	`Allocator`
`size_type`	implementation-defined
`difference_type`	implementation-defined
`reference`	proxy class representing a reference to a single bool (class)
`const_reference`	`bool`
`pointer`	implementation-defined
`const_pointer`	implementation-defined
`iterator`	implementation-defined
`const_iterator`	implementation-defined
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member type	Definition
`value_type`	`bool`
`allocator_type`	`Allocator`
`size_type`	implementation-defined
`difference_type`	implementation-defined
`reference`	proxy class representing a reference to a single bool (class)
`const_reference`	`bool`
`pointer`	implementation-defined
`const_pointer`	implementation-defined
`iterator`	implementation-defined
`const_iterator`	implementation-defined
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

“When Is a Container Not a Container?”

- Herb Sutter, *C++ Report*, May 1999

```
1  template <class T>
2  void f(T& t)
3  {
4      typename T::value_type* p1 = &t[0];
5      typename T::value_type* p2 = &*t.begin();
6      // ... do something with *p1 and *p2 ...
7  }
```

What can we do about this?

Don't write code like this:

```
1  template <class T>
2  void f(T& t)
3  {
4      typename T::value_type* p1 = &t[0];
5      typename T::value_type* p2 = &*t.begin();
6      // ... do something with *p1 and *p2 ...
7  }
```

Write code expecting "proxies".

that conforms

Write code ~~expecting "proxies".~~

with C++20

that conforms ^

Write code ~~expecting "proxies".~~

If you want more, read this:

P0022: Proxy Iterators for the Ranges Extensions

- Eric Niebler, 2015

proxy reference

noun

> A type which provides read (and write) operations for another type, even when it’s an rvalue.

proxy iterator

noun

> An iterator whose reference type is a proxy reference.

Proxy iterator examples:

Proxy iterator examples:

`std::vector<bool>::iterator`:

- Dereference returns rvalue of type `vector<bool>::reference`.
- `reference` type provides read and write operations to bits.

Proxy iterator examples:

`std::vector<bool>::iterator`:

- Dereference returns rvalue of type `vector<bool>::reference`.
- `reference` type provides read and write operations to bits.

`all_pairs_view::inner_iterator`:

- Dereference returns an rvalue of type `std::pair<T&, T&>`.
- `pair` provides read and write operations to underlying values.

"Are proxy iterators even allowed?"

"Are proxy iterators even allowed?"

Yes!

"Are proxy iterators even allowed?"

Yes! (with C++20)

Pre-C++20

```
 1  template <class I>
 2  concept __LegacyForwardIterator = // then called a “named requirement”
 3      __LegacyInputIterator<I> &&
 4      std::constructible_from<I> &&
 5      std::is_lvalue_reference_v<std::iter_reference_t<I>> &&
 6      std::same_as<
 7          std::remove_cvref_t<std::iter_reference_t<I>>,
 8          typename std::indirectly_readable_traits<I>::value_type> &&
 9      requires (I i) {
10          {  i++ } -> std::convertible_to<const I&>;
11          { *i++ } -> std::same_as<std::iter_reference_t<I>>;
12  };
```

Pre-C++20

```
 1  template <class I>
 2  concept __LegacyForwardIterator = // then called a “named requirement”
 3      __LegacyInputIterator<I> &&
 4      std::constructible_from<I> &&
 5      std::is_lvalue_reference_v<std::iter_reference_t<I>> &&
 6      std::same_as<
 7          std::remove_cvref_t<std::iter_reference_t<I>>,
 8          typename std::indirectly_readable_traits<I>::value_type> &&
 9      requires (I i) {
10          {  i++ } -> std::convertible_to<const I&>;
11          { *i++ } -> std::same_as<std::iter_reference_t<I>>;
12  };
```

C++20

```
1  template<class I>
2  concept forward_iterator =
3      std::input_iterator<I> &&
4      std::derived_from<std::iterator_traits<I>, std::forward_iterator_tag> &&
5      std::incrementable<I> &&
6      std::sentinel_for<I, I>;
```

What this means:

What this means:

Before C++20, everything was confusing:

- `vector<bool>` uses a proxy type for its “reference”.
- `ForwardIterator::reference` must be an lvalue reference.
- `vector<bool>::iterator` wasn’t even a forward iterator. 

What this means:

Before C++20, everything was confusing:

- `vector<bool>` uses a proxy type for its “reference”.
- `ForwardIterator::reference` must be an lvalue reference.
- `vector<bool>::iterator` wasn’t even a forward iterator. 

C++20 makes iterators easier:

- Non-lvalue proxy reference types are actually supported.
- `vector<bool>` is “okay” now!

What else is required for proxy iterators to work?

Customization Point Objects

This has always been an awkward pattern:

```
1  template <class T>
2  void AlgorithmThatSwaps(T& t1, T& t2)
3  {
4      // Algorithm stuff...
5      using std::swap;
6      swap(t1, t2);
7      // More algorithm
8  }
```

This has always been an awkward pattern:

```
1  template <class T>
2  void AlgorithmThatSwaps(T& t1, T& t2)
3  {
4      // Algorithm stuff...
5      using std::swap;
6      swap(t1, t2);
7      // More algorithm
8  }
```

Pre-C++20

- `swap`, `begin`, and `end` are “customization points”.
- Special behavior for user types.
- Used Argument-Dependent Lookup (ADL) to find user version.
- Much easier to get wrong than right.

C++20 solution (N4381)

- Moving forward: *semiregular objects* that do ADL internally.
- One place for type constraints == no user circumventing.
- Always easy to use.
- Currently, only in the `std::ranges` namespace.

```cpp
template <class T>
void AlgorithmThatSwaps(T& t1, T& t2)
{
    // Algorithm stuff...
    std::ranges::swap(t1, t2);
    // More algorithm
}
```

Calls the custom version, if it exists
(same for `begin` and `end`):

```
1  template <class T>
2  void AlgorithmThatSwaps(T& t1, T& t2)
3  {
4      // Algorithm stuff...
5      std::ranges::swap(t1, t2);
6      // More algorithm
7  }
```

```
std::ranges::iter_swap
std::ranges::iter_move
```

```
1  template <class I>
2  void SwapIters(I lhs, I rhs)
3  {
4      using std::swap;
5      swap(*lhs, *rhs);
6  }
```

Proxy reference type could be `pair<T&, T&>`, so this just swapped two rvalues.

```
1  template <class I>
2  void SwapIters(I lhs, I rhs)
3  {
4      using std::swap;
5      swap(*lhs, *rhs);
6  }
```

std::ranges::iter_swap

`std::ranges::iter_swap`

- Improvement upon `std::iter_swap` from C++98.
- Intentionally designed *customization point* for proxy iterators.
- Used by all `std::ranges` algorithms that swap stuff.

`std::ranges::iter_swap`

- Improvement upon `std::iter_swap` from C++98.
- Intentionally designed *customization point* for proxy iterators.
- Used by all `std::ranges` algorithms that swap stuff.

If your generic code needs to swap elements in a container, use this.

You usually won't have to customize this; calls `iter_move` by default.

```cpp
template <class I>
std::iter_value_t<I> Extract(I iter)
{
    return std::move(*iter);
}
```

Proxy reference type could be `pair<T&, T&>`, so this moved an rvalue.

```
1  template <class I>
2  std::iter_value_t<I> Extract(I iter)
3  {
4      return std::move(*iter);
5  }
```

std::ranges::iter_move

`std::ranges::iter_move`

- Didn’t exist before, even after `std::move` came along.
- Another customization point for proxy iterators.
- Used by all `std::ranges` algorithms that move values.

`std::ranges::iter_move`

- Didn’t exist before, even after `std::move` came along.
- Another customization point for proxy iterators.
- Used by all `std::ranges` algorithms that move values.

If `std::move` won’t work for your iterators, you have to customize this.

Customization gives you semantically correct `iter_swap` for free.

Another benefit of sentinels:
Won’t work with the old algorithms.*

But how does one “customize” `std::ranges::iter_move`?

cppreference: `iter_move`

Obtains an rvalue reference or a prvalue temporary from a given iterator.

A call to `ranges::iter_move` is expression-equivalent to:

- `iter_move(std::forward<T>(t))`, if `std::remove_cvref_t<T>` is a class or enumeration type and the expression is well-formed in unevaluated context, where the overload resolution is performed with the following candidates:
  - `void iter_move();`
  - any declarations of `iter_move` found by argument-dependent lookup.
- otherwise, `std::move(*std::forward<T>(t))` if `*std::forward<T>(t)` is well-formed and is an lvalue,
- otherwise, `*std::forward<T>(t)` if `*std::forward<T>(t)` is well-formed and is an rvalue.

In all other cases, a call to `ranges::iter_move` is ill-formed, which can result in substitution failure when `ranges::iter_move(e)` appears in the immediate context of a template instantiation.

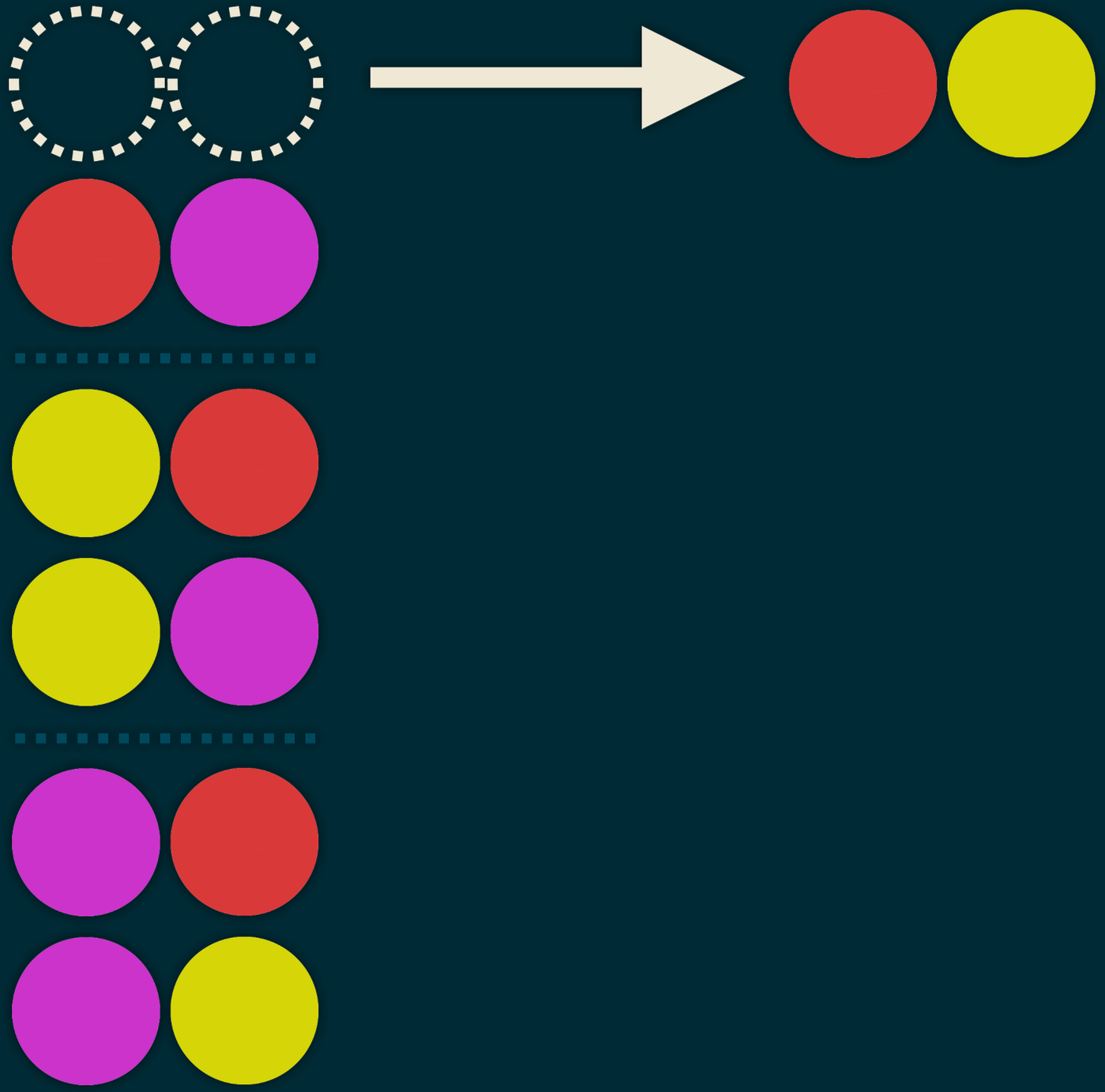
If `ranges::iter_move(e)` is not equal to `*e`, the program is ill-formed, no diagnostic required.

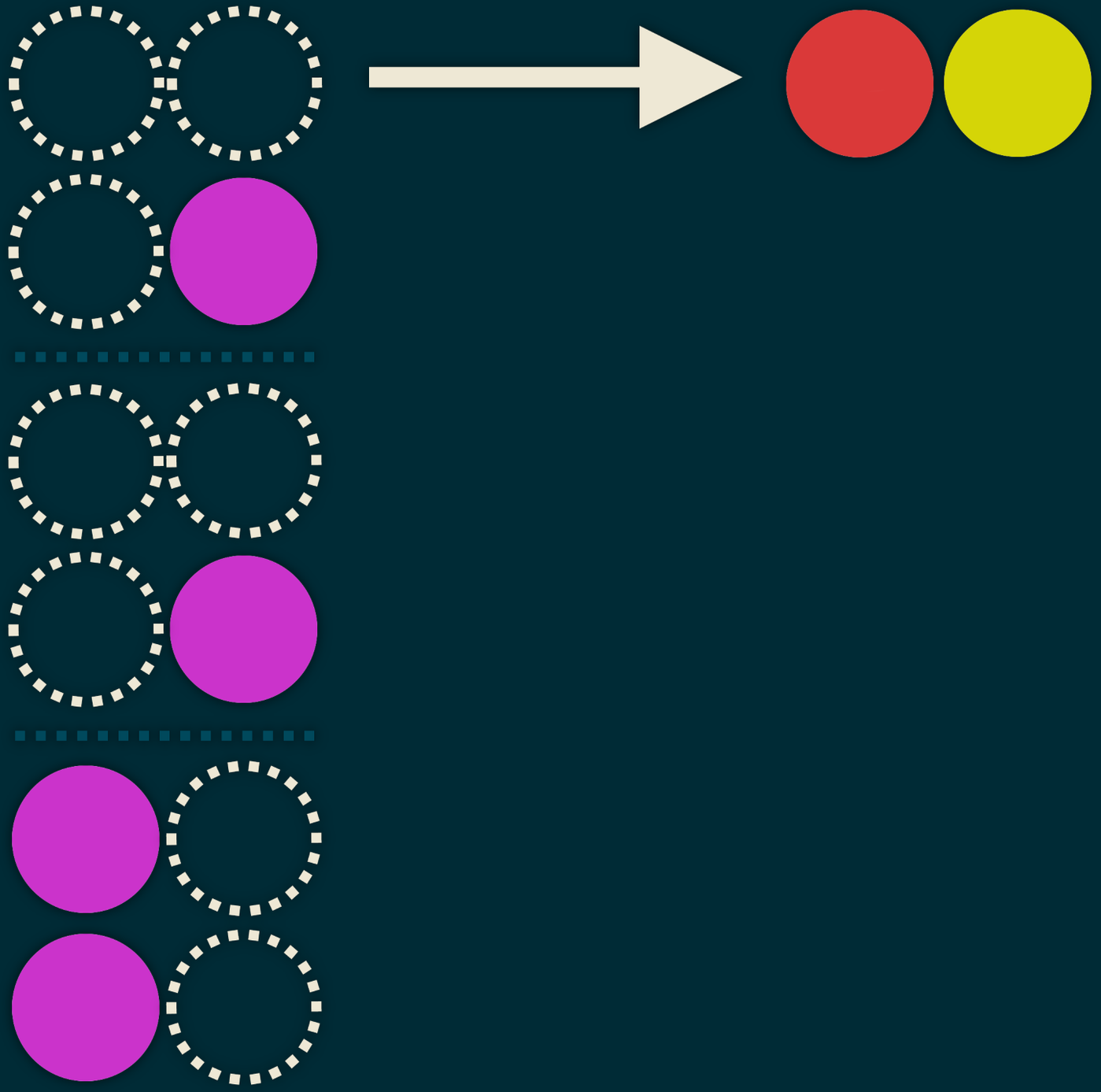
Simplified: `iter_move`

- Define a function `your_iter_namespace::iter_move` that accepts your iterator.
- If you don't, it'll just use `std::move(*i)`.

```cpp
class inner_iterator
{
private:
    using base_value_type = std::ranges::range_value_t<TBase>;
    using base_reference = std::ranges::range_reference_t<TBase>;

public:
    using value_type = std::pair<base_value_type, base_value_type>;
    using reference = std::pair<base_reference, base_reference>;

    friend constexpr auto iter_move(inner_iterator i)
    {
        using base_rref = std::ranges::range_rvalue_reference_t<TBase>;
        return std::pair<base_rref, base_rref> {
            std::ranges::iter_move(i._current_outer),
            std::ranges::iter_move(i._current_inner)
        };
    }
};
```

```cpp
class inner_iterator
{
private:
    using base_value_type = std::ranges::range_value_t<TBase>;
    using base_reference = std::ranges::range_reference_t<TBase>;

public:
    using value_type = std::pair<base_value_type, base_value_type>;
    using reference = std::pair<base_reference, base_reference>;

    friend constexpr auto iter_move(inner_iterator i)
    {
        return std::pair {
            std::ranges::iter_move(i._current_outer),
            std::ranges::iter_move(i._current_inner)
        };
    }
};
```





Views that expose elements multiple times can't use `iter_move`.

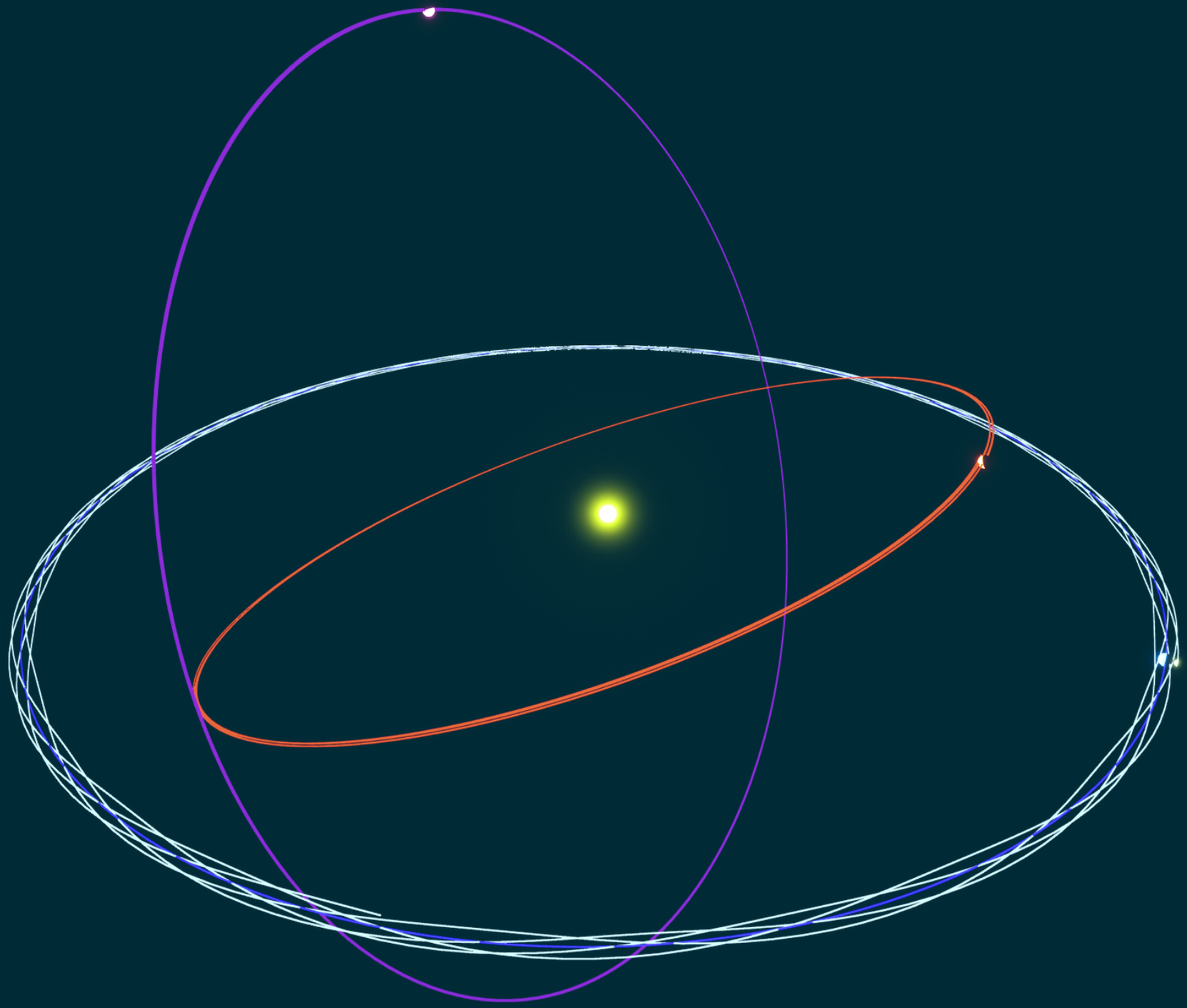
`std::common_reference`

I would skip it. Really. It's not worth the amount of time it would take to explain.

Eric Niebler #BLM · Fri 9:58 AM

Further Reading:

- Standard: wg21.link/meta.trans.other
- Original: wg21.link/p0022 § 4.1.3
- Examples: wg21.link/p2321r0 § 3.1.1, § 3.2.1
- Application: stackoverflow.com/a/66938093/1403459



```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
private:
    TBase _vw;
    class inner_iterator { /* ... */ };
    class inner_sentinel { /* ... */ };
    class inner_view { /* ... */ };

    class outer_iterator { /* ... */ };

public:
    using iterator = outer_iterator;

    all_pairs_view() = default;
    constexpr all_pairs_view(TBase vw);

    [[nodiscard]] constexpr iterator begin() const;
    [[nodiscard]] constexpr iterator end() const;
};
```

```cpp
template <std::ranges::view TBase> requires std::ranges::forward_range<TBase>
class all_pairs_view :
    public std::ranges::view_interface<all_pairs_view<TBase>>
{
    /* ... */

    [[nodiscard]] constexpr iterator begin() const
    {
        return { *this, std::ranges::begin(_vw) };
    }

    [[nodiscard]] constexpr iterator end() const
    {
        return { *this, std::ranges::end(_vw) };
    }
};
```

What about the piping operator?

```
1  using namespace std::views;
2  for (int i : iota(2) | filter(even) | transform(square))
3      std::cout << i << ' ';
```

The piping operator isn't done.

P2387 proposes a fix for C++23.

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

Creating Views

`std::views::reverse` is typically used to create a `reverse_view`.

`std::views::filter` is typically used to create a `filter_view`.

View Parameters

`reverse_view` takes a range and represents it differently (adapts it).

`filter_view` takes a range *and a function* and adapts the range.

Piping Views

`std::views::reverse` can be used in a pipeline.

`std::views::filter` needs a function to be used in a pipeline.

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

Range Adaptor Closure Objects [24.7.2.1]

A *range adaptor closure object* is a unary function object that accepts a `viewable_range` argument and returns a `view`. For a range adaptor closure object `C` and an expression `R` such that `decltype((R))` models `viewable_range`, the following expressions are equivalent and yield a `view`:

- `C(R)`
- `R | C`

Range Adaptor Closure Objects [24.7.2.1]

A *range adaptor closure object* is a unary function object that accepts a `viewable_range` argument and returns a `view`. For a range adaptor closure object `C` and an expression `R` such that `decltype((R))` models `viewable_range`, the following expressions are equivalent and yield a `view`:

- `C(R)`
- `R | C`

Range Adaptor Closure Objects [24.7.2.1]

A *range adaptor closure object* is a unary function object that accepts a `viewable_range` argument and returns a `view`. For a range adaptor closure object `C` and an expression `R` such that `decltype((R))` models `viewable_range`, the following expressions are equivalent and yield a `view`:

- `C(R)`
- `R | C`

Range Adaptor Closure Objects [24.7.2.1]

Given an additional range adaptor closure object `D`, the expression `C | D` produces another range adaptor closure object `E`.

Simplification: The following expressions are equivalent:

- `E(R)`
- `D(C(R))`
- `R | C | D`
- `R | (C | D)`

Range Adaptor Closure Objects [24.7.2.1]

Given an additional range adaptor closure object `D`, the expression `C | D` produces another range adaptor closure object `E`.

Simplification: The following expressions are equivalent:

- `E(R)`
- `D(C(R))`
- `R | C | D`
- `R | (C | D)`

Range Adaptor Closure Objects [24.7.2.1]

Given an additional range adaptor closure object `D`, the expression `C | D` produces another range adaptor closure object `E`.

Simplification: The following expressions are equivalent:

- `E(R)`
- `D(C(R))`
- `R | C | D`
- `R | (C | D)`

Range Adaptor Objects [24.7.2.2-4]

A *range adaptor object* is a customization point object that accepts a `viewable_range` as its first argument and returns a `view`.

If a range adaptor object accepts only one argument, then it is a range adaptor closure object.

Range Adaptor Objects [24.7.2.2-4]

A *range adaptor object* is a customization point object that accepts a `viewable_range` as its first argument and returns a `view`.

If a range adaptor object accepts only one argument, then it is a range adaptor closure object.

Range Adaptor Objects [24.7.2.2-4]

A *range adaptor object* is a customization point object that accepts a `viewable_range` as its first argument and returns a `view`.

If a range adaptor object accepts only one argument, then it is a range adaptor closure object.

Range Adaptor Objects [24.7.2.2-4]

If a range adaptor object `adaptor` accepts more than one argument, then:

[**Simplification**] let `args...` be arguments such that the expression `adaptor(R, args...)` produces a `view`. In this case, `adaptor(args...)` is a range adaptor closure object.

Range Adaptor Objects [24.7.2.2-4]

If a range adaptor object `adaptor` accepts more than one argument, then:

[**Simplification**] let `args...` be arguments such that the expression `adaptor(R, args...)` produces a `view`. In this case, `adaptor(args...)` is a range adaptor closure object.

Range Adaptor Objects [24.7.2.2-4]

If a range adaptor object `adaptor` accepts more than one argument, then:

[**Simplification**] let `args...` be arguments such that the expression `adaptor(R, args...)` produces a `view`. In this case, `adaptor(args...)` is a range adaptor closure object.

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

```
1  constexpr auto reversed_evens = std::views::reverse | std::views::filter(even);
2  auto const x = some_range | reversed_evens;
```

Why can't we customize this?

Range Adaptor Object Problems

- Standard doesn’t provide a common type for composition.
- Implementations use a common type to compose closure objects.
- You can’t use the type because it’s unspecified.

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::_Pipe::_Base<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::_Pipe::_Base<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::_Pipe::_Base<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

Solution (P2387)

- New type: `std::ranges::range_adaptor_closure<T>`.
- Ensure all closure objects inherit from this.
- Custom closure objects can then use it, too.

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::range_adaptor_closure<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::range_adaptor_closure<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::range_adaptor_closure<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

```cpp
namespace detail
{
template <class T>
concept can_construct_all_pairs_view = requires(T&& t)
{
    all_pairs_view{ static_cast<T&&>(t) };
};

struct all_pairs_fn : public std::ranges::range_adaptor_closure<all_pairs_fn>
{
    template <std::ranges::viewable_range Rng>
    [[nodiscard]] constexpr auto operator()(Rng&& rng) const
        requires can_construct_all_pairs_view<Rng>
    {
        return all_pairs_view{ std::forward<Rng>(rng) };
    }
};
} // namespace detail

inline constexpr detail::all_pairs_fn all_pairs;
```

Topics Lessons

Topics

`all_pairs_view`

Lessons

- Views are tricky to write but easy to use.
- C++20 isn't *quite* ready for custom views.
- range-v3 may help until closures are fixed.

Topics	Lessons
`all_pairs_view`	• Views are tricky to write but easy to use. • C++20 isn't *quite* ready for custom views. • range-v3 may help until closures are fixed.
`vector<bool>`, proxy iterators	• Proxies are unintuitive, but valid in C++20. • Create a new world of novel iterator types. • Keep them in mind in your generic code.

Topics	Lessons
`all_pairs_view`	• Views are tricky to write but easy to use. • C++20 isn't *quite* ready for custom views. • range-v3 may help until closures are fixed.
`vector<bool>`, proxy iterators	• Proxies are unintuitive, but valid in C++20. • Create a new world of novel iterator types. • Keep them in mind in your generic code.
`iter_swap`, `iter_move`	• Customization points are powerful. • Easy to use and (usually) easy to customize. • Prefer `std::ranges` algorithms.

Thank you.