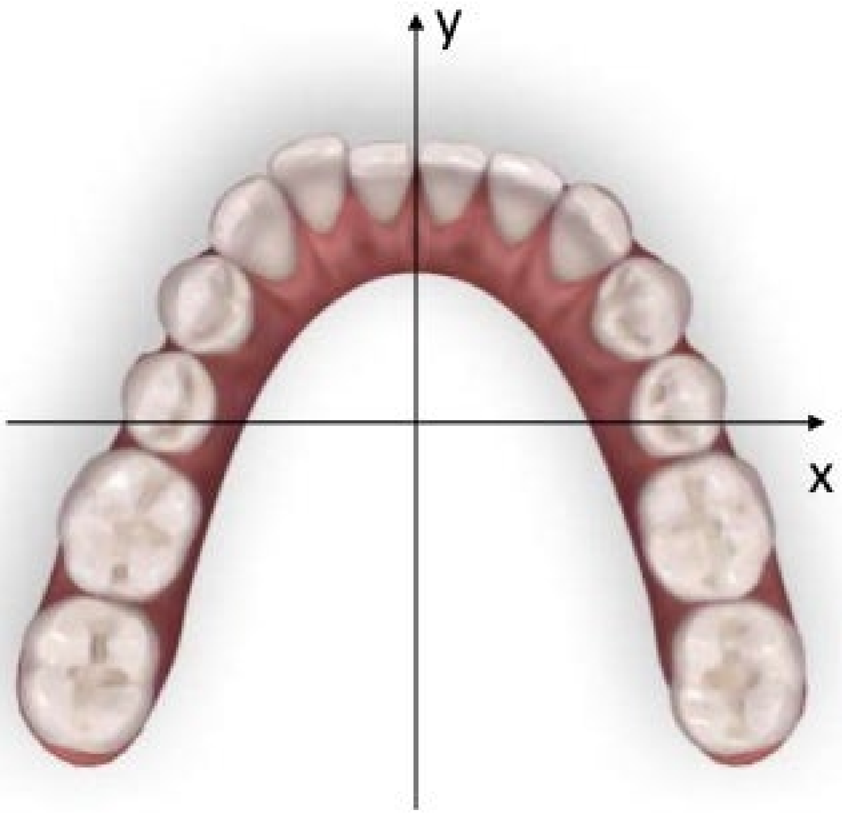
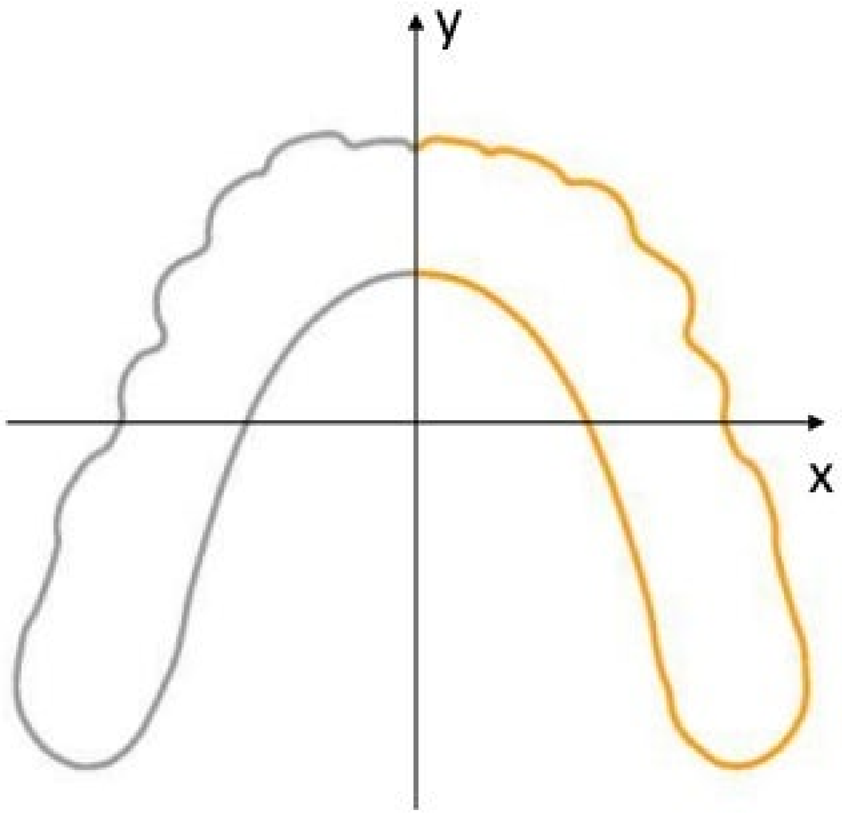
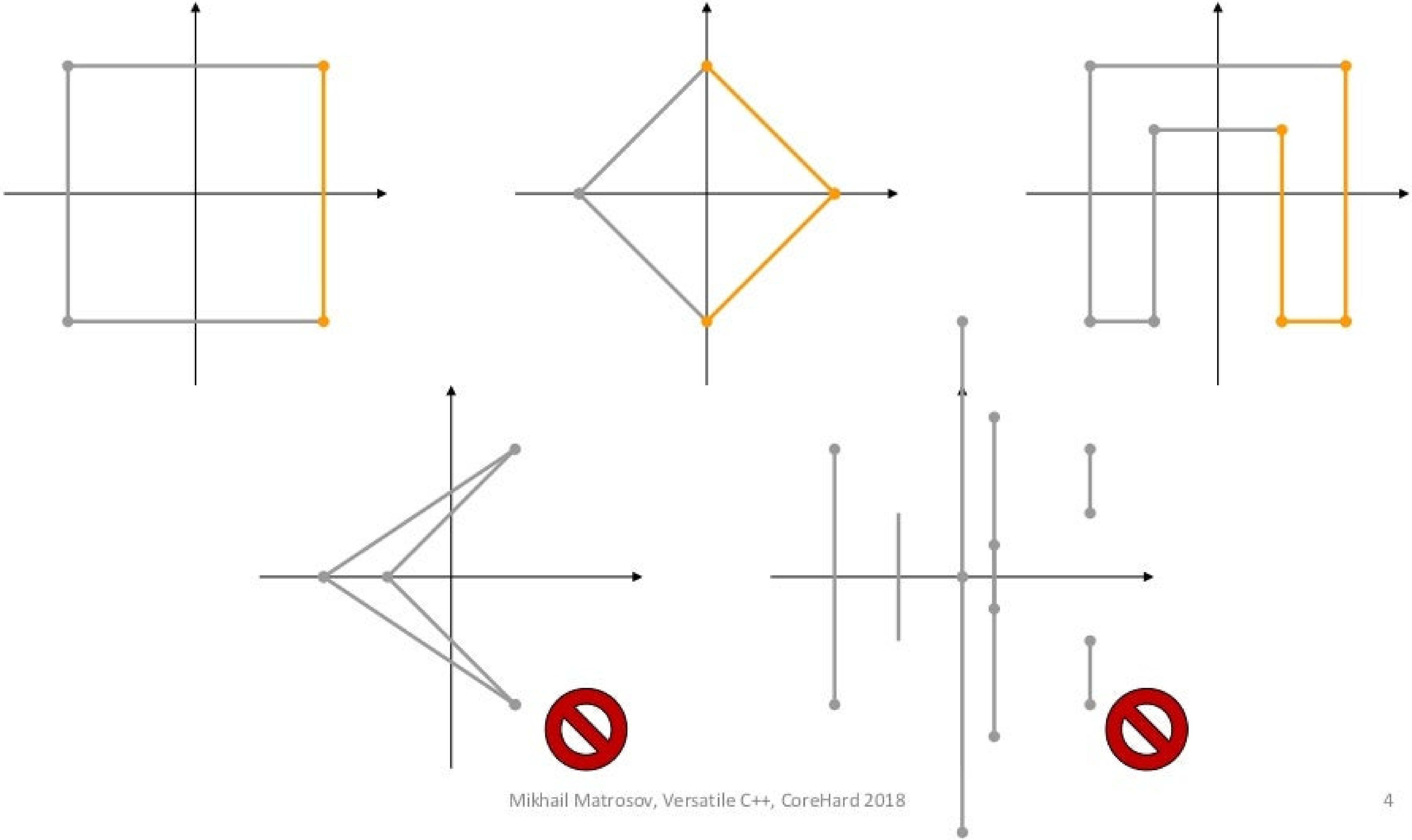
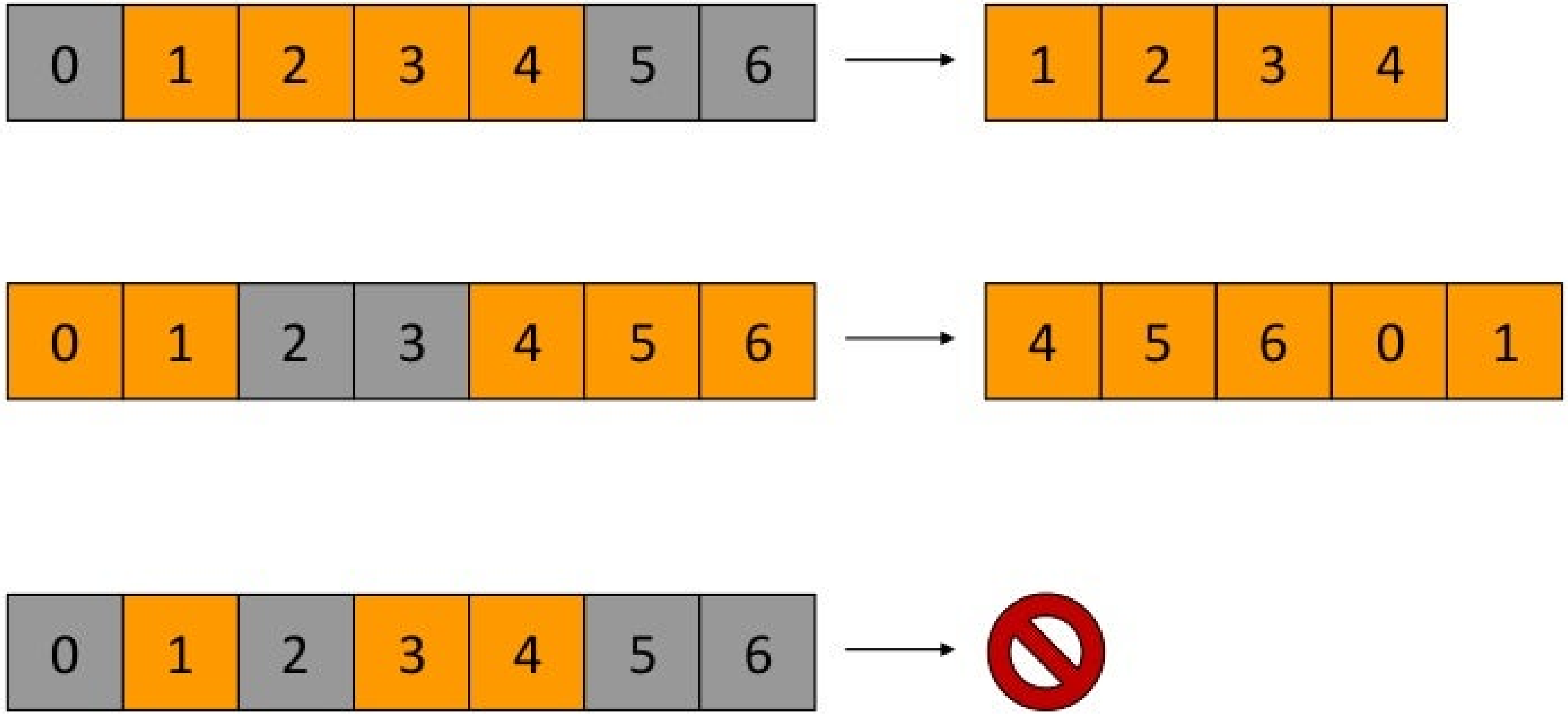
Versatile C++

Mikhail Matrosov, Expert Software Engineer, Align Technology


y
x


y
x





0	1	2	3	4	5	6

→

1	2	3	4

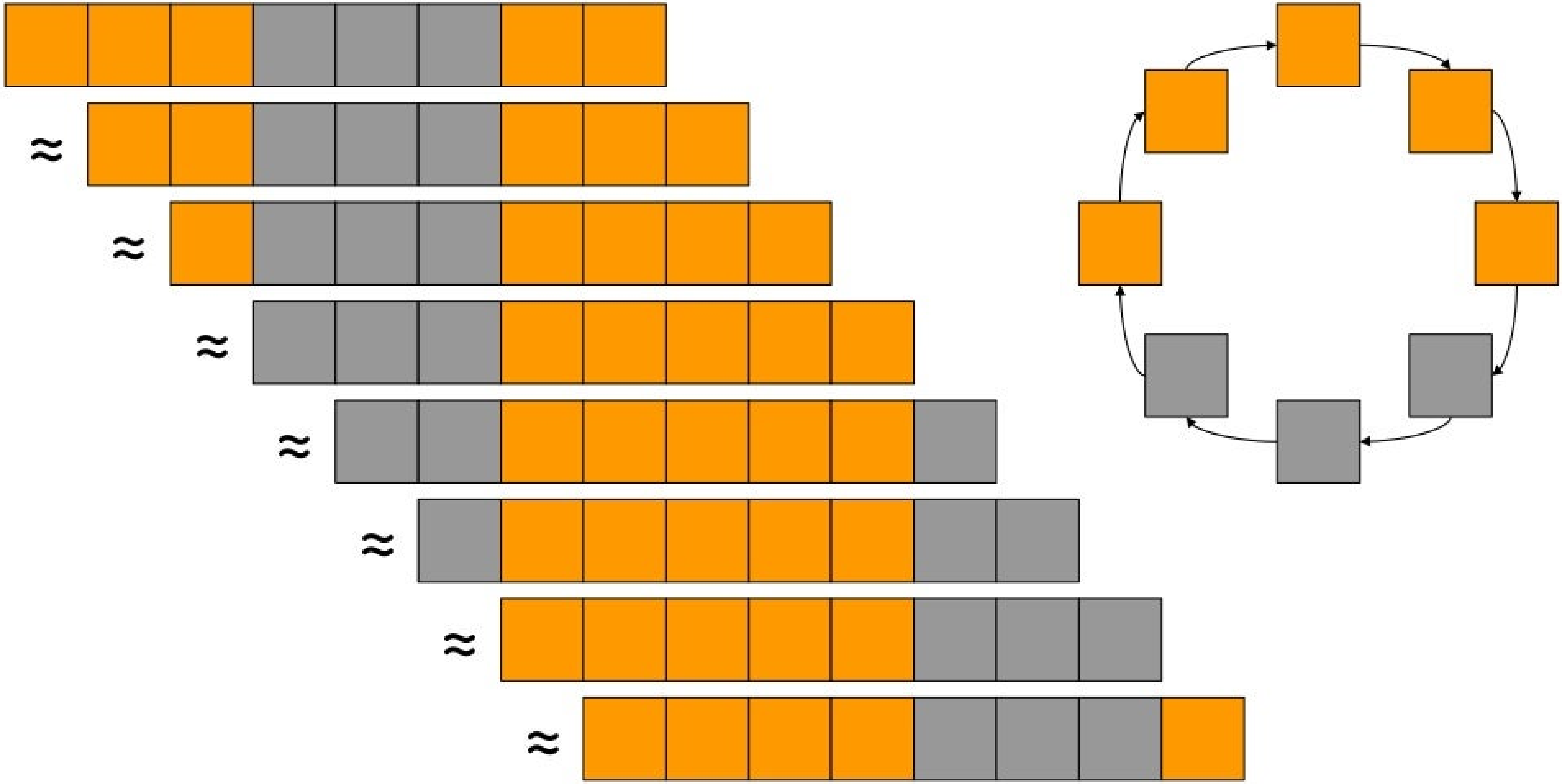
0	1	2	3	4	5	6

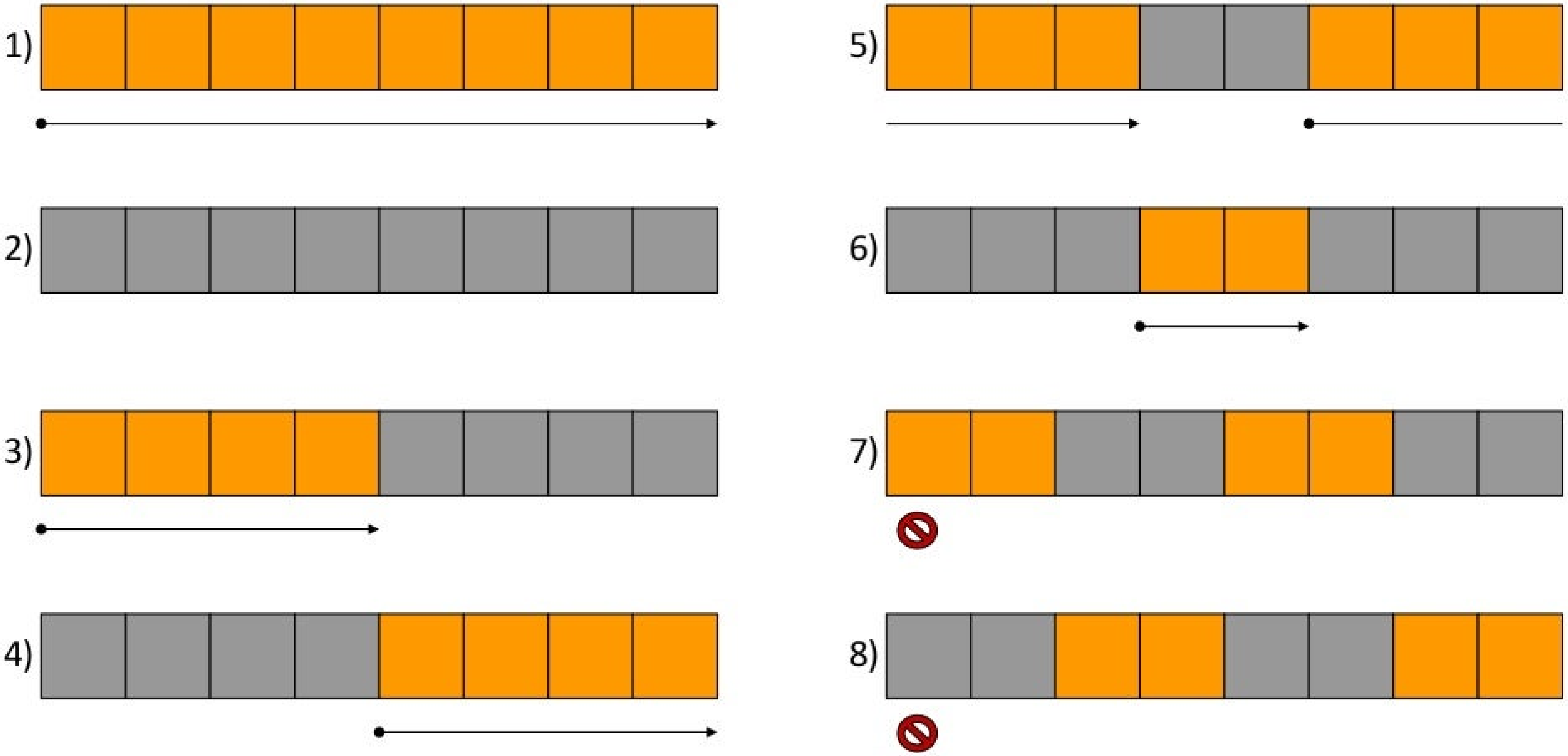
→

4	5	6	0	1

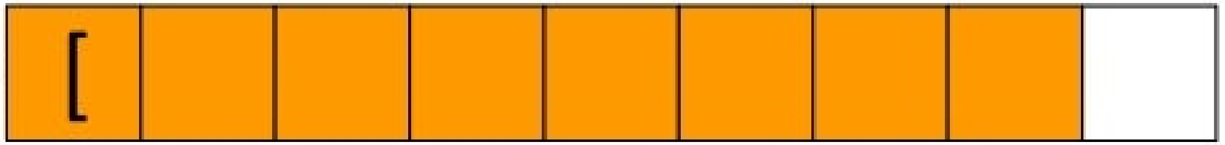
0	1	2	3	4	5	6

→ 🚫

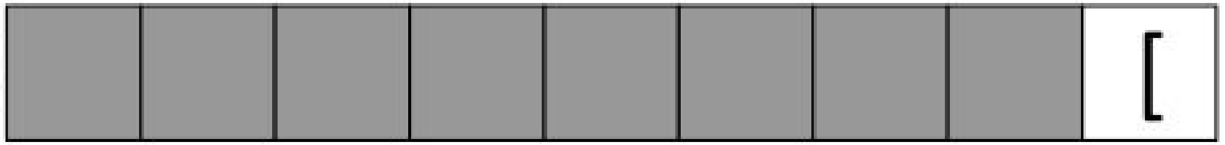



1)
2)
3)
4)
5)
6)
7)
8)

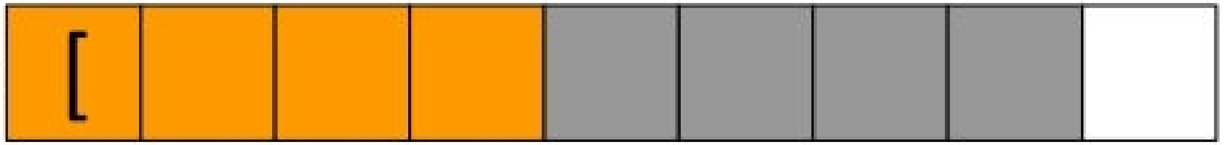
1)



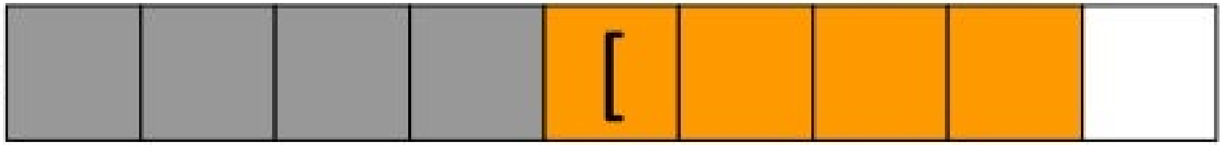
2)



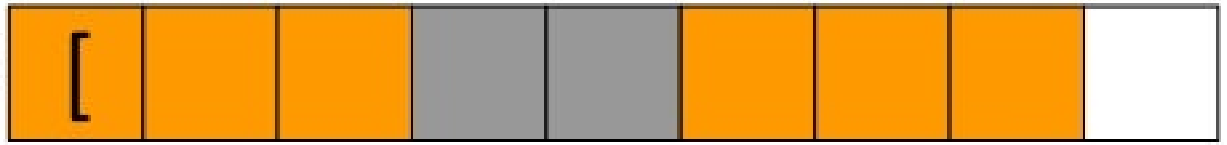
3)



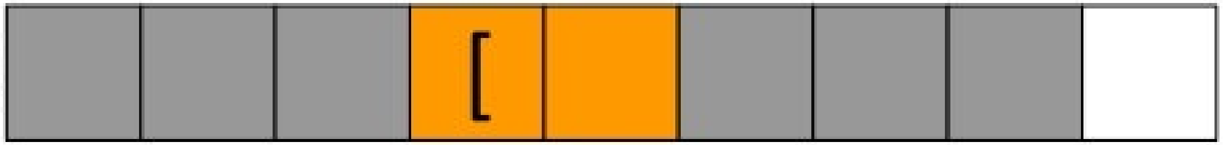
4)



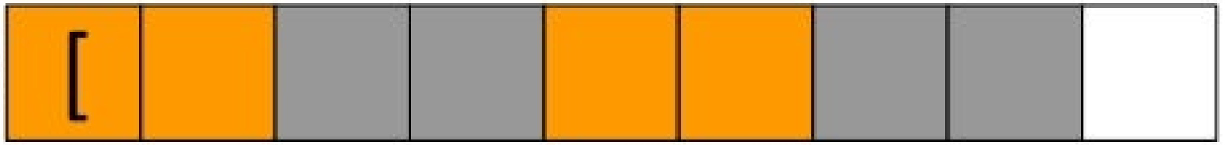
5)



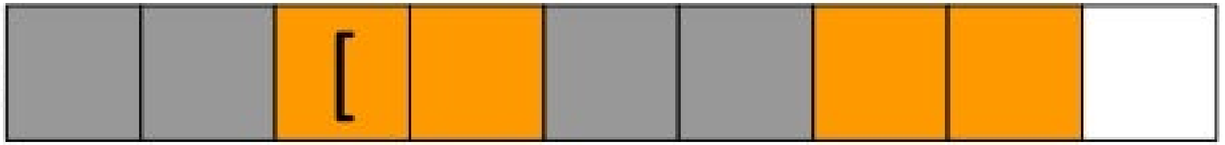
6)



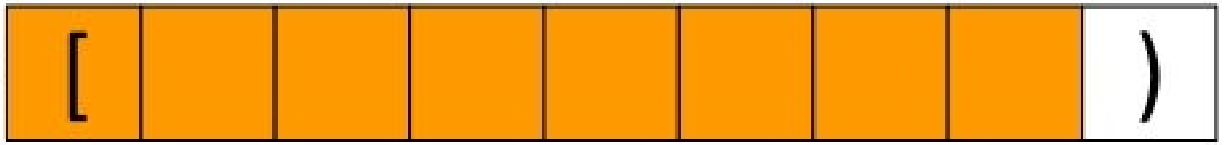
7)



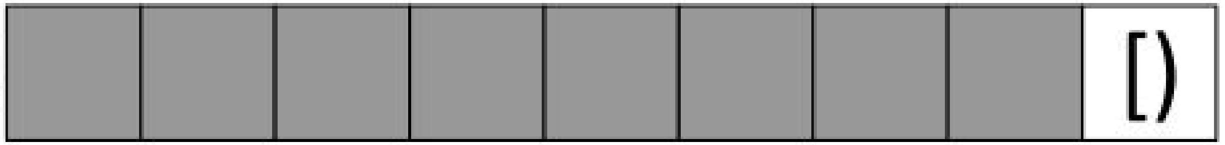
8)



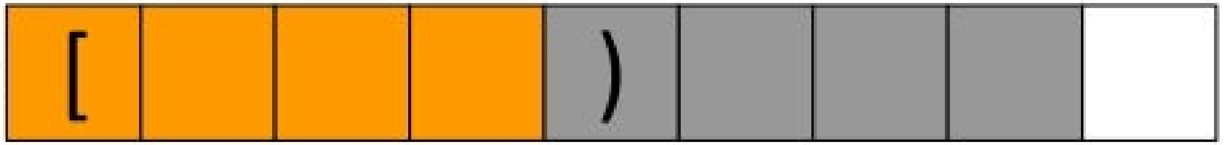
1)



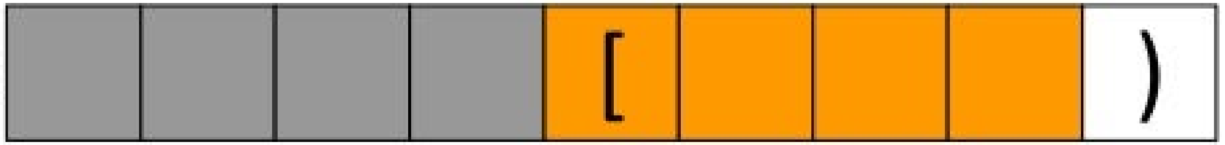
2)



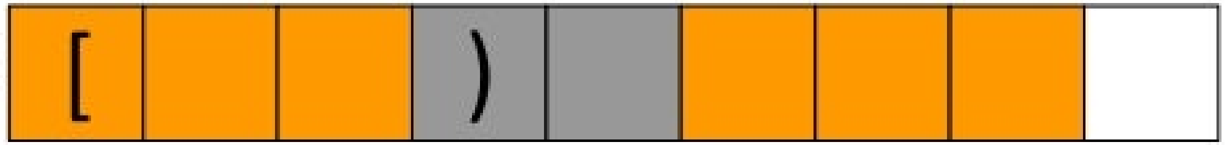
3)



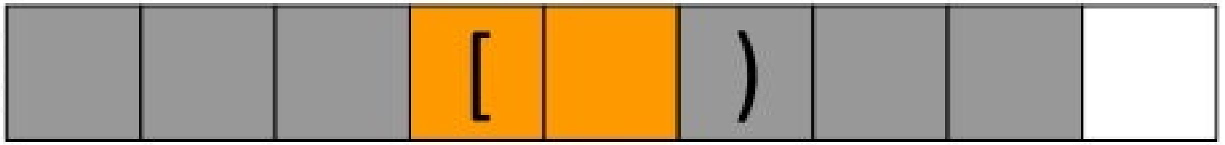
4)



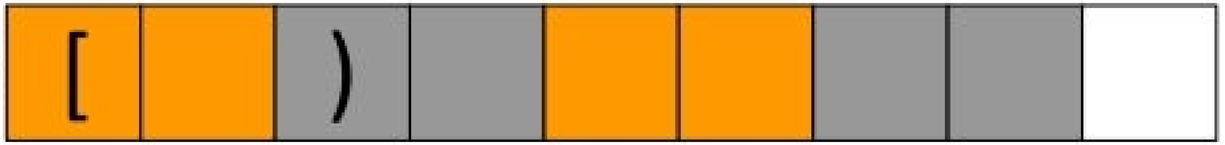
5)



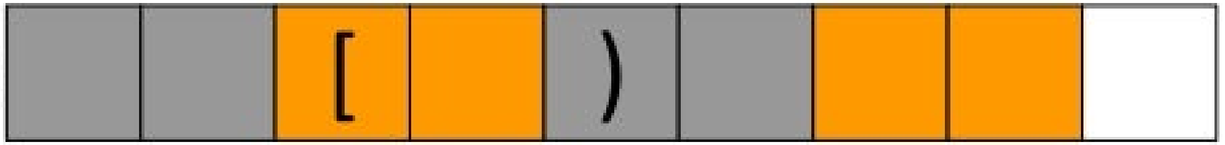
6)



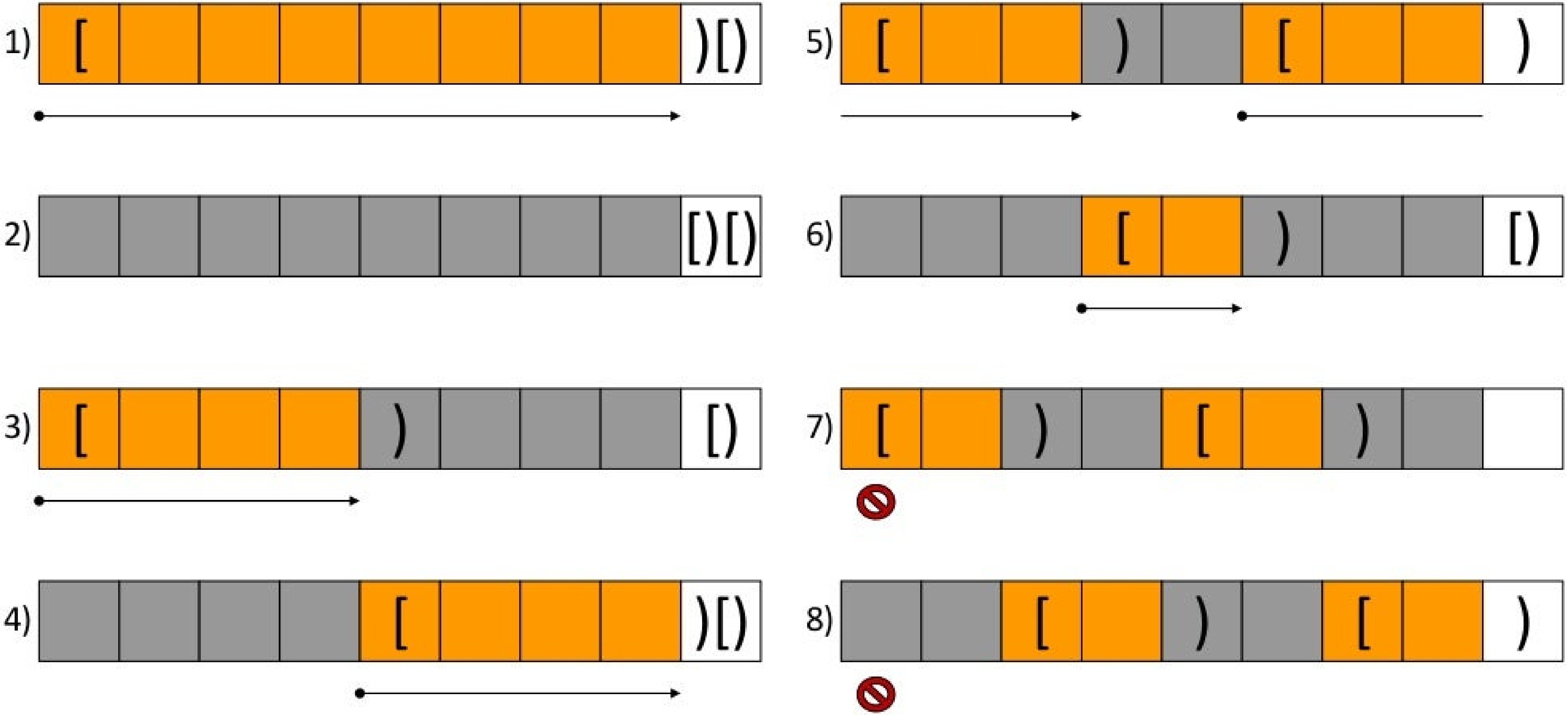
7)

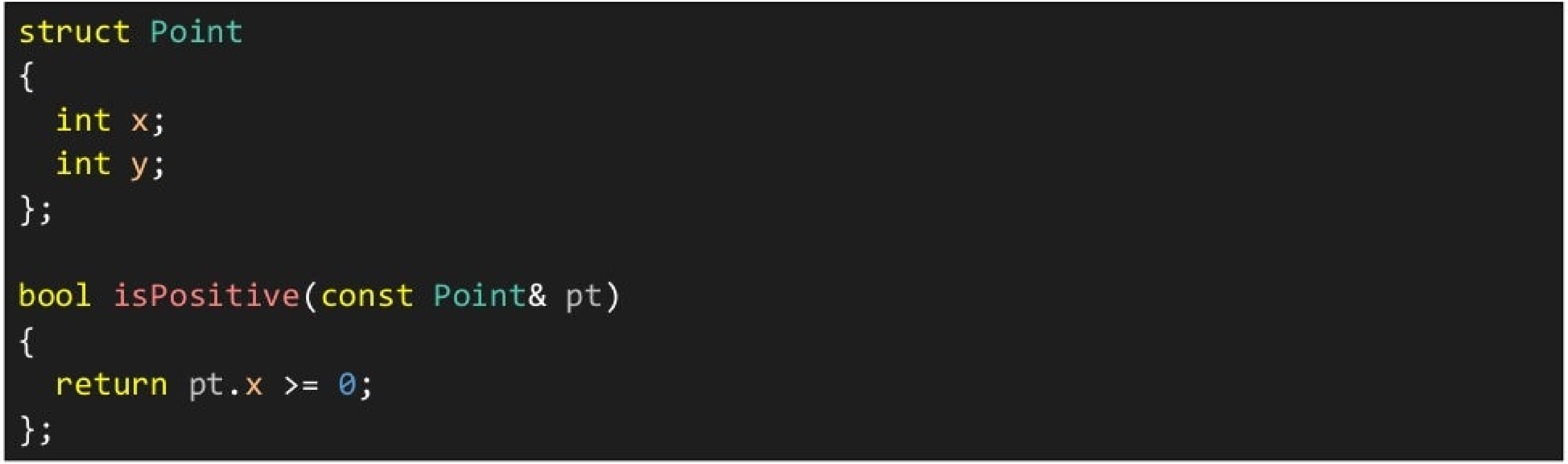


8)



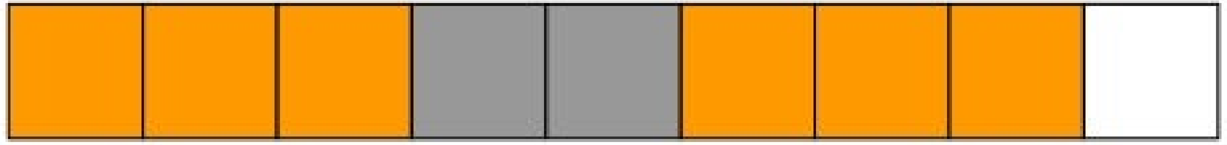
1)	🟧 [	🟧	🟧	🟧	🟧	🟧	🟧	🟧	⬜)[
2)	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬜ [)[
3)	🟧 [	🟧	🟧	🟧	⬛)	⬛	⬛	⬛	⬜ [
4)	⬛	⬛	⬛	⬛	🟧 [	🟧	🟧	🟧	⬜)[
5)	🟧 [	🟧	🟧	⬛)	⬛	🟧 [	🟧	🟧	⬜
6)	⬛	⬛	⬛	🟧 [	🟧	⬛)	⬛	⬛	⬜ [
7)	🟧 [	🟧	⬛)	⬛	🟧 [	🟧	⬛	⬛	⬜
8)	⬛	⬛	🟧 [	🟧	⬛)	⬛	🟧 [	🟧	⬜


1)
[
)[)
2)
[)[)
3)
[
)
[)
4)
[
)[)
5)
[
)
[
)
6)
[
)
[)
7)
[
)
[
)
8)
[
)
[
)

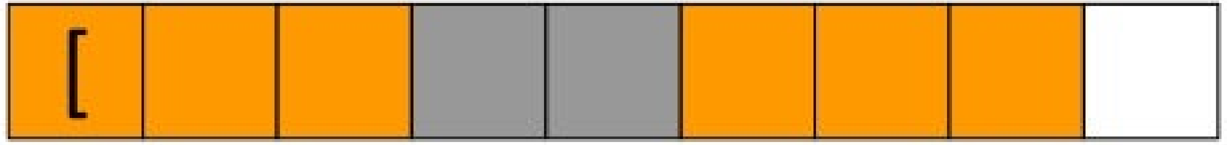


```cpp
struct Point
{
  int x;
  int y;
};

bool isPositive(const Point& pt)
{
  return pt.x >= 0;
};
```

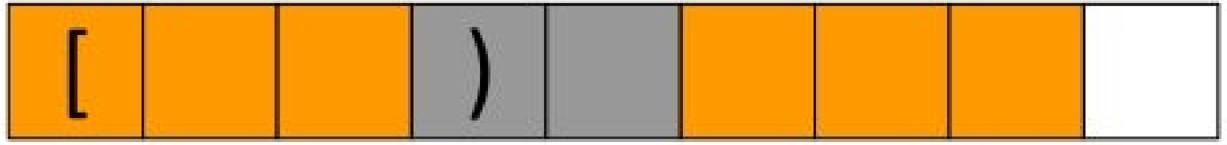
```
std::vector<Point> extract(const std::vector<Point>& points)
{
```



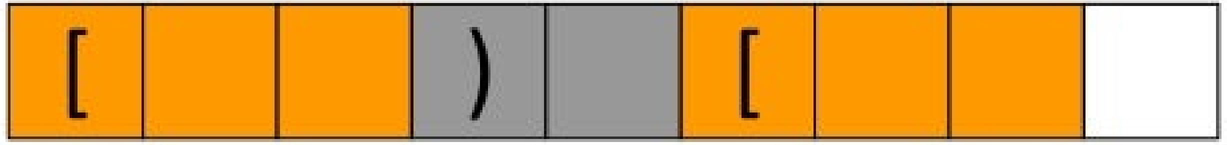
```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
```



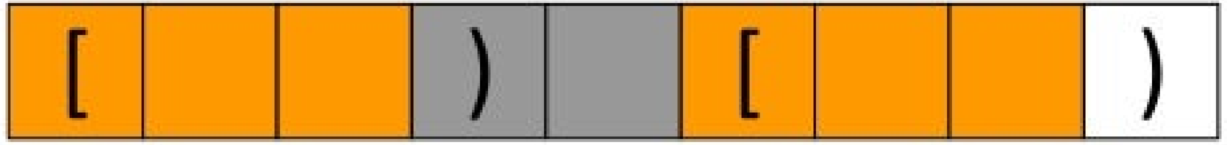
```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
```



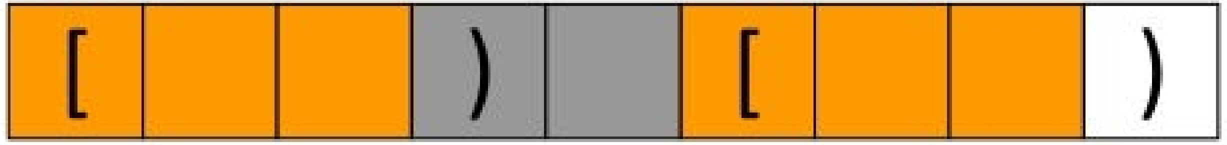
```cpp
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
```

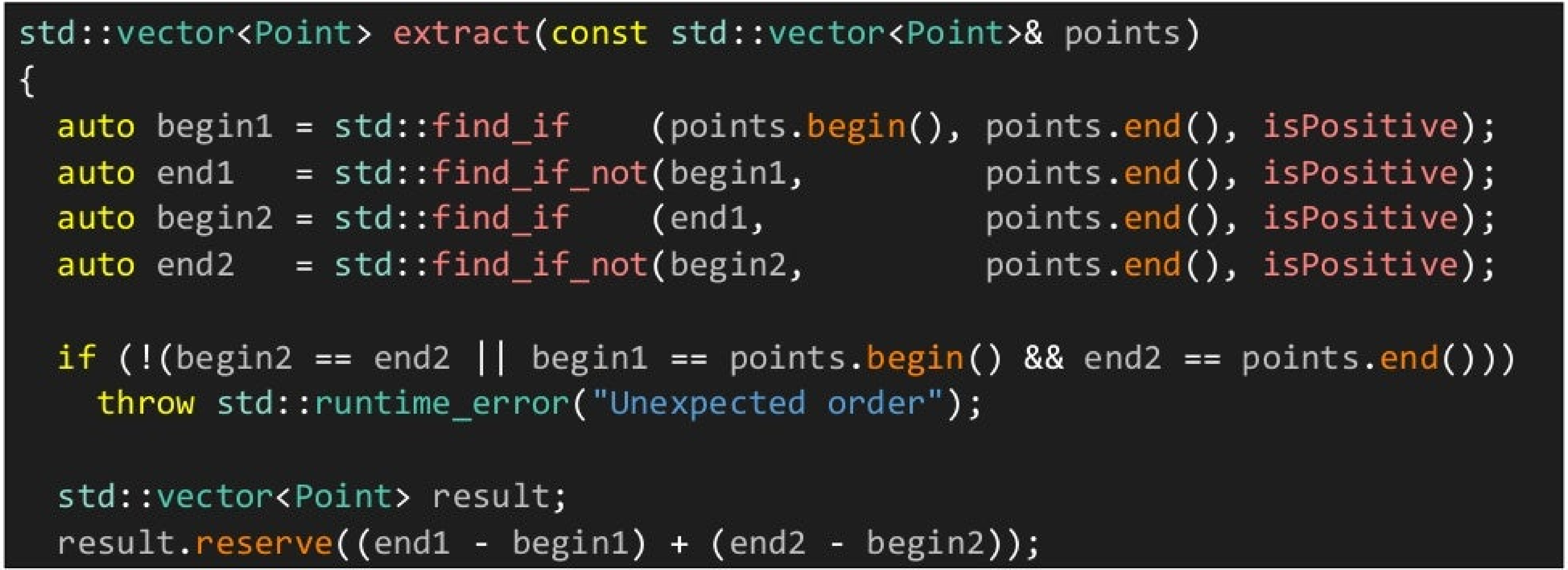


```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);
```

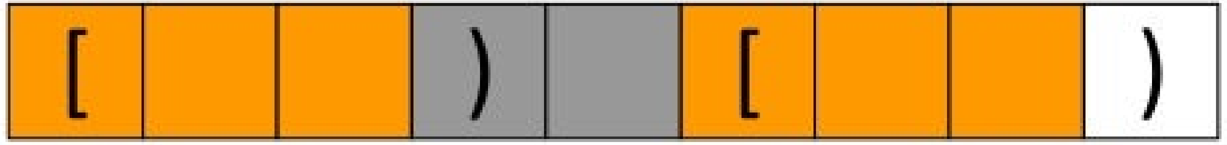


```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");
```

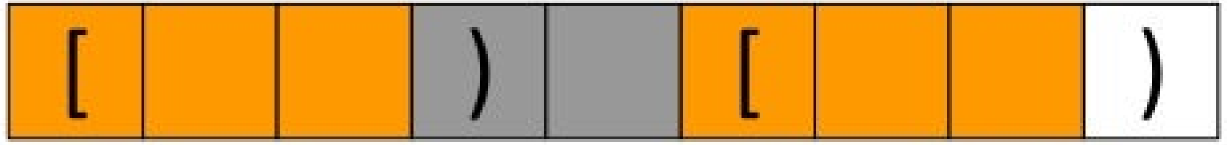




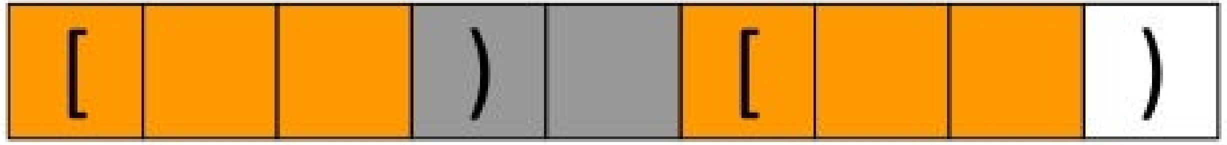
```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  std::vector<Point> result;
  result.reserve((end1 - begin1) + (end2 - begin2));
```



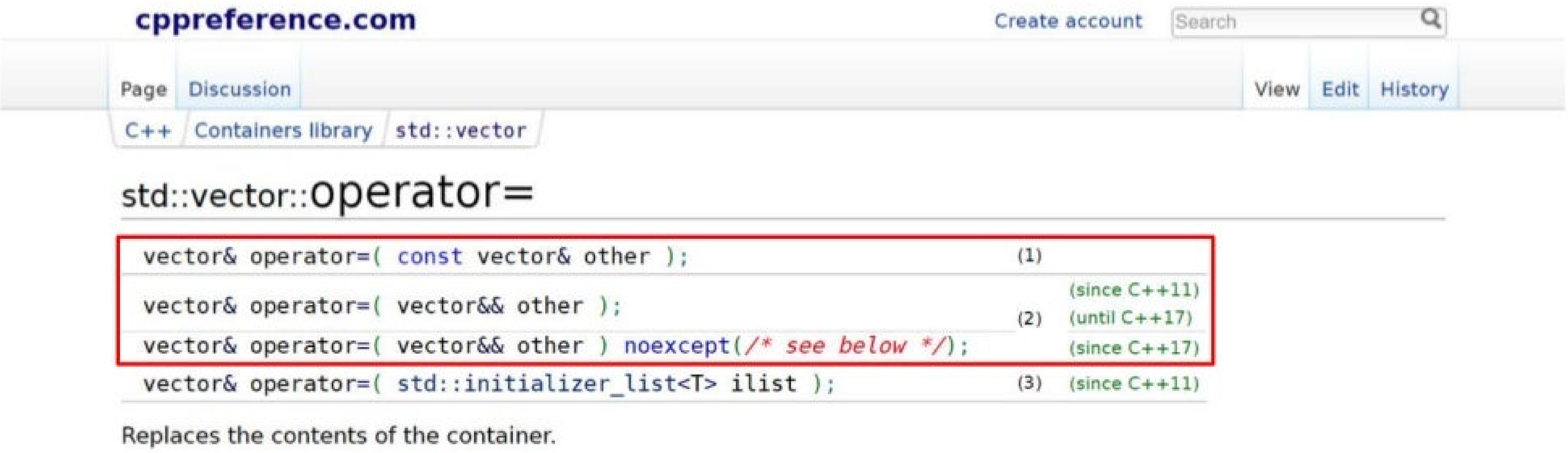
```cpp
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  std::vector<Point> result;
  result.reserve((end1 - begin1) + (end2 - begin2));
  result.insert(result.end(), begin2, end2);
  result.insert(result.end(), begin1, end1);
```



```
std::vector<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  std::vector<Point> result;
  result.reserve((end1 - begin1) + (end2 - begin2));
  result.insert(result.end(), begin2, end2);
  result.insert(result.end(), begin1, end1);

  return result;
}
```



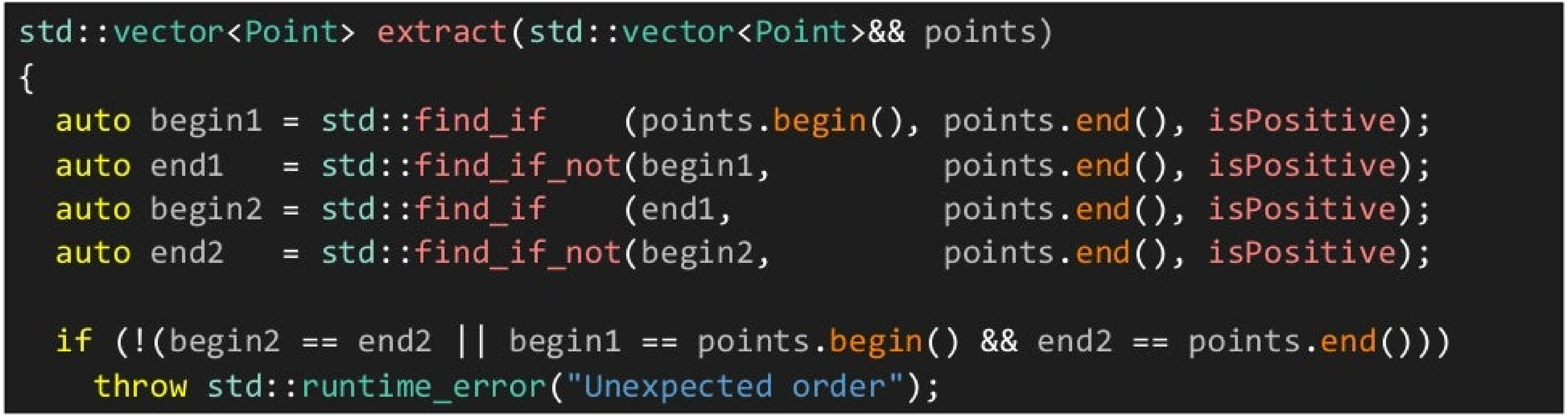
```
std::vector<Point> extract(std::vector<Point>&& points)
{
```



std::vector::operator=

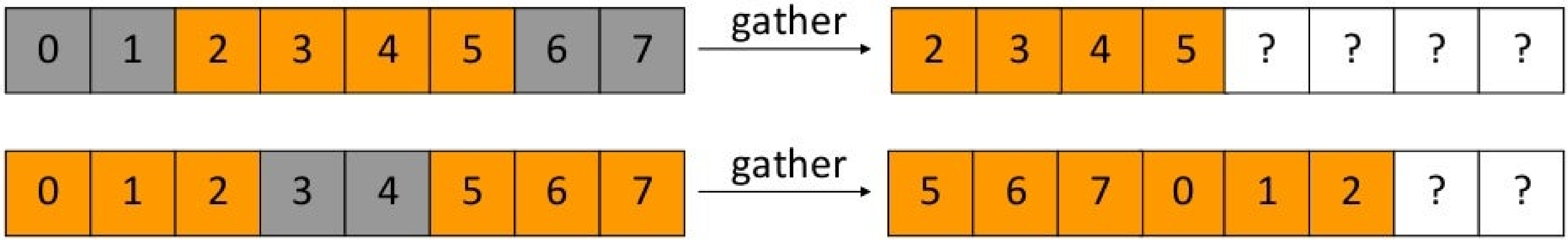
`vector& operator=( const vector& other );`	(1)	
`vector& operator=( vector&& other );`	(2)	(since C++11) (until C++17)
`vector& operator=( vector&& other ) noexcept(/* see below */);`		(since C++17)
`vector& operator=( std::initializer_list<T> ilist );`	(3)	(since C++11)

Replaces the contents of the container.

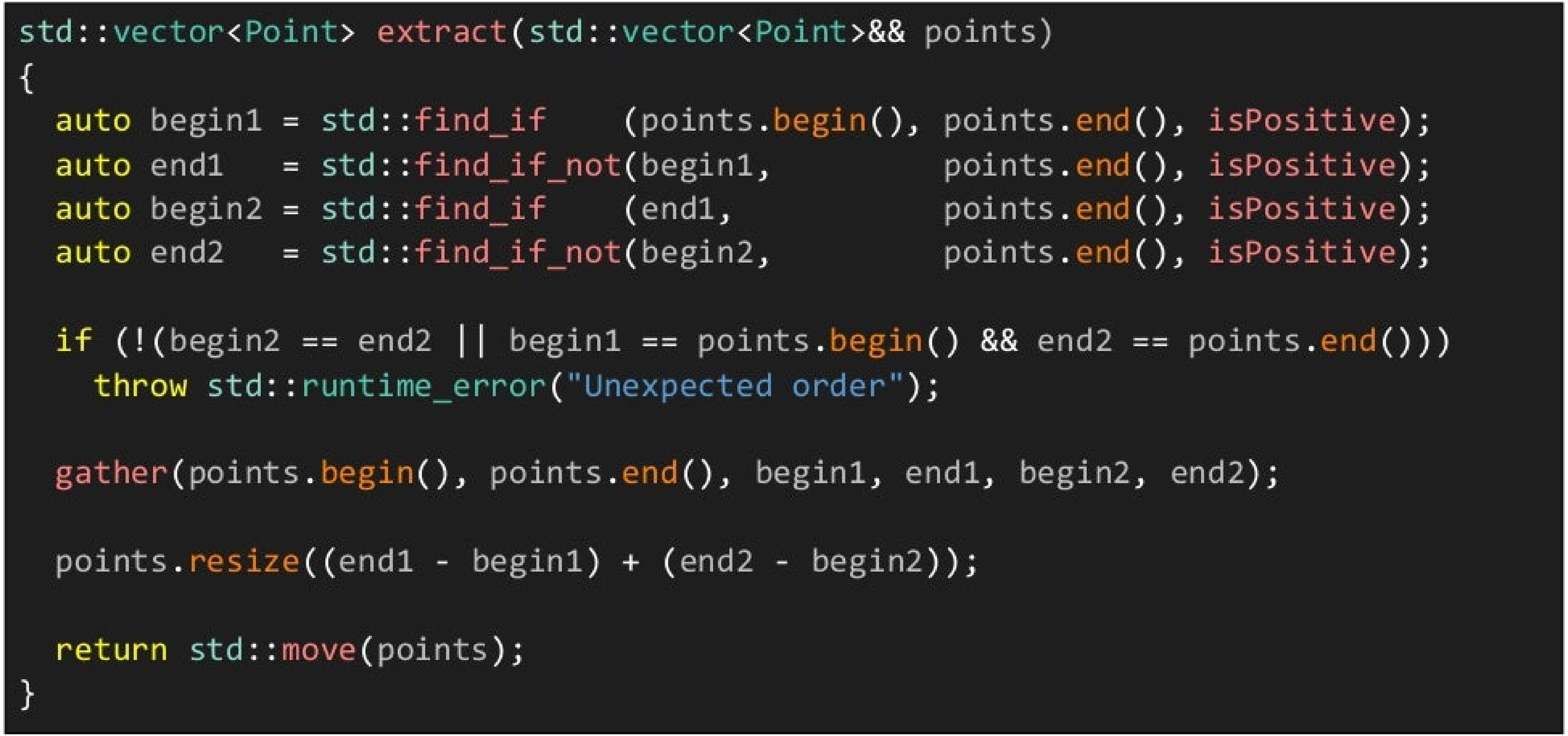


```
std::vector<Point> extract(std::vector<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");
```

```
std::vector<Point> extract(std::vector<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  gather(points.begin(), points.end(), begin1, end1, begin2, end2);
```



```
std::vector<Point> extract(std::vector<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  gather(points.begin(), points.end(), begin1, end1, begin2, end2);

  points.resize((end1 - begin1) + (end2 - begin2));
```

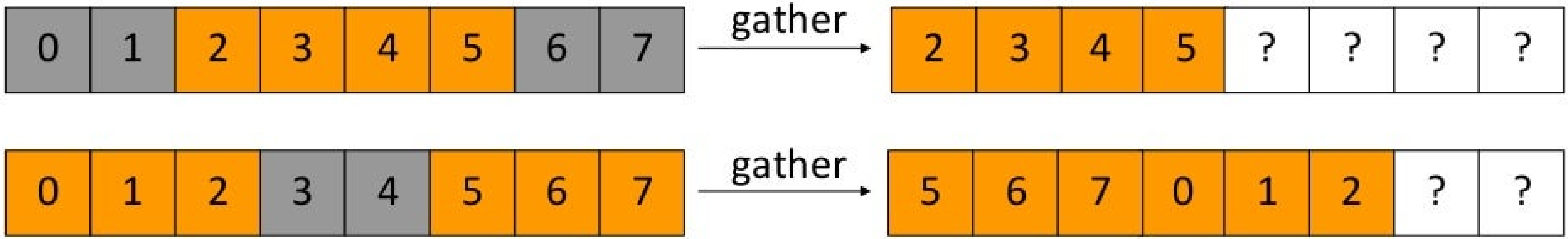


```cpp
std::vector<Point> extract(std::vector<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  gather(points.begin(), points.end(), begin1, end1, begin2, end2);

  points.resize((end1 - begin1) + (end2 - begin2));

  return std::move(points);
}
```

Never apply std::move after return…
Always apply std::move on r-value references!

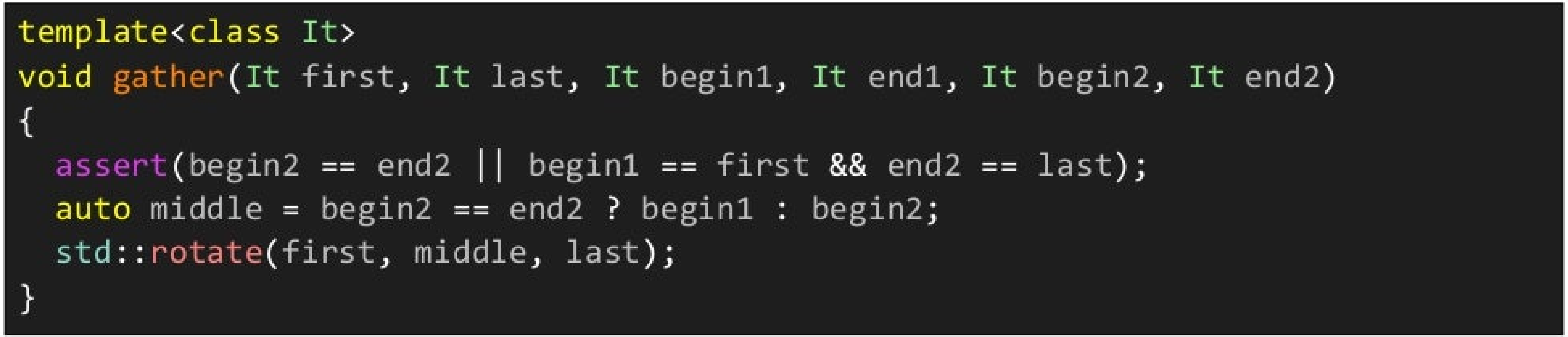
[c++11 Return value optimization or move?]

```
template<class It>
void gather(It first, It last, It begin1, It end1, It begin2, It end2)
{
  assert(begin2 == end2 || begin1 == first && end2 == last);
```

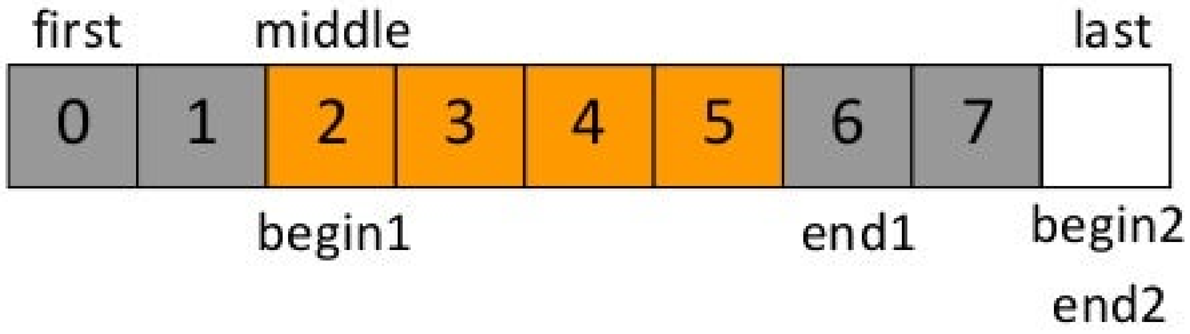


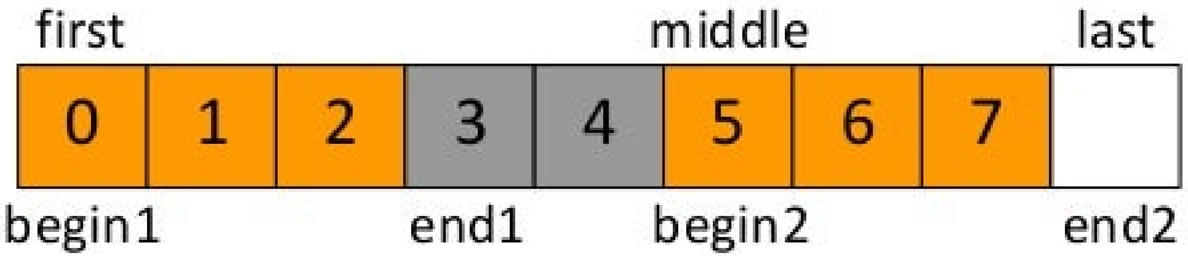
0 1 2 3 4 5 6 7 → gather → 2 3 4 5 ? ? ? ?

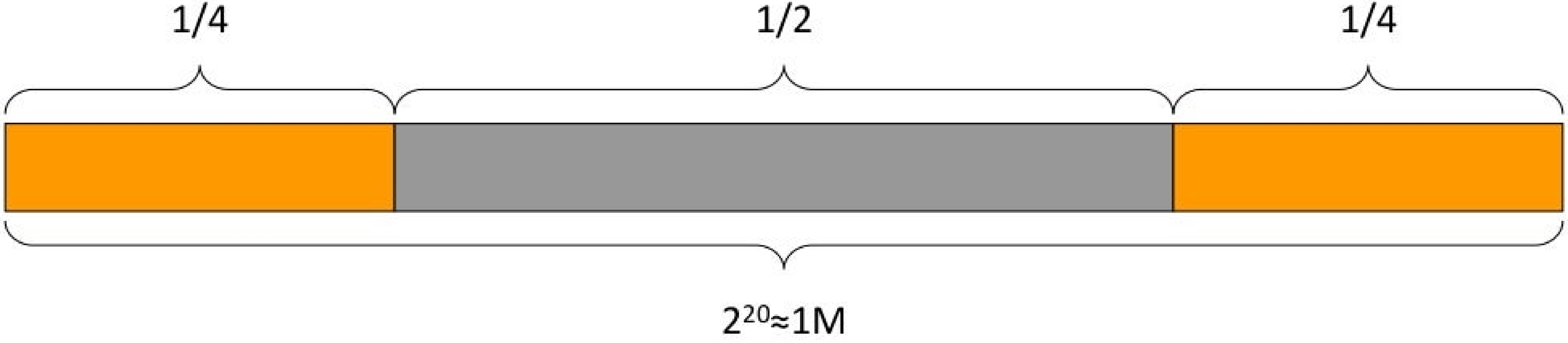
0 1 2 3 4 5 6 7 → gather → 5 6 7 0 1 2 ? ?

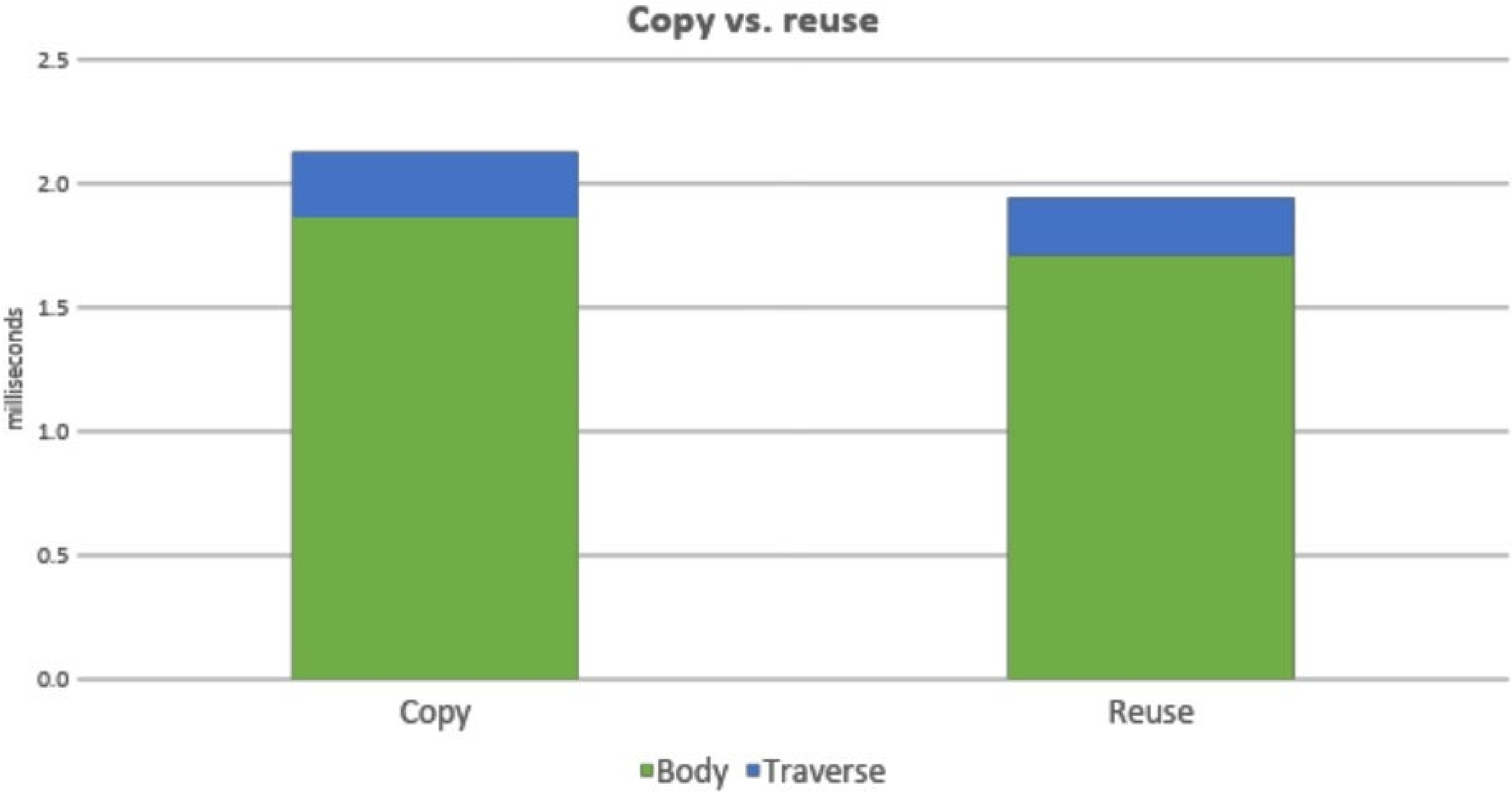


```
template<class It>
void gather(It first, It last, It begin1, It end1, It begin2, It end2)
{
  assert(begin2 == end2 || begin1 == first && end2 == last);
  auto middle = begin2 == end2 ? begin1 : begin2;
  std::rotate(first, middle, last);
}
```





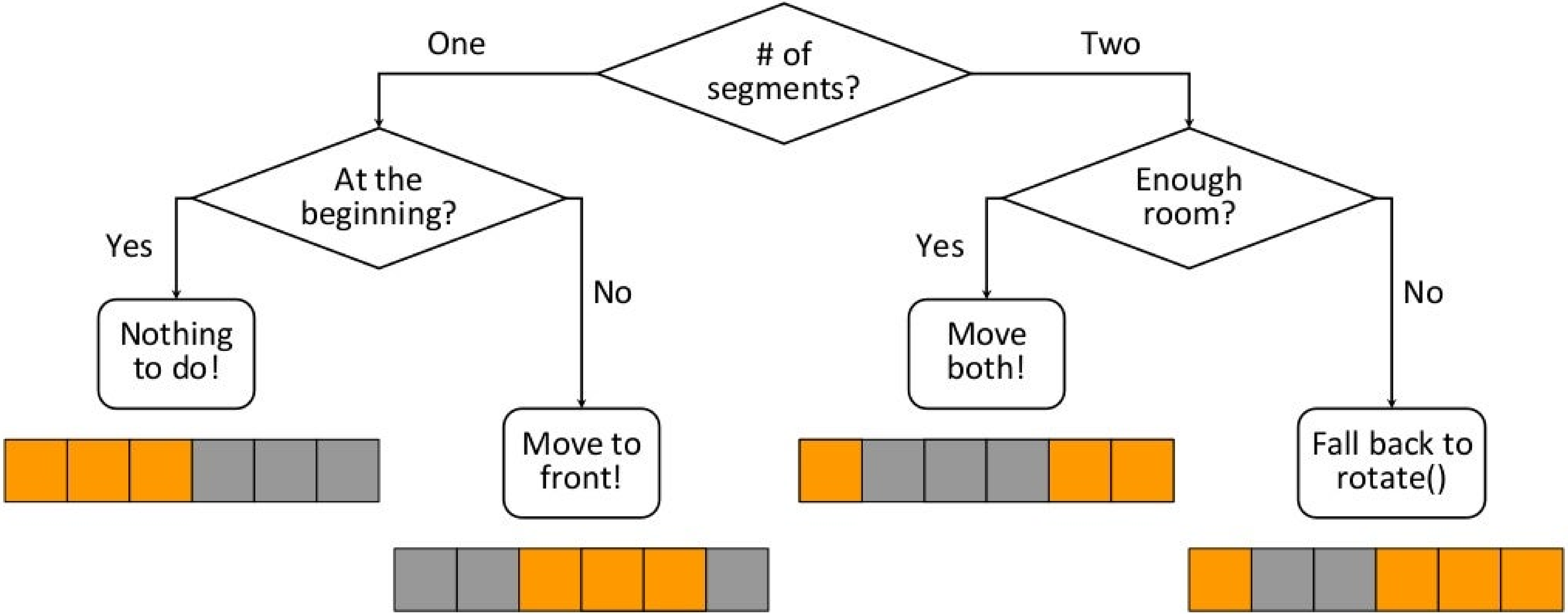

1/4
1/2
1/4
2²⁰≈1M


Copy vs. reuse
2.5
2.0
1.5
1.0
0.5
0.0
milliseconds
Copy
Reuse
🟩Body 🟦Traverse

cmake ../
/usr/bin/clang++
-- broken
sudo update-alternatives
--list clang
/usr/bin/clang++-6.0
-stdlib=libc++ -v

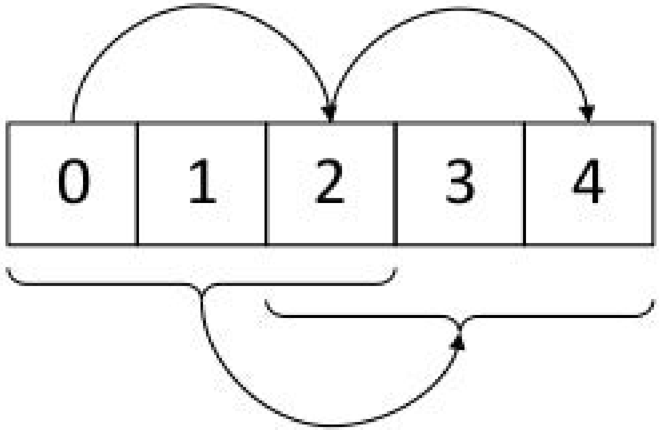
```
Selected GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/7.3.0
Candidate multilib: .;@m64
Selected multilib: .;@m64
clang -cc1 version 6.0.0 based upon LLVM 6.0.0 default target x86_64-pc-linux-gnu
ignoring nonexistent directory "/include"
ignoring duplicate directory "/usr/include/clang/6.0.0/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/include/c++/v1
 /usr/include/clang/6.0.0/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
../../clang_test/hello.cpp:2:10: fatal error: 'experimental/generator' file not found
#include <experimental/generator>
         ^~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

HTTP://WWW.COMPLETELYSERIOUSCOMICS.COM/

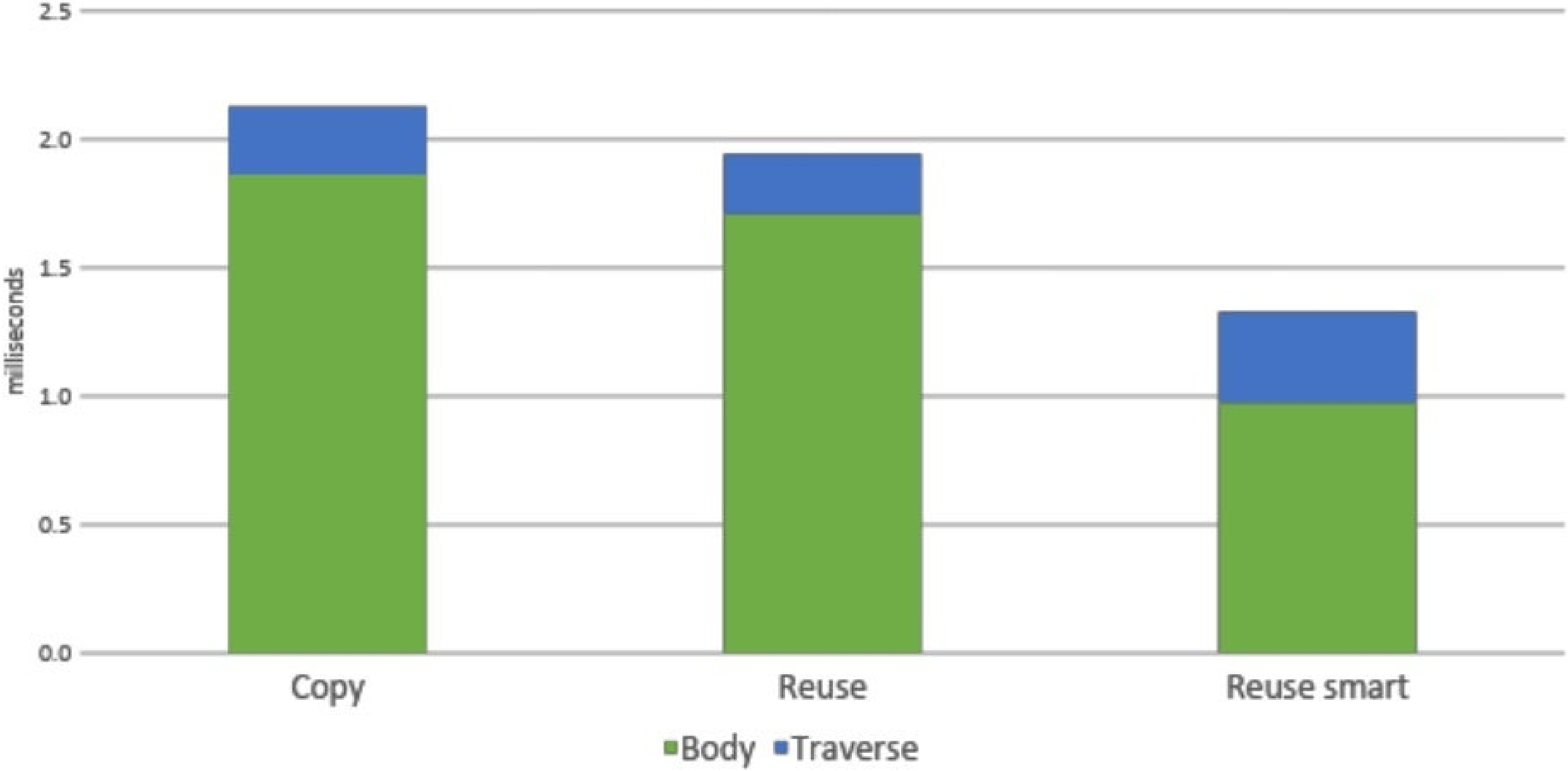


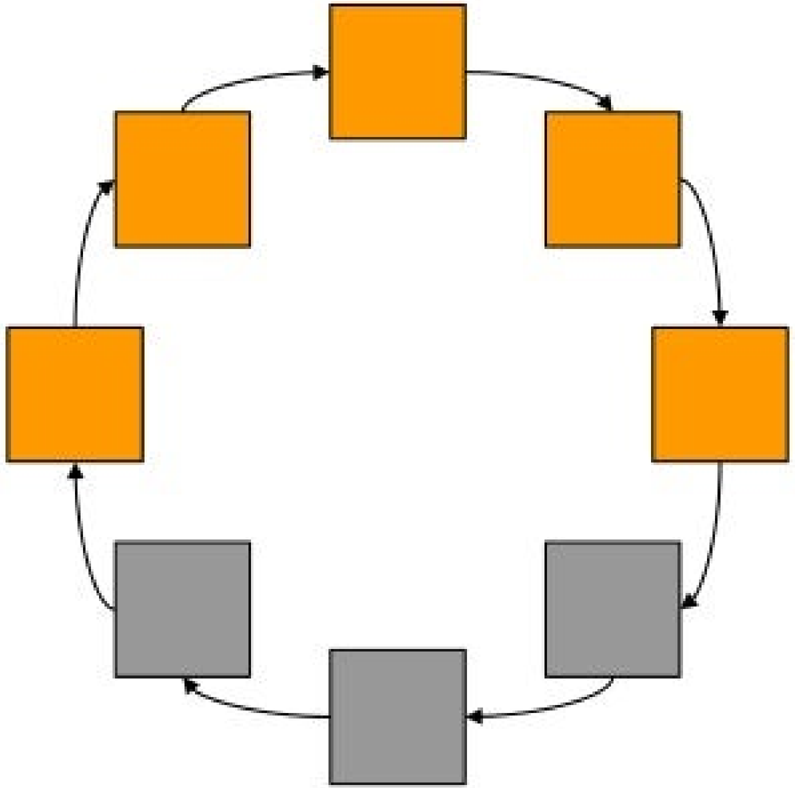
# of segments?
One
Two
At the beginning?
Yes
No
Nothing to do!
Move to front!
Enough room?
Yes
No
Move both!
Fall back to rotate()

```
template<class It>
void gather(It first, It last, It begin1, It end1, It begin2, It end2)
{
  assert(begin2 == end2 || begin1 == first && end2 == last);
  if (begin2 == end2) {
    if (begin1 != first)
      std::move(begin1, end1, first);  // Like std::copy(), but moves elements
    return;
  }
  auto len2 = std::distance(begin2, end2);  // Better for generic code
  auto lenFree = std::distance(end1, begin2);
  if (len2 <= lenFree) {
    auto len1 = std::distance(begin1, end1);
    std::move_backward(begin1, end1, first + len1 + len2);
    std::move(begin2, end2, first);
    return;
  }
  std::rotate(first, begin2, last);
}
```

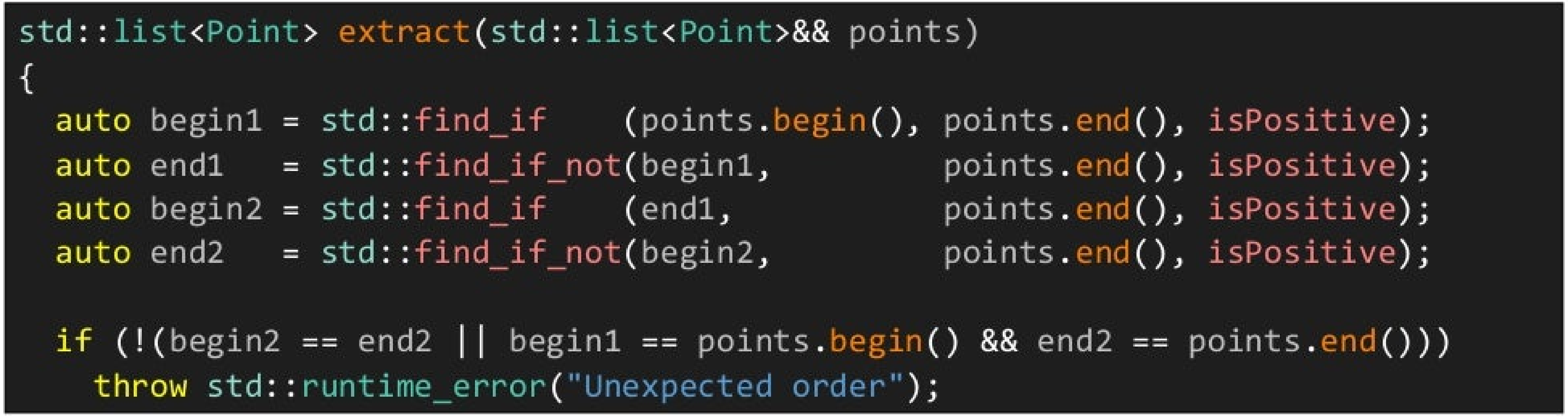


Copy vs. reuse

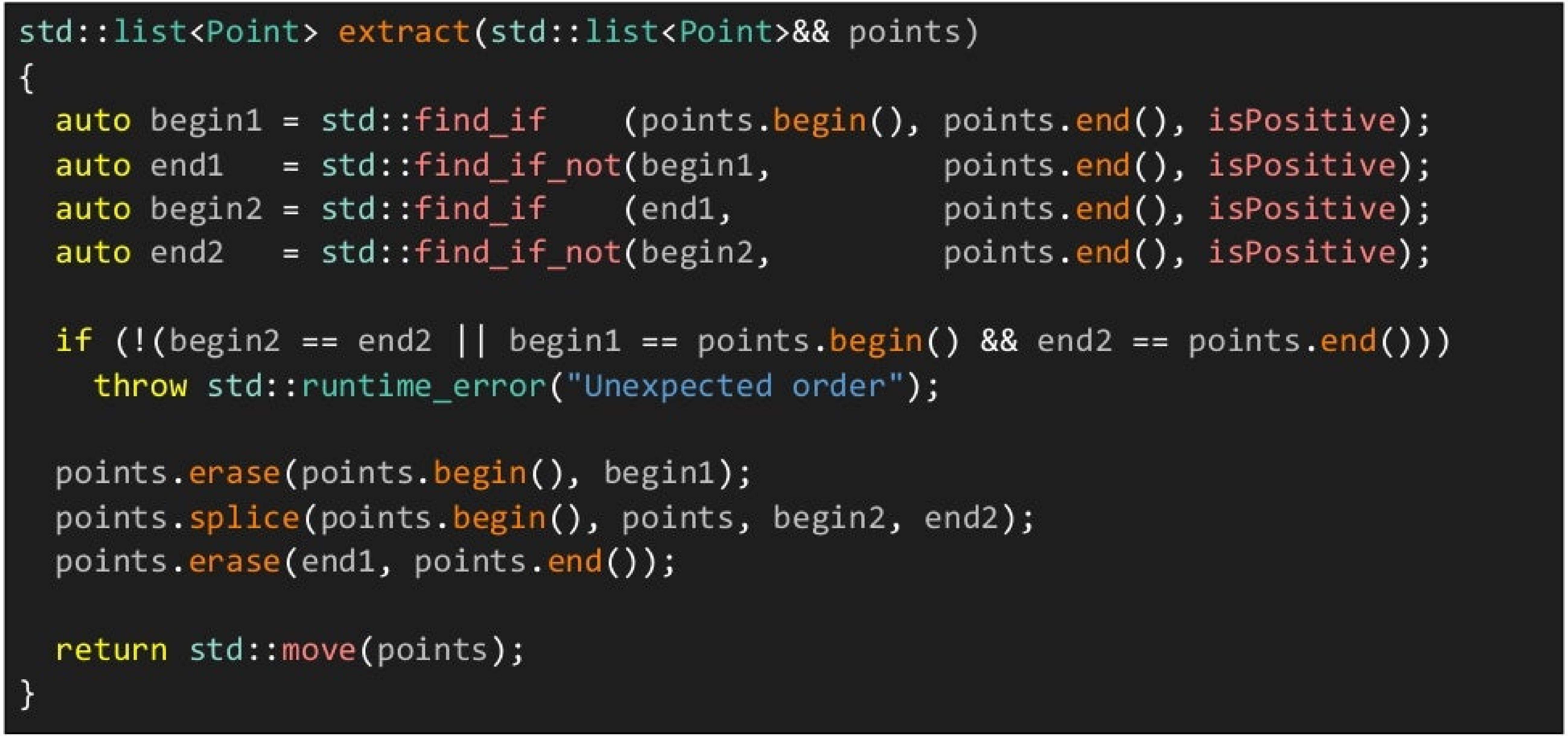




```
std::list<Point> extract(std::list<Point>&& points)
{
```

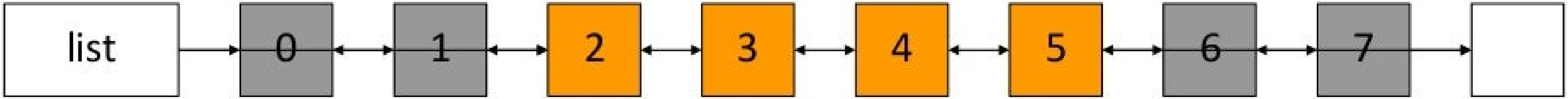


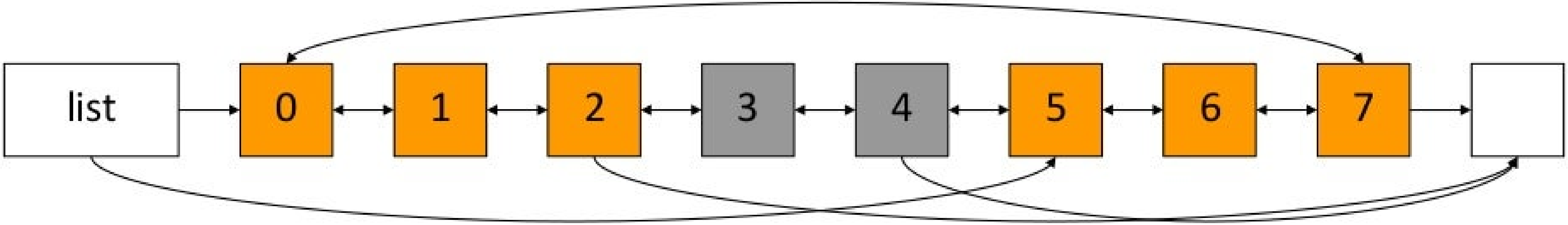
```cpp
std::list<Point> extract(std::list<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");
```

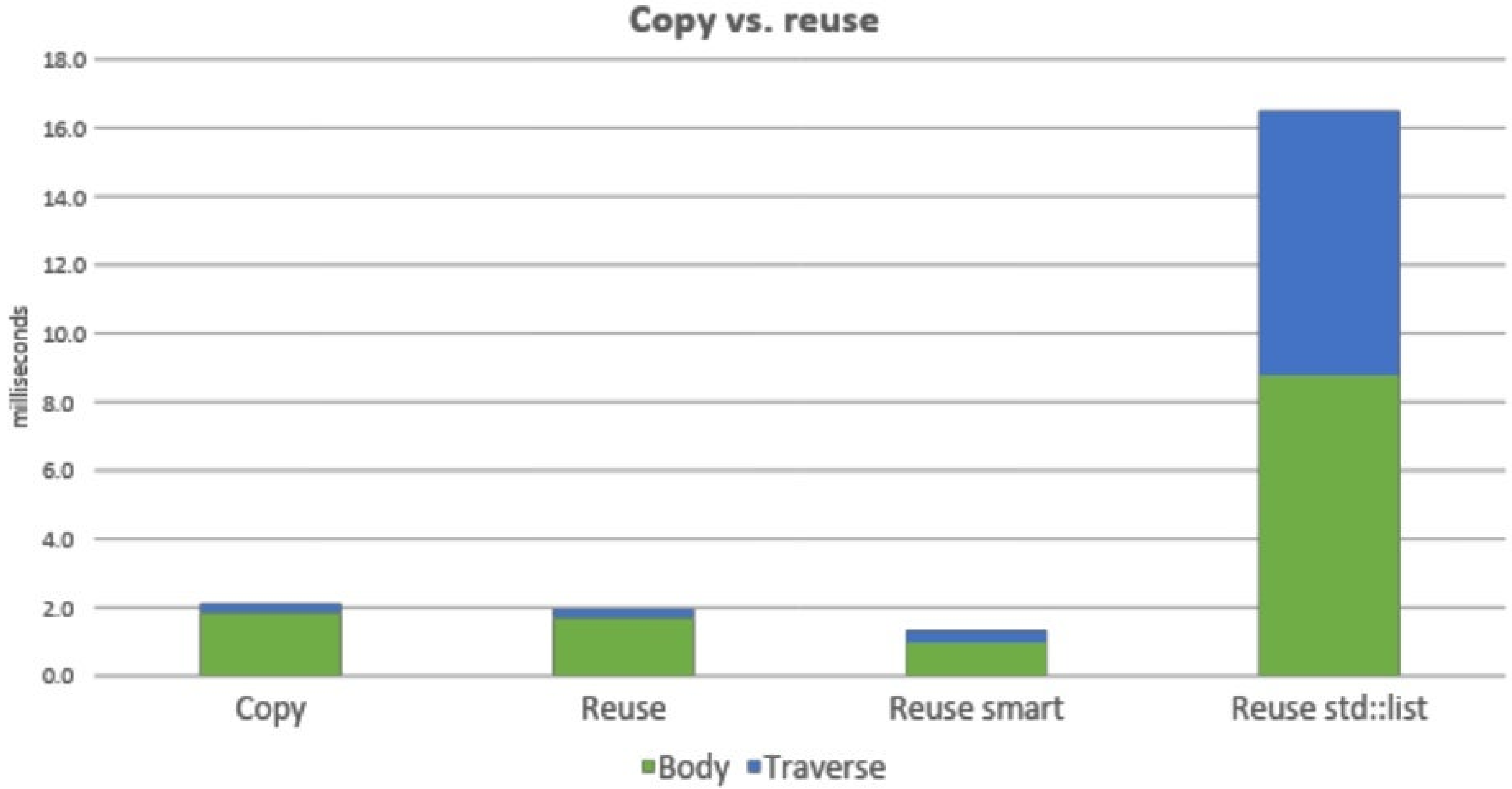


```cpp
std::list<Point> extract(std::list<Point>&& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  points.erase(points.begin(), begin1);
  points.splice(points.begin(), points, begin2, end2);
  points.erase(end1, points.end());

  return std::move(points);
}
```

```
points.erase(points.begin(), begin1);
points.splice(points.begin(), points, begin2, end2);
points.erase(end1, points.end());
```

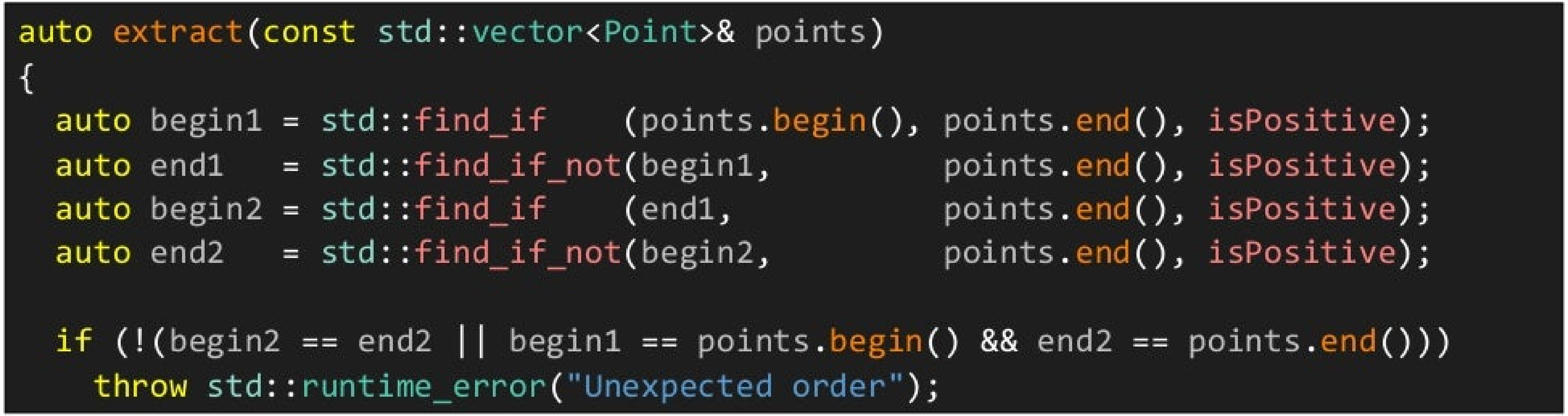




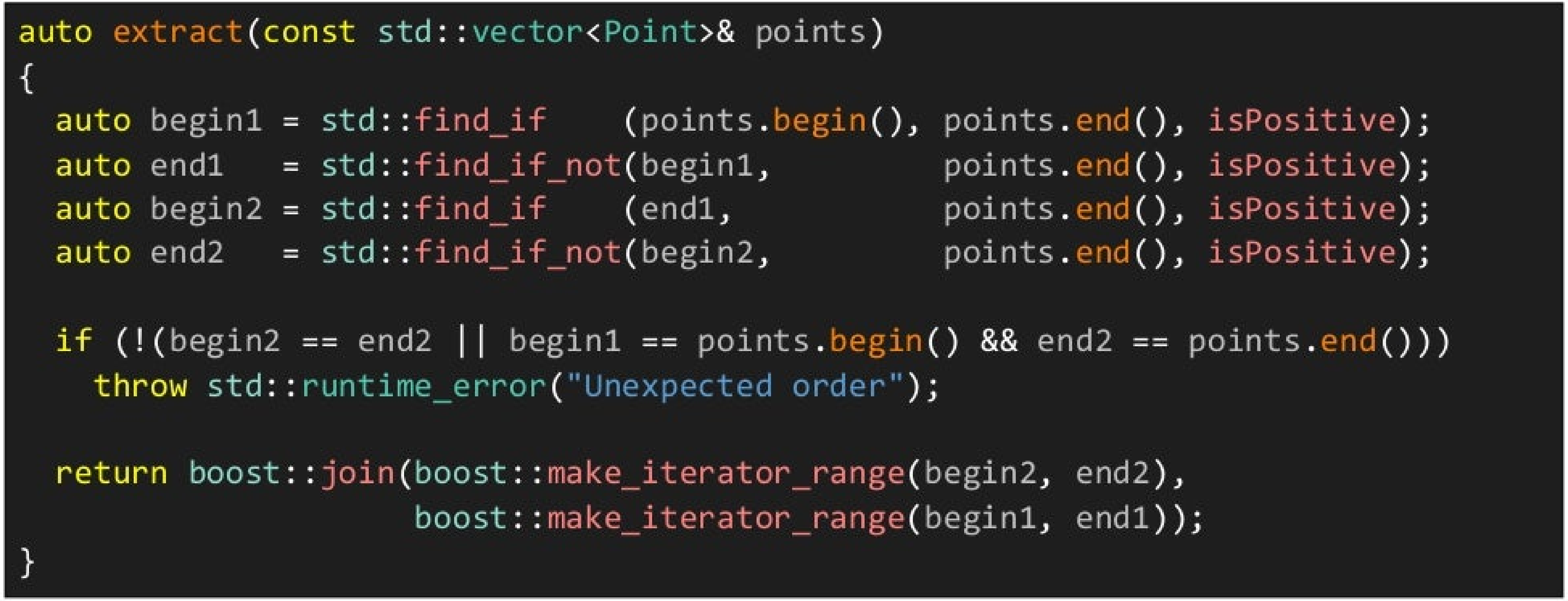

Copy vs. reuse
milliseconds
18.0
16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
Copy
Reuse
Reuse smart
Reuse std::list
🟩Body
🟦Traverse

YOU WERE THE CHOSEN ONE!

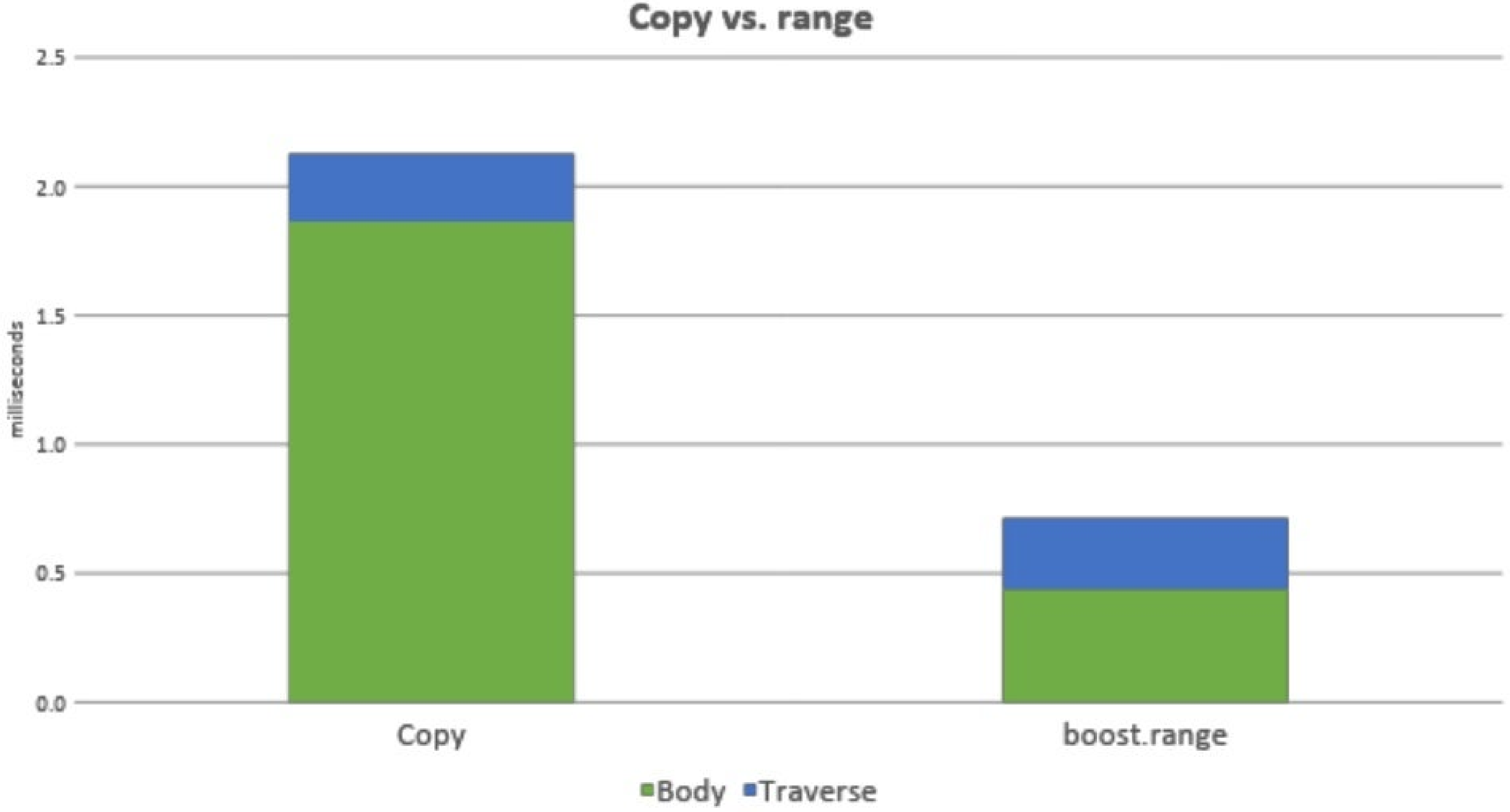
std::list

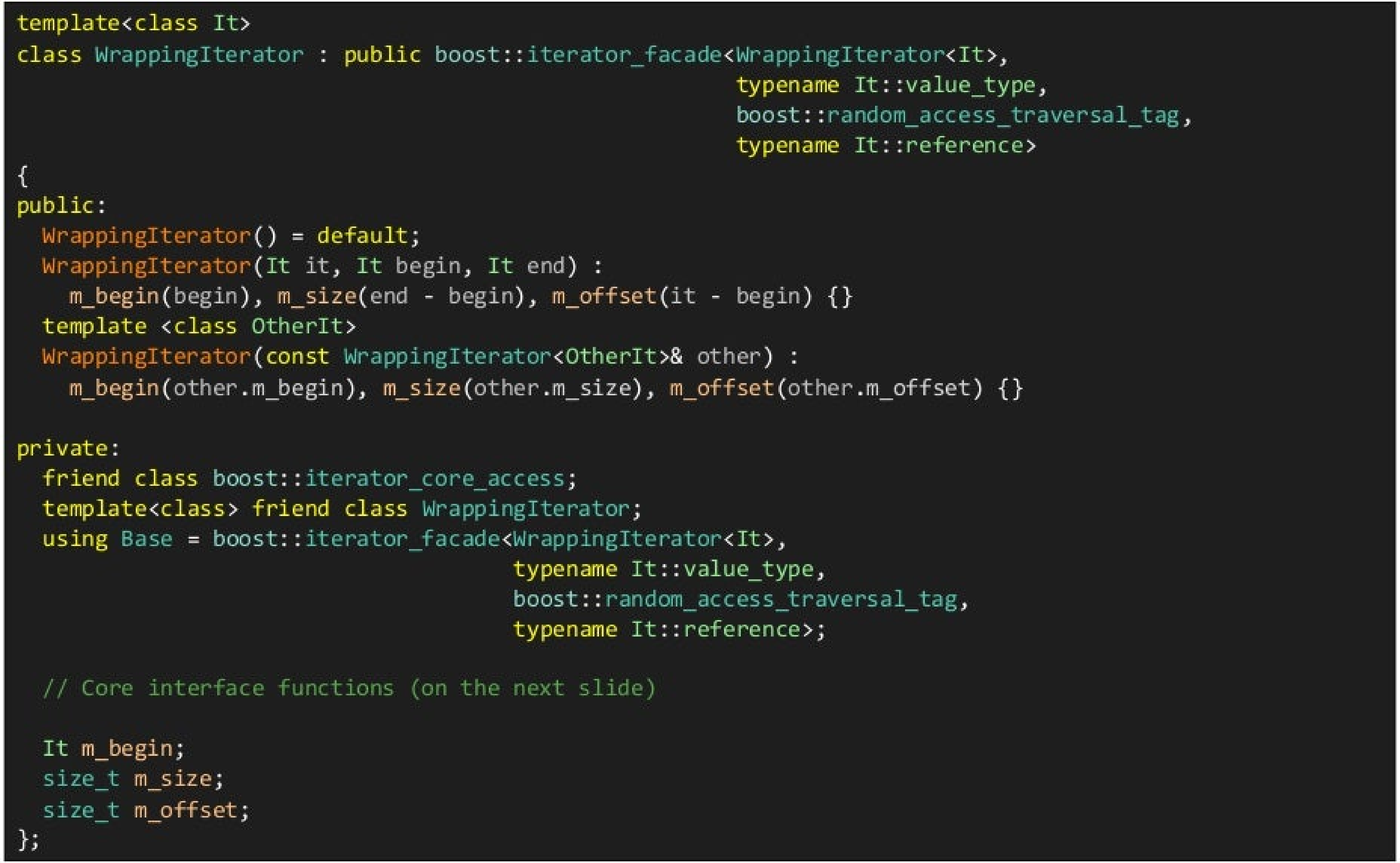


```
auto extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");
```

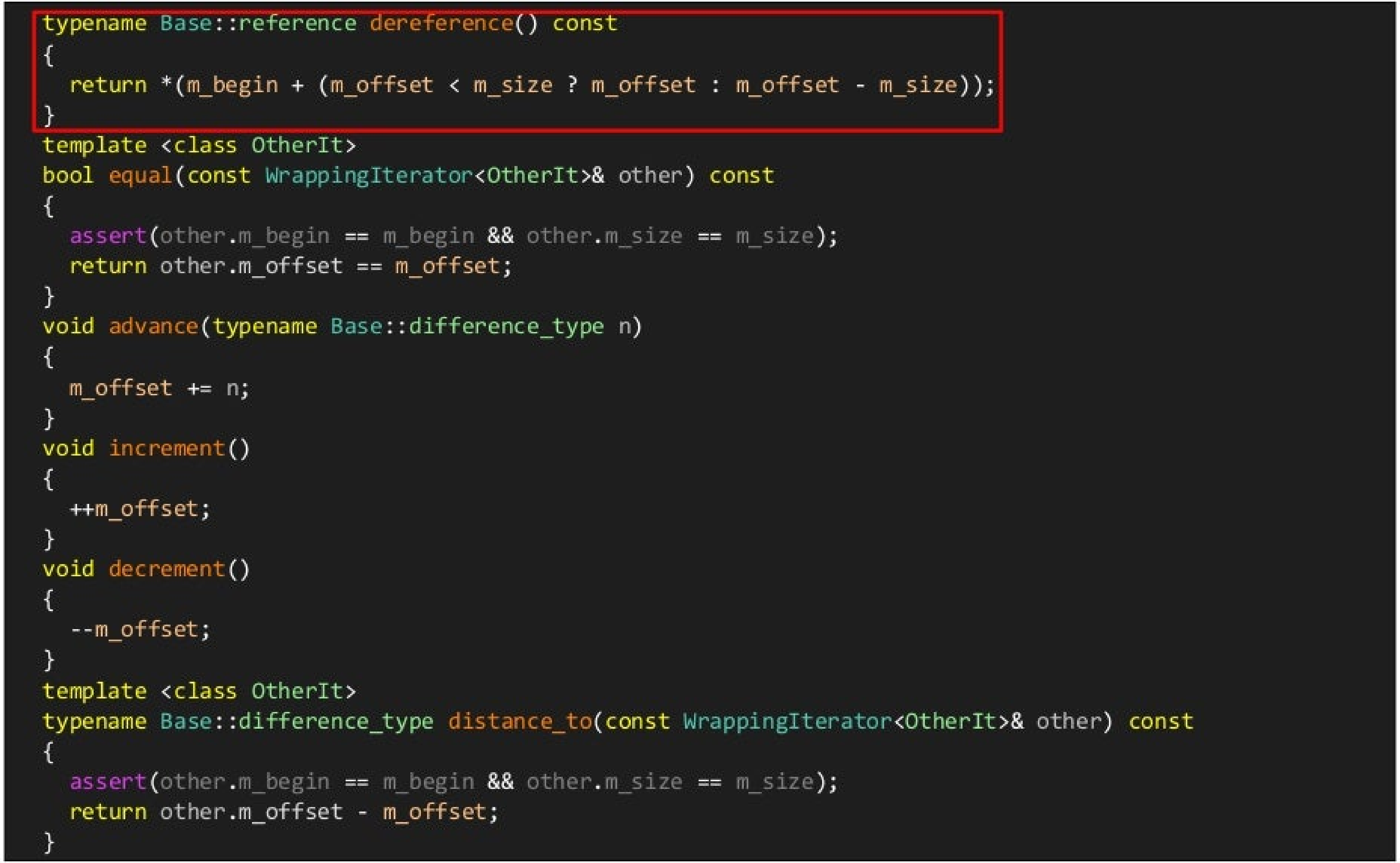


```
auto extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```


Copy vs. range
milliseconds
2.5
2.0
1.5
1.0
0.5
0.0
Copy
boost.range
🟩Body 🟦Traverse

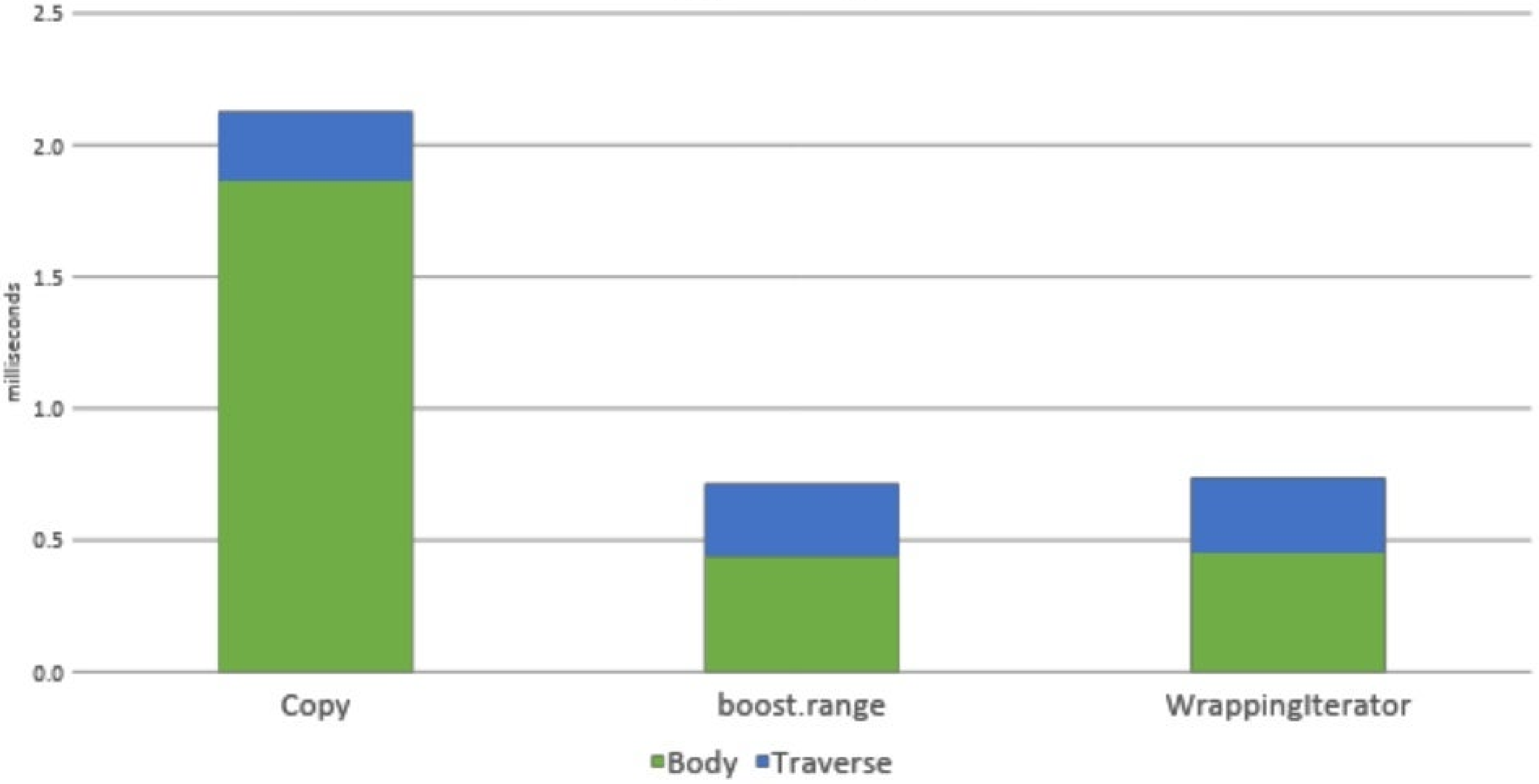


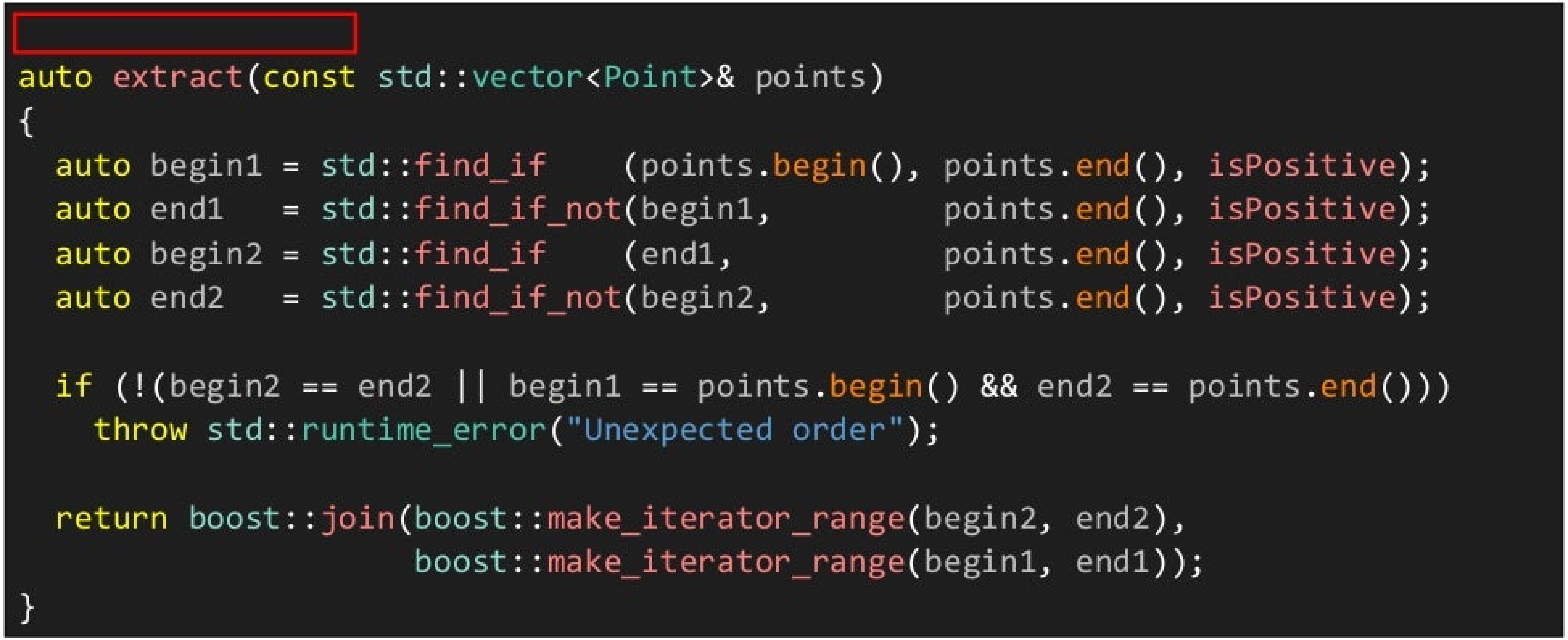
```cpp
template<class It>
class WrappingIterator : public boost::iterator_facade<WrappingIterator<It>,
                                                       typename It::value_type,
                                                       boost::random_access_traversal_tag,
                                                       typename It::reference>
{
public:
  WrappingIterator() = default;
  WrappingIterator(It it, It begin, It end) :
    m_begin(begin), m_size(end - begin), m_offset(it - begin) {}
  template <class OtherIt>
  WrappingIterator(const WrappingIterator<OtherIt>& other) :
    m_begin(other.m_begin), m_size(other.m_size), m_offset(other.m_offset) {}

private:
  friend class boost::iterator_core_access;
  template<class> friend class WrappingIterator;
  using Base = boost::iterator_facade<WrappingIterator<It>,
                                      typename It::value_type,
                                      boost::random_access_traversal_tag,
                                      typename It::reference>;

  // Core interface functions (on the next slide)

  It m_begin;
  size_t m_size;
  size_t m_offset;
};
```



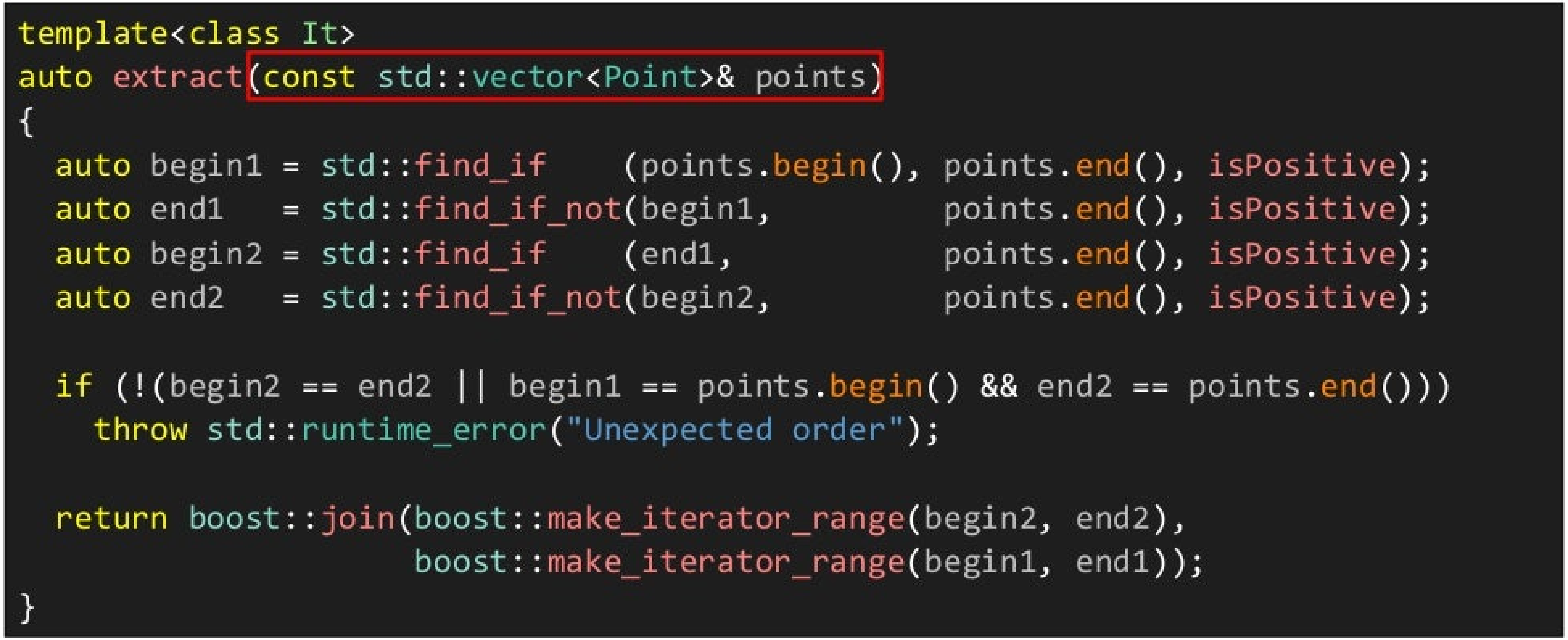
```
typename Base::reference dereference() const
{
  return *(m_begin + (m_offset < m_size ? m_offset : m_offset - m_size));
}
template <class OtherIt>
bool equal(const WrappingIterator<OtherIt>& other) const
{
  assert(other.m_begin == m_begin && other.m_size == m_size);
  return other.m_offset == m_offset;
}
void advance(typename Base::difference_type n)
{
  m_offset += n;
}
void increment()
{
  ++m_offset;
}
void decrement()
{
  --m_offset;
}
template <class OtherIt>
typename Base::difference_type distance_to(const WrappingIterator<OtherIt>& other) const
{
  assert(other.m_begin == m_begin && other.m_size == m_size);
  return other.m_offset - m_offset;
}
```

Copy vs. range

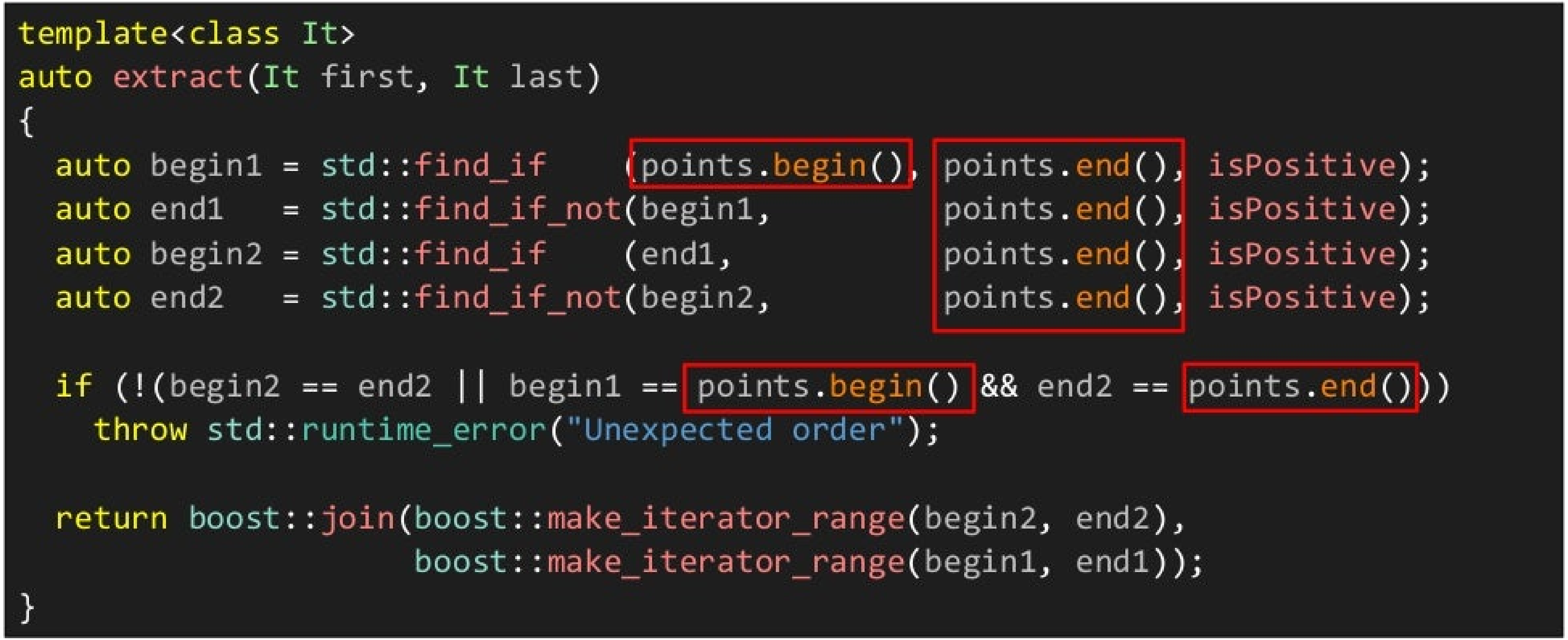




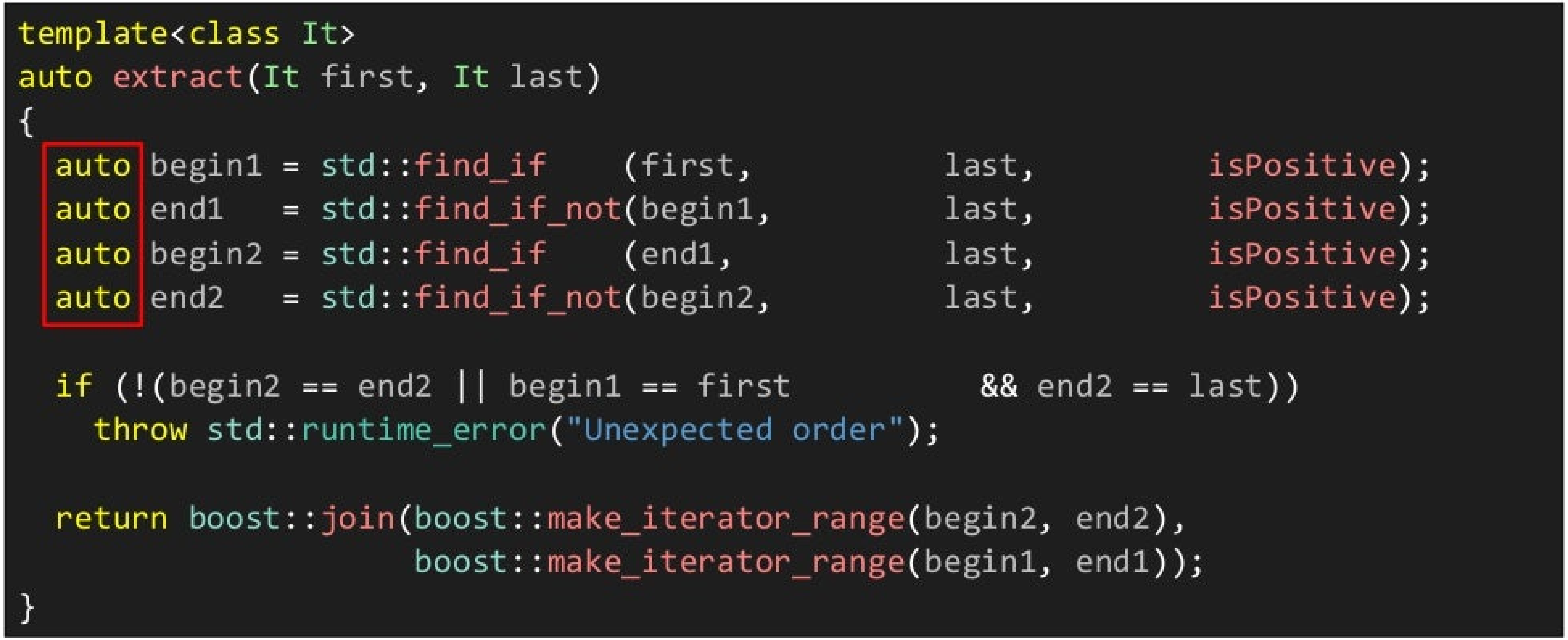
```cpp
auto extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```



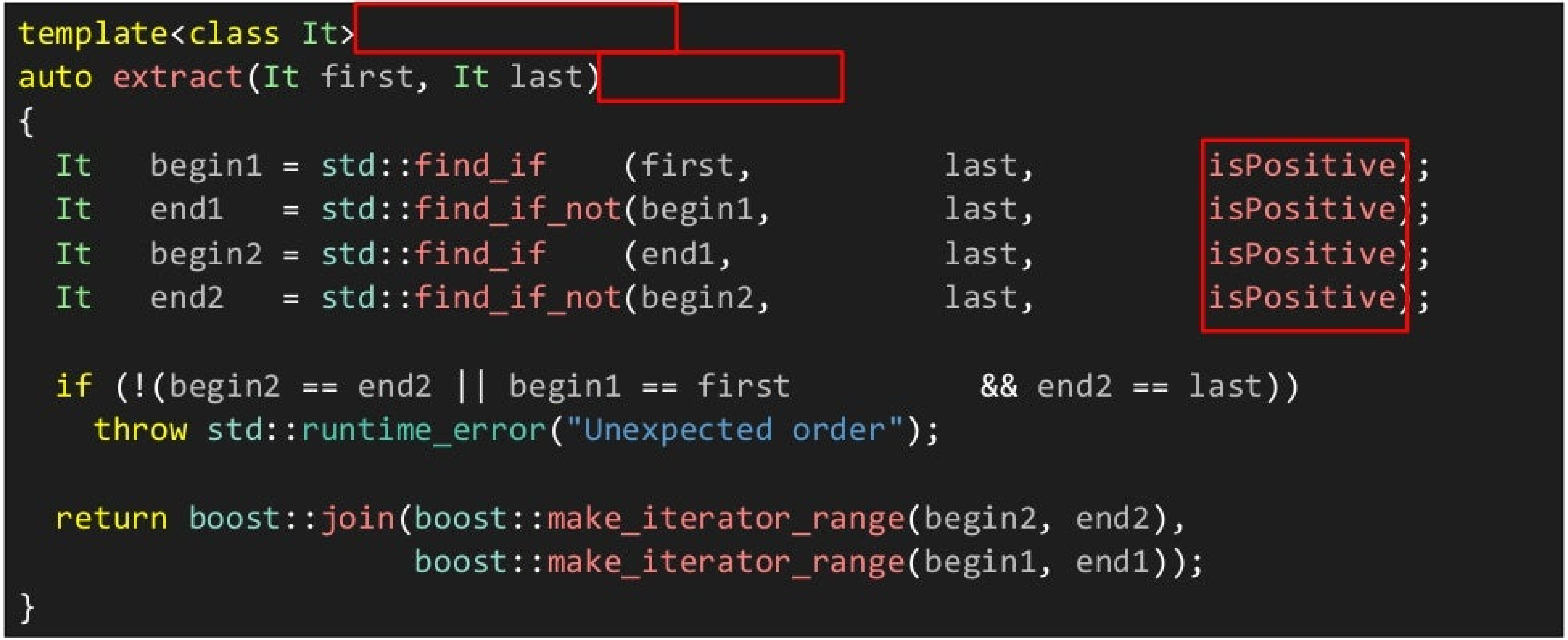
```cpp
template<class It>
auto extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```



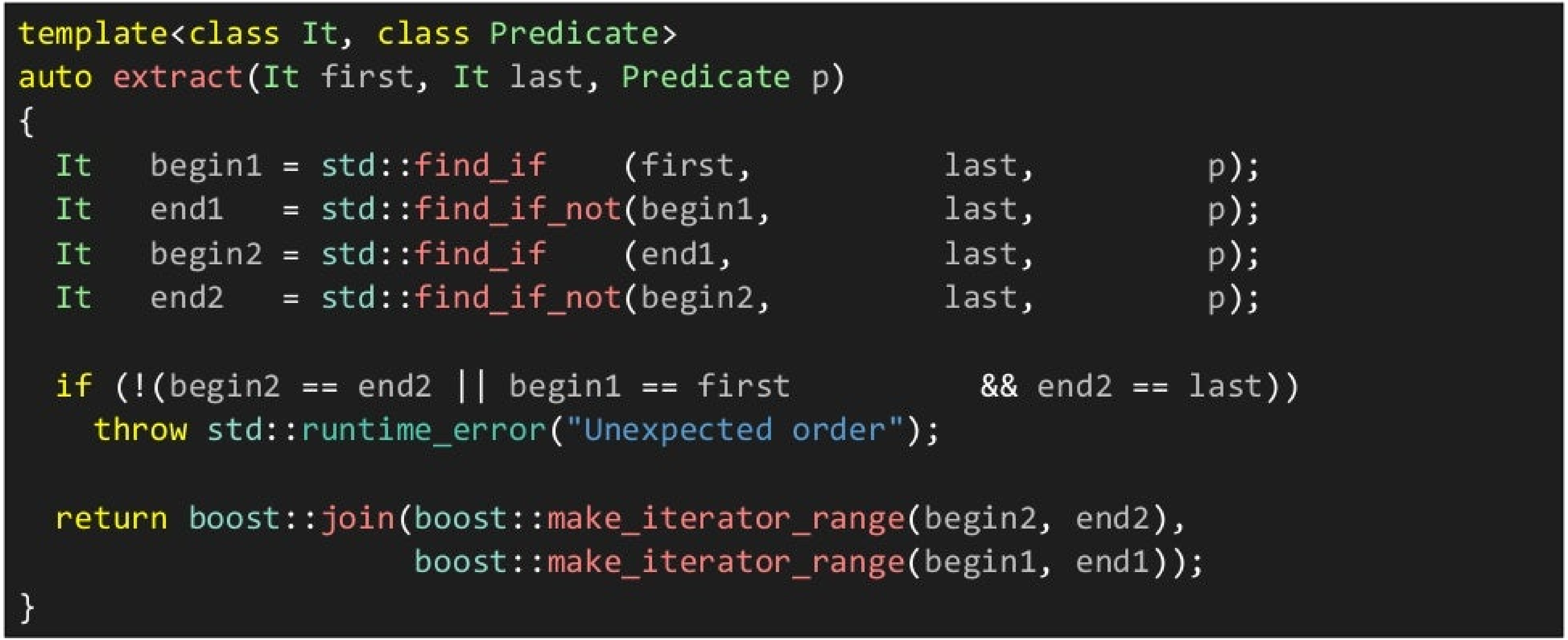
```
template<class It>
auto extract(It first, It last)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```



```
template<class It>
auto extract(It first, It last)
{
  auto begin1 = std::find_if    (first,          last,         isPositive);
  auto end1   = std::find_if_not(begin1,         last,         isPositive);
  auto begin2 = std::find_if    (end1,           last,         isPositive);
  auto end2   = std::find_if_not(begin2,         last,         isPositive);

  if (!(begin2 == end2 || begin1 == first          && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

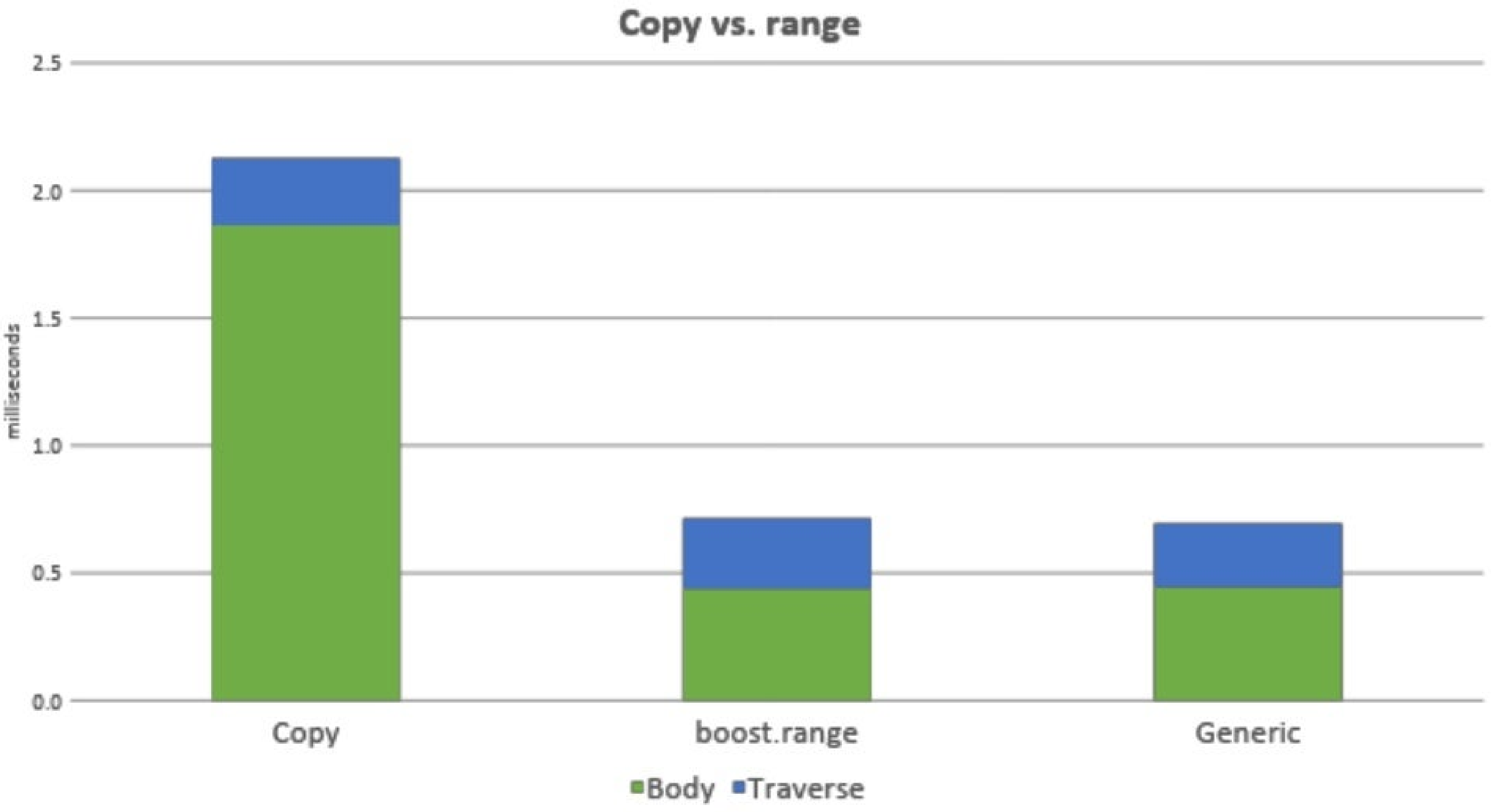


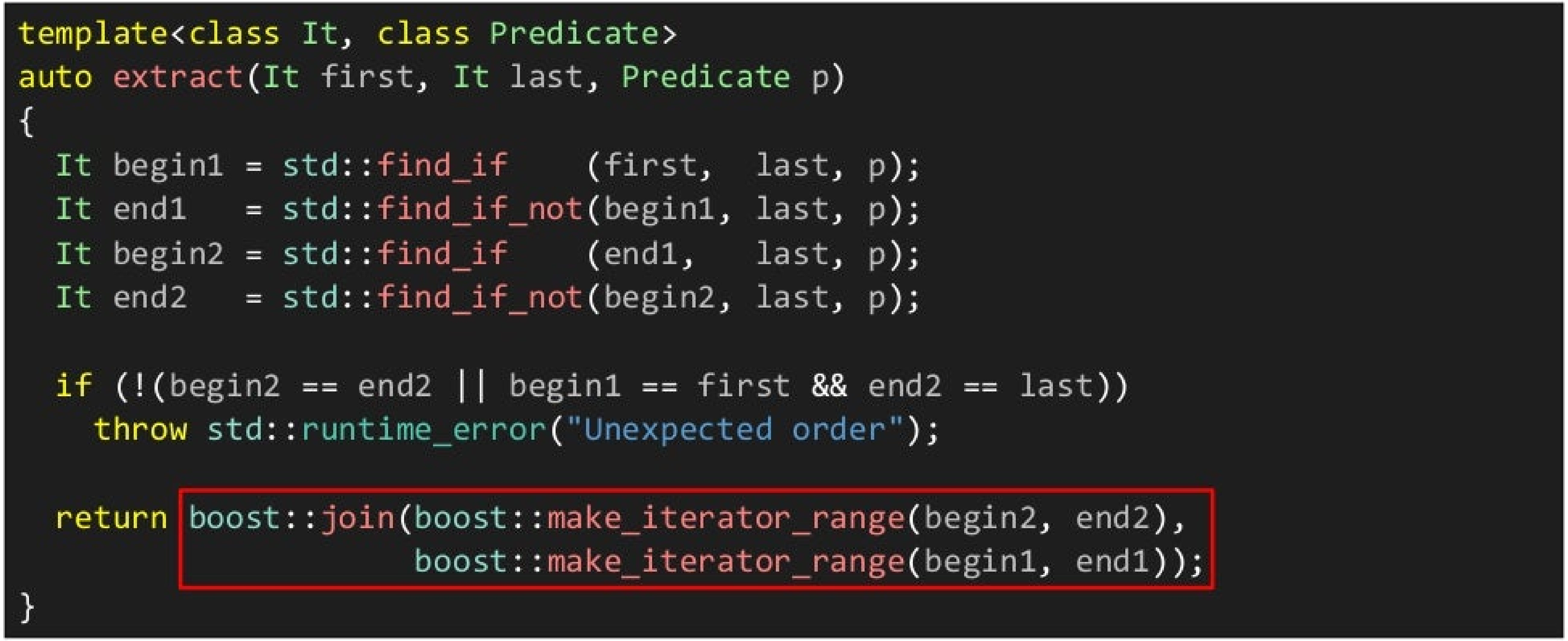
```cpp
template<class It>
auto extract(It first, It last)
{
  It   begin1 = std::find_if    (first,          last,         isPositive);
  It   end1   = std::find_if_not(begin1,         last,         isPositive);
  It   begin2 = std::find_if    (end1,           last,         isPositive);
  It   end2   = std::find_if_not(begin2,         last,         isPositive);

  if (!(begin2 == end2 || begin1 == first          && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```



```cpp
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It   begin1 = std::find_if    (first,          last,         p);
  It   end1   = std::find_if_not(begin1,         last,         p);
  It   begin2 = std::find_if    (end1,           last,         p);
  It   end2   = std::find_if_not(begin2,         last,         p);

  if (!(begin2 == end2 || begin1 == first          && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

```cpp
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = std::find_if_not(begin2, last, p);

  if (!(begin2 == end2 || begin1 == first && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

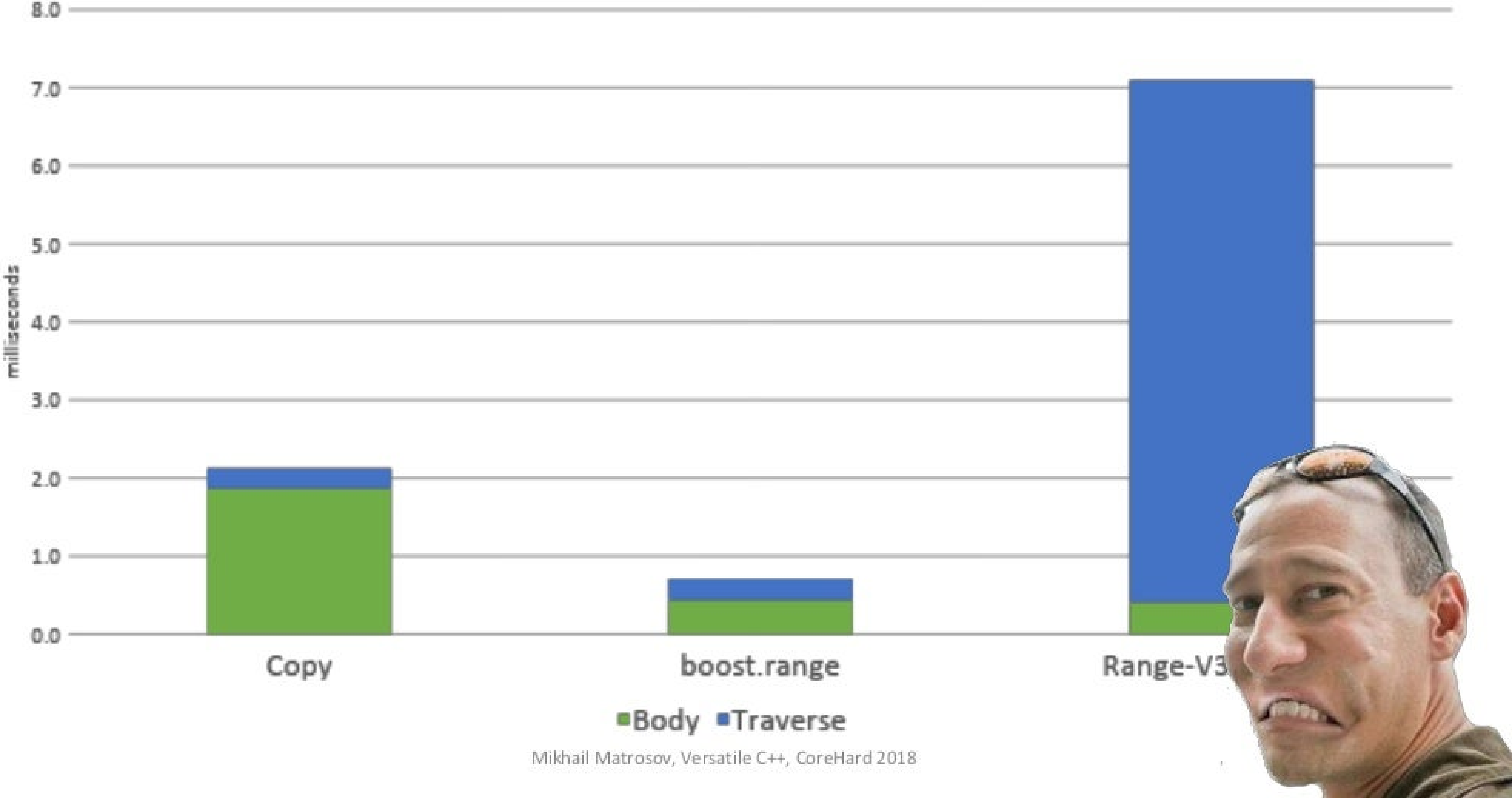

Copy vs. range
2.5
2.0
1.5
1.0
0.5
0.0
milliseconds
Copy
boost.range
Generic
🟩Body 🟦Traverse



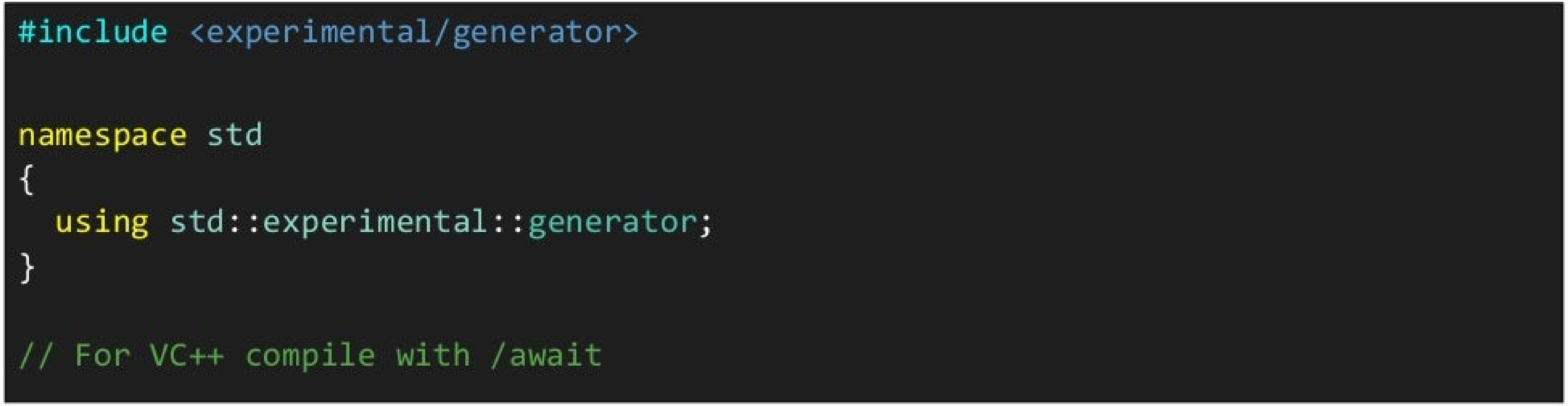
```
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = std::find_if_not(begin2, last, p);

  if (!(begin2 == end2 || begin1 == first && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

```
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = std::find_if_not(begin2, last, p);

  if (!(begin2 == end2 || begin1 == first && end2 == last))
    throw std::runtime_error("Unexpected order");

  return ranges::view::concat(ranges::range<It>(begin2, end2),
                              ranges::range<It>(begin1, end1));
}
```

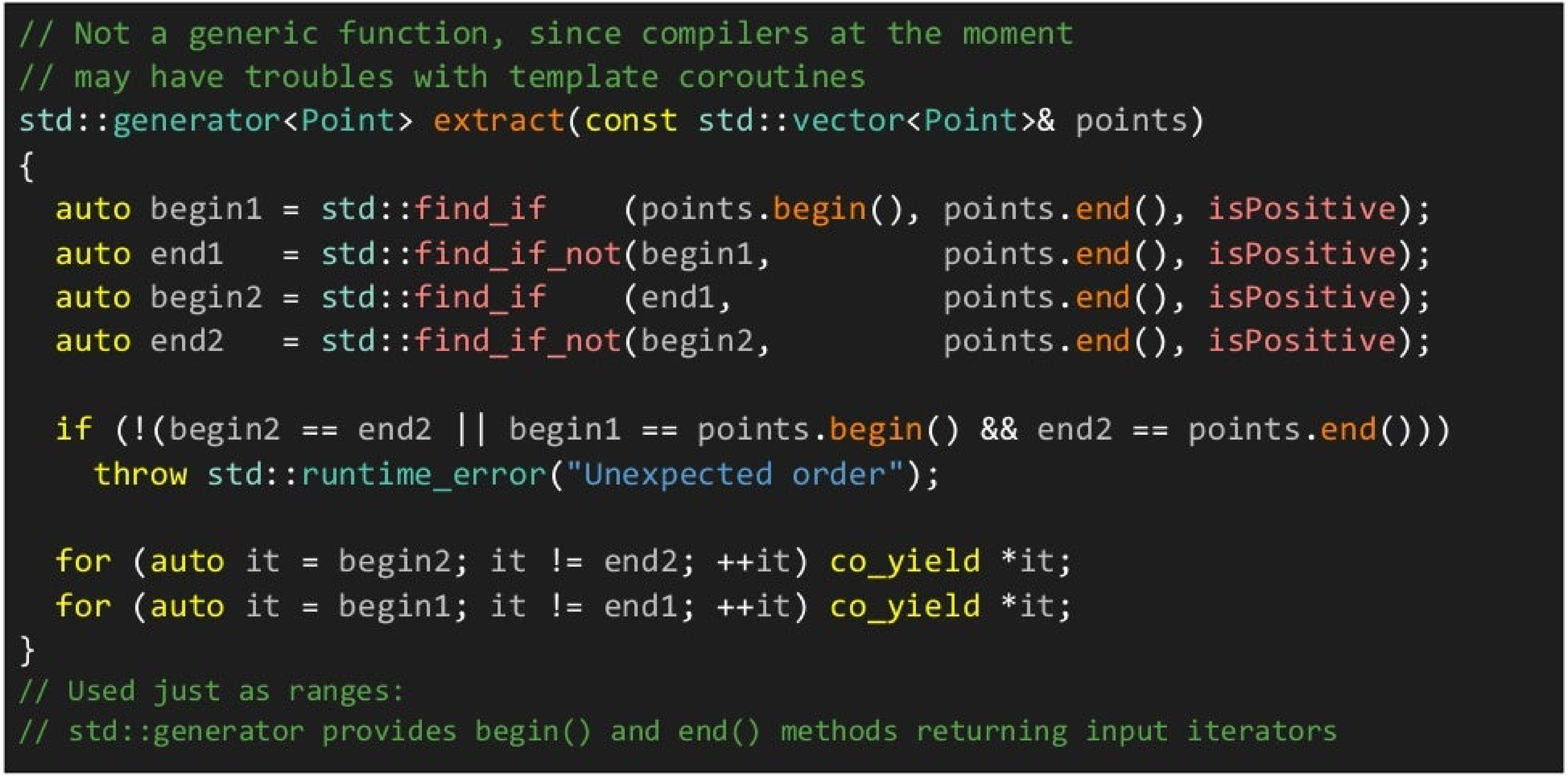
Copy vs. range



Coroutines!

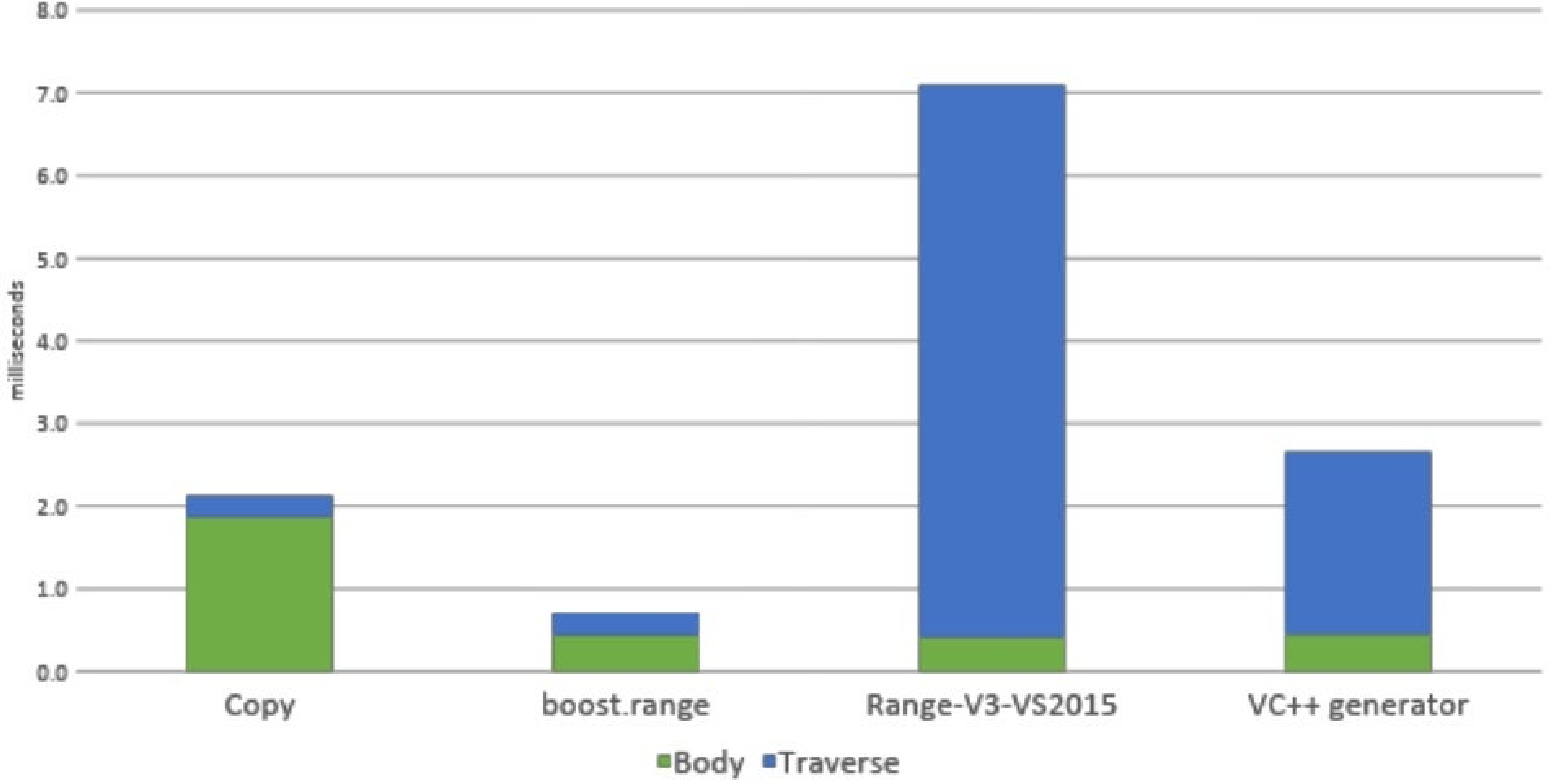


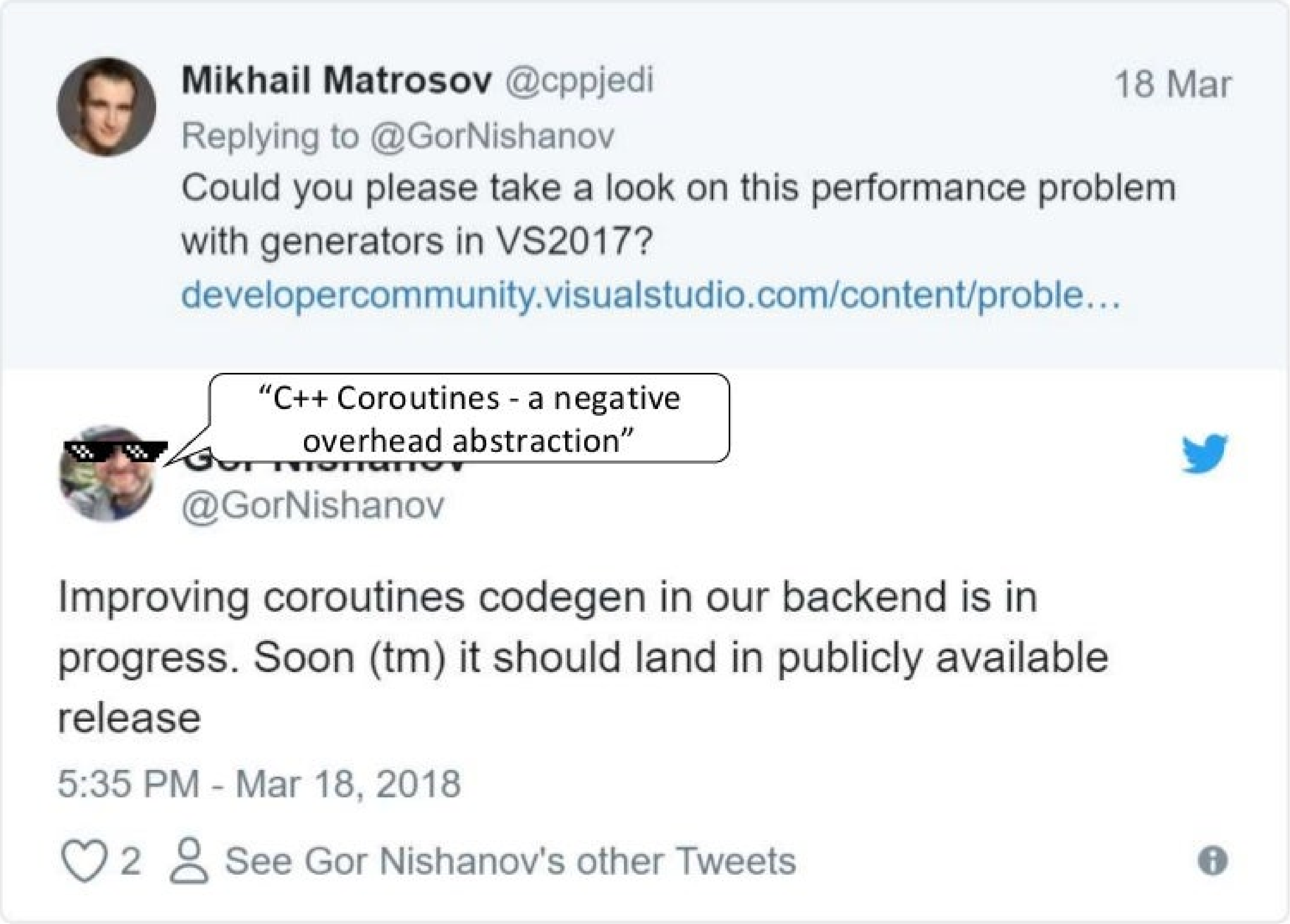
```
#include <experimental/generator>

namespace std
{
  using std::experimental::generator;
}

// For VC++ compile with /await
```



```
// Not a generic function, since compilers at the moment
// may have troubles with template coroutines
std::generator<Point> extract(const std::vector<Point>& points)
{
  auto begin1 = std::find_if    (points.begin(), points.end(), isPositive);
  auto end1   = std::find_if_not(begin1,         points.end(), isPositive);
  auto begin2 = std::find_if    (end1,           points.end(), isPositive);
  auto end2   = std::find_if_not(begin2,         points.end(), isPositive);

  if (!(begin2 == end2 || begin1 == points.begin() && end2 == points.end()))
    throw std::runtime_error("Unexpected order");

  for (auto it = begin2; it != end2; ++it) co_yield *it;
  for (auto it = begin1; it != end1; ++it) co_yield *it;
}
// Used just as ranges:
// std::generator provides begin() and end() methods returning input iterators
```

Copy vs. range





Mikhail Matrosov @cppjedi 18 Mar

Replying to @GorNishanov

Could you please take a look on this performance problem with generators in VS2017? developercommunity.visualstudio.com/content/proble…

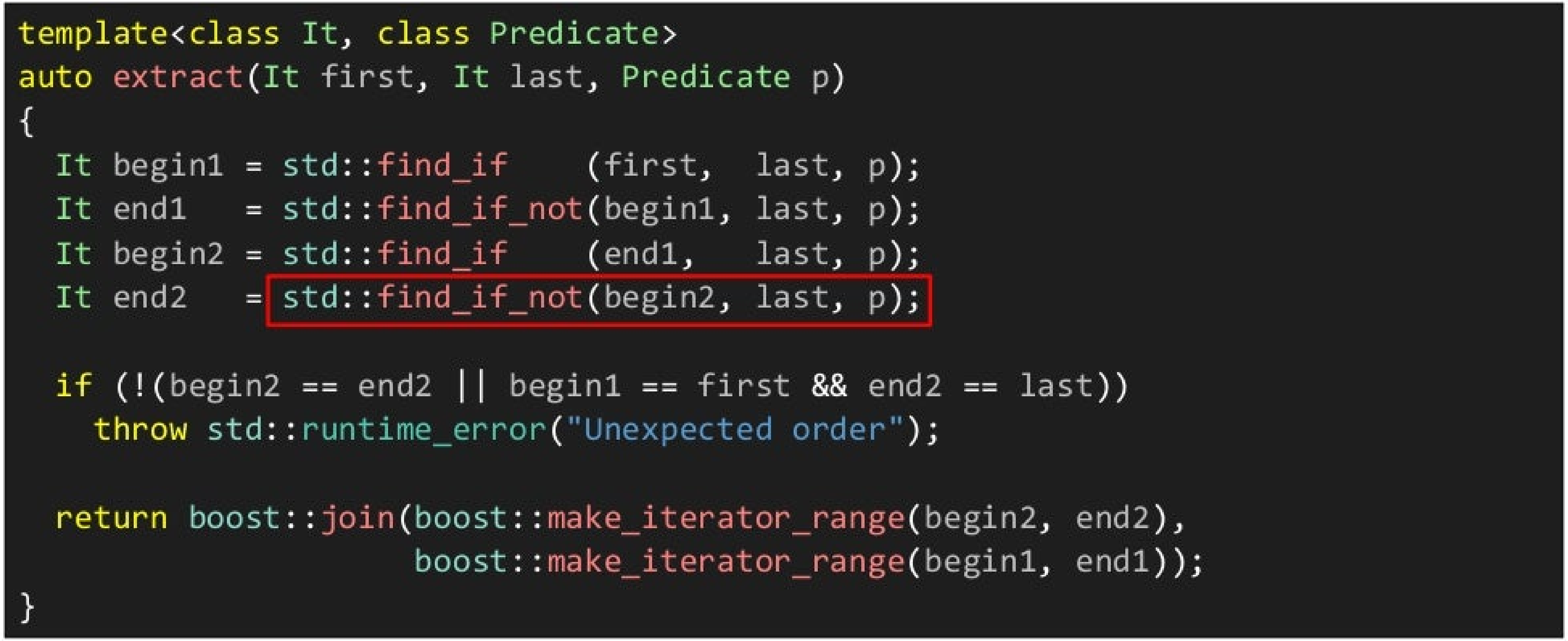
"C++ Coroutines - a negative overhead abstraction"

Gor Nishanov
@GorNishanov

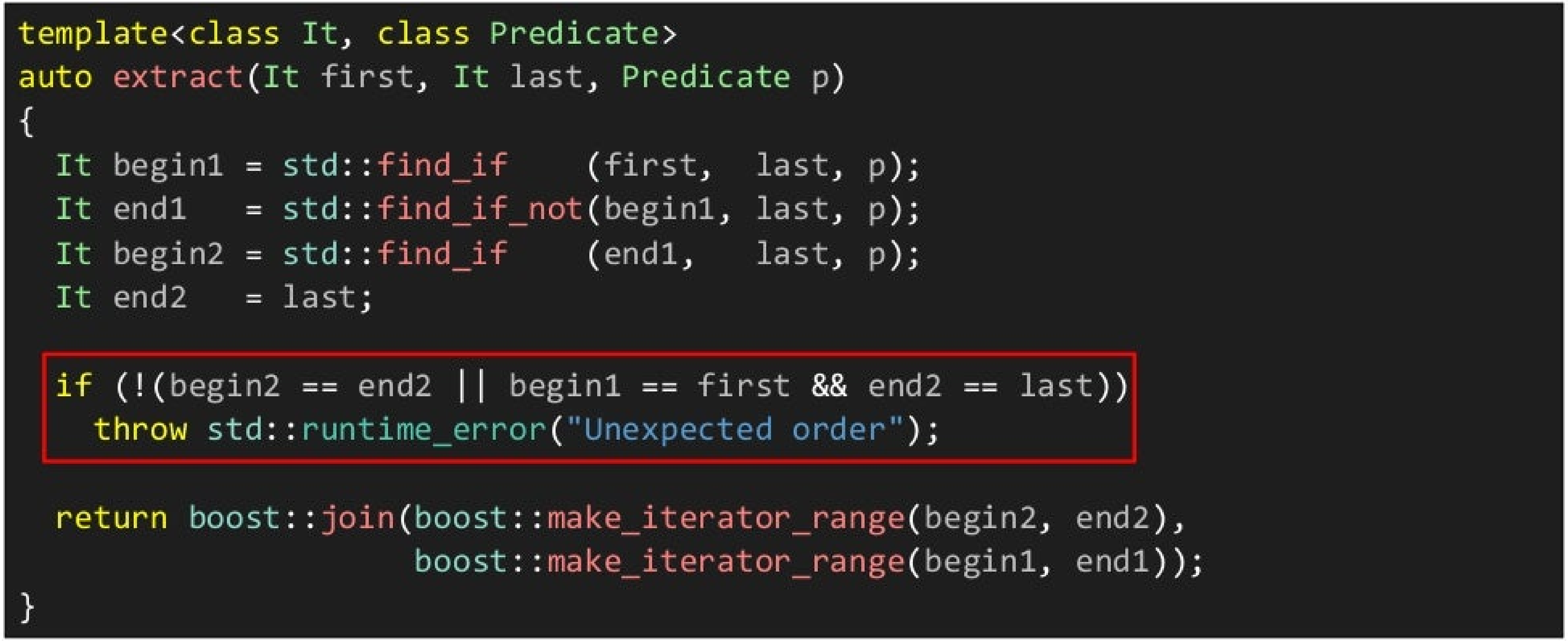
Improving coroutines codegen in our backend is in progress. Soon (tm) it should land in publicly available release

5:35 PM - Mar 18, 2018

♡ 2 See Gor Nishanov's other Tweets



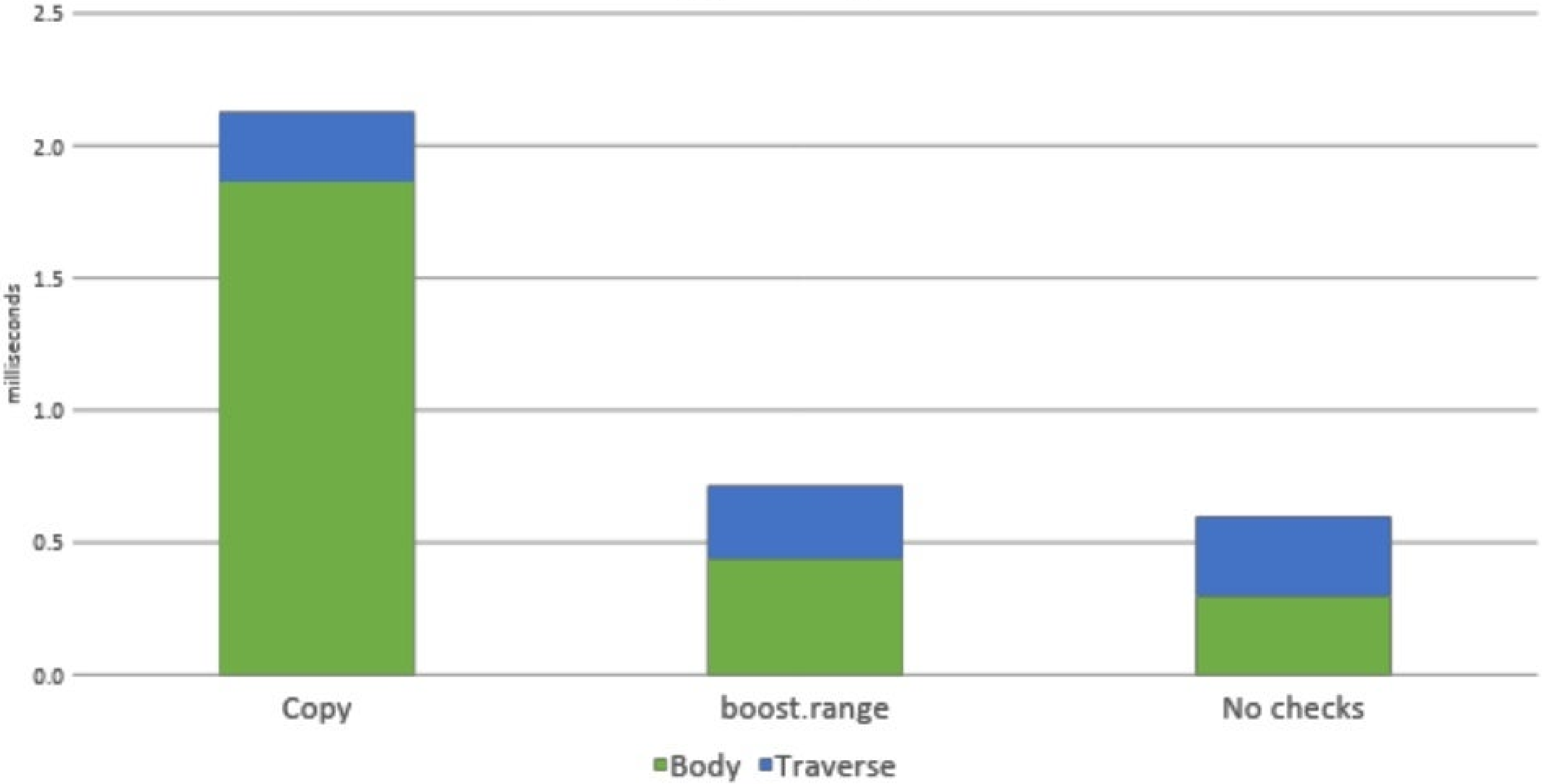
```cpp
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = std::find_if_not(begin2, last, p);

  if (!(begin2 == end2 || begin1 == first && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```



```cpp
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = last;

  if (!(begin2 == end2 || begin1 == first && end2 == last))
    throw std::runtime_error("Unexpected order");

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

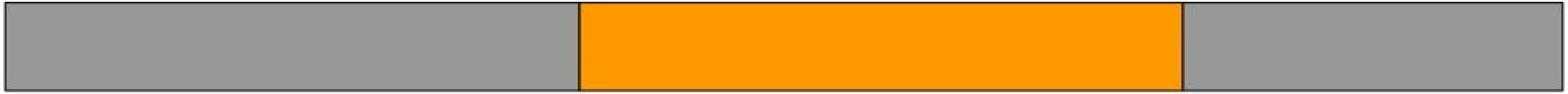
```
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = last;

  assert(end2 == std::find_if_not(begin2, last, p) &&
    (begin2 == end2 || begin1 == first && end2 == last));

  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

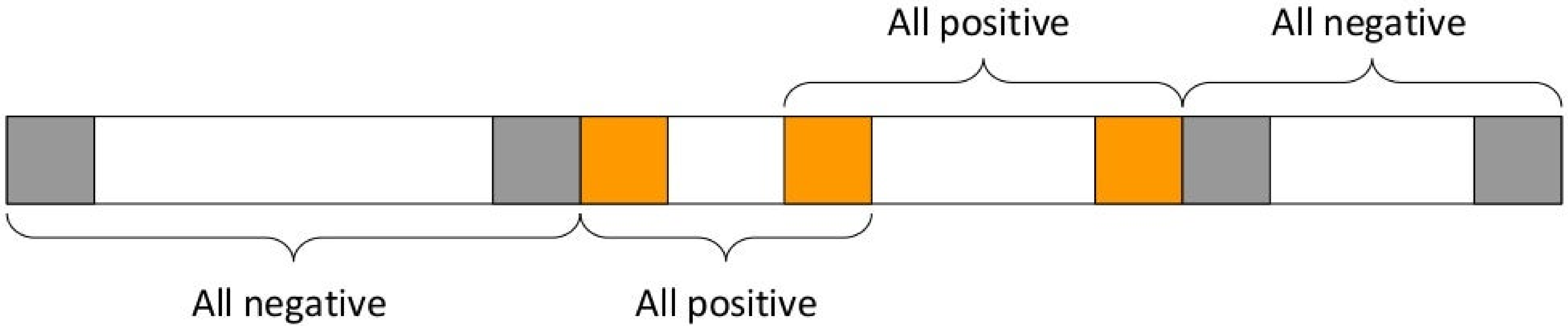
Copy vs. range



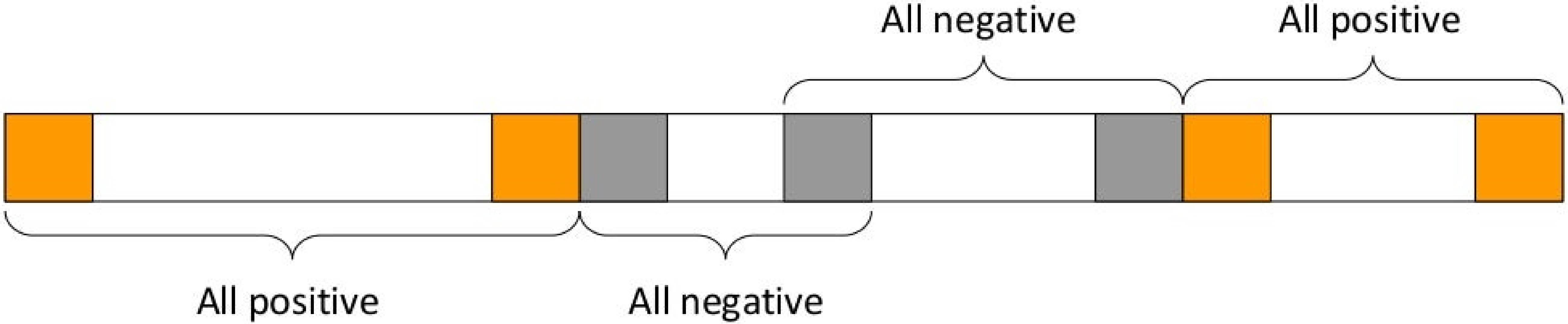
```cpp
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  It end2   = last;

  assert(end2 == std::find_if_not(begin2, last, p) &&
    (begin2 == end2 || begin1 == first && end2 == last));


  return boost::join(boost::make_iterator_range(begin2, end2),
                     boost::make_iterator_range(begin1, end1));
}
```

```
template<class It>
struct Bounds
{
  It begin1, end1, begin2, end2;
};
template<class It, class Predicate>
Bounds<It> findBounds(It first, It last, Predicate p)
{
  It begin1 = std::find_if    (first,  last, p);
  It end1   = std::find_if_not(begin1, last, p);
  It begin2 = std::find_if    (end1,   last, p);
  return { begin1, end1, begin2, last };
}
template<class It, class Predicate>
auto extract(It first, It last, Predicate p)
{
  auto bounds = findBounds(first, last, p);
  return boost::join(boost::make_iterator_range(bounds.begin2, bounds.end2),
                     boost::make_iterator_range(bounds.begin1, bounds.end1));
}
```



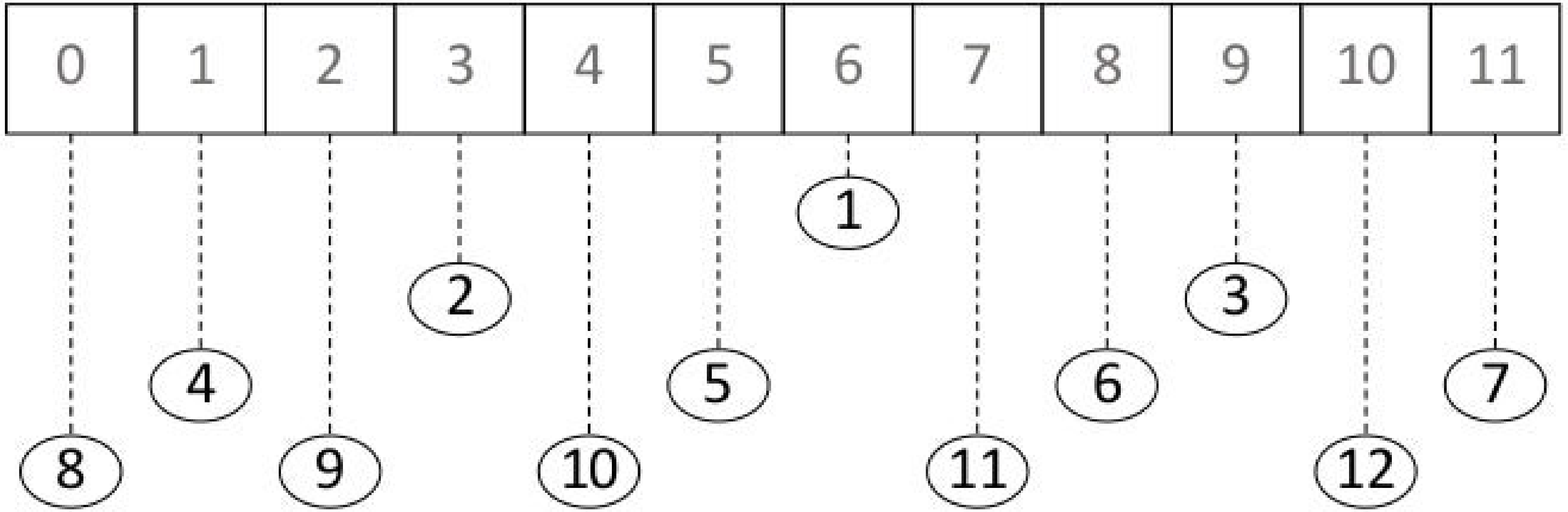


All positive
All negative
All negative
All positive

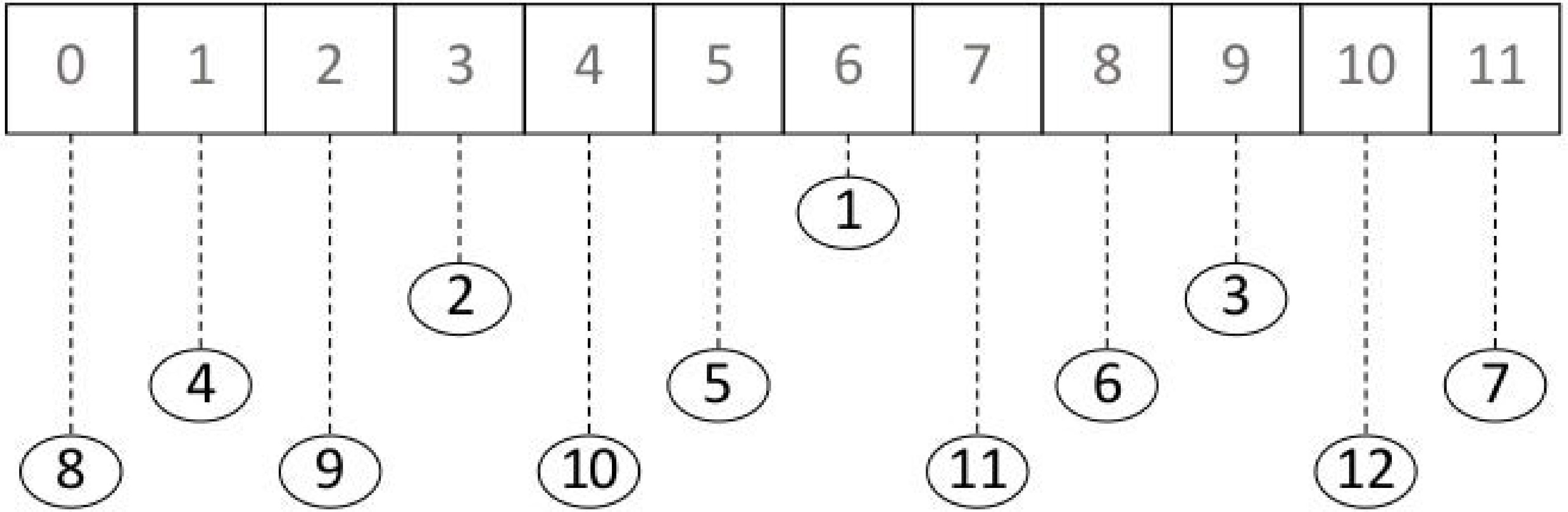

All negative
All positive
All positive
All negative

```cpp
template<class It, class Predicate>
Bounds<It> findBounds(It first, It last, Predicate p)
{
  Bounds<It> bounds = { first, last, last, last };  // Whole sequence by default

  if (first == last)
    return bounds;

  if (!p(*first) || !p(last[-1]))
  {
    // One segment, or empty
    It sample = findAny(first, last, p);
    bounds.begin1 = std::partition_point(first, sample, std::not_fn(p));
    bounds.end1 = std::partition_point(sample, last, p);
  }
  else if (It hole = findAny(first, last, std::not_fn(p)); hole != last)
  {
    // Two segments
    bounds.end1 = std::partition_point(first, hole, p);
    bounds.begin2 = std::partition_point(hole, last, std::not_fn(p));
  }

  return bounds;
}
```

Take element in the middle, then recurse to left and right

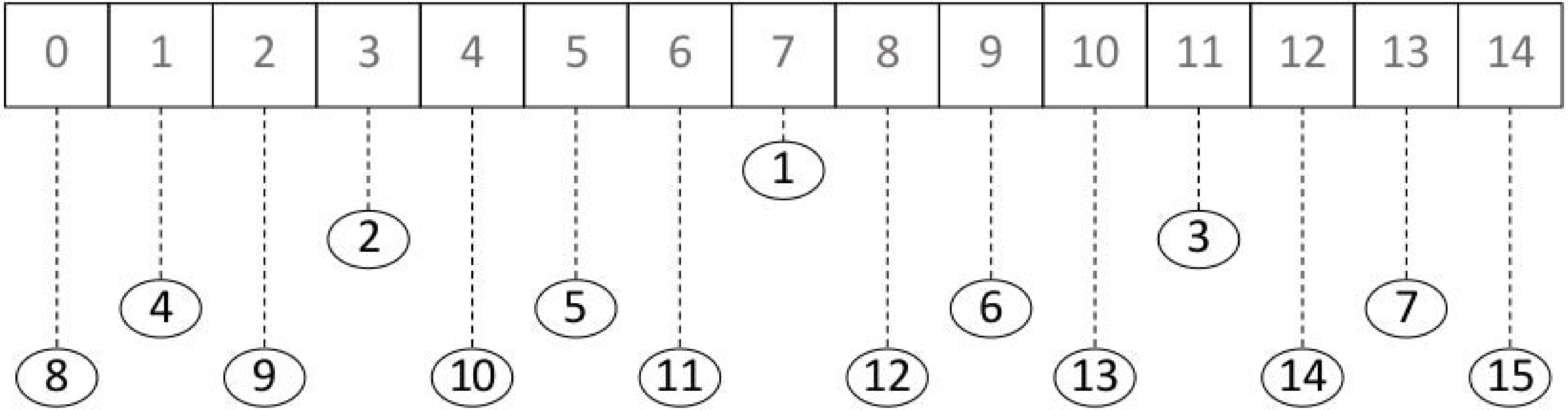


Take element in the middle, then recurse to left and right

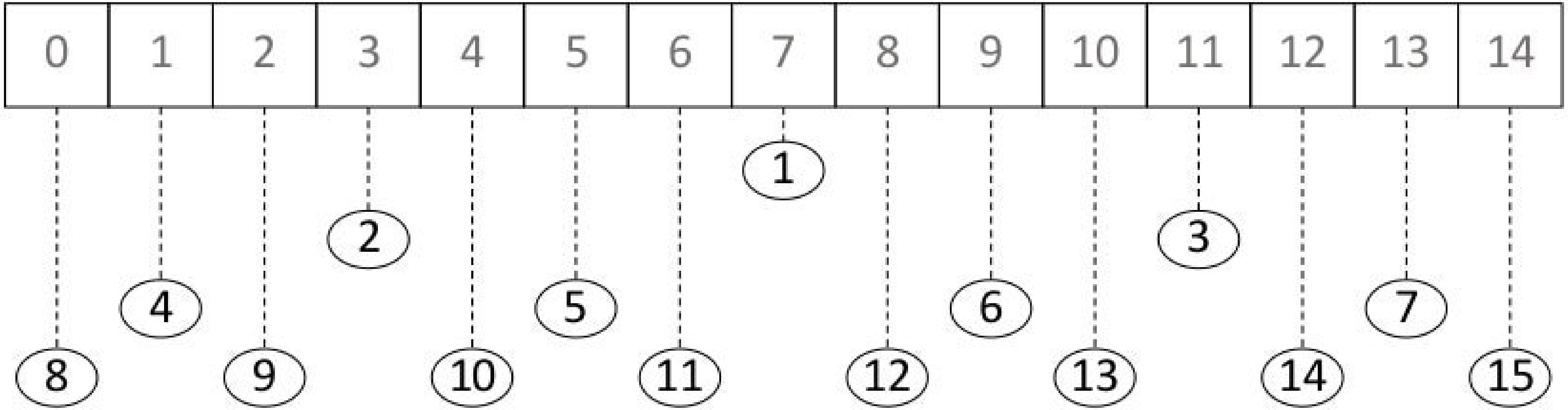


Level	First index	Distances to next
1	6	
2	3	6
3	1	4, 3, 3
4	0	2, 2, 3, 3

Take element in the middle, then recurse to left and right

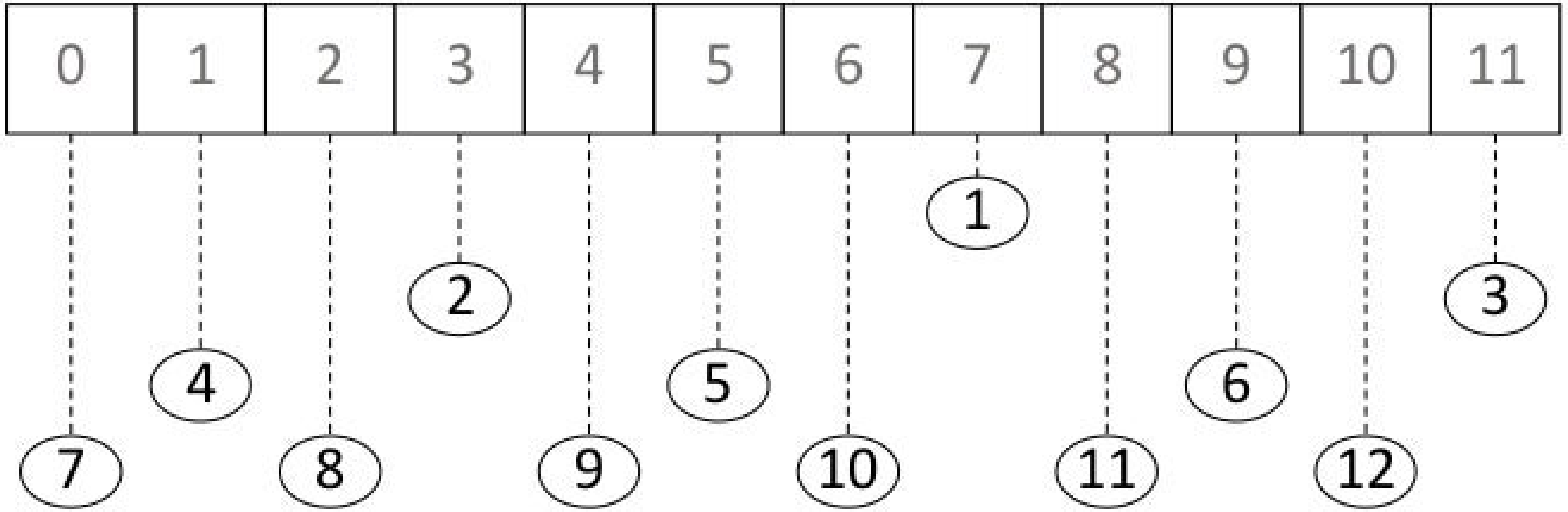


Take element in the middle, then recurse to left and right

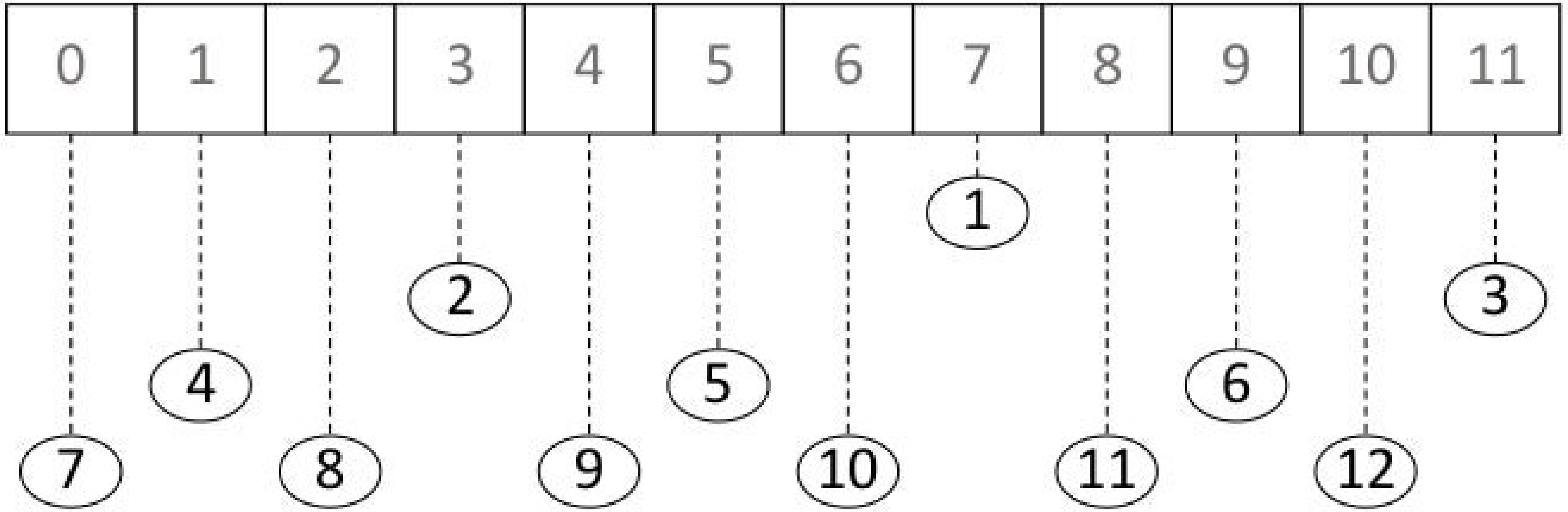


Level	First index	Distances to next
1	7	
2	3	8
3	1	4, 4, 4
4	0	2, 2, 2, 2, 2, 2, 2

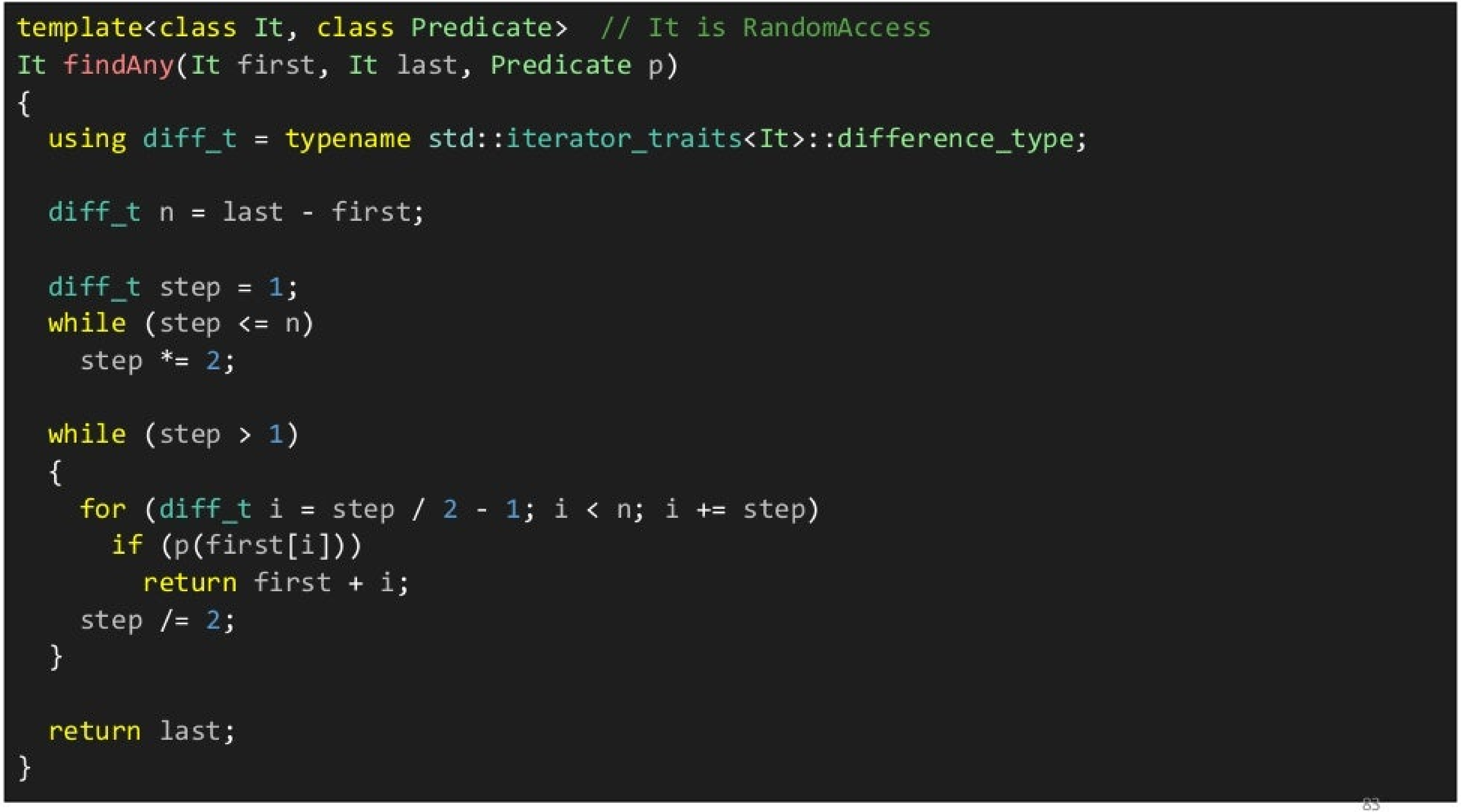
Repeat as if it was for the power of two



Repeat as if it was for the power of two

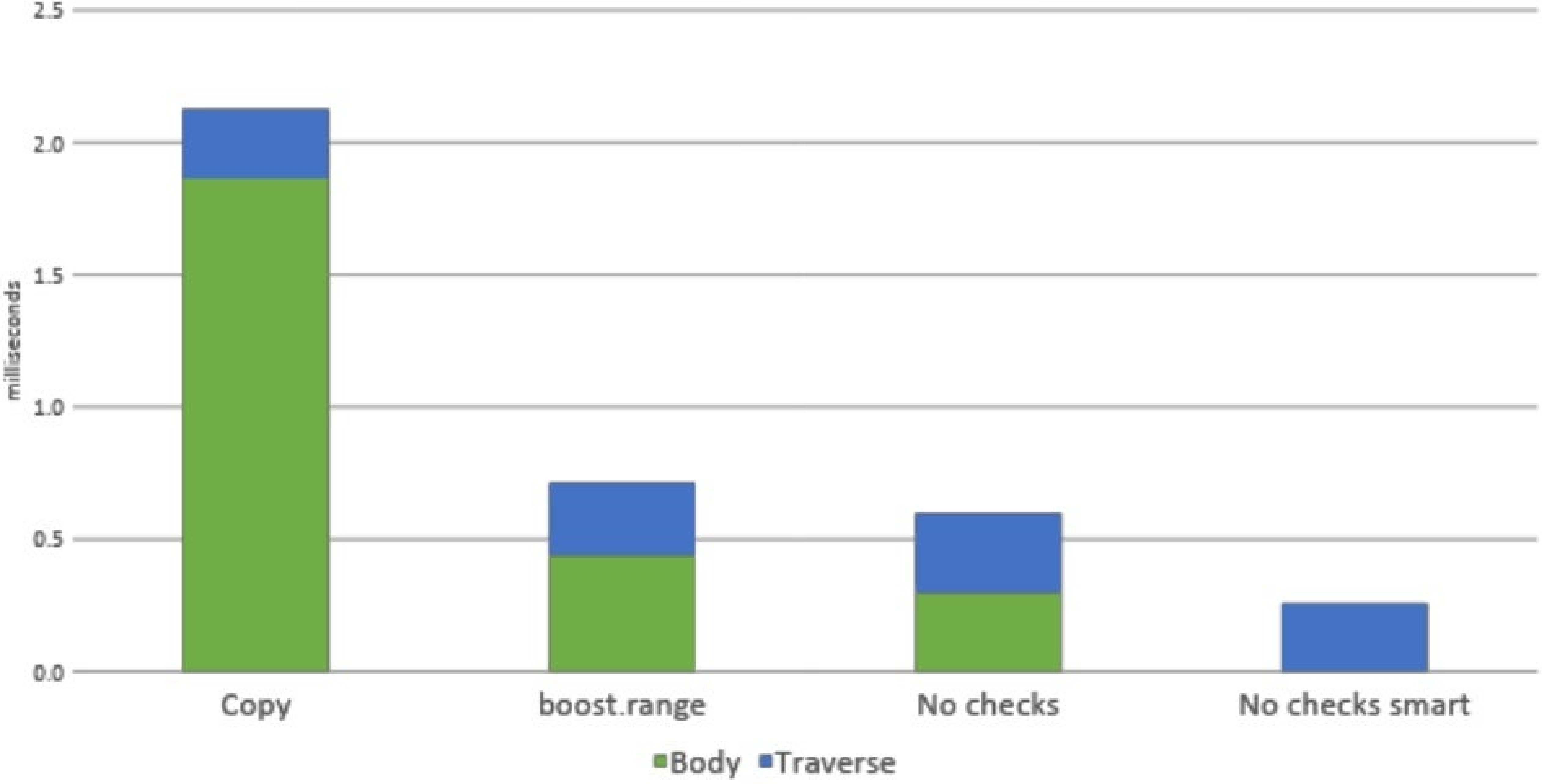


Level	First index	Distances to next
1	7	
2	3	8
3	1	4, 4
4	0	2, 2, 2, 2, 2

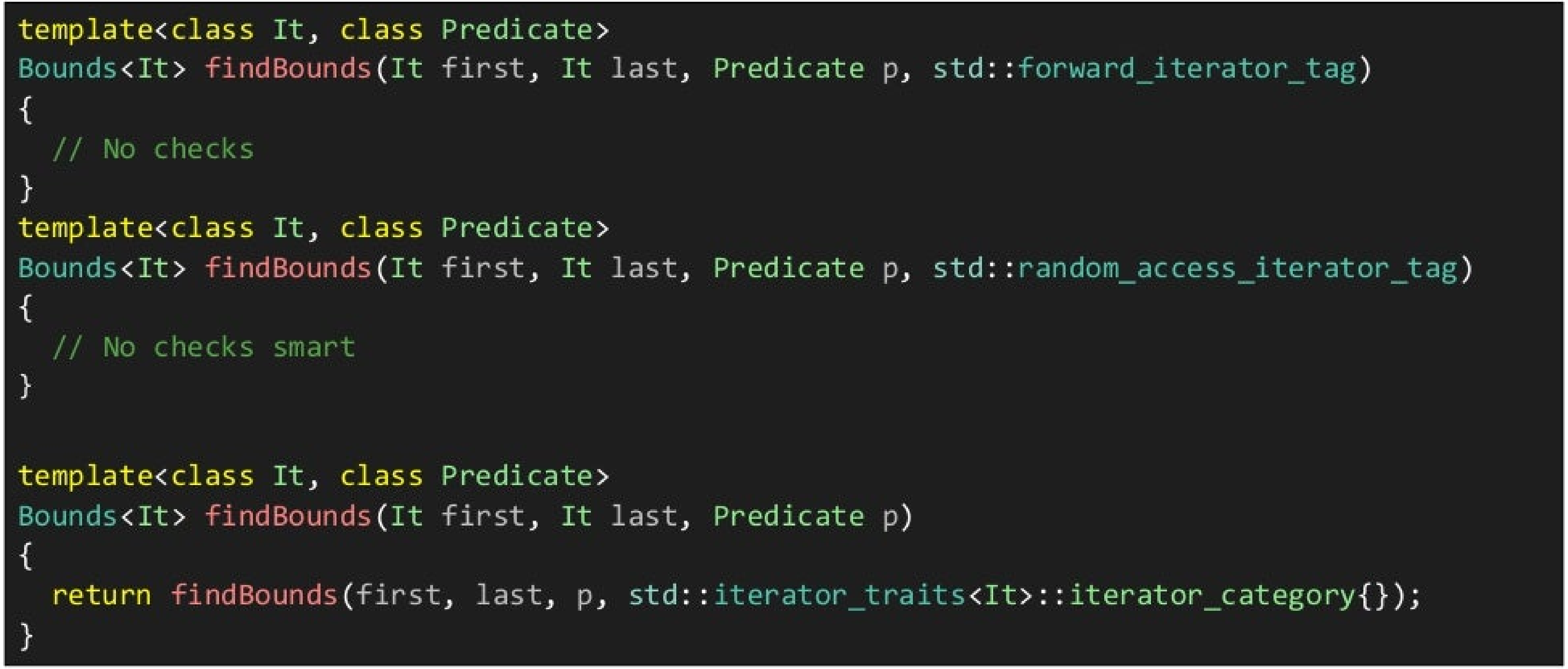


```cpp
template<class It, class Predicate>  // It is RandomAccess
It findAny(It first, It last, Predicate p)
{
  using diff_t = typename std::iterator_traits<It>::difference_type;

  diff_t n = last - first;

  diff_t step = 1;
  while (step <= n)
    step *= 2;

  while (step > 1)
  {
    for (diff_t i = step / 2 - 1; i < n; i += step)
      if (p(first[i]))
        return first + i;
    step /= 2;
  }

  return last;
}
```

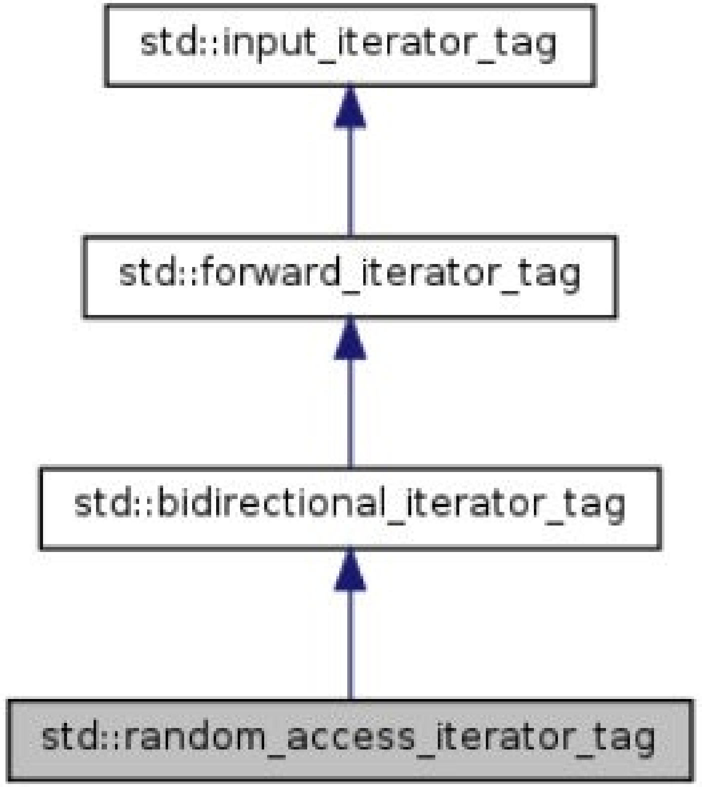
Copy vs. range



Tag dispatching



```
template<class It, class Predicate>
Bounds<It> findBounds(It first, It last, Predicate p, std::forward_iterator_tag)
{
  // No checks
}
template<class It, class Predicate>
Bounds<It> findBounds(It first, It last, Predicate p, std::random_access_iterator_tag)
{
  // No checks smart
}


template<class It, class Predicate>
Bounds<It> findBounds(It first, It last, Predicate p)
{
  return findBounds(first, last, p, std::iterator_traits<It>::iterator_category{});
}
```


std::input_iterator_tag
std::forward_iterator_tag
std::bidirectional_iterator_tag
std::random_access_iterator_tag

Computational complexity

- Total number of elements: N
- Number of positive elements: K
- Number of negative elements: $N - K$

Complexity of findAny()

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Computational complexity

- Total number of elements: N
- Number of positive elements: K
- Number of negative elements: $N - K$
- findAny() for positive: $O\left(\frac{N}{K}\right)$
- findAny() for negative: $O\left(\frac{N}{N-K}\right)$
- std::partition_point: $O(\log N)$
- Traverse: $O(K)$

$$f(N,K) = O\left(\frac{N}{K}\right) + O\left(\frac{N}{N-K}\right) + O(\log N) + O(K)$$

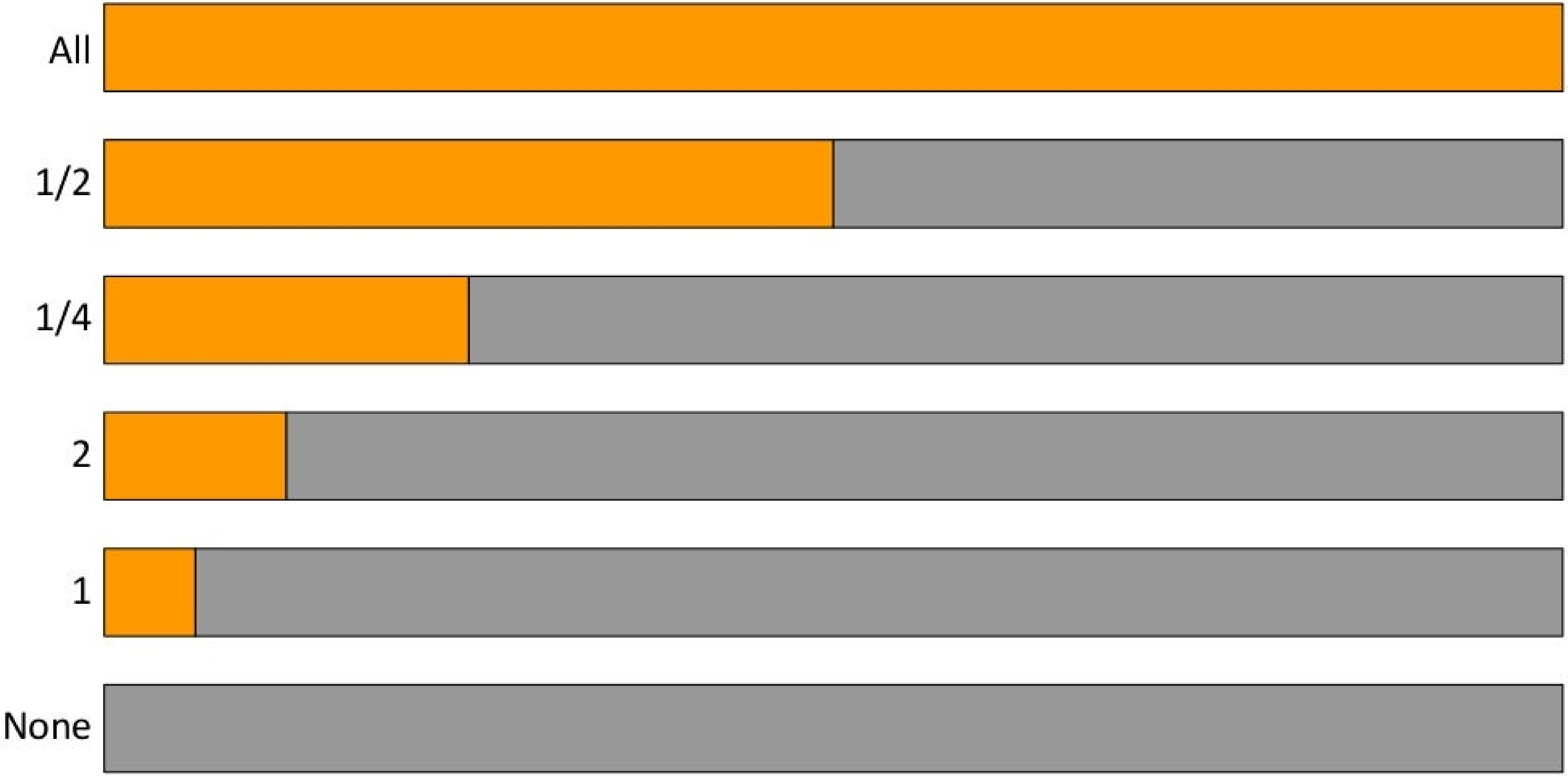
- $$f(N,1) = O(N)$$

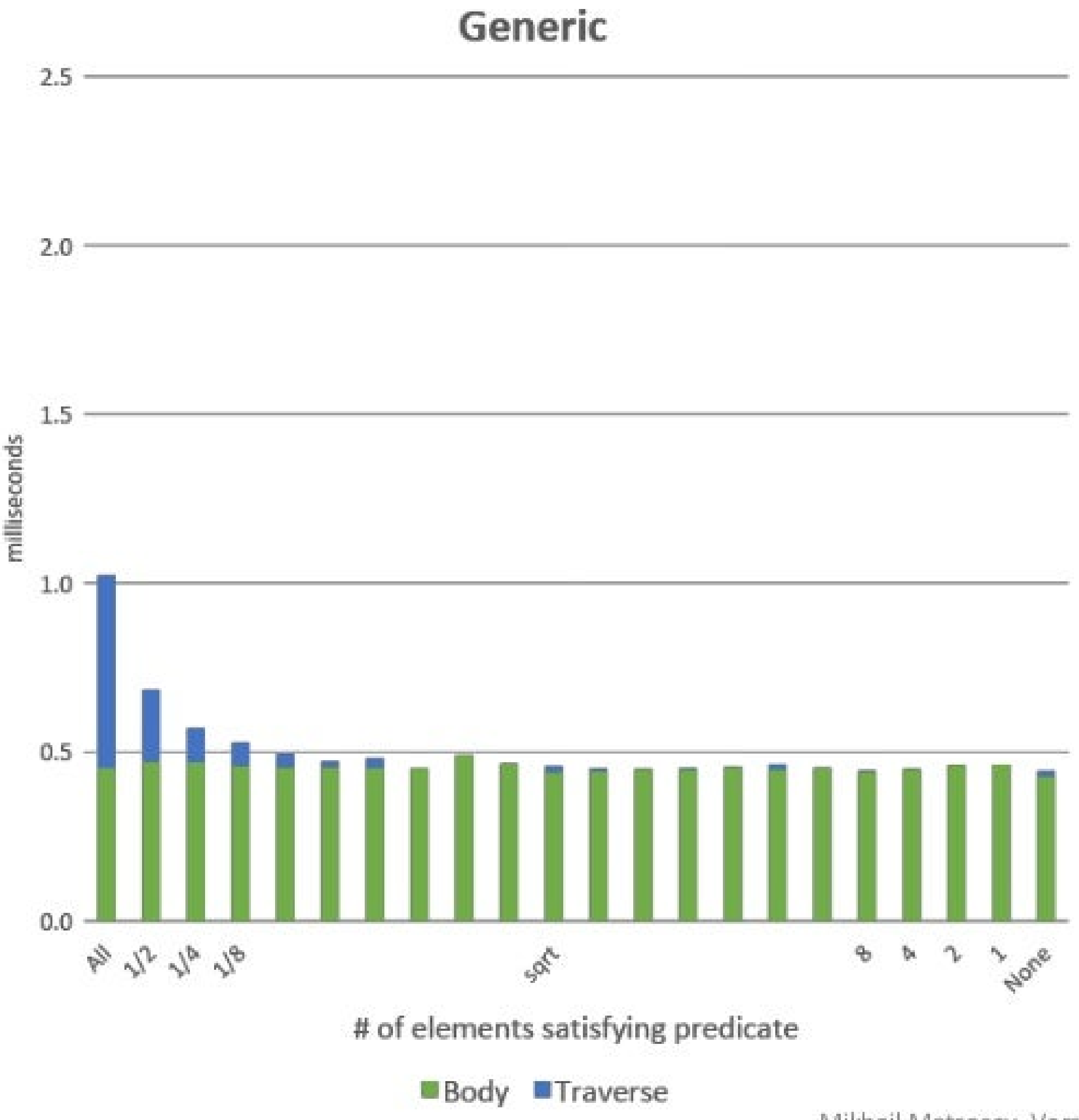
$$f(N,N) = O(N)$$

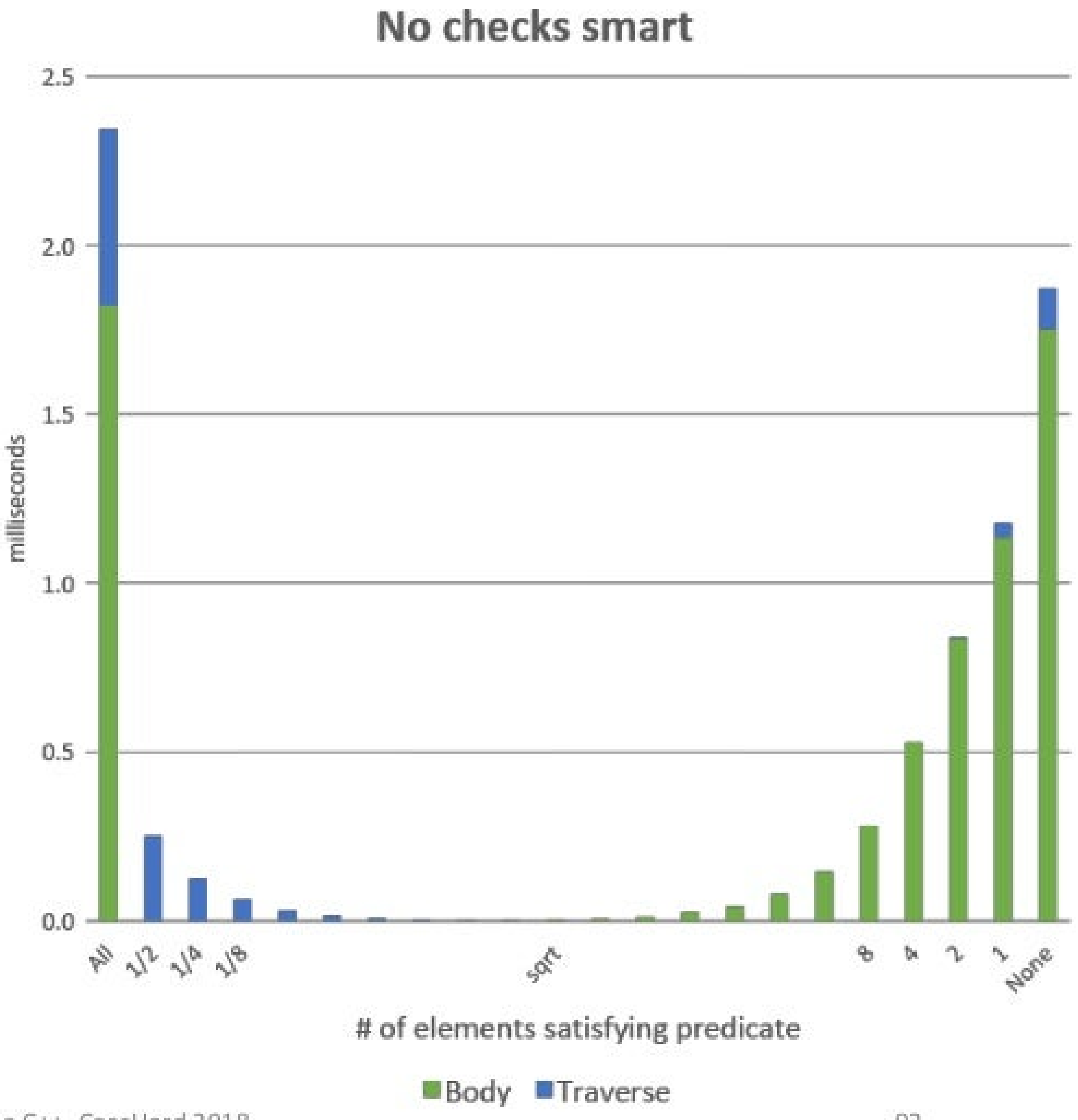
$$f\left(N,\sqrt{N}\right) = O\left(\frac{N}{\sqrt{N}}\right) + O\left(\frac{N}{N-\sqrt{N}}\right) + O(\log N) + O\left(\sqrt{N}\right) \sim$$

$$\sim O\left(\sqrt{N}\right) + O(1) + \bar{\bar{o}}\left(\sqrt{N}\right) + O\left(\sqrt{N}\right) \sim$$

$$O\left(\sqrt{N}\right)$$


All
1/2
1/4
2
1
None


Generic
2.5
2.0
1.5
1.0
0.5
0.0
milliseconds
All
1/2
1/4
1/8
sqrt
8
4
2
1
None
# of elements satisfying predicate
🟩Body
🟦Traverse


No checks smart
2.5
2.0
1.5
1.0
0.5
0.0
milliseconds
All
1/2
1/4
1/8
sqrt
8
4
2
1
None
# of elements satisfying predicate
🟩Body
🟦Traverse

Method	Input	Output	Modify input	Check structure	Assume layout
Copy	const vector&	vector	no	yes	no
Reuse	vector&&	vector	yes	yes	no
Reuse smart	vector&&	vector	yes	yes	no
Reuse std::list	list&&	list	yes	yes	no
boost.range	vector	boost::range::joined_range	no	yes	no
WrappingIterator	vector	boost::iterator_range<WrappingIterator>	no	yes	no
Generic	pair of iterators	boost::range::joined_range	no	yes	no
Range-V3-VS2015	pair of iterators	ranges::concat_view	no	yes	no
VC++ generator	pair of iterators	std::generator	no	yes	no
No checks	pair of iterators	boost::range::joined_range	no	no	no
No checks smart	pair of random-access iterators	boost::range::joined_range	no	no	yes

Take aways

- Analyze! Keyboard after paper
- Recognize patterns, implement them with standard algorithms
- Think about the interface
- Don’t be afraid to go generic
- Use modern C++
- Care about asymptotic complexity
- Measure performance