Филипп Хандельянц

Лекция 6/12

STL: функциональные объекты, алгоритмы

Докладчик

Хандельянц

Филипп Александрович

- Ведущий разработчик в команде PVS-Studio (C++/C#)
- 3 года участвую в разработке ядра C++ анализатора
- Автор статей о проверке open source-проектов

Iterator adaptors

std::reverse_iterator

```cpp
#include <string>
#include <iterator>

void foo()
{
  std::string str { "Hello, World" };

  std::reverse_iterator<std::string::iterator> rit  { str.end() };
  auto rit = std::make_reverse_iterator(str.end()); // since C++14
  std::reverse_iterator rit { str.end() };          // since C++17
  auto rit = str.rbegin();

  std::reverse_iterator<std::string::iterator> rend { str.begin() };
  auto rend = std::make_reverse_iterator(str.begin()); // since C++14
  std::reverse_iterator rend { str.begin() };          // since C++17
  auto rend = str.rend();

  std::string reversed { rit, rend };
}
```

std::back_insert_iterator

```
#include <vector>
#include <functional>
#include <iterator>

std::vector<uint64_t> generate_vector(size_t n, std::function<uint64_t()> g)
{
  std::vector<uint64_t> res;
  res.reserve(n);

  std::generate_n(std::back_insert_iterator<std::vector<uint64_t>> { res },
                  n, g);

  std::generate_n(std::back_inserter(res), n, g);

  std::generate_n(std::back_insert_iterator { res }, n, g); // since C++17

  return res;
}
```

std::front_insert_iterator

```
#include <deque>
#include <functional>
#include <iterator>

std::deque<uint64_t> generate_vector(size_t n, std::function<uint64_t()> g)
{
  std::deque<uint64_t> res;
  res.reserve(n);

  std::generate_n(std::front_insert_iterator<std::deque<uint64_t>> { res },
                  n, g);

  std::generate_n(std::front_inserter(res), n, g);

  std::generate_n(std::front_insert_iterator { res }, n, g); // since C++17

  return res;
}
```

std::insert_iterator

```
#include <vector>
#include <list>
#include <iterator>

void foo()
{
  std::vector<int> v { 1, 2, 3, 4, 5 };

  std::list<int> l { -1, -2, -3 };

  std::copy(v.begin(), v.end(),
            std::insert_iterator<std::list<int>>(l, std::next(l.begin())));

  std::copy(v.begin(), v.end(), std::inserter(l, std::next(l.begin())));

  std::copy(v.begin(), v.end(),
            std::insert_iterator { l, std::next(l.begin()) }); // since C++17
}
```

std::move_iterator

```
#include <string>
#include <vector>
#include <iterator>

void foo()
{
  std::vector<std::string> v1 { "Hello", "My", "Beautiful", "World" };

  std::vector<std::string> v2;
  v2.reserve(v1.size());

  using iter_t = decltype(v1)::iterator;
  std::copy(std::move_iterator<iter_t> { v1.begin() },
            std::move_iterator<iter_t> { v1.end() },
            std::back_inserter(v2));

  // v1 { "" "" "" "" }
  // v2 { "Hello" "My" "Beautiful" "World" }
}
```

std::move_iterator

```
#include <string>
#include <vector>
#include <iterator>

void foo()
{
  std::vector<std::string> v1 { "Hello", "My", "Beautiful", "World" };

  std::vector<std::string> v2;
  v2.reserve(v1.size());

  std::copy(std::move_iterator { v1.begin() }, // since C++17
            std::move_iterator { v1.end() },   // since C++17
            std::back_inserter(v2));

  // v1 { "" "" "" "" }
  // v2 { "Hello" "My" "Beautiful" "World" }
}
```

Function objects

```
#include <vector>

void foo(std::vector<int> v)
{
  std::sort(v.begin(), v.end());       // Ascending order
  std::sort(v.begin(), v.end(), ???); // Descending order
}
```

```
#include <vector>

template <class T>
bool greater(const T &lhs, const T &rhs)
{
  return lhs > rhs;
}

void foo(std::vector<int> v)
{
  std::sort(v.begin(), v.end());          // Ascending order
  std::sort(v.begin(), v.end(), greater); // Descending order
}
```

```
// <functional>

template <class T>
struct greater
{
  bool operator()(const T &lhs, const T &rhs)
  {
    return lhs > rhs;
  }
}

#include <vector>

void foo(std::vector<int> v)
{
  std::sort(v.begin(), v.end());                  // Ascending order
  std::sort(v.begin(), v.end(), greater<int> {}); // Descending order
}
```

Type	Name	Function object type	Return type	Equivalent
Comparisons	equal_to	Binary	bool	lhs == rhs
	not_equal_to	Binary	bool	lhs != rhs
	less	Binary	bool	lhs < rhs
	less_equal	Binary	bool	lhs <= rhs
	greater	Binary	bool	lhs > rhs
	greater_equal	Binary	bool	lhs >= rhs
Logical	logical_and	Binary	bool	lhs && rhs
	logical_or	Binary	bool	lhs || rhs
	logical_not	Unary	bool	!op
Arithmetic	plus	Binary	T	lhs + rhs
	minus	Binary	T	lhs – rhs
	multiplies	Binary	T	lhs * rhs
	divides	Binary	T	lhs / rhs
	modulus	Binary	T	lhs % rhs
	negate	Unary	T	-op
Bitwise	bit_and	Binary	T	lhs & rhs
	bit_or	Binary	T	lhs | rhs
	bit_xor	Binary	T	lhs ^ rhs
	bit_not	Unary	T	~op

std::hash

```
#include <unordered_set> // or <multiset>

template <class Key,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator<Key>>
class unordered_set // or unordered_multiset
{
  using key_type = Key;
  using value_type = Key;
  using key_compare = Compare;
  using value_compare = Compare;

  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

```
template <class Key>
struct hash;
```

```
template <class Key>
struct hash;
```

```
template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<int8_t>;
template<> struct hash<uint8_t>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<int16_t>;
template<> struct hash<uint16_t>;
template<> struct hash<int32_t>;
template<> struct hash<uint32_t>;
template<> struct hash<long>;
template<> struct hash<unsigned long>;
template<> struct hash<int64_t>;
template<> struct hash<uint64_t>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;

template <class T> struct hash<T*>;
```

```
template<> std::hash<std::string>
template<> std::hash<std::u8string>
template<> std::hash<std::u16string>
template<> std::hash<std::u32string>
template<> std::hash<std::wstring>
template<> std::hash<std::error_code>
template<> std::hash<std::bitset>
template<> std::hash<std::unique_ptr>
template<> std::hash<std::shared_ptr>
template<> std::hash<std::type_index>
template<> std::hash<std::vector<bool>>
template<> std::hash<std::thread::id>
template<> std::hash<std::optional>
template<> std::hash<std::variant>
template<> std::hash<std::string_view>
template<> std::hash<std::wstring_view>
template<> std::hash<std::u8string_view>
template<> std::hash<std::u16string_view>
template<> std::hash<std::u32string_view>
```

```
struct Person
{
  std::string m_name, m_surname;
  uint8_t m_age;
};
```

```
struct Person
{
  std::string m_name, m_surname;
  uint8_t m_age;
};

template <typename T, typename ...Rest>
size_t hash_combine(const T &obj1, const Rest &...objN)
{
  size_t res = 0;

  res ^= std::hash<T>{}(obj1) + 0x9e3779b9 + (res << 6) + (res >> 2);
  ( ... , (res ^= std::hash<Rest>{}(objN) + 0x9e3779b9 + (res << 6) + (res >> 2)) );

  return res;
}
```

```
struct Person
{
  std::string m_name, m_surname;
  uint8_t m_age;
};

template <typename T, typename ...Rest>
size_t hash_combine(const T &obj1, const Rest &...objN);

namespace std
{
  template <>
  struct hash<Person>
  {
    size_t operator()(const SomeClass &obj) const noexcept
    {
      return hash_combine(obj.m_name, obj.m_surname, obj.m_age);
    }
  };
}
```

```
void foo()
{
  Person obj { "Phillip", "Khandeliants", 24 };

  const auto hash_value = std::hash<decltype(obj)>{}(obj);
}
```

Partial function application

```
void f1(int n1, int n2, int n3, const int &n4, int n5)
{
  std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

// void f2(int n1, int n3)
// f2(n1, n3) ≡ f1(n1, 42, n3, var, 24)
```

```
// <functional>

template <class F, class ...Args>
/*unspecified*/ bind(F&& f, Args &&...args);

template <class R, class F, class ...Args>
/*unspecified*/ bind(F&& f, Args &&...args);

std::placeholders::_1,
std::placeholders::_2,
...,
std::placeholders::_N
```

```cpp
#include <functional>

void f1(int n1, int n2, int n3, const int &n4, int n5)
{
  std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

// void f2(int n1, int n3)
// f2(n1, n3) ≡ f1(n1, 42, n3, var, 24)

void foo()
{
  using namespace std::placeholders;  // for _1, _2, _3...

  int n = 7;
  auto f2 = std::bind(f1, _1, 42, _2, std::cref(n), 24);
  f2(1, 2, 3); // 1 is bound by _1, 2 is bound by _2, 3 is unused
               // makes a call to f1(1, 42, 2, n, 24)
}
```

```cpp
#include <functional>
#include <random>

std::vector<int64_t> foo(int64_t a, int64_t b, size_t n)
{
  std::mt19937_64 engine { std::random_device{}() };
  std::uniform_int_distribution<int64_t> distr { a, b };

  std::vector<int64_t> res;
  res.reserve(n);

  std::generate_n(std::back_inserter(res), n,
                  [distr, engine]() mutable { return distr(engine); })

  // std::generate_n(std::back_inserter(res), n,
  //                 [&distr, &engine] { return distr(engine); })

  return res;
}
```

```cpp
#include <functional>
#include <random>

std::vector<int64_t> foo(int64_t a, int64_t b, size_t n)
{
  std::mt19937_64 engine { std::random_device{}() };
  std::uniform_int_distribution<int64_t> distr { a, b };

  std::vector<int64_t> res;
  res.reserve(n);

  std::generate_n(std::back_inserter(res), n,
                  std::bind(distr, engine));

  // std::generate_n(std::back_inserter(res), n,
  //                 std::bind(std::ref(distr), std::ref(engine)));

  return res;
}
```

algorithms

Зачем использовать?

- Делает код чище и читабельнее

```cpp
bool doCompare(const Bakery* oldBakery, const Bakery* newBakery)
{
  if (!oldBakery || !newBakery)
  {
    return false;
  }

  std::vector<Cookie> oldCookies;
  std::vector<Cookie> newCookies;

  collectCookies(*oldBakery, oldCookies);
  collectCookies(*newBakery, newCookies);

  std::sort(oldCookies.begin(), oldCookies.end(), compareCookies);
  std::sort(newCookies.begin(), newCookies.end(), compareCookies);

  auto oldIt = oldCookies.begin();
  auto newIt = newCookies.begin();

  std::vector<std::pair<Cookie, int>> diff;

  while ((oldIt != oldCookies.end()) || (newIt != newCookies.end()))
  {
    if (compareCookies(*oldIt, *newIt))
    {
      diff.push_back(std::pair<Cookie, int>(*oldIt, 1));
      ++oldIt;
    }
    else if (compareCookies(*newIt, *oldIt))
    {
      diff.push_back(std::pair<Cookie, int>(*newIt, 2));
      ++newIt;
    }
    else
    {
      ++oldIt;
      ++newIt;
    }
  }

  if (oldIt != oldCookies.end())
  {
    for (; oldIt < oldCookies.end(); oldIt++)
    {
      diff.push_back(std::pair<Cookie, int>(*oldIt, 1));
    }
  }

  if (newIt != newCookies.end())
  {
    for (; newIt < newCookies.end(); newIt++)
    {
      diff.push_back(std::pair<Cookie, int>(*newIt, 2));
    }
  }

  for (const auto& item : diff)
  {
    std::cout << "Bakery #" << item.second <<
      " has a different cookie \"" << item.first.name <<
      "\" (weight=" << item.first.weight <<
      ", volume=" << item.first.volume <<
      ", tastiness=" << item.first.tastiness << ")." << std::endl;
  }

  if (diff.size() > 0)
    return false;
  return true;
}
```

```cpp
bool doCompare(const Bakery& oldBakery, const Bakery& newBakery)
{
  std::vector<Cookie> oldCookies = collectSortedCookies(oldBakery);
  std::vector<Cookie> newCookies = collectSortedCookies(newBakery);

  std::vector<Cookie> lostCookies;
  boost::range::set_difference(oldCookies, newCookies, std::back_inserter(lostCookies));

  std::vector<Cookie> appearedCookies;
  boost::range::set_difference(newCookies, oldCookies, std::back_inserter(appearedCookies));

  for (const auto& cookie : lostCookies)
  {
    std::cout << "We've lost a friend: " << cookie << "." << std::endl;
  }
  for (const auto& cookie : appearedCookies)
  {
    std::cout << "We got a new friend: " << cookie << "." << std::endl;
  }

  return lostCookies.empty() && appearedCookies.empty();
}
```

Зачем использовать?

- Делает код чище и читабельнее
- **Меньше ошибок, связанных с границами**

Зачем использовать?

- Делает код чище и читабельнее
- Меньше ошибок, связанных с границами
- **Написаны гораздо лучше, чем собственные велосипеды**

Зачем использовать?

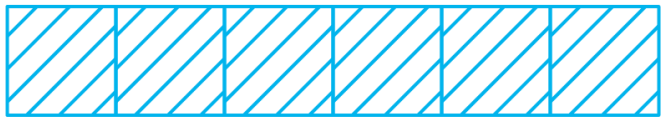
- Делает код чище и читабельнее
- Меньше ошибок, связанных с границами
- Написаны гораздо лучше, чем собственные велосипеды
- **Есть документация ☺**


Province of Value Queries
Province of Property Queries
Province of 2-ranges properties
Glorious County of Algos on Sets
Lonely Islands
Lands of Queries
Peninsula of Raw Memory
Territory of Movers
Province of Research
Relative Value Searchers
Island of Structure Changers
Land of Value Modifiers
Secret Runes
Lands of Permutations
The World of C++ STL Algorithms
Fluent C++
Fluent C++
fluentcpp.com

Non-modifying sequence algorithms

std::*_of

```
template <class InputIt, class UnaryPredicate>
bool all_of(InputIt first, InputIt last, UnaryPredicate p);

template <class InputIt, class UnaryPredicate>
bool any_of(InputIt first, InputIt last, UnaryPredicate p);

template <class InputIt, class UnaryPredicate>
bool none_of(InputIt first, InputIt last, UnaryPredicate p);
```



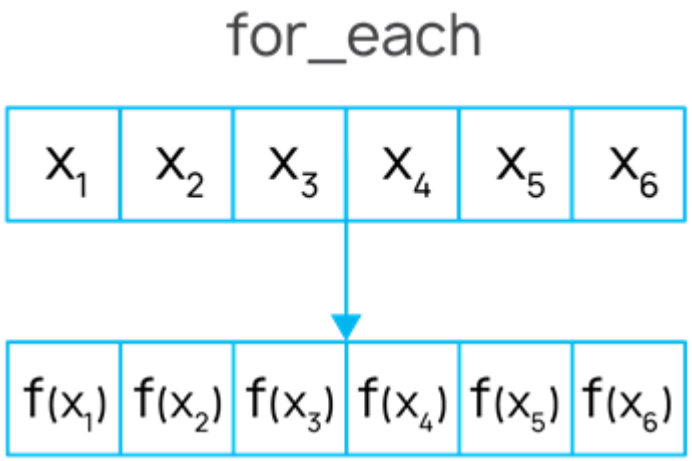
all_of

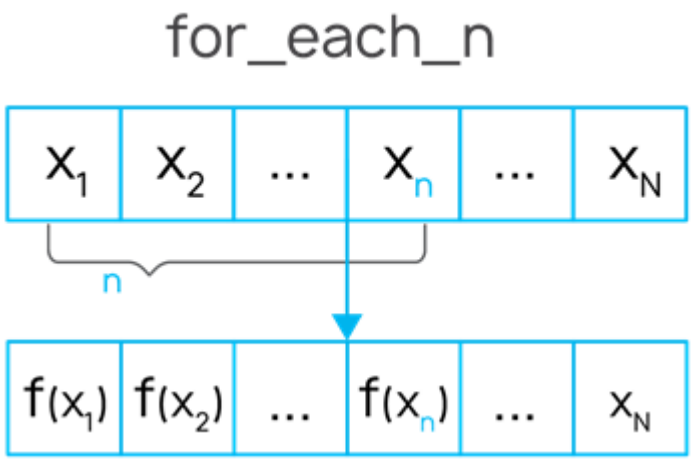
any_of

none_of

std::for_each

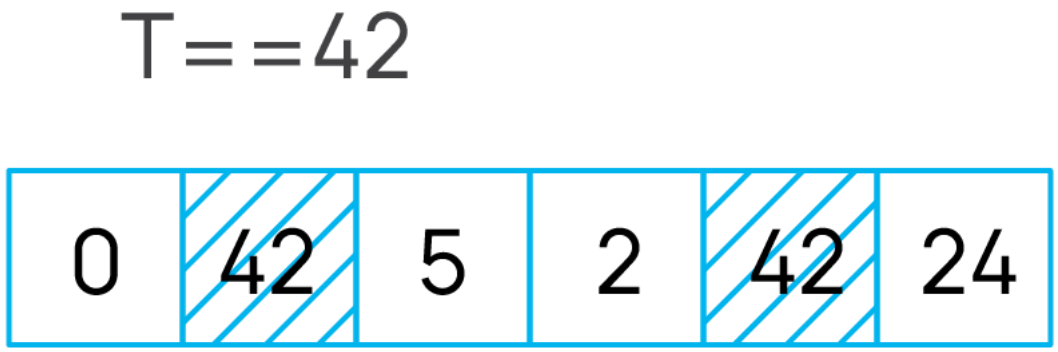
```
template <class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f);

template <class InputIt, class Size, class UnaryFunction>
InputIt for_each_n(InputIt first, Size n, UnaryFunction f);
```





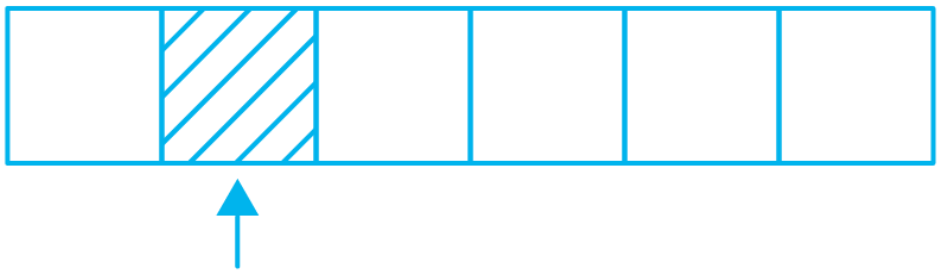
std::count

```
template <class InputIt, class T>
difference_type count(InputIt first, InputIt last, const T &value);

template <class InputIt, class UnaryPredicate>
difference_type count_if(InputIt first, InputIt last, UnaryPredicate p);
```



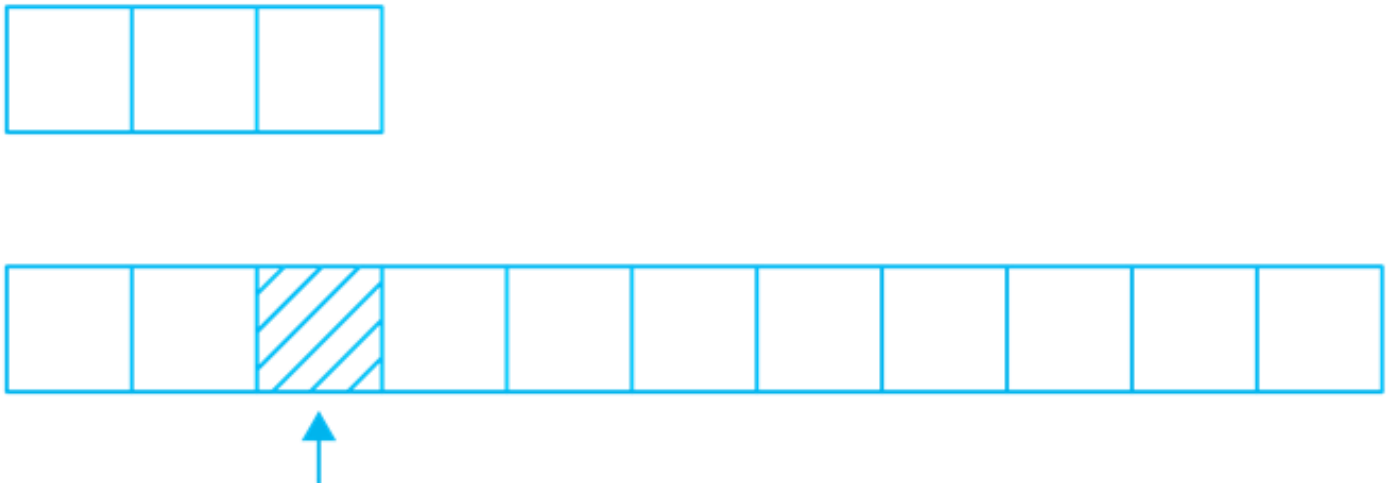
std::find

```
template <class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T &value);

template <class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p);

template <class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q);
```



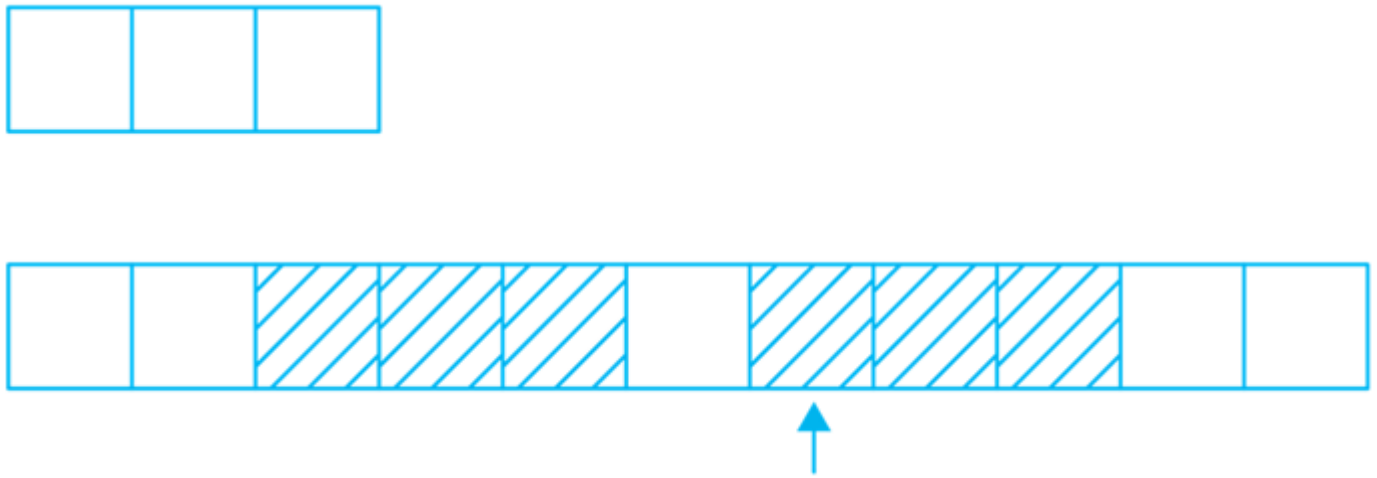
std::find_first_of

```
template <class InputIt, class ForwardIt>
InputIt find_first_of(InputIt first, InputIt last,
                      ForwardIt s_first, ForwardIt s_last );

template <class InputIt, class ForwardIt, class BinaryPredicate>
InputIt find_first_of(InputIt first, InputIt last,
                       ForwardIt s_first, ForwardIt s_last, BinaryPredicate p);
```



std::find_end

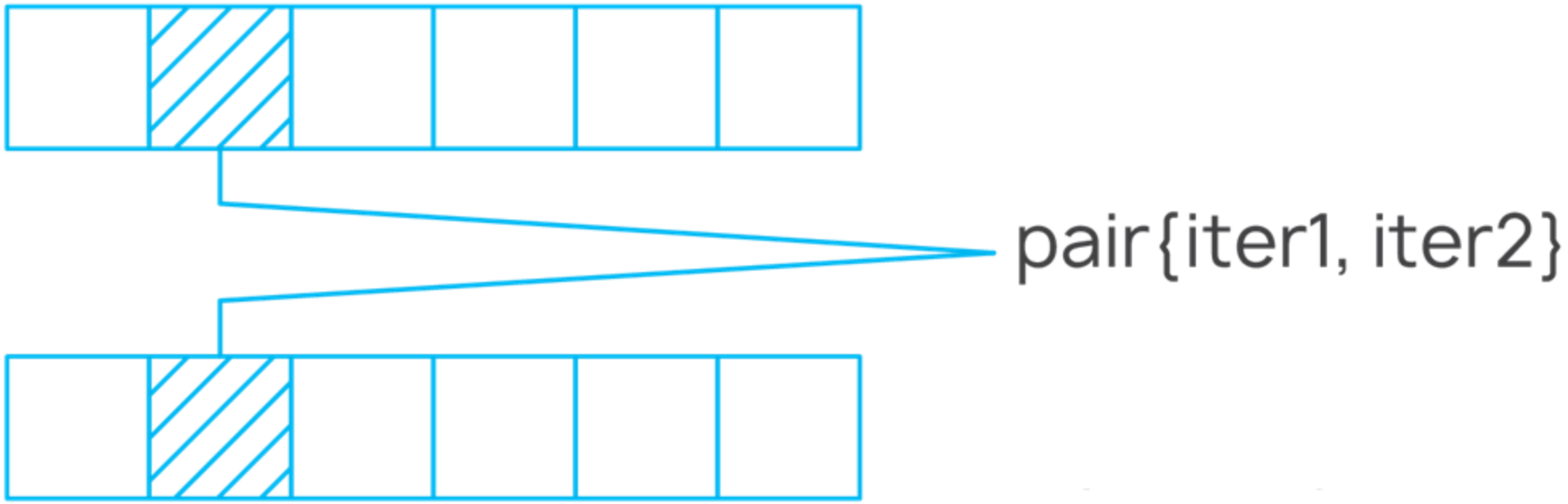
```
template <class ForwardIt1, class ForwardIt2>
ForwardIt1 find_end(ForwardIt1 first, ForwardIt1 last,
                    ForwardIt2 s_first, ForwardIt2 s_last);

template <class ForwardIt1, class ForwardIt2, class BinaryPredicate>
ForwardIt1 find_end(ForwardIt1 first, ForwardIt1 last,
                    ForwardIt2 s_first, ForwardIt2 s_last, BinaryPredicate p);
```



std::mismatch

```
template <class InputIt1, class InputIt2>
std::pair<InputIt1, InputIt2> mismatch(InputIt1 first1, InputIt1 last1,
                                       InputIt2 first2);

template <class InputIt1, class InputIt2, class BinaryPredicate>
std::pair<InputIt1, InputIt2> mismatch(InputIt1 first1, InputIt1 last1,
                                       InputIt2 first2, BinaryPredicate p);

template <class InputIt1, class InputIt2>
std::pair<InputIt1, InputIt2> mismatch(InputIt1 first1, InputIt1 last1,
                                       InputIt2 first2, InputIt2 last2);

template <class InputIt1, class InputIt2, class BinaryPredicate>
std::pair<InputIt1,InputIt2> mismatch(InputIt1 first1, InputIt1 last1,
                                      InputIt2 first2, InputIt2 last2,
                                      BinaryPredicate p);
```

std::mismatch

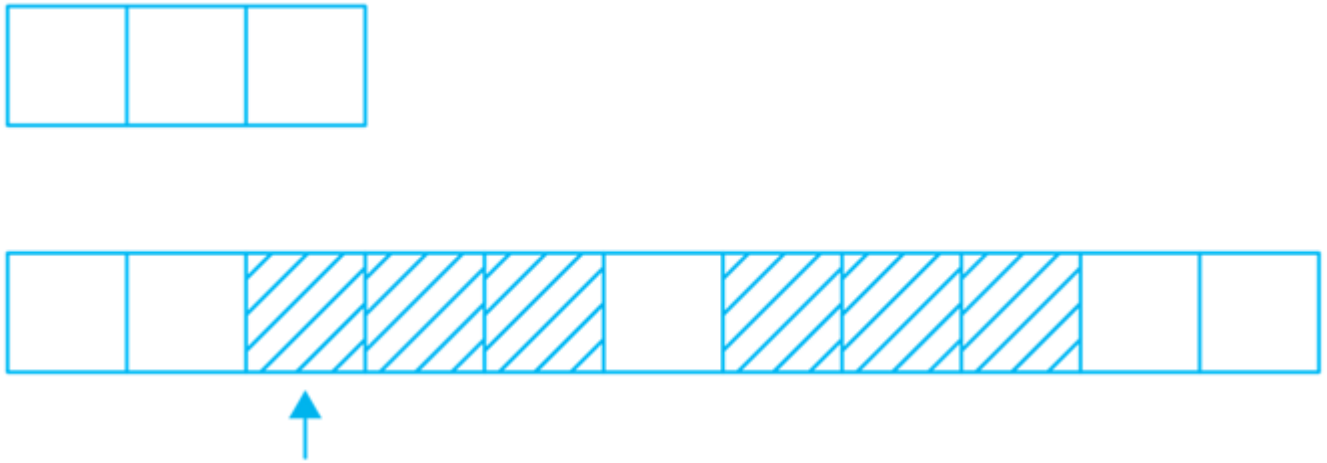


std::adjacent_find

```
template <class ForwardIt>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last);

template <class ForwardIt, class BinaryPredicate>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last,
                        BinaryPredicate p);
```

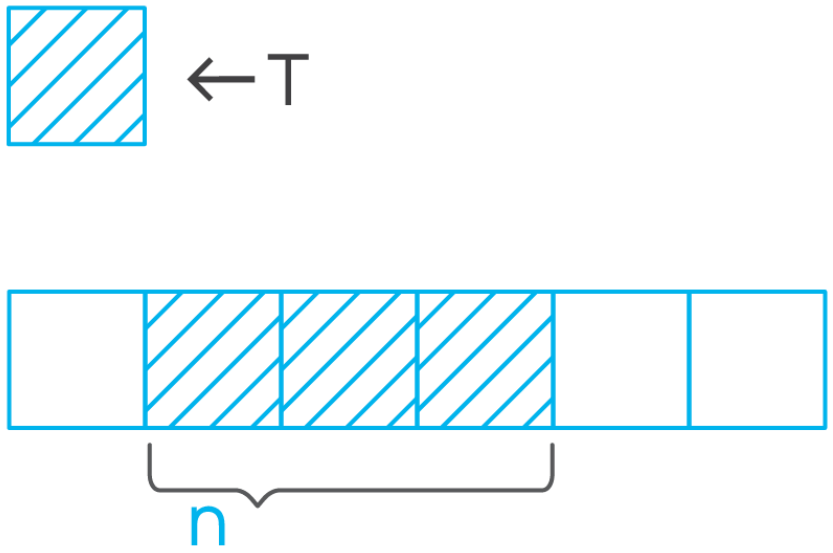
std::search

```
template <class ForwardIt1, class ForwardIt2>
ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
                  ForwardIt2 s_first, ForwardIt2 s_last);

template <class ForwardIt1, class ForwardIt2, class BinaryPredicate>
ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
                  ForwardIt2 s_first, ForwardIt2 s_last, BinaryPredicate p);

template <class ForwardIt, class Searcher>
ForwardIt search(ForwardIt first, ForwardIt last, const Searcher &searcher);
```



std::search_n

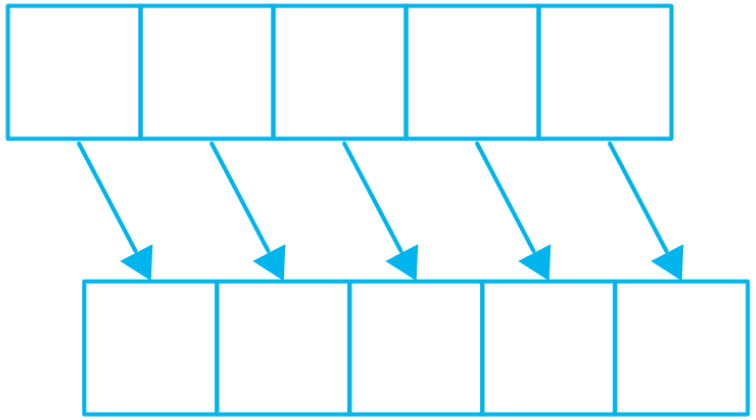
```
template <class ForwardIt, class Size, class T >
ForwardIt search_n(ForwardIt first, ForwardIt last,
                   Size count, const T &value);

template <class ForwardIt, class Size, class T, class BinaryPredicate>
ForwardIt search_n(ForwardIt first, ForwardIt last,
                   Size count, const T &value, BinaryPredicate p);
```



Modifying sequence algorithms

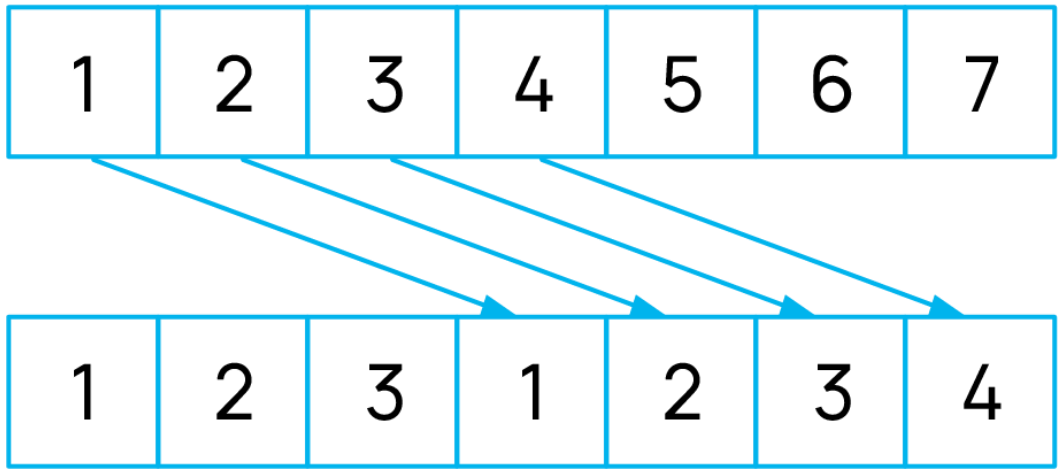
std::copy

```
template <class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first);

template <class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first, UnaryPredicate pred);

template <class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt result);
```



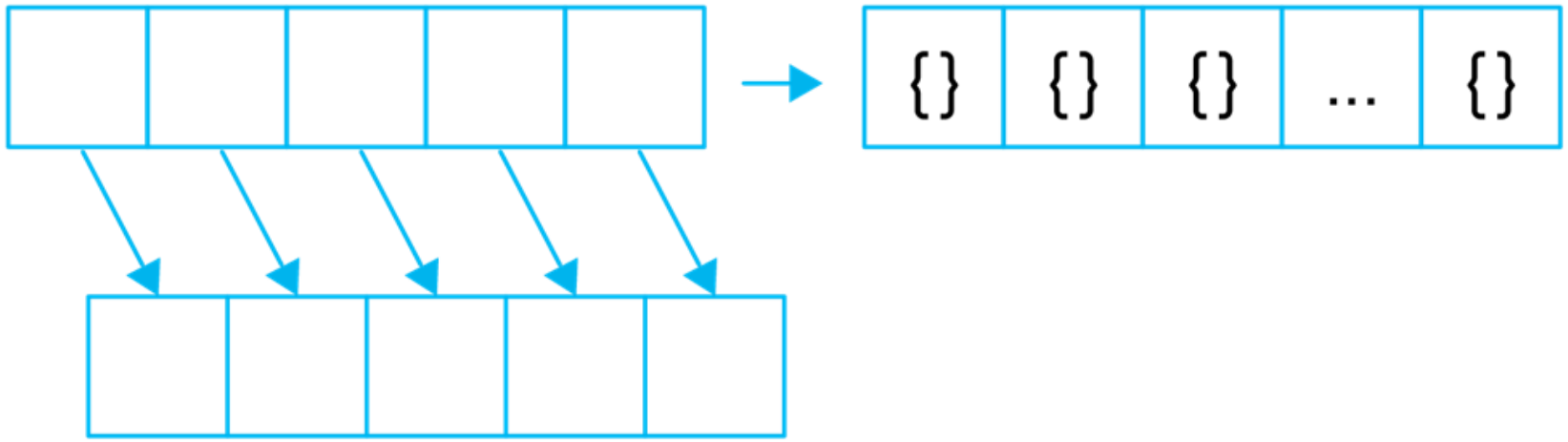
std::copy_backward

```
template <class BidirIt1, class BidirIt2>
BidirIt2 copy_backward(BidirIt1 first, BidirIt1 last, BidirIt2 d_last);
```



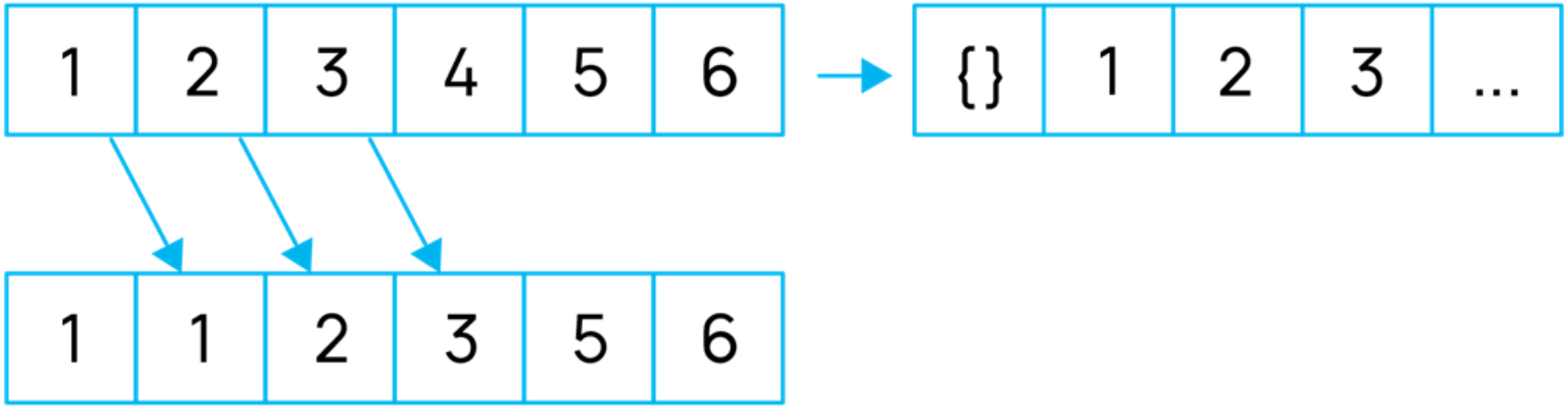
std::move

```
template <class InputIt, class OutputIt>
OutputIt move(InputIt first, InputIt last, OutputIt d_first);
```



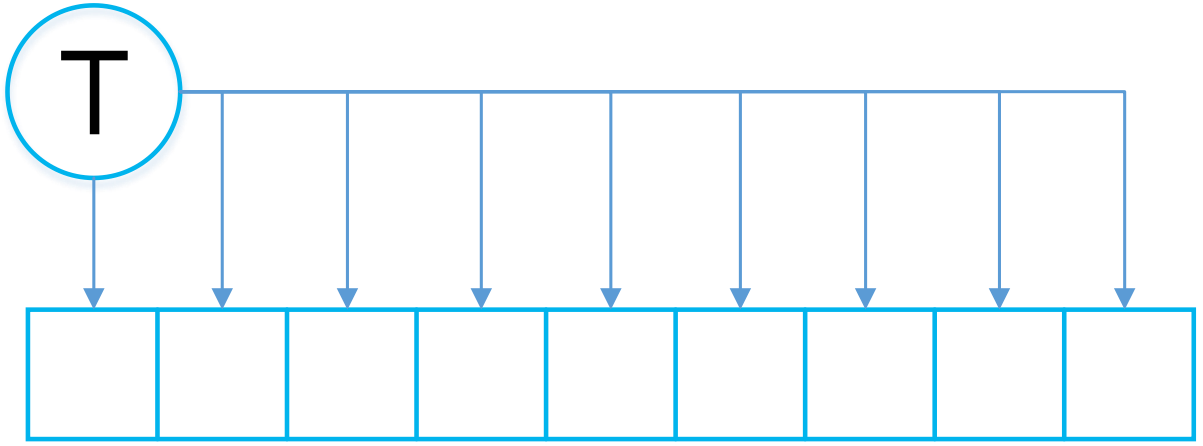
std::move_backward

```
template <class BidirIt1, class BidirIt2>
BidirIt2 move_backward(BidirIt1 first, BidirIt1 last, BidirIt2 d_last);
```



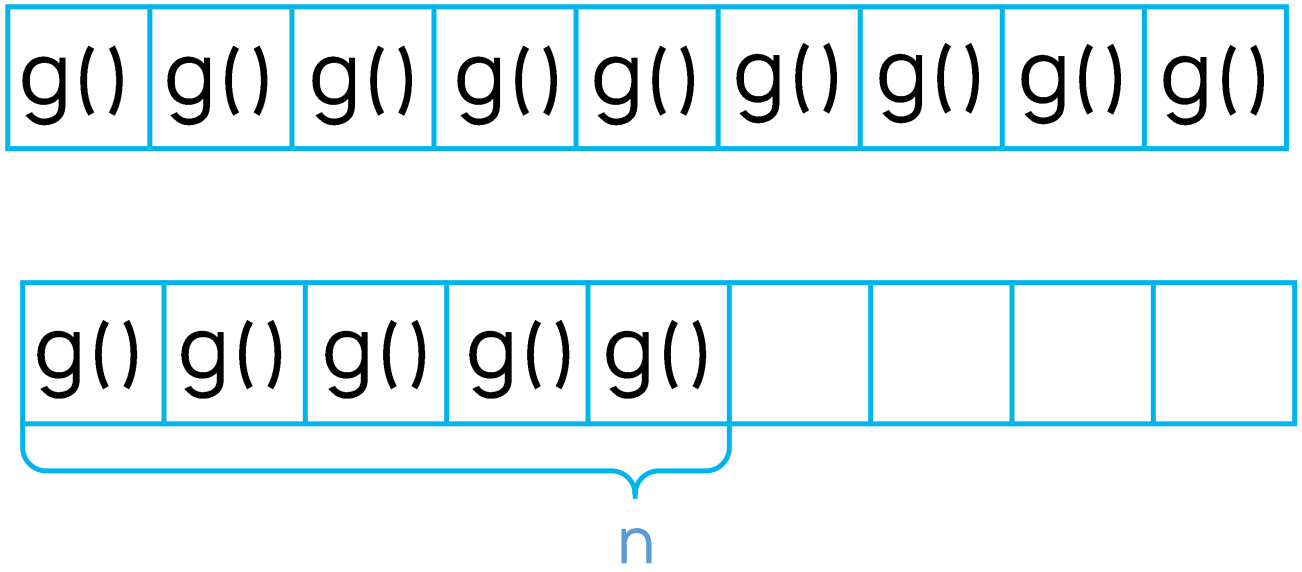
std::fill

```
template <class ForwardIt, class T>
void fill(ForwardIt first, ForwardIt last, const T &value);

template <class OutputIt, class Size, class T>
OutputIt fill_n(OutputIt first, Size count, const T &value);
```



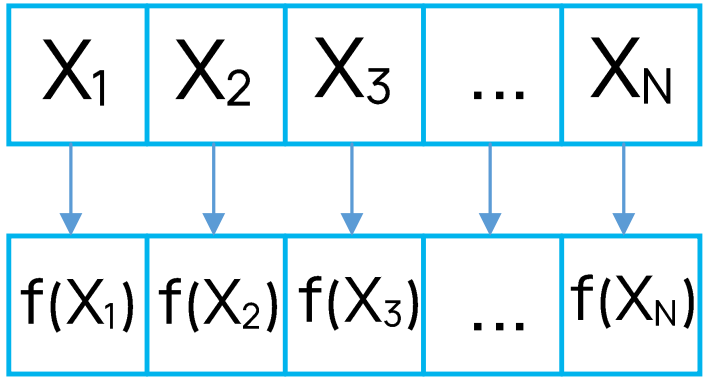
std::generate

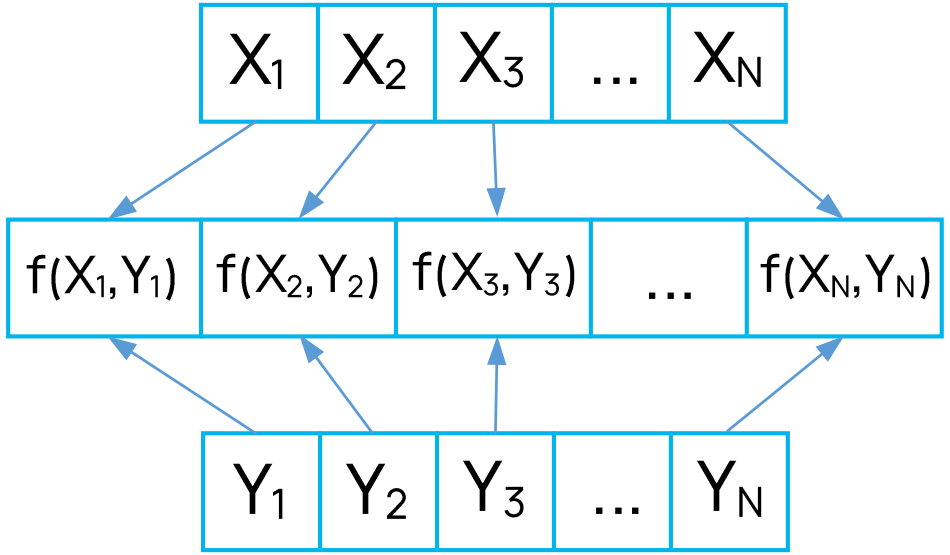
```
template <class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g);

template <class OutputIt, class Size, class Generator>
OutputIt generate_n(OutputIt first, Size count, Generator g);
```



std::transform

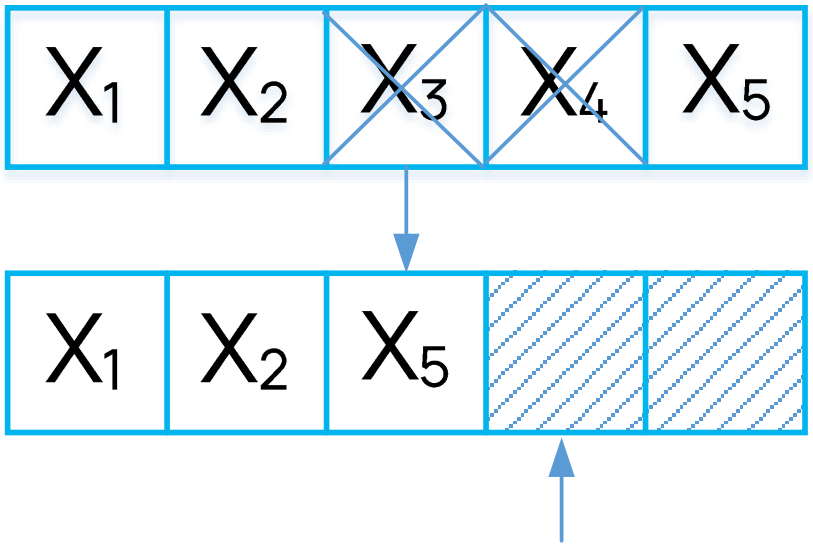
```
template <class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first,
                   UnaryOperation op);

template <class InputIt1, class InputIt2, class OutputIt, class BinaryOperation>
OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   OutputIt d_first, BinaryOperation op);
```





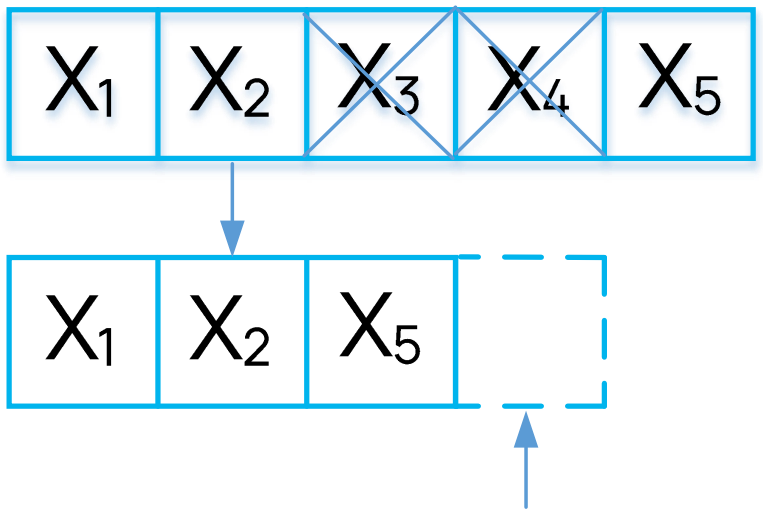
std::remove

```
template <class ForwardIt, class T>
ForwardIt remove(ForwardIt first, ForwardIt last, const T &value);

template <class ForwardIt, class UnaryPredicate>
ForwardIt remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p);
```



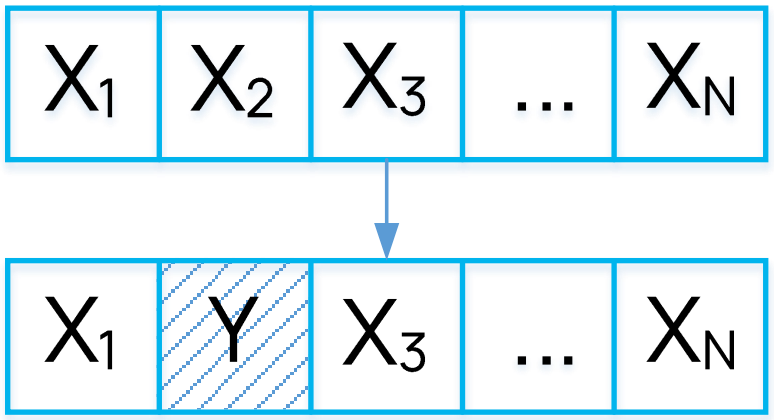
std::remove_copy

```
template <class InputIt, class OutputIt, class T>
OutputIt remove_copy(InputIt first, InputIt last, OutputIt d_first,
                     const T& value);

template <class InputIt, class OutputIt, class UnaryPredicate>
OutputIt remove_copy_if(InputIt first, InputIt last,
                        OutputIt d_first, UnaryPredicate p);
```



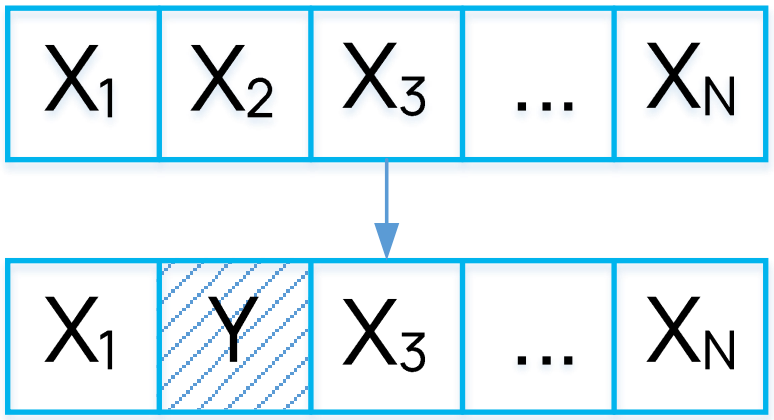
std::replace

```
template <class ForwardIt, class T>
void replace(ForwardIt first, ForwardIt last,
             const T &old_value, const T &new_value);

template <class ForwardIt, class UnaryPredicate, class T>
void replace_if(ForwardIt first, ForwardIt last,
                UnaryPredicate p, const T &new_value);
```



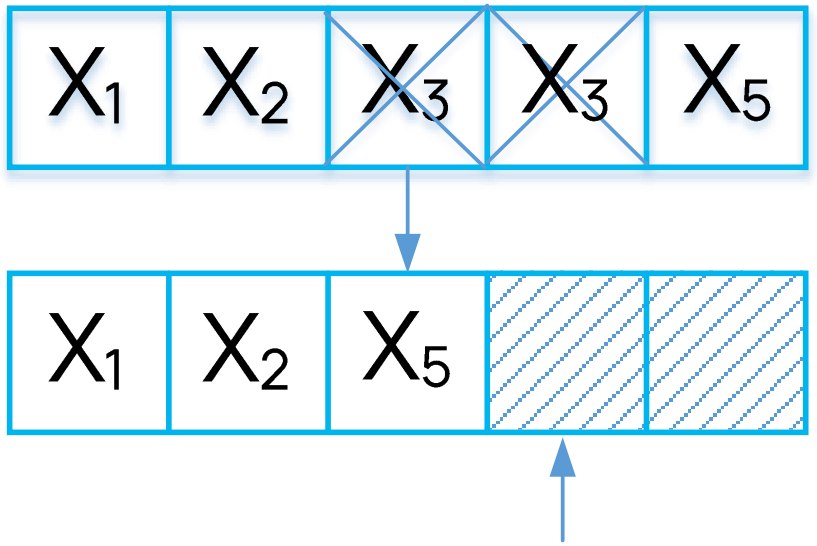
std::replace_copy

```
template <class InputIt, class OutputIt, class T>
OutputIt replace_copy(InputIt first, InputIt last, OutputIt d_first,
                      const T &old_value, const T &new_value);

template <class InputIt, class OutputIt, class UnaryPredicate, class T>
OutputIt replace_copy_if(InputIt first, InputIt last, OutputIt d_first,
                         UnaryPredicate p, const T &new_value);
```



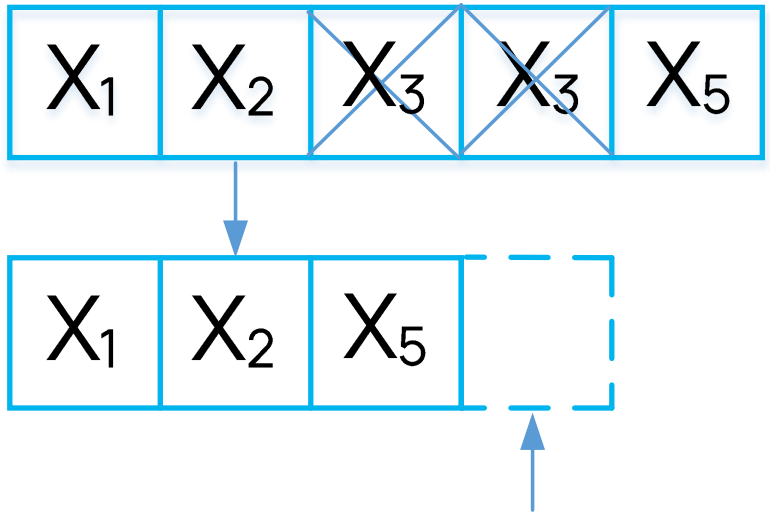
std::unique

```
template <class ForwardIt>
ForwardIt unique(ForwardIt first, ForwardIt last);

template <class ForwardIt, class BinaryPredicate>
ForwardIt unique(ForwardIt first, ForwardIt last, BinaryPredicate p);
```



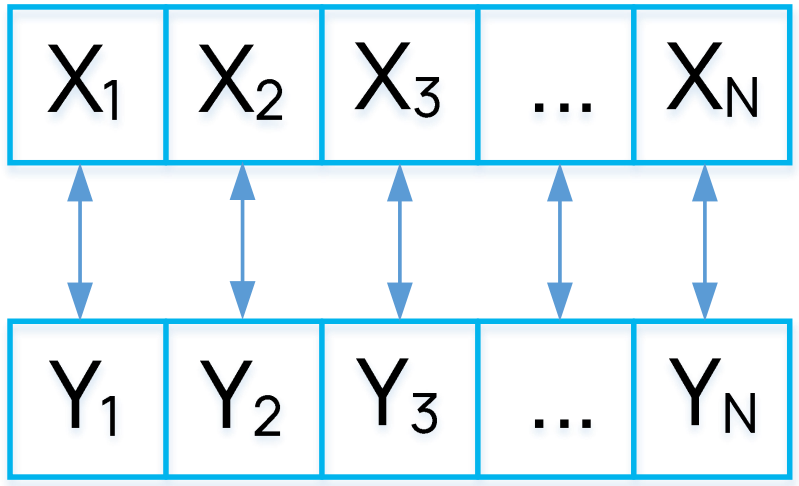
std::unique_copy

```
template <class InputIt, class OutputIt>
OutputIt unique_copy(InputIt first, InputIt last,
                     OutputIt d_first);

template <class InputIt, class OutputIt, class BinaryPredicate>
OutputIt unique_copy(InputIt first, InputIt last,
                     OutputIt d_first, BinaryPredicate p);
```



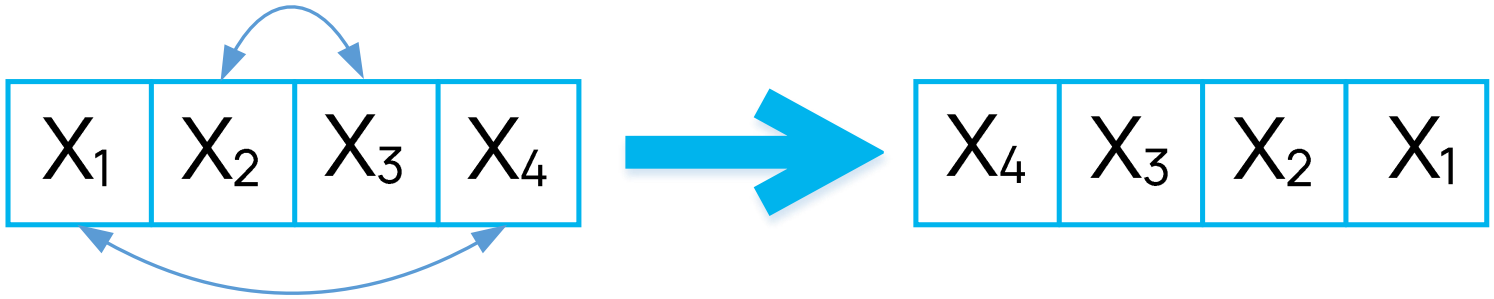
std::swap_ranges

```
template <class ForwardIt1, class ForwardIt2>
ForwardIt2 swap_ranges(ForwardIt1 first1, ForwardIt1 last1,
                       ForwardIt2 first2);
```



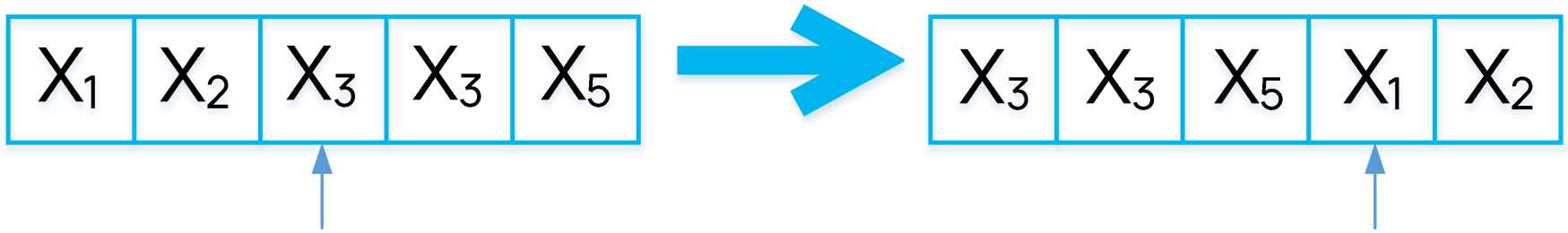
std::reverse

```
template <class BidirIt>
void reverse(BidirIt first, BidirIt last);

template <class BidirIt, class OutputIt>
OutputIt reverse_copy(BidirIt first, BidirIt last, OutputIt d_first);
```



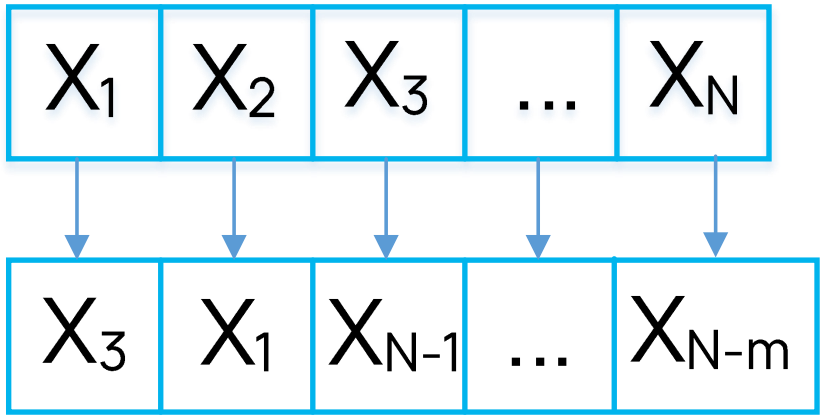
std::rotate

```
template <class ForwardIt>
ForwardIt rotate(ForwardIt first, ForwardIt n_first, ForwardIt last);

template <class ForwardIt, class OutputIt>
OutputIt rotate_copy(ForwardIt first, ForwardIt n_first,
                     ForwardIt last, OutputIt d_first);
```



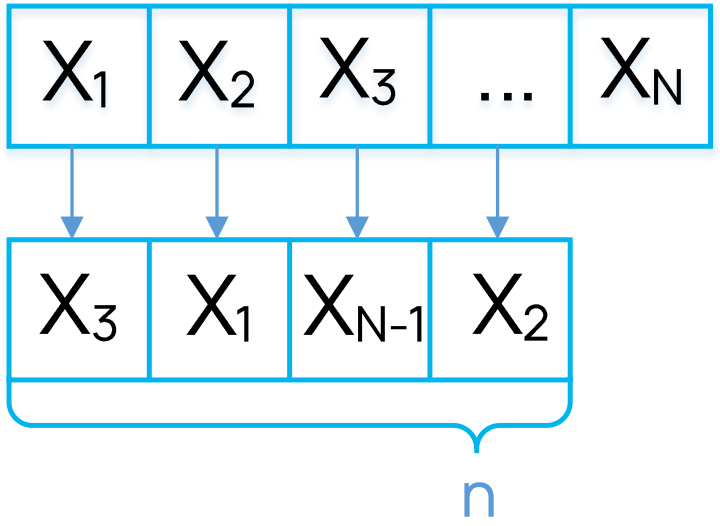
std::shuffle

```
template <class RandomIt, class URBG>
void shuffle(RandomIt first, RandomIt last, URBG&& g);
```



std::sample

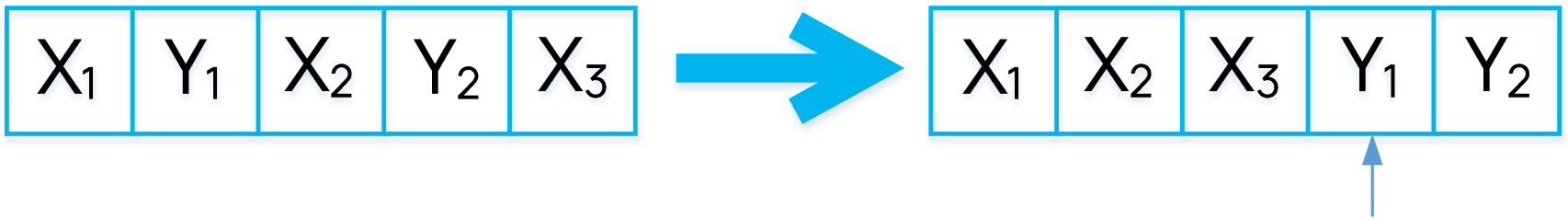
```
template <class PopulationIterator, class SampleIterator,
          class Distance, class URBG>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
                      SampleIterator out, Distance n,
                      URBG&& g);
```



Partitioning algorithms

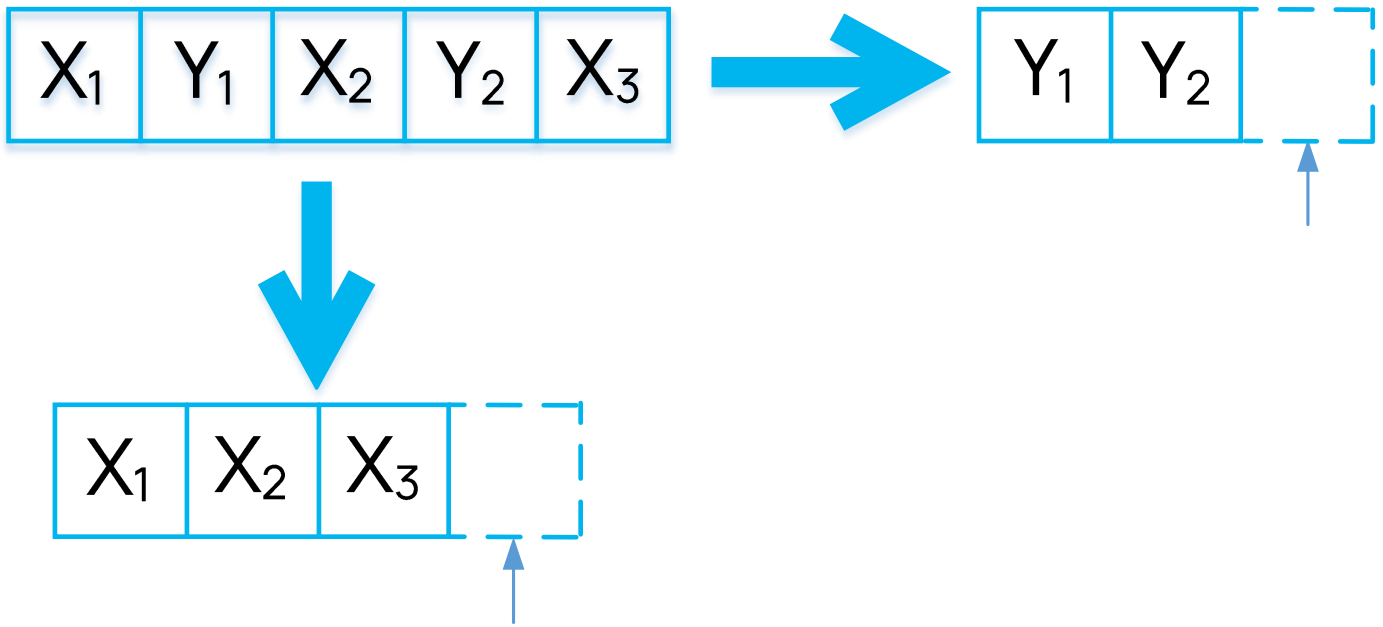
std::partition

```
template <class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p);

template <class BidirIt, class UnaryPredicate>
BidirIt stable_partition(BidirIt first, BidirIt last, UnaryPredicate p);

template <class InputIt, class UnaryPredicate>
bool is_partitioned(InputIt first, InputIt last, UnaryPredicate p);

template <class ForwardIt, class UnaryPredicate>
ForwardIt partition_point(ForwardIt first, ForwardIt last, UnaryPredicate p);
```



std::partition_copy

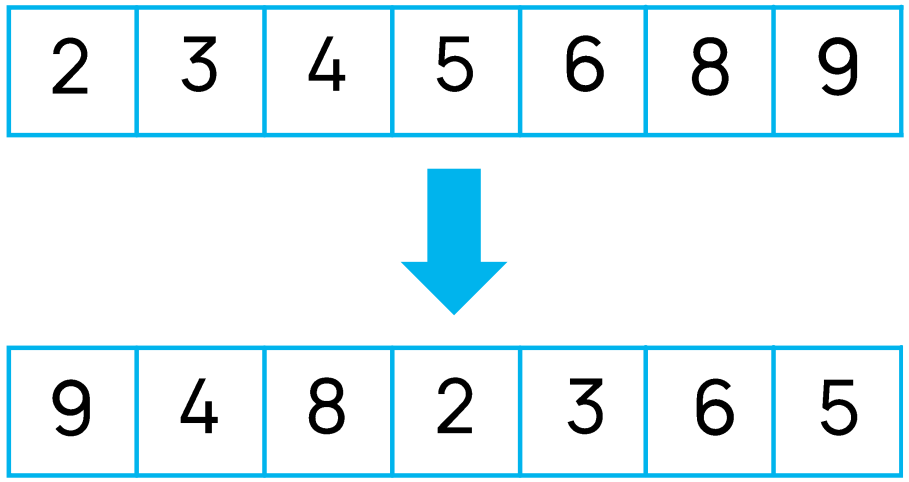
```
template <class InputIt, class OutputIt1,
          class OutputIt2, class UnaryPredicate>
std::pair<OutputIt1, OutputIt2> partition_copy(InputIt first, InputIt last,
                                               OutputIt1 d_first_true,
                                               OutputIt2 d_first_false,
                                               UnaryPredicate p);
```

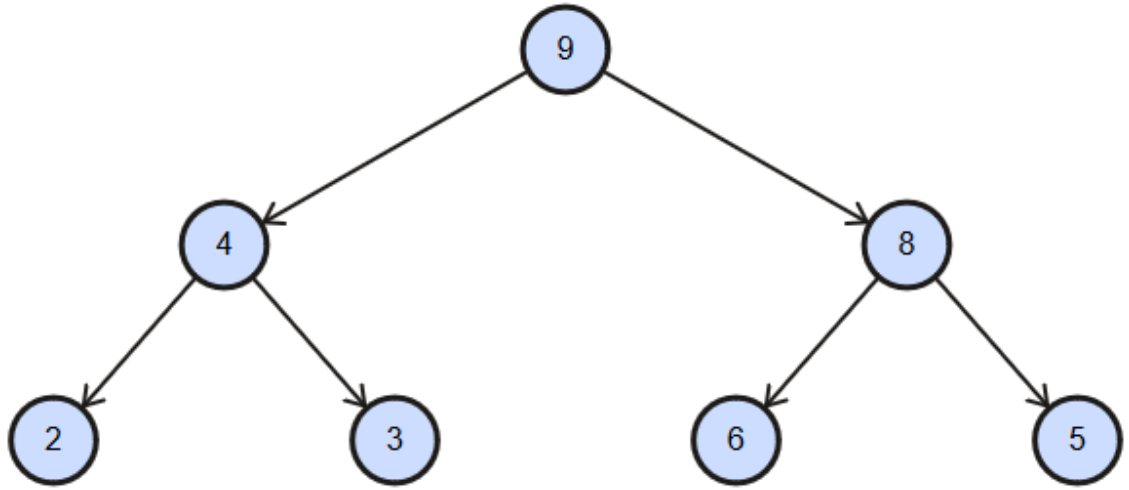


Heap algorithms

std::make_heap

```
template <class RandomIt>
void make_heap(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void make_heap(RandomIt first, RandomIt last, Compare comp);
```



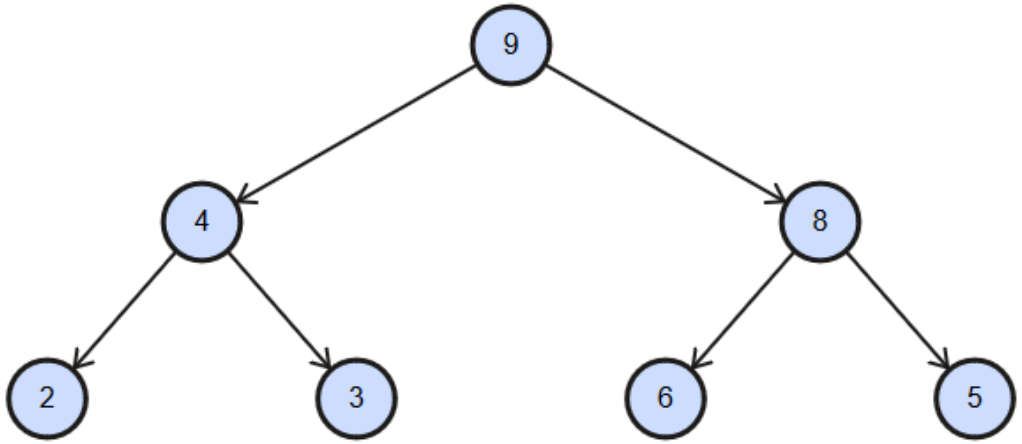


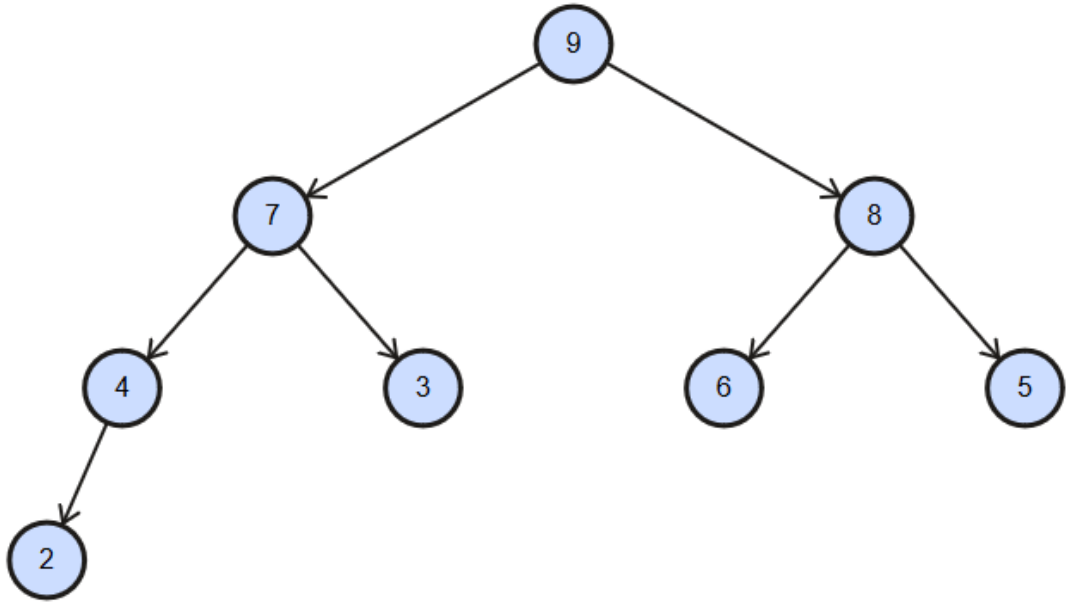
std::is_heap

```
template <class RandomIt>
bool is_heap(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
bool is_heap(RandomIt first, RandomIt last, Compare comp);

template <class RandomIt>
RandomIt is_heap_until(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
RandomIt is_heap_until(RandomIt first, RandomIt last, Compare comp);
```

std::push_heap

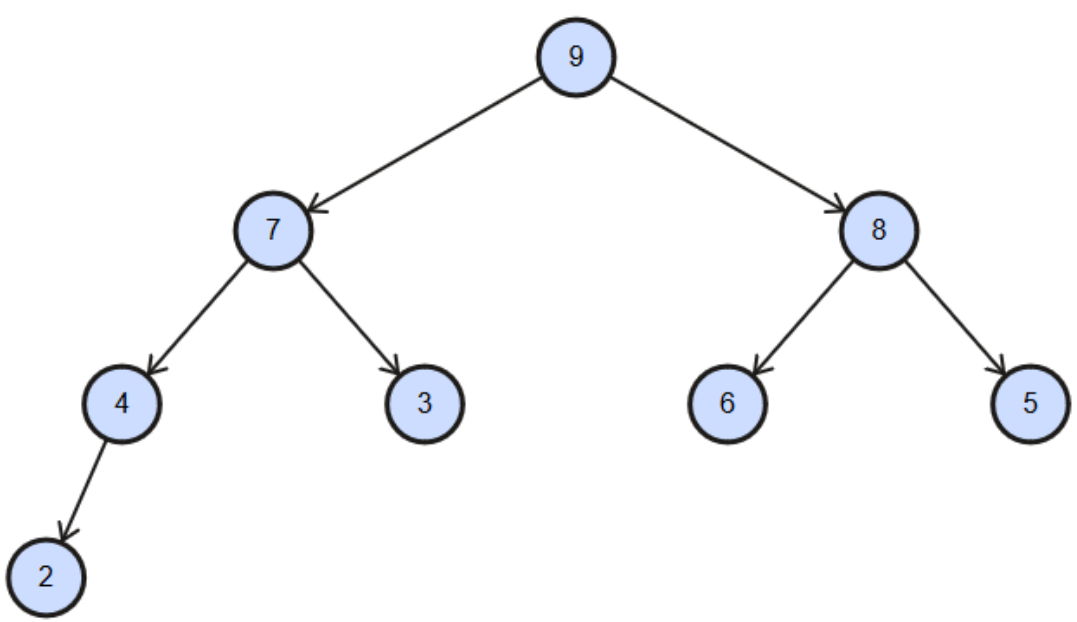
```
template <class RandomIt>
void push_heap(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void push_heap(RandomIt first, RandomIt last, Compare comp);
```

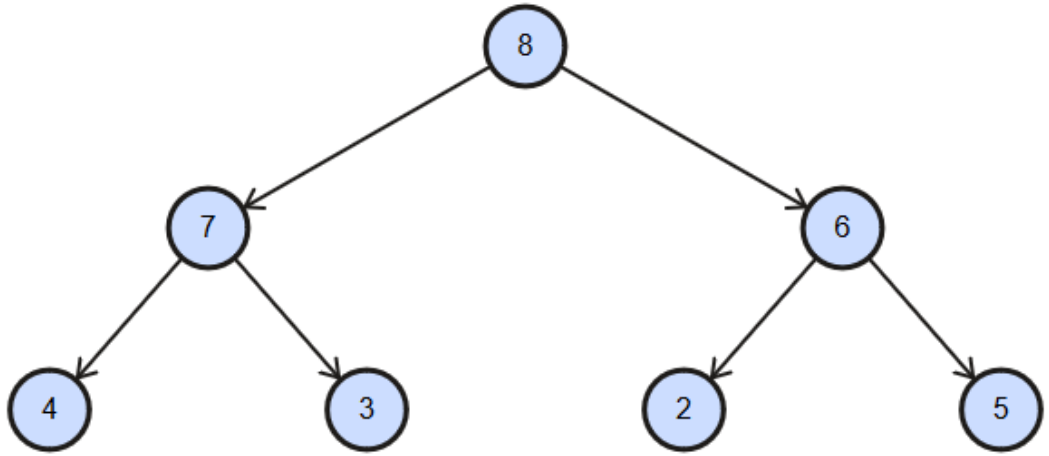




std::pop_heap

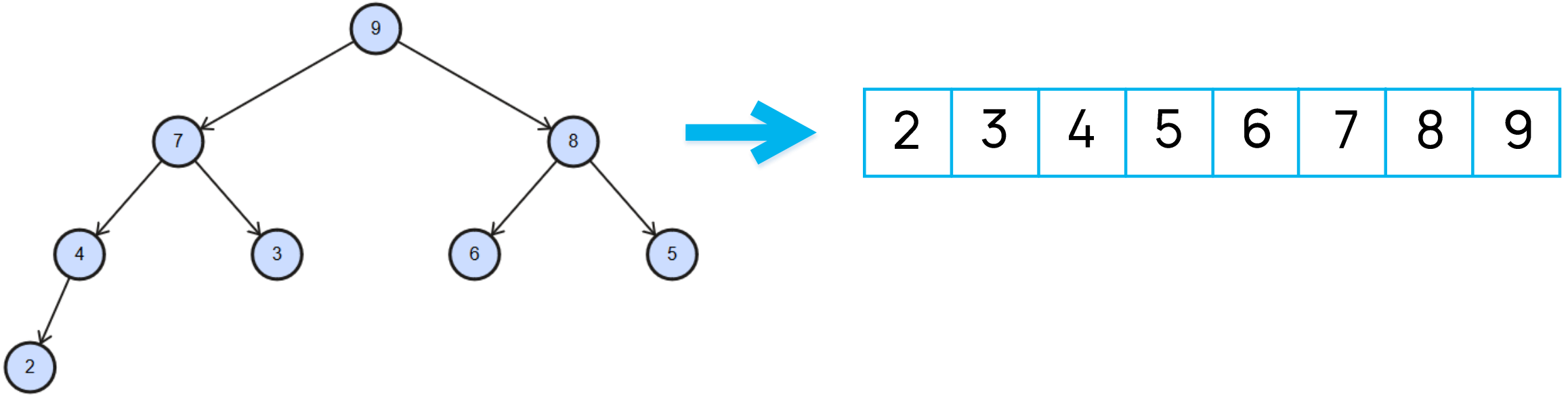
```
template <class RandomIt>
void pop_heap(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void pop_heap(RandomIt first, RandomIt last, Compare comp);
```





std::sort_heap

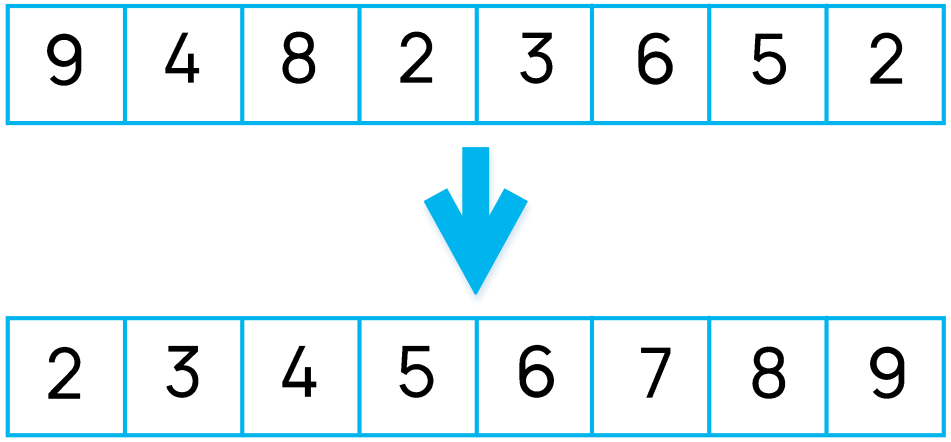
```
template <class RandomIt>
void sort_heap(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void sort_heap(RandomIt first, RandomIt last, Compare comp);
```



Sorting algorithms

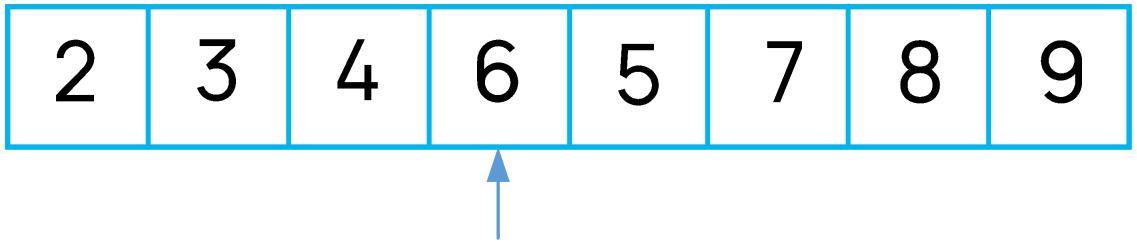
std::sort

```
template <class RandomIt>
void sort(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);

template <class RandomIt>
void stable_sort(RandomIt first, RandomIt last);

template <class RandomIt, class Compare>
void stable_sort(RandomIt first, RandomIt last, Compare comp);
```



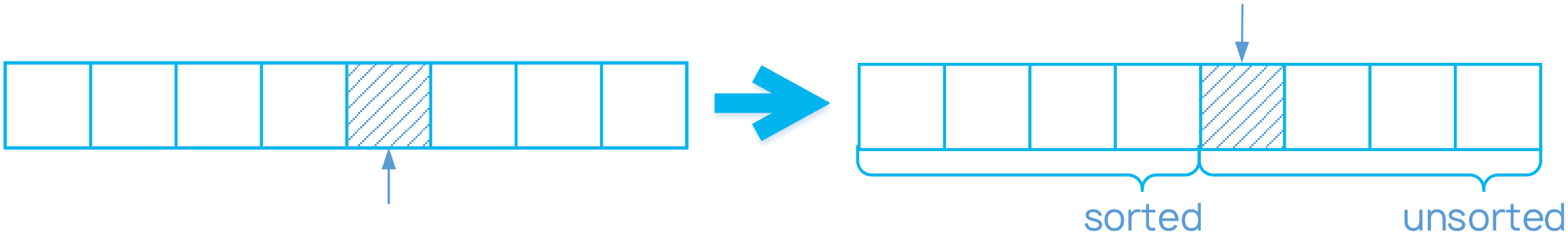
std::is_sorted

```
template <class ForwardIt>
bool is_sorted(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
bool is_sorted(ForwardIt first, ForwardIt last, Compare comp);

template <class ForwardIt>
ForwardIt is_sorted_until(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
ForwardIt is_sorted_until(ForwardIt first, ForwardIt last, Compare comp);
```



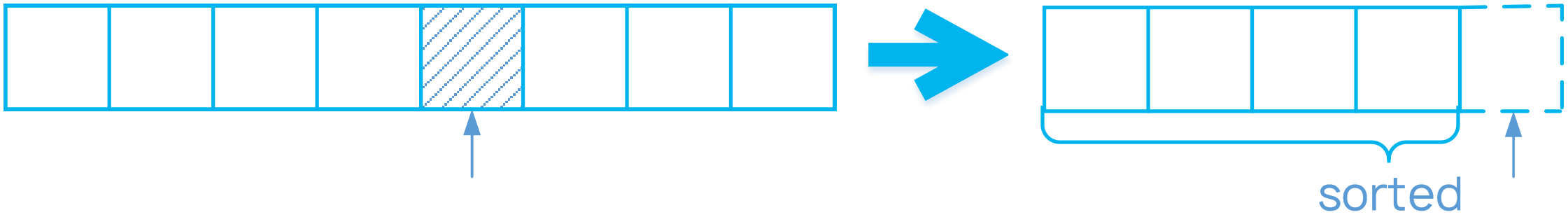
std::partial_sort

```
template <class RandomIt>
void partial_sort(RandomIt first, RandomIt middle, RandomIt last);

template <class RandomIt, class Compare>
void partial_sort(RandomIt first, RandomIt middle, RandomIt last,
                  Compare comp);
```



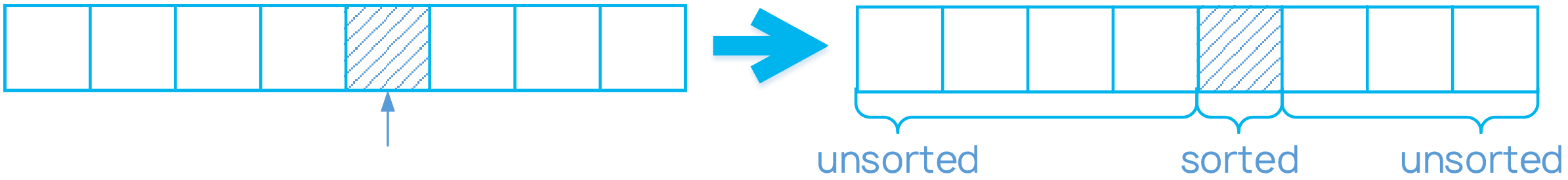
std::partial_sort_copy

```
template <class InputIt, class RandomIt>
RandomIt partial_sort_copy(InputIt first, InputIt last,
                           RandomIt d_first, RandomIt d_last);

template <class InputIt, class RandomIt, class Compare>
RandomIt partial_sort_copy(InputIt first, InputIt last,
                           RandomIt d_first, RandomIt d_last,
                           Compare comp);
```



std::nth_element

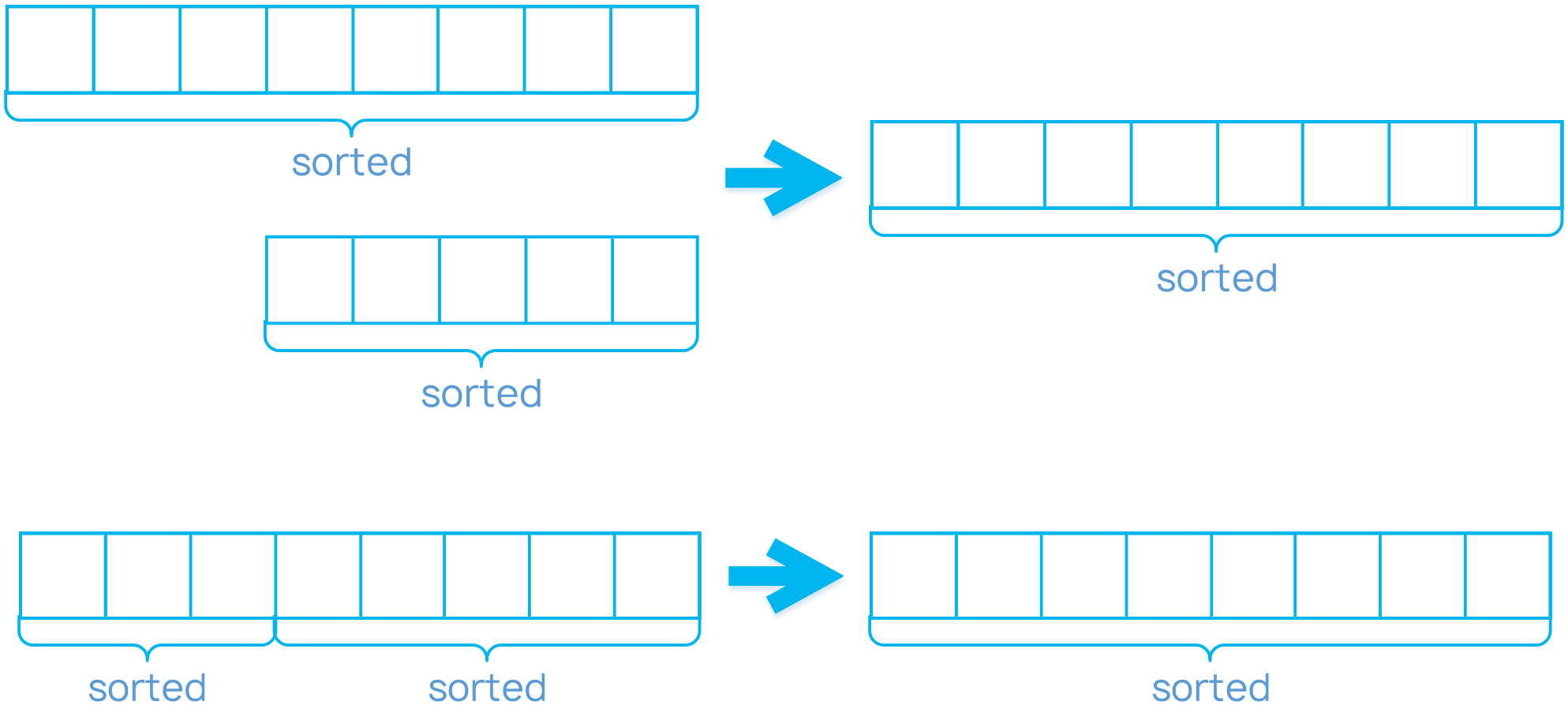
```
template <class RandomIt>
void nth_element(RandomIt first, RandomIt nth, RandomIt last);

template <class RandomIt, class Compare>
void nth_element(RandomIt first, RandomIt nth, RandomIt last,
                 Compare comp);
```



std::merge

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first);

template <class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first, Compare comp);

template <class BidirIt>
void inplace_merge(BidirIt first, BidirIt middle, BidirIt last);

template <class BidirIt, class Compare>
void inplace_merge(BidirIt first, BidirIt middle, BidirIt last, Compare comp);
```

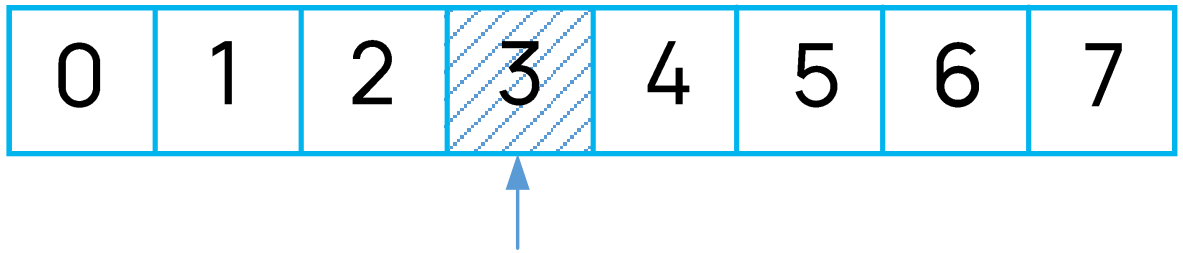
std::merge



Binary search algorithms

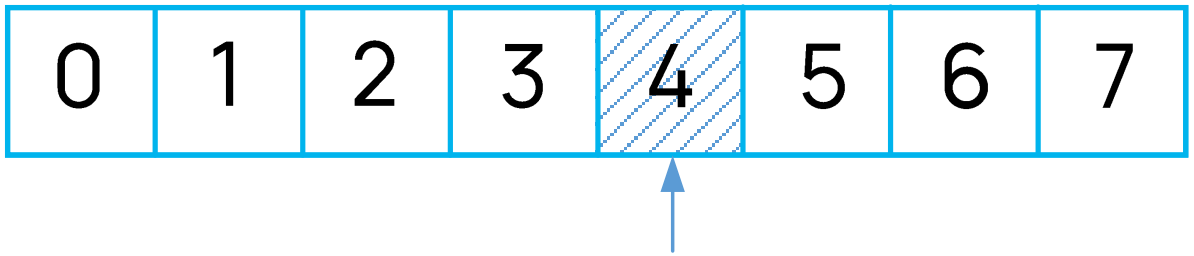
std::lower_bound

```
template <class ForwardIt, class T>
ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T &value);

template <class ForwardIt, class T, class Compare>
ForwardIt lower_bound(ForwardIt first, ForwardIt last,
                      const T &value, Compare comp);
```



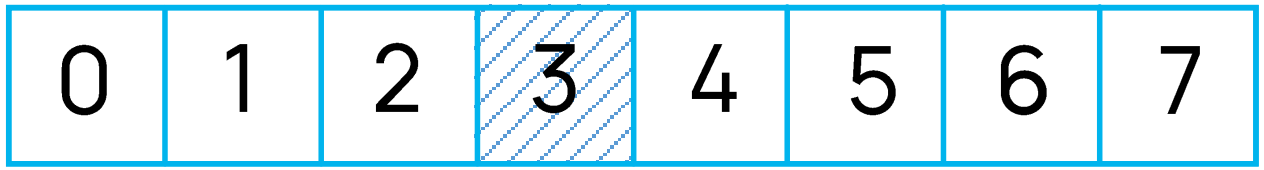
std::upper_bound

```
template <class ForwardIt, class T>
ForwardIt upper_bound(ForwardIt first, ForwardIt last, const T &value);

template <class ForwardIt, class T, class Compare>
ForwardIt upper_bound(ForwardIt first, ForwardIt last,
                      const T &value, Compare comp);
```



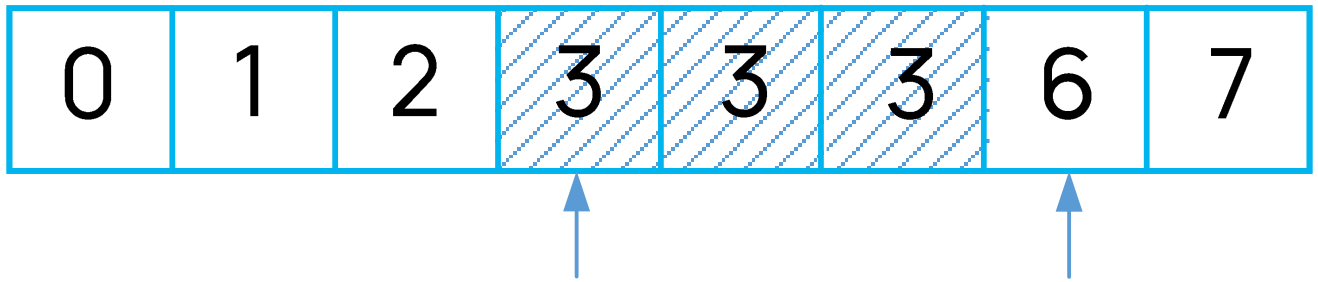
std::binary_search

```
template< class ForwardIt, class T >
bool binary_search(ForwardIt first, ForwardIt last, const T &value);

template <class ForwardIt, class T, class Compare>
bool binary_search(ForwardIt first, ForwardIt last,
                   const T &value, Compare comp);
```



std::equal_range

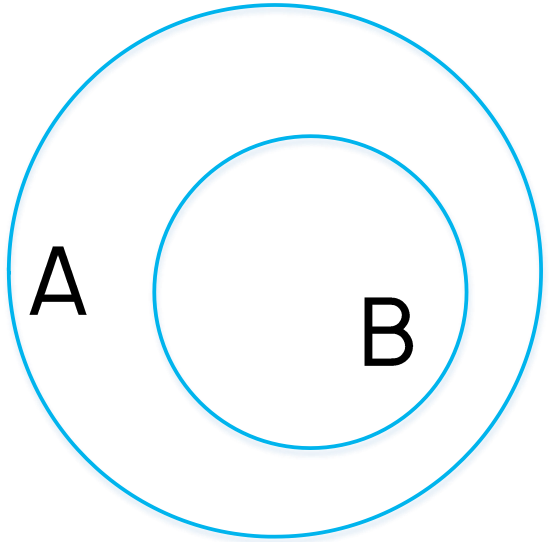
```
template< class ForwardIt, class T >
std::pair<ForwardIt,ForwardIt> equal_range(ForwardIt first, ForwardIt last,
                                           const T &value);

template <class ForwardIt, class T, class Compare>
std::pair<ForwardIt,ForwardIt> equal_range(ForwardIt first, ForwardIt last,
                                           const T &value, Compare comp);
```



'Set' algorithms

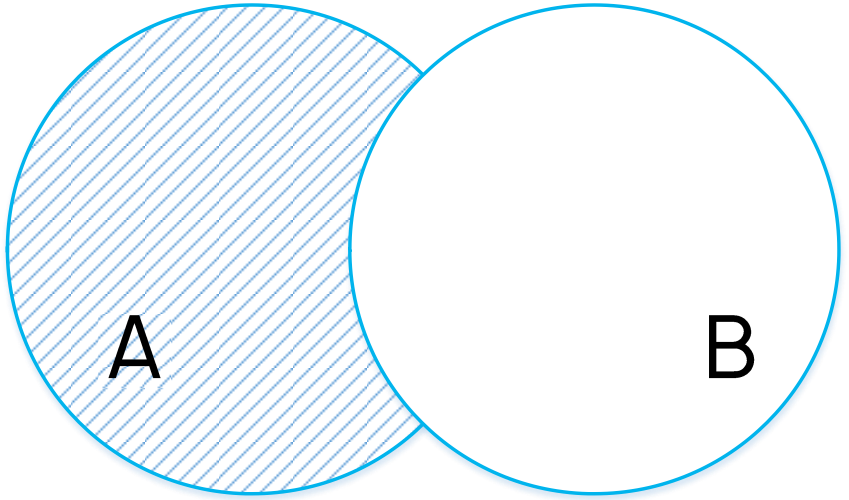
std::includes

```
template <class InputIt1, class InputIt2>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2);

template <class InputIt1, class InputIt2, class Compare>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2, Compare comp);
```



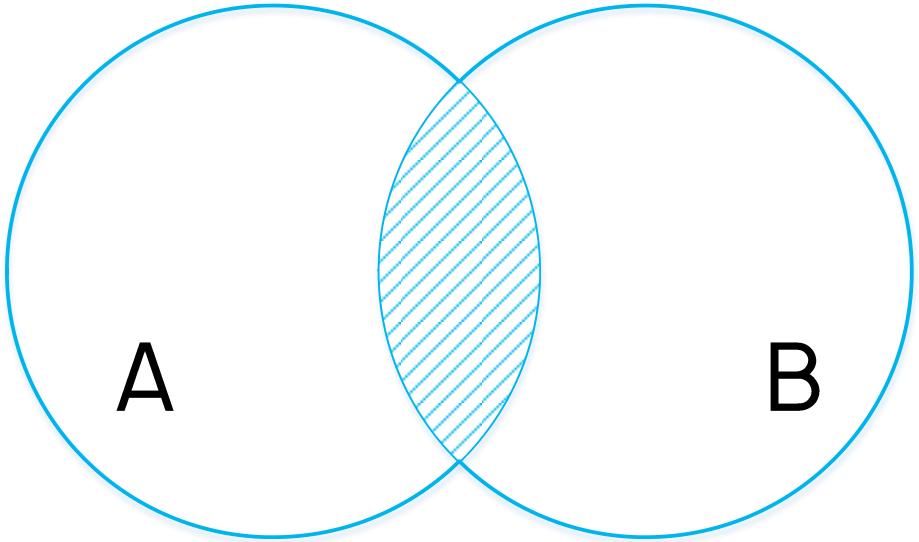
std::set_difference

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt set_difference(InputIt1 first1, InputIt1 last1,
                        InputIt2 first2, InputIt2 last2,
                        OutputIt d_first);

template <class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_difference(InputIt1 first1, InputIt1 last1,
                        InputIt2 first2, InputIt2 last2,
                        OutputIt d_first, Compare comp);
```



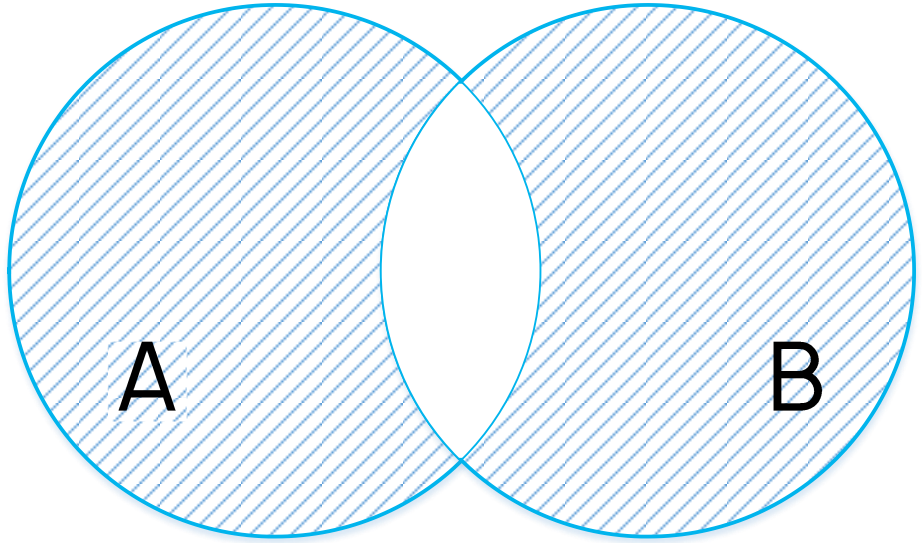
std::set_intersection

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first);

template <class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first, Compare comp);
```



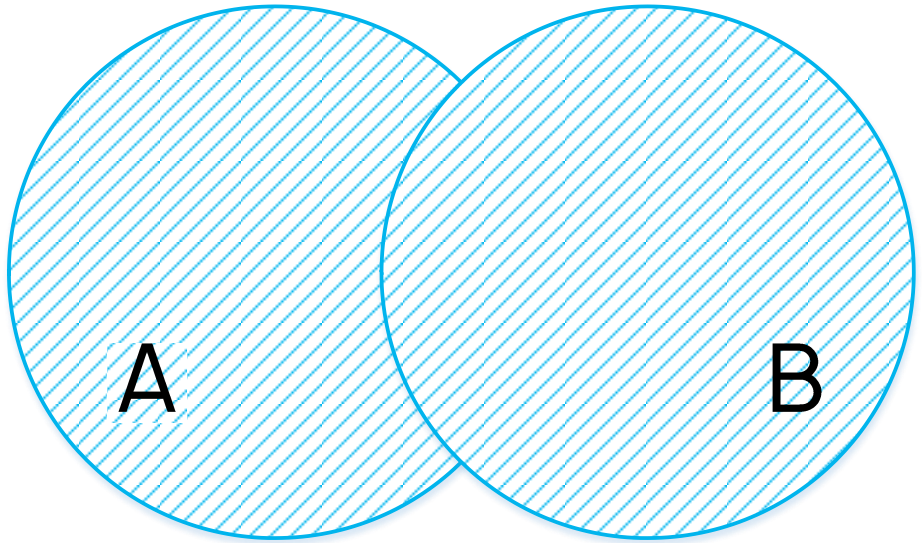
std::set_symmetric_difference

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                  InputIt2 first2, InputIt2 last2,
                                  OutputIt d_first);

template <class InputIt1, class InputIt2,
          class OutputIt, class Compare>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                  InputIt2 first2, InputIt2 last2,
                                  OutputIt d_first, Compare comp);
```



std::set_union

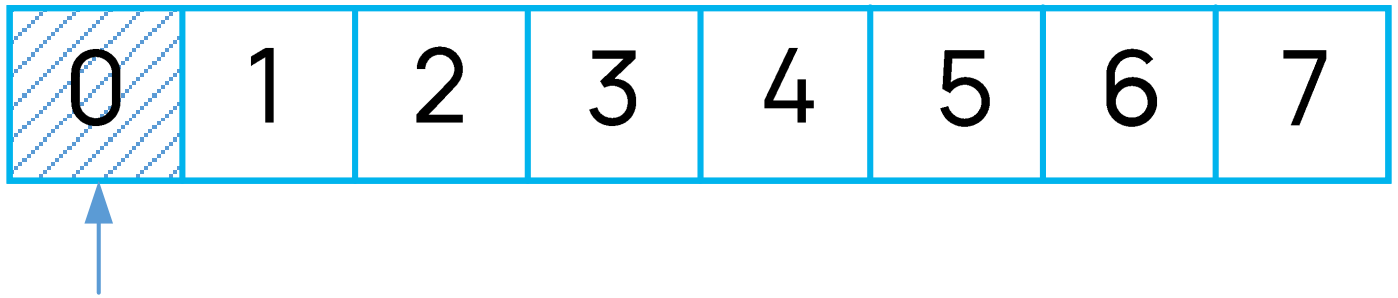
```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, InputIt2 last2,
                   OutputIt d_first);

template <class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, InputIt2 last2,
                   OutputIt d_first, Compare comp);
```



minmax algorithms

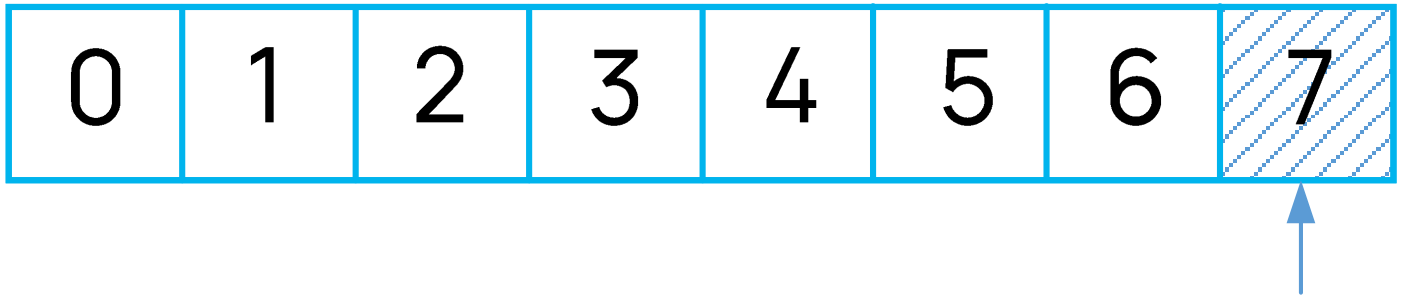
std::min_element

```
template <class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
ForwardIt min_element(ForwardIt first, ForwardIt last, Compare comp);
```



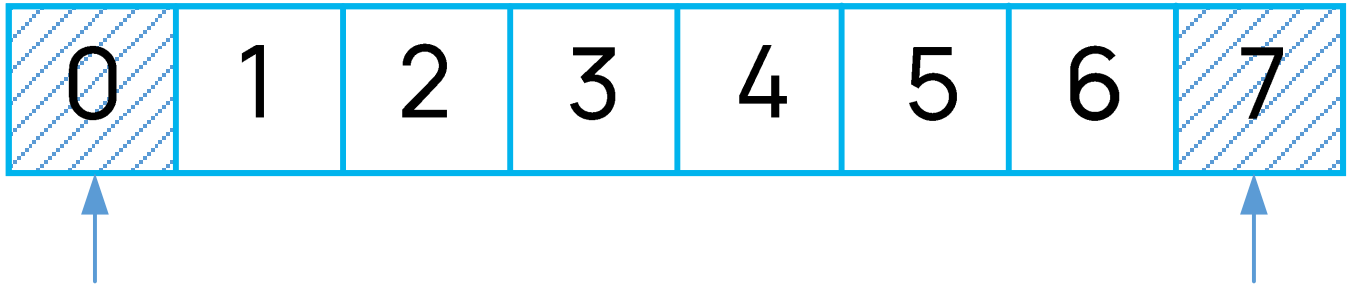
std::max_element

```
template <class ForwardIt>
ForwardIt max_element(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
ForwardIt max_element(ForwardIt first, ForwardIt last, Compare comp);
```



std::minmax_element

```
template <class ForwardIt>
std::pair<ForwardIt, ForwardIt> minmax_element(ForwardIt first,
                                               ForwardIt last);

template <class ForwardIt, class Compare>
std::pair<ForwardIt, ForwardIt> minmax_element(ForwardIt first,
                                               ForwardIt last,
                                               Compare comp);
```



Comparison algorithms

std::equal

```
template <class InputIt1, class InputIt2>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2);

template <class InputIt1, class InputIt2, class BinaryPredicate>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2, BinaryPredicate p);

template <class InputIt1, class InputIt2>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2, InputIt2 last2);

template <class InputIt1, class InputIt2, class BinaryPredicate>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2, InputIt2 last2,
           BinaryPredicate p);
```

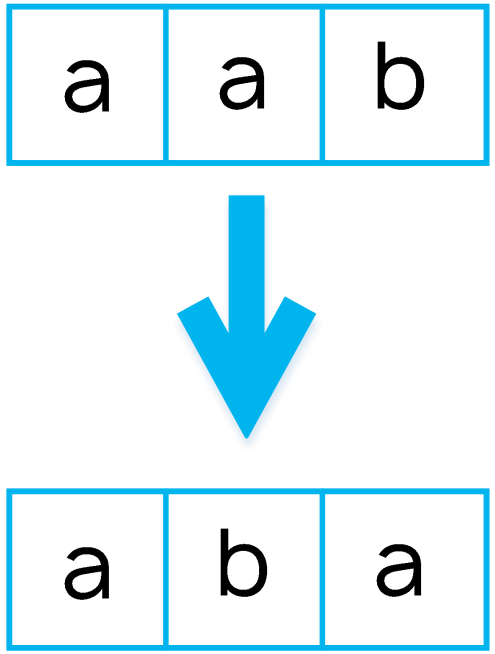
std::lexicographical_compare

```
template <class InputIt1, class InputIt2>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2);

template <class InputIt1, class InputIt2, class Compare>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2,
                             Compare comp);
```

Permutation algorithms

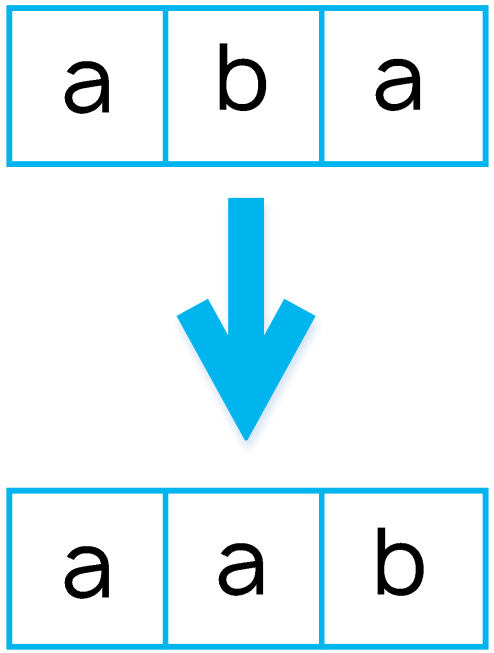
std::next_permutation

```
template <class BidirIt>
bool next_permutation(BidirIt first, BidirIt last);

template <class BidirIt, class Compare>
bool next_permutation(BidirIt first, BidirIt last, Compare comp);
```



std::prev_permutation

```
template <class BidirIt>
bool prev_permutation(BidirIt first, BidirIt last);

template <class BidirIt, class Compare>
bool prev_permutation(BidirIt first, BidirIt last, Compare comp);
```



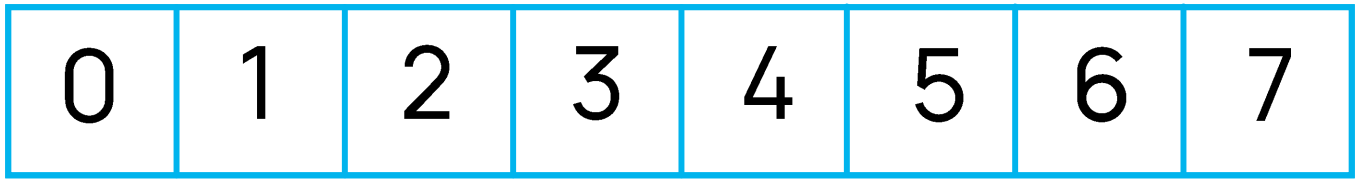
std::is_permutation

```
template <class ForwardIt1, class ForwardIt2>
bool is_permutation(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2);

template <class ForwardIt1, class ForwardIt2, class BinaryPredicate>
bool is_permutation(ForwardIt1 first1, ForwardIt1 last1,
                    ForwardIt2 first2, BinaryPredicate p);

template <class ForwardIt1, class ForwardIt2>
bool is_permutation(ForwardIt1 first1, ForwardIt1 last1,
                    ForwardIt2 first2, ForwardIt2 last2);

template <class ForwardIt1, class ForwardIt2, class BinaryPredicate>
bool is_permutation(ForwardIt1 first1, ForwardIt1 last1,
                    ForwardIt2 first2, ForwardIt2 last2,
                    BinaryPredicate p);
```

Numeric algorithms

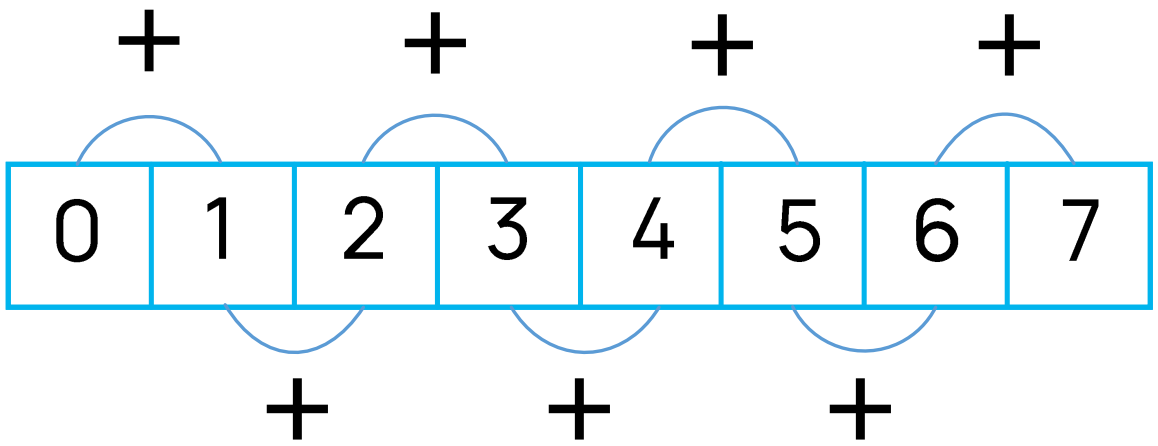
std::iota

```
template <class ForwardIt, class T>
void iota(ForwardIt first, ForwardIt last, T value);
```



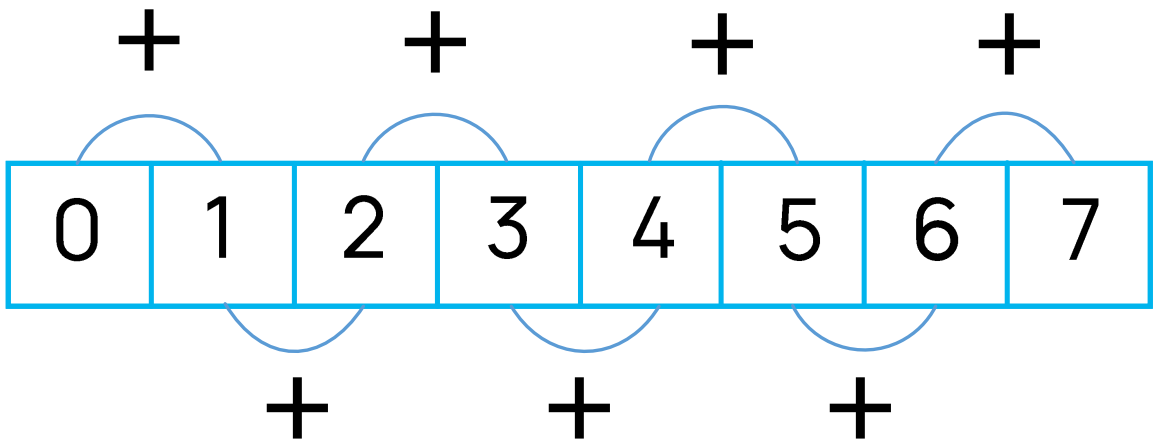
std::accumulate

```
template <class InputIt, class T>
T accumulate(InputIt first, InputIt last, T init);

template <class InputIt, class T, class BinaryOperation>
T accumulate(InputIt first, InputIt last, T init, BinaryOperation op);
```



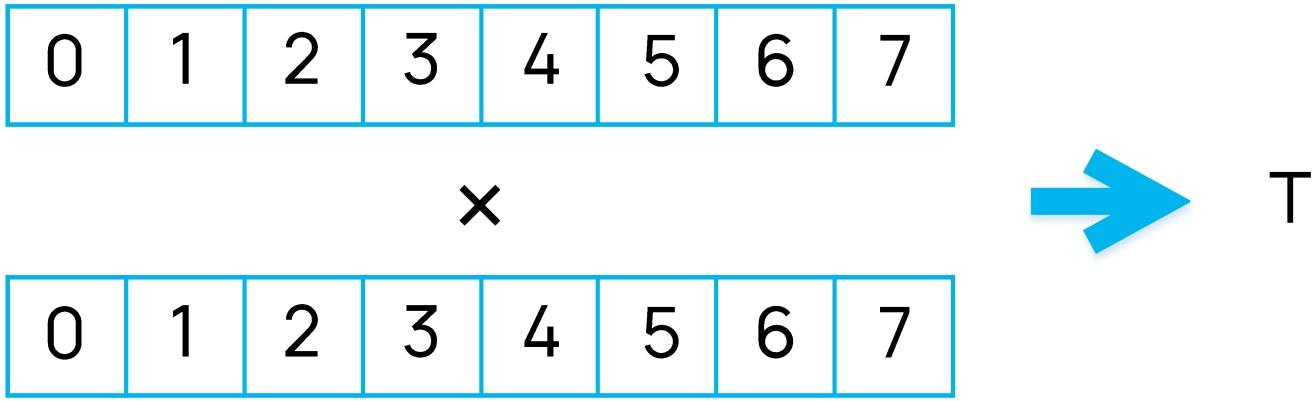
std::reduce

```
template <class InputIt>
typename std::iterator_traits<InputIt>::value_type reduce(InputIt first,
                                                          InputIt last);

template <class InputIt, class T>
T reduce(InputIt first, InputIt last, T init);

template <class InputIt, class T, class BinaryOp>
T reduce(InputIt first, InputIt last, T init, BinaryOp op);
```



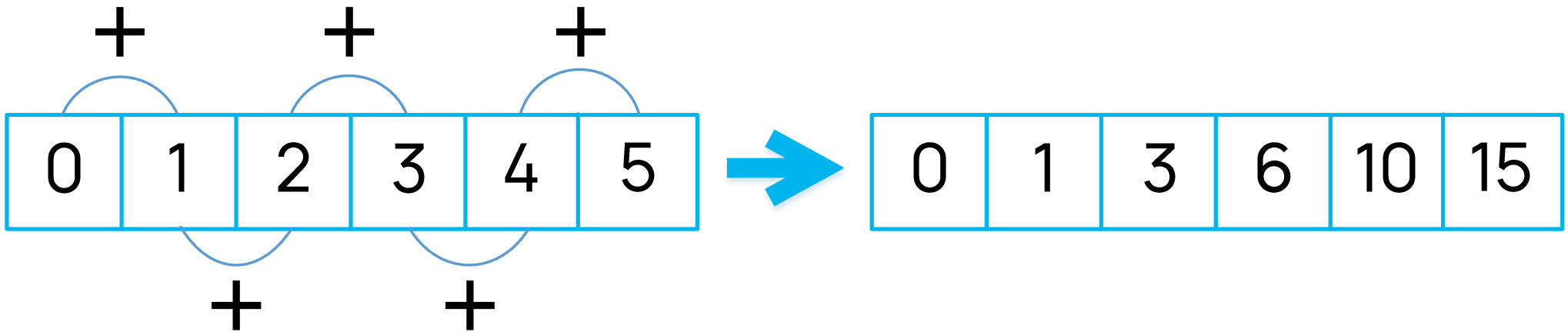
std::inner_product

```
template <class InputIt1, class InputIt2, class T>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T init);

template <class InputIt1, class InputIt2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T init,
                BinaryOperation1 op1,
                BinaryOperation2 op2);
```



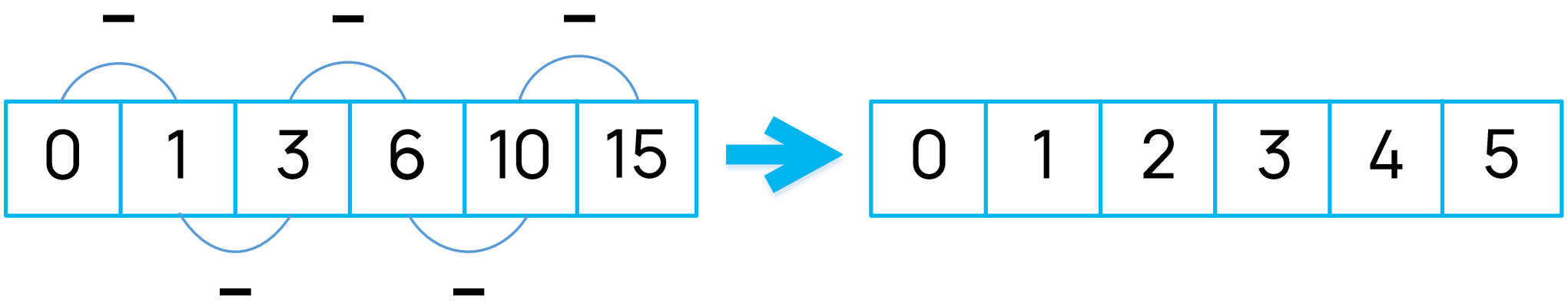
std::partial_sum

```
template <class InputIt, class OutputIt>
OutputIt partial_sum(InputIt first, InputIt last, OutputIt d_first);

template <class InputIt, class OutputIt, class BinaryOperation>
OutputIt partial_sum(InputIt first, InputIt last, OutputIt d_first,
                     BinaryOperation op);
```



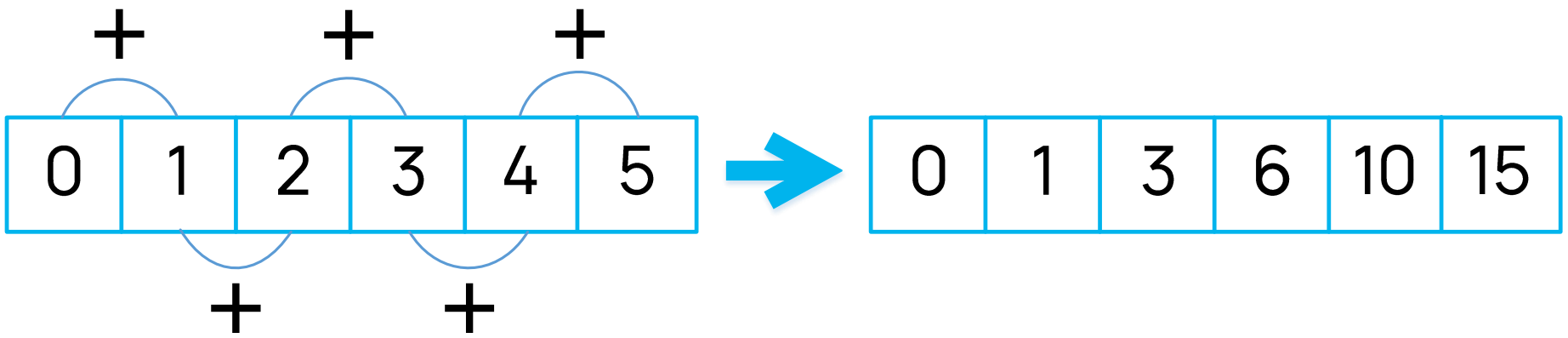
std::adjacent_difference

```
template <class InputIt, class OutputIt>
OutputIt adjacent_difference(InputIt first, InputIt last, OutputIt d_first);

template <class InputIt, class OutputIt, class BinaryOperation>
OutputIt adjacent_difference(InputIt first, InputIt last,
                             OutputIt d_first, BinaryOperation op );
```



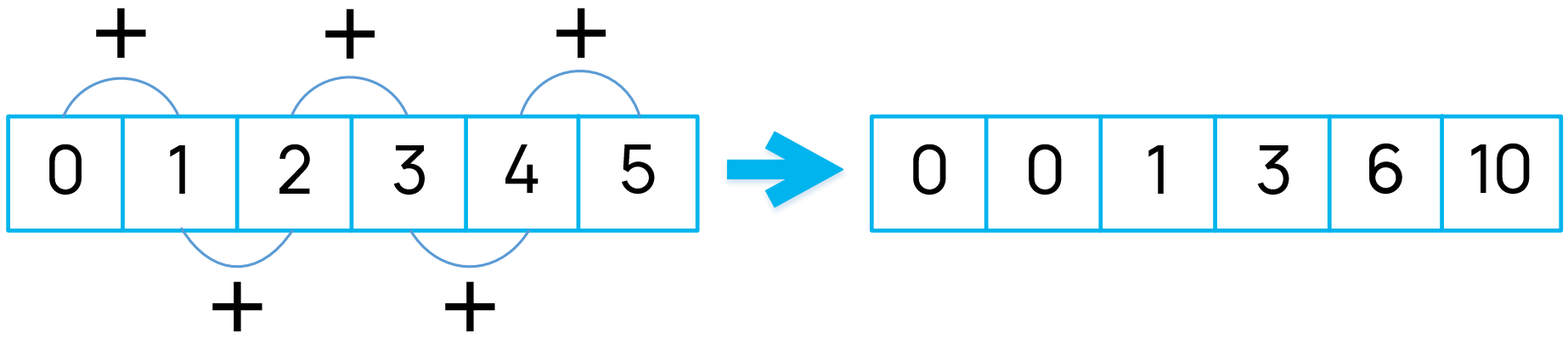
std::inclusive_scan

```
template <class InputIt, class OutputIt>
OutputIt inclusive_scan(InputIt first, InputIt last, OutputIt d_first);

template <class InputIt, class OutputIt, class BinaryOperation>
OutputIt inclusive_scan(InputIt first, InputIt last,
                        OutputIt d_first, BinaryOperation op);

template <class InputIt, class OutputIt, class BinaryOperation, class T>
OutputIt inclusive_scan(InputIt first, InputIt last, OutputIt d_first,
                        BinaryOperation op, T init);
```



std::exclusive_scan

```
template <class InputIt, class OutputIt, class T>
OutputIt exclusive_scan(InputIt first, InputIt last,
                        OutputIt d_first, T init);

template <class InputIt, class OutputIt, class T, class BinaryOperation>
OutputIt exclusive_scan(InputIt first, InputIt last,
                        OutputIt d_first, T init, BinaryOperation op);
```

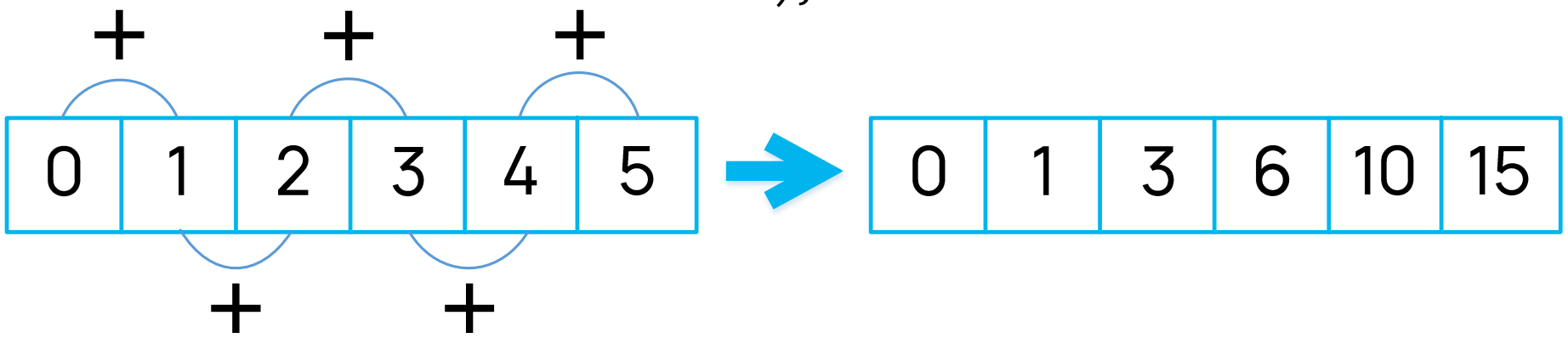


std::transform_reduce

```
template <class InputIt1, class InputIt2, class T>
T transform_reduce(InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, T init);

template <class InputIt1, class InputIt2, class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   T init, BinaryOp1 op1, BinaryOp2 op2);

template <class InputIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(InputIt first, InputIt last, T init,
                   BinaryOp binop, UnaryOp unary_op);
```

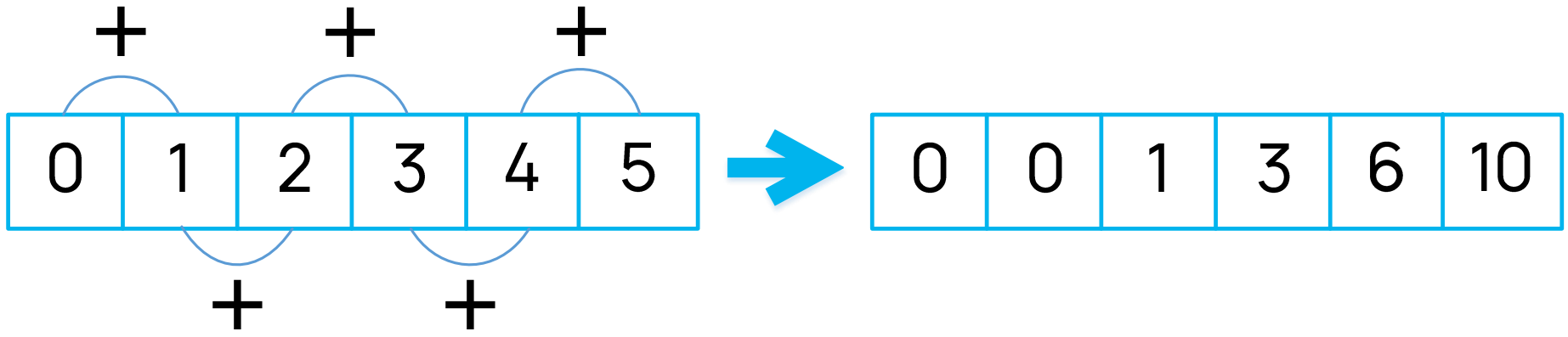
std::transform_inclusive_scan

```
template <class InputIt, class OutputIt,
          class BinaryOperation, class UnaryOperation>
OutputIt transform_inclusive_scan(InputIt first, InputIt last,
                                  OutputIt d_first,
                                  BinaryOperation op1, UnaryOperation op2);

template <class InputIt, class OutputIt,
          class BinaryOperation, class UnaryOperation, class T>
OutputIt transform_inclusive_scan(InputIt first, InputIt last,
                                  OutputIt d_first,
                                  BinaryOperation op1, UnaryOperation op2,
                                  T init);
```



std::transform_exclusive_scan

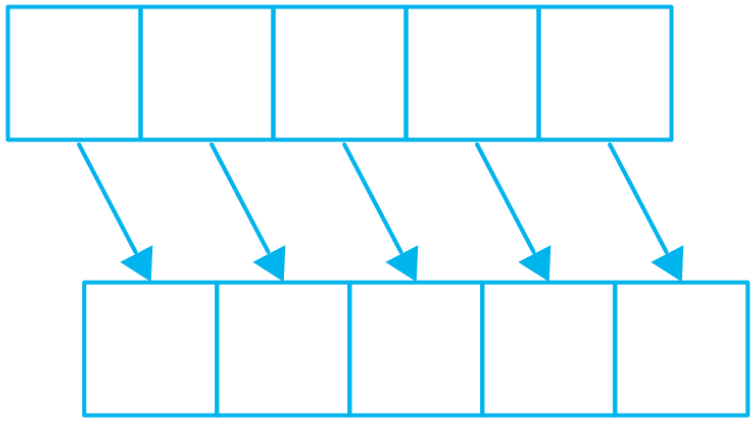
```
template <class InputIt, class OutputIt, class T,
          class BinaryOperation, class UnaryOperation>
OutputIt transform_exclusive_scan(InputIt first, InputIt last,
                                  OutputIt d_first, T init,
                                  BinaryOperation op1, UnaryOperation op2);
```



Algorithms with uninitialized memory

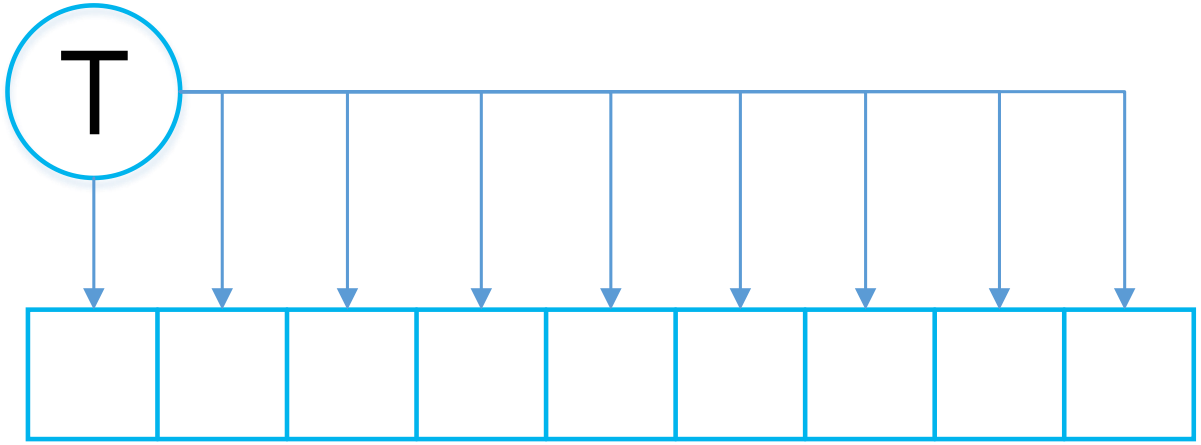
std::uninitialized_copy

```
template <class InputIt, class ForwardIt>
ForwardIt uninitialized_copy(InputIt first, InputIt last,
                             ForwardIt d_first);

template <class InputIt, class Size, class ForwardIt>
ForwardIt uninitialized_copy_n(InputIt first, Size count,
                               ForwardIt d_first);
```



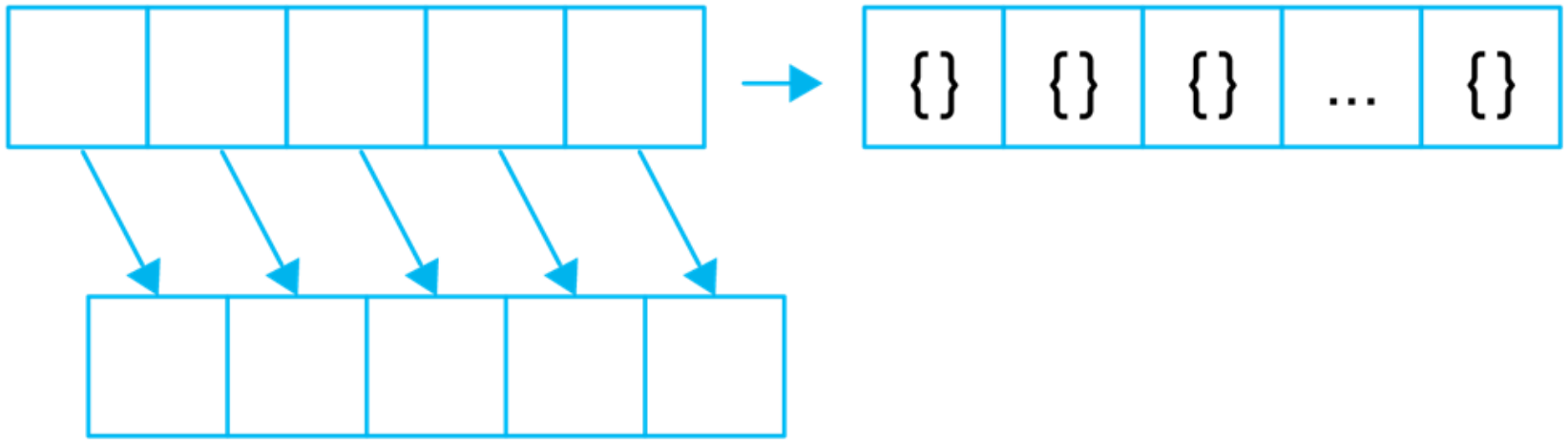
std::uninitialized_fill

```
template <class ForwardIt, class T>
void uninitialized_fill(ForwardIt first, ForwardIt last, const T &value);

template <class ForwardIt, class Size, class T>
ForwardIt uninitialized_fill_n(ForwardIt first, Size count, const T &value);
```



std::uninitialized_move

```
template <class InputIt, class ForwardIt>
ForwardIt uninitialized_move(InputIt first, InputIt last, ForwardIt d_first);

template <class InputIt, class Size, class ForwardIt>
std::pair<InputIt, ForwardIt> uninitialized_move_n(InputIt first,
                                                   Size count,
                                                   ForwardIt d_first);
```



std::uninitialized_default_construct

```
template <class ForwardIt>
void uninitialized_default_construct(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Size>
ForwardIt uninitialized_default_construct_n(ForwardIt first, Size n);
```

std::uninitialized_value_construct

```
template <class ForwardIt>
void uninitialized_value_construct(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Size>
ForwardIt uninitialized_value_construct_n(ForwardIt first, Size n);
```

std::destroy

```
template <class ForwardIt>
void destroy(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Size>
ForwardIt destroy_n(ForwardIt first, Size n);
```

END

Q&A