Working Draft, Technical Specification for C++ Extensions for Parallelism, Revision 1

Note: this is an early draft. It’s known to be incomplete and incorrect, and it has lots of bad formatting.
Contents

1 General ................................................ 3
   1.1 Scope .............................................. 3
   1.2 Normative references ............................. 3
   1.3 Namespaces and headers ......................... 3
   1.4 Terms and definitions ............................ 3

2 Execution policies ................................. 4
   2.1 In general ........................................... 4
   2.2 Header <experimental/execution_policy> synopsis 4
   2.3 Execution policy type trait ...................... 5
   2.4 Sequential execution policy ..................... 5
   2.5 Parallel execution policy ....................... 6
   2.6 Vector execution policy ........................ 6
   2.7 Dynamic execution policy ....................... 7
      2.7.1 execution_policy construct/assign/swap ........ 8
      2.7.2 execution_policy object access ............... 8
   2.8 Execution policy specialized algorithms ........ 8
   2.9 Execution policy objects ....................... 9

3 Parallel exceptions ............................... 9
   3.1 Exception reporting behavior .................. 9
   3.2 Header <experimental/exception> synopsis .... 9

4 Parallel algorithms ............................... 10
   4.1 In general ......................................... 10
      4.1.1 Effect of execution policies on parallel algorithm execution 10
      4.1.2 ExecutionPolicy algorithm overloads ........ 12
   4.2 Definitions ....................................... 13
   4.3 Novel algorithms ................................ 13
      4.3.1 Header <experimental/algorithm> synopsis .... 13
      4.3.2 For each ....................................... 14
      4.3.3 Header <experimental/numeric> synopsis .... 15
      4.3.4 Reduce ......................................... 16
      4.3.5 Exclusive scan ................................ 17
      4.3.6 Inclusive scan ................................. 18
1 General

1.1 Scope

This Technical Specification describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution. The algorithms described by this Technical Specification are realizable across a broad class of computer architectures.

This Technical Specification is non-normative. Some of the functionality described by this Technical Specification may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this Technical Specification may never be standardized, and other functionality may be standardized in a substantially changed form.

The goal of this Technical Specification is to build widespread existing practice for parallelism in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references

The following reference document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 14882:2011, Programming Languages – C++


Unless otherwise specified, the whole of the C++ Standard Library introduction [lib.library] is included into this Technical Specification by reference.

1.3 Namespaces and headers

Since the extensions described in this Technical Specification are experimental and not part of the C++ Standard Library, they should not be declared directly within namespace std. Unless otherwise specified, all components described in this Technical Specification are declared in namespace std::experimental::parallel.

[Note: Once standardized, the components described by this Technical Specification are expected to be promoted to namespace std. – end note]

Unless otherwise specified, references to other entities described in this Technical Specification are assumed to be qualified with std::experimental::parallel, and references to entities described in the C++ Standard Library are assumed to be qualified with std::.

Extensions that are expected to eventually be added to an existing header <meow> are provided inside the <experimental/meow> header, which shall include the standard contents of <meow> as if by

```cpp
#include <meow>
```

1.4 Terms and definitions

For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.
A parallel algorithm is a function template described by this Technical Specification declared in namespace std::experimental::parallel with a formal template parameter named ExecutionPolicy.

2 Execution policies

2.1 In general

This subclause describes classes that represent execution policies. An execution policy is an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a standard algorithm. Execution policies afford standard algorithms the discretion to execute in parallel.

[Example:

```cpp
std::vector<int> v = ...

// standard sequential sort
std::sort(vec.begin(), vec.end());

using namespace std::experimental::parallel;

// explicitly sequential sort
sort(seq, v.begin(), v.end());

// permitting parallel execution
sort(par, v.begin(), v.end());

// permitting vectorization as well
sort(vec, v.begin(), v.end());

// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if(v.size() > threshold)
{
    exec = par;
}

sort(exec, v.begin(), v.end());
```

– end example]

[Note: Because different parallel architectures may require idiosyncratic parameters for efficient execution, implementations of the Standard Library should provide additional execution policies to those described in this Technical Specification as extensions. – end note]

2.2 Header <experimental/execution_policy> synopsis

namespace std {
namespace experimental {
namespace parallel {
    // 2.3, execution policy type trait
    template<class T> struct is_execution_policy;

    // 2.4, sequential execution policy
```
class sequential_execution_policy;

// 2.5, parallel execution policy
class parallel_execution_policy;

// 2.6, vector execution policy
class vector_execution_policy;

// 2.7, dynamic execution policy
class execution_policy;

--// 2.8, specialized algorithms
--void swap(sequential_execution_policy& a, sequential_execution_policy& b);
--void swap(parallel_execution_policy& a, parallel_execution_policy& b);
--void swap(vector_execution_policy& a, vector_execution_policy& b);
--void swap(execution_policy& a, execution_policy& b);

// 2.9, standard execution policy objects
extern const sequential_execution_policy seq;
extern const parallel_execution_policy par;
extern const vector_execution_policy vec;
}
}
}

2.3 Execution policy type trait
[parallel.execpol.type]

namespace std {
namespace experimental {
namespace parallel {

    template<class T> struct is_execution_policy
        : integral_constant<bool, see below> { };

}
}

1. is_execution_policy can be used to detect parallel execution policies for the purpose of excluding
   function signatures from otherwise ambiguous overload resolution participation.

2. If T is the type of a standard or implementation-defined execution policy, is_execution_policy<T>
   shall be publicly derived from integral_constant<bool,true>, otherwise from integral_constant<bool,false>.

3. The behavior of a program that adds specializations for is_execution_policy is undefined.

2.4 Sequential execution policy
[parallel.execpol.seq]

namespace std {
namespace experimental {
namespace parallel {

    class sequential_execution_policy
    {
        void swap(sequential_execution_policy& other);
    }


1. The class `sequential_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm’s execution may not be parallelized.

```cpp
void swap(sequential_execution_policy& other);
```

2. Effects: Swaps the state of *this and other.

2.5 Parallel execution policy [parallel.execpol.par]

```cpp
namespace std {
namespace experimental {
namespace parallel {

---
_class_parallel_execution_policy{
---
--- void swap(parallel_execution_policy& other);
---
_class_parallel_execution_policy{};
}
}
}
```

1. The class `parallel_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm’s execution may be parallelized.

```cpp
void swap(parallel_execution_policy& other);
```

2. Effects: Swaps the state of *this and other.

2.6 Vector execution policy [parallel.execpol.vec]

```cpp
namespace std {
namespace experimental {
namespace parallel {

---
_class_vector_execution_policy{
---
--- void swap(vector_execution_policy& other);
---
_class_vector_execution_policy{};
}
}
}
```
1. The class `vector_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm’s execution may be vectorized.

   ```cpp
   void swap(vector_execution_policy& other);
   ```

2. Effects: Swaps the state of *this and other.

### 2.7 Dynamic execution policy

```
namespace std {
namespace experimental {
namespace parallel {

    class execution_policy
    {
      public:
        // 2.7.1, construct/assign/swap
        template<class T> execution_policy(const T& exec);
        template<class T> execution_policy& operator=(const T& exec);
        void swap(execution_policy& other);

        // 2.7.2, object access
        const type_info& target_type() const;
        const type_info& type() const noexcept;
        template<class T> T* target();
        template<class T> const T* target() const noexcept;
        template<class T> T* get() noexcept;
        template<class T> const T* get() const noexcept;
    }
};
}
}
```

1. The class `execution_policy` is a dynamic container for execution policy objects. `execution_policy` allows dynamic control over standard algorithm execution.

   *Example:*

   ```cpp
   std::vector<float> sort_me = ...;

   std::execution_policy exec = std::seq;

   if(sort_me.size() > threshold)
   {
     exec = std::par;
   }

   std::sort(exec, sort_me.begin(), sort_me.end());
   ```

   – end example

2. Objects of type `execution_policy` shall be constructible and assignable from objects of type `T` for which `is_execution_policy<T>::value` is true.
2.7.1 execution_policy construct/assign/swap

```
template<class T> execution_policy(execution_policy(const T &exec));
```

1. **Effects:** Constructs an `execution_policy` object with a copy of `exec`'s state.
2. **Requires:** `is_execution_policy<T>::value` is true

```
template<class T> execution_policy& operator=(const T& exec);
```

3. **Effects:** Assigns a copy of `exec`'s state to `*this`.
4. **Returns:** `*this`.
5. **Requires:** `is_execution_policy<T>::value` is true

```
void swap(execution_policy& other);
```

1. **Effects:** Swaps the stored object of `*this` with that of `other`.

2.7.2 execution_policy object access

```
const type_info& target_type() const;
const type_info& type() const noexcept;
```

1. **Returns:** `typeid(T)`, such that `T` is the type of the execution policy object contained by `*this`.

```
template<class T> T* target();
template<class T> const T* target() const;
template<class T> T* get() noexcept;
template<class T> const T* get() const noexcept;
```

2. **Returns:** If `target_type()` == `typeid(T)`, a pointer to the stored execution policy object; otherwise a null pointer.
3. **Requires:** `is_execution_policy<T>::value` is true

2.8 Execution-policy specialized algorithms

```
void swap(sequential_execution_policy& a, sequential_execution_policy& b);
void swap(parallel_execution_policy& a, parallel_execution_policy& b);
void swap(vector_execution_policy& a, vector_execution_policy& b);
void swap(execution_policy& a, execution_policy& b);
```

1. **Effects:** `a.swap(b)`.  

2.9 Execution policy objects
[parallel.execpol.objects]

namespace std {
namespace experimental {
namespace parallel {
    constexpr sequential_execution_policy seq = sequential_execution_policy();
    constexpr parallel_execution_policy par = parallel_execution_policy();
    constexpr vector_execution_policy vec = vector_execution_policy();
}
}
}

1. The header <execution_policy> declares a global object associated with each type of execution policy defined by this technical specification.

3 Parallel exceptions
[parallel.exceptions]

3.1 Exception reporting behavior
[parallel.exceptions.behavior]

1. If temporary memory resources are required by the algorithm and none are available, the algorithm throws a std::bad_alloc exception.

2. During the execution of a standard parallel algorithm, if the application of a function object terminates with an uncaught exception, the behavior of the program is determined by the type of execution policy used to invoke the algorithm:

- If the execution policy object is of type vector_execution_policy, std::terminate shall be called.
- If the execution policy object is of type sequential_execution_policy or parallel_execution_policy, the execution of the algorithm terminates with an exception_list exception. All uncaught exceptions thrown during the application of user-provided function objects shall be contained in the exception_list.

[Note: For example, the number of invocations of the user-provided function object in for_each is unspecified. When for_each is executed sequentially, only one exception will be contained in the exception_list object – end note]

[Note: These guarantees imply that, unless the algorithm has failed to allocate memory and terminated with std::bad_alloc, all exceptions thrown during the execution of the algorithm are communicated to the caller. It is unspecified whether an algorithm implementation will “forge ahead” after encountering and capturing a user exception. – end note]

[Note: The algorithm may terminate with the std::bad_alloc exception even if one or more user-provided function objects have terminated with an exception. For example, this can happen when an algorithm fails to allocate memory while creating or adding elements to the exception_list object – end note]

- If the execution policy object is of any other type, the behavior is implementation-defined.

3.2 Header <experimental/exception> synopsis
[parallel.exceptions.synop]

namespace std {
namespace experimental {
namespace parallel {

class exception_list : public exception
{
    public:
        typedef exception_ptr value_type;
        typedef const value_type& reference;
        typedef const value_type& const_reference;
        typedef implementation-defined const_iterator;
        typedef const_iterator iterator;
        typedef typename iterator_traits<const_iterator>::difference_type difference_type;
        typedef size_t size_type;

        size_t size() const noexcept;
        iterator begin() const noexcept;
        iterator end() const noexcept;

    private:
        std::list<exception_ptr> exceptions_; // exposition only
};

1. The class exception_list is a container of exception_ptr objects parallel algorithms may use to communicate uncaught exceptions encountered during parallel execution to the caller of the algorithm.
2. The type exception_list::const_iterator shall fulfill the requirements of ForwardIterator.

size_t size() const noexcept;

2. Returns: The number of exception_ptr objects contained within the exception_list.
3. Complexity: Constant time.

exception_list::iterator begin() const noexcept;

4. Returns: An iterator referring to the first exception_ptr object contained within the exception_list.

exception_list::iterator end() const noexcept;

5. Returns: An iterator which is the past-the-end value for the exception_list.

4 Parallel algorithms [parallel.alg]

4.1 In general [parallel.alg.general]

This clause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

4.1.1 Effect of execution policies on parallel algorithm execution [parallel.alg.general.exec]

1. Parallel algorithms have template parameters named ExecutionPolicy which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply
user-provided function objects.

2. The applications of function objects in parallel algorithms invoked with an execution policy object of type `sequential_execution_policy` execute in sequential order in the calling thread.

3. The applications of function objects in parallel algorithms invoked with an execution policy object of type `parallel_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread. [Note: It is the caller's responsibility to ensure correctness, for example that the invocation does not introduce data races or deadlocks. — end note]

   [Example:
   ```cpp
   using namespace std::experimental::parallel;
   int a[] = {0,1};
   std::vector<int> v;
   for_each(par, std::begin(a), std::end(a), [&](int i) {
       v.push_back(i*2+1);
   });
   ```
   The program above has a data race because of the unsynchronized access to the container `v` — end example]

   [Example:
   ```cpp
   using namespace std::experimental::parallel;
   std::atomic<int> x = 0;
   int a[] = {1,2};
   for_each(par, std::begin(a), std::end(a), [&](int n) {
       x.fetch_add(1, std::memory_order_relaxed);
       // spin wait for another iteration to change the value of x
       while (x.load(std::memory_order_relaxed) == 1) { }
   });
   ```
   The above example depends on the order of execution of the iterations, and is therefore undefined (may deadlock). — end example]

   [Example:
   ```cpp
   using namespace std::experimental::parallel;
   int x;
   std::mutex m;
   int a[] = {1,2};
   for_each(par, std::begin(a), std::end(a), [&](int) {
       m.lock();
       ++x;
       m.unlock();
   });
   ```
   The above example synchronizes access to object `x` ensuring that it is incremented correctly. — end example]

4. The applications of function objects in parallel algorithms invoked with an execution policy of type `vector_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, and unsequenced within each thread. [Note: as a consequence, function objects governed by the `vector_execution_policy` policy must not synchronize with each other. Specifically, they must not acquire locks. — end note]

   [Example:
using namespace std::experimental::parallel;
int x;
std::mutex m;
int a[] = {1,2};
for_each(vec, std::begin(a), std::end(a), [&] (int) {
    m.lock();
    ++x;
    m.unlock();
});

The above program is invalid because the applications of the function object are not guaranteed to run on different threads.

- *end example*

[Note: the application of the function object may result in two consecutive calls to m.lock on the same thread, which may deadlock — *end note*]

[Note: The semantics of the parallel_execution_policy or the vector_execution_policy invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. — *end note*.]

5. If they exist, a parallel algorithm invoked with an execution policy object of type parallel_execution_policy or vector_execution_policy may apply iterator member functions of a stronger category than its specification requires. In this case, the application of these member functions are subject to provisions 3. and 4. above, respectively.

[Note: For example, an algorithm whose specification requires InputIterator but receives a concrete iterator of the category RandomAccessIterator may use operator[]. In this case, it is the algorithm caller’s responsibility to ensure operator[] is race-free. — *end note*.]

6. Algorithms invoked with an execution policy object of type execution_policy execute internally as if invoked with instances of type sequential_execution_policy, parallel_execution_policy, or an implementation-defined execution policy type depending on the dynamic value of the execution_policy object.

7. The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are unspecified.

### 4.1.2 ExecutionPolicy algorithm overloads

1. Parallel algorithms coexist alongside their sequential counterparts as overloads distinguished by a formal template parameter named ExecutionPolicy. This template parameter corresponds to the parallel algorithm’s first function parameter, whose type is ExecutionPolicy&.

2. Unless otherwise specified, the semantics of ExecutionPolicy algorithm overloads are identical to their overloads without.

3. Parallel algorithms have the requirement is_execution_policy<ExecutionPolicy>::value is true.

4. The algorithms listed in table 1 shall have ExecutionPolicy overloads.

<table>
<thead>
<tr>
<th>adjacent_find</th>
<th>for_each</th>
<th>none_of</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_of</td>
<td>for_each_n</td>
<td>nth_element</td>
<td>search_n</td>
</tr>
<tr>
<td>any_of</td>
<td>generate</td>
<td>partial_sort</td>
<td>set_difference</td>
</tr>
<tr>
<td>copy</td>
<td>generate_n</td>
<td>partial_sort_copy</td>
<td>set_intersection</td>
</tr>
</tbody>
</table>
### 4.2 Definitions [parallel.alg.defns]

1. Define \( \text{GENERALIZED\_SUM} (\text{op}, a_1, \ldots, a_N) \) as follows:
   - \( a_1 \) when \( N \) is 1
   - \( \text{op}(\text{GENERALIZED\_SUM} (\text{op}, b_1, \ldots, b_M), \text{GENERALIZED\_SUM} (\text{op}, b_M, \ldots, b_N)) \) where
     - \( b_1, \ldots, b_N \) may be any permutation of \( a_1, \ldots, a_N \) and
     - \( 0 < M < N \).

2. Define \( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM} (\text{op}, a_1, \ldots, a_N) \) as follows:
   - \( a_1 \) when \( N \) is 1
   - \( \text{op}(\text{GENERALIZED\_NONCOMMUTATIVE\_SUM} (\text{op}, a_1, \ldots, a_M), \text{GENERALIZED\_NONCOMMUTATIVE\_SUM} (\text{op}, a_M, \ldots, a_N)) \) where \( 0 < M < N \).

### 4.3 Novel algorithms [parallel.alg.novel]

This subclause describes novel algorithms introduced by this technical specification.

#### 4.3.1 Header <experimental/algorithm> synopsis [parallel.alg.novel.algorithm.synop]

```cpp
namespace std {
  namespace experimental {
    namespace parallel {
      template<class ExecutionPolicy,
        class InputIterator, class Function>
        InputIterator for_each(ExecutionPolicy&& exec,
```
template<class ExecutionPolicy, class InputIterator, class Function>
    InputIterator for_each_n(InputIterator first, Size n, Function f);

template<class ExecutionPolicy, class InputIterator, class Size, class Function>
    Function for_each_n(InputIterator first, Size n, Function f);

1. **Requires:** Function shall meet the requirements of MoveConstructible [Note: Function need not meet the requirements of CopyConstructible]. – end note]

2. **Effects:** Applies f to the result of dereferencing every iterator in the range [first,first + n), starting from first and proceeding to first + n - 1. [Note: If the type of first satisfies the requirements of a mutable iterator, f may apply nonconstant functions through the dereferenced iterator. – end note]
3. **Returns:** `std::move(f)(first + n).

4. **Complexity:** Applies f exactly n times.

5. **Remarks:** If f returns a result, the result is ignored.

```cpp
template<class ExecutionPolicy,
         class InputIterator, class Size, class Function>
InputIterator for_each_n(ExecutionPolicy&& exec,
                         InputIterator first, Size n,
                         Function f);
```

1. **Requires:** Function shall meet the requirements of `MoveConstructible` [Note: Function need not meet the requirements of `CopyConstructible` — end note]

2. **Effects:** Applies f to the result of dereferencing every iterator in the range `{first, first + n}`, starting from first and proceeding to first + n - 1. [Note: If the type of first satisfies the requirements of a mutable iterator, f may apply nonconstant functions through the dereferenced iterator. — end note]

3. **Returns:** `first + n`.

4. **Complexity:** Applies f exactly n times.

5. **Remarks:** If f returns a result, the result is ignored.

### 4.3.3 Header `<experimental/numeric>` synopsis

```cpp
namespace std {
namespace experimental {
namespace parallel {
    template<class InputIterator>
    typename iterator_traits<InputIterator>::value_type
    reduce(InputIterator first, InputIterator last);
    template<class InputIterator, class T>
    T reduce(InputIterator first, InputIterator last T init);
    template<class InputIterator, class T, class BinaryOperation>
    T reduce(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);

    template<class InputIterator, class OutputIterator>
    OutputIterator
    exclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result);
    template<class InputIterator, class OutputIterator, class T>
    OutputIterator
    exclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result, T init);
    template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
    OutputIterator
    exclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result, T init, BinaryOperation binary_op);
```
template<class InputIterator, class OutputIterator>
  OutputIterator
  inclusive_scan(InputIterator first, InputIterator last,
  OutputIterator result);

template<class InputIterator, class OutputIterator, 
class BinaryOperation>
  OutputIterator
  inclusive_scan(InputIterator first, InputIterator last,
  OutputIterator result,
  BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, 
class T, class BinaryOperation>
  OutputIterator
  inclusive_scan(InputIterator first, InputIterator last,
  OutputIterator result,
  T init, BinaryOperation binary_op);

4.3.4 Reduce

template<class InputIterator>
  typename iterator_traits<InputIterator>::value_type
  reduce(InputIterator first, InputIterator last);

1. Returns: reduce(first, last, typename iterator_traits<InputIterator>::value_type(0){})

2. Requires: typename iterator_traits<InputIterator>::value_type(0){} shall be a valid expres-
   sion. The operator+ function associated with iterator_traits<InputIterator>::value_type shall
   not invalidate iterators or subranges, nor modify elements in the range [first,last).

3. Complexity: O(last - first) applications of operator+.

4. Note: The primary difference between reduce and accumulate is that the behavior of reduce may be
   non-deterministic for non-associative or non-commutative operator+.

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

1. Returns: reduce(first, last, init, plus<>())

2. Requires: The operator+ function associated with T shall not invalidate iterators or subranges, nor
   modify elements in the range [first,last).

3. Complexity: O(last - first) applications of operator+.

4. Note: The primary difference between reduce and accumulate is that the behavior of reduce may be
   non-deterministic for non-associative or non-commutative operator+.

template<class InputIterator, class T, class BinaryOperation>
  T reduce(InputIterator first, InputIterator last, T init,
  BinaryOperation binary_op);
1. **Returns:** \( \text{GENERALIZED\_SUM}(\text{binary\_op}, \text{init}, *\text{first}, \ldots, *(\text{first} + \text{last} - \text{first} - 1)) \).

2. **Requires:** \( \text{binary\_op} \) shall not invalidate iterators or subranges, nor modify elements in the range \([\text{first}, \text{last})\).

3. **Complexity:** \( O(\text{last} - \text{first}) \) applications of \( \text{binary\_op} \).

4. **Note:** The primary difference between \text{reduce} and \text{accumulate} is that the behavior of \text{reduce} may be non-deterministic for non-associative or non-commutative \text{operator+}.

### 4.3.5 Exclusive scan

[parallel.alg.novel.exclusive.scan]

\[ \text{template}<\text{class InputIterator, class OutputIterator,}
\text{ class T}>
\text{OutputIterator exclusive\_scan(InputIterator first, InputIterator last,}
\text{ OutputIterator result,}
\text{ T init);} \]

1. **Returns:** \( \text{exclusive\_scan}(\text{first}, \text{last}, \text{result}, \text{init}, \text{plus}<>()) \)

2. **Requires:** The \text{operator+} function associated with \text{iterator\_traits}<\text{InputIterator}>::\text{value\_type}
shall not invalidate iterators or subranges, nor modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

3. **Complexity:** \( O(\text{last} - \text{first}) \) applications of \text{operator+}.

4. **Notes:** The primary difference between \text{exclusive\_scan} and \text{inclusive\_scan} is that \text{exclusive\_scan}
excludes the \( i \)th input element from the \( i \)th sum. If the \text{operator+} function is not mathematically associative, the behavior of \text{exclusive\_scan} may be non-deterministic.

\[ \text{template}<\text{class InputIterator, class OutputIterator,}
\text{ class T, class BinaryOperation}>
\text{OutputIterator exclusive\_scan(InputIterator first, InputIterator last,}
\text{ OutputIterator result,}
\text{ T init, BinaryOperation binary\_op);} \]

1. **Effects:** Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of
\( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{binary\_op}, \text{init}, *\text{first}, \ldots, (*\text{first} + i - \text{result} - 1)) \).

2. **Returns:** The end of the resulting range beginning at \text{result}.

3. **Requires:** \( \text{binary\_op} \) shall not invalidate iterators or subranges, nor modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

4. **Complexity:** \( O(\text{last} - \text{first}) \) applications of \( \text{binary\_op} \).

5. **Notes:** The primary difference between \text{exclusive\_scan} and \text{inclusive\_scan} is that \text{exclusive\_scan}
excludes the \( i \)th input element from the \( i \)th sum. If \( \text{binary\_op} \) is not mathematically associative, the behavior of \text{exclusive\_scan} may be non-deterministic.
4.3.6 Inclusive scan

```
template<class InputIterator, class OutputIterator>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
              OutputIterator result);
```

1. **Returns:** `inclusive_scan(first, last, result, plus<>())`
2. **Requires:** The `operator+` function associated with `iterator_traits<InputIterator>::value_type` shall not invalidate iterators or subranges, nor modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.
3. **Complexity:** $O(last - first)$ applications of `operator+`.
4. **Notes:** The primary difference between `exclusive_scan` and `inclusive_scan` is that `exclusive_scan` excludes the $i$th input element from the $i$th sum. If the `operator+` function is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

```
template<class InputIterator, class OutputIterator,
         class BinaryOperation>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
              OutputIterator result,
              BinaryOperation binary_op);
```

```
template<class InputIterator, class OutputIterator,
         class T, class BinaryOperation>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
              OutputIterator result,
              T init, BinaryOperation binary_op);
```

1. **Effects:** Assigns through each iterator $i$ in `[result,result + (last - first))` the value of 
   `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, *first, ..., (*first + i - result))` or 
   `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, ..., (*first + i - result))` 
   if `init` is provided.
2. **Returns:** The end of the resulting range beginning at `result`.
3. **Requires:** `binary_op` shall not invalidate iterators or subranges, nor modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.
4. **Complexity:** $O(last - first)$ applications of `binary_op`.
5. **Notes:** The primary difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the $i$th input element in the $i$th sum. If `binary_op` is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.