
Contents

1	General [general]	1
1.1	Scope [general.scope]	1
1.2	Normative references [general.references]	1
1.3	Namespaces and headers [general.namespaces]	2
1.4	Terms and definitions [general.defns]	2
2	Execution policies [execpol]	2
2.1	In general [execpol.general]	2
2.2	Header <code><experimental/execution_policy></code> synopsis [execpol.synop]	3
2.3	Execution policy type trait [execpol.type]	4
2.4	Sequential execution policy [execpol.seq]	4
2.5	Parallel execution policy [execpol.par]	4
2.6	Vector execution policy [execpol.vec]	5
2.7	Dynamic execution policy [execpol.dynamic]	5
2.7.1	<code>execution_policy</code> construct/assign/swap	7
2.7.2	<code>execution_policy</code> object access	7
2.8	Execution policy specialized algorithms [execpol.algorithms]	7
2.9	Standard execution policy objects [execpol.objects]	8

1 General [general]

1.1 Scope [general.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 [general.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++

ISO/IEC 14882:2011 is herein called the C++ Standard. The library described in ISO/IEC 14882:2011 clauses 17-30 is herein called the C++ Standard Library. The C++ Standard Library components described in ISO/IEC 14882:2011 clauses 25 and 26.7 are herein called the C++ Standard Algorithms Library.

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 [general.namespaces]

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::parallelism`.

Unless otherwise specified, references to other entities described in this Technical Specification are assumed to be qualified with `std::experimental::parallelism`, 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

```
#include <meow>
```

1.4 Terms and definitions [general.defns]

For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

2 Execution policies [execpol]

2.1 In general [execpol.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:

```
std::vector<int> vec = ...

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

using namespace std::experimental::parallelism;

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

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

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

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

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

– *end example*]

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

2.2 Header <experimental/execution_policy> synopsis [execpol.synop]

```
#include <type_traits>

namespace std {
namespace experimental {
namespace parallelism {
    // 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 [execpol.type]

```
namespace std {
namespace experimental {
namespace parallelism {
    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 non-standard execution policy, `is_execution_policy<T>` shall be publicly derived from `integral_constant<bool,true>`, otherwise from `integral_constant<bool,false>`.
3. The effect of specializing `is_execution_policy` for a type which is not defined by the library is unspecified.

[*Note:* This provision reserves the privilege of creating non-standard execution policies to the library implementation. – *end note.*]

2.4 Sequential execution policy [execpol.seq]

1. The class `sequential_execution_policy` provides a mechanism to require a standard algorithm invocation to execute in a sequential order.

```
namespace std {
namespace experimental {
namespace parallelism {

    class sequential_execution_policy
    {
    public:
        void swap(sequential_execution_policy &other);
    };
}
}
}
```

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

1. *Effects:* Swaps the state of `*this` and `other`.

2.5 Parallel execution policy [execpol.par]

1. The class `parallel_execution_policy` provides a mechanism to allow a standard algorithm invocation to execute in an unordered fashion when executed on separate threads, and indeterminately sequenced when executed on a single thread.

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

```

namespace parallelism {

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

void swap(parallel_execution_policy &other);

```

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

2.6 Vector execution policy [exepol.vec]

1. The class `vector_execution_policy` provides a mechanism to allow a standard algorithm invocation to execute in an unordered fashion when executed on separate threads, and unordered when executed on a single thread.

```

namespace std {
namespace experimental {
namespace parallelism {

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

void swap(vector_execution_policy &other);

```

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

2.7 Dynamic execution policy [exepol.dynamic]

```

namespace std {
namespace experimental {
namespace parallelism {

    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;
    template<class T> T *target();
    template<class T> const T *target() const;
};
}
}
}

```

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

[*Example:*

```

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*]

3. The stored dynamic value of an `execution_policy` object may be retrieved.

[*Example:*

```

void some_api(std::execution_policy exec, int arg1, double arg2)
{
    if(exec.target_type() == typeid(std::seq))
    {
        std::cout << "Received a sequential policy" << std::endl;
        auto *exec_ptr = exec.target<std::sequential_execution_policy>();
    }
    else if(exec.target_type() == typeid(std::par))
    {
        std::cout << "Received a parallel policy" << std::endl;
        auto *exec_ptr = exec.target<std::parallel_execution_policy>();
    }
    else if(exec.target_type() == typeid(std::vec))
    {
        std::cout << "Received a vector policy" << std::endl;
        auto *exec_ptr = exec.target<std::vector_execution_policy>();
    }
    else
    {
        std::cout << "Received some other kind of policy" << std::endl;
    }
}

```

– *end example*]

2.7.1 `execution_policy` construct/assign/swap

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

1. *Effects*: Constructs an `execution_policy` object with a copy of `exec`'s state.
2. *Remarks*: This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

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

1. *Effects*: Assigns a copy of `exec`'s state to `*this`.
2. *Returns*: `*this`.
3. *Remarks*: This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

```
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;
```

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;
```

2. *Returns*: If `target_type() == typeid(T)`, a pointer to the stored execution policy object; otherwise a null pointer.
3. *Remarks*: This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

2.8 Execution policy specialized algorithms [`execpol.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 Standard execution policy objects [execpol.objects]

```
namespace std {  
namespace experimental {  
namespace parallelism {  
    extern const sequential_execution_policy seq;  
    extern const parallel_execution_policy par;  
    extern const vector_execution_policy vec;  
}  
}  
}
```

1. The header `<execution_policy>` declares a global object associated with each standard execution policy.
`const sequential_execution_policy seq;`
2. Concurrent access to these objects shall not result in a data race.
3. The object `seq` requires a standard algorithm to execute sequentially.
`const parallel_execution_policy par;`
4. The object `par` allows a standard algorithm to execute in an unordered fashion when executed on separate threads, and indeterminately sequenced when executed on a single thread.
`const vector_execution_policy vec;`
5. The object `vec` allows a standard algorithm to execute in an unordered fashion when executed on separate threads, and unordered when executed on a single thread.