

Doc No:

Date:

Author:

Allocators post Removal of C++ Concepts (Rev 1)

Contents

⊙

allocator_traits

<memory>
addressof

uses_allocator
pointer_traits
allocator_traits

match_results

Motivation and Background

pair

decltype

Changes from N2946

-

- `pointer_rebind`
`pointer_traits`

Howard Hinnant's

-

-

-

-

- `Allocator::rebind` `Allocator::pointer`
`uses_allocator`

-

Issues and National Body Comments Addressed in this Paper

Document Conventions

[N2960](#)

Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text. Large proposed insertions into the working paper are shown in the same dark blue indented format (no green underline).

Summary

The allocator_traits struct

```
allocator_traits
Allocator
C<T, Alloc>
allocator_traits<Alloc>
n
auto p = allocator_traits<Alloc>::allocate(myalloc, n);

auto p = myalloc.allocate(n);
iterator_traits
allocator_traits
allocator_traits
allocator_traits
```

```

template <class Tp>
class SimpleAllocator
{
public:
    typedef Tp value_type;

    SimpleAllocator(ctor args);

    template <class T> SimpleAllocator(const SimpleAllocator<T>& other);

    Tp* allocate(std::size_t n);
    void deallocate(Tp* p, std::size_t n);
};

```

```

        pointer      reference      rebind
construct destroy   max_size
allocator_traits
        select_on_container_copy_construction
propagate_on_container_copy_assignment
propagate_on_container_move_assignment   propagate_on_container_swap
        allocator_traits

```

allocator_traits

construct destroy

```

allocator_traits
        vector<PodType, SimpleAllocator<PodType>>
memcpy      SimpleAllocator      construct

```

Generalized pointer types

ation to assume that an allocator's pointer value_type*
Allocator pointer value_type* (a.k.a. "fancy" pointers)

pointer

void_pointer

```

        an allocator's pointer
        operator*)
void_pointer                void_pointer
        pointer_traits
        pointer
allocator_traits            pointer const_pointer
void_pointer    const_void_pointer    value_type* const value* void*
const void*

```

Simplified traits and segregation of scoped-allocator functionality

pair tuple

scoped_allocator_adaptor

allocator_traits

pair - pair
 scoped_allocator_adaptor

```

is_scoped_allocator,
constructible_with_allocator_prefix,
constructible_with_allocator_suffix,
allocator_propagate_never,

```

```
allocator_propagate_on_copy_construction,  
allocator_propagate_on_move_assignment,  
allocator_propagate_on_copy_assignment,  
allocator_propagation_map  
  
construct_element
```

Implementation experience

```
list                                     std::list  
  
list                                     allocator_traits  
typedef  
  
scoped_allocator_adaptor                list  
  
allocator_traits pointer_traits  
scoped_allocator_adaptor                list  
allocator_traits
```

Formal Wording

Header <memory> changes

```
// 20.8.1, allocator argument tag  
struct allocator_arg_t { };  
constexpr allocator_arg_t allocator_arg = allocator_arg_t();  
  
// 20.8.2, uses_allocator  
template <class T, class Alloc> struct uses_allocator;
```

```

template <class Alloc> struct is_scoped_allocator;
template <class T> struct constructible_with_allocator_suffix;
template <class T> struct constructible_with_allocator_prefix;

//20.8.3, allocation propagation traits
template <class Alloc> struct allocator_propagate_never;
template <class Alloc> struct allocator_propagate_on_copy_construction;
template <class Alloc> struct allocator_propagate_on_move_assignment;
template <class Alloc> struct allocator_propagate_on_copy_assignment;
template <class Alloc> struct allocator_propagation_map;

//20.8.3 pointer traits
template <class Ptr> struct pointer_traits;
template <class T> struct pointer_traits<T*>;

//20.8.4 allocator traits
template <class Alloc> struct allocator_traits;

//20.8.5, the default allocator:
template <class T> class allocator;
template <> class allocator<void>;
template <class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) throw();
template <class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) throw();

//20.8.6, scoped allocator adaptor
template <class OuterAlloc, class... InnerAllocs = void>
    class scoped_allocator_adaptor;
template <class Alloc>
class scoped_allocator_adaptor<Alloc, void>;
template <class OuterA, class InnerA>
struct is_scoped_allocator<scoped_allocator_adaptor<OuterA, InnerA>>
    : true_type {};
template <class OuterA, class InnerA>
struct allocator_propagate_never<scoped_allocator_adaptor<OuterA, InnerA>>
    : true_type {};
template <class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator==(const scoped_allocator_adaptor<OuterA1, InnerA1Allocs...>& a,
                    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);
template <class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA1Allocs...>& a,
                    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);

//20.8.7, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

//20.8.8, temporary buffers:
template <class T>
    pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
    void return_temporary_buffer(T* p);

//20.8.9, construct element
template <class Alloc, class T, class... Args>
void construct_element(Alloc& alloc, T& r, Args&&... args);

```

// 20.8.10, specialized algorithms:

```
template <class T> T* addressof(T& r);
template <class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
    ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template <class ForwardIterator, class T>
    void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                           const T& x);
template <class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

The addressof function template

```
template <class T> T* addressof(T& r);
template <class T> T* addressof(T&& r);
```

Returns: The actual address of the object referenced by r, even in the presence of an overloaded operator&.

Throws: nothing.

Note to the editor

Allocator Requirements

The library describes a standard set of requirements for allocators, which are [class-type](#) objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the [string types \(Clause 21\)](#), [containers \(except array \(Clause 23\)\)](#), [string buffers and string streams \(Clause 27\)](#), and [match results \(Clause 28\)](#) are parameterized in terms of allocators.

The [template struct allocator_traits \(allocator.traits\)](#) supplies a uniform interface to all allocator types. Table 39 describes the ~~requirements on~~ types manipulated through allocators. ~~All the operations on the allocators are expected to be amortized constant time.~~ Table 40 describes the requirements on allocator types and thus on types used to instantiate [allocator_traits](#). A requirement is *optional* if the last column of Table 40 specifies a default for a given expression. Within the standard library [allocator_traits](#) template, an optional requirement that is not supplied by an allocator is replaced by the specified default

expression. A user specialization of `allocator_traits` may provide different defaults, and may provide defaults for different requirements, than the primary template. Within Tables 39 and 40, the use of `move` and `forward` always refer to `std::move` and `std::forward`, respectively.

Table 39 – Descriptive variable definitions

Variable	Definition
<code>T, U, C</code>	any non-const, non-reference <u>object</u> type
<code>V</code>	a type convertible to <code>T</code>
<code>X</code>	an Allocator class for type <code>T</code>
<code>Y</code>	the corresponding Allocator class for type <code>U</code>
<code>XX</code>	<u>The type <code>allocator_traits<X></code></u>
<code>YY</code>	<u>The type <code>allocator_traits<Y></code></u>
<code>t</code>	a value of type <code>const T&</code>
<code>a, a1, a2</code>	values of type <code>X&</code>
<code>a3</code>	<u>rvalue of type <code>X</code></u>
<code>b</code>	a value of type <code>Y</code>
<code>c</code>	<u>a dereferenceable pointer of type <code>C*</code></u>
<code>p</code>	a value of type <code>X::pointer</code> , obtained by calling <code>a1.allocate</code> , where <code>a1 == a</code>
<code>q</code>	a value of type <code>X::const_pointer</code> obtained by conversion from a value <code>p</code>
<code>w</code>	<u>a value of type <code>XX::void_pointer</code> obtained by conversion from a value <code>p</code></u>
<code>z</code>	<u>a value of type <code>XX::const_void_pointer</code> obtained by conversion from a value <code>q</code> or a value <code>w</code></u>
<code>r</code>	a value of type <code>X::reference T&</code> obtained by the expression <code>*p</code> .
<code>s</code>	a value of type <code>X::const_reference const T&</code> obtained by the expression <code>*q</code> or by conversion from a value <code>r</code> .
<code>u</code>	a value of type <code>Y::const_pointer</code> obtained by calling <code>Y::allocate</code> , or else <code>0nullptr</code> .
<code>v</code>	a value of type <code>V</code>
<code>n</code>	a value of type <code>X::size_type</code>
<code>Args</code>	a template parameter pack
<code>args</code>	a function parameter pack with the pattern <code>Args&&</code>

Table 40 – Allocator requirements

Expression	Return type	Assertion/note pre-/post-condition	Default
<code>X::pointer</code>	Pointer to <code>T</code>		<code>T*</code>
<code>X::const_pointer</code>	Pointer to const <code>T</code>	<code>X::pointer</code> is convertible to <code>X::const_pointer</code>	<code>pointer_traits<X>::pointer::rebind<const T></code>
<code>X::void_pointer</code> <code>Y::void_pointer</code>		<code>X::pointer</code> is convertible to <code>X::void_pointer</code> and <code>X::void_pointer</code>	<code>pointer_traits<X>::pointer::rebind<void></code>

		<u>Y::void_pointer</u> are the same type.	
<u>X::const void pointer</u> <u>Y::const void pointer</u>		<u>X::pointer</u> , <u>X::const_pointer</u> , and <u>X::void_pointer</u> are all convertible to <u>X::const void_pointer</u> . <u>X::const void_pointer</u> and <u>Y::const void_pointer</u> are the same type.	<u>pointer_traits<X::pointer>::rebind<const void></u>
X::reference	T&		
X::const_reference	T const&		
X::value_type	Identical to T		
X::size_type	unsigned integral type	a type that can represent the size of the largest object in the allocation model.	<u>size_t</u>
X::difference_type	signed integral type	a type that can represent the difference between any two pointers in the allocation model.	<u>ptrdiff_t</u>
typename X::template rebind<U>::other	Y	For all U (including T), Y::template rebind<T>::other is X.	<u>See Note A, below.</u>
*p	<u>T&</u>		
*q	<u>T const&</u>	<u>*q refers to the same object as *p</u>	
<u>p->m</u>	<u>type of T::m</u>	<u>pre: (*p).m is well-defined.</u> <u>equivalent to (*p).m</u>	
<u>q->m</u>	<u>type of T::m</u>	<u>pre: (*q).m is well-defined.</u> <u>equivalent to (*q).m</u>	
<u>static cast<X::pointer>(w)</u>	<u>X::pointer</u>	<u>static cast<X::pointer>(w) == p</u>	
<u>static cast<X::const pointer>(z)</u>	<u>X::const_pointer</u>	<u>static cast<X::const pointer>(z) == q</u>	
<u>pointer d(nullptr);</u> <u>pointer d = nullptr;</u> <u>const_pointer e(nullptr);</u> <u>const_pointer e = nullptr;</u>		<u>d and e are null pointers and need not be dereferenceable.</u> <u>static_cast<bool>(d) == false,</u> <u>static_cast<bool>(e) == false</u>	
<u>void_pointer d(nullptr);</u> <u>void_pointer d = nullptr;</u> <u>const void_pointer e(nullptr);</u> <u>const void_pointer e = nullptr;</u>		<u>d and e are null pointers and need not be dereferenceable.</u> <u>static_cast<bool>(d) == false,</u> <u>static_cast<bool>(e) == false</u>	
<u>p</u>	<u>contextually convertible to bool</u>	<u>false if p is a null pointer, else true</u>	
<u>q</u>	<u>contextually convertible to bool</u>	<u>false if q is a null pointer, else true</u>	
<u>w</u>	<u>contextually convertible to bool</u>	<u>false if w is a null pointer, else true</u>	
<u>z</u>	<u>contextually convertible</u>	<u>false if z is a null pointer, else true</u>	

	<u>to bool</u>		
<u>!p</u>	<u>convertible to bool</u>	<u>true if p is a null pointer, else false</u>	
<u>!q</u>	<u>convertible to bool</u>	<u>true if q is a null pointer, else false</u>	
<u>!w</u>	<u>convertible to bool</u>	<u>true if w is a null pointer, else false</u>	
<u>!z</u>	<u>convertible to bool</u>	<u>true if z is a null pointer, else false</u>	
a.address(r)	X::pointer		
a.address(s)	X::const_pointer		
a.allocate(n) a.allocate(n,u)	X::pointer	Memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception. The result is a random access iterator. ²³¹ [Note: If n == 0, the return value is unspecified. — end note]	
<u>a.allocate(n,u)</u>	<u>X::pointer</u>	<u>a.allocate(n)</u> <u>u</u>	<u>a.allocate(n)</u>
a.deallocate(p,n)	(not used)	All n T objects in the area pointed to by p shall be destroyed prior to this call. n shall match the value passed to allocate to obtain this memory. Does not throw exceptions. [Note: p shall not be null <u>singular</u> . — end note]	
a.max_size()	X::size_type	the largest value that can meaningfully be passed to X::allocate()	<u>numeric_limits<size_type>::max()</u>
a1 == a2	bool	returns true iff storage allocated from each can be deallocated via the other. operator== shall be reflexive, symmetric, and transitive <u>and shall not exit via an exception.</u>	
a1 != a2	bool	same as !(a1 == a2)	
<u>a == b</u>	<u>bool</u>	<u>same as a ==</u> <u>Y::rebind<T>::other(b)</u>	
<u>a != b</u>	<u>bool</u>	<u>same as !(a == b)</u>	
X()		creates a default instance. [Note: a destructor is assumed. — end note]	
X a1(a);		post: a1 == a. <u>Shall not exit via an exception.</u>	
X a(b);		post: Y(a) == b, a == X(b). <u>Shall not exit via an exception.</u>	
<u>X a1(move(a));</u>		<u>post: a1 equals the prior value of a. Shall not exit via an exception.</u>	
<u>X a(move(b));</u>		<u>post: a equals the prior value of X(b). Shall not exit via an exception.</u>	
a.construct(p c, args)	(not used)	Effect: Constructs an object of type T C at p c by invoking T C(forward<Args>(args)...) <u>T(forward<Args>(args)...) </u>	<u>new ((void*)c)</u> <u>C(forward<Args>(args)...) </u>
a.destroy(p c)	(not used)	Effect: Destroys the object at p c	<u>c->~T()</u>
<u>a.select on contain</u>	<u>X</u>	<u>Typically returns either a or X()</u>	<u>return a;</u>

<code>er copy construction()</code>			
<code>X::propagate_on_container_copy_assignment</code>	<u>identical-to or derived-from true_type or false_type</u>	<u>true_type if an allocator of type X should be copied when the client container is copy-assigned.</u>	<u>false_type</u>
<code>X::propagate_on_container_move_assignment</code>	<u>identical-to or derived-from true_type or false_type</u>	<u>true_type if an allocator of type X should be moved when the client container is move-assigned.</u>	<u>false_type</u>
<code>X::propagate_on_container_swap</code>	<u>identical-to or derived-from true_type or false_type</u>	<u>true_type if an allocator of type X should be swapped when the client container is swapped.</u>	<u>false_type</u>

Note A: The member class template `rebind` in the table above is effectively a typedef template. [Note: In general, if the name `Allocator` is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`, where `SomeAllocator<T>::value_type` is `T` and `SomeAllocator<U>::value_type` is `U`. – end note] If `Allocator` is a class template instantiation of the form `SomeAllocator<T, args>`, where `args` is zero or more type arguments, and `Allocator` does not supply a `rebind` member template, the standard allocator traits template uses `SomeAllocator<U, args>` in place of `Allocator::rebind<U>::other`, by default. For allocator types that are not template instantiations of the above form, no default is provided.

The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall satisfy the requirements of `EqualityComparable`, `DefaultConstructible`, `CopyConstructible`, `CopyAssignable`, `Swappable`, and `Destructible` (20.2.1 [utility.arg.requirements]). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. A default initialized object may have a singular value. A value-initialized object shall compare equal to `nullptr`. `X::pointer` and `X::const_pointer` shall also satisfy the requirements for a random-access iterator (24.1 [iterator.requirements]).

```

void_pointer    const_void_pointer

                construct    destroy

                value_type

                address                address
pointer_to     pointer_traits

                select_on_container_move_construction()

```

An allocator may constrain the types on which it may be instantiated or the arguments for which its `construct` member may be called. If a type cannot be used with a particular allocator, the allocator class or the call to `construct` may fail to instantiate.

[Example: The following is an allocator class template supporting the minimal interface that satisfies the requirements in Table 40:

```
template <class Tp>
class SimpleAllocator
{
public:
    typedef Tp value_type;

    SimpleAllocator(ctor args) ;

    template <class T> SimpleAllocator(const SimpleAllocator<T>& other);

    Tp* allocate(std::size_t n);
    void deallocate(Tp* p, std::size_t n);
};
```

– end example]

~~Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following requirement beyond those in Table 40.~~

~~—The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `std::size_t`, and `std::ptrdiff_t`, respectively.~~

~~Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 40 are implementation defined.~~

If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [Note:

additionally, the member function `allocate` for that type may fail by throwing an object of type `std::bad_alloc`.— *end note*]

The `uses_allocator` trait

20.8.1 Allocator argument tag [allocator.tag]

```
namespace std {
    struct allocator_arg_t { };
    constexpr allocator_arg_t allocator_arg = allocator_arg_t();
}
```

The `allocator_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see [pair, 20.3.3](#) [tuple 20.5.2.1](#)) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (20.2.2).

~~20.8.2 Allocator-related traits [allocator.traits]~~

~~Etc.~~

20.8.2 uses_allocator [uses.allocator]

20.8.2.1 uses_allocator_trait [uses.allocator.trait]

```
template <class T, class Alloc> struct uses_allocator;
```

Remark: Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the `BinaryTypeTrait` requirements (20.5.1). The implementation shall provide a definition that is derived from `true_type` if a type `T::allocator_type` exists and is convertible<Alloc, T::allocator_type>::value != false, otherwise it shall be derived from `false_type`. A program may specialize this ~~type~~template to derive from `true_type` for a user-defined type `T` that does not have a nested `allocator_type` but ~~is~~ nonetheless ~~constructible using the specified Alloc~~ can be constructed with an allocator where either:

- The first two arguments of a constructor have types `allocator_arg_t`, `Alloc`, or
- The last argument of a constructor has type `Alloc`.

~~Remark: `uses_allocator<T, Alloc>` shall be derived from `true_type` if `Convertible<Alloc, T::allocator_type>`, otherwise derived from `false_type`.~~

~~The class templates `is_scoped_allocator`, `constructible_with_allocator_suffix`, and `constructible`, ... [rest of section removed]~~

20.8.2.2 uses-allocator construction [uses.allocator.construction]

uses-allocator construction with allocator Alloc refers to the construction of an object, `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN` and an allocator `alloc` of type `Alloc` using the following rules:

- If `uses_allocator<T,Alloc>::value == false` && `is_constructible<T,V1,V2, ..., VN>::value == true`, then `obj` is initialized as `obj(v1, v2, ..., vN)`.
- Otherwise, if `(uses_allocator<T,Alloc>::value && is_constructible<T, allocator_arg_t, Alloc,V1, V2, ..., VN>::value) == true`, then `obj` is initialized as `obj(allocator_arg, alloc, v1, v2, ..., vN)`.
- Otherwise, if `(uses_allocator<T,Alloc>::value && is_constructible<T,V1, V2, ..., VN, Alloc>::value) == true`, then `obj` is initialized as `obj(v1, v2, ..., vN, alloc)`.
- Otherwise the request for uses-allocator construction is ill formed. [Note: an error will result if `uses_allocator<T,Alloc>::value` is true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to an element. – end note]

The `pointer_traits` template

~~20.8.3 Allocator propagation traits [allocator.propagation]~~

~~Etc.~~

~~20.8.4 Allocator-related element concepts [allocator.element.concepts]~~

~~Etc.~~

20.8.3 Pointer traits [pointer.traits]

The struct template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
    template <class Ptr> struct pointer_traits
    {
        typedef Ptr    pointer;
        typedef see below element_type;
        typedef see below difference_type;

        template <class U> using rebind = see below;

        static pointer pointer_to(see below r);
    };
}
```

```

};

template <class T>
struct pointer_traits<T*>
{
    typedef T          element_type;
    typedef T*        pointer;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r);
};
}

```

20.8.3.1 Pointer traits member types

```
typedef see below element_type;
```

Type: `Ptr::element_type` if such a type exists; otherwise, `T` if `Ptr` is a class template instantiation of the form `SomePointer<T, args>`, where `args` is zero or more type arguments; otherwise the specialization is ill-formed.

```

boost::pointer_to_other

template <class _Tp> struct __first_param;

template <template <class, class...> class _Tmpl, class _Tp,
         class... _Args>
struct __first_param<_Tmpl<_Tp, _Args...> > {
    typedef _Tp type;
};

```

```
typedef see below difference_type;
```

Type: `Ptr::difference_type` if such a type exists; otherwise `std::ptrdiff_t`.

```
template <class _U> using rebind = see below;
```

Template Alias: `Ptr::rebind<U>` if such a type exists; otherwise, if `Ptr` is a class template instantiation of the form `SomePointer<T, args>`, where `args` is zero or more type arguments; otherwise the instantiation of `rebind` is ill-formed.

```
boost::pointer_to_other
```

20.8.3.2 Pointer traits member functions

```
Static pointer pointer_to(see below r);
```

Preconditions: For every object type `T`, `r` shall be `T&`; for `cv void`, the type of `r` is unspecified.

Returns: a dereferenceable pointer to `r` obtained by calling `Ptr::pointer_to(r)`. An instantiation of this function is ill formed if `Ptr` does not have matching `pointer_to` static member function.

```
Static pointer pointer_to(see below r); (for the partial specialization, pointer_traits<T*>)
```


Preconditions: For every object type `T`, `r` shall be `T&`; for `cv void`, the type of `r` is unspecified.

Returns: `std::addressof(r)`.

The `allocator_traits` template

20.8.4 Allocator traits [allocator.traits]

The struct template `allocator_traits` supplies a uniform interface to all allocator types. A user specialization of `allocator_traits` in the `std` namespace for a specific set of allocators, `X` shall implement the interface described below in such a way that `allocator_traits<X>` meets the requirements of table 40 (see 20.2.2 [allocator.requirements]). An allocator cannot be a non-class type, however, even if `allocator_traits` supplies the entire required interface. [*Note:* thus, it is always possible to create a derived class from an allocator – *end note*]

```
namespace std {
    template <class Alloc> struct allocator_traits {

        typedef Alloc allocator_type;

        typedef typename Alloc::value_type value_type;

        typedef see below pointer;
        typedef see below const_pointer;
        typedef see below void_pointer;
        typedef see below const_void_pointer;

        typedef see below difference_type;
        typedef see below size_type;

        typedef see below propagate_on_container_copy_assignment;
        typedef see below propagate_on_container_move_assignment;
        typedef see below propagate_on_container_swap;

        template <class T> using rebind_alloc = see below;
        template <class T> using rebind_traits =
            allocator_traits<rebind_alloc<T> >;

        static pointer allocate(Alloc& a, size_type n);
        static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);

        static void deallocate(Alloc& a, pointer p, size_type n);

        template <class T, class... Args>
            static void construct(Alloc& a, T* p, Args&&... args);

        template <class T>
            static void destroy(Alloc& a, T* p);
    };
}
```

```

    static size_type max_size(const Alloc& a);

    static Alloc select_on_container_copy_construction(const Alloc& rhs);
};
}

```

20.8.3.1 Allocator traits member types

```
typedef see below pointer;
```

Type: Alloc::pointer if such a type exists, otherwise value_type*.

```
typedef see below const_pointer;
```

Type: Alloc::const_pointer if such a type exists, otherwise pointer_traits<pointer>::rebind<const value_type>.

```
typedef see below void_pointer;
```

Type: Alloc::void_pointer if such a type exists, otherwise pointer_traits<pointer>::rebind<void>.

```
typedef see below const_void_pointer;
```

Type: Alloc::const_void_pointer if such a type exists, otherwise pointer_traits<pointer>::rebind<const void>.

```
typedef see below difference_type;
```

Type: Alloc::difference_type if such a type exists, otherwise ptrdiff_t.

```
typedef see below size_type;
```

Type: Alloc::size_type if such a type exists, otherwise size_t.

```
typedef see below propagate_on_container_copy_assignment;
```

Type: Alloc::propagate_on_container_copy_assignment if such a type exists, otherwise false_type.

```
typedef see below propagate_on_container_move_assignment;
```

Type: Alloc::propagate_on_container_move_assignment if such a type exists, otherwise false_type.

```
typedef see below propagate_on_container_swap;
```

Type: Alloc::propagate_on_container_swap if such a type exists, otherwise false_type.

```
template <class T> using rebind_alloc = see below;
```

Template Alias; Alloc::rebind<U>::other if such a type exists; otherwise, if Alloc is a class template instantiation of the form Alloc<T, args>, where args is zero or more type arguments; otherwise the instantiation of rebind_alloc is ill-formed.

20.8.3.2 Allocator traits static member functions

```
static pointer allocate(Alloc& a, size_type n);
```

Returns: a.allocate(n).

```
static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
```

Returns: a.allocate(n, hint) if such an expression would be well formed, otherwise a.allocate(n).

```
static void deallocate(Alloc& a, pointer p, size_type n);
```

Effects: calls a.deallocate(p, n).

```
template <class T, class... Args>  
static void construct(Alloc& a, T* p, Args&&... args);
```

Effects: calls a.construct(p, std::forward<Args>(args)...) if such a call would be well formed, otherwise invokes

```
new (static_cast<void*>(p)) T(std::forward<Args>(args)...).
```

```
template <class T>  
static void destroy(Alloc& a, T* p);
```

Effects: calls a.destroy(p) if such a call would be well formed, otherwise invokes p->~T().

```
static size_type max_size(const Alloc& a);
```

Returns: a.max_size() if such a call would be well-formed, otherwise numeric_limits<size_type>::max().

```
static Alloc select_on_container_copy_construction(const Alloc& rhs);
```

Returns: rhs.select_on_container_copy_construction() if such a call would be well formed, otherwise rhs.

Changes to the default allocator

```
construct() destroy()
```

```
template<class U, class... Args> void construct(pointerU* p, Args&&... args);  
template<class U> void destroy(pointerU* p);
```

```
construct() destroy()
```

```
template<class U, class... Args> void construct(pointerU* p, Args&&... args);
```

Effects: ::new((void *)p) ~~T~~U(std::forward<Args>(args)...)

```
template<class U> void destroy(pointerU* p);
```

Effects: p->~~~T~~U()

20.8.9 construct_element [construct.element]

~~Etc.~~

Scoped allocator adaptors

20.8.6 Scoped allocator adaptor [allocator.adaptor]

The `scoped_allocator_adaptor` class template is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element in the container, and, if the elements are themselves containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements are themselves containers, the third allocator is passed to the elements' elements, etc.. If containers are nested to a depth greater than the number of allocators, then the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [Note: The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. —end note]

```
namespace std {
    template <class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor : public OuterAlloc
    {
        typedef allocator_traits<OuterAlloc>      OuterTraits; // exposition only
        scoped_allocator_adaptor<InnerAllocs...> inner;          // exposition only

    public:
        typedef OuterAlloc                          outer_allocator_type;
        typedef see below                          inner_allocator_type;

        typedef typename OuterTraits::size_type     size_type;
        typedef typename OuterTraits::difference_type difference_type;
        typedef typename OuterTraits::pointer       pointer;
        typedef typename OuterTraits::const_pointer const_pointer;
        typedef typename OuterTraits::void_pointer void_pointer;
        typedef typename OuterTraits::const_void_pointer const_void_pointer;
        typedef typename OuterTraits::value_type    value_type;

        typedef see below propagate_on_container_copy_assignment;
        typedef see below propagate_on_container_move_assignment;
        typedef see below propagate_on_container_swap;

        template <class Tp>
        struct rebind {
            typedef scoped_allocator_adaptor<
                OuterTraits::template rebind_alloc<Tp>, InnerAllocs...> other;
        };

        scoped_allocator_adaptor();
        template <class OuterA2>
            scoped_allocator_adaptor(OuterA2&& outerAlloc,
                                    const InnerAllocs&... innerAllocs);

        scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

        template <class OuterA2>
            scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
                                    InnerAllocs...>& other);
    };
};
```

```

template <class OuterA2>
    scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
                            InnerAllocs...>&& other);

~scoped_allocator_adaptor();

inner_allocator_type      & inner_allocator();
inner_allocator_type const& inner_allocator() const;
outer_allocator_type      & outer_allocator();
outer_allocator_type const& outer_allocator() const;

pointer allocate(size_type n);
pointer allocate(size_type n, const_void_pointer hint);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template <class T, class... Args>
    void construct(T* p, Args&&... args);

```

```

// Specializations to pass inner_allocator to pair::first and pair::second
template <class T1, class T2>
    void construct(std::pair<T1,T2>* p);
template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, U&& x, V&& y);
template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, const std::pair<U, V>& pr);
template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, std::pair<U, V>&& pr);

```

```

template <class T>
    void destroy(T* p);

// Allocator propagation functions.
scoped_allocator_adaptor select_on_container_copy_construction() const;
};

template <class OuterA1, class OuterA2, class... InnerAllocs>
inline
bool operator==(const scoped_allocator_adaptor<OuterA1,InnerAllocs...>& a,
                const scoped_allocator_adaptor<OuterA2,InnerAllocs...>& b);

template <class OuterA1, class OuterA2, class... InnerAllocs>
inline
bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerAllocs...>& a,
                const scoped_allocator_adaptor<OuterA2,InnerAllocs...>& b);
}

```

20.8.6.1 `scoped_allocator_adaptor` member types [allocator.adaptor.types]

typedef *see below* inner_allocator_type;

Type: If `sizeof... (InnerAllocs)` is zero, `scoped_allocator_adaptor<OuterAlloc>`, otherwise `scoped_allocator_adaptor<InnerAllocs...>`

```
typedef see below propagate_on_container_copy_assignment;
```

Type: true_type if

allocator_traits<A>::propagate_on_container_copy_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs... ; otherwise false_type.

```
typedef see below propagate_on_container_move_assignment;
```

Type: true_type if

allocator_traits<A>::propagate_on_container_move_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs... ; otherwise false_type.

```
typedef see below propagate_on_container_swap;
```

Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any A in the set of OuterAlloc and InnerAllocs... ; otherwise false_type.

20.8.6.2 scoped_allocator_adaptor constructors [allocator.adaptor.cntnr]

```
scoped_allocator_adaptor();
```

Effects: value-initializes the OuterAlloc base class and the inner allocator object

```
template <class OuterA2>
    scoped_allocator_adaptor(OuterA2&& outerAlloc,
                            const InnerAllocs&... innerAllocs);
```

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes the OuterAlloc base class with std::forward<OuterA2>(outerAlloc) and inner with innerAllocs... (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

```
scoped_allocator_adaptor(const scoped_allocator_adaptor& other);
```

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

```
template <class OuterA2>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
                            InnerAllocs...>& other);
```

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

```
template <class OuterA2>
    scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
                            InnerAllocs...>&& other);
```

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator rvalue from other.

20.8.6.3 `scoped_allocator_adaptor` members [`allocator.adaptor.members`]

```
inner_allocator_type      & inner_allocator();  
inner_allocator_type const& inner_allocator() const;
```

Returns: if `sizeof... (InnerAllocs)` is zero, `*this`, else `inner`

```
outer_allocator_type      & outer_allocator();  
outer_allocator_type const& outer_allocator() const;
```

Returns: `static_cast<Outer&>(*this)` or `static_cast<Outer const&>(*this)`, respectively.

```
pointer allocate(size_type n);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`

```
pointer allocate(size_type n, const_void_pointer hint);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`

```
void deallocate(pointer p, size_type n);
```

Effects: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n)`

```
size_type max_size() const;
```

Returns: `allocator_traits<OuterAlloc>::max_size(outer_allocator())`

```
template <class T, class... Args>  
void construct(T* p, Args&&... args);
```

Effects: let `OUTERMOST(x)` be `x` if `x` does not have an `outer_allocator()` function, and `OUTERMOST(x.outer_allocator())` otherwise.

- If `uses_allocator<T, inner_allocator_type>::value == false` && `is_constructible<T, Args...>::value == true`, call `OUTERMOST(*this).construct(p, std::forward<Args>(args)...) .`
- Otherwise, if `(uses_allocator<T, inner_allocator_type>::value && is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value) == true`, then calls `OUTERMOST(*this).construct(p, allocator_arg, inner_allocator(), std::forward<Args>(args)...) .`
- Otherwise, if `(uses_allocator<T, inner_allocator_type>::value && is_constructible<T, Args..., inner_allocator_type>::value) == true`, then calls `OUTERMOST(*this).construct(p, std::forward<Args>(args)..., inner_allocator()) .`
- Otherwise the program is ill formed. [Note: an error will result if `uses_allocator` evaluates true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. – end note]

```
template <class T>  
void destroy(T* p);
```

Effects: calls `outer_allocator().destroy(p)`

```
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

Returns: a new `scoped_allocator_adaptor` where each allocator, A, in the adaptor is initialized from the result of calling `allocator_traits<A>::select_on_container_copy_construction` on the corresponding allocator in `*this`.

Changes to container and string wording

In all container and string classes and `match_results`, and `sub_match`, make the following replacements in the `typedef` sections of the class template definitions:

Old type	New type
<code>Allocator::value_type</code>	<code>allocator_traits<value_type>::value_type</code>
<code>Allocator::reference</code>	<code>value_type&</code>
<code>Allocator::const_reference</code>	<code>const value_type&</code>
<code>Allocator::pointer</code>	<code>allocator_traits<Allocator>::pointer</code>
<code>Allocator::const_pointer</code>	<code>allocator_traits<Allocator>::const_pointer</code>
<code>Allocator::difference_type</code>	<code>allocator_traits<Allocator>::difference_type</code>
<code>Allocator::size_type</code>	<code>allocator_traits<Allocator>::size_type</code>

The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects.²³⁵

The class template `basic_string` conforms to the requirements for a Sequence Container (23.2.3), for a Reversible Container (23.2), and for an Allocator-aware container (93), except that `basic_string` does not construct or destroy its elements using `allocator_traits<Alloc>::construct` and `allocator_traits<Alloc>::destroy`. The iterators supported by `basic_string` are random access iterators (24.2.5).

In every specialization `basic_string<charT, traits, Allocator>`, the nested type `allocator_traits<Allocator>::value_type` shall name the same type as `charT`. Every object of type `basic_string<charT, traits, Allocator>` shall use an object of type `Allocator` to allocate and free storage for the contained `charT` objects as needed. The `Allocator` object used shall be obtained as described in Section 23.2.1 [container.requirements.general] ~~a copy of the `Allocator` object passed to the `basic_string` object's constructor or, if the constructor does not take an `Allocator` argument, a copy of a default-constructed `Allocator` object.~~

```
basic_string(const basic_string<charT, traits, Allocator>& str);
basic_string(basic_string<charT, traits, Allocator>&& str);
```

Effects: Constructs an object of class `basic_string` as indicated in Table 60. ~~In the first form, the stored `Allocator` value is copied from `str.get_allocator()`. In the second form, the stored `Allocator` value is move constructed from `str.get_allocator()`, and `str` is left in a valid state with an unspecified value.~~

237) `reserve()` uses `allocator_traits<Allocator>::allocate()` which may throw an appropriate exception.

```
allocator_type get_allocator() const;
```

Returns: a copy of the `Allocator` object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

- 3 For the components affected by this clause that declare an `allocator_type`, objects stored in these components shall be constructed using `construct_element(20.8.9)` the `allocator_traits<allocator_type>::construct` function and destroyed using the `allocator_traits<allocator_type>::destroy` function (20.8.3.2 [allocator.traits.funcs]). These `construct` and `destroy` functions are called only for the container's element type, not for internal types used by the container. [Note: This means, for example, that a node-based container might need to construct nodes containing aligned buffers and call `construct` to place the element into the buffer. — end note] For each operation that inserts an element of type `T` into a container (`insert`, `push_back`, `push_front`, `emplace`, etc.) with arguments `args...`, `T` shall be `ConstructibleAsElement`, as described in table 89. [Note: If the component is instantiated with a scoped allocator of type `A` (i.e., an allocator for which `is_scoped_allocator<A>::value` is true), then `construct_element` may pass an inner allocator argument to `T`'s constructor. — end note]
- 4 In table 90, `T` denotes an object type, `A` denotes an allocator, `I` denotes an allocator of type `A::inner_allocator_type` (if any), and `Args` denotes a template parameter pack

⊙

Table 90 — `ConstructibleAsElement<A, T, Args>` requirements [constructibleaselement]

Etc.

Notes: the algorithms `swap()`, `equal()` and `lexicographical_compare()` are defined in Clause 25. Those entries marked “(Note A)” or “(Note B)” should have constant complexity. ~~Those entries marked “(Note B)” have constant complexity unless `allocator_propagate_never<X::allocator_type>::value` is true, in which case they have linear complexity.~~

Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (See 20.2.2). Copy ~~and move~~ constructors for these container types obtain an allocator by calling ~~`Allocator<allocator_type>::select_for_copy_construction` or `Allocator<allocator_type>::select_for_move_construction`~~ `allocator_traits<allocator_type>::select` on container copy construction on their ~~respective~~ first parameters. Move constructors obtain an allocator by move-construction of the allocator belonging to the container being moved. Such move-construction of the allocator shall not exit with an

exception. All other constructors for these container types take an `Allocator` argument (20.2.2), an allocator whose value type is the same as the container's value type. [Note: If an invocation of a constructor uses the default value of an optional allocator argument, then the `Allocator` type must support value initialization – end note] A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by calling `Allocator<allocator_type>::do_on_container_copy_assignment`, `Allocator<allocator_type>::do_on_container_copy_assignment`, `Allocator<allocator_type>::do_on_swap` copy assignment, move assignment, or swapping of the allocator only if `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true within the implementation of the corresponding container operation. A call to a container's `swap` function will yield undefined behavior unless the objects being swapped have allocators that compare equal or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true. Calling the preceding `allocator_traits` functions may or may not modify the allocator, depending on the implementation of those functions for the specific allocator type. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator object used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

<pre>X(t,m) X u(t,m);</pre>	<p>Requires: <code>ConstructibleAsElement<A, T, T></code> <u>post:</u> <code>u == t</code>, <code>get_allocator() == m</code></p>	<p>linear</p>
<pre>X(rv) X u(rv);</pre>	<p><u>requires:</u> move-construction of A shall not exit with an exception. <u>post:</u> <code>u</code> shall have the same elements as <code>rv</code> had before this construction; <code>get_allocator()</code> shall be the same as the value of <code>rv.get_allocator()</code> before this construction.</p>	<p>constant</p>
<pre>X(rv,m) X u(rv,m);</pre>	<p>Requires: <code>ConstructibleAsElement<A, T, T&&></code> <u>post:</u> <code>u</code> shall be equal to the value <u>have the same elements, or copies of the elements</u>, that <code>rv</code> had before this construction; <code>get_allocator() == m</code></p>	<p>constant if <code>m == rv.get_allocator()</code>, otherwise linear</p>

`construct_element` `construct`

Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the `bool` value type map to bit values in the container storage and `AllocatableElement::construct` (23.2) (20.8.4) `allocator_traits::construct` (20.8.4.2) is not used to construct these values.

Tuple changes

constructible_with_allocator_prefix

```
template <class... Types>  
struct constructible_with_allocator_prefix<tuple<Types...>>;
```

```
template <class... Types>  
struct constructible_with_allocator_prefix<tuple<Types...>>  
: true_type {};
```

~~[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator prefix argument. end note]~~

```
template <class Alloc>  
tuple(allocator_arg_t, const Alloc& a);  
template <class Alloc>  
tuple(allocator_arg_t, const Alloc& a, const Types&...);  
template <class Alloc, class... UTypes>  
tuple(allocator_arg_t, const Alloc& a, const UTypes&&...);  
template <class Alloc>  
tuple(allocator_arg_t, const Alloc& a, const tuple&);  
template <class Alloc>  
tuple(allocator_arg_t, const Alloc& a, tuple&&);  
template <class Alloc, class... UTypes>  
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);  
template <class Alloc, class... UTypes>  
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);  
template <class Alloc, class U1, class U2>  
tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);  
template <class Alloc, class U1, class U2>  
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

Requires: Alloc shall ~~be an~~ meet the requirements for an Allocator (20.2.2).

Effects: Equivalent to the preceding constructors except that each element is constructed with *uses-allocator construction* with allocator a (20.8.2.2 [uses.allocator.construction]). ~~the allocator argument is passed conditionally to the constructor of each element. Each member is allocator-constructed (20.8.2) with a.~~

allocator constructed

Function changes

```
template<class R, class... ArgTypes>
struct constructible_with_allocator_prefix<
function<R(ArgTypes...)>>
: true_type { };
```

20.7.15.2.1 function construct/copy/destroy [func.wrap.func.con]

For all constructors in this clause that have arguments lists beginning with types `allocator_arg_t`, `const A&`, type `A` shall be assumed by the implementation to conform to the allocator requirements in Table 40 [allocator.requirements]. A copy of the allocator argument is used to allocate memory, if necessary for the internal data structures of the constructed function object.

```
explicit function();
template <class A> function(allocator arg t, const A& a);
```

Postconditions: `!*this`.

Throws: nothing.

```
function(nullptr_t);
template <class A> function(allocator arg t, const A& a, nullptr_t);
```

Postconditions: `!*this`.

Throws: nothing.

```
function(const function& f);
template <class A> function(allocator arg t, const A& a, const function& f);
```

Postconditions: `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

Throws: shall not throw exceptions if `f`'s target is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object. [*Note:* Implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, e.g., where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. —*end note*]

```
function(function&& f);
template <class A> function(allocator arg t, const A& a, function&& f);
```

Effects: If `!f`, `*this` has no target; otherwise, move-constructs the target of `f` into the target of `*this`, leaving `f` in a valid state with an unspecified value.

```
template<class F> function(F f);
template<class F, class A> function(allocator arg t, const A& a, F f);
template<class F, class A> function(F f, const A& a);
```

Requires: `f` shall be callable for argument types `ArgTypes` and return type `R`. The copy constructor and destructor of `A` shall not throw exceptions.

Postconditions: `!*this` if any of the following hold:

— `f` is a `NULL` function pointer.

— f is a NULL member function pointer.

— F is an instance of the function class template, and !f

Otherwise, *this targets a copy of f or std::move(f) if f is not a pointer to member function, and targets a copy of mem_fn(f) if f is a pointer to member function. [*Note*: implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, for example, where f's target is an object holding only a pointer or reference to an object and a member function pointer. —*end note*]

Throws: shall not throw exceptions when f is a function pointer or a reference_wrapper<T> for some T. Otherwise, may throw bad_alloc or any exception thrown by F's copy or move constructor.

function

Changes to match_results

The class template match_results shall satisfy the requirements [of an allocator-aware container and](#) of a sequence container, as specified in 23.2.3, except that only operations defined for const-qualified sequence containers are supported.

28.10.5 match_results allocator [re.results.all]

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor [or, if that allocator has been replaced, a copy of the most recent replacement](#).

Interaction with N2913

```
allocator_traits<Alloc>::void_pointer
```

```
allocator_traits<Alloc>::void_pointer  
allocator_traits<Alloc>::const_void_pointer
```

Acknowledgements

⊗

References

⊗

⊗

 ⊗
