The disjoint qualifier

Eric Lengyel, December 4, 2019

lengyel@terathon.com

This proposal introduces a new type qualifier called `disjoint` that is intended as an alternative to the `restrict` qualifier defined by the C language. Whereas the `restrict` qualifier identifies a property of a pointer to an object, the `disjoint` qualifier identifies a property of an object's storage in much the same way that `const` and `volatile` do. This distinction allows use of the `disjoint` qualifier in many situations where similar use of the `restrict` qualifier would not be possible in idiomatic C++, as described below.

Inverse qualifier

The `disjoint` qualifier works in the opposite way that the `const` and `volatile` qualifiers work. Whereas `const` and `volatile` indicate properties of storage that are *present*, `disjoint` indicates a property of storage that is *absent*. Specifically, the `disjoint` qualifier means that storage does not have the property that it can be aliased. This means that a type is actually *less* qualified when the `disjoint` qualifier is specified than it is otherwise. Because implicit qualification conversions go from less qualified to more qualified, the `disjoint` qualifier can always be removed in the same contexts in which the `const` and `volatile` qualifiers can be added. For example:

```
int *a;
const int *c;
disjoint int *d;
...
const int *p = a;       // OK: const qualifier added
disjoint int *q = a;    // error: disjoint qualifier cannot be added
int *r = c;             // error: const qualifier cannot be removed
int *s = d;             // OK: disjoint qualifier can be removed
```

To be consistent with `const` and `volatile`, it would be necessary to introduce a qualifier possibly named `alias` that must be specified for every object that could be referenced through multiple paths. Of course, this would be horribly impractical due to it breaking backwards compatibility with almost all existing code. The only choice is to introduce the `disjoint` qualifier with inverse semantics.

Note that a `noalias` qualifier was proposed by the X3J11 committee in the 1980s at the same time that the `const` and `volatile` qualifiers were introduced to the C language. The `noalias` qualifier had the same intended meaning as the `disjoint` qualifier, but it still worked by making a type *more* qualified. This would have allowed the `noalias` qualifier to be implicitly added to a type at any time, which is technically unsound. Because the `disjoint` qualifier makes a type *less* qualified, it does not suffer from the same problems that prevented `noalias` from ever seeing the light of day.

Type safety

The `restrict` qualifier can be implicitly added to a pointer at any time. Consider the following code:

```
void f(int *restrict a, int *restrict b)
{
    // Storage referenced by a and b assumed disjoint here
}

int *p, *q;
...
f(p, q);        // OK
```

The pointers `a` and `b` are implicitly made restricted through the call to the function `f()`, but there is no safety mechanism that requires pointers to truly disjoint storage to be passed in through the arguments `p` and `q`. Consider the case when `f()` belongs to some external API, and the `restrict` qualifiers are added to the function declaration during a version update. When the calling code is recompiled, no error occurs even though the requirements on the arguments have changed substantially. This can cause the sudden appearance of devastating and difficult-to-find bugs.

Now consider code using the `disjoint` qualifier:

```
void f(disjoint int *a, disjoint int *b)
{
    // Storage referenced by a and b assumed disjoint here
}

int *p, *q;
...
f(p, q);          // error: p and q must point to disjoint objects
```

In this case, pointers to non-disjoint storage cannot be passed to the function `f()` because the `disjoint` qualifier cannot be implicitly added. The `disjoint` qualifier makes good use of the type system to provide the necessary safety. Had the `disjoint` qualifiers in the declaration of `f()` been added during an API version update, then the calling code would suddenly fail to compile, highlighting the exact cause of the problem.

Function overloading

Because the `restrict` qualifier applies to a pointer (or reference), and not to the storage it refers to, functions cannot be overloaded in a such way that aliasing and non-aliasing versions can both be provided. For example, consider this code:

```
void Multiply(const Matrix *m1, const Matrix *m2, Matrix *result);
void Multiply(const Matrix *m1, const Matrix *m2, Matrix *restrict result);
```

The second function declaration is equivalent to the first, and it is not a separate overload. The `disjoint` qualifier, however, allows proper overloading because it applies to the actual storage:

```
void Multiply(const Matrix *m1, const Matrix *m2, Matrix *result);
{
    // Possible aliasing between result and m1 or m2 must be assumed here
}

void Multiply(const Matrix *m1, const Matrix *m2, disjoint Matrix *result);
{
    // The compiler can assume that result does not alias m1 or m2 here
}
```

Through this mechanism, a program can supply one version of a function that must assume that aliasing can occur and a second version of a function that can assume no aliasing and thus achieve greater optimization. Templates could be used to generate both versions of the function from the same source code, relying on the compiler to generate better object code in the disjoint case. The second version of the function would be selected only when the caller explicitly passes a disjoint object to it. It could not be selected by accident.

Non-static member functions

The `disjoint` qualifier can be applied to a non-static member function just as `const` and `volatile` can. This has the effect of making the object `*this` disjoint inside the body of that member function. The `restrict` qualifier is inconsistent in this regard because it applied to the pointer value `this`, as illustrated in the following code.

```
struct X {
    void f();
    void f() const;          // const is applied to *this
    void f() volatile;       // volatile is applied to *this
    void f() disjoint;       // disjoint is applied to *this
    void f() restrict;       // restrict is applied to this, which is inconsistent
                             // error: cannot overload f() with f() restrict
};
```

Furthermore, member functions cannot be overloaded by adding the `restrict` qualifier as they can with the `const` and `volatile` qualifiers. The ability to overload member functions by adding the `disjoint` qualifier makes it possible for a different member function to be selected when the object for which it is invoked is a disjoint object.

Disjoint from birth

Because the `disjoint` qualifier cannot be added to a type, an existing non-disjoint object can never be made disjoint. An object can be disjoint only if it has *always* been disjoint, starting at the beginning of its lifetime. Objects with static, thread, or automatic storage duration must be explicitly declared `disjoint` if they are to ever enjoy the benefit of the disjoint qualification. For example, the possibly more optimal version of the `Multiply()` function described under "Function overloading" above could be invoked using the following code:

```
Matrix a, b;
disjoint Matrix product;
...
Multiply(&a, &b, &product);
```

As described below, all objects with dynamic storage duration begin their lives as disjoint objects.

Heap allocation

Newly allocated memory is always disjoint. The `new` operator is modified so that it returns a pointer to an object (or array thereof) having the disjoint-qualified version of the type to which it is applied. Because the `disjoint` qualifier can always be implicitly removed by a qualification conversion, backwards compatibility is guaranteed. For example:

```
disjoint T *object = new T();  // OK. The new expression has type disjoint T *
T *object = new T();           // Also OK. The disjoint qualifier can be implicitly removed
```

The return type of overloaded `new` operators remains "pointer to `void`" and is not changed to "pointer to `disjoint void`" so that backwards compatibility is maintained. The `disjoint` qualifier is added automatically after allocation and before object construction.

The C memory allocation functions (such as `malloc()`) are modified so that they return "pointer to `disjoint void`".

Constructors and destructors

The `disjoint` qualifier is always applied to an object under construction or destruction. It makes sense to do this because at the time when construction begins, the object couldn't possibly be referenced through any path other than the `this` pointer. Thus, the type of `this` can safely and correctly be qualified as `disjoint`. Similarly, no other valid reference to an object under destruction can possibly exist, so such an object can always be considered disjoint. This is illustrated by the following code:

```
struct X
{
    X()
    {
        // Inside the body of the constructor, *this has type disjoint X
    }

    ~X()
    {
        // Inside the body of the destructor, *this has type disjoint X
    }
};
```

The implicit addition of the `disjoint` qualifier to objects under construction and destruction is similar to the implicit removal of the `const` qualifier that already happens at the same time.

Temporary objects

Temporary objects are always disjoint because they cannot initially be accessed through multiple paths. For example:

```
struct X {...}

void f(X& a);
void f(disjoint X& a);

X x = X();
f(x);           // Calls f(X&)
f(X());         // Calls f(disjoint X&)
```

Overloaded operators that would often return temporary objects should add the disjoint to their return value. For example:

```
disjoint String operator +(const String& a, const String& b);
```

Literals

All literal values are automatically disjoint. This allows initializations such as the following.

```
const disjoint char text[] = "string";
```

This also allows a literal value to be passed by reference to a function expecting a disjoint-qualified argument.

Modifications to the standard

The following are the changes that would need to be made to the C++17 standard to incorporate the `disjoint` qualifier. This is not an exhaustive list, but it aims to highlight the most important and consequential changes. In particular, there are many instances that have been omitted where "*cv*" would simply need to be changed to "*cvd*".

5.11 Keywords [lex.key]

...

Table 5 — Keywords

alignas	continue	for	public	throw
alignof	decltype	friend	register	true
asm	default	goto	reinterpret_cast	try
auto	delete	if	return	typedef
bool	disjoint	inline	short	typeid
break	do	int	signed	typename
case	double	long	sizeof	union
catch	dynamic_cast	mutable	static	unsigned
char	else	namespace	static_assert	using
char16_t	enum	new	static_cast	virtual
char32_t	explicit	noexcept	struct	void
class	export	nullptr	switch	volatile
const	extern	operator	template	wchar_t
constexpr	false	private	this	while
const_cast	float	protected	thread_local	

5.13.2 Integer literals [lex.icon]

...

2 The type of an integer literal is the disjoint-qualified version of the first of the corresponding list in Table 7 in which its value can be represented.

...

5.13.3 Character literals [lex.ccon]

...

2 A character literal that does not begin with `u8`, `u`, `U`, or `L` is an *ordinary character literal*. An ordinary character literal that contains a single *c-char* representable in the execution character set has type `disjoint char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal, or an ordinary character literal containing a single *c-char* not representable in the execution character set, is conditionally-supported, has type `disjoint int`, and has an implementation-defined value.

3 A character literal that begins with `u8`, such as `u8'w'`, is a character literal of type `disjoint char`, known as a *UTF-8 character literal*. The value of a UTF-8 character literal is equal to its ISO 10646 code point value, provided that the code point value is representable with a single UTF-8 code unit (that is, provided it is in the C0 Controls and Basic Latin Unicode block). If the value is not representable with a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing multiple *c-chars* is ill-formed.

4 A character literal that begins with the letter `u`, such as `u'x'`, is a character literal of type `disjoint char16_t`. The value of a `char16_t` character literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A `char16_t` character literal containing multiple *c-chars* is ill-formed.

5 A character literal that begins with the letter `U`, such as `U'y'`, is a character literal of type `disjoint char32_t`. The value of a `char32_t` character literal containing a single *c-char* is equal to its ISO 10646 code point value. A `char32_t` character literal containing multiple *c-chars* is ill-formed.

6 A character literal that begins with the letter `L`, such as `L'z'`, is a *wide-character literal*. A wide-character literal has type `disjoint wchar_t`.[24] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined. [*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set (see 6.9.1). — *end note*] The value of a wide-character literal containing multiple *c-chars* is implementation-defined.

5.13.4 Floating literals [lex.fcon]

...

1 ... The type of a floating literal is `disjoint double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `disjoint float`, the suffixes `l` and `L` specify `disjoint long double`. ...

5.13.5 String literals [lex.string]

...

8 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type "array of *n* `const disjoint char`", where n is the size of the string as defined below, and has static storage duration (6.7).

...

10 A *string-literal* that begins with `u`, such as `u"asdf"`, is a `char16_t` string literal. A `char16_t` string literal has type "array of *n* `const disjoint char16_t`", where *n* is the size of the string as defined below; it is initialized with the given characters. A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

11 A *string-literal* that begins with `U`, such as `U"asdf"`, is a `char32_t` string literal. A `char32_t` string literal has type "array of *n* `const disjoint char32_t`", where *n* is the size of the string as defined below; it is initialized with the given characters.

[12] A *string-literal* that begins with `L`, such as `L"asdf"`, is a *wide string literal*. A wide string literal has type "array of *n* `const disjoint wchar_t`", where *n* is the size of the string as defined below; it is initialized with the given characters.

...

5.13.6 Boolean literals [lex.bool]

...

[1] The Boolean literals are the keywords `false` and `true`. Such literals are prvalues and have type `disjoint bool`.

5.13.7 Pointer literals [lex.nullptr]

...

[1] The pointer literal is the keyword `nullptr`. It is a prvalue of type `std::nullptr_t`. [*Note*: `std::nullptr_t` is a distinct type that is neither a pointer type nor a pointer to member type; rather, a prvalue of this type is a disjoint null pointer constant and can be converted to a null pointer value or null member pointer value. See 7.11 and 7.12. — *end note*]

6.9.3 CVD-qualifiers [basic.type.qualifier]

[1] A type mentioned in 6.9.1 and 6.9.2 is a *cvd-unqualified type*. Each type which is a cvd-unqualified complete or incomplete object type or is `void` (6.9) has ~~three~~seven corresponding cvd-qualified versions of its type: a *const-qualified* version, a *volatile-qualified* version, ~~and~~ a *const-volatile-qualified* version, a *disjoint-qualified* version, a *const-disjoint-qualified* version, a *volatile-disjoint-qualified* version, and a *const-volatile-disjoint-qualified* version. The type of an object (4.5) includes the *cvd-qualifier*s specified in the *decl-specifier-seq* (10.1), *declarator* (Clause 11), *type-id* (11.1), or *new-type-id* (8.3.4) when the object is created.

— A *const object* is an object of type `const T` or a non-mutable subobject of such an object.

— A *volatile object* is an object of type `volatile T`, a subobject of such an object, or a mutable subobject of a const volatile object.

— A *const volatile object* is an object of type `const volatile T`, a non-mutable subobject of such an object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object.

— A *disjoint object* is an object of type `disjoint T` or a subobject of such an object.

— A *const disjoint object* is an object of type `const disjoint T`, a non-mutable subobject of such an object, a const subobject of a disjoint object, or a disjoint subobject of a const object.

— A *volatile disjoint object* is an object of type `volatile disjoint T`, a subobject of such an object, a volatile subobject of a disjoint object, a disjoint subobject of a volatile object, or a mutable subobject of a const volatile disjoint object.

— A *const volatile disjoint object* is an object of type `const volatile disjoint T`, a non-mutable subobject of such an object, a const subobject of a volatile disjoint object, a non-mutable volatile subobject of a const disjoint object, a disjoint subobject of a const volatile object, a const volatile subobject of a disjoint object, a const disjoint subobject of a volatile object, or a non-mutable volatile disjoint subobject of a const object.

The cvd-qualified or cvd-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (6.11).

[2] A compound type (6.9.2) is not cvd-qualified by the cvd-qualifiers (if any) of the types from which it is compounded. Any cvd-qualifiers applied to an array type affect the array element type (11.3.4).

[3] See 11.3.5 and 12.2.2.1 regarding function types that have *cvd-qualifier*s.

[4] There is a partial ordering on cvd-qualifiers, so that a type can be said to be *more cvd-qualified* than another. Table 10 shows the relations that constitute this ordering. [*Note:* The presence of the `const` or `volatile` qualifier causes a

type to be more cvd-qualified, but in the opposite sense, the absence of the `disjoint` qualifier causes a type to be more cvd-qualified. — *end note*]

5 In this International Standard, the notation *cvd* (or *cvd1*, *cvd2*, etc.), used in the description of types, represents an arbitrary set of cvd-qualifiers, i.e., one of {`const`}, {`volatile`}, {`const, volatile`}, {`disjoint`}, {`const, disjoint`}, {`volatile, disjoint`}, {`const, volatile, disjoint`}, or the empty set. For a type *cvd* `T`, the *top-level cvd-qualifiers* of that type are those denoted by *cvd*. [*Example:* The type corresponding to the *type-id* `const int&` has no top-level cvd-qualifiers. The type corresponding to the *type-id* `volatile int * const` has the top-level cvd-qualifier `const`. For a class type `C`, the type corresponding to the *type-id* `void (C::* volatile)(int) const` has the top-level cvd-qualifier `volatile`. — *end example*]

6 Cvd-qualifiers applied to an array type attach to the underlying element type, so the notation "*cvd* `T`", where `T` is an array type, refers to an array whose elements are so-qualified. An array type whose elements are cvd-qualified is also considered to have the same cvd-qualifications as its elements. [*Example:*

```
typedef char CA[5];
typedef const char CC;
CC arr1[5] = { 0 };
const CA arr2 = { 0 };
```

The type of both `arr1` and `arr2` is "array of 5 `const char`", and the array type is considered to be const-qualified. — *end example*]

Table 10 — Relations on `const`, ~~and~~ `volatile`, and `disjoint`

no cvd-qualifier	<	`const`
no cvd-qualifier	<	`volatile`
no cvd-qualifier	<	`const volatile`
`const`	<	`const volatile`
`volatile`	<	`const volatile`
`disjoint`	<	*no cvd-qualifier*
`const disjoint`	<	`const`
`volatile disjoint`	<	`volatile`
`const volatile disjoint`	<	`const volatile`

6.10 Lvalues and rvalues [basic.lval]

...

6 Unless otherwise indicated (8.2.2), a prvalue shall always have complete type or the `void` type. A glvalue shall not have type *cvd* `void`. [*Note:* A glvalue may have complete or incomplete non-`void` type. Class and array prvalues can have cvd-qualified types; other prvalues always have ~~cv-unqualified~~ non-const, non-volatile, disjoint types. See Clause 8. — *end note*]

...

7.1 Lvalue-to-rvalue conversion [conv.lval]

1 A glvalue (6.10) of a non-function, non-array type `T` can be converted to a prvalue.[57] If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If `T` is a non-class type, the type of the prvalue is the ~~cv-unqualified~~ non-const, non-volatile version of `disjoint T` (6.9.3). Otherwise, the type of the prvalue is `T`.[58]

...

7.4 Temporary materialization conversion [conv.rval]

1 A prvalue of type `T` can be converted to an xvalue of type `disjoint T`. This conversion initializes a temporary object (15.2) of type `disjoint T` from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object. `T` shall be a complete type. [*Note:* If `T` is a class type (or array thereof), it must have an accessible and non-deleted destructor; see 15.4. — *end note*]

[*Example:*

```
struct X { int n; };
int k = X().n;          // OK, X() prvalue is converted to xvalue
```

— *end example*]

7.5 Qualification conversions [conv.qual]

1 A *cvd-decomposition* of a type `T` is a sequence of cvd_i and P_i such that `T` is

"$cvd_0\ P_0\ cvd_1\ P_1 \cdots cvd_{n-1}\ P_{n-1}\ cvd_n$ `U`" for $n > 0$,

where each cvd_i is a set of cvd-qualifiers (6.9.3), and each P_i is "pointer to" (11.3.1), "pointer to member of class C_i of type" (11.3.3), "array of N_i", or "array of unknown bound of" (11.3.4). If P_i designates an array, the cvd-qualifiers cvd_{i+1} on the element type are also taken as the cvd-qualifiers cvd_i of the array. [*Example:* The type denoted by the *type-id* `const int **` has two cvd-decompositions, taking `U` as "`int`" and as "pointer to `const int`". — *end example*] The n-tuple of cvd-qualifiers after the first one in the longest cvd-decomposition of `T`, that is, $cvd_1, cvd_2, \ldots, cvd_n$, is called the *cvd-qualification signature* of `T`.

2 Two types T_1 and T_2 are similar if they have cvd-decompositions with the same n such that corresponding P_i components are the same and the types denoted by `U` are the same.

3 A prvalue expression of type T_1 can be converted to type T_2 if the following conditions are satisfied, where cvd_i^j denotes the cvd-qualifiers in the cvd-qualification signature of T_j:[60]

- T_1 and T_2 are similar.
- For every $i > 0$, if `const` is in cvd_i^1 then `const` is in cvd_i^2, and similarly for `volatile`.
- For every $i > 0$, if `disjoint` is not in cvd_i^1 then `disjoint` is not in cvd_i^2.
- If the cvd_i^1 and cvd_i^2 are different, then `const` is in every cvd_k^2 for $0 < k < i$ and `disjoint` is not in every cvd_k^2 for $0 < k < i$.

[*Note:* If a program could assign a pointer of type `T**` to a pointer of type `const T**` (that is, if line #1 below were allowed), a program could inadvertently modify a `const` object (as it is done on line #2). For example,

```
int main() {
    const char c = 'c';
    char* pc;
    const char** pcc = &pc;     // #1: not allowed
    *pcc = &c;
    *pc = 'C';                  // #2: modifies a const object
}
```

— *end note*]

4 [*Note:* A prvalue of type "pointer to cvd_1 `T`" can be converted to a prvalue of type "pointer to cvd_2 `T`" if "cvd_2 `T`" is more cvd-qualified than "cvd_1 `T`". A prvalue of type "pointer to member of `X` of type cvd_1 `T`" can be converted to a prvalue of type "pointer to member of `X` of type cvd_2 `T`" if "cvd_2 `T`" is more cvd-qualified than "cvd_1 `T`". — *end note*]

5 [*Note:* Function types (including those used in pointer to member function types) are never cvd-qualified (11.3.5). — *end note*]

8 Expressions [expr]

...

6 If a prvalue initially has the type "*cvd* `T`", where `T` is a cvd-unqualified non-class, non-array type, the type of the expression is adjusted to `disjoint T` prior to any further analysis.

...

8.1.2 This [expr.prim.this]

...

3 Otherwise, if a member-declarator declares a non-static data member (12.2) of a class `X`, the expression `this` is a prvalue of type "pointer to `disjoint X`" within the optional default member initializer (12.2). It shall not appear elsewhere in the member-declarator.

...

8.2.9 Static cast [expr.static.cast]

1 The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The `static_cast` operator shall not cast away constness (8.2.11). The `static_cast` operator shall not inject the `disjoint` qualification.

...

8.3.4 New [expr.new]

1 The *new-expression* attempts to create a~~n~~ disjoint object of the *type-id* (11.1) or *new-type-id* to which it is applied. The type of that object is the *allocated type*. This type shall be a complete object type, but not an abstract class type or array thereof (4.5, 6.9, 13.4). [*Note:* Because references are not objects, references cannot be created by *new-expressions*. — *end note*] [*Note:* The *type-id* may be a ~~cv-qualified~~ const-qualified or volatile-qualified type, in which case the object created by the *new-expression* has a ~~cv-qualified~~ const-qualified or volatile-qualified type. The object created by the *new-expression* always has a disjoint-qualified type regardless of whether the *type-id* is a disjoint-qualified type. — *end note*]

...

8.17 Throwing an exception [expr.throw]

...

2 Evaluating a throw-expression with an operand throws an exception (18.1); the type of the exception object is determined by removing any top-level ~~cv-qualifiers~~ const or volatile qualifiers from the static type of the operand, adding the top-level disjoint qualifier to the static type of the operand, and adjusting the type from "array of `disjoint T`" or function type `disjoint T` to "pointer to `disjoint T`".

...

10.1.7.1 The *cvd-qualifiers* [dcl.type.cvd]

1 There are ~~two~~ three *cvd-qualifiers*, `const`, ~~and~~ `volatile`, and `disjoint`. Each *cvd-qualifier* shall appear at most once in a *cvd-qualifier-seq*. If a *cvd-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* or *member-declarator-list* of the declaration shall not be empty. [*Note:* 6.9.3 and 11.3.5 describe how cv-qualifiers affect object and function types. — *end note*] Redundant cvd-qualifications are ignored. [*Note:* For example, these could be introduced by typedefs. — *end note*]

...

3 A pointer or reference to a ~~cv-qualified~~ const-qualified or volatile-qualified type need not actually point or refer to a ~~cv-qualified~~ const-qualified or volatile-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. A pointer or reference to a non-disjoint-qualified type need not actually point or refer to a non-disjoint-qualified object, but it is treated as if it does; possible aliasing of an object is assumed in a non-disjoint-qualified access path even if the object referenced is a disjoint object for which no aliasing can be assumed through some other access path. [*Note:* Cvd-qualifiers are supported by the type system so that they cannot be subverted without casting (8.2.11). — *end note*]

...

7 The `disjoint` qualifier is a promise to the implementation that an object defined with a disjoint-qualified type is not accessed through multiple paths in such a way that possible aliasing must be assumed. Thus, the implementation may assume that no aliasing with a disjoint-qualified object occurs. If an attempt is made to access a disjoint-qualified object through multiple non-const-qualified access paths, then the behavior is undefined. [*Note:* Such an attempt can be made by duplicating a pointer to a disjoint-qualified object, as would happen if two pointers to the same object were passed to a function expecting pointers to two separate disjoint-qualified objects. — *end note*]

11 Declarators [dcl.decl]

...

4 Declarators have the syntax

...

cvd-qualifier-seq:
    cvd-qualifier cvd-qualifier-seq$_{opt}$
cvd-qualifier:
    `const`
    `volatile`
    `disjoint`

...

11.4.1 In general [dcl.fct.def.general]

...

8 The function-local predefined variable `__func__` is defined as if a definition of the form

```
static const disjoint char __func__[] = "function-name";
```

had been provided, where function-name is an implementation-defined string. ...

...

12.2.2 Non-static member functions [class.mfct.non-static]

...

4 A non-static member function may be declared `const`, `volatile`, ~~or~~ `const volatile`, `disjoint`, `const disjoint`, `volatile disjoint`, or `const volatile disjoint`. These *cvd-qualifiers* affect the type of the `this` pointer (12.2.2.1). They also affect the function type (11.3.5) of the member function; a member function declared `const` is a *const* member function, a member function declared `volatile` is a *volatile* member function, ~~and~~ a member function declared `const volatile` is a *const volatile* member function, a member function declared `disjoint` is a *disjoint* member function, a member function declared `const disjoint` is a *const disjoint* member function, a member function declared `volatile disjoint` is a *volatile disjoint* member function, a member function declared `const volatile disjoint` is a *const volatile disjoint* member function. [*Example:*

```
struct X {
    void g() const;
```

```
        void h() const volatile;
        void j() disjoint;
        void k() const disjoint;
    };
```

`X::g` is a `const` member function, ~~and~~ `X::h` is a `const volatile` member function, `X::j` is a `disjoint` member function, and `X::k` is a `const disjoint` member function. — *end example*]

...

12.2.2.1 The this pointer [class.this]

1 In the body of a non-static (12.2.1) member function, the keyword `this` is a prvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, ~~and~~ if the member function is declared `const volatile`, the type of `this` is `const volatile X*`, if the member function is declared `disjoint`, the type of `this` is `disjoint X*`, if the member function is declared `const disjoint`, the type of `this` is `const disjoint X*`, if the member function is declared `volatile disjoint`, the type of `this` is `volatile disjoint X*`, and if the member function is declared `const volatile disjoint`, the type of `this` is `const volatile disjoint X*`. [*Note:* Thus in a `const` member function, the object for which the function is called is accessed through a `const` access path. — *end note*] [*Example:*

```
    struct s {
        int a;
        int f() const;
        int g() { return a++; }
        int h() const { return a++; }      // error
    };

    int s::f() const { return a; }
```

The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function because `this` is a pointer to `const`; that is, `*this` has `const` type. — *end example*]

2 Similarly, `volatile` semantics (10.1.7.1) apply in `volatile` member functions and `disjoint` semantics apply in `disjoint` member functions when accessing the object and its non-static data members.

3 A cvd-qualified member function can be called on an object-expression (8.2.5) only if the object-expression is as cvd-qualified or less-cvd-qualified than the member function. [*Example:*

```
    void k(s& x, const s& y) {
        x.f();
        x.g();
        y.f();
        y.g();      // error
    }
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-const member function, that is, `s::g()` is less-qualified than the object-expression `y`. — *end example*]

4 Constructors (15.1) and destructors (15.4) shall not be declared `const`, `volatile`, ~~or~~ `const volatile`, `disjoint`, `const disjoint`, `volatile disjoint`, or `const volatile disjoint`. [*Note:* However, these functions can be invoked to create and destroy objects with cvd-qualified types, see 15.1 and 15.4. — *end note*]

12.2.3.1 Static member functions [class.static.mfct]

...

2 [*Note:* A static member function does not have a this pointer (12.2.2.1). — *end note*] A static member function shall not be `virtual`. There shall not be a static and a non-static member function with the same name and the same

parameter types (16.1). A static member function shall not be declared `const`, `volatile`, ~~or~~ `const volatile`, `disjoint`, `const disjoint`, `volatile disjoint`, or `const volatile disjoint`.

15.1 Constructors [class.ctor]

...

3 A constructor can be invoked for a `const`, `volatile`, ~~or~~ `const volatile`, `disjoint`, `const disjoint`, `volatile disjoint`, or `const volatile disjoint` object. `const` and `volatile` semantics (10.1.7.1) are ~~not~~never applied on an object under construction, and `disjoint` semantics are always applied on an object under construction. ~~They~~ The object's actual *cvd*-qualifications come into effect when the constructor for the most derived object (4.5) ends.

...

15.4 Destructors [class.dtor]

...

2 A destructor is used to destroy objects of its class type. The address of a destructor shall not be taken. A destructor can be invoked for a `const`, `volatile`, ~~or~~ `const volatile`, `disjoint`, `const disjoint`, `volatile disjoint`, or `const volatile disjoint` object. `const` and `volatile` semantics (10.1.7.1) are ~~not~~never applied on an object under destruction, and `disjoint` semantics are always applied on an object under destruction. ~~They~~ The object's actual *cvd*-qualifications stop being in effect when the destructor for the most derived object (4.5) starts.

...

15.8.1 Copy/move constructors [class.copy.ctor]

1 A non-template constructor for class `X` is a copy constructor if its first parameter is of type `X&`, `const X&`, `volatile X&`, ~~or~~ `const volatile X&`, `disjoint X&`, `const disjoint X&`, `volatile disjoint X&`, or `const volatile disjoint X&`, and either there are no other parameters or else all other parameters have default arguments (11.3.6). [*Example:* `X::X(const X&)` and `X::X(X&,int=1)` are copy constructors.

```
struct X {
    X(int);
    X(const X&, int = 1);
};
X a(1);                 // calls X(int);
X b(a, 0);              // calls X(const X&, int);
X c = b;                // calls X(const X&, int);
```

— *end example*]

2 A non-template constructor for class `X` is a move constructor if its first parameter is of type `X&&`, `const X&&`, `volatile X&&`, ~~or~~ `const volatile X&&`, `disjoint X&&`, `const disjoint X&&`, `volatile disjoint X&&`, or `const volatile disjoint X&&`, and either there are no other parameters or else all other parameters have default arguments (11.3.6). [*Example:* `Y::Y(Y&&)` is a move constructor.

```
struct Y {
    Y(const Y&);
    Y(Y&&);
};
extern Y f(int);
Y d(f(1));              // calls Y(Y&&)
Y e = d;                // calls Y(const Y&)
```

— *end example*]

...

4 [*Note:* If a class `X` only has a copy constructor with a parameter of type `X&`, an initializer of type ~~`const X` or `volatile X`~~ `X` that is const-qualified or volatile-qualified cannot initialize an object of type (possibly cvd-qualified) `X`. If a class `X` only has a copy constructor with a parameter of type `disjoint X&`, an initializer of type `X` without the disjoint qualification cannot initialize an object of type (possibly cvd-qualified) `X`. [*Example:*

```
struct X {
    X();                    // default constructor
    X(X&);                  // copy constructor with a non-const parameter
};
struct Y {
    Y();                    // default constructor
    Y(disjoint Y&);         // copy constructor with a disjoint parameter
};
const X cx;
Y ndy;
X x = cx;                   // error: X::X(X&) cannot copy cx into x
Y y = ndy;                  // error: Y::Y(disjoint Y&) cannot copy ndy into y
```

— *end example*] — *end note*]

...

15.8.2 Copy/move assignment operator [class.copy.assign]

1 A user-declared copy assignment operator `X::operator=` is a non-static non-template member function of class `X` with exactly one parameter of type `X`, `X&`, `const X&`, `volatile X&`, ~~or~~ `const volatile X&`, `disjoint X&`, `const disjoint X&`, `volatile disjoint X&`, or `const volatile disjoint X&`.[120] [*Note:* An overloaded assignment operator must be declared to have only one parameter; see 16.5.3. — *end note*] [*Note:* More than one form of copy assignment operator may be declared for a class. — *end note*] [*Note:* If a class `X` only has a copy assignment operator with a parameter of type `X&`, an expression of type ~~`const X`~~ `X` that is const-qualified or volatile-qualified cannot be assigned to an object of type `X`. If a class `X` only has a copy assignment operator with a parameter of type `disjoint X&`, an expression of type `X` without the disjoint qualification cannot be assigned to an object of type `X`. [*Example:*

```
struct X {
    X();
    X& operator=(X&);
};
struct Y {
    Y();
    Y& operator=(disjoint Y&);
};
const X cx;
X x;
Y ndy;
Y y;
void f() {
    x = cx;                 // error: X::operator=(X&) cannot assign cx into x
    y = ndy;                // error: Y::operator=(disjoint Y&) cannot assign ndy into y
}
```

— *end example*] — *end note*]

...

16.1 Overloadable declarations [over.load]

...

(3.4) — Parameter declarations that differ only in the presence or absence of `const`, ~~and/or~~ `volatile`, and/or `disjoint` are equivalent. That is, the `const`, ~~and~~ `volatile`, and `disjoint` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [*Example:*

```
typedef const int cInt;

int f(int);
int f(const int); // redeclaration of f(int)
int f(int) { /* ... */ } // definition of f(int)
int f(cInt) { /* ... */ } // error: redefinition of f(int)
```

— *end example*]

Only the `const`, ~~and~~ `volatile`, and `disjoint` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const`, ~~and~~ `volatile`, and `disjoint` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.[123] In particular, for any type `T`, "pointer to `T`", "pointer to `const T`", ~~and~~ "pointer to `volatile T`", and "pointer to `disjoint T`" are considered distinct parameter types, as are "reference to `T`", "reference to `const T`", ~~and~~ "reference to `volatile T`", and "reference to `disjoint T`".

...

21.2.2 Header `<cstdlib>` synopsis [cstdlib.syn]

...

// 23.10.11, C library memory allocation

```
disjoint void* aligned_alloc(size_t alignment, size_t size);
disjoint void* calloc(size_t nmemb, size_t size);
void free(void* ptr);
disjoint void* malloc(size_t size);
disjoint void* realloc(void* ptr, size_t size);
```

...