(7.2) — The expressions ++a, --a, and &a shall be expression-equivalent to a += 1, a -= 1, and addressof(a), respectively.

(7.3) — For every *unary-operator* @ other than & for which the expression @x is well-formed, @a shall also be well-formed and have the same value, effects, and value category as @x. If @x has type bool, so too does @a; if @x has type B(I), then @a has type I.

(7.4) — For every assignment operator @= for which c @= x is well-formed, c @= a shall also be well-formed and shall have the same value and effects as c @= x. The expression c @= a shall be an lvalue referring to c.

(7.5) — For every assignment operator @= for which x @= y is well-formed, a @= b shall also be well-formed and shall have the same effects as x @= y, except that the value that would be stored into x is stored into a. The expression a @= b shall be an lvalue referring to a.

(7.6) — For every non-assignment binary operator @ for which x @ y and y @ x are well-formed, a @ b and b @ a shall also be well-formed and shall have the same value, effects, and value category as x @ y and y @ x, respectively. If x @ y or y @ x has type B(I), then a @ b or b @ a, respectively, has type I; if x @ y or y @ x has type B(I2), then a @ b or b @ a, respectively, has type I2; if x @ y or y @ x has any other type, then a @ b or b @ a, respectively, has that type.

8 An expression E of integer-class type I is contextually convertible to bool as if by bool(E != I(0)).

9 All integer-class types model regular (18.6) and three_way_comparable<strong_ordering> (17.11.4).

10 A value-initialized object of integer-class type has value 0.

11 For every (possibly cv-qualified) integer-class type I, numeric_limits<I> is specialized such that each static data member m has the same value as numeric_limits<B(I)>::m, and each static member function f returns I(numeric_limits<B(I)>::f()).

12 For any two integer-like types I1 and I2, at least one of which is an integer-class type, common_type_t<I1, I2> denotes an integer-class type whose width is not less than that of I1 or I2. If both I1 and I2 are signed-integer-like types, then common_type_t<I1, I2> is also a signed-integer-like type.

13 *is-integer-like*<I> is true if and only if I is an integer-like type. *is-signed-integer-like*<I> is true if and only if I is a signed-integer-like type.

14 Let i be an object of type I. When i is in the domain of both pre- and post-increment, i is said to be *incrementable*. I models weakly_incrementable<I> only if

(14.1) — The expressions ++i and i++ have the same domain.

(14.2) — If i is incrementable, then both ++i and i++ advance i to the next element.

(14.3) — If i is incrementable, then addressof(++i) is equal to addressof(i).

15 *Recommended practice*: The implementaton of an algorithm on a weakly incrementable type should never attempt to pass through the same incrementable value twice; such an algorithm should be a single-pass algorithm.

[*Note 3*: For weakly_incrementable types, a equals b does not imply that ++a equals ++b. (Equality does not guarantee the substitution property or referential transparency.) Such algorithms can be used with istreams as the source of the input data through the istream_iterator class template. — *end note*]

25.3.4.5 Concept incrementable [iterator.concept.inc]

1 The incrementable concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be equality_comparable.

[*Note 1*: This supersedes the annotations on the increment expressions in the definition of weakly_incrementable. — *end note*]

```
template<class I>
  concept incrementable =
    regular<I> &&
    weakly_incrementable<I> &&
    requires(I i) {
      { i++ } -> same_as<I>;
    };
```