(3.2) — Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessing token by itself and not as the first character of the alternative token `<:`.

(3.3) — Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (2.8) is only formed within a `#include` directive (16.2).

[ *Example:*

```
#define R "x"
const char* s = R"y";           // ill-formed raw string, not "x" "y"
```

— *end example* ]

4 [ *Example:* The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` might produce a valid expression (for example, if `foo` were a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. — *end example* ]

5 [ *Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. — *end example* ]

## 2.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[17]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[18] The set of alternative tokens is defined in Table 1.

Table 1 — Alternative tokens

| Alternative | Primary | Alternative | Primary | Alternative | Primary |
|---|---|---|---|---|---|
| `<%` | `{` | `and` | `&&` | `and_eq` | `&=` |
| `%>` | `}` | `bitor` | `\|` | `or_eq` | `\|=` |
| `<:` | `[` | `or` | `\|\|` | `xor_eq` | `^=` |
| `:>` | `]` | `xor` | `^` | `not` | `!` |
| `%:` | `#` | `compl` | `~` | `not_eq` | `!=` |
| `%:%:` | `##` | `bitand` | `&` | | |

## 2.6 Tokens [lex.token]

*token:*
*identifier*
*keyword*
*literal*
*operator*
*punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals,[19] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described

17) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

18) Thus the "stringized" values (16.3.2) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

19) Literals include strings and character and numeric literals.



©ISO/IEC Dxxxx

(3.2) — Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessing token by itself and not as the first character of the alternative token `<:`.

(3.3) — Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (2.8) is only formed within a `#include` directive (16.2).

[ *Example:*

```
#define R "x"
const char* s = R"y";           // ill-formed raw string, not "x" "y"
```

— *end example* ]

4 [ *Example:* The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` might produce a valid expression (for example, if `foo` were a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. — *end example* ]

5 [ *Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. — *end example* ]

## 2.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[17]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[18] The set of alternative tokens is defined in Table 1.

Table 1 — Alternative tokens

| Alternative | Primary | Alternative | Primary | Alternative | Primary |
|---|---|---|---|---|---|
| `<%` | `{` | `and` | `&&` | `and_eq` | `&=` |
| `%>` | `}` | `bitor` | `\|` | `or_eq` | `\|=` |
| `<:` | `[` | `or` | `\|\|` | `xor_eq` | `^=` |
| `:>` | `]` | `xor` | `^` | `not` | `!` |
| `%:` | `#` | `compl` | `~` | `not_eq` | `!=` |
| `%:%:` | `##` | `bitand` | `&` | | |

## 2.6 Tokens [lex.token]

*token:*
*identifier*
*keyword*
*literal*
*operator*
*punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals,[19] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described

17) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

18) Thus the "stringized" values (16.3.2) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

19) Literals include strings and character and numeric literals.

§ 2.6 22

Table 11 — *simple-type-specifier*s and the types they specify

| Specifier(s) | Type |
|---|---|
| *type-name* | the type named |
| *simple-template-id* | the type as defined in 14.2 |
| *template-name* | placeholder for a type to be deduced |
| char | "char" |
| unsigned char | "unsigned char" |
| signed char | "signed char" |
| char16_t | "char16_t" |
| char32_t | "char32_t" |
| bool | "bool" |
| unsigned | "unsigned int" |
| unsigned int | "unsigned int" |
| signed | "int" |
| signed int | "int" |
| int | "int" |
| unsigned short int | "unsigned short int" |
| unsigned short | "unsigned short int" |
| unsigned long int | "unsigned long int" |
| unsigned long | "unsigned long int" |
| unsigned long long int | "unsigned long long int" |
| unsigned long long | "unsigned long long int" |
| signed long int | "long int" |
| signed long | "long int" |
| signed long long int | "long long int" |
| signed long long | "long long int" |
| long long int | "long long int" |
| long long | "long long int" |
| long int | "long int" |
| long | "long int" |
| signed short int | "short int" |
| signed short | "short int" |
| short int | "short int" |
| short | "short int" |
| wchar_t | "wchar_t" |
| float | "float" |
| double | "double" |
| long double | "long double" |
| void | "void" |
| auto | placeholder for a type to be deduced |
| decltype(auto) | placeholder for a type to be deduced |
| decltype(*expression*) | the type as defined below |

[3] When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. [ *Note:* It is implementation-defined whether objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects to be signed; it is redundant in other contexts. — *end note* ]



©ISO/IEC Dxxxx

Table 11 — *simple-type-specifier*s and the types they specify

| **Specifier(s)** | **Type** |
|---|---|
| *type-name* | the type named |
| *simple-template-id* | the type as defined in 14.2 |
| *template-name* | placeholder for a type to be deduced |
| char | "char" |
| unsigned char | "unsigned char" |
| signed char | "signed char" |
| char16_t | "char16_t" |
| char32_t | "char32_t" |
| bool | "bool" |
| unsigned | "unsigned int" |
| unsigned int | "unsigned int" |
| signed | "int" |
| signed int | "int" |
| int | "int" |
| unsigned short int | "unsigned short int" |
| unsigned short | "unsigned short int" |
| unsigned long int | "unsigned long int" |
| unsigned long | "unsigned long int" |
| unsigned long long int | "unsigned long long int" |
| unsigned long long | "unsigned long long int" |
| signed long int | "long int" |
| signed long | "long int" |
| signed long long int | "long long int" |
| signed long long | "long long int" |
| long long int | "long long int" |
| long long | "long long int" |
| long int | "long int" |
| long | "long int" |
| signed short int | "short int" |
| signed short | "short int" |
| short int | "short int" |
| short | "short int" |
| wchar_t | "wchar_t" |
| float | "float" |
| double | "double" |
| long double | "long double" |
| void | "void" |
| auto | placeholder for a type to be deduced |
| decltype(auto) | placeholder for a type to be deduced |
| decltype(*expression*) | the type as defined below |

[3] When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. [ *Note:* It is implementation-defined whether objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects to be signed; it is redundant in other contexts. — *end note* ]

### 17.5.3 Requirements on types and expressions [utility.requirements]

1 17.5.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 17.5.3.2 describes the requirements on swappable types and swappable expressions. 17.5.3.3 describes the requirements on pointer-like types that support null values. 17.5.3.4 describes the requirements on hash function objects. 17.5.3.5 describes the requirements on storage allocators.

#### 17.5.3.1 Template argument requirements [utility.arg.requirements]

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in Tables 20–27. In these tables, `T` is an object or reference type to be supplied by a C++ program instantiating a template; `a`, `b`, and `c` are values of type (possibly `const`) `T`; `s` and `t` are modifiable lvalues of type `T`; `u` denotes an identifier; `rv` is an rvalue of type `T`; and `v` is an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`.

2 In general, a default constructor is not required. Certain container class member function signatures specify `T()` as a default argument. `T()` shall be a well-defined expression (8.6) if one of those signatures is called using the default argument (8.3.6).

Table 20 — `EqualityComparable` requirements [**equalitycomparable**]

| Expression | Return type | Requirement |
|---|---|---|
| `a == b` | convertible to `bool` | `==` is an equivalence relation, that is, it has the following properties:<br>— For all `a`, `a == a`.<br>— If `a == b`, then `b == a`.<br>— If `a == b` and `b == c`, then `a == c`. |

Table 21 — `LessThanComparable` requirements [**lessthancomparable**]

| Expression | Return type | Requirement |
|---|---|---|
| `a < b` | convertible to `bool` | `<` is a strict weak ordering relation (25.7) |

Table 22 — `DefaultConstructible` requirements [**defaultconstructible**]

| Expression | Post-condition |
|---|---|
| `T t;` | object `t` is default-initialized |
| `T u{};` | object `u` is value-initialized or aggregate-initialized |
| `T()`<br>`T{}` | an object of type `T` is value-initialized or aggregate-initialized |

#### 17.5.3.2 `Swappable` requirements [swappable.requirements]

1 This subclause provides definitions for swappable types and expressions. In these definitions, let `t` denote an expression of type `T`, and let `u` denote an expression of type `U`.



©ISO/IEC Dxxxx

17.5.3 Requirements on types and expressions [utility.requirements]

1 17.5.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 17.5.3.2 describes the requirements on swappable types and swappable expressions. 17.5.3.3 describes the requirements on pointer-like types that support null values. 17.5.3.4 describes the requirements on hash function objects. 17.5.3.5 describes the requirements on storage allocators.

17.5.3.1 Template argument requirements [utility.arg.requirements]

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in Tables 20–27. In these tables, `T` is an object or reference type to be supplied by a C++ program instantiating a template; `a`, `b`, and `c` are values of type (possibly `const`) `T`; `s` and `t` are modifiable lvalues of type `T`; `u` denotes an identifier; `rv` is an rvalue of type `T`; and `v` is an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`.

2 In general, a default constructor is not required. Certain container class member function signatures specify `T()` as a default argument. `T()` shall be a well-defined expression (8.6) if one of those signatures is called using the default argument (8.3.6).

Table 20 — `EqualityComparable` requirements [**equalitycomparable**]

| Expression | Return type | Requirement |
|---|---|---|
| `a == b` | convertible to `bool` | `==` is an equivalence relation, that is, it has the following properties:<br>— For all `a`, `a == a`.<br>— If `a == b`, then `b == a`.<br>— If `a == b` and `b == c`, then `a == c`. |

Table 21 — `LessThanComparable` requirements [**lessthancomparable**]

| Expression | Return type | Requirement |
|---|---|---|
| `a < b` | convertible to `bool` | `<` is a strict weak ordering relation (25.7) |

Table 22 — `DefaultConstructible` requirements [**defaultconstructible**]

| Expression | Post-condition |
|---|---|
| `T t;` | object `t` is default-initialized |
| `T u{};` | object `u` is value-initialized or aggregate-initialized |
| `T()`<br>`T{}` | an object of type `T` is value-initialized or aggregate-initialized |

17.5.3.2 `Swappable` requirements [swappable.requirements]

1 This subclause provides definitions for swappable types and expressions. In these definitions, let `t` denote an expression of type `T`, and let `u` denote an expression of type `U`.

§ 17.5.3.2 456

Table 23 — `MoveConstructible` requirements **[moveconstructible]**

| Expression | Post-condition |
|---|---|
| `T u = rv;` | `u` is equivalent to the value of `rv` before the construction |
| `T(rv)` | `T(rv)` is equivalent to the value of `rv` before the construction |
| `rv`'s state is unspecified [*Note:* `rv` must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*] | |

Table 24 — `CopyConstructible` requirements (in addition to `MoveConstructible`) **[copyconstructible]**

| Expression | Post-condition |
|---|---|
| `T u = v;` | the value of `v` is unchanged and is equivalent to `u` |
| `T(v)` | the value of `v` is unchanged and is equivalent to `T(v)` |

Table 25 — `MoveAssignable` requirements **[moveassignable]**

| Expression | Return type | Return value | Post-condition |
|---|---|---|---|
| `t = rv` | `T&` | `t` | If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment |
| `rv`'s state is unspecified. [*Note:* `rv` must still meet the requirements of the library component that is using it, whether or not `t` and `rv` refer to the same object. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*] | | | |

Table 26 — `CopyAssignable` requirements (in addition to `MoveAssignable`) **[copyassignable]**

| Expression | Return type | Return value | Post-condition |
|---|---|---|---|
| `t = v` | `T&` | `t` | `t` is equivalent to `v`, the value of `v` is unchanged |

Table 27 — `Destructible` requirements **[destructible]**

| Expression | Post-condition |
|---|---|
| `u.~T()` | All resources owned by `u` are reclaimed, no exception is propagated. |

[2] An object `t` is *swappable with* an object `u` if and only if:

(2.1) — the expressions `swap(t, u)` and `swap(u, t)` are valid when evaluated in the context described below, and

(2.2) — these expressions have the following effects:

(2.2.1) — the object referred to by `t` has the value originally held by `u` and

(2.2.2) — the object referred to by `u` has the value originally held by `t`.



©ISO/IEC Dxxxx

Table 23 — `MoveConstructible` requirements **[moveconstructible]**

| Expression | Post-condition |
|---|---|
| `T u = rv;` | `u` is equivalent to the value of `rv` before the construction |
| `T(rv)` | `T(rv)` is equivalent to the value of `rv` before the construction |
| `rv`'s state is unspecified [*Note:* `rv` must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*] | |

Table 24 — `CopyConstructible` requirements (in addition to `MoveConstructible`) **[copyconstructible]**

| Expression | Post-condition |
|---|---|
| `T u = v;` | the value of `v` is unchanged and is equivalent to `u` |
| `T(v)` | the value of `v` is unchanged and is equivalent to `T(v)` |

Table 25 — `MoveAssignable` requirements **[moveassignable]**

| Expression | Return type | Return value | Post-condition |
|---|---|---|---|
| `t = rv` | `T&` | `t` | If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment |
| `rv`'s state is unspecified. [*Note:* `rv` must still meet the requirements of the library component that is using it, whether or not `t` and `rv` refer to the same object. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*] | | | |

Table 26 — `CopyAssignable` requirements (in addition to `MoveAssignable`) **[copyassignable]**

| Expression | Return type | Return value | Post-condition |
|---|---|---|---|
| `t = v` | `T&` | `t` | `t` is equivalent to `v`, the value of `v` is unchanged |

Table 27 — `Destructible` requirements **[destructible]**

| Expression | Post-condition |
|---|---|
| `u.~T()` | All resources owned by `u` are reclaimed, no exception is propagated. |

[2] An object `t` is *swappable with* an object `u` if and only if:

(2.1) — the expressions `swap(t, u)` and `swap(u, t)` are valid when evaluated in the context described below, and

(2.2) — these expressions have the following effects:

(2.2.1) — the object referred to by `t` has the value originally held by `u` and

(2.2.2) — the object referred to by `u` has the value originally held by `t`.

27 [ *Note:* The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with `operator==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. — *end note* ]

28 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 140 — Atomic arithmetic computations

| Key | Op | Computation | Key | Op | Computation |
|---|---|---|---|---|---|
| `add` | `+` | addition | `sub` | `-` | subtraction |
| `or` | `\|` | bitwise inclusive or | `xor` | `^` | bitwise exclusive or |
| `and` | `&` | bitwise and | | | |

```
C atomic_fetch_key(volatile A* object, M operand) noexcept;
C atomic_fetch_key(A* object, M operand) noexcept;
C atomic_fetch_key_explicit(volatile A* object, M operand, memory_order order) noexcept;
C atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;
```

29 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the result of the computation applied to the value pointed to by `object` or by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

30 *Returns:* Atomically, the value pointed to by `object` or by `this` immediately before the effects.

31 *Remarks:* For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile noexcept;
C A::operator op=(M operand) noexcept;
```

32 *Effects:* As if by `fetch_key(operand)`.

33 *Returns:* `fetch_key(operand) op operand`.

```
C A::operator++(int) volatile noexcept;
C A::operator++(int) noexcept;
```

34 *Returns:* `fetch_add(1)`.

```
C A::operator--(int) volatile noexcept;
C A::operator--(int) noexcept;
```

35 *Returns:* `fetch_sub(1)`.

```
C A::operator++() volatile noexcept;
C A::operator++() noexcept;
```

36 *Effects:* As if by `fetch_add(1)`.

37 *Returns:* `fetch_add(1) + 1`.

```
C A::operator--() volatile noexcept;
C A::operator--() noexcept;
```



©ISO/IEC Dxxxx

27 [ *Note:* The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with `operator==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. — *end note* ]

28 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 140 — Atomic arithmetic computations

| *key* | Op | Computation | *key* | Op | Computation |
|---|---|---|---|---|---|
| `add` | `+` | addition | `sub` | `-` | subtraction |
| `or` | `\|` | bitwise inclusive or | `xor` | `^` | bitwise exclusive or |
| `and` | `&` | bitwise and | | | |

```
C atomic_fetch_key(volatile A* object, M operand) noexcept;
C atomic_fetch_key(A* object, M operand) noexcept;
C atomic_fetch_key_explicit(volatile A* object, M operand, memory_order order) noexcept;
C atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;
```

29 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the result of the computation applied to the value pointed to by `object` or by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

30 *Returns:* Atomically, the value pointed to by `object` or by `this` immediately before the effects.

31 *Remarks:* For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile noexcept;
C A::operator op=(M operand) noexcept;
```

32 *Effects:* As if by `fetch_key(operand)`.

33 *Returns:* `fetch_key(operand) op operand`.

```
C A::operator++(int) volatile noexcept;
C A::operator++(int) noexcept;
```

34 *Returns:* `fetch_add(1)`.

```
C A::operator--(int) volatile noexcept;
C A::operator--(int) noexcept;
```

35 *Returns:* `fetch_sub(1)`.

```
C A::operator++() volatile noexcept;
C A::operator++() noexcept;
```

36 *Effects:* As if by `fetch_add(1)`.

37 *Returns:* `fetch_add(1) + 1`.

```
C A::operator--() volatile noexcept;
C A::operator--() noexcept;
```

§ 29.6.5 1335