8.4.1 The `if` statement [stmt.if]

1 If the condition (8.4) yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. If the first substatement is reached via a label, the condition is not evaluated and the second substatement is not executed. In the second form of `if` statement (the one including `else`), if the first substatement is also an `if` statement then that inner `if` statement shall contain an `else` part.[84]

2 If the `if` statement is of the form `if constexpr`, the value of the condition shall be a contextually converted constant expression of type `bool` (7.7); this form is called a *constexpr if* statement. If the value of the converted condition is `false`, the first substatement is a *discarded statement*, otherwise the second substatement, if present, is a discarded statement. During the instantiation of an enclosing templated entity (Clause 13), if the condition is not value-dependent after its instantiation, the discarded substatement (if any) is not instantiated. [*Note*: Odr-uses (6.2) in a discarded statement do not require an entity to be defined. — *end note*] A `case` or `default` label appearing within such an `if` statement shall be associated with a `switch` statement (8.4.2) within the same `if` statement. A label (8.1) declared in a substatement of a constexpr if statement shall only be referred to by a statement (8.6.5) in the same substatement. [*Example*:

```
template<typename T, typename ... Rest> void g(T&& p, Rest&& ...rs) {
  // ... handle p

  if constexpr (sizeof...(rs) > 0)
    g(rs...);         // never instantiated with an empty argument list
}

extern int x;         // no definition of x required

int f() {
  if constexpr (true)
    return 0;
  else if (x)
    return x;
  else
    return -x;
}
```

— *end example*]

3 An `if` statement of the form

`if constexpr`$_{opt}$ `(` *init-statement condition* `)` *statement*

is equivalent to

```
{
    init-statement
    if constexpr(opt) ( condition ) statement
}
```

and an `if` statement of the form

`if constexpr`$_{opt}$ `(` *init-statement condition* `)` *statement* `else` *statement*

is equivalent to

```
{
    init-statement
    if constexpr(opt) ( condition ) statement else statement
}
```

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*.

8.4.2 The `switch` statement [stmt.switch]

1 The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (7.3) to an integral or enumeration type. If the (possibly converted) type

84) In other words, the `else` is associated with the nearest un-elsed `if`.

©ISO/IEC Dxxxx

8.4.1 The if statement [stmt.if]

1 If the condition (8.4) yields true the first substatement is executed. If the else part of the selection statement is present and the condition yields false, the second substatement is executed. If the first substatement is reached via a label, the condition is not evaluated and the second substatement is not executed. In the second form of if statement (the one including else), if the first substatement is also an if statement then that inner if statement shall contain an else part.[84]

2 If the if statement is of the form if constexpr, the value of the condition shall be a contextually converted constant expression of type bool (7.7); this form is called a *constexpr if* statement. If the value of the converted condition is false, the first substatement is a *discarded statement*, otherwise the second substatement, if present, is a discarded statement. During the instantiation of an enclosing templated entity (Clause 13), if the condition is not value-dependent after its instantiation, the discarded substatement (if any) is not instantiated. [*Note*: Odr-uses (6.2) in a discarded statement do not require an entity to be defined. — *end note*] A case or default label appearing within such an if statement shall be associated with a switch statement (8.4.2) within the same if statement. A label (8.1) declared in a substatement of a constexpr if statement shall only be referred to by a statement (8.6.5) in the same substatement. [*Example*:

```
template<typename T, typename ... Rest> void g(T&& p, Rest&& ...rs) {
  // ... handle p

  if constexpr (sizeof...(rs) > 0)
    g(rs...);         // never instantiated with an empty argument list
}

extern int x;         // no definition of x required

int f() {
  if constexpr (true)
    return 0;
  else if (x)
    return x;
  else
    return -x;
}
```

— *end example*]

3 An if statement of the form

if constexpr$_{opt}$ (*init-statement condition*) *statement*

is equivalent to

```
{
    init-statement
    if constexpr(opt) ( condition ) statement
}
```

and an if statement of the form

if constexpr$_{opt}$ (*init-statement condition*) *statement* else *statement*

is equivalent to

```
{
    init-statement
    if constexpr(opt) ( condition ) statement else statement
}
```

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*.

8.4.2 The switch statement [stmt.switch]

1 The switch statement causes control to be transferred to one of several statements depending on the value of a condition.

84) In other words, the else is associated with the nearest un-elsed if.

§ 8.4.2 144

is subject to integral promotions (7.3.6), the condition is converted to the promoted type. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* :

where the *constant-expression* shall be a converted constant expression (7.7) of the adjusted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion.

3 There shall be at most one label of the form

`default` :

within a `switch` statement.

4 Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

5 When the `switch` statement is executed, its condition is evaluated. If one of the case constants has the same value as the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

6 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 8.6.1. [*Note*: Usually, the substatement that is the subject of a switch is compound and `case` and `default` labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a `switch` statement. — *end note*]

7 A `switch` statement of the form

`switch` (*init-statement condition*) *statement*

is equivalent to

```
{
    init-statement
    switch ( condition ) statement
}
```

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*.

8.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

iteration-statement:
`while` (*condition*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*init-statement* $condition_{opt}$; $expression_{opt}$) *statement*
`for` ($init\text{-}statement_{opt}$ *for-range-declaration* : *for-range-initializer*) *statement*

for-range-declaration:
$attribute\text{-}specifier\text{-}seq_{opt}$ *decl-specifier-seq declarator*
$attribute\text{-}specifier\text{-}seq_{opt}$ *decl-specifier-seq* $ref\text{-}qualifier_{opt}$ `[` *identifier-list* `]`

for-range-initializer:
expr-or-braced-init-list

See 9.2.3 for the optional *attribute-specifier-seq* in a *for-range-declaration*. [*Note*: An *init-statement* ends with a semicolon. — *end note*]

2 The substatement in an *iteration-statement* implicitly defines a block scope (6.3) which is entered and exited each time through the loop. If the substatement in an *iteration-statement* is a single statement and not a *compound-statement*, it is as if it was rewritten to be a *compound-statement* containing the original statement. [*Example*:

```
while (--x >= 0)
  int i;
```

can be equivalently rewritten as

```
while (--x >= 0) {
  int i;
}
```


2 The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (7.3) to an integral or enumeration type. If the (possibly converted) type is subject to integral promotions (7.3.6), the condition is converted to the promoted type. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* :

where the *constant-expression* shall be a converted constant expression (7.7) of the adjusted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion.

3 There shall be at most one label of the form

`default` :

within a `switch` statement.

4 Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

5 When the `switch` statement is executed, its condition is evaluated. If one of the case constants has the same value as the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

6 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 8.6.1. [*Note*: Usually, the substatement that is the subject of a switch is compound and `case` and `default` labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a `switch` statement. — *end note*]

7 A `switch` statement of the form

`switch` (*init-statement condition*) *statement*

is equivalent to

```
{
    init-statement
    switch ( condition ) statement
}
```

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*.

8.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

iteration-statement:
`while` (*condition*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*init-statement* $condition_{opt}$; $expression_{opt}$) *statement*
`for` ($init\text{-}statement_{opt}$ *for-range-declaration* : *for-range-initializer*) *statement*

for-range-declaration:
$attribute\text{-}specifier\text{-}seq_{opt}$ *decl-specifier-seq declarator*
$attribute\text{-}specifier\text{-}seq_{opt}$ *decl-specifier-seq* $ref\text{-}qualifier_{opt}$ `[` *identifier-list* `]`

for-range-initializer:
expr-or-braced-init-list

See 9.2.3 for the optional *attribute-specifier-seq* in a *for-range-declaration*. [*Note*: An *init-statement* ends with a semicolon. — *end note*]

2 The substatement in an *iteration-statement* implicitly defines a block scope (6.3) which is entered and exited each time through the loop. If the substatement in an *iteration-statement* is a single statement and not a *compound-statement*, it is as if it was rewritten to be a *compound-statement* containing the original statement. [*Example*:

```
while (--x >= 0)
  int i;
```

can be equivalently rewritten as

Thus after the `while` statement, `i` is no longer in scope. — *end example*]

3 If a name introduced in an *init-statement* or *for-range-declaration* is redeclared in the outermost block of the substatement, the program is ill-formed. [*Example*:

```
void f() {
  for (int i = 0; i < 10; ++i)
    int i = 0;          // error: redeclaration
  for (int i : { 1, 2, 3 })
    int i = 1;          // error: redeclaration
}
```

— *end example*]

8.5.1 The `while` statement [stmt.while]

1 In the `while` statement the substatement is executed repeatedly until the value of the condition (8.4) becomes `false`. The test takes place before each execution of the substatement.

2 When the condition of a `while` statement is a declaration, the scope of the variable that is declared extends from its point of declaration (6.3.2) to the end of the `while` *statement*. A `while` statement is equivalent to

```
label :
{
    if ( condition ) {
        statement
        goto label ;
    }
}
```

[*Note*: The variable created in the condition is destroyed and created with each iteration of the loop. [*Example*:

```
struct A {
  int val;
  A(int i) : val(i) { }
  ~A() { }
  operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) {
  // ...
  i = 0;
}
```

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. — *end example*] — *end note*]

8.5.2 The `do` statement [stmt.do]

1 The expression is contextually converted to `bool` (7.3); if that conversion is ill-formed, the program is ill-formed.

2 In the `do` statement the substatement is executed repeatedly until the value of the expression becomes `false`. The test takes place after each execution of the statement.

8.5.3 The `for` statement [stmt.for]

1 The `for` statement

`for` (*init-statement* $condition_{opt}$; $expression_{opt}$) *statement*

is equivalent to

```
{
    init-statement
    while ( condition ) {
        statement
        expression ;
    }
}
```



```
while (--x >= 0) {
  int i;
}
```

Thus after the `while` statement, `i` is no longer in scope. — *end example*]

3 If a name introduced in an *init-statement* or *for-range-declaration* is redeclared in the outermost block of the substatement, the program is ill-formed. [*Example*:

```
void f() {
  for (int i = 0; i < 10; ++i)
    int i = 0;          // error: redeclaration
  for (int i : { 1, 2, 3 })
    int i = 1;          // error: redeclaration
}
```

— *end example*]

8.5.1 The `while` statement [stmt.while]

1 In the `while` statement the substatement is executed repeatedly until the value of the condition (8.4) becomes `false`. The test takes place before each execution of the substatement.

2 When the condition of a `while` statement is a declaration, the scope of the variable that is declared extends from its point of declaration (6.3.2) to the end of the `while` *statement*. A `while` statement is equivalent to

```
label :
{
    if ( condition ) {
        statement
        goto label ;
    }
}
```

[*Note*: The variable created in the condition is destroyed and created with each iteration of the loop. [*Example*:

```
struct A {
  int val;
  A(int i) : val(i) { }
  ~A() { }
  operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) {
  // ...
  i = 0;
}
```

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. — *end example*] — *end note*]

8.5.2 The `do` statement [stmt.do]

1 The expression is contextually converted to `bool` (7.3); if that conversion is ill-formed, the program is ill-formed.

2 In the `do` statement the substatement is executed repeatedly until the value of the expression becomes `false`. The test takes place after each execution of the statement.

8.5.3 The `for` statement [stmt.for]

1 The `for` statement

`for` (*init-statement* $condition_{opt}$; $expression_{opt}$) *statement*

is equivalent to

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*, and except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression* before re-evaluating *condition*. [*Note*: Thus the first statement specifies initialization for the loop; the condition (8.4) specifies a test, sequenced before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is sequenced after each iteration. — *end note*]

2 Either or both of the *condition* and the *expression* can be omitted. A missing *condition* makes the implied `while` clause equivalent to `while(true)`.

3 If the *init-statement* is a declaration, the scope of the name(s) declared extends to the end of the `for` statement. [*Example*:

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
  a[i] = i;

int j = i;          // j = 42
```

— *end example*]

8.5.4 The range-based `for` statement [stmt.ranged]

1 The range-based `for` statement

`for` (*init-statement*$_{opt}$ *for-range-declaration* : *for-range-initializer*) *statement*

is equivalent to

```
{
    init-statement_opt
    auto &&range = for-range-initializer ;
    auto begin = begin-expr ;
    auto end = end-expr ;
    for ( ; begin != end; ++begin ) {
        for-range-declaration = * begin ;
        statement
    }
}
```

where

(1.1) — if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarator*s);

(1.2) — `range`, `begin`, and `end` are variables defined for exposition only; and

(1.3) — *begin-expr* and *end-expr* are determined as follows:

(1.3.1) — if the *for-range-initializer* is an expression of array type `R`, *begin-expr* and *end-expr* are `range` and `range + N`, respectively, where `N` is the array bound. If `R` is an array of unknown bound or an array of incomplete type, the program is ill-formed;

(1.3.2) — if the *for-range-initializer* is an expression of class type `C`, the *unqualified-id*s `begin` and `end` are looked up in the scope of `C` as if by class member access lookup (6.4.5), and if both find at least one declaration, *begin-expr* and *end-expr* are `range.begin()` and `range.end()`, respectively;

(1.3.3) — otherwise, *begin-expr* and *end-expr* are `begin(range)` and `end(range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (6.4.2). [*Note*: Ordinary unqualified lookup (6.4.1) is not performed. — *end note*]

[*Example*:

```
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
  x *= 2;
```

— *end example*]

2 In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`. The *decl-specifier-seq* shall not define a class or enumeration.


```
{
    init-statement
    while ( condition ) {
        statement
        expression ;
    }
}
```

except that names declared in the *init-statement* are in the same declarative region as those declared in the *condition*, and except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression* before re-evaluating *condition*. [*Note*: Thus the first statement specifies initialization for the loop; the condition (8.4) specifies a test, sequenced before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is sequenced after each iteration. — *end note*]

2 Either or both of the *condition* and the *expression* can be omitted. A missing *condition* makes the implied `while` clause equivalent to `while(true)`.

3 If the *init-statement* is a declaration, the scope of the name(s) declared extends to the end of the `for` statement. [*Example*:

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
  a[i] = i;

int j = i;          // j = 42
```

— *end example*]

8.5.4 The range-based `for` statement [stmt.ranged]

1 The range-based `for` statement

`for` (*init-statement*$_{opt}$ *for-range-declaration* : *for-range-initializer*) *statement*

is equivalent to

```
{
    init-statement_opt
    auto &&range = for-range-initializer ;
    auto begin = begin-expr ;
    auto end = end-expr ;
    for ( ; begin != end; ++begin ) {
        for-range-declaration = * begin ;
        statement
    }
}
```

where

(1.1) — if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarator*s);

(1.2) — `range`, `begin`, and `end` are variables defined for exposition only; and

(1.3) — *begin-expr* and *end-expr* are determined as follows:

(1.3.1) — if the *for-range-initializer* is an expression of array type `R`, *begin-expr* and *end-expr* are `range` and `range + N`, respectively, where `N` is the array bound. If `R` is an array of unknown bound or an array of incomplete type, the program is ill-formed;

(1.3.2) — if the *for-range-initializer* is an expression of class type `C`, the *unqualified-id*s `begin` and `end` are looked up in the scope of `C` as if by class member access lookup (6.4.5), and if both find at least one declaration, *begin-expr* and *end-expr* are `range.begin()` and `range.end()`, respectively;

(1.3.3) — otherwise, *begin-expr* and *end-expr* are `begin(range)` and `end(range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (6.4.2). [*Note*: Ordinary unqualified lookup (6.4.1) is not performed. — *end note*]

8.6 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
`break` ;
`continue` ;
`return` *expr-or-braced-init-list*$_{opt}$;
coroutine-return-statement
`goto` *identifier* ;

2 On exit from a scope (however accomplished), objects with automatic storage duration (6.6.5.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note*: For temporaries, see 6.6.7. — *end note*] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 8.7 for transfers into blocks). [*Note*: However, the program can be terminated (by calling `std::exit()` or `std::abort()` (17.5), for example) without destroying objects with automatic storage duration. — *end note*] [*Note*: A suspension of a coroutine (7.6.2.3) is not considered to be an exit from a scope. — *end note*]

8.6.1 The `break` statement [stmt.break]

1 The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

8.6.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {
  {
    // ...
  }
contin: ;
}
```

```
do {
  {
    // ...
  }
contin: ;
} while (foo);
```

```
for (;;) {
  {
    // ...
  }
contin: ;
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto` *contin*.

8.6.3 The `return` statement [stmt.return]

1 A function returns to its caller by the `return` statement.

2 The *expr-or-braced-init-list* of a `return` statement is called its operand. A `return` statement with no operand shall be used only in a function whose return type is *cv* `void`, a constructor (11.3.4), or a destructor (11.3.6). A `return` statement with an operand of type `void` shall be used only in a function whose return type is *cv* `void`. A `return` statement with any other operand shall be used only in a function whose return type is not *cv* `void`; the `return` statement initializes the glvalue result or prvalue result object of the (explicit or implicit) function call by copy-initialization (9.3) from the operand. [*Note*: A `return` statement can involve an invocation of a constructor to perform a copy or move of the operand if it is not a prvalue or if its type differs from the return type of the function. A copy operation associated with a `return` statement may be elided or converted to a move operation if an automatic storage duration variable is returned (11.9.5). — *end note*] [*Example*:

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— *end example*] Flowing off the end of a constructor, a destructor, or a non-coroutine function with a *cv* `void` return type is equivalent to a `return` with no operand. Otherwise, flowing off the end of a function other than `main` (6.8.3.1) or a coroutine (9.4.4) results in undefined behavior.

3 The copy-initialization of the result of the call is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the `return` statement, which, in turn, is sequenced before the destruction of local variables (8.6) of the block enclosing the `return` statement.

[*Example*:

```
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
  x *= 2;
```

— *end example*]

2 In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`. The *decl-specifier-seq* shall not define a class or enumeration.

8.6 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
`break` ;
`continue` ;
`return` *expr-or-braced-init-list*$_{opt}$;
coroutine-return-statement
`goto` *identifier* ;

2 On exit from a scope (however accomplished), objects with automatic storage duration (6.6.5.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note*: For temporaries, see 6.6.7. — *end note*] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 8.7 for transfers into blocks). [*Note*: However, the program can be terminated (by calling `std::exit()` or `std::abort()` (17.5), for example) without destroying objects with automatic storage duration. — *end note*] [*Note*: A suspension of a coroutine (7.6.2.3) is not considered to be an exit from a scope. — *end note*]

8.6.1 The `break` statement [stmt.break]

1 The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

8.6.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {
  {
    // ...
  }
contin: ;
}
```

```
do {
  {
    // ...
  }
contin: ;
} while (foo);
```

```
for (;;) {
  {
    // ...
  }
contin: ;
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto` *contin*.

8.6.3 The `return` statement [stmt.return]

1 A function returns to its caller by the `return` statement.

2 The *expr-or-braced-init-list* of a `return` statement is called its operand. A `return` statement with no operand shall be used only in a function whose return type is *cv* `void`, a constructor (11.3.4), or a destructor (11.3.6). A `return` statement with an operand of type `void` shall be used only in a function whose return type is *cv* `void`. A `return` statement with any other operand shall be used only in a function whose return type is not *cv* `void`; the `return` statement initializes the glvalue result or prvalue result object of the (explicit or implicit) function call by copy-initialization (9.3) from the operand. [*Note*: A `return` statement can involve an invocation of a constructor to perform a copy or move of the operand if it is not a prvalue or if its type differs from the return type of the function. A copy operation associated with a `return` statement may be

8.6.4 The `co_return` statement [stmt.return.coroutine]

coroutine-return-statement:
`co_return` *expr-or-braced-init-list*$_{opt}$ `;`

1 A coroutine returns to its caller or resumer (9.4.4) by the `co_return` statement or when suspended (7.6.2.3). A coroutine shall not return to its caller or resumer by a `return` statement (8.6.3).

2 The *expr-or-braced-init-list* of a `co_return` statement is called its operand. Let p be an lvalue naming the coroutine promise object (9.4.4). A `co_return` statement is equivalent to:

`{` S`;` `goto` *final-suspend* `;` `}`

where *final-suspend* is the exposition-only label defined in 9.4.4 and S is defined as follows:

(2.1) — S is `p.return_value(`*expr-or-braced-init-list*`)`, if the operand is a *braced-init-list* or an expression of non-`void` type;

(2.2) — S is `{` *expression*$_{opt}$ `;` `p.return_void();` `}`, otherwise;

S shall be a prvalue of type `void`.

3 If `p.return_void()` is a valid expression, flowing off the end of a coroutine is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine results in undefined behavior.

8.6.5 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (8.1) located in the current function.

8.7 Declaration statement [stmt.dcl]

1 A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

2 Variables with automatic storage duration (6.6.5.3) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (8.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization (including ones in *conditions* and *init-statements*). A program that jumps[85] from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has vacuous initialization (9.3). In such a case, the variables with vacuous initialization are constructed in the order of their declaration. [*Example*:

```
void f() {
  // ...
  goto lx;          // ill-formed: jump into scope of a
  // ...
ly:
  X a = 1;
  // ...
lx:
  goto ly;          // OK, jump implies destructor call for a followed by
                    // construction again immediately following label ly
}
```

— *end example*]

4 Dynamic initialization of a block-scope variable with static storage duration (6.6.5.1) or thread storage duration (6.6.5.2) is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution

85) The transfer from the condition of a `switch` statement to a `case` label is considered a jump in this respect.

elided or converted to a move operation if an automatic storage duration variable is returned (11.9.5). — *end note*] [*Example*:

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— *end example*] Flowing off the end of a constructor, a destructor, or a non-coroutine function with a *cv* `void` return type is equivalent to a `return` with no operand. Otherwise, flowing off the end of a function other than `main` (6.8.3.1) or a coroutine (9.4.4) results in undefined behavior.

3 The copy-initialization of the result of the call is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the `return` statement, which, in turn, is sequenced before the destruction of local variables (8.6) of the block enclosing the `return` statement.

8.6.4 The `co_return` statement [stmt.return.coroutine]

coroutine-return-statement:
`co_return` *expr-or-braced-init-list*$_{opt}$ `;`

1 A coroutine returns to its caller or resumer (9.4.4) by the `co_return` statement or when suspended (7.6.2.3). A coroutine shall not return to its caller or resumer by a `return` statement (8.6.3).

2 The *expr-or-braced-init-list* of a `co_return` statement is called its operand. Let p be an lvalue naming the coroutine promise object (9.4.4). A `co_return` statement is equivalent to:

`{` S`;` `goto` *final-suspend* `;` `}`

where *final-suspend* is the exposition-only label defined in 9.4.4 and S is defined as follows:

(2.1) — S is `p.return_value(`*expr-or-braced-init-list*`)`, if the operand is a *braced-init-list* or an expression of non-`void` type;

(2.2) — S is `{` *expression*$_{opt}$ `;` `p.return_void();` `}`, otherwise;

S shall be a prvalue of type `void`.

3 If `p.return_void()` is a valid expression, flowing off the end of a coroutine is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine results in undefined behavior.

8.6.5 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (8.1) located in the current function.

8.7 Declaration statement [stmt.dcl]

1 A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

2 Variables with automatic storage duration (6.6.5.3) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (8.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization (including ones in *conditions* and *init-statements*). A program that jumps[85] from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has vacuous initialization (9.3). In such a case, the variables with vacuous initialization are constructed in the order of their declaration. [*Example*:

```
void f() {
  // ...
  goto lx;          // ill-formed: jump into scope of a
  // ...
ly:
  X a = 1;
```

85) The transfer from the condition of a `switch` statement to a `case` label is considered a jump in this respect.

shall wait for completion of the initialization.[86] If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined. [*Example*:

```
int foo(int i) {
  static int s = foo(2*i);      // recursive call - undefined
  return i+1;
}
```

— *end example*]

5 A block-scope object with static or thread storage duration will be destroyed if and only if it was constructed. [*Note*: 6.8.3.4 describes the order in which block-scope objects with static and thread storage duration are destroyed. — *end note*]

8.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statements* and *declarations*: An *expression-statement* with a function-style explicit type conversion (7.6.1.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*.

2 [*Note*: If the *statement* cannot syntactically be a *declaration*, there is no ambiguity, so this rule does not apply. The whole *statement* might need to be examined to determine whether this is the case. This resolves the meaning of many examples. [*Example*: Assuming `T` is a *simple-type-specifier* (9.1.8),

```
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement

T(*d)(int);           // declaration
T(e)[5];              // declaration
T(f) = { 1, 2 };      // declaration
T(*g)(double(3));     // declaration
```

In the last example above, `g`, which is a pointer to `T`, is initialized to `double(3)`. This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. — *end example*]

The remaining cases are *declarations*. [*Example*:

```
class T {
  // ...
public:
  T();
  T(int);
  T(int, int);
};
T(a);                 // declaration
T(*b)();              // declaration
T(c)=7;               // declaration
T(d),e,f=3;           // declaration
extern int h;
T(g)(h,2);            // declaration
```

— *end example*] — *end note*]

3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are *type-names* or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a *type-name*. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [*Note*: This can occur only when the name is declared earlier in the declaration. — *end note*] [*Example*:

```
struct T1 {
  T1 operator()(int x) { return T1(x); }
  int operator=(int x) { return x; }
  // ...
lx:
  goto ly;            // OK, jump implies destructor call for a followed by
                      // construction again immediately following label ly
}
```

— *end example*]

4 Dynamic initialization of a block-scope variable with static storage duration (6.6.5.1) or thread storage duration (6.6.5.2) is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.[86] If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined. [*Example*:

```
int foo(int i) {
  static int s = foo(2*i);      // recursive call - undefined
  return i+1;
}
```

— *end example*]

5 A block-scope object with static or thread storage duration will be destroyed if and only if it was constructed. [*Note*: 6.8.3.4 describes the order in which block-scope objects with static and thread storage duration are destroyed. — *end note*]

8.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statements* and *declarations*: An *expression-statement* with a function-style explicit type conversion (7.6.1.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*.

2 [*Note*: If the *statement* cannot syntactically be a *declaration*, there is no ambiguity, so this rule does not apply. The whole *statement* might need to be examined to determine whether this is the case. This resolves the meaning of many examples. [*Example*: Assuming `T` is a *simple-type-specifier* (9.1.8),

```
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement

T(*d)(int);           // declaration
T(e)[5];              // declaration
T(f) = { 1, 2 };      // declaration
T(*g)(double(3));     // declaration
```

In the last example above, `g`, which is a pointer to `T`, is initialized to `double(3)`. This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. — *end example*]

The remaining cases are *declarations*. [*Example*:

```
class T {
  // ...
public:
  T();
  T(int);
  T(int, int);
};
T(a);                 // declaration
T(*b)();              // declaration
T(c)=7;               // declaration
T(d),e,f=3;           // declaration
extern int h;
T(g)(h,2);            // declaration
```

[86] The implementation must not introduce any deadlock around execution of the initializer. Deadlocks might still be caused by the program logic; the implementation need only avoid deadlocks due to its own synchronization operations.

©ISO/IEC Dxxxx

[86] The implementation must not introduce any deadlock around execution of the initializer. Deadlocks might still be caused by the program logic; the implementation need only avoid deadlocks due to its own synchronization operations.

§ 8.8 150

```
  T1(int) { }
};
struct T2 { T2(int){ } };
int a, (*(*b)(T2))(int), c, d;

void f() {
  // disambiguation requires this to be parsed as a declaration:
  T1(a) = 3,
  T2(4),                        // T2 will be declared as a variable of type T1, but this will not
  (*(*b)(T2(c)))(int(d));       // allow the last part of the declaration to parse properly,
                                // since it depends on T2 being a type-name
}
```

— *end example*]

— *end example*] — *end note*]

3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are *type-names* or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a *type-name*. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [*Note*: This can occur only when the name is declared earlier in the declaration. — *end note*] [*Example*:

```
struct T1 {
  T1 operator()(int x) { return T1(x); }
  int operator=(int x) { return x; }
  T1(int) { }
};
struct T2 { T2(int){ } };
int a, (*(*b)(T2))(int), c, d;

void f() {
  // disambiguation requires this to be parsed as a declaration:
  T1(a) = 3,
  T2(4),                        // T2 will be declared as a variable of type T1, but this will not
  (*(*b)(T2(c)))(int(d));       // allow the last part of the declaration to parse properly,
                                // since it depends on T2 being a type-name
}
```

— *end example*]

where T is of the form *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq* and D is a declarator. Following is a recursive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

4 First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* T determines the type T. [*Example*: In the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (9.1.8.2). — *end example*]

5 In a declaration *attribute-specifier-seq*$_{opt}$ `T D` where `D` is an unadorned identifier the type of this identifier is "`T`".

6 In a declaration `T D` where `D` has the form

```
( D1 )
```

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

9.2.3.1 Pointers [dcl.ptr]

1 In a declaration `T D` where `D` has the form

* *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to `T`". The *cv-qualifier*s apply to the pointer and not to the object pointed to. Similarly, the optional *attribute-specifier-seq* (9.11.1) appertains to the pointer and not to the object pointed to.

2 [*Example*: The declarations

```
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
```

declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer; `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

```
i = ci;
*cp = ci;
pc++;
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```
ci = 1;          // error
ci++;            // error
*pc = 2;         // error
cp = &ci;        // error
cpc++;           // error
p = pc;          // error
ppc = &p;        // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

```
*ppc = &ci;      // OK, but would make p point to ci because of previous error
*p = 5;          // clobber ci
```

— *end example*]

3 See also 7.6.19 and 9.3.

©ISO/IEC Dxxxx

where T is of the form *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq* and D is a declarator. Following is a recursive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

4 First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* T determines the type T. [*Example*: In the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (9.1.8.2). — *end example*]

5 In a declaration *attribute-specifier-seq*$_{opt}$ `T D` where `D` is an unadorned identifier the type of this identifier is "`T`".

6 In a declaration `T D` where `D` has the form

```
( D1 )
```

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

9.2.3.1 Pointers [dcl.ptr]

1 In a declaration `T D` where `D` has the form

* *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to `T`". The *cv-qualifier*s apply to the pointer and not to the object pointed to. Similarly, the optional *attribute-specifier-seq* (9.11.1) appertains to the pointer and not to the object pointed to.

2 [*Example*: The declarations

```
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
```

declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer; `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

```
i = ci;
*cp = ci;
pc++;
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```
ci = 1;          // error
ci++;            // error
*pc = 2;         // error
cp = &ci;        // error
cpc++;           // error
p = pc;          // error
ppc = &p;        // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

```
*ppc = &ci;      // OK, but would make p point to ci because of previous error
*p = 5;          // clobber ci
```

— *end example*]

3 See also 7.6.19 and 9.3.

§ 9.2.3.1 174

[4] [*Note*: Forming a pointer to reference type is ill-formed; see 9.2.3.2. Forming a function pointer type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 9.2.3.5. Since the address of a bit-field (11.3.9) cannot be taken, a pointer can never point to a bit-field. — *end note*]

9.2.3.2 References [dcl.ref]

[1] In a declaration `T D` where `D` has either of the forms

`&` *attribute-specifier-seq*$_{opt}$ `D1`
`&&` *attribute-specifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`". The optional *attribute-specifier-seq* appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (9.1.3, 13.1) or *decltype-specifier* (9.1.8.2), in which case the cv-qualifiers are ignored. [*Example*:

```
typedef int& A;
const A aref = 3;    // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`", not "lvalue reference to `const int`". — *end example*] [*Note*: A reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

[2] A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

[3] [*Example*:

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = 0;
}

void k() {
   link* q = new link;
   h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 9.3.3. — *end example*]

[4] It is unspecified whether or not a reference requires storage (6.6.5).

[5] There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (9.3.3) except when the declaration contains an explicit `extern` specifier (9.1.1), is a class member (11.3) declaration within a class definition, or is the declaration of

©ISO/IEC Dxxxx

[4] [*Note*: Forming a pointer to reference type is ill-formed; see 9.2.3.2. Forming a function pointer type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 9.2.3.5. Since the address of a bit-field (11.3.9) cannot be taken, a pointer can never point to a bit-field. — *end note*]

9.2.3.2 References [dcl.ref]

[1] In a declaration `T D` where `D` has either of the forms

`&` *attribute-specifier-seq*$_{opt}$ `D1`
`&&` *attribute-specifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`". The optional *attribute-specifier-seq* appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (9.1.3, 13.1) or *decltype-specifier* (9.1.8.2), in which case the cv-qualifiers are ignored. [*Example*:

```
typedef int& A;
const A aref = 3;    // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`", not "lvalue reference to `const int`". — *end example*] [*Note*: A reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

[2] A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

[3] [*Example*:

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = 0;
}

void k() {
   link* q = new link;
   h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 9.3.3. — *end example*]

[4] It is unspecified whether or not a reference requires storage (6.6.5).

[5] There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (9.3.3) except when the declaration contains an explicit `extern` specifier (9.1.1), is a class member (11.3) declaration within a class definition, or is the declaration of

§ 9.2.3.2 175

a parameter or a return type (9.2.3.5); see 6.1. A reference shall be initialized to refer to a valid object or function. [*Note*: In particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by indirection through a null pointer, which causes undefined behavior. As described in 11.3.9, a reference cannot be bound directly to a bit-field. — *end note*]

6 If a *typedef-name* (9.1.3, 13.1) or a *decltype-specifier* (9.1.8.2) denotes a type `TR` that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TR`" creates the type "lvalue reference to `T`", while an attempt to create the type "rvalue reference to *cv* `TR`" creates the type `TR`. [*Note*: This rule is known as reference collapsing. — *end note*] [*Example*:

```
int i;
typedef int& LRI;
typedef int&& RRI;

LRI& r1 = i;                    // r1 has the type int&
const LRI& r2 = i;              // r2 has the type int&
const LRI&& r3 = i;             // r3 has the type int&

RRI& r4 = i;                    // r4 has the type int&
RRI&& r5 = 5;                   // r5 has the type int&&

decltype(r2)& r6 = i;           // r6 has the type int&
decltype(r2)&& r7 = i;          // r7 has the type int&
```

— *end example*]

7 [*Note*: Forming a reference to function type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 9.2.3.5. — *end note*]

9.2.3.3 Pointers to members [dcl.mptr]

1 In a declaration `T D` where `D` has the form

nested-name-specifier `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$ `D1`

and the *nested-name-specifier* denotes a class, and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of class *nested-name-specifier* of type `T`". The optional *attribute-specifier-seq* (9.11.1) appertains to the pointer-to-member.

2 [*Example*:

```
struct X {
  void f(int);
  int a;
};
struct Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
// ...
obj.*pmi = 7;       // assign 7 to an integer member of obj
(obj.*pmf)(7);      // call a function member of obj with the argument 7
```

— *end example*]

3 A pointer to member shall not point to a static member of a class (11.3.8), a member with reference type, or "*cv* `void`".

©ISO/IEC Dxxxx

a parameter or a return type (9.2.3.5); see 6.1. A reference shall be initialized to refer to a valid object or function. [*Note*: In particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by indirection through a null pointer, which causes undefined behavior. As described in 11.3.9, a reference cannot be bound directly to a bit-field. — *end note*]

6 If a *typedef-name* (9.1.3, 13.1) or a *decltype-specifier* (9.1.8.2) denotes a type `TR` that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TR`" creates the type "lvalue reference to `T`", while an attempt to create the type "rvalue reference to *cv* `TR`" creates the type `TR`. [*Note*: This rule is known as reference collapsing. — *end note*] [*Example*:

```
int i;
typedef int& LRI;
typedef int&& RRI;

LRI& r1 = i;                    // r1 has the type int&
const LRI& r2 = i;              // r2 has the type int&
const LRI&& r3 = i;             // r3 has the type int&

RRI& r4 = i;                    // r4 has the type int&
RRI&& r5 = 5;                   // r5 has the type int&&

decltype(r2)& r6 = i;           // r6 has the type int&
decltype(r2)&& r7 = i;          // r7 has the type int&
```

— *end example*]

7 [*Note*: Forming a reference to function type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 9.2.3.5. — *end note*]

9.2.3.3 Pointers to members [dcl.mptr]

1 In a declaration `T D` where `D` has the form

nested-name-specifier `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$ `D1`

and the *nested-name-specifier* denotes a class, and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of class *nested-name-specifier* of type `T`". The optional *attribute-specifier-seq* (9.11.1) appertains to the pointer-to-member.

2 [*Example*:

```
struct X {
  void f(int);
  int a;
};
struct Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
// ...
obj.*pmi = 7;       // assign 7 to an integer member of obj
(obj.*pmf)(7);      // call a function member of obj with the argument 7
```

— *end example*]

3 A pointer to member shall not point to a static member of a class (11.3.8), a member with reference type, or "*cv* `void`".

§ 9.2.3.3 176

[4] [*Note*: See also 7.6.2 and 7.6.4. The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer-to-member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — *end note*]

9.2.3.4 Arrays [dcl.array]

[1] In a declaration `T D` where `D` has the form

`D1 [` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* array of `N T`". The *constant-expression* shall be a converted constant expression of type `std::size_t` (7.7). Its value `N` specifies the *array bound*, i.e., the number of elements in the array; `N` shall be greater than zero.

[2] In a declaration `T D` where `D` has the form

`D1 [ ]` *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* array of unknown bound of `T`", except as specified below.

[3] A type of the form "array of `N U`" or "array of unknown bound of `U`" is an *array type*. The optional *attribute-specifier-seq* appertains to the array type.

[4] `U` is called the array *element type*; this type shall not be a placeholder type (9.1.8.5), a reference type, a function type, an array of unknown bound, or *cv* `void`. [*Note*: An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from an array of known bound. — *end note*] [*Example*:

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. — *end example*]

[5] Any type of the form "*cv-qualifier-seq* array of `N U`" is adjusted to "array of `N` *cv-qualifier-seq* `U`", and similarly for "array of unknown bound of `U`". [*Example*:

```
typedef int A[5], AA[2][3];
typedef const A CA;             // type is "array of 5 const int"
typedef const AA CAA;           // type is "array of 2 array of 3 const int"
```

— *end example*] [*Note*: An "array of `N` *cv-qualifier-seq* `U`" has cv-qualified type; see 6.7.3. — *end note*]

[6] An object of type "array of `N U`" contains a contiguously allocated non-empty set of `N` subobjects of type `U`, known as the *elements* of the array, and numbered `0` to `N-1`.

[7] In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (9.2.3.5). An array bound may also be omitted when an object (but not a non-static data member) of array type is initialized and the declarator is followed by an initializer (9.3, 11.3, 7.6.1.3, 7.6.2.7). In these cases, the array bound is calculated from the number of initial elements (say, `N`) supplied (9.3.1), and the type of the array is "array of `N U`".

[8] Furthermore, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class. [*Example*:

```
extern int x[10];
struct S {
  static int y[10];
};

int x[];                // OK: bound is 10
int S::y[];             // OK: bound is 10

void f() {
  extern int x[];
  int i = sizeof(x);    // error: incomplete object type
}
```

— *end example*]

[4] [*Note*: See also 7.6.2 and 7.6.4. The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer-to-member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — *end note*]

9.2.3.4 Arrays [dcl.array]

[1] In a declaration `T D` where `D` has the form

`D1 [` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* array of `N T`". The *constant-expression* shall be a converted constant expression of type `std::size_t` (7.7). Its value `N` specifies the *array bound*, i.e., the number of elements in the array; `N` shall be greater than zero.

[2] In a declaration `T D` where `D` has the form

`D1 [ ]` *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* array of unknown bound of `T`", except as specified below.

[3] A type of the form "array of `N U`" or "array of unknown bound of `U`" is an *array type*. The optional *attribute-specifier-seq* appertains to the array type.

[4] `U` is called the array *element type*; this type shall not be a placeholder type (9.1.8.5), a reference type, a function type, an array of unknown bound, or *cv* `void`. [*Note*: An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from an array of known bound. — *end note*] [*Example*:

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. — *end example*]

[5] Any type of the form "*cv-qualifier-seq* array of `N U`" is adjusted to "array of `N` *cv-qualifier-seq* `U`", and similarly for "array of unknown bound of `U`". [*Example*:

```
typedef int A[5], AA[2][3];
typedef const A CA;             // type is "array of 5 const int"
typedef const AA CAA;           // type is "array of 2 array of 3 const int"
```

— *end example*] [*Note*: An "array of `N` *cv-qualifier-seq* `U`" has cv-qualified type; see 6.7.3. — *end note*]

[6] An object of type "array of `N U`" contains a contiguously allocated non-empty set of `N` subobjects of type `U`, known as the *elements* of the array, and numbered `0` to `N-1`.

[7] In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (9.2.3.5). An array bound may also be omitted when an object (but not a non-static data member) of array type is initialized and the declarator is followed by an initializer (9.3, 11.3, 7.6.1.3, 7.6.2.7). In these cases, the array bound is calculated from the number of initial elements (say, `N`) supplied (9.3.1), and the type of the array is "array of `N U`".

[8] Furthermore, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class. [*Example*:

```
extern int x[10];
struct S {
  static int y[10];
};

int x[];                // OK: bound is 10
int S::y[];             // OK: bound is 10

void f() {
  extern int x[];
  int i = sizeof(x);    // error: incomplete object type
}
```

— *end example*]

```
  void V::f() {                 // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();            // ... so this declares Q::V::h
  }
  void V::C::m() {              // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```

— *end example*]

5 If the optional initial **inline** keyword appears in a *namespace-definition* for a particular namespace, that namespace is declared to be an *inline namespace*. The **inline** keyword may be used on a *namespace-definition* that extends a namespace only if it was previously used on the *namespace-definition* that initially declared the *namespace-name* for that namespace.

6 The optional *attribute-specifier-seq* in a *named-namespace-definition* appertains to the namespace being defined or extended.

7 Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are both added to the set of associated namespaces used in argument-dependent lookup (6.4.2) whenever one of them is, and a *using-directive* (9.7.3) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (9.7.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (13.6.5), explicitly instantiated (13.8.2), or explicitly specialized (13.8.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (6.4.3.2) will include members of the inline namespace brought in by the *using-directive* even if there are declarations of that name in the enclosing namespace.

8 These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. The *inline namespace set* of `N` is the transitive closure of all inline namespaces in `N`. The *enclosing namespace set* of `O` is the set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `O`, together with any intervening inline namespaces.

9 A *nested-namespace-definition* with an *enclosing-namespace-specifier* `E`, *identifier* `I` and *namespace-body* `B` is equivalent to

```
namespace E { inline_opt namespace I { B } }
```

where the optional **inline** is present if and only if the *identifier* `I` is preceded by **inline**. [*Example*:

```
namespace A::inline B::C {
  int i;
}
```

The above has the same effect as:

```
namespace A {
  inline namespace B {
    namespace C {
      int i;
    }
  }
}
```

— *end example*]

9.7.1.1 Unnamed namespaces [namespace.unnamed]

1 An *unnamed-namespace-definition* behaves as if it were replaced by

```
inline_opt namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where **inline** appears if and only if it appears in the *unnamed-namespace-definition* and all occurrences of **unique** in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional *attribute-specifier-seq* in the *unnamed-namespace-definition* appertains to **unique**. [*Example*:

```
namespace { int i; }            // unique::i
void f() { i++; }               // unique::i++
```


©ISO/IEC Dxxxx

```
  void V::f() {                 // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();            // ... so this declares Q::V::h
  }
  void V::C::m() {              // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```

— *end example*]

5 If the optional initial **inline** keyword appears in a *namespace-definition* for a particular namespace, that namespace is declared to be an *inline namespace*. The **inline** keyword may be used on a *namespace-definition* that extends a namespace only if it was previously used on the *namespace-definition* that initially declared the *namespace-name* for that namespace.

6 The optional *attribute-specifier-seq* in a *named-namespace-definition* appertains to the namespace being defined or extended.

7 Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are both added to the set of associated namespaces used in argument-dependent lookup (6.4.2) whenever one of them is, and a *using-directive* (9.7.3) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (9.7.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (13.6.5), explicitly instantiated (13.8.2), or explicitly specialized (13.8.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (6.4.3.2) will include members of the inline namespace brought in by the *using-directive* even if there are declarations of that name in the enclosing namespace.

8 These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. The *inline namespace set* of `N` is the transitive closure of all inline namespaces in `N`. The *enclosing namespace set* of `O` is the set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `O`, together with any intervening inline namespaces.

9 A *nested-namespace-definition* with an *enclosing-namespace-specifier* `E`, *identifier* `I` and *namespace-body* `B` is equivalent to

```
namespace E { inline_opt namespace I { B } }
```

where the optional **inline** is present if and only if the *identifier* `I` is preceded by **inline**. [*Example*:

```
namespace A::inline B::C {
  int i;
}
```

The above has the same effect as:

```
namespace A {
  inline namespace B {
    namespace C {
      int i;
    }
  }
}
```

— *end example*]

9.7.1.1 Unnamed namespaces [namespace.unnamed]

1 An *unnamed-namespace-definition* behaves as if it were replaced by

```
inline_opt namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where **inline** appears if and only if it appears in the *unnamed-namespace-definition* and all occurrences of **unique** in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional *attribute-specifier-seq* in the *unnamed-namespace-definition* appertains to **unique**. [*Example*:

```
namespace { int i; }            // unique::i
void f() { i++; }               // unique::i++
```

§ 9.7.1.1 211

[8] A defaulted move special function (11.3.4.2, 11.3.5) that is defined as deleted is excluded from the set of candidate functions in all contexts. A constructor inherited from class type `C` (11.9.3) that has a first parameter of type "reference to *cv1* `P`" (including such a constructor instantiated from a template) is excluded from the set of candidate functions when constructing an object of type *cv2* `D` if the argument list has exactly one argument and `C` is reference-related to `P` and `P` is reference-related to `D`. [*Example*:

```
struct A {
  A();
  A(A &&);                          // #1
  template<typename T> A(T &&);     // #2
};
struct B : A {
  using A::A;
  B(const B &);                     // #3
  B(B &&) = default;                // #4, implicitly deleted

  struct X { X(X &&) = delete; } x;
};
extern B b1;
B b2 = static_cast<B&&>(b1);        // calls #3: #1, #2, and #4 are not viable
struct C { operator B&&(); };
B b3 = C();                         // calls #3
```

— *end example*]

12.3.1.1 Function call syntax [over.match.call]

[1] In a function call (7.6.1.2)

postfix-expression (*expression-list*$_{opt}$)

if the *postfix-expression* denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 12.3.1.1.1. If the *postfix-expression* denotes an object of class type, overload resolution is applied as specified in 12.3.1.1.2.

[2] If the *postfix-expression* denotes the address of a set of overloaded functions and/or function templates, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed. [*Note*: The resolution of the address of an overload set in other contexts is described in 12.4. — *end note*]

12.3.1.1.1 Call to named function [over.call.func]

[1] Of interest in 12.3.1.1.1 are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
postfix-expression . *id-expression*
postfix-expression -> *id-expression*
primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

[2] In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of Clause 12 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, Clause 12 assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class.[119] Under this assumption, the *id-expression* in the call is looked up as a member function of `T` following the rules for looking up names in classes (11.7). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (12.3.1).

[3] In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal

119) Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

©ISO/IEC Dxxxx

[8] A defaulted move special function (11.3.4.2, 11.3.5) that is defined as deleted is excluded from the set of candidate functions in all contexts. A constructor inherited from class type `C` (11.9.3) that has a first parameter of type "reference to *cv1* `P`" (including such a constructor instantiated from a template) is excluded from the set of candidate functions when constructing an object of type *cv2* `D` if the argument list has exactly one argument and `C` is reference-related to `P` and `P` is reference-related to `D`. [*Example*:

```
struct A {
  A();
  A(A &&);                          // #1
  template<typename T> A(T &&);     // #2
};
struct B : A {
  using A::A;
  B(const B &);                     // #3
  B(B &&) = default;                // #4, implicitly deleted

  struct X { X(X &&) = delete; } x;
};
extern B b1;
B b2 = static_cast<B&&>(b1);        // calls #3: #1, #2, and #4 are not viable
struct C { operator B&&(); };
B b3 = C();                         // calls #3
```

— *end example*]

12.3.1.1 Function call syntax [over.match.call]

[1] In a function call (7.6.1.2)

postfix-expression (*expression-list*$_{opt}$)

if the *postfix-expression* denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 12.3.1.1.1. If the *postfix-expression* denotes an object of class type, overload resolution is applied as specified in 12.3.1.1.2.

[2] If the *postfix-expression* denotes the address of a set of overloaded functions and/or function templates, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed. [*Note*: The resolution of the address of an overload set in other contexts is described in 12.4. — *end note*]

12.3.1.1.1 Call to named function [over.call.func]

[1] Of interest in 12.3.1.1.1 are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
postfix-expression . *id-expression*
postfix-expression -> *id-expression*
primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

[2] In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of Clause 12 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, Clause 12 assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class.[119] Under this assumption, the *id-expression* in the call is looked up as a member function of `T` following the rules for looking up names in classes (11.7). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (12.3.1).

[3] In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal

119) Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

§ 12.3.1.1.1 310

©ISO/IEC Dxxxx

©ISO/IEC Dxxxx

Index 1711

W

X

Y

Z

©ISO/IEC Dxxxx

valid but unspecified state, 451
value, 65
  call by, 108
  denormalized, *see* number, subnormal
  indeterminate, 56
  invalid pointer, 68
  null member pointer, 90
  null pointer, 68, 89
  undefined unrepresentable integral, 89
value category, 83
value computation, 62, 72–73, 75, 76, 110, 123, 132, 133, 135, 136
value-initialization, 186
variable, 24
  function-local predefined, 200
  indeterminate uninitialized, 185
  inline, 161
  needed for constant evaluation, 141
  program semantics affected by the existence of a variable definition, 393
variable arguments, 438
variable template
  definition of, 339
`<variant>`, 581, 1628
variant member, 269
`<vector>`, 815, 817, 951
vectorization-unsafe, 1017
`<version>`, 481, 1634
virt-specifier, 246, 1604
virt-specifier-seq, 246, 1604
`virtual`, 15, 156, 270, 1598, 1604
virtual base class, *see* base class, virtual, *see* base class, virtual
virtual function, *see* function, virtual, *see* function, virtual
virtual function call, 276
  constructor and, 297
  destructor and, 297
  undefined pure, 278
visibility, 36
visible, 36
visible side effect, *see* side effect, visible
`void`, 15, 163, 1599
`void*`
  type, 69
`void&`, 175
`volatile`, 15, 69, 171, 1600
  constructor and, 251, 253
  destructor and, 251, 259
  implementation-defined, 163
  overloading and, 306
volatile member function, 251
volatile object, *see* object, volatile
volatile-qualified, 69

W

waiting function, 1575
`<wchar.h>`, 779, 1639

X

`X(X&)`, *see* constructor, copy
`xor`, 15
`xor_eq`, 15
xvalue, 83

Y

Y_ℓ^m (spherical associated Legendre functions), 1206
yield-expression, 134, 1596

Z

zero
  division by undefined, 82
  remainder undefined, 82
  undefined division by, 128
zero-initialization, 185
zeta functions ζ, 1205

©ISO/IEC Dxxxx

cv-qualifier-seq, 91, 127, 162, **171**, 174, 176–179, 200, 311, 566, **1600**

d-char, 11, **20**, **1592**
d-char-sequence, 20, **20**, 1591, **1591**
decimal-floating-literal, **19**, **1591**
decimal-literal, **15**, **1590**
decl-specifier, 94, 95, 143, 148, 152–154, **154**, 156, 164, 167, 169, 170, 252, 259, 262, 375, **1598**, 1632
decl-specifier-seq, 32, 69, 94, 95, 142, 143, 148, 152–154, **154**, 155, 156, 162, 166, 167, 169, 170, 173, 174, 179, 181, 200, 205, 207, 247, 252, 259, 262, 286, 375, **1598**, 1615
declaration, 24, 25, 35, 49, 150, **152**, 157, 162, 210, 216, 218, 233, 245, 286, 339, 340, 373, 396, 397, **1597**
declaration-seq, 8, 24, **152**, 233, **1597**
declaration-statement, 117, 149, **149**, **1597**
declarator, 25, 69, 91, 142, 150, 152, 153, 156, 167, 170, **171**, 173, 179, 182, 200, 247, 252, 258, 375, 396, **1599**
declarator-id, 34, 37–39, 42, 43, 46, 93, 99, 153, **171**, 173, 174, 177, 178, 180–182, 205, 212, 226, 229, 230, 247, 264, 334, 340, 345, 375, 376, 409, 413, 417, 420, **1600**, 1632
decltype-specifier, 42, 61, 84, 93, 97, 117, 164, **165**, 166, 173, 175, 176, 207, 238, 242, 260, 345, 356, 414, **1599**
deduction-guide, 25, 152, 316, 317, 320, 339, 421, **421**, **1606**
defined-macro-expression, **432**, 433, **1607**
defining-type-id, 31, 157, 163, **172**, 316, 375, **1600**
defining-type-specifier, 153, 154, 156, 162, **162**, 263, **1598**
defining-type-specifier-seq, 157, 162, **162**, **1598**
delete-expression, 27, 54, 58, 122, 124, **124**, 125, 126, 138, 260, 303, 304, 496, 497, **1595**
designated-initializer-clause, **185**, 189, 195, **1601**
designated-initializer-list, **185**, 188, 189, 195, 196, 316, 325, **1601**
designator, **185**, 188, 189, 196, **1601**
digit, **13**, 232, 435, **1589**
digit-sequence, 19, **19**, **1591**
directory-separator, 1428, **1428**, 1429, 1433, 1435, 1437, 1438

elaborated-enum-specifier, **165**, 209, **1599**
elaborated-type-specifier, 25, 26, 31, 32, 43, 47, 153, 157, 165, **165**, 212, 213, 225, 242, 244, 245, 278, 345, 364, 375, 378, 396, **1599**
elif-group, **431**, **1607**
elif-groups, **431**, **1607**
else-group, **431**, **1607**
empty-declaration, 24, 25, **152**, 153, 246, **1598**
enclosing-namespace-specifier, **210**, 211, **1602**
encoding-prefix, **17**, 20, 21, 334, **1590**

endif-line, **431**, **1607**
enum-base, **206**, 207, 208, **1601**
enum-head, **206**, 207, **1601**
enum-head-name, **206**, 207, 212, 345, **1601**
enum-key, **206**, 207, **1601**
enum-name, 47, 153, 165, 206, **206**, 208, **1601**
enum-specifier, 31, 35, 39, 153, 162, **206**, 207–209, 246, 247, **1601**
enumerator, 39, 42, 153, 207, **207**, 208, **1601**
enumerator-definition, 31, 39, 207, **207**, **1601**
enumerator-list, 207, **207**, 208, **1601**
equality-expression, **131**, **1595**
escape-sequence, **17**, **1590**
exception-declaration, 28, 32, 40, 225, 298, 299, 422, **422**, 424–426, **1606**
exclusive-or-expression, **132**, **1595**
explicit-instantiation, 153, 155, 396, **396**, 397, **1606**
explicit-specialization, 153, **398**, **1606**
explicit-specifier, 156, **156**, 173, 252, 309, 316, 317, 406, **1598**
exponent-part, **19**, **1591**
export-declaration, 210, 233, **233**, 236, 340, **1603**
expr-or-braced-init-list, 107, 136, 148, 149, **185**, **1601**, 1638
expression, 39, 52, 63, 85, 105, 108, 112, 120, 121, **136**, 142, 147, 149, 298, 299, 352, 354, 356, 384–386, 414, 705, **1596**
expression-list, 63, 70, **107**, 108, 109, 123, 167, 186–188, 289, 291, 293, 310, 311, 314, 316, 333, 362, 380, **1594**
expression-statement, **143**, 150, **1596**

fallback-separator, **1428**
filename, 1425, 1428, **1428**, 1436, 1438
floating-literal, **19**, **1591**
floating-suffix, **19**, **1591**
fold-expression, **103**, 361, 362, 385, **1593**
fold-operator, 103, **103**, 362, **1593**
for-init-statement, 117
for-range-declaration, 32, 145, **145**, 146, 148, **1597**
for-range-initializer, 32, **145**, 147, 195, **1597**
fractional-constant, **19**, **1591**
function-body, 24, 32, 98, 117, 159, 160, 200, **200**, 201–203, 425, **1601**
function-definition, 25, 91, 152, 153, 156, 200, **200**, 375, **1601**
function-specifier, 156, **156**, **1598**
function-try-block, 33, 40, 108, 299, 422, **422**, 423, 425, 426, 524, **1606**

global-module-fragment, 233, 237, **237**, 460, **1603**
group, **431**, **1606**
group-part, **431**, **1606**

h-char, **13**, **1589**
h-char-sequence, **12**, 13, **1589**
h-pp-tokens, **432**, **1607**

Index of grammar productions 1718

©ISO/IEC Dxxxx

storage-class-specifier, 154, **154**, 155, 205, 247, 286, 396, 398, **1598**, 1626
string-literal, **19**, 20, 21, 153, 193, 222, 227, 230, 334, 439, **1591**

template-argument, 23, 39, 41, 43, 44, 48, 179, 282, 341–344, **344**, 345–351, 355, 358, 361, 363, 373, 378, 386, 391, 397, 401, 403–405, 408, 411, 418–420, **1606**
template-argument-list, 344, **344**, 345, 347, 356, 361, 362, 367, 378, 384, **1606**
template-declaration, 35, 152, 153, 157, 286, 339, **339**, 340, 373, **1605**
template-head, 4, 5, 181, 339, **339**, 350, 351, 356, 359, 370, 371, **1605**
template-id, 24, 29, 41, 43, 44, 93, 206, 213, 217, 330, 340, 344, **344**, 345, 346, 348, 355, 356, 360, 363, 367, 373, 380, 385, 390, 396, 401, 403, 405, **1606**
template-name, 36, 43, 157, 163, 164, 237, 238, 245, 316, 341, 344, **344**, 346, 355, 356, 364, 373, 378, 379, 396, 421, **1606**, 1633
template-parameter, 25, 32, 35, 143, 157, 165, 167, 170, 180–182, 330, 335, 341, **341**, 342, 343, 345–351, 355, 370, 373–375, 378–381, 384, 386, 402, 404, 408, 410, 411, 414, 418, 419, 421, **1605**
template-parameter-list, 4, 35, 95, 180, 181, 335, 339, **339**, 340–344, 346, 353, 367, 370, 402, **1605**
template-parameters, 181
text-line, **431**, **1607**
throw-expression, 133, 135, **135**, 138, 298, 299, 423, 425, 427, 429, **1596**
token, **12**, 225, 233, **1588**, 1603
top-level-declaration, 49, **49**, 236, 238, **1592**
top-level-declaration-seq, **49**, 236–239, 387, **1592**
trailing-return-type, 91, 95, 97, 167, **171**, 178, 180, 375, **1599**
translation unit, 238
translation-unit, **48**, 236, 387, **1592**
try-block, 142, 298, 299, 422, **422**, 423, 524, **1606**
type-constraint, 96, 105, 166, 169, 181, **339**, 340, 342, 343, 353, 356, 370, 371, 376, 392, 395, **1605**
type-id, 3, 69, 70, 87, 107, 112, 119–121, 167, 170, 171, **172**, 173, 179, 226, 344, 347, 348, 373, 375, 385, 386, **1600**
type-name, 32, 43, 47, 48, 93, 117, 151, 154, **163**, 164, 165, 173, 237, 260, 356, 378, **1599**, 1620
type-parameter, 165, 179, 181, 341, **341**, 342, 343, 361, 375, 381, 419, **1605**
type-parameter-key, 341, **341**, **1605**
type-requirement, 105, **105**, **1594**
type-specifier, 142, 143, 148, 154, **161**, 162, 163, 167, 170, 178, 245, 247, **1598**, 1615
type-specifier-seq, 120, **161**, 162, 167, 170, 207, **1598**

typedef-name, 8, 36, 153, 154, 156, **156**, 157, 158, 165, 166, 175, 176, 179, 221, 227, 229, 341, 345, 356, 515, 516, 695, 708–710, 714, 737, 742, 903, 926, 934, 939, 1004, 1140, 1339, 1469, **1598**
typename-specifier, 109, 162, 345, 375, **375**, **1606**

ud-suffix, 22, **22**, 23, 334, **1592**
unary-expression, **116**, 260, 385, **1594**
unary-operator, **116**, **1594**
universal-character-name, 9, 10, **10**, 11, 13, 18, 21, **1588**
unnamed-namespace-definition, **210**, 211, **1602**
unqualified-id, 40, 46–48, 93, **93**, 94, 117, 118, 147, 173, 204, 206, 212, 216, 250, 264, 278, 341, 344, 363, 380, 396, **1593**, 1633
unsigned-suffix, **16**, **1590**
user-defined-character-literal, **22**, 23, **1592**
user-defined-floating-literal, 22, **22**, **1592**
user-defined-integer-literal, 22, **22**, **1592**
user-defined-literal, 22, **22**, **1592**
user-defined-string-literal, **22**, 23, 334, **1592**
using-declaration, 25, 40, 41, 43, 44, 50, 152, 173, 184, 207, 209, 216, **216**, 217–221, 234, 238, 242, 246, 257, 258, 278, 281, 305, 309, 321, 334, 345, 360, 361, 366, 377, 458, **1602**, 1627, 1640
using-declarator, 31, 43, 216, **216**, 217, 220, 221, 234, 361, **1602**
using-declarator-list, **216**, **1602**
using-directive, 25, 31, 33, 34, 36, 41, 42, 44, 46, 48, 50, 94, 152, 211, 214, **214**, 215, 305, **1602**
using-enum-declaration, 25, 209, **209**, **1601**

va-opt-replacement, 438, **438**, **1608**
virt-specifier, **246**, 247, 274, **1604**
virt-specifier-seq, 200, **246**, 247, **1604**

yield-expression, 118, 134, **134**, 138, 203, **1596**

Index of grammar productions 1721