

# 1 Basic

[atomicc.basic]

## 1.1 Introduction

[atomicc.intro]

AtomicC is a structural hardware description language that extends C++ with Bluespec-style modules, rules, interfaces, and methods.

AtomicC is structural in that all state elements in the hardware netlist are explicit in the source code of the design. AtomicC is a timed HDL, using SystemC terminology. Atomic actions (rules and method invocations) execute in a single clock cycle.

Like Connectal, AtomicC designs may include both hardware and software, using interfaces to specify hardware/software communication in a type safe way. The AtomicC compiler generates the code to pass arguments between hardware and software.

## 1.2 Compilation

[atomicc.compilation]

AtomicC execution consists of 3 phases:

- netlist generation,
- netlist compilation or implementation
- and runtime.

During netlist generation, modules are instantiated by executing their constructors. During this phase, any C++ constructs may be used, but the resulting netlist may only contain synthesizable components.

During netlist compilation, the netlist is analyzed and translated to an intermediate representation and then to Verilog for simulation or synthesis. Alternate translations are possible: to native code via LLVM, to System C, to Gallina for formal verification with the Coq Proof Assistant, etc.

## 1.3 Scheduling

[atomicc.schedule]

Unlike C++'s serialized execution model, AtomicC allows all 'ready' rules (transactions) to be executed on every clock cycle. The AtomicC compiler verifies that it is possible to consider all rules executed during a given clock cycle as though they were serialized into some linear, atomic ordering (in other words, "sequentially consistent"(SC)). Even though all concurrent rules are executed during the same clock cycle, SC allows us to examine the effects of each rule independent of any rules that could be executing at the same time.

Each rule has a set of state elements that it reads and another set of element that it writes. For the execution of a group of rules to be considered to be SC, the following must be true:

- Atomic: All operations for a given rule occur at the same point in the sequence.
- Read-before-write: A rule that writes a state element must occur later in the sequence than any rules that read the same state element.
- Non-conflicting: A given state element cannot be written by more than one concurrently executing rule.

Scheduling is done by building a graph:

- Nodes are rules/methods within a module.
- For all state elements, insert directed links from each node that writes the state element to every node that reads it.
- Verify that this graph has no cycles.

Cycles can be broken in 2 ways:

- Rules default to have lower priority than methods within a module.
- In the source text, "priority" statements can be used to break cycles when necessary.

The compiler and linker do not break cycles automatically and issue and require the user to annotate the source text with "priority" statements to resolve conflicts.

## 2 Classes

[class]

### 2.1 `__module`, `__emodule`

[atomicc.module]

A module is defined using the keyword "`__module`", resulting in generation of a corresponding output verilog module. It includes local state elements, interfaces exported, interfaces imported and rules for clustering operations into atomic transactions.

Modules are independently compiled, even if they exist in the same compilation unit. Rule and interface method scheduling logic is generated as part of the generated module. Scheduling constraints (read set, write set and relation to other scheduled elements) are generated into a metadata file, allowing schedule consistency between modules to be verified by the linker.

[Example:

```
__module Echo {
    EchoRequest      request;           // exported interface
    EchoIndication   *indication;       // imported interface
    bool busy;
    __int(32) itemSay;
    ...
    // implementation of method request.say(). Note the guard "if (!busy)".
    void request.say(__int(32) v) if(!busy) {
        itemSay = v;
        ...
    }
    void request.saw(__int(16) a, __int(16) b) if(!busy) {
        ...
    }
};
```

— end example]

To reference a separately compiled module, use "`__emodule`". These external module definitions only need to include the exported/imported interfaces.

[Example:

```
__module EchoResponder {
    EchoIndication   indication;         // exported interface
};
```

— end example]

### 2.2 `__interface`

[atomicc.interface]

An AtomicC interface is essentially an abstract class similar to a Java interface. All the methods are virtual and no default implementations are provided. AtomicC style uses composition of interfaces rather than inheritance.

The `__interface` keyword defines a list of methods that are exposed from an object. Instead of using object inheritance to define reusable interfaces, they are defined/exported explicitly by objects, allowing fine-grained specification of interface method visibility.

Methods of a module are translated to value ports for passing the method arguments and a pair of handshaking ports used for scheduling method invocations.

References to an object can only be done through interface methods. State element declarations inside an object (member variables) are private.

[Example:

```
__interface EchoRequest {
    void say(__int(32) v);
    void say2(__int(16) a, __int(16) b);
};
```

— end example]

## 2.3 guard clauses on module interface methods

[atomicc.guard]

<sup>1</sup> Method definitions in `__module` declarations have the form:

```
atomicc-method-definition:
    decl-specifier-seqopt interface-qualifier-seq identifier parameters-and-qualifiers function-body

interface-qualifier:
    identifier .

interface-qualifier-seq:
    interface-qualifier
    interface-qualifier-seq interface-qualifier

atomicc-function-body:
    ctor-initializeropt if-guardopt compound-statement

if-guard:
    if ( condition )
```

Rules are only ready to fire if the rule's guard is true and all the guards on methods invoked within the rule are also true.

```
void request.say(__int(32) v) if(!busy) {
    itemSay = v;
    ...
}
```

## 2.4 `__connect`

[atomicc.connect]

The `__connect` statement allows exported interface declarations to be connected with imported interface references between objects within a module declaration.

```
connect-declaration:
    __connect identifier = identifier ;
```

[Example:

```
__interface ExampleRequest {
    void say(__int(32) v);
};

__module A {
    ExampleRequest callIn;
};

__module B {
    ExampleRequest *callOut;
};

__module C {
    A consumer;
    B producer;
    __connect producer.callOut = consumer.callIn;
};
```

— end example]

Comparison with BSV:

- The declaration for 'A' is just like BSV. In BSV, the declaration for B requires the interface be passed in as an interface parameter (forcing a textual ordering to the source code declaration sequence).
- In AtomicC, the interfaces are stitched together outside in any convenient sequence in a location where both the concrete instances for A and B are visible.

## 2.5 To export interfaces from contained objects

[atomicc.export]

```
forward-declaration:
    __forward identifier = identifier ;
```

[*Example:*

```
__module CWrapper {  
    A consumer;  
    ExampleRequest request = A.callIn;  
};
```

— *end example*]

CWrapper just forwards the interface 'request' down into the instance 'consumer'.

## 2.6 Syntax extension to C++

[`atomicc.classsyn`]

*atomicc-class-key:*

```
__interface  
__emodule  
__module
```

## 3 Statements

[stmt.stmt]

### 3.1 `__rule`

[atomicc.rule]

Rules specify the behavior with a design. A rule operates transactionally: when a rule's guard and the guards of all of its method invocations are satisfied, then it is ready to fire. It will fire on a clock cycle when it does not conflict with any higher priority rule. A rule executes atomically.

*rule-statement:*

`--rule identifier if-guardopt compound-statement`

[*Example:*

```
// default guard is true
__rule respond_rule {
    fifo->out.deq();
    ind->heard(fifo->out.first());
}
```

— *end example*]

## 4 Declarations

[dcl.dcl]

### 4.1 bitstring

[atomicc.bitdecl]

To declare a bitstring with or without sign extension.

*bit-type-specifier:*

```
--uint ( constant-expression )  
--int  ( constant-expression )
```

## 5 Expressions

[**expr**]

### 5.1 Built-in functions

[**atomicc.builtin**]

#### 5.1.1 `__bitsize`

[**atomicc.bitsize**]

Function to return size in bits of a type or variable.

#### 5.1.2 `__bitsubstr`

[**atomicc.bitsubstr**]

Function to return bit slice of bitstring

#### 5.1.3 `__bitconcat`

[**atomicc.bitconcat**]

Function to bitstring that is the concatenation of all of the member values of the call.

### 5.2 `__bit__cast`

[**atomicc.cast**]

This can now be used to cast any datatype to/from `__int(A)`, allowing operations to be performed on a bit level.

*atomic-bit-cast:*

```
__bit_cast < type-id > ( expression )
```

## 6 Modularization [atomicc.modularization]

### 6.1 Independant compilation of modules [atomicc.independant]

The design is separated into modules that can export and import interfaces to other modules. Each source language module compiles into a single verilog module. Modules are independantly compiled, depending only on the interface definitions for referenced modules. Referencing modules do not depend on the internal implementation of referenced modules, even if they textually exist in the same compilation unit. Scheduling of rules in a module is performed "inside out", with the resulting schedule dependancies written to a metadata file during compilation.

### 6.2 Execution control [atomicc.econtrol]

There are 2 common styles for communication of execution control information for a method:

- Asymmetric (ready/enable signalling) A method/rule is invoked by asserting the "enable" signal. This signal can only be asserted if the "ready" signal was valid, allowing the called module to restrict permissible execution sequences.
- Symmetric (ready/valid signalling) Both caller/callee have "able to be executed" signals. Execution is deemed to take place in each cycle where both "ready" (from the callee) and "valid" (from the caller) are asserted.

Bluespec uses the Asymmetric signalling style, collecting all scheduling control into a central location for analysis/generation. AtomicC uses the Symmetric signalling style, giving modules local control over their allowable execution patterns. Conflicts between local schedules for modules when they are connected together are detected by the linker.

### 6.3 Linking of groups of modules [atomicc.linker]

To verify that an instantiated group of modules has SC compliant execution characteristics, a linker is used to cross check information from the metadata files for each module.

### 6.4 Interfacing with verilog modules [atomicc.verilog]

To reference a module in verilog, fields can be declared in `__interface` items.

[Example:

```
__interface CNCONNECTNET2 {
    __input  __int(1)    IN1;
    __input  __int(1)    IN2;
    __output __int(1)    OUT1;
    __output __int(1)    OUT2;
};
__emodule CONNECTNET2 {
    CNCONNECTNET2 _;
};
```

— end example]

This will allow references/instantiation of an externally defined verilog module `CONNECTNET2` that has 2 'input' ports, `IN1` and `IN2`, as well as 2 'output' ports, `OUT1` and `OUT2`.

#### 6.4.1 Parameterized modules [atomicc.param]

Verilog modules that have module instantiation parameters can also be declared/referenced.

[Example:

```
__interface Mmcme2MMCME2_ADV {
    __parameter const char * BANDWIDTH;
    __parameter float      CLKFBOUT_MULT_F;
    __input  __uint(1)      CLKFBIN;
    __output __uint(1)      CLKFBOUT;
    __output __uint(1)      CLKFBOUTB;
```



```
};
__emodule MMCME2_ADV {
    Mmcme2MMCME2_ADV _;
};
```

— end example]

This example can be instantiated as:

[Example:

```
__module Test {
    ...
    MMCME2_ADV#(BANDWIDTH="WIDE",CLKFBOUT_MULT_F=1.0) mmcm;
    ...
    Test() {
        __rule initRule {
            mmcm._.CLKFBIN = mmcm._.CLKFBOUT;
        }
    }
}
```

— end example]

### 6.4.2 Reference syntax

[atomicc.refsyntax]

*atomicc-method-declaration:*

*attribute-specifier-seq<sub>opt</sub> pin-type<sub>opt</sub> decl-specifier-seq<sub>opt</sub> member-declarator-list<sub>opt</sub> ;*

*pin-type:*

```
__input
__output
__inout
__parameter
```

[Example:

```
__interface <interfaceName> {
    __input __uint(1) executeMethod;
    __input __uint(16) methodArgument;
    __output __uint(1) methodReady;
}
```

— end example]

For '\_\_\_parameter' items, supported datatypes include: "const char \*", "float", "int".

Factoring of interfaces into sub interfaces is also supported.

### 6.4.3 Clock/reset ports

[atomicc.clockReset]

Note that if interface port pins are declared in a module interface declaration, then CLK and nRST are `__not__` automatically declared/instantiated. (Since the user needs the flexibility to not require them when interfacing with legacy code).

Note that this also allows arbitrary signals (like the output of clock generators) to be passed to modules as CLK/nRST signals. (For Atomicc generated modules, please note that the default clock/reset signals for a module will always have these names)

### 6.4.4 Import tooling

[atomicc.itool]

There is a tool to automate the creation of AtomicC header files from verilog source files. [Example:

```
atomiccImport -o MMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 zynq.lib
atomiccImport -o VMMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 MMCME2_ADV.v
```

— end example]

## 7 Usage

[atomicc.usage]

### 7.1 Command line switches

[atomicc.command]

Command line switches...

### 7.2 debugging with printf

[atomicc.printf]

To aid debugging with a simulator, "printf" statements in \_\_\_module declarations are translated to "\$display" statements in the generated verilog. For debugging with synthesized hardware, "printf" statements are translated into indication packets sent through the NOC back to the software side host program. The format strings for the printf statements are placed into a generated file in generated/xxx.generated.printf along with a list of the bit lengths for each parameter to the printf.

To use the NOC printf:

- add the following line to the \_\_\_module being tested:

```
__printf;
```

- add a line similar to the following (with the 'xxx' replaced) to the test program:

```
atomiccPrintfInit("generated/rulec.generated.printf");
```

*printf-declaration:*

```
__printf ;
```

# Annex A (informative)

## Grammar summary

[agram]

<sup>1</sup> Summary of grammar.

### A.1 Classes

[agram.class]

*atomicc-method-definition:*  
    *decl-specifier-seq<sub>opt</sub> interface-qualifier-seq identifier parameters-and-qualifiers function-body*

*interface-qualifier:*  
    *identifier .*

*interface-qualifier-seq:*  
    *interface-qualifier*  
    *interface-qualifier-seq interface-qualifier*

*atomicc-function-body:*  
    *ctor-initializer<sub>opt</sub> if-guard<sub>opt</sub> compound-statement*

*if-guard:*  
    *if ( condition )*

*connect-declaration:*  
    *\_\_connect identifier = identifier ;*

*forward-declaration:*  
    *\_\_forward identifier = identifier ;*

*atomicc-class-key:*  
    *\_\_interface*  
    *\_\_emodule*  
    *\_\_module*

### A.2 Statements

[agram.stmt]

*rule-statement:*  
    *\_\_rule identifier if-guard<sub>opt</sub> compound-statement*

### A.3 Declarations

[agram.dcl]

*bit-type-specifier:*  
    *\_\_uint ( constant-expression )*  
    *\_\_int ( constant-expression )*

### A.4 Expressions

[agram.expr]

*atomic-bit-cast:*  
    *\_\_bit\_cast < type-id > ( expression )*

*atomicc-method-declaration:*  
    *attribute-specifier-seq<sub>opt</sub> pin-type<sub>opt</sub> decl-specifier-seq<sub>opt</sub> member-declarator-list<sub>opt</sub> ;*

*pin-type:*  
    *\_\_input*  
    *\_\_output*  
    *\_\_inout*  
    *\_\_parameter*

*printf-declaration:*  
    *\_\_printf ;*

# Annex B (informative)

## Scheduling examples [scheduleExample]

<sup>1</sup> Examples of how scheduling is computed

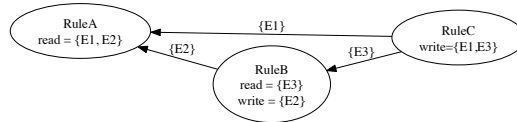


Figure 1 — Simple ordering example

A simple SC example is shown in Figure 1. There are 3 rules: RuleA, RuleB and RuleC. There are 3 state elements: E1, E2 and E3.

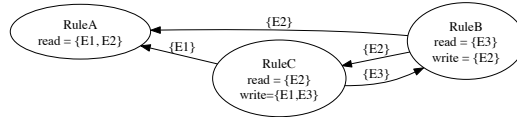


Figure 2 — non-SC ordering example

A non-SC example is shown in Figure 2.

# Annex C (informative)

## Grammar integration with C++ summary

### [gram]

<sup>1</sup> Summary of C++ grammar

#### C.1 Keywords

[gram.key]

<sup>1</sup> New context-dependent keywords are introduced into a program by `typedef` (??), `namespace` (??), `class` (Clause 2), `enumeration` (??), and `template` (??) declarations.

*typedef-name:*

*identifier*

*namespace-name:*

*identifier*

*namespace-alias*

*namespace-alias:*

*identifier*

*class-name:*

*identifier*

*simple-template-id*

*enum-name:*

*identifier*

*template-name:*

*identifier*

Note that a *typedef-name* naming a class is also a *class-name* (??).

#### C.2 Lexical conventions

[gram.lex]

*hex-quad:*

*hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit*

*universal-character-name:*

*\u hex-quad*

*\U hex-quad hex-quad*

*preprocessing-token:*

*header-name*

*identifier*

*pp-number*

*character-literal*

*user-defined-character-literal*

*string-literal*

*user-defined-string-literal*

*preprocessing-op-or-punc*

each non-white-space character that cannot be one of the above

*token:*

*identifier*

*keyword*

*literal*

*operator*

*punctuator*

*header-name:*

*< h-char-sequence >*

*" q-char-sequence "*

*h-char-sequence*:

*h-char*  
*h-char-sequence h-char*

*h-char*:

any member of the source character set except new-line and >

*q-char-sequence*:

*q-char*  
*q-char-sequence q-char*

*q-char*:

any member of the source character set except new-line and "

*pp-number*:

*digit*  
*. digit*  
*pp-number digit*  
*pp-number identifier-nondigit*  
*pp-number ' digit*  
*pp-number ' nondigit*  
*pp-number e sign*  
*pp-number E sign*  
*pp-number p sign*  
*pp-number P sign*  
*pp-number .*

*identifier*:

*identifier-nondigit*  
*identifier identifier-nondigit*  
*identifier digit*

*identifier-nondigit*:

*nondigit*  
*universal-character-name*

*nondigit*: one of

a b c d e f g h i j k l m  
n o p q r s t u v w x y z  
A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z \_

*digit*: one of

0 1 2 3 4 5 6 7 8 9

*preprocessing-op-or-punc*: one of

{	}	[	]	#	##	(	)	
<:	>:	<%	%>	%:	%::	;	:	...
new	delete	?	::	.	.*	->	->*	~
!	+	-	*	/	%	^	&	
=	+=	-=	*=	/=	%=	^=	&=	=
==	!=	<	>	<=	>=	<=>	&&	
<<	>>	<<=	>>=	++	--	,		
and	or	xor	not	bitand	bitor	compl		
and_eq	or_eq	xor_eq	not_eq					

*literal*:

*integer-literal*  
*character-literal*  
*floating-literal*  
*string-literal*  
*boolean-literal*  
*pointer-literal*  
*user-defined-literal*

*integer-literal*:

*binary-literal integer-suffix<sub>opt</sub>*  
*octal-literal integer-suffix<sub>opt</sub>*  
*decimal-literal integer-suffix<sub>opt</sub>*  
*hexadecimal-literal integer-suffix<sub>opt</sub>*

*binary-literal*:  
    **0b** *binary-digit*  
    **0B** *binary-digit*  
    *binary-literal* ' <sub>opt</sub> *binary-digit*

*octal-literal*:  
    **0**  
    *octal-literal* ' <sub>opt</sub> *octal-digit*

*decimal-literal*:  
    *nonzero-digit*  
    *decimal-literal* ' <sub>opt</sub> *digit*

*hexadecimal-literal*:  
    *hexadecimal-prefix* *hexadecimal-digit-sequence*

*binary-digit*: one of  
    **0 1**

*octal-digit*: one of  
    **0 1 2 3 4 5 6 7**

*nonzero-digit*: one of  
    **1 2 3 4 5 6 7 8 9**

*hexadecimal-prefix*: one of  
    **0x 0X**

*hexadecimal-digit-sequence*:  
    *hexadecimal-digit*  
    *hexadecimal-digit-sequence* ' <sub>opt</sub> *hexadecimal-digit*

*hexadecimal-digit*: one of  
    **0 1 2 3 4 5 6 7 8 9**  
    **a b c d e f**  
    **A B C D E F**

*integer-suffix*:  
    *unsigned-suffix* *long-suffix*<sub>opt</sub>  
    *unsigned-suffix* *long-long-suffix*<sub>opt</sub>  
    *long-suffix* *unsigned-suffix*<sub>opt</sub>  
    *long-long-suffix* *unsigned-suffix*<sub>opt</sub>

*unsigned-suffix*: one of  
    **u U**

*long-suffix*: one of  
    **l L**

*long-long-suffix*: one of  
    **ll LL**

*character-literal*:  
    *encoding-prefix*<sub>opt</sub> ' *c-char-sequence* '

*encoding-prefix*: one of  
    **u8 u U L**

*c-char-sequence*:  
    *c-char*  
    *c-char-sequence* *c-char*

*c-char*:  
    any member of the source character set except the single-quote ' , backslash \ , or new-line character  
    *escape-sequence*  
    *universal-character-name*

*escape-sequence*:  
    *simple-escape-sequence*  
    *octal-escape-sequence*  
    *hexadecimal-escape-sequence*

*simple-escape-sequence*: one of  
    **\ ' \" \? \\**  
    **\a \b \f \n \r \t \v**

*octal-escape-sequence*:

- \ *octal-digit*
- \ *octal-digit octal-digit*
- \ *octal-digit octal-digit octal-digit*

*hexadecimal-escape-sequence*:

- \x *hexadecimal-digit*
- hexadecimal-escape-sequence hexadecimal-digit*

*floating-literal*:

- decimal-floating-literal*
- hexadecimal-floating-literal*

*decimal-floating-literal*:

- fractional-constant exponent-part<sub>opt</sub> floating-suffix<sub>opt</sub>*
- digit-sequence exponent-part floating-suffix<sub>opt</sub>*

*hexadecimal-floating-literal*:

- hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix<sub>opt</sub>*
- hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffix<sub>opt</sub>*

*fractional-constant*:

- digit-sequence<sub>opt</sub> . digit-sequence*
- digit-sequence .*

*hexadecimal-fractional-constant*:

- hexadecimal-digit-sequence<sub>opt</sub> . hexadecimal-digit-sequence*
- hexadecimal-digit-sequence .*

*exponent-part*:

- e** *sign<sub>opt</sub> digit-sequence*
- E** *sign<sub>opt</sub> digit-sequence*

*binary-exponent-part*:

- p** *sign<sub>opt</sub> digit-sequence*
- P** *sign<sub>opt</sub> digit-sequence*

*sign*: one of

- +** **-**

*digit-sequence*:

- digit*
- digit-sequence ' <sub>opt</sub> digit*

*floating-suffix*: one of

- f** **l** **F** **L**

*string-literal*:

- encoding-prefix<sub>opt</sub> " s-char-sequence<sub>opt</sub> "*
- encoding-prefix<sub>opt</sub> R raw-string*

*s-char-sequence*:

- s-char*
- s-char-sequence s-char*

*s-char*:

- any member of the source character set except the double-quote **"**, backslash **\**, or new-line character
- escape-sequence*
- universal-character-name*

*raw-string*:

- " d-char-sequence<sub>opt</sub> ( r-char-sequence<sub>opt</sub> ) d-char-sequence<sub>opt</sub> "*

*r-char-sequence*:

- r-char*
- r-char-sequence r-char*

*r-char*:

- any member of the source character set, except a right parenthesis **)** followed by the initial *d-char-sequence* (which may be empty) followed by a double quote **"**.

*d-char-sequence*:

- d-char*
- d-char-sequence d-char*



*d-char*:  
 any member of the basic source character set except:  
 space, the left parenthesis (, the right parenthesis ), the backslash \, and the control characters  
 representing horizontal tab, vertical tab, form feed, and newline.

*boolean-literal*:  
**false**  
**true**

*pointer-literal*:  
**nullptr**

*user-defined-literal*:  
*user-defined-integer-literal*  
*user-defined-floating-literal*  
*user-defined-string-literal*  
*user-defined-character-literal*

*user-defined-integer-literal*:  
*decimal-literal* *ud-suffix*  
*octal-literal* *ud-suffix*  
*hexadecimal-literal* *ud-suffix*  
*binary-literal* *ud-suffix*

*user-defined-floating-literal*:  
*fractional-constant* *exponent-part*<sub>opt</sub> *ud-suffix*  
*digit-sequence* *exponent-part* *ud-suffix*  
*hexadecimal-prefix* *hexadecimal-fractional-constant* *binary-exponent-part* *ud-suffix*  
*hexadecimal-prefix* *hexadecimal-digit-sequence* *binary-exponent-part* *ud-suffix*

*user-defined-string-literal*:  
*string-literal* *ud-suffix*

*user-defined-character-literal*:  
*character-literal* *ud-suffix*

*ud-suffix*:  
*identifier*

### C.3 Basics

[gram.basic]

*translation-unit*:  
*declaration-seq*<sub>opt</sub>

### C.4 Expressions

[gram.expr]

*primary-expression*:  
*literal*  
**this**  
 ( *expression* )  
*id-expression*  
*lambda-expression*  
*fold-expression*  
*requires-expression*

*id-expression*:  
*unqualified-id*  
*qualified-id*

*unqualified-id*:  
*identifier*  
*operator-function-id*  
*conversion-function-id*  
*literal-operator-id*  
 ~ *class-name*  
 ~ *decltype-specifier*  
*template-id*

*qualified-id*:  
*nested-name-specifier* **template**<sub>opt</sub> *unqualified-id*

*nested-name-specifier*:

```

::
type-name ::
namespace-name ::
decltype-specifier ::
nested-name-specifier identifier ::
nested-name-specifier templateopt simple-template-id ::

```

*lambda-expression*:

```

lambda-introducer compound-statement
lambda-introducer lambda-declarator requires-clauseopt compound-statement
lambda-introducer < template-parameter-list > requires-clauseopt compound-statement
lambda-introducer < template-parameter-list > requires-clauseopt
    lambda-declarator requires-clauseopt compound-statement

```

*lambda-introducer*:

```

[ lambda-captureopt ]

```

*lambda-declarator*:

```

( parameter-declaration-clause ) decl-specifier-seqopt
    noexcept-specifieropt attribute-specifier-seqopt trailing-return-typeopt

```

*lambda-capture*:

```

capture-default
capture-list
capture-default , capture-list

```

*capture-default*:

```

&
=

```

*capture-list*:

```

capture
capture-list , capture

```

*capture*:

```

simple-capture ...opt
...opt init-capture

```

*simple-capture*:

```

identifier
& identifier
this
* this

```

*init-capture*:

```

identifier initializer
& identifier initializer

```

*fold-expression*:

```

( cast-expression fold-operator ... )
( ... fold-operator cast-expression )
( cast-expression fold-operator ... fold-operator cast-expression )

```

*fold-operator*: one of

```

+ - * / % ^ & | << >>
+= -= *= /= %= ^= &= |= <<= >>= =
== != < > <= >= && || , .* ->*

```

*requires-expression*:

```

requires requirement-parameter-listopt requirement-body

```

*requirement-parameter-list*:

```

( parameter-declaration-clauseopt )

```

*requirement-body*:

```

{ requirement-seq }

```

*requirement-seq*:

```

requirement
requirement-seq requirement

```

*requirement:*  
*simple-requirement*  
*type-requirement*  
*compound-requirement*  
*nested-requirement*

*simple-requirement:*  
*expression* ;

*type-requirement:*  
**typename** *nested-name-specifier*<sub>opt</sub> *type-name* ;

*compound-requirement:*  
 { *expression* } **noexcept**<sub>opt</sub> *return-type-requirement*<sub>opt</sub> ;

*return-type-requirement:*  
*trailing-return-type*  
 -> *cv-qualifier-seq*<sub>opt</sub> *constrained-parameter* *cv-qualifier-seq*<sub>opt</sub> *abstract-declarator*<sub>opt</sub>

*nested-requirement:*  
**requires** *constraint-expression* ;

*postfix-expression:*  
*primary-expression*  
*postfix-expression* [ *expr-or-braced-init-list* ]  
*postfix-expression* ( *expression-list*<sub>opt</sub> )  
*simple-type-specifier* ( *expression-list*<sub>opt</sub> )  
*typename-specifier* ( *expression-list*<sub>opt</sub> )  
*simple-type-specifier* *braced-init-list*  
*typename-specifier* *braced-init-list*  
*postfix-expression* . **template**<sub>opt</sub> *id-expression*  
*postfix-expression* -> **template**<sub>opt</sub> *id-expression*  
*postfix-expression* . *pseudo-destructor-name*  
*postfix-expression* -> *pseudo-destructor-name*  
*postfix-expression* ++  
*postfix-expression* --  
**dynamic\_cast** < *type-id* > ( *expression* )  
**static\_cast** < *type-id* > ( *expression* )  
**reinterpret\_cast** < *type-id* > ( *expression* )  
**const\_cast** < *type-id* > ( *expression* )  
*atomicc-bit-cast*  
**typeid** ( *expression* )  
**typeid** ( *type-id* )

*expression-list:*  
*initializer-list*

*pseudo-destructor-name:*  
*nested-name-specifier*<sub>opt</sub> *type-name* :: ~ *type-name*  
*nested-name-specifier* **template** *simple-template-id* :: ~ *type-name*  
 ~ *type-name*  
 ~ *decltype-specifier*

*unary-expression:*  
*postfix-expression*  
 ++ *cast-expression*  
 -- *cast-expression*  
*unary-operator* *cast-expression*  
**sizeof** *unary-expression*  
**sizeof** ( *type-id* )  
**sizeof** ... ( *identifier* )  
**alignof** ( *type-id* )  
*noexcept-expression*  
*new-expression*  
*delete-expression*

*unary-operator:* one of  
 \* & + - ! ~

■ ■

```

new-expression:
    ::opt new new-placementopt new-type-id new-initializeropt
    ::opt new new-placementopt ( type-id ) new-initializeropt

new-placement:
    ( expression-list )

new-type-id:
    type-specifier-seq new-declaratoropt

new-declarator:
    ptr-operator new-declaratoropt
    noptr-new-declarator

noptr-new-declarator:
    [ expression ] attribute-specifier-seqopt
    noptr-new-declarator [ constant-expression ] attribute-specifier-seqopt

new-initializer:
    ( expression-listopt )
    braced-init-list

delete-expression:
    ::opt delete cast-expression
    ::opt delete [ ] cast-expression

noexcept-expression:
    noexcept ( expression )

cast-expression:
    unary-expression
    ( type-id ) cast-expression

pm-expression:
    cast-expression
    pm-expression .* cast-expression
    pm-expression ->* cast-expression

multiplicative-expression:
    pm-expression
    multiplicative-expression * pm-expression
    multiplicative-expression / pm-expression
    multiplicative-expression % pm-expression

additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression

shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression

compare-expression:
    shift-expression
    compare-expression <=> shift-expression

relational-expression:
    compare-expression
    relational-expression < compare-expression
    relational-expression > compare-expression
    relational-expression <= compare-expression
    relational-expression >= compare-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression

and-expression:
    equality-expression
    and-expression & equality-expression

```

*exclusive-or-expression:*  
*and-expression*  
*exclusive-or-expression*  $\wedge$  *and-expression*

*inclusive-or-expression:*  
*exclusive-or-expression*  
*inclusive-or-expression*  $\mid$  *exclusive-or-expression*

*logical-and-expression:*  
*inclusive-or-expression*  
*logical-and-expression* **&&** *inclusive-or-expression*

*logical-or-expression:*  
*logical-and-expression*  
*logical-or-expression* **||** *logical-and-expression*

*conditional-expression:*  
*logical-or-expression*  
*logical-or-expression* **?** *expression* **:** *assignment-expression*

*throw-expression:*  
**throw** *assignment-expression*<sub>opt</sub>

*assignment-expression:*  
*conditional-expression*  
*logical-or-expression* *assignment-operator* *initializer-clause*  
*throw-expression*

*assignment-operator:* one of  
**=** **\*** **/** **%** **+=** **-=** **>>=** **<<=** **&=** **^=** **|=**

*expression:*  
*assignment-expression*  
*expression* **,** *assignment-expression*

*constant-expression:*  
*conditional-expression*

## C.5 Statements

[gram.stmt]

*statement:*  
*labeled-statement*  
*attribute-specifier-seq*<sub>opt</sub> *expression-statement*  
*attribute-specifier-seq*<sub>opt</sub> *compound-statement*  
*attribute-specifier-seq*<sub>opt</sub> *selection-statement*  
*attribute-specifier-seq*<sub>opt</sub> *iteration-statement*  
*attribute-specifier-seq*<sub>opt</sub> *jump-statement*  
*declaration-statement*  
*attribute-specifier-seq*<sub>opt</sub> *try-block*  
*rule-statement*

*init-statement:*  
*expression-statement*  
*simple-declaration*

*condition:*  
*expression*  
*attribute-specifier-seq*<sub>opt</sub> *decl-specifier-seq* *declarator* *brace-or-equal-initializer*

*labeled-statement:*  
*attribute-specifier-seq*<sub>opt</sub> *identifier* **:** *statement*  
*attribute-specifier-seq*<sub>opt</sub> **case** *constant-expression* **:** *statement*  
*attribute-specifier-seq*<sub>opt</sub> **default** **:** *statement*

*expression-statement:*  
*expression*<sub>opt</sub> **;**

*compound-statement:*  
**{** *statement-seq*<sub>opt</sub> **}**

*statement-seq:*  
*statement*  
*statement-seq* *statement*

■

*selection-statement:*  
 if constexpr<sub>opt</sub> ( *init-statement*<sub>opt</sub> *condition* ) *statement*  
 if constexpr<sub>opt</sub> ( *init-statement*<sub>opt</sub> *condition* ) *statement* else *statement*  
 switch ( *init-statement*<sub>opt</sub> *condition* ) *statement*

*iteration-statement:*  
 while ( *condition* ) *statement*  
 do *statement* while ( *expression* ) ;  
 for ( *init-statement* *condition*<sub>opt</sub> ; *expression*<sub>opt</sub> ) *statement*  
 for ( *init-statement*<sub>opt</sub> *for-range-declaration* : *for-range-initializer* ) *statement*

*for-range-declaration:*  
*attribute-specifier-seq*<sub>opt</sub> *decl-specifier-seq* *declarator*  
*attribute-specifier-seq*<sub>opt</sub> *decl-specifier-seq* *ref-qualifier*<sub>opt</sub> [ *identifier-list* ]

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

*jump-statement:*  
 break ;  
 continue ;  
 return *expr-or-braced-init-list*<sub>opt</sub> ;  
 goto *identifier* ;

*declaration-statement:*  
*block-declaration*

## C.6 Declarations

[gram.dcl]

*declaration-seq:*  
*declaration*  
*declaration-seq* *declaration*

*declaration:*  
*block-declaration*  
*nodeclspec-function-declaration*  
*function-definition*  
*template-declaration*  
*deduction-guide*  
*explicit-instantiation*  
*explicit-specialization*  
*linkage-specification*  
*namespace-definition*  
*empty-declaration*  
*attribute-declaration*

*block-declaration:*  
*simple-declaration*  
*asm-definition*  
*namespace-alias-definition*  
*using-declaration*  
*using-directive*  
*static\_assert-declaration*  
*alias-declaration*  
*opaque-enum-declaration*

*nodeclspec-function-declaration:*  
*attribute-specifier-seq*<sub>opt</sub> *declarator* ;

*alias-declaration:*  
 using *identifier* *attribute-specifier-seq*<sub>opt</sub> = *defining-type-id* ;

*simple-declaration:*  
*decl-specifier-seq* *init-declarator-list*<sub>opt</sub> ;  
*attribute-specifier-seq* *decl-specifier-seq* *init-declarator-list* ;  
*attribute-specifier-seq*<sub>opt</sub> *decl-specifier-seq* *ref-qualifier*<sub>opt</sub> [ *identifier-list* ] *initializer* ;

*static\_assert-declaration:*  
 static\_assert ( *constant-expression* ) ;  
 static\_assert ( *constant-expression* , *string-literal* ) ;

```

empty-declaration:
    ;
attribute-declaration:
    attribute-specifier-seq ;
decl-specifier:
    storage-class-specifier
    defining-type-specifier
    function-specifier
    friend
    typedef
    constexpr
    inline
decl-specifier-seq:
    decl-specifier attribute-specifier-seqopt
    decl-specifier decl-specifier-seq
storage-class-specifier:
    static
    thread_local
    extern
    mutable
function-specifier:
    virtual
    explicit-specifier
explicit-specifier:
    explicit ( constant-expression )
    explicit
typedef-name:
    identifier
type-specifier:
    simple-type-specifier
    elaborated-type-specifier
    typename-specifier
    cv-qualifier
type-specifier-seq:
    type-specifier attribute-specifier-seqopt
    type-specifier type-specifier-seq
defining-type-specifier:
    type-specifier
    class-specifier
    enum-specifier
defining-type-specifier-seq:
    defining-type-specifier attribute-specifier-seqopt
    defining-type-specifier defining-type-specifier-seq

```

*simple-type-specifier*:

- nested-name-specifier*<sub>opt</sub> *type-name*
- nested-name-specifier* **template** *simple-template-id*
- nested-name-specifier*<sub>opt</sub> *template-name*
- char**
- char16\_t**
- char32\_t**
- wchar\_t**
- bool**
- short**
- int**
- long**
- signed**
- unsigned**
- float**
- double**
- void**
- auto**
- decltype-specifier*
- bit-type-specifier*

*type-name*:

- class-name*
- enum-name*
- typedef-name*
- simple-template-id*

*decltype-specifier*:

- decltype** ( *expression* )
- decltype** ( **auto** )

*elaborated-type-specifier*:

- class-key* *attribute-specifier-seq*<sub>opt</sub> *nested-name-specifier*<sub>opt</sub> *identifier*
- class-key* *simple-template-id*
- class-key* *nested-name-specifier* **template**<sub>opt</sub> *simple-template-id*
- enum** *nested-name-specifier*<sub>opt</sub> *identifier*

*init-declarator-list*:

- init-declarator*
- init-declarator-list* , *init-declarator*

*init-declarator*:

- declarator* *initializer*<sub>opt</sub>
- declarator* *requires-clause*

*declarator*:

- ptr-declarator*
- noptr-declarator* *parameters-and-qualifiers* *trailing-return-type*

*ptr-declarator*:

- noptr-declarator*
- ptr-operator* *ptr-declarator*

*noptr-declarator*:

- declarator-id* *attribute-specifier-seq*<sub>opt</sub>
- noptr-declarator* *parameters-and-qualifiers*
- noptr-declarator* [ *constant-expression*<sub>opt</sub> ] *attribute-specifier-seq*<sub>opt</sub>
- ( *ptr-declarator* )

*parameters-and-qualifiers*:

- ( *parameter-declaration-clause* ) *cv-qualifier-seq*<sub>opt</sub>
- ref-qualifier*<sub>opt</sub> *noexcept-specifier*<sub>opt</sub> *attribute-specifier-seq*<sub>opt</sub>

*trailing-return-type*:

- > *type-id*

■ ■



*ptr-operator*:

- \* *attribute-specifier-seq<sub>opt</sub>* *cv-qualifier-seq<sub>opt</sub>*
- & *attribute-specifier-seq<sub>opt</sub>*
- && *attribute-specifier-seq<sub>opt</sub>*
- nested-name-specifier* \* *attribute-specifier-seq<sub>opt</sub>* *cv-qualifier-seq<sub>opt</sub>*

*cv-qualifier-seq*:

- cv-qualifier* *cv-qualifier-seq<sub>opt</sub>*

*cv-qualifier*:

- const
- volatile

*ref-qualifier*:

- &
- &&

*declarator-id*:

- ...<sub>opt</sub> *id-expression*

*type-id*:

- type-specifier-seq* *abstract-declarator<sub>opt</sub>*

*defining-type-id*:

- defining-type-specifier-seq* *abstract-declarator<sub>opt</sub>*

*abstract-declarator*:

- ptr-abstract-declarator*
- noptr-abstract-declarator<sub>opt</sub>* *parameters-and-qualifiers* *trailing-return-type*
- abstract-pack-declarator*

*ptr-abstract-declarator*:

- noptr-abstract-declarator*
- ptr-operator* *ptr-abstract-declarator<sub>opt</sub>*

*noptr-abstract-declarator*:

- noptr-abstract-declarator<sub>opt</sub>* *parameters-and-qualifiers*
- noptr-abstract-declarator<sub>opt</sub>* [ *constant-expression<sub>opt</sub>* ] *attribute-specifier-seq<sub>opt</sub>*
- ( *ptr-abstract-declarator* )

*abstract-pack-declarator*:

- noptr-abstract-pack-declarator*
- ptr-operator* *abstract-pack-declarator*

*noptr-abstract-pack-declarator*:

- noptr-abstract-pack-declarator* *parameters-and-qualifiers*
- noptr-abstract-pack-declarator* [ *constant-expression<sub>opt</sub>* ] *attribute-specifier-seq<sub>opt</sub>*
- ...

*parameter-declaration-clause*:

- parameter-declaration-list<sub>opt</sub>* ...<sub>opt</sub>
- parameter-declaration-list* , ...

*parameter-declaration-list*:

- parameter-declaration*
- parameter-declaration-list* , *parameter-declaration*

*parameter-declaration*:

- attribute-specifier-seq<sub>opt</sub>* *decl-specifier-seq* *declarator*
- attribute-specifier-seq<sub>opt</sub>* *decl-specifier-seq* *declarator* = *initializer-clause*
- attribute-specifier-seq<sub>opt</sub>* *decl-specifier-seq* *abstract-declarator<sub>opt</sub>*
- attribute-specifier-seq<sub>opt</sub>* *decl-specifier-seq* *abstract-declarator<sub>opt</sub>* = *initializer-clause*

*initializer*:

- brace-or-equal-initializer*
- ( *expression-list* )

*brace-or-equal-initializer*:

- = *initializer-clause*
- braced-init-list*

*initializer-clause*:

- assignment-expression*
- braced-init-list*

*braced-init-list*:

```

{ initializer-list ,opt }
{ designated-initializer-list ,opt }
{ }

```

*initializer-list*:

```

initializer-clause ...opt
initializer-list , initializer-clause ...opt

```

*designated-initializer-list*:

```

designated-initializer-clause
designated-initializer-list , designated-initializer-clause

```

*designated-initializer-clause*:

```

designator brace-or-equal-initializer

```

*designator*:

```

. identifier

```

*expr-or-braced-init-list*:

```

expression
braced-init-list

```

*function-definition*:

```

atomicc-method-definition
attribute-specifier-seqopt decl-specifier-seqopt declarator virt-specifier-seqopt function-body
attribute-specifier-seqopt decl-specifier-seqopt declarator requires-clause function-body

```

*function-body*:

```

atomicc-function-body
function-try-block
= default ;
= delete ;

```

*enum-name*:

```

identifier

```

*enum-specifier*:

```

enum-head { enumerator-listopt }
enum-head { enumerator-list , }

```

*enum-head*:

```

enum-key attribute-specifier-seqopt enum-head-nameopt enum-baseopt

```

*enum-head-name*:

```

nested-name-specifieropt identifier

```

*opaque-enum-declaration*:

```

enum-key attribute-specifier-seqopt nested-name-specifieropt identifier enum-baseopt ;

```

*enum-key*:

```

enum
enum class
enum struct

```

*enum-base*:

```

: type-specifier-seq

```

*enumerator-list*:

```

enumerator-definition
enumerator-list , enumerator-definition

```

*enumerator-definition*:

```

enumerator
enumerator = constant-expression

```

*enumerator*:

```

identifier attribute-specifier-seqopt

```

*namespace-name*:

```

identifier
namespace-alias

```

■ ■

■ ■

*namespace-definition:*  
     *named-namespace-definition*  
     *unnamed-namespace-definition*  
     *nested-namespace-definition*

*named-namespace-definition:*  
     **inline**<sub>opt</sub> **namespace** *attribute-specifier-seq*<sub>opt</sub> *identifier* { *namespace-body* }

*unnamed-namespace-definition:*  
     **inline**<sub>opt</sub> **namespace** *attribute-specifier-seq*<sub>opt</sub> { *namespace-body* }

*nested-namespace-definition:*  
     **namespace** *enclosing-namespace-specifier* :: *identifier* { *namespace-body* }

*enclosing-namespace-specifier:*  
     *identifier*  
     *enclosing-namespace-specifier* :: *identifier*

*namespace-body:*  
     *declaration-seq*<sub>opt</sub>

*namespace-alias:*  
     *identifier*

*namespace-alias-definition:*  
     **namespace** *identifier* = *qualified-namespace-specifier* ;

*qualified-namespace-specifier:*  
     *nested-name-specifier*<sub>opt</sub> *namespace-name*

*using-directive:*  
     *attribute-specifier-seq*<sub>opt</sub> **using** **namespace** *nested-name-specifier*<sub>opt</sub> *namespace-name* ;

*using-declaration:*  
     **using** *using-declarator-list* ;

*using-declarator-list:*  
     *using-declarator* ...<sub>opt</sub>  
     *using-declarator-list* , *using-declarator* ...<sub>opt</sub>

*using-declarator:*  
     **typename**<sub>opt</sub> *nested-name-specifier* *unqualified-id*

*asm-definition:*  
     *attribute-specifier-seq*<sub>opt</sub> **asm** ( *string-literal* ) ;

*linkage-specification:*  
     **extern** *string-literal* { *declaration-seq*<sub>opt</sub> }  
     **extern** *string-literal* *declaration*

*attribute-specifier-seq:*  
     *attribute-specifier-seq*<sub>opt</sub> *attribute-specifier*

*attribute-specifier:*  
     [ [ *attribute-using-prefix*<sub>opt</sub> *attribute-list* ] ]  
     *contract-attribute-specifier*  
     *alignment-specifier*

*alignment-specifier:*  
     **alignas** ( *type-id* ...<sub>opt</sub> )  
     **alignas** ( *constant-expression* ...<sub>opt</sub> )

*attribute-using-prefix:*  
     **using** *attribute-namespace* :

*attribute-list:*  
     *attribute*<sub>opt</sub>  
     *attribute-list* , *attribute*<sub>opt</sub>  
     *attribute* ...  
     *attribute-list* , *attribute* ...

*attribute:*  
     *attribute-token* *attribute-argument-clause*<sub>opt</sub>

*attribute-token:*  
     *identifier*  
     *attribute-scoped-token*

```

attribute-scoped-token:
    attribute-namespace :: identifier

attribute-namespace:
    identifier

attribute-argument-clause:
    ( balanced-token-seqopt )

balanced-token-seq:
    balanced-token
    balanced-token-seq balanced-token

balanced-token:
    ( balanced-token-seqopt )
    [ balanced-token-seqopt ]
    { balanced-token-seqopt }
    any token other than a parenthesis, a bracket, or a brace

contract-attribute-specifier:
    [ [ expects contract-levelopt : conditional-expression ] ]
    [ [ ensures contract-levelopt identifieropt : conditional-expression ] ]
    [ [ assert contract-levelopt : conditional-expression ] ]

contract-level:
    default
    audit
    axiom

```

## C.7 Classes

[gram.class]

```

class-name:
    identifier
    simple-template-id

class-specifier:
    class-head { member-specificationopt }

class-head:
    class-key attribute-specifier-seqopt class-head-name class-virt-specifieropt base-clauseopt
    class-key attribute-specifier-seqopt base-clauseopt

class-head-name:
    nested-name-specifieropt class-name

class-virt-specifier:
    final

class-key:
    class
    struct
    union
    atomicc-class-key ■■

member-specification:
    member-declaration member-specificationopt
    access-specifier : member-specificationopt

member-declaration:
    attribute-specifier-seqopt decl-specifier-seqopt member-declarator-listopt ;
    atomicc-method-declaration ■■
    function-definition
    using-declaration
    static_assert-declaration
    template-declaration
    deduction-guide
    alias-declaration
    connect-declaration ■■
    forward-declaration ■■
    printf-declaration ■■
    empty-declaration

```

*member-declarator-list*:

- member-declarator*
- member-declarator-list* , *member-declarator*

*member-declarator*:

- declarator virt-specifier-seq<sub>opt</sub> pure-specifier<sub>opt</sub>*
- declarator requires-clause*
- declarator brace-or-equal-initializer<sub>opt</sub>*
- identifier<sub>opt</sub> attribute-specifier-seq<sub>opt</sub> : constant-expression brace-or-equal-initializer<sub>opt</sub>*

*virt-specifier-seq*:

- virt-specifier*
- virt-specifier-seq virt-specifier*

*virt-specifier*:

- override**
- final**

*pure-specifier*:

- = 0**

*conversion-function-id*:

- operator** *conversion-type-id*

*conversion-type-id*:

- type-specifier-seq conversion-declarator<sub>opt</sub>*

*conversion-declarator*:

- ptr-operator conversion-declarator<sub>opt</sub>*

*base-clause*:

- :** *base-specifier-list*

*base-specifier-list*:

- base-specifier* ...<sub>opt</sub>
- base-specifier-list* , *base-specifier* ...<sub>opt</sub>

*base-specifier*:

- attribute-specifier-seq<sub>opt</sub> class-or-decltype*
- attribute-specifier-seq<sub>opt</sub> virtual access-specifier<sub>opt</sub> class-or-decltype*
- attribute-specifier-seq<sub>opt</sub> access-specifier virtual<sub>opt</sub> class-or-decltype*

*class-or-decltype*:

- nested-name-specifier<sub>opt</sub> class-name*
- nested-name-specifier* **template** *simple-template-id*
- decltype-specifier*

*access-specifier*:

- private**
- protected**
- public**

*ctor-initializer*:

- :** *mem-initializer-list*

*mem-initializer-list*:

- mem-initializer* ...<sub>opt</sub>
- mem-initializer-list* , *mem-initializer* ...<sub>opt</sub>

*mem-initializer*:

- mem-initializer-id* ( *expression-list<sub>opt</sub>* )
- mem-initializer-id* *braced-init-list*

*mem-initializer-id*:

- class-or-decltype*
- identifier*

## C.8 Overloading

[gram.over]

*operator-function-id*:

- operator** *operator*

*operator*: one of

<b>new</b>	<b>delete</b>	<b>new[]</b>	<b>delete[]</b>	<b>( )</b>	<b>[]</b>	<b>-&gt;</b>	<b>-&gt;*</b>	<b>~</b>
<b>!</b>	<b>+</b>	<b>-</b>	<b>*</b>	<b>/</b>	<b>%</b>	<b>^</b>	<b>&amp;</b>	<b> </b>
<b>=</b>	<b>+=</b>	<b>-=</b>	<b>*=</b>	<b>/=</b>	<b>%=</b>	<b>^=</b>	<b>&amp;=</b>	<b> =</b>
<b>==</b>	<b>!=</b>	<b>&lt;</b>	<b>&gt;</b>	<b>&lt;=</b>	<b>&gt;=</b>	<b>&lt;=&gt;</b>	<b>&amp;&amp;</b>	<b>  </b>
<b>&lt;&lt;</b>	<b>&gt;&gt;</b>	<b>&lt;&lt;=</b>	<b>&gt;&gt;=</b>	<b>++</b>	<b>--</b>	<b>,</b>		

*literal-operator-id*:

**operator** *string-literal identifier*  
**operator** *user-defined-string-literal*

## C.9 Templates

[gram.temp]

*template-declaration*:

*template-head declaration*  
*template-head concept-definition*

*template-head*:

**template** < *template-parameter-list* > *requires-clause*<sub>opt</sub>

*template-parameter-list*:

*template-parameter*  
*template-parameter-list* , *template-parameter*

*requires-clause*:

**requires** *constraint-logical-or-expression*

*constraint-logical-or-expression*:

*constraint-logical-and-expression*  
*constraint-logical-or-expression* || *constraint-logical-and-expression*

*constraint-logical-and-expression*:

*primary-expression*  
*constraint-logical-and-expression* && *primary-expression*

*concept-definition*:

**concept** *concept-name* = *constraint-expression* ;

*concept-name*:

*identifier*

*template-parameter*:

*type-parameter*  
*parameter-declaration*  
*constrained-parameter*

*type-parameter*:

*type-parameter-key* ...<sub>opt</sub> *identifier*<sub>opt</sub>  
*type-parameter-key* *identifier*<sub>opt</sub> = *type-id*  
*template-head* *type-parameter-key* ...<sub>opt</sub> *identifier*<sub>opt</sub>  
*template-head* *type-parameter-key* *identifier*<sub>opt</sub> = *id-expression*

*type-parameter-key*:

**class**  
**typename**

*constrained-parameter*:

*qualified-concept-name* ... *identifier*<sub>opt</sub>  
*qualified-concept-name* *identifier*<sub>opt</sub> *default-template-argument*<sub>opt</sub>

*qualified-concept-name*:

*nested-name-specifier*<sub>opt</sub> *concept-name*  
*nested-name-specifier*<sub>opt</sub> *partial-concept-id*

*partial-concept-id*:

*concept-name* < *template-argument-list*<sub>opt</sub> >

*default-template-argument*:

= *type-id*  
 = *id-expression*  
 = *initializer-clause*

*simple-template-id*:  
*template-name* < *template-argument-list*<sub>opt</sub> >

*template-id*:  
*simple-template-id*  
*operator-function-id* < *template-argument-list*<sub>opt</sub> >  
*literal-operator-id* < *template-argument-list*<sub>opt</sub> >

*template-name*:  
*identifier*

*template-argument-list*:  
*template-argument* ...<sub>opt</sub>  
*template-argument-list* , *template-argument* ...<sub>opt</sub>

*template-argument*:  
*constant-expression*  
*type-id*  
*id-expression*

*constraint-expression*:  
*logical-or-expression*

*typename-specifier*:  
**typename** *nested-name-specifier* *identifier*  
**typename** *nested-name-specifier* **template**<sub>opt</sub> *simple-template-id*

*explicit-instantiation*:  
**extern**<sub>opt</sub> **template** *declaration*

*explicit-specialization*:  
**template** < > *declaration*

*deduction-guide*:  
**explicit**<sub>opt</sub> *template-name* ( *parameter-declaration-clause* ) -> *simple-template-id* ;

## C.10 Exception handling

[gram.exception]

*try-block*:  
**try** *compound-statement* *handler-seq*

*function-try-block*:  
**try** *ctor-initializer*<sub>opt</sub> *compound-statement* *handler-seq*

*handler-seq*:  
*handler* *handler-seq*<sub>opt</sub>

*handler*:  
**catch** ( *exception-declaration* ) *compound-statement*

*exception-declaration*:  
*attribute-specifier-seq*<sub>opt</sub> *type-specifier-seq* *declarator*  
*attribute-specifier-seq*<sub>opt</sub> *type-specifier-seq* *abstract-declarator*<sub>opt</sub>  
...

*noexcept-specifier*:  
**noexcept** ( *constant-expression* )  
**noexcept**

## C.11 Preprocessing directives

[gram.cpp]

*preprocessing-file*:  
*group*<sub>opt</sub>

*group*:  
*group-part*  
*group* *group-part*

*group-part*:  
*control-line*  
*if-section*  
*text-line*  
**#** *conditionally-supported-directive*

*control-line:*

```

# include pp-tokens new-line
# define identifier replacement-list new-line
# define identifier lparen identifier-listopt ) replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new-line
# undef identifier new-line
# line pp-tokens new-line
# error pp-tokensopt new-line
# pragma pp-tokensopt new-line
# new-line

```

*if-section:*

```

if-group elif-groupsopt else-groupopt endif-line

```

*if-group:*

```

# if constant-expression new-line groupopt
# ifdef identifier new-line groupopt
# ifndef identifier new-line groupopt

```

*elif-groups:*

```

elif-group
elif-groups elif-group

```

*elif-group:*

```

# elif constant-expression new-line groupopt

```

*else-group:*

```

# else new-line groupopt

```

*endif-line:*

```

# endif new-line

```

*text-line:*

```

pp-tokensopt new-line

```

*conditionally-supported-directive:*

```

pp-tokens new-line

```

*lparen:*

```

a ( character not immediately preceded by white-space

```

*identifier-list:*

```

identifier
identifier-list , identifier

```

*replacement-list:*

```

pp-tokensopt

```

*pp-tokens:*

```

preprocessing-token
pp-tokens preprocessing-token

```

*new-line:*

```

the new-line character

```

*defined-macro-expression:*

```

defined identifier
defined ( identifier )

```

*h-preprocessing-token:*

```

any preprocessing-token other than >

```

*h-pp-tokens:*

```

h-preprocessing-token
h-pp-tokens h-preprocessing-token

```

*has-include-expression:*

```

__has_include ( < h-char-sequence > )
__has_include ( " q-char-sequence " )
__has_include ( string-literal )
__has_include ( < h-pp-tokens > )

```

*has-attribute-expression:*

```

__has_cpp_attribute ( pp-tokens )

```