

---

# Appendix A

---

## Grammar

*There is no worse danger for a teacher  
than to teach words instead of things.*  
– Marc Block

Introduction — keywords — lexical conventions — programs — expressions — statements — declarations — declarators — classes — derived classes — special member functions — overloading — templates — exception handling — preprocessing directives.

### A.1 Introduction

This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (§A.5, §A.7) must be applied to distinguish expressions from declarations. Moreover, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

The C and C++ standard grammars express very minor distinctions syntactically rather than through constraints. That gives precision, but it doesn't always improve readability.

### A.2 Keywords

New context-dependent keywords are introduced into a program by *typedef* (§4.9.7), namespace (§8.2), class (Chapter 10), enumeration (§4.8), and *template* (Chapter 13) declarations.

*typedef-name:*  
*identifier*

*namespace-name:*  
*original-namespace-name*  
*namespace-alias*

*original-namespace-name:*  
*identifier*

*namespace-alias:*  
*identifier*

*class-name:*  
*identifier*  
*template-id*

*enum-name:*  
*identifier*

*template-name:*  
*identifier*

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

Unless an identifier is explicitly declared to name a type, it is assumed to name something that is not a type (see §C.13.5).

The C++ keywords are:

C++ Keywords					
<i>and</i>	<i>and_eq</i>	<i>asm</i>	<i>auto</i>	<i>bitand</i>	<i>bitor</i>
<i>bool</i>	<i>break</i>	<i>case</i>	<i>catch</i>	<i>char</i>	<i>class</i>
<i>compl</i>	<i>const</i>	<i>const_cast</i>	<i>continue</i>	<i>default</i>	<i>delete</i>
<i>do</i>	<i>double</i>	<i>dynamic_cast</i>	<i>else</i>	<i>enum</i>	<i>explicit</i>
<i>export</i>	<i>extern</i>	<i>false</i>	<i>float</i>	<i>for</i>	<i>friend</i>
<i>goto</i>	<i>if</i>	<i>inline</i>	<i>int</i>	<i>long</i>	<i>mutable</i>
<i>namespace</i>	<i>new</i>	<i>not</i>	<i>not_eq</i>	<i>operator</i>	<i>or</i>
<i>or_eq</i>	<i>private</i>	<i>protected</i>	<i>public</i>	<i>register</i>	<i>reinterpret_cast</i>
<i>return</i>	<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>	<i>static_cast</i>
<i>struct</i>	<i>switch</i>	<i>template</i>	<i>this</i>	<i>throw</i>	<i>true</i>
<i>try</i>	<i>typedef</i>	<i>typeid</i>	<i>typename</i>	<i>union</i>	<i>unsigned</i>
<i>using</i>	<i>virtual</i>	<i>void</i>	<i>volatile</i>	<i>wchar_t</i>	<i>while</i>
<i>xor</i>	<i>xor_eq</i>				

### A.3 Lexical Conventions

The standard C and C++ grammars present lexical conventions as grammar productions. This adds precision but also makes for large grammars and doesn't always increase readability:

*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*  
*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 nondigit*  
*pp-number e sign*  
*pp-number E sign*  
*pp-number .*

*identifier:*

*nondigit*  
*identifier nondigit*  
*identifier digit*

*nondigit: one of*

*universal-character-name*

\_ 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	and_eq		bitand	
bitor		compl		not	or	not_eq		xor	or_eq		xor_eq	

*literal:*

*integer-literal*

*character-literal*

*floating-literal*

*string-literal*

*boolean-literal*

*integer-literal:*

*decimal-literal integer-suffix<sub>opt</sub>*

*octal-literal integer-suffix<sub>opt</sub>*

*hexadecimal-literal integer-suffix<sub>opt</sub>*

*decimal-literal:*

*nonzero-digit*

*decimal-literal digit*

*octal-literal:*

0

*octal-literal octal-digit*

*hexadecimal-literal:*

0x *hexadecimal-digit*

0X *hexadecimal-digit*

*hexadecimal-literal hexadecimal-digit*

*nonzero-digit: one of*

1 2 3 4 5 6 7 8 9

*octal-digit: one of*

0 1 2 3 4 5 6 7

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

*unsigned-suffix: one of*

*u U*

*long-suffix: one of*

*l L*

*character-literal:*

*' c-char-sequence '*  
*L' c-char-sequence '*

*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:*

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

*fractional-constant:*

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

*exponent-part:*

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

*sign: one of*

*+ -*

*digit-sequence*:  
     *digit*  
     *digit-sequence digit*

*floating-suffix*: one of  
     f   l   F   L

*string-literal*:  
     " *s-char-sequence*<sub>opt</sub> "  
     L " *s-char-sequence*<sub>opt</sub> "

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

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

*boolean-literal*:  
     false  
     true

## A.4 Programs

A program is a collection of *translation-units* combined through linking (§9.4). A *translation-unit*, often called a *source file*, is a sequence of *declarations*:

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

## A.5 Expressions

See §6.2.

*primary-expression*:  
     *literal*  
     this  
     :: *identifier*  
     :: *operator-function-id*  
     :: *qualified-id*  
     ( *expression* )  
     *id-expression*

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

```

id-expression:
    unqualified-id
    qualified-id

unqualified-id:
    identifier
    operator-function-id
    conversion-function-id
    ~ class-name
    template-id

qualified-id:
    nested-name-specifier templateopt unqualified-id

nested-name-specifier:
    class-or-namespace-name :: nested-name-specifieropt
    class-or-namespace-name :: template nested-name-specifier

class-or-namespace-name:
    class-name
    namespace-name

postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( expression-listopt )
    simple-type-specifier ( expression-listopt )
    typename ::opt nested-name-specifier identifier ( expression-listopt )
    typename ::opt nested-name-specifier templateopt template-id ( expression-listopt )
    postfix-expression . templateopt ::opt id-expression
    postfix-expression -> templateopt ::opt 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 )
    typeid ( expression )
    typeid ( type-id )

expression-list:
    assignment-expression
    expression-list , assignment-expression

pseudo-destructor-name:
    ::opt nested-name-specifieropt type-name :: ~ type-name
    ::opt nested-name-specifier template template-id :: ~ type-name
    ::opt nested-name-specifieropt ~ type-name

```

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

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

*new-expression:*  
 ::<sub>opt</sub> new *new-placement*<sub>opt</sub> *new-type-id* *new-initializer*<sub>opt</sub>  
 ::<sub>opt</sub> new *new-placement*<sub>opt</sub> ( *type-id* ) *new-initializer*<sub>opt</sub>

*new-placement:*  
 ( *expression-list* )

*new-type-id:*  
*type-specifier-seq* *new-declarator*<sub>opt</sub>

*new-declarator:*  
*ptr-operator* *new-declarator*<sub>opt</sub>  
*direct-new-declarator*

*direct-new-declarator:*  
 [ *expression* ]  
*direct-new-declarator* [ *constant-expression* ]

*new-initializer:*  
 ( *expression-list*<sub>opt</sub> )

*delete-expression:*  
 ::<sub>opt</sub> delete *cast-expression*  
 ::<sub>opt</sub> delete [ ] *cast-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*

*relational-expression:*  
*shift-expression*  
*relational-expression* < *shift-expression*  
*relational-expression* > *shift-expression*  
*relational-expression* <= *shift-expression*  
*relational-expression* >= *shift-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* ^ *and-expression*

*inclusive-or-expression:*  
*exclusive-or-expression*  
*inclusive-or-expression* | *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*

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

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

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

*constant-expression:*  
*conditional-expression*

Grammar ambiguities arise from the similarity between function style casts and declarations. For example:

```
int x;

void f( )
{
    char(x); // conversion of x to char or declaration of a char called x?
}
```

All such ambiguities are resolved to declarations. That is, “if it could possibly be interpreted as a declaration, it is a declaration.” For example:

```
T(a) -> m; // expression statement
T(a) ++; // expression statement

T(*e)(int(3)); // declaration
T(f)[4]; // declaration

T(a); // declaration
T(a)=m; // declaration
T(*b)(); // declaration
T(x),y,z=7; // declaration
```

This disambiguation is purely syntactic. The only information used for a name is whether it is known to be a name of a type or a name of a template. If that cannot be determined, the name is assumed to name something that isn’t a template or a type.

The construct **template** *unqualified-id* is used to state that the *unqualified-id* is the name of a template in a context in which that cannot be deduced (see §C.13.5).

## A.6 Statements

See §6.3.

*statement:*  
*labeled-statement*  
*expression-statement*  
*compound-statement*  
*selection-statement*  
*iteration-statement*  
*jump-statement*  
*declaration-statement*  
*try-block*

```

labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement

expression-statement:
    expressionopt ;

compound-statement:
    { statement-seqopt }

statement-seq:
    statement
    statement-seq statement

selection-statement:
    if ( condition ) statement
    if ( condition ) statement else statement
    switch ( condition ) statement

condition:
    expression
    type-specifier-seq declarator = assignment-expression

iteration-statement:
    while ( condition ) statement
    do statement while ( expression ) ;
    for ( for-init-statement conditionopt ; expressionopt ) statement

for-init-statement:
    expression-statement
    simple-declaration

jump-statement:
    break ;
    continue ;
    return expressionopt ;
    goto identifier ;

declaration-statement:
    block-declaration

```

## A.7 Declarations

The structure of declarations is described in Chapter 4, enumerations in §4.8, pointers and arrays in Chapter 5, functions in Chapter 7, namespaces in §8.2, linkage directives in §9.2.4, and storage classes in §10.4.

```

declaration-seq:
    declaration
    declaration-seq declaration

```

*declaration:*  
    *block-declaration*  
    *function-definition*  
    *template-declaration*  
    *explicit-instantiation*  
    *explicit-specialization*  
    *linkage-specification*  
    *namespace-definition*

*block-declaration:*  
    *simple-declaration*  
    *asm-definition*  
    *namespace-alias-definition*  
    *using-declaration*  
    *using-directive*

*simple-declaration:*  
    *decl-specifier-seq*<sub>opt</sub> *init-declarator-list*<sub>opt</sub> ;

*decl-specifier:*  
    *storage-class-specifier*  
    *type-specifier*  
    *function-specifier*  
    *friend*  
    *typedef*

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

*storage-class-specifier:*  
    *auto*  
    *register*  
    *static*  
    *extern*  
    *mutable*

*function-specifier:*  
    *inline*  
    *virtual*  
    *explicit*

*typedef-name:*  
    *identifier*

*type-specifier:*  
    *simple-type-specifier*  
    *class-specifier*  
    *enum-specifier*  
    *elaborated-type-specifier*  
    *cv-qualifier*

*simple-type-specifier:*

```

::opt nested-name-specifieropt type-name
::opt nested-name-specifier templateopt template-id
char
wchar_t
bool
short
int
long
signed
unsigned
float
double
void

```

*type-name:*

```

class-name
enum-name
typedef-name

```

*elaborated-type-specifier:*

```

class-key ::opt nested-name-specifieropt identifier
enum ::opt nested-name-specifieropt identifier
typename ::opt nested-name-specifier identifier
typename ::opt nested-name-specifier templateopt template-id

```

*enum-name:*

```

identifier

```

*enum-specifier:*

```

enum identifieropt { enumerator-listopt }

```

*enumerator-list:*

```

enumerator-definition
enumerator-list , enumerator-definition

```

*enumerator-definition:*

```

enumerator
enumerator = constant-expression

```

*enumerator:*

```

identifier

```

*namespace-name:*

```

original-namespace-name
namespace-alias

```

*original-namespace-name:*

```

identifier

```

*namespace-definition:*

```

named-namespace-definition
unnamed-namespace-definition

```

```

named-namespace-definition:
    original-namespace-definition
    extension-namespace-definition

original-namespace-definition:
    namespace identifier { namespace-body }

extension-namespace-definition:
    namespace original-namespace-name { namespace-body }

unnamed-namespace-definition:
    namespace { namespace-body }

namespace-body:
    declaration-seqopt

namespace-alias:
    identifier

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

qualified-namespace-specifier:
    ::opt nested-name-specifieropt namespace-name

using-declaration:
    using typenameopt ::opt nested-name-specifier unqualified-id ;
    using :: unqualified-id ;

using-directive:
    using namespace ::opt nested-name-specifieropt namespace-name ;

asm-definition:
    asm ( string-literal ) ;

linkage-specification:
    extern string-literal { declaration-seqopt }
    extern string-literal declaration

```

The grammar allows for arbitrary nesting of declarations. However, some semantic restrictions apply. For example, nested functions (functions defined local to other functions) are not allowed.

The list of specifiers that starts a declaration cannot be empty (there is no “implicit *int*,” §B.2) and consists of the longest possible sequence of specifiers. For example:

```

typedef int I;
void f(unsigned I) { /* ... */ }

```

Here, *f*( ) takes an unnamed *unsigned int*.

An *asm*( ) is an assembly code insert. Its meaning is implementation-defined, but the intent is for the string to be a piece of assembly code that will be inserted into the generated code at the place where it is specified.

Declaring a variable *register* is a hint to the compiler to optimize for frequent access; doing so is redundant with most modern compilers.

**A.7.1 Declarators**

See §4.9.1, Chapter 5 (pointers and arrays), §7.7 (pointers to functions), and §15.5 (pointers to members).

```

init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator

init-declarator:
    declarator initializeropt

declarator:
    direct-declarator
    ptr-operator declarator

direct-declarator:
    declarator-id
    direct-declarator ( parameter-declaration-clause ) cv-qualifier-seqopt exception-specificationopt
    direct-declarator [ constant-expressionopt ]
    ( declarator )

ptr-operator:
    * cv-qualifier-seqopt
    &
    ::opt nested-name-specifier * cv-qualifier-seqopt

cv-qualifier-seq:
    cv-qualifier cv-qualifier-seqopt

cv-qualifier:
    const
    volatile

declarator-id:
    ::opt id-expression
    ::opt nested-name-specifieropt type-name

type-id:
    type-specifier-seq abstract-declaratoropt

type-specifier-seq:
    type-specifier type-specifier-seqopt

abstract-declarator:
    ptr-operator abstract-declaratoropt
    direct-abstract-declarator

direct-abstract-declarator:
    direct-abstract-declaratoropt ( parameter-declaration-clause ) cv-qualifier-seqopt exception-specificationopt
    direct-abstract-declaratoropt [ constant-expressionopt ]
    ( abstract-declarator )

```

```

parameter-declaration-clause:
    parameter-declaration-listopt . . .opt
    parameter-declaration-list , . . .

parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list , parameter-declaration

parameter-declaration:
    decl-specifier-seq declarator
    decl-specifier-seq declarator = assignment-expression
    decl-specifier-seq abstract-declaratoropt
    decl-specifier-seq abstract-declaratoropt = assignment-expression

function-definition:
    decl-specifier-seqopt declarator ctor-initializeropt function-body
    decl-specifier-seqopt declarator function-try-block

function-body:
    compound-statement

initializer:
    = initializer-clause
    ( expression-list )

initializer-clause:
    assignment-expression
    { initializer-list ,opt }
    { }

initializer-list:
    initializer-clause
    initializer-list , initializer-clause

```

A *volatile* specifier is a hint to a compiler that an object may change its value in ways not specified by the language so that aggressive optimizations must be avoided. For example, a real time clock might be declared:

```
extern const volatile clock;
```

Two successive reads of *clock* might give different results.

## A.8 Classes

See Chapter 10.

```

class-name:
    identifier
    template-id

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

```



```

class-head:
    class-key identifieropt base-clauseopt
    class-key nested-name-specifier identifier base-clauseopt
    class-key nested-name-specifier template template-id base-clauseopt

class-key:
    class
    struct
    union

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

member-declaration:
    decl-specifier-seqopt member-declarator-listopt ;
    function-definition ;opt
    :opt nested-name-specifier templateopt unqualified-id ;
    using-declaration
    template-declaration

member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator

member-declarator:
    declarator pure-specifieropt
    declarator constant-initializeropt
    identifieropt : constant-expression

pure-specifier:
    = 0

constant-initializer:
    = constant-expression

```

To preserve C compatibility, a class and a non-class of the same name can be declared in the same scope (§5.7). For example:

```

struct stat { /* ... */ };
int stat(char* name, struct stat* buf);

```

In this case, the plain name (*stat*) is the name of the non-class. The class must be referred to using a *class-key* prefix.

Constant expressions are defined in §C.5.

### A.8.1 Derived Classes

See Chapter 12 and Chapter 15.

```

base-clause:
    : base-specifier-list

```

```

base-specifier-list:
    base-specifier
    base-specifier-list , base-specifier

base-specifier:
    ::opt nested-name-specifieropt class-name
    virtual access-specifieropt ::opt nested-name-specifieropt class-name
    access-specifier virtualopt ::opt nested-name-specifieropt class-name

access-specifier:
    private
    protected
    public

```

### A.8.2 Special Member Functions

See §11.4 (conversion operators), §10.4.6 (class member initialization), and §12.2.2 (base initialization).

```

conversion-function-id:
    operator conversion-type-id

conversion-type-id:
    type-specifier-seq conversion-declaratoropt

conversion-declarator:
    ptr-operator conversion-declaratoropt

ctor-initializer:
    : mem-initializer-list

mem-initializer-list:
    mem-initializer
    mem-initializer , mem-initializer-list

mem-initializer:
    mem-initializer-id ( expression-listopt )

mem-initializer-id:
    : :opt nested-name-specifieropt class-name
    identifier

```

### A.8.3 Overloading

See Chapter 11.

```

operator-function-id:
    operator operator

```

*operator: one of*

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

## A.9 Templates

Templates are explained in Chapter 13 and §C.13.

*template-declaration:*

`exportopt template < template-parameter-list > declaration`

*template-parameter-list:*

`template-parameter`  
`template-parameter-list , template-parameter`

*template-parameter:*

`type-parameter`  
`parameter-declaration`

*type-parameter:*

`class identifieropt`  
`class identifieropt = type-id`  
`typename identifieropt`  
`typename identifieropt = type-id`  
`template < template-parameter-list > class identifieropt`  
`template < template-parameter-list > class identifieropt = template-name`

*template-id:*

`template-name < template-argument-listopt >`

*template-name:*

`identifier`

*template-argument-list:*

`template-argument`  
`template-argument-list , template-argument`

*template-argument:*

`assignment-expression`  
`type-id`  
`template-name`

*explicit-instantiation:*

`template declaration`

*explicit-specialization:*

`template < > declaration`

The explicit template argument specification opens up the possibility of an obscure syntactic ambiguity. Consider:

```

void h ( )
{
    f<1>(0); // ambiguity: ((f<1)>(0) or (f<1>)(0) ?
            // resolution: f<1> is called with argument 0
}

```

The resolution is simple and effective: if *f* is a template name, *f*< is the beginning of a qualified template name and the subsequent tokens must be interpreted based on that; otherwise, < means less-than. Similarly, the first non-nested > terminates a template argument list. If a greater-than is needed, parentheses must be used:

```

f< a>b >(0); // syntax error
f< (a>b) >(0); // ok

```

A similar lexical ambiguity can occur when terminating >s get too close. For example:

```

list<vector<int>>> lv1; // syntax error: unexpected >> (right shift)
list< vector<int> > lv2; // correct: list of vectors

```

Note the space between the two >s; >> is the right-shift operator. That can be a real nuisance.

## A.10 Exception Handling

See §8.3 and Chapter 14.

```

try-block:
    try compound-statement handler-seq

function-try-block:
    try ctor-initializeropt function-body handler-seq

handler-seq:
    handler handler-seqopt

handler:
    catch ( exception-declaration ) compound-statement

exception-declaration:
    type-specifier-seq declarator
    type-specifier-seq abstract-declarator
    type-specifier-seq
    ...

throw-expression:
    throw assignment-expressionopt

exception-specification:
    throw ( type-id-listopt )

type-id-list:
    type-id
    type-id-list , type-id

```

## A.11 Preprocessing Directives

The preprocessor is a relatively unsophisticated macro processor that works primarily on lexical tokens rather than individual characters. In addition to the ability to define and use macros (§7.8), the preprocessor provides mechanisms for including text files and standard headers (§9.2.1) and conditional compilation based on macros (§9.3.3). For example:

```
#if OPT==4
#include "header4.h"
#elif 0<OPT
#include "someheader.h"
#else
#include <stdlib.h>
#endif
```

All preprocessor directives start with a #, which must be the first non-whitespace character on its line.

```
preprocessing-file:
    groupopt

group:
    group-part
    group group-part

group-part:
    pp-tokensopt new-line
    if-section
    control-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
```

*control-line:*

```
# include pp-tokens new-line
# define identifier replacement-list new-line
# define identifier lparen identifier-listopt ) 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
```

*lparen:*

*the left-parenthesis character without preceding white-space*

*replacement-list:*

*pp-tokens<sub>opt</sub>*

*pp-tokens:*

```
preprocessing-token
pp-tokens preprocessing-token
```

*new-line:*

*the new-line character*