Motor

motor.devel@gmail.com

Jul 07, 2023

GUIDES:

CHAPTER

ONE

MOTOR DOCUMENTATION

1.1 Using Motor and developing

1.1.1 Building Motor from source

1.1.1.1 Setting up a build environment

In order to build the engine from source, the following components are required:

- A host running Linux, macOS, FreeBSD, Solaris or Windows
- Python (version 2.7 or 3.4+)
- Flex and Bison
- **A C++ compiler in the path or installed in a standard location**
  - Clang version 2.9 or above
  - GCC version 3.4 or above
  - SunCC version 5.11 or above
  - Microsoft Visual Studio 2003 or above
  - Intel compiler, version 9 or above
- Many plugins require third party libraries in order to be enabled.

The build system will usually automatically detect compilers, Flex and Bison from the environment and/or the registry. Some of these components can be downloaded from the GitHub release page for certain platforms/architectures.

Since Motor is very modular, it does not strictly require any development library to be installed, at the cost of seeing the plugin disabled. The core engine depends only on standard or operating system libraries (threading, filesystem, etc.).

1.1.1.2 Building from the command line

The Motor uses WAF as the build system. The build happens in two phases:

1. The build environments will be created. This step detects all available compilers (including cross compilers) and creates toolchain environments for each detected target. This step needs to be executed only once for the host;

```
~/motor > python waf configure
```

```
Setting top to                           : ~/motor
Setting out to                           : ~/motor/bld/.waf
Checking for program 'flex'              : /usr/bin/flex
Checking for program 'bison'             : /usr/bin/bison
Looking for clang compilers              : done
Looking for msvc compilers               : done
Looking for gcc compilers                : done
Looking for suncc compilers              : done
Looking for intel compilers              : done
Looking for clang 10+                    : /usr/lib/llvm-12/bin/clang++
Checking for Android tools               : done
+ configuring for platform Linux
`- linux_gnu_amd64-clang_amd64-11.0.1  : gnu {unit tests}
`- linux_gnu_amd64-clang_amd64-12.0.0  : gnu {unit tests}
`- linux_gnu_amd64-gcc_amd64-9.3.0     : gnu {unit tests}
`- linux_gnu_amd64-gcc_amd64-10.2.1    : gnu {unit tests}
`- linux_gnu_amd64-suncc_amd64-5.15    : sun {unit tests}
`- linux_gnu_x86-clang_x86-11.0.1      : gnu {unit tests}
`- linux_gnu_x86-clang_x86-12.0.0      : gnu {unit tests}
`- linux_gnu_x86-gcc_x86-9.3.0         : gnu {unit tests}
`- linux_gnu_x86-gcc_x86-10.2.1        : gnu {unit tests}
`- linux_gnu_x86-suncc_x86-5.15        : sun {unit tests}
+ configuring for platform windows
`- mingw_amd64-gcc_amd64-10            : mingw
`- mingw_x86-gcc_x86-10                : mingw
+ configuring for platform android
`- android_nougat_7.0-clang-9.0.8      : androideabi
    `- arm64                             : androideabi
    `- armv7a                            : androideabi
    `- amd64                             : androideabi
    `- x86                               : androideabi
Looking for CUDA                         : 11.3
'configure' finished successfully (19.802s)
```

2. The build can be started for one of the detected target environments.

```
~/motor > python waf build:linux_gnu_amd64-clang_amd64-12.0.0:debug
```

```
setup not run; setting up the toolchain
'build:linux_gnu_amd64-clang_amd64-12.0.0:debug' finished successfully (0.004s)
Setting up environment                   : linux_gnu_amd64-clang_amd64-12.0.0
  compute.CUDA                           : cuda 11.3 [3.5, 3.7, 5.0, 5.2, 5.3, 6.0,␣
↪6.1, 6.2, 7.0, 7.2, 7.5, 8.0]
  compute.OpenCL                         : from pkg-config
  compute.cpu                            : vanilla, .sse3, .sse4, .avx, .avx2
  graphics.OpenGL                        : from pkg-config
  graphics.OpenGLES2                     : from pkg-config
  graphics.freetype                      : from pkg-config
  gui.gtk3                               : from pkg-config
  physics.bullet                         : from pkg-config
  scripting.lua                          : version -5.4 from pkg-config
  scripting.python                       : 2.7, 3.9
  system.X11                             : from pkg-config
```

(continues on next page)

(continued from previous page)

```
  system.zlib                           : from pkg-config
  system.minizip                        : from pkg-config
'setup:linux_gnu_amd64-clang_amd64-12.0.0' finished successfully (1.008s)
Generating LALR tables
Generating LALR tables
[  1/618] {bison}       motor.reflection.pp/valueparser.cc
[  2/618] {bison}       plugin.scripting.package.pp/packageparser.cc
[  3/618] {master}      motor.minitl/master-cxx-0.cc
[  4/618] {master}      motor.core/master-c-0.c
[  5/618] {master}      motor.core/master-cxx-2.cc
[  6/618] {master}      motor.core/master-cxx-1.cc
[  7/618] {master}      motor.core/master-cxx-0.cc
[  8/618] {master}      motor.network/master-cxx-0.cc
[  9/618] {master}      motor.filesystem/master-cxx-0.cc
[ 10/618] {master}      motor.introspect/master-cxx-0.cc
[ 11/618] {master}      motor.settings/master-cxx-0.cc
[ 12/618] {master}      motor.scheduler/master-cxx-0.cc
[ 13/618] {master}      motor.plugin/master-cxx-0.cc
[ 14/618] {master}      plugin.graphics.shadermodel1/master-cxx-1.cc
[ 15/618] {master}      plugin.graphics.shadermodel1/master-cxx-0.cc
[ 16/618] {master}      plugin.compute.cpu/master-cxx-0.cc
[ 17/618] {kernel_ast}  test.compute.unittests.pp/loop.ast
[ 18/618] {kernel_ast}  test.compute.unittests.pp/if.ast
[ 19/618] {master}      plugin.graphics.shadermodel2/master-cxx-0.cc
[ 20/618] {master}      plugin.scripting.pythonlib/master-cxx-1.cc
[ 21/618] {master}      plugin.scripting.pythonlib/master-cxx-0.cc
[ 22/618] {master}      plugin.graphics.shadermodel3/master-cxx-0.cc
[ 23/618] {master}      plugin.compute.opencl/master-cxx-0.cc
[ 24/618] {master}      plugin.compute.cuda/master-cxx-0.cc
[ 25/618] {master}      plugin.graphics.windowing/master-cxx-0.cc
[ 26/618] {master}      plugin.debug.runtime/master-cxx-0.cc
[ 27/618] {master}      plugin.graphics.shadermodel4/master-cxx-0.cc
[ 28/618] {clc64}       test.compute.unittests.statement.if.cl/if.64.ll
[ 29/618] {clc32}       test.compute.unittests.statement.if.cl/if.32.ll
[ 30/618] {clc64}       test.compute.unittests.statement.loop.cl/loop.64.ll
[ 31/618] {clc32}       test.compute.unittests.statement.loop.cl/loop.32.ll
[ 32/618] {nvcc}        test.compute.unittests.statement.if.cuda/if.fatbin
[ 33/618] {nvcc}        test.compute.unittests.statement.loop.cuda/loop.fatbin
[ 34/618] {master}      motor.launcher/master-cxx-0.cc
[ 35/618] {master}      plugin.debug.assert/master-cxx-0.cc
...
[612/618] {cxxshlib}    plugin.graphics.nullrender/libplugin.graphics.nullrender.so
[613/618] {dbg_copy}    plugin.graphics.shadermodel4/libplugin.graphics.
↪shadermodel4.so.debug
[614/618] {dbg_strip}   plugin.graphics.shadermodel4/libplugin.graphics.
↪shadermodel4.so
[615/618] {install}     plugin.graphics.shadermodel4/libplugin.graphics.
↪shadermodel4.so.debug
[616/618] {dbg_copy}    plugin.graphics.nullrender/libplugin.graphics.nullrender.so.
↪debug
[617/618] {dbg_strip}   plugin.graphics.nullrender/libplugin.graphics.nullrender.so
[618/618] {install}     plugin.graphics.nullrender/libplugin.graphics.nullrender.so.
```

(continues on next page)

(continued from previous page)

```
↪debug
'build:linux_gnu_amd64-clang_amd64-12.0.0:debug' finished successfully (4.422s)
```

1.1.2 Integrated Development Environment support

1.2 Extra tools

1.2.1 *Pyxx*, the C/C++/Objective-C/Objective-C++ parser

The architecture of Motor relies very much on inspecting type information in the runtime. The type database is used by the editor to create and manipulate game objects, and by the LUA and Python plugins to allow the scripts to interact with C++ objects.

The C++ type database is so important that a lot of effort was spent creating a good workflow to extract type information at compile time from certain header files and create the type database, with minimal input from the developer. Creating a class in C++ should be sufficient to make it available in the runtime through the reflection library, without additional boilerplate code. This means the C++ code is parsed in the build system and additional type information is compiled into the binary file and available at runtime.

Some tools already exist to parse C++ code and extract some information, such as Qt's MOC, Doxygen and Clang. Adding a dependency on such a tool is a difficult decision to make though. The tool might be more or less difficult to install on the hosts, or even not available on some hosts, or its API might not be stable across major revisions. For instance, Clang plugins to extract the AST have changed a lot between Clang 3.0 and Clang 13. Qt is installed pretty easily on Windows, but comes with many unnecessary dependencies, and the build system would need to locate the Qt SDK through probing the registry. Motor relies currently on very few mandatory dependencies (Python, flex, bison) and adding a dependency on Qt or Clang would create a lot of friction.

For these reasons, the decision was made to create a C++ parser in Python (which is already a dependency of the build system) to extract the type information from C++ headers. Building a complete C++ parser is a notoriously difficult task. But with the right parser generator and a deep analysis of the grammar, it is possible to create such a parser. The next sections describe the resulting parsing tool *Pyxx*, a frontend that can parse C, C++, Objective-C and Objective-C++ source files, and present all conflicts in the C++ grammar and their resolution in the annotated grammar that the parser generator uses in *Pyxx*.

Contents

  - selection-statement, *else*
  - elaborated-enum-specifier / opaque-enum-declaration
  - enum-base / *bitfield specifier*
  - base-clause / *bitfield specifier*
  - new-expression, *{ and* assignment-expression, *=*
  - nested-name-specifier, *::*
  - conversion-type-id, attribute-specifier-seq
  - explicit-specifier/noexcept-specifier, *(*
  - *Operators* *new*, *new[]*, *delete*, *delete[]*
  - *Operators* *delete*, *delete[]* *and* lambda-expression
  - conversion-declarator, *binary operators*
  - new-type-id, *binary operators*
  - *destructor, unary ~ operator*
  - constraint-logical-and-expression, *&&*
  - global-module-fragment
  - requirement-expression, nested-requirement
  - id-expression *in a* template-argument
  - export-declaration, *export* module-import-declaration
- *Dynamic conflict resolution*
  - class-name, enum-name, typedef-name *in a* type-name
  - declarator, decl-specifier
  - template-id, *<*
  - variadic-parameter-list, pack-declarator
  - primary-expression, pure-specifier / *bitfield declaration*, mem-initializer / compound-statement, brace-or-equal-initializer
  - typename-specifier, typename-parameter
  - initializer, parameters-and-qualifiers
  - trailing-return-type, abstract-declarator / parameters-and-qualifiers / initializer
  - *type template parameter*, *non-type class template parameter*
  - deduction-guide, template-declaration

1.2.1.1 Parsing methodology

Parsing stages

The C++ compilation process is traditionally done in several steps, some are common for compiling any programming language. Processing a C++ source file usually starts by the following steps:

- A preprocessing step is responsible for concatenating all source files into a single text that is sent to the next phase. Preprocessor tokens are evaluated, included files are replaced by their preprocessed content and macros are expanded.
- A lexical analysis is then responsible to break down the text into tokens. The tokenizing process cuts down the text into substrings, and classify those substrings into categories.
- A syntax analysis is then responsible for matching sequences of tokens with rules of a formal grammar. This step helps recognize longer sequences of tokens that match patterns defined by the grammar, and replace the whole sequence with a symbol summarizing the content of the sequence.
- After the input has been verified to conform to the grammar, a semantic analysis is then responsible for understanding and verifying the semantic information of the code. The semantic analyzer is responsible for determining the intention of the different parts of the program and verify that they are correct.

After the semantic information has been processed and verified, a compiler would now have enough information to move on to the code generation steps. These steps can vary from compiler to compiler, and are not presented here in detail. *Pyxx* stops its processing after the semantic analysis and uses the semantic information to generate run-time type information that can be compiled into the program. Moreover, *Pyxx* is built to not run the preprocessing step even if it results in loss of semantic information. *Pyxx* will only run the lexical, syntax and part of the semantic analysis.

While this theoretical breakdown of the first few stages of a compilation process seems to show that the steps can be run sequentially and independently, the convolution of the C++ grammar actually causes some of these steps to become interdependent. For instance, the C++ language was changed in 2011 to allow the >> token to be broken down into two > tokens to serve as template brackets. The operation of breaking down the token is dependent on the previous tokens encountered, and one way to implement this feature is to create a feedback channel between the syntactic analyzer which holds the information about template parameter lists, and the lexical analyzer which does not have the information about the current state of parsing.

Pyxx is the frontend tool that will drive the different analysis steps. Under the hood, it uses the *glrp* library which is a parser generator. *glrp* uses a formal grammar description with annotations in order to generate a state machine that will be used by the lexical and syntactic analyzers.

Conflict resolution

In several parts of the grammar, parser generators such as Bison will emit a warning that a conflict has been encountered. A conflict happens when two actions could legally be considered when encountering a token. There are several causes as to why conflicts happen, for instance:

- When a sequence of tokens could be reduced by the same rules, but in a different order. In many grammars, parsing of binary operations fall in this category:

```
int i = 1 + 2 * 3; // (1+2) * 3 or 1 + (2*3)?
```

  The grammar described in the C++ standard explicitly avoid all ambiguities in the expression rules in order to avoid generating such conflicts. It does not however disambiguate the dangling else construction.

  These conflicts are usually solved by assigning priority and associativity to the tokens involved in the conflict. Priority is involved when there is a conflict between two different tokens (`x + y * z`), while associativity is used to resolve the order of the rule reduction when the same operation is chained (`x + y + z`).

- When a sequence of tokens could legally be interpreted by two different rules. In this case the grammar is truly ambiguous, and an arbitrary choice is made to use one of the two rules. In C++, such an ambiguity exists between a cast expression and function declarations:

```
// A function named i takes an integer and returns an integer.
// Not an integer variable initialized with a cast expression.
int i(int(my_dbl));
```

- When the sequence of tokens are ambiguous due to a lack of semantic information. The most important example in C++ is the role that identifiers can play. An identifier can refer to a variable (for instance a value in an expression) or a type (for instance the type specifier in a declaration). When such a conflict is encountered, *Pyxx* shifts to a dynamic conflict resolution method by using a GLR parser implementation.
- When the sequence of tokens matches different rules up to a certain token, but the rule construction forces the parser to make a decision before that disambiguating token is encountered. In those cases, peeking at the next few tokens would lift the ambiguity. This is usually not an operations that parsers provide though.

  An example in the C++ grammar occurs around the definition of inline namespaces and inline declarations. An inline namespace is defined by the rule:

```
inline-namespace = inline namespace attribute-specifier-seq?  identifier {␣
↪namespace-body }
```

  While an inline declaration is defined by the rule:

```
inline-declaration = attribute-specifier-seq? inline decl-specifier-seq declarator ;
```

  When the parser encounters the `inline` keyword, it is already forced to make a decision about the optional *attribute-specifier-seq* symbol. If the parser could only see the inline namespace rule, it would shift the `inline` symbol onto the symbol stack and move on to the next token. If an inline declaration was the only rule though, the parser would shift two symbols: the empty `attribute-specifier-seq` followed by the `inline` token. When both rules exist, the parser is now finding a conflict; should it favor the inline namespace rule and push one symbol onto the stack, or the inline declaration and push two symbols onto the stack?

  As can be seen in this example, the token following `inline` would already be sufficient to resolve this conflict. But in a parser that uses only one token of lookahead, this token is not yet available and there is not enough information to resolve the conflict at the time it is encountered. Some conflicts occur within series of tokens that will eventually lead to a disambiguation; but there are cases where the series of tokens encountered before finding the disambiguation is not bounded, i.e. for any parser using k tokens in its lookahead list, one can find an input text where the disambiguation is found in token $k+1$ or beyond.

  A common solution employed to resolve those conflicts is to increase the overlap between the conflicting rules. For instance, one could change the grammar above to allow an optional *attribute-specifier-seq* in front of an inline namespace. There is now no conflict anymore; when the `inline` token is encountered, the parser can shift an empty `attribute-specifier-seq` symbol onto the stack, followed by the `inline` token, and move on to the next state. The parser generator does not have to make a choice at this stage and can still take both rules in consideration for now.

  Extending the rules to solve conflicts actually changes the language that is parsed; it creates rules that allow invalid code to be parsed successfully. It is however easy to add a verification during semantic analysis to reject the constructions that were accepted by the extended rule. This often allows for clearer error messages. In the original grammar, the following C++ code generates very different error messages in the GNU compiler GCC and in Clang:

```
[[a::b]] inline namespace A
{
};
```

```
$> g++ main.cc
main.cc:1:17: error: expected unqualified-id before ‘namespace’
    1 | [[a::b]] inline namespace A
      |                 ^~~~~~~~~

$> clang++ main.cc
main.cc:1:1: error: an attribute list cannot appear here
[[a::b]] inline namespace A
^~~~~~~~
```

Notice that Clang's error message was clearer than the one of GCC.

Tip: Rules that are similar do not confuse only the parsers; they also confuse the users of the language. Increasing the overlap between rules and shifting the analysis work to the semantic analyzer is not only a good way to solve an ambiguity for the parser, it allows to emit clear error messages to the users too.

Dynamic conflict resolution

Applying associativity and precedence rules to tokens, and relaxing grammar rules to make the grammar less ambiguous, are techniques that help solve conflicts directly when the parsing tables are constructed. When entering the conflict situation, the parser will always make the same choice, the choice that was implemented in the annotations of the grammar. There are cases however where the correct solution to the conflict depends on the context and cannot be decided during table generation. In that case, the conflict resolution has to be delayed until the semantic analysis pass, which has access to the semantic representation of the whole translation unit to make a decision.

For instance, the C++ grammar does not specify tightly what a valid declaration is, and a large amount of work is delegated to the semantic analyzer. In declarations, `{ }` can be interpreted as an initializer for a variable or constant, or as a code block for a function definition. But the declarator grammar rules do not differentiate between function declarations and variable declarations, so the syntactic analyzer delegates this interpretation to the semantic analyzer, which has sufficient context to differentiate a variable declaration from a function declaration.

Another undecidable conflict occurs when an identifier is encountered. The parser does not know yet what the identifier refers to, as this information is only computed during the semantic analysis pass. In the C++ 23 grammar, identifiers can name 8 different entities:

- an unqualified id for a variable, constant, parameter or method
- a namespace or namespace alias
- a class/struct/union name
- an enum name
- a typedef name
- a template name
- a concept name
- a bitfield name

In C, parsers rely sometimes on a lexer hack to decide what the identifier refers to. This is possible because in C, all typedefs need to be defined before they are used. An identifier that is not yet encountered before is considered to be a value type and not a typedef. In C++ however, it is possible to use a type before it has been declared:

```
class X
{
   void f()
   {
      Y y; // first use of Y here
   }

   class Y // declaration of Y here
   {
   };
};
```

The consequence is that a lexer hack cannot disambiguate all cases. Due to name resolution rules, it is even possible that a lexer hack would incorrectly find another entity declared before the point of usage, which would lead to a different interpretation of the sequence of tokens.

All parsers can handle conflict resolution at table generation time. But for conflicts that can't be resolved until parsing time, other strategies need to be applied. Here are some strategies that can help handle these conflicts:

- Create a breadth-first parser instead of depth-first. In this scenario, the parser starts recording the input stream when it encounters the { token, and pauses interpretation until it finds the matching } token. When the current scope is closed, the parser can run a partial semantic analysis then reopen all saved input streams and interpret them recursively.
- Modify the grammar to be even more permissive. In some cases this would be a perfectly valid strategy. In other cases though, the grammar would need a massive refactoring to accept all conflicting rules into a single, relaxed rule.
- Allow the parse method to maintain several valid states in parallel. This kind of parser is called a generalized parser and allows to explore several possibilities during the parsing phase, until the incorrect ones are eliminated or until all valid possibilities are merged into one. This tool quickly helps solve all conflicts where the ambiguity would eventually disappear after more tokens are parsed. When a true ambiguity is encountered, a merge strategy helps collapse all available possibilities into one to continue parsing. The collapsed possibilities are then unpacked by the semantic analyzer, and at this point it can make a decision about the correct interpretation.

For this tool, the choice was made to generate a GLR parser from LALR tables. The LALR tables allow precedence rules which helps resolving conflicts during table generation. For cases that cannot be decided during table generation, the GLR parser allows multiple options to be explored, and later collapsed into a single node in the syntax tree by merging the multiple options into one. The semantic analyzers removes invalid constructions at a later stage.

The library *glrp* was forked from open-source parser generators PLY and SLY. The table creation algorithm is the same but was enhanced with conflict resolution tools. The parsing method is rewritten as a GLR parser. The parser tool creates LALR tables from an annotated BNF grammar. *glrp* processes the C++ grammar declared in *Pyxx* and creates tables that *Pyxx* loads to create its state machine.

The grammars are extracted from the C 23 standard draft and the C++ 23 standard draft. The parser generator lists all conflicts in the standard grammar and drives annotations, either to prioritize rules (static conflict resolution) or to split parsing into branches and schedule merges when the branches reduce to the trunk (dynamic conflict resolution).

While the parser generator is very similar to Bison, it contains more debugging tools to analyze the grammar in order to provide better context for conflicts, more solutions to achieve conflict resolution, and a static analyzer of merge possibilities after splitting the parsing.

Conflict context and counter-examples

In order to apply any conflict resolution technique, it is necessary to understand the context in which a conflict occurs to apply the correct solution. The parser generator is usually giving very little context when a conflict is encountered. It does list all rules that are in conflict, and the token that is causing the conflict. This information alone is rarely sufficient in order to fully understand why the conflict occurs. Here is an example of the conflict report given for the dangling else construction:

```
shift-reduce conflict for token else in state 1750

Shift using rule  selection-statement -> if ( condition ) compound-statement  else␣
↪statement
Reduce using rule selection-statement -> if ( condition ) compound-statement
```

Just reading the report from the parser generator does not clarify why a conflict is happening here. A few years of experience in compiler construction already helps a lot in understanding what the parser generator is complaining about: it finds that there is a possibility to continue the *selection-statement* by shifting the `else` token, or to end the *selection-statement* here without an *else* clause because another rule allows a *selection-statement* to be followed by the `else` token.

People who have looked at a few grammars can now understand that there is a possibility that the *compound-statement* that is inside the *selection-statement* is a *selection-statement* itself. The consequence is that it is unclear which one the `else` token is for: is it introducing the else clause of the inner *selection-statement* or the else clause of the outer *selection-statement*?

This is a fairly simple, well known example of conflict, so it is relatively easy to discover what the parser generator is warning about, and how to fix it. The `else` token is not employed often in the grammar either, so exploring all its uses and finding the one that is causing the conflict is done quickly. But other instances of conflicts are much more difficult to understand without a deep knowledge of the grammar. Here is a second example:

```
shift-reduce conflict for token [[ in state 127

Shift using rule  attribute-specifier ->   [[ attribute-using-prefix? attribute-list ] ]
Reduce using rule attribute-specifier-seq? ->
```

In this instance, the parser generator is encountering a token that introduces an *attribute-specifier*. It finds two possibilities: start to match the *attribute-specifier* rule, or first reduce the current *attribute-specifier-seq*.

After some investigation, one starts to see that the problem here is that there is a rule somewhere in the grammar that allows two (optional) *attribute-specifier-seq* symbols to follow one another. When two sequences appear in a row, the parser cannot decide where the first sequence ends and where the second starts.

One would need to know the C++ grammar inside out to find which rule (or set of rules) cause this sequence to happen. There is no obvious point in the grammar where two *attribute-specifier-seq* seem to follow each other. Knowing the rules that allow this to happen is critical to apply the correct resolution though; the two possible resolutions are that the first *attribute-specifier-seq* takes precedence over the second one, or the other way around. In order to know what to prioritize, it is imperative to find out in the grammar where those two symbols are allowed to follow each other by looking at rules that can end with an *attribute-specifier-seq* and find a rule that continues with an *attribute-specifier-seq*.

As said above, a parser generator tool warns about around 3000 conflicts in the C++ grammar. Many of those conflicts are actually duplicates (luckily!) but they happen in different contexts, with different tokens. Investigating all 3000 conflicts and the context in which they occur would be an almost impossible task without assistance. Bison comes with an analysis tool that helps the investigation: it can optionally provide counter-examples for each conflict that is encountered. Unfortunately, Bison is attempting a deep analysis of the context to identify if the conflict is a true ambiguity in the grammar, or if it is eventually resolved after a few symbols have been shifted onto the stack. This feature is very useful when developing a grammar from the ground up; a computer language engineer would be able to always measure the effect of modifying the grammar. But it works best on a grammar that is currently conflict-free

or almost. When a grammar that has 3000 conflicts is used as an input, it takes several hours to provide the counter-examples.

Pyxx uses the *glrp* library as a parser generator. The *glrp* tool was built specifically to assist with debugging the C++ grammars; its table generation algorithm is the same as Bison and other tools, but it implements fast counter-example diagnostics to assist with debugging a very large grammar with many conflicts. Unlike Bison, when the parser generator encounters a conflict that causes a fork instead of a static resolution, the grammar can be annotated to indicate that the author expects a *split* action. The warning is then not emitted, since it is considered that the conflict is handled. Bison does not allow silencing specific warnings or counter-examples, so all conflicts that will leverage the GLR parser algorithm will continue to emit warnings. It is possible to tell Bison how many conflicts are expected in the grammar, but not specifically which conflicts. In *Pyxx*, To ensure that the author's intentions are clear, when a rule is annotated as causing a *split*, then all rules involved in the conflict need the same annotation, otherwise a warning will be emitted.

Counter-examples provided by *glrp* allow to find the origin of the conflicts described above. The dangling-else counter-example shows clearly that the conflict happens in nested *selection-statement*s:

```
shift using rule selection-statement -> if constexpr? ( condition ) statement  else␣
↪statement

| if constexpr? ( condition ) attribute-specifier-seq? if constexpr? ( condition )␣
↪statement  else statement
|                                                     selection-
↪statement──────────────────────────────────
|                             ␣
↪statement──────────────────────────────────────────────────────────────────────
| selection-
↪statement──────────────────────────────────────────────────────────────────────────────────────────────────────

reduce using rule selection-statement -> if constexpr? ( condition ) statement

| if constexpr? ( condition ) attribute-specifier-seq? if constexpr? ( condition )␣
↪statement  else statement
|                                                     selection-
↪statement──────────────────
|                             ␣
↪statement──────────────────────────────────────────────────
| selection-
↪statement──────────────────────────────────────────────────────────────────────────────────────────────────────
```

After filtering out some very similar contexts, we see the following two contexts where two *attribute-specifier-seq*s are allowed to follow each other:

```
reduce using rule attribute-specifier-seq? ->

|                           [[ attribute-using-prefix? attribute-list ] ] : constant-
↪expression brace-or-equal-initializer? ;
| attribute-specifier-seq? attribute-specifier──────────────────────────
|                           attribute-specifier-seq──────────────────────
|                           attribute-specifier-seq?─────────────────────
|                           member-
↪declarator──────────────────────────────────────────────────────────────────────────
|                           member-declarator-list?
↪──────────────────────────────────────────────────────────────
| member-
```

(continues on next page)

(continued from previous page)

```
↪declaration────────────────────────────────────────────────────────────────────────────────
├
│ operator type-specifier-seq *                                                      ␣
↪              [[ attribute-using-prefix? attribute-list ] ]
│                               attribute-specifier-seq? cv-qualifier-seq? conversion-
↪declarator? attribute-specifier────────────────────────
│                             ptr-operator──────────────────────────────────         ␣
↪             attribute-specifier-seq────────────────────
│                             conversion-declarator?
↪───────────────────────────────────────────────── attribute-specifier-seq?
↪───────────────────
│         conversion-type-
↪id────────────────────────────────────────────────────────────────────────
│ conversion-function-
↪id─────────────────────────────────────────────────────────────────────────────
│ unqualified-
↪id─────────────────────────────────────────────────────────────────────────────────────
│ id-
↪expression──────────────────────────────────────────────────────────────────────────────────────
│ declarator-
↪id──────────────────────────────────────────────────────────────────────────────────────────
│ noptr-
↪declarator─────────────────────────────────────────────────────────────────────────────────────────

shift using rule attribute-specifier ->   [[ attribute-using-prefix? attribute-list ] ]

│  [[ attribute-using-prefix? attribute-list ] ] member-declarator-list? ;
  attribute-specifier──────────────────────────
  attribute-specifier-seq──────────────────────
  attribute-specifier-seq?─────────────────────
  member-declaration─────────────────────────────────────────────────────
├
│ operator type-specifier-seq *  [[ attribute-using-prefix? attribute-list ] ] cv-
↪qualifier-seq? conversion-declarator? attribute-specifier-seq?
│                               attribute-specifier──────────────────────────
│                               attribute-specifier-seq──────────────────────
│                               attribute-specifier-seq?─────────────────────
│                             ptr-
↪operator─────────────────────────────────────────────────────
│                             conversion-declarator?
↪──────────────────────────────────────────────────────────────────
│         conversion-type-
↪id──────────────────────────────────────────────────────────────────────────────────────────
│ conversion-function-
↪id───────────────────────────────────────────────────────────────────────────────────────────────
│ unqualified-
↪id─────────────────────────────────────────────────────────────────────────────────────────────────────
│ id-
↪expression─────────────────────────────────────────────────────────────────────────────────────────────
│ declarator-
↪id─────────────────────────────────────────────────────────────────────────────────────────────────────
│ noptr-
```

(continues on next page)

(continued from previous page)

```
↪declarator
```

With this information, it is now possible to decide which priority to apply on the rules in order to solve the conflict.

1.2.1.2 Rule tweaks

This section lists all amendments to the rules that were applied in order to simplify the grammar. In all cases, the resulting grammar is either equivalent or more permissive than the official C++ grammar described in the C++ standard. The rule modifications allow conflicts to be simplified away without applying any priority changes or dynamic conflict resolution.

In the cases where the grammar is expanded to accept more, extra work is shifted towards the semantic analyzer to properly log errors in case an invalid construct was incorrectly accepted during the parsing phase. This is not necessarily a downside though, as the semantic analyzer can usually offer much better error messages than the parser.

Allowing extra *attribute-specifier-seq?*

The position of *attribute-specifier*s in the grammar is not consistent; for instance, some declarations can start with an *attribute-specifier* but namespace may not.

This causes two kinds of conflicts:

- Declaration rules that do not start with the optional *attribute-specifier-seq?* conflict with declaration rules that are allowed to start with it
- In contexts that allow both a declaration or an expression (*init-statement*, *compound-statement*, *template-parameter* for instance) expression rules (that do not start with the optional *attribute-specifier-seq?*) are in conflict with declaration rules.

To help the parser generator, the grammar was modified to allow *attribute-specifier-seq?* symbols in the following rules:

```
alias-declaration:
   attribute-specifier-seq? "using" "identifier" attribute-specifier-seq? "="␣
↪defining-type-id ";"'

opaque-enum-declaration:
   attribute-specifier-seq? enum-key attribute-specifier-seq? enum-head-name enum-base? ;

linkage-specification:
   attribute-specifier-seq? extern string-literal { declaration-seq? }
   attribute-specifier-seq? extern string-literal declaration


using-enum-declaration:
   attribute-specifier-seq? using elaborated-enum-specifier ;

c++ 98-14:
using-declaration:
   attribute-specifier-seq? using typename? nested-name-specifier unqualified-id ;
   attribute-specifier-seq? using :: unqualified-id ;

c++ 17:
```

```
using-declaration:
   attribute-specifier-seq? using using-declarator-list ;

named-namespace-definition:
   attribute-specifier-seq? inline? namespace attribute-specifier-seq? identifier {␣
↪namespace-body }

unnamed-namespace-definition:
   attribute-specifier-seq? inline? namespace attribute-specifier-seq? { namespace-body }

nested-namespace-definition:
   attribute-specifier-seq? namespace enclosing-namespace-specifier :: inline?␣
↪identifier { namespace-body }

namespace-alias-definition:
   attribute-specifier-seq? namespace identifier = qualified-namespace-specifier ;

explicit-specialization:
   attribute-specifier-seq? template < > declaration

explicit-instantiation:
   attribute-specifier-seq? extern? template declaration

deduction-guide:
   attribute-specifier-seq? explicit-specifier? template-name ( parameter-declaration-clause␣
↪) -> simple-template-id ;

type-parameter:
   attribute-specifier-seq? type-parameter-key ...? identifier?
   attribute-specifier-seq? type-parameter-key identifier? = type-id
   attribute-specifier-seq? type-constraint ...? identifier?
   attribute-specifier-seq? type-constraint identifier? = type-id
   attribute-specifier-seq? template-head type-parameter-key ...? identifier?
   attribute-specifier-seq? template-head type-parameter-key identifier? = id-expression

init-statement:
   attribute-specifier-seq? expression-statement

condition:
   attribute-specifier-seq? expression
```

The semantic analyzer is responsible for raising warnings in a later stage.

class-head / *elaborated-type-specifier* and *enum-head* / *elaborated-enum-specifier*

*elaborated-type-specifier*s and *defining-type-specifier*s are sometimes both accepted as *type-specifier*s. This leads to problems as the rules are very similar and the parsers needs many tokens to disambiguate the two.

The following rules have a lot of overlap but optional symbols force the parser generator to make an early choice:

```
elaborated-type-specifier:
   class-key attribute-specifier-seq? nested-name-specifier? identifier
   class-key simple-template-id
```

(continues on next page)

(continued from previous page)

```
   class-key nested-name-specifier template? simple-template-id
   ...

class-head:
   class-key attribute-specifier-seq? class-head-name class-virt-specifier? base-clause?
   class-key attribute-specifier-seq? base-clause?

class-head-name:
   nested-name-specifier? class-name
```

The *elaborated-enum-specifier* syntax is close to, but different from the *enum-head*.

```
elaborated-enum-specifier:
   enum nested-name-specifier? identifier

enum-head:
   enum-key attribute-specifier-seq? enum-head-name? enum-base?
```

The conflicts disappear if the *elaborated-type-specifier* rules are amended to accept the same syntax as the *class-head*.

```
elaborated-type-specifier:
   elaborated-type-specifier : class-key attribute-specifier-seq? class-head-name

class-head:
   class-key attribute-specifier-seq? class-head-name class-virt-specifier? base-clause?
   class-key attribute-specifier-seq? base-clause?

class-head-name:
   nested-name-specifier? class-name
```

Similarly, the *elaborated-enum-specifier* rule can be amended to accept the same syntax as *enum-head*.

```
elaborated-enum-specifier:
   enum-key attribute-specifier-seq? enum-head-name

enum-head:
   enum-key attribute-specifier-seq? enum-head-name? enum-base?
```

1.2.1.3 Static conflict resolution

This section lists the conflicts that are resolved through making explicit choices in the grammar at the point the token is encountered (i.e. without additional lookahead). The choice is specified by annotating the grammar with priority attributes.

template-parameter-list/*template-argument-list*, >

In a *template-parameter-list* or *template-argument-list*, the C++ parser encounters an expression (in the case of a *template-argument-list* as a *constant-expression*, and in the case of a *template-parameter-list* as a default value of a *template-parameter*). The > token could be interpreted as starting a *relational-expression*, or could be the closing bracket of the *template-parameter-list* or *template-argument-list*.

```
equality-expression[split:] -> relational-expression
relational-expression -> relational-expression  > compare-expression
```

In a *template-parameter-list*:

```
reduce using rule equality-expression[split:] -> relational-expression

| template < attribute-specifier-seq? decl-specifier-seq abstract-declarator? =␣
↪relational-expression    >
|                                                                              equality-
↪expression——
|                                                                              and-
↪expression————————
|                                                                             ␣
↪exclusive-or-expression
|                                                                             ␣
↪inclusive-or-expression
|                                                                              logical-
↪and-expression—
|                                                                              logical-
↪or-expression——
|                                                                             ␣
↪conditional-expression—
|                                                                             ␣
↪assignment-expression——
|                                                                             ␣
↪initializer-clause———————
|            parameter-
↪declaration———————————————————————————————————————————————————————————————
|            template-
↪parameter—————————————————————————————————————————————————————————————————
|            template-parameter-
↪list——————————————————————————————————————————————————————
| template-
↪head———————————————————————————————————————————————————————————————————————————————————————————————————————————————————————————————————————

shift using rule relational-expression -> relational-expression  > compare-expression

| template < attribute-specifier-seq? decl-specifier-seq abstract-declarator? =␣
↪relational-expression  > compare-expression assignment-operator initializer-clause >
|                                                                             ␣
↪relational-expression————————————————————
|                                                                              equality-
↪expression————————————————————
|                                                                              and-
↪expression——————————————————————————————
```

(continues on next page)

(continued from previous page)

```
│                                                                                 ␣
↪exclusive-or-expression──────────────────
│                                                                                 ␣
↪inclusive-or-expression──────────────────
│                                                                                  logical-
↪and-expression──────────────────
│                                                                                  logical-
↪or-expression──────────────────
│                                                                                 ␣
↪assignment-expression──────────────────────────────────────────────────────────
│                                                                                 ␣
↪initializer-clause─────────────────────────────────────────────────────────────
│            parameter-
↪declaration──────────────────────────────────────────────────────────────────────
│            template-
↪parameter──────────────────────────────────────────────────────────────────────
│            template-parameter-
↪list──────────────────────────────────────────────────────────────────────
 template-
↪head──────────────────────────────────────────────────────────────────────
```

In a *template-argument-list*:

```
reduce using rule equality-expression[split:] -> relational-expression

 template-name < relational-expression    >
                 equality-expression──
                 and-expression───────
                 exclusive-or-expression
                 inclusive-or-expression
                 logical-and-expression─
                 logical-or-expression──
                 conditional-expression─
                 constant-expression────
                 template-argument──────
                 template-argument-list?
 simple-template-id──────────────────────

shift using rule relational-expression -> relational-expression  > compare-expression

 template-name < relational-expression  > compare-expression >
                 relational-expression──────────────────────
                 equality-expression────────────────────────
                 and-expression─────────────────────────────
                 exclusive-or-expression────────────────────
                 inclusive-or-expression────────────────────
                 logical-and-expression─────────────────────
                 logical-or-expression──────────────────────
                 conditional-expression─────────────────────
                 constant-expression────────────────────────
                 template-argument──────────────────────────
```

(continues on next page)

(continued from previous page)

```
                  template-argument-list?──────────────────
simple-template-id──────────────────────────────────────────
```

The C++ standard disambiguates the *template-parameter-list* conflict in section 13.2.16[1]. It also disambiguates the *template-argument-list* conflict in section 13.3.4[2]. The resolution is to favor a reduce of the *relational-expression* over a shift of the > symbol.

selection-statement, `else`

When parsing nested *selection-statement*s, a conflict arises when the `else` token is encountered. In the sequence of symbols shown in the counterexample, it is not specified in the grammar if the `else` keyword opens the *else clause* of the rightmost *selection-statement*, or if it reduces the rightmost *selection-statement* and continues the leftmost *selection-statement*.

```
selection-statement -> if constexpr? ( init-statement? condition ) statement  else␣
↪statement
selection-statement -> if constexpr? ( init-statement? condition ) statement
```

```
shift using rule selection-statement -> if constexpr ( init-statement condition )␣
↪statement  else statement

 if constexpr? ( init-statement? condition ) if constexpr? ( init-statement? condition␣
↪) statement  else statement
                                          selection-
↪statement──────────────────────────────────────────────
                                         ␣
↪statement──────────────────────────────────────────────────────────
 selection-
```

(continues on next page)

[1] When parsing a default *template-argument* for a non-type *template-parameter*, the first non-nested > is taken as the end of the *template-parameter-list* rather than a greater-than operator.

[Example 9:

```
template<int i = 3 > 4 >        // syntax error
class X { /* ... */ };

template<int i = (3 > 4) >      // OK
class Y { /* ... */ };
```

— end example]

[2] When parsing a *template-argument-list*, the first non-nested > is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the *template-argument-list* and completes the *template-id*.

[Note 2: The second > token produced by this replacement rule can terminate an enclosing *template-id* construct or it can be part of a different construct (e.g., a cast). — end note]

[Example 2:

```
template<int i> class X { /* ... */ };

X< 1>2 > x1;                        // syntax error
X<(1>2)> x2;                        // OK

template<class T> class Y { /* ... */ };
Y<X<1>> x3;                         // OK, same as Y<X<1> > x3;
Y<X<6>>1>> x4;                      // syntax error
Y<X<(6>>1)>> x5;                    // OK
```

— end example]

(continued from previous page)

```
↪statement──────────────────────────────────────────────────────────────────────────────────────────────────────────────

reduce using rule selection-statement -> if constexpr ( init-statement condition )␣
↪statement

│ if constexpr? ( init-statement? condition ) if constexpr? ( init-statement? condition␣
↪) statement  else statement
│                                           selection-
↪statement──────────────────────────────
│                                           ␣
↪statement──────────────────────────────────────────
│ selection-
↪statement──────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

```
selection-statement -> if !? consteval compound-statement  else statement
selection-statement -> if !? consteval compound-statement
```

```
shift using rule selection-statement -> if consteval compound-statement  else statement

│ if constexpr? ( init-statement? condition ) if !? consteval compound-statement  else␣
↪statement
│                                           selection-
↪statement──────────────────────────────
│                                           ␣
↪statement──────────────────────────────────────────
│ selection-
↪statement──────────────────────────────────────────────────────────────────────────────

reduce using rule selection-statement -> if !? consteval compound-statement

│ if constexpr? ( init-statement? condition ) if !? consteval compound-statement  else␣
↪statement
│                                           selection-statement──────────────
│                                           statement────────────────────────
│ selection-
↪statement──────────────────────────────────────────────────────────────────────────────
```

The C++ standard explicitely excludes the second possibility in section 8.5.2[3]. The conflict is resolved by annotating the grammar with a priority for the first form of the *selection-statement*.

[3] 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.

elaborated-enum-specifier / *opaque-enum-declaration*

The *opaque-enum-declaration* syntax is the same as a *simple-declaration* of type *elaborated-enum-specifier*.

```
enum-head-name -> identifier
elaborated-enum-specifier -> enum-key identifier
```

```
reduce using rule enum-head-name -> identifier

| enum-key identifier      ;
|          enum-head-name
| opaque-enum-declaration—
| block-declaration————————
| declaration-statement———
| statement———————————————

reduce using rule elaborated-enum-specifier -> enum-key identifier

| enum-key identifier        ;
| elaborated-enum-specifier
| elaborated-type-specifier
| type-specifier——————————
| defining-type-specifier—
| decl-specifier——————————
| decl-specifier-seq———————
| simple-declaration———————
| block-declaration————————
| declaration-statement————
| statement———————————————
```

During semantic analysis, some of these valid grammatical constructs will be rejected:

> An *opaque-enum-declaration* declaring an unscoped enumeration shall not omit the *enum-base*.

In the context of a statement, it is not allowed to forward declare an enumeration. In order to support opaque enum declarations properly, the parser will discard *declaration-statement*s that only declare an *elaborated-enum-specifier*.

enum-base / bitfield specifier

In a member declaration, `:` token can introduce either a bitfield specifier of a member, or a *enum-base* of an *opaque-enum-declaration*.

```
enum-base? ->   : type-specifier-seq
elaborated-enum-specifier -> enum-key attribute-specifier-seq? enum-head-name
```

```
shift using rule enum-base? ->   : type-specifier-seq

| attribute-specifier-seq? enum-key attribute-specifier-seq? enum-head-name  : type-
↪specifier-seq ;
|                                                                          enum-base?
↪——————————
| opaque-enum-
```

(continues on next page)

(continued from previous page)

```
↪declaration──────────────────────────────────────────────────────────────────
│ member-
↪declaration──────────────────────────────────────────────────────────────────

reduce using rule elaborated-enum-specifier -> enum-key attribute-specifier-seq? enum-
↪head-name

│ attribute-specifier-seq? decl-specifier continue-decl-specifier-seq enum-key attribute-
↪specifier-seq? enum-head-name                                                :␣
↪constant-expression brace-or-equal-initializer? ;
│                                                                  elaborated-enum-
↪specifier──────────────────────── attribute-specifier-seq? attribute-specifier-seq?
│                                                                  elaborated-type-
↪specifier──────────────────────                          member-
↪declarator──────────────────────────────────────────────────────
│                                                                  type-
↪specifier──────────────────────────────────                        member-
↪declarator-list?────────────────────────────────────────────────
│                                                                  defining-type-
↪specifier──────────────────────────
│                                                                  decl-
↪specifier──────────────────────────────────
│                                                                  decl-specifier-
↪seq──────────────────────────────────────────────────────
│                          decl-specifier-
↪seq──────────────────────────────────────────────────────────────────────────
│ member-
↪declaration──────────────────────────────────────────────────────────────────
```

The C++ standard specifies which resolution to apply in section 9.7.1[4].

base-clause / bitfield specifier

In a member declaration, a `:` token can introduce either a bit-field specifier of a member, or a *base-clause* of a *class-specifier*.

```
elaborated-type-specifier -> class-key attribute-specifier-seq? class-head-name
base-clause? ->    : base-specifier-list
```

```
reduce using rule elaborated-type-specifier -> class-key attribute-specifier-seq? class-
↪head-name

│ attribute-specifier-seq? class-key attribute-specifier-seq? class-head-name          ␣
↪                                          : constant-expression brace-or-equal-
↪initializer? ;
│                          elaborated-type-specifier──────────────────────────  attribute-
```

(continues on next page)

[4] A `:` following `enum nested-name-specifier? identifier` within the *decl-specifier-seq* of a *member-declaration* is parsed as part of an *enum-base*.

[Note 1: This resolves a potential ambiguity between the declaration of an enumeration with an *enum-base* and the declaration of an unnamed bit-field of enumeration type.]

(continued from previous page)

```
↪specifier-seq? attribute-specifier-seq?
│                         type-specifier──────────────────────────────────────────
↪                  member-
↪declarator──────────────────────────────────────────────────────────────────────
│                         defining-type-specifier─────────────────────────────────
↪                  member-declarator-list?
↪──────────────────────────────────────────────────
│                         decl-specifier──────────────────────────────────────────
│                         decl-specifier-
↪seq─────────────────────────────────────────────────────────────
│ member-
↪declaration──────────────────────────────────────────────────────────────────────

shift using rule base-clause? ->   : base-specifier-list

│ attribute-specifier-seq? class-key attribute-specifier-seq? class-head-name  : base-
↪specifier-list { member-specification? } continue-decl-specifier-seq decl-specifier-
↪seq member-declarator-list? ;
│                                                                          base-
↪clause?─────────
│                         class-
↪head────────────────────────────────────────────────────────────
│                         class-
↪specifier──────────────────────────────────────────────────────────────────────
│                         defining-type-
↪specifier──────────────────────────────────────────────────────────────
│                         decl-
↪specifier──────────────────────────────────────────────────────────────────────
│                         decl-specifier-
↪seq──────────────────────────────────────────────────────────────────────
│ member-
↪declaration──────────────────────────────────────────────────────────────────────
```

Unlike the *enum-base / bitfield specifier* conflict, there is no mention of this ambiguity in the standard. The reason is that bit-fields are only allowed on integral and enumeration types. The grammar can be adjusted to reject the bit-field option.

new-expression, { and *assignment-expression*, =

The conflict arises after either *new-expression*, or a *conditional-expression* has been parsed. The following {/= token will be opening an *initializer-clause*. The counterexample context shows that when parsing a *member-declarator*, if the bitfield specifier (a *constant-expression*) expands to a *new-expression* or a *conditional-expression*, there is a conflict between matching the *initializer-clause* to the *expression* or to the *member-declarator*.

```
braced-init-list ->   { initializer-list? ,? }
braced-init-list ->   { designated-initializer-list ,? }
new-expression -> ::? new new-placement? new-type-id
new-expression -> ::? new new-placement? ( type-id )
```

```
shift using rule braced-init-list ->   { initializer-list? ,? }

| identifier? attribute-specifier-seq? : ::? new new-placement? ( type-id )  {␣
↪(designated-)initializer-list? ,? }
|                                                                            braced-init-
↪list———————————————————
|                                                                            new-
↪initializer———————————————————
|                                       new-
↪expression———————————————————————————————————————————————
|                                       unary-
↪expression———————————————————————————————————————————
|                                       cast-
↪expression————————————————————————————————————————————
|                                       pm-
↪expression———————————————————————————————————————————————
|                                       multiplicative-
↪expression———————————————————————————————
|                                       additive-
↪expression———————————————————————————————————————
|                                       shift-
↪expression———————————————————————————————————————
|                                       compare-
↪expression———————————————————————————————————————
|                                       relational-
↪expression———————————————————————————————————
|                                       equality-
↪expression———————————————————————————————————————
|                                       and-
↪expression———————————————————————————————————————————————
|                                       exclusive-or-
↪expression————————————————————————————————————
|                                       inclusive-or-
↪expression————————————————————————————————————
|                                       logical-and-
↪expression—————————————————————————————————————
|                                       logical-or-
↪expression————————————————————————————————————
|                                       conditional-
↪expression—————————————————————————————————————
|                                       constant-
↪expression———————————————————————————————————————
| member-
↪declarator————————————————————————————————————————————————————————————————————————————

reduce using rule new-expression -> ::? new new-placement new-type-id

| identifier? attribute-specifier-seq? : ::? new new-placement? new-type-id  { }
|                                       new-expression————————————————— braced-init-
↪list
|                                       unary-expression————————————————— brace-or-
↪equal-initializer
|                                       cast-expression———————————————
```

(continues on next page)

(continued from previous page)

```
                                        pm-expression──────────────────────
                                        multiplicative-expression──────────
                                        additive-expression────────────────
                                        shift-expression───────────────────
                                        compare-expression─────────────────
                                        relational-expression──────────────
                                        equality-expression────────────────
                                        and-expression─────────────────────
                                        exclusive-or-expression────────────
                                        inclusive-or-expression────────────
                                        logical-and-expression─────────────
                                        logical-or-expression──────────────
                                        conditional-expression─────────────
                                        constant-expression────────────────
 member-
↪declarator──────────────────────────────────────────────────────────────────
```

```
assignment-operator ->   =
conditional-expression -> logical-or-expression
```

```
shift using rule assignment-operator ->   =

 attribute-specifier-seq? : logical-or-expression ? expression : logical-or-expression ␣
↪=                   initializer-clause brace-or-equal-initializer?
                                                                                ␣
↪assignment-operator
                                                          assignment-
↪expression──────────────────────────────────────
                            conditional-
↪expression──────────────────────────────────────────────────────────────────
                            constant-
↪expression──────────────────────────────────────────────────────────────────
 member-
↪declarator──────────────────────────────────────────────────────────────────

reduce using rule conditional-expression -> logical-or-expression

 attribute-specifier-seq? : logical-or-expression ? expression : logical-or-expression ␣
↪ = initializer-clause
                                                          conditional-expression␣
↪brace-or-equal-initializer?
                                                          assignment-expression─
                            conditional-expression────────────────────────────
                            constant-expression───────────────────────────────
 member-
↪declarator──────────────────────────────────────────────────────────────────
```

The conflict is resolved in the C++ standard in section 11.4.1[5] by assigning a priority to shifting into the *brace-init-list*.

[5] In a *member-declarator* for a bit-field, the *constant-expression* is parsed as the longest sequence of tokens that could syntactically form a *constant-expression*.

nested-name-specifier, `::`

The `::` being used both as a binary operator (name lookup operator) and a unary operator (root namespace name lookup), there is an ambiguity when two qualified names are allowed to follow each other. When encountering a `::` token, it is possible to continue a previous qualified name, or close the previous qualified name and start a new name lookup in the root namespace.

```
simple-type-specifier -> type-name
nested-name-specifier -> type-name  ::
```

```
reduce using rule simple-type-specifier -> type-name

| type-name             attribute-specifier-seq? ::                    * attribute-
↪specifier-seq? cv-qualifier-seq? )
| simple-type-specifier                          nested-name-specifier
| type-specifier———————                          ptr-
↪operator———————————————————————————————————————
| type-specifier-seq———————————————————— ptr-abstract-
↪declarator———————————————————————————
|                                                  abstract-declarator?
↪———————————————————————————————————
| type-
↪id———————————————————————————————————————————————————————————————————————————

shift using rule nested-name-specifier -> type-name  ::

| type-name  ::          template? template-name attribute-specifier-seq? abstract-
↪declarator? )
| nested-name-specifier
| simple-type-specifier———————————————
| type-specifier———————————————
| type-specifier-seq———————————————————————————————————
| type-
↪id———————————————————————————————————————————————————————————————————————
```

The resolution is to continue the previous name lookup. There does not seem to be any mention of this in the C++ standard, but compilers seem to implement it this way.

conversion-type-id, *attribute-specifier-seq*

The counter-examples show a context where a *conversion-type-id* can be directly followed by an *attribute-specifier-sequence*. Since the *conversion-type-id* can also end with an *attribute-specifier-sequence*, there is an ambiguity as to where the two sequences are split.

The example shown below is with the `*` operator and `[[` token. Variations of the conflict exist for all *ptr-operator* constructs, and all *attribute-specifier*s.

```
attribute-specifier ->   [[ attribute-using-prefix? attribute-list ] ]
ptr-operator -> *
```

```
struct S {
  int z : 1 || new int { 0 };   // OK, brace-or-equal-initializer is absent
};
```

```
shift using rule attribute-specifier ->   [[ attribute-using-prefix? attribute-list ] ]

| operator type-specifier-seq &&  [[ attribute-using-prefix? attribute-list ] ]
                                  attribute-specifier————————————————————————
                                  attribute-specifier-seq———————————————————
                               ptr-operator———————————————————————————————
                               conversion-declarator——————————————————————
          conversion-type-id————————————————————————————————————————————
| conversion-function-id————————————————————————————————————————————————
| unqualified-id———————————————————————————————————————————————————————
| id-expression————————————————————————————————————————————————————————
| declarator-id————————————————————————————————————————————————————————
| noptr-declarator—————————————————————————————————————————————————————

reduce using rule ptr-operator -> *

| operator type-specifier-seq *                       [[ attribute-using-prefix?␣
↪attribute-list ] ]
|                              ptr-operator          attribute-
↪specifier————————————————————————
|                              conversion-declarator attribute-specifier-
↪seq———————————————————
|          conversion-type-id———————————————————
| conversion-function-id———————————————————
| unqualified-id———————————————————————————
| id-expression————————————————————————————
| declarator-id————————————————————————————
| noptr-
↪declarator———————————————————————————————————————————————————————————————
```

According to the standard in section 11.4.8.3[6], the attribute specifier sequence is consumed by the *conversion-type-id* by applying a priority on shifting the *attribute-specifier*s and *cv-qualifier*s over the reductions of *ptr-operator*s.

[6] The *conversion-type-id* in a *conversion-function-id* is the longest sequence of tokens that could possibly form a *conversion-type-id*.
[Note 1: This prevents ambiguities between the declarator operator * and its expression counterparts.
[Example 3:

```
&ac.operator int*i; // syntax error:
                    // parsed as: &(ac.operator int *)i
                    // not as: &(ac.operator int)*i
```

The * is the pointer declarator and not the multiplication operator. — end example]
This rule also prevents ambiguities for attributes.

[Example 4:

```
operator int [[noreturn]] ();   // error: noreturn attribute applied to a type
```

—end example]

— end note]

explicit-specifier/*noexcept-specifier*, (

A conflict arises in all declarations (narrowed down to one counter-example here) when encountering the (token after the `explicit` keyword or the `noexcept` keyword:

```
explicit-specifier -> explicit  ( constant-expression )
explicit-specifier -> explicit
```

```
reduce using rule explicit-specifier -> explicit

| attribute-specifier-seq explicit            ( ptr-declarator ) parameters-and-
↪qualifiers trailing-return-type declarator function-body
|                         explicit-specifier noptr-declarator
|                         function-specifier␣
↪declarator──────────────────────────────────────────────────
|                         decl-specifier────
|                         decl-specifier-seq
| function-
↪definition──────────────────────────────────────────────────────────────────────────────────────────────────────────────
|␣
↪declaration─────────────────────────────────────────────────────────────────────────────────────────────────────────────

shift using rule explicit-specifier -> explicit  ( constant-expression )

| attribute-specifier-seq explicit  ( constant-expression ) declarator function-body
|                         explicit-specifier──────────────
|                         function-specifier──────────────
|                         decl-specifier──────────────────
|                         decl-specifier-seq──────────────
| function-definition──────────────────────────────────────────────────
| declaration──────────────────────────────────────────────────────────
```

```
noexcept-specification -> noexcept  ( constant-expression )
noexcept-specification -> noexcept
```

```
reduce using rule noexcept-specification -> noexcept

| noptr-declarator ( parameter-declaration-clause ) cv-qualifier-seq? ref-qualifier?␣
↪noexcept                                              ( expression-list )
|                                                                                 ␣
↪noexcept-specification   attribute-specifier-seq? initializer?─────
|                                                                                 ␣
↪exception-specification?
|                   parameters-and-
↪qualifiers──────────────────────────────────────────────────────────────────────────────────────────────────────────────
| noptr-
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────
| ptr-
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────
|␣
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

(continues on next page)

(continued from previous page)

```
| init-
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────

shift using rule noexcept-specification -> noexcept  ( constant-expression )

| noptr-declarator ( parameter-declaration-clause ) cv-qualifier-seq? ref-qualifier?␣
↪noexcept  ( constant-expression ) attribute-specifier-seq? trailing-return-type␣
↪initializer?
|                                                                                  ␣
↪noexcept-specification──────────
|                                                                                  ␣
↪exception-specification?─────────
|                   parameters-and-
↪qualifiers──────────────────────────────────────────────────────────────────────────────────────────────────────────────
|␣
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────
| init-
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

The standard disambiguates the conflict for `explicit` in section 9.2.3[7] and for `noexcept` in section 14.5.2[8]. In both cases, The grammar conflict is resolved by prioritizing the shift.

Operators `new`, `new[]`, `delete`, `delete[]`

When using the operators `new` and `delete` as *declarator-id*s in a declaration, or as *unqualified-id*s in an expression, it can be followed by the array operator `[`. It is then ambiguous if the array operator is specifying the array form of the `new`/`delete` operators or a subscript expression or an array declaration.

```
overloadable-operator -> new  [ ]
overloadable-operator -> new
```

In an expression:

```
shift using rule overloadable-operator -> new  [ ]

| operator new  [ ]
|          overloadable-operator
| operator-function-id──────────
| unqualified-id────────────────
| id-expression─────────────────
| primary-expression────────────
| postfix-expression────────────

reduce using rule overloadable-operator -> new

| operator new                    [ expr-or-braced-init-list ]
|          overloadable-operator
| operator-function-id──────────
```

(continues on next page)

[7] A `(` token that follows `explicit` is parsed as part of the *explicit-specifier*.
[8] A `(` token that follows `noexcept` is part of the *noexcept-specifier* and does not commence an *initializer*.

(continued from previous page)

```
  unqualified-id──────────────────
  id-expression───────────────────
  primary-expression──────────────
  postfix-expression──────────────
  postfix-expression──────────────────────────────────────────
```

In a declaration:

```
shift using rule overloadable-operator -> new  [ ]

  operator new  [ ]
           overloadable-operator
  operator-function-id────────────
  unqualified-id──────────────────
  id-expression───────────────────
  declarator-id───────────────────
  noptr-declarator────────────────

reduce using rule overloadable-operator -> new

  operator new                    [ constant-expression? ] attribute-specifier-seq?
           overloadable-operator
  operator-function-id────────────
  unqualified-id──────────────────
  id-expression───────────────────
  declarator-id───────────────────
  noptr-declarator────────────────
  noptr-declarator──────────────────────────────────────────────────────────────────
```

```
overloadable-operator -> delete  [ ]
overloadable-operator -> delete
```

In an expression:

```
shift using rule overloadable-operator -> delete  [ ]

  operator delete  [ ]
           overloadable-operator
  operator-function-id────────────
  unqualified-id──────────────────
  id-expression───────────────────
  primary-expression──────────────
  postfix-expression──────────────

reduce using rule overloadable-operator -> delete

  operator delete                 [ expr-or-braced-init-list ]
           overloadable-operator
  operator-function-id────────────
  unqualified-id──────────────────
```

(continues on next page)

(continued from previous page)

```
| id-expression———————————
| primary-expression———————————
| postfix-expression———————————
| postfix-expression———————————————————————————————
```

In a declaration:

```
shift using rule overloadable-operator -> delete  [ ]

| operator delete  [ ]
|          overloadable-operator
| operator-function-id———————
| unqualified-id———————————————
| id-expression———————————————
| declarator-id———————————————
| noptr-declarator————————————

reduce using rule overloadable-operator -> delete

| operator delete                 [ constant-expression? ] attribute-specifier-seq?
|          overloadable-operator
| operator-function-id———————
| unqualified-id———————————————
| id-expression———————————————
| declarator-id———————————————
| noptr-declarator————————————
| noptr-declarator———————————————————————————————————————————————————————————
```

There does not seem to be any priority defined in the C++ standard, but in all similar cases the standard defines the *operator-id* as the longest sequence of valid tokens, and major compilers resolve the conflict by using the `new[]/delete[]` version.

```
int main()
{
    return &::operator new[0] ? 0 : 1;
}
```

Using GCC:

```
main.cc: In function ‘int main()’:
main.cc:3:28: error: expected ‘]’ before numeric constant
   3 |     return &::operator new[0] ? 0 : 1;
```

Using Clang:

```
main.cc:3:28: error: expected ']'
      return &::operator new[0] ? 0 : 1;
```

Operators `delete`, `delete[]` and *lambda-expression*

In a delete expression, an array subscript token `[` could introduce either the array form of the *delete-expression* or open a new *lambda-expression*.

```
delete [ ]  cast-expression
lambda-introducer -> [ ]
```

```
shift using rule delete [ ]  cast-expression

| delete [ ]  cast-expression
| delete-expression———————

reduce using rule lambda-introducer -> [ ]

| delete [ ]                lambda-declarator
|        lambda-introducer
|        lambda-expression———————————————
|        primary-expression———————————————
|        postfix-expression———————————————
|        unary-expression———————————————
|        cast-expression———————————————
| delete-expression———————————————————
```

There does not seem to be any priority defined in the C++ standard, but just as when resolving *Operators new, new[], delete, delete[]* the parser will resolve by using the `delete[]` version.

```
int main()
{
    delete []() { return (int*) 0; }();
}
```

Using GCC:

```
main.cc: In function ‘int main()’:
main.cc:3:15: error: expected primary-expression before ‘)’ token
    3 |     delete []() { return (int*) 0; }();
      |               ^
```

Using Clang:

```
main.cc:3:5: error: '[]' after delete interpreted as 'delete[]'; add parentheses to␣
↪treat this as a lambda-expression
    delete []() { return (int*) 0; }();
   ^~~~~~~~
           (                       )
```

The conflict occurs only at the closing subscript token `]`, which indicates the parser will succeed parsing a *delete-expression* of the result of a *lambda-expression* provided that the *lambda-introducer* is not an empty capture. Interestingly, GCC and Clang disagree on this case.

```
int main()
{
```

(continues on next page)

(continued from previous page)

```
    delete [&]() { return (int*) 0; }();
}
```

Using GCC:

```
main.cc: In function ‘int main()’:
main.cc:3:13: error: expected ‘]’ before ‘&’ token
    3 |     delete [&]() { return (int*) 0; }();
      |             ^
      |             ]
main.cc:3:14: error: expected primary-expression before ‘]’ token
    3 |     delete [&]() { return (int*) 0; }();
      |              ^
main.cc:3:39: error: expected primary-expression before ‘)’ token
    3 |     delete [&]() { return (int*) 0; }();
```

Using Clang successfully compiles.

conversion-declarator, binary operators

When using a *conversion-function-id* as an *unqualified-id* in an expression, the parser encounters a conflict when encountering tokens that are used either as *ptr-operator*s or binary operators (&, &&, *). The token can be interpreted as either continuing the *conversion-type-id*, or starting a binary operation using the shorter version of the *conversion-type-id*.

```
ptr-operator ->   * attribute-specifier-seq? cv-qualifier-seq?
conversion-declarator? ->
```

```
shift using rule ptr-operator ->   * attribute-specifier-seq? cv-qualifier-seq?

| operator type-specifier-seq  * attribute-specifier-seq? cv-qualifier-seq? conversion-
↪declarator?
|                             ptr-operator──────────────────────────────
|                             conversion-declarator?
↪──────────────────────────────────────────────
|          conversion-type-
↪id──────────────────────────────────────────────────────────────
| conversion-function-
↪id──────────────────────────────────────────────────────────────────
| unqualified-
↪id──────────────────────────────────────────────────────────────────────
| id-
↪expression──────────────────────────────────────────────────────────────────────
| primary-
↪expression──────────────────────────────────────────────────────────────────
| postfix-
↪expression──────────────────────────────────────────────────────────────────
| unary-
↪expression──────────────────────────────────────────────────────────────────
| cast-
↪expression──────────────────────────────────────────────────────────────────
```

(continues on next page)

(continued from previous page)

```
| pm-
↪expression──────────────────────────────────────────────────────────────────────────────────────
| multiplicative-
↪expression──────────────────────────────────────────────────────────────────────

reduce using rule conversion-declarator? ->

| operator type-specifier-seq                          * pm-expression
|                            conversion-declarator?
|          conversion-type-id──────────────────────
| conversion-function-id──────────────────────────
| unqualified-id──────────────────────────────────
| id-expression──────────────────────────────────
| primary-expression──────────────────────────────
| postfix-expression──────────────────────────────
| unary-expression────────────────────────────────
| cast-expression─────────────────────────────────
| pm-expression───────────────────────────────────
| multiplicative-expression──────────────────────
| multiplicative-expression──────────────────────────────────────────
```

The C++ standard disambiguates this case in section 11.4.8.3[9] by prioritizing a shift of the *ptr-operator* over reducing the *conversion-type-id*.

new-type-id, binary operators

In a similar way to *conversion-declarator, binary operators*, a *new-expression* can appear as a left operand of a *multiplicative-expression* causing a conflict when encountering the * token.

```
ptr-operator ->   * attribute-specifier-seq? cv-qualifier-seq?
new-declarator -> ptr-operator
```

```
shift using rule ptr-operator ->   * attribute-specifier-seq? cv-qualifier-seq?

| ::? new type-specifier-seq ptr-operator  * attribute-specifier-seq? cv-qualifier-seq?␣
↪new-initializer?
|                                        ptr-operator──────────────────────────────
|                                        new-declarator──────────────────────────────
|                              new-declarator──────────────────────────────────────
|         new-type-id──────────────────────────────────────────────────────────────
| new-
↪expression──────────────────────────────────────────────────────────────────────────────────────
| unary-
↪expression──────────────────────────────────────────────────────────────────────────────────────
```

(continues on next page)

[9] The *conversion-type-id* in a *conversion-function-id* is the longest sequence of tokens that could possibly form a *conversion-type-id*.
[Note 1: This prevents ambiguities between the declarator operator * and its expression counterparts.

```
&ac.operator int*i; // syntax error:
                    // parsed as: &(ac.operator int *)i
                    // not as: &(ac.operator int)*i
```

The * is the pointer declarator and not the multiplication operator. — end example]

(continued from previous page)

```
| cast-
↪expression———————————————————————————————————————————————————————————————
| pm-
↪expression———————————————————————————————————————————————————————————————————
| multiplicative-
↪expression———————————————————————————————————————————————————————————————

reduce using rule new-declarator -> ptr-operator

| ::? new type-specifier-seq ptr-operator    new-initializer? * pm-expression
|                            new-declarator
|        new-type-id————————————————————
| new-expression———————————————————————————————————————
| unary-expression——————————————————————————————————————
| cast-expression———————————————————————————————————————
| pm-expression————————————————————————————————————————
| multiplicative-expression————————————————————————————
| multiplicative-expression———————————————————————————————————————————
```

The standard specifies in section 7.6.2.8[10] by prioritizing a shift of the *ptr-operator* over reducing the *new-type-id*.

destructor, unary ~ operator

Everywhere in the grammar that allows an *expression*, starting with a ~ token can lead to two different expansions, using a destructor name as an *unqualified-id* or building a *unary-expression* with the ~ operator.

```
enum-name ->   identifier
typedef-name ->   identifier
class-name ->   identifier
template-name ->   identifier
unary-operator -> ~
```

```
shift using rule enum-name ->   identifier

| ~  identifier
|   enum-name—
|   type-name—
| unqualified-id
| id-expression—
| primary-expression
| postfix-expression
| unary-expression——

shift using rule typedef-name ->   identifier
```

(continues on next page)

[10] The *new-type-id* in a *new-expression* is the longest possible sequence of *new-declarators*.
[Note 3: This prevents ambiguities between the declarator operators &, &&, *, and [] and their expression counterparts. — end note]
[Example 2:

```
new int * i;                    // syntax error: parsed as (new int*) i, not as (new int)*i
```

The * is the pointer declarator and not the multiplication operator. — end example]

(continued from previous page)

```
| ~  identifier
|   typedef-name
|   type-name——
| unqualified-id
| id-expression—
| primary-expression
| postfix-expression
| unary-expression——

shift using rule class-name ->   identifier

| ~  identifier
|   class-name
|   type-name—
| unqualified-id
| id-expression—
| primary-expression
| postfix-expression
| unary-expression——

shift using rule template-name ->   identifier

| ~  identifier    < template-argument-list? >
|   template-name
|   simple-template-id————————————————————
|   typedef-name——————————————————————————
|   type-name—————————————————————————————
| unqualified-id——————————————————————————
| id-expression———————————————————————————
| primary-expression——————————————————————
| postfix-expression——————————————————————
| unary-expression————————————————————————

reduce using rule unary-operator -> ~

| ~               identifier              braced-init-list
| unary-operator template-name
|                  simple-type-specifier
|                  postfix-expression———————————————————
|                  unary-expression————————————————————
|                  cast-expression—————————————————————
| unary-expression————————————————————————————————————
```

```
decltype-specifier[split:] ->   decltype ( expression )
unary-operator -> ~
```

```
shift using rule decltype-specifier[split:] ->   decltype ( expression )

| ~  decltype ( expression )
```

(continues on next page)

(continued from previous page)

```
  decltype-specifier─────
unqualified-id──────────
id-expression───────────
primary-expression──────
postfix-expression──────
unary-expression────────

reduce using rule unary-operator -> ~

~               decltype ( auto )            braced-init-list
unary-operator placeholder-type-specifier
                simple-type-specifier─────
                postfix-expression──────────────────────────
                unary-expression────────────────────────────
                cast-expression─────────────────────────────
unary-expression────────────────────────────────────────────
```

The conflict is resolved in the C++ standard in section 7.6.2.2[11] by prioritizing the *unary-operator* rule.

constraint-logical-and-expression, &&

In the grammar, a function declaration can sometimes only consist of a *declarator* without return type in order to allow constructors, destructors and cast operators. The grammar is very generic and allows the rule `function-definition : declarator function-body`. It means the sequence `&& identifier { }` is grammatically correct but is rejected during the semantic analysis.

A function declaration/definition can also appear in a *template-declaration* and have constraints attached to it, which can use the && operator.

```
constraint-logical-or-expression -> constraint-logical-or-expression || constraint-
↪logical-and-expression
constraint-logical-and-expression -> constraint-logical-and-expression  && constraint-
↪primary-expression
```

```
reduce using rule constraint-logical-or-expression -> constraint-logical-or-expression␣
↪|| constraint-logical-and-expression

attribute-specifier-seq? extern? template < template-parameter-list > requires␣
↪constraint-logical-or-expression || constraint-logical-and-expression                ␣
↪           && attribute-specifier-seq? ptr-declarator function-body
                                                                                      ␣
↪constraint-logical-or-expression──────────────────────────────────── attribute-
↪specifier-seq? ptr-operator──────────────
                                                                      requires-
↪clause──────────────────────────────────────────────────────────────────────────────  ␣
↪          ptr-declarator──────────────────────────
```

(continues on next page)

[11] There is an ambiguity in the grammar when ~ is followed by a *type-name* or *decltype-specifier*. The ambiguity is resolved by treating ~ as the unary complement operator rather than as the start of an *unqualified-id* naming a destructor.

[Note 6: Because the grammar does not permit an operator to follow the `.`, `->`, or `::` tokens, a ~ followed by a *type-name* or *decltype-specifier* in a member access expression or *qualified-id* is unambiguously parsed as a destructor name. — end note]

(continued from previous page)

```
│                                template-
↪head──────────────────────────────────────────────────────────────────────────────
↪                           declarator──────────────────────────────
│                                                                                ␣
↪                                                        function-
↪definition──────────────────────────────────────────────────────────
│                                                                                ␣
↪                                                        noexport-
↪declaration─────────────────────────────────────────────────────────
│                                                                                ␣
↪                                                    ␣
↪declaration─────────────────────────────────────────────────────────────────────
│ template-
↪declaration──────────────────────────────────────────────────────────────────────

shift using rule constraint-logical-and-expression -> constraint-logical-and-expression ␣
↪&& constraint-primary-expression

│ attribute-specifier-seq? extern? template < template-parameter-list > requires␣
↪constraint-logical-or-expression || constraint-logical-and-expression  && constraint-
↪primary-expression declaration
│                                                                                ␣
↪                            constraint-logical-and-
↪expression──────────────────────────────────
│                                                                             ␣
↪constraint-logical-or-
↪expression──────────────────────────────────────────────────────────────────
│                                                                  requires-
↪clause──────────────────────────────────────────────────────────────────────────
│                                template-
↪head──────────────────────────────────────────────────────────────────────────────
│ template-
↪declaration──────────────────────────────────────────────────────────────────────
```

Choosing the semantically invalid rule would also shadow the valid *constraint-logical-and-expression*, so the parser uses the shift clause.

global-module-fragment

In the C++ standard, the C++ grammar defines the *global-module-fragment* to be a sequence of declarations following the global module specifier. The *global-module-fragment* can be followed by another sequence of declarations, but the note states

> Prior to phase 4 of translation, only preprocessing directives can appear in the *declaration-seq*

```
global-module-fragment -> module ; declaration-seq?
export-declaration ->   export { noexport-declaration-seq? }
export-declaration ->   export noexport-declaration
export-declaration ->   export module-import-declaration
```

```
reduce using rule global-module-fragment -> module ; declaration-seq?

| module ; declaration-seq?  export module module-name ; declaration-seq? private-module-
↪fragment?
| global-module-fragment—— module-declaration———————
| translation-
↪unit————————————————————————————————————————————————————————————————————

shift using rule export-declaration ->   export module-import-declaration

| module ; declaration-seq  export module-import-declaration module-declaration
|                           export-declaration————————————
|                           declaration———————————————————
|          declaration-seq——————————————————————————————
| global-module-fragment————————————————————————————————
| translation-unit———————————————————————————————————————————————————————

shift using rule export-declaration ->   export { noexport-declaration-seq? }

| module ; declaration-seq  export { noexport-declaration-seq? } module-declaration
|                           export-declaration————————————————
|                           declaration———————————————————————
|          declaration-seq——————————————————————————————————
| global-module-fragment————————————————————————————————————
| translation-unit———————————————————————————————————————————————————————

shift using rule export-declaration ->   export noexport-declaration

| module ; declaration-seq  export noexport-declaration module-declaration
|                           export-declaration———————————
|                           declaration——————————————————
|          declaration-seq—————————————————————————————
| global-module-fragment———————————————————————————————
| translation-unit———————————————————————————————————————————————
```

Since the parser is parsing non-preprocessed source files, it is safe to annotate the grammar to reduce.

requirement-expression, nested-requirement

In places allowing a requirement, the `requires` keyword leads to two possible expansions: it could introduce a *requirement-expression* as part of a *simple-requirement*, or it could start a *nested-requirement*. The possible expansions lead to conflicts after a few tokens have been parsed. The standard indicates that the `requires` keyword in this situation always introduces a *nested-requirement*. It is therefore possible to resolve all conflicts by preferring the *nested-requirement* option. This would lead to a lot of annotations spread across the grammar.

In order to simplify the grammar, it is modified to introduce an earlier conflict, an empty production before the `requires` keyword. This empty production causes a shift-reduce conflict that hides all subsequent conflicts in the expansion. Resolving this single conflicts removes the possibility of expanding a *requirement-expression* altogether, hiding both the ambiguous constructs but also the unambiguous ones as required by the standard.

```
requires-disambiguation ->
nested-requirement ->   requires constraint-expression ;
```

```
shift using rule nested-requirement ->   requires constraint-expression ;

  requires constraint-expression ;
 nested-requirement———————————————
 requirement——————————————————————

reduce using rule requires-disambiguation ->

                          requires requirement-parameter-list? requirement-body␣
↪assignment-operator initializer-clause ;
 requires-disambiguation
 requires-expression———————————————————————————————————————————————————————
 primary-expression————————————————————————————————————————————————————————
 postfix-expression————————————————————————————————————————————————————————
 unary-expression——————————————————————————————————————————————————————————
 cast-expression———————————————————————————————————————————————————————————
 pm-expression—————————————————————————————————————————————————————————————
 multiplicative-expression—————————————————————————————————————————————————
 additive-expression———————————————————————————————————————————————————————
 shift-expression——————————————————————————————————————————————————————————
 compare-expression————————————————————————————————————————————————————————
 relational-expression—————————————————————————————————————————————————————
 equality-expression———————————————————————————————————————————————————————
 and-expression————————————————————————————————————————————————————————————
 exclusive-or-expression———————————————————————————————————————————————————
 inclusive-or-expression———————————————————————————————————————————————————
 logical-and-expression————————————————————————————————————————————————————
 logical-or-expression—————————————————————————————————————————————————————
 assignment-
↪expression———————————————————————————————————————————————————————————————
 ␣
↪expression———————————————————————————————————————————————————————————————
 simple-
↪requirement——————————————————————————————————————————————————————————————
 ␣
↪requirement——————————————————————————————————————————————————————————————
```

In this modified grammar, resolving as shift prioritizes the *nested-requirement* over a *requires-expression*.

id-expression in a *template-argument*

The C++ grammar rules list three valid template arguments:

- a *constant-expression*
- a *type-id*
- an *id-expression*

Constant expressions are not only constant values; these are expressions that can be computed and reduced to a constant. For instance, `1 + 1` is a constant expression in C++. This computation is actually performed during semantic analysis; the grammar does not yet differentiate *expression*s from *constant-expression*s. At the grammar level, `1 + x` is a *constant-expression*. During semantic analysis, `1 + x` is a *constant-expression* if `x` is a constant.

Since the grammar allows *id-expression*s as *constant-expression*s, it means that the third rule above is redundant and already covered in rule 1.

```
primary-expression -> id-expression
template-argument -> id-expression
```

```
reduce using rule primary-expression -> id-expression

| template-name < id-expression              >
|                 primary-expression
|                 postfix-expression
|                 unary-expression——
|                 cast-expression———
|                 pm-expression———–
|                 multiplicative-expression
|                 additive-expression———–
|                 shift-expression————–
|                 compare-expression———–
|                 relational-expression———
|                 equality-expression———–
|                 and-expression————————
|                 exclusive-or-expression——
|                 inclusive-or-expression——
|                 logical-and-expression———
|                 logical-or-expression———
|                 conditional-expression———
|                 constant-expression———–
|                 template-argument————–
|                 template-argument-list?——
| simple-template-id————————————————

reduce using rule template-argument -> id-expression

| template-name < id-expression            >
|                 template-argument
|                 template-argument-list?
| simple-template-id————————————————
```

In this case the second possibility is ignored by the parser.

export-declaration, `export` *module-import-declaration*

The grammar accepts two ways of exporting a *module-import-declaration*. The *module-import-declaration* could first be reduced into a *declaration*, then the general rule `export-declaration : export declaration` applies. Alternatively, the rule `export-declaration : export module-import-declaration` can be applied directly, shortcutting the intermediate reduction:

```
declaration -> module-import-declaration
export-declaration -> export module-import-declaration
```

```
reduce using rule declaration -> module-import-declaration

  export module-import-declaration
         declaration──────────────
  export-declaration──────────────
  declaration─────────────────────

reduce using rule export-declaration -> export module-import-declaration

  export module-import-declaration
  export-declaration──────────────
  declaration─────────────────────
```

The standard forbids in section 10.2[12] to reduce the *module-import-declaration* into a *declaration*. The other cases described in the standard do not cause additional syntax conflicts, so the restrictions can be applied during the semantic analysis in order to emit better error messages.

1.2.1.4 Dynamic conflict resolution

This section lists the conflicts that can't be decided with the next lookahead. The resolution depends on information that is not available to the parser at the moment it has to make a decision. The parser then splits parsing into two or more branches that will be resolved at a later point.

The parser will continue to maintain several branches until either the lookaheads generate parse errors in some of the tentative branches, or the branches reduce in the same state and the state can execute a merge action.

There are two possible outcomes:

- The parse will naturally return to one possibility after a few tokens have been parsed. When the grammar requires additional tokens to disambiguate, the parser simply keep all options alive.
- The parse will produce two or more valid reductions. Those reductions need to be merged into a single production as early as possible in order to reduce parsing time; as long as there are two options available to the parser, it will have to maintain both in parallel. Since there are a few ambiguities that can lead to further ambiguities, the combination of options can quickly increase.

class-name, *enum-name*, *typedef-name* in a *type-name*

When expecting a *type-name*, an `identifier` could be interpreted as either a *class-name*, an *enum-name*, or a *typedef-name*. While it would be possible to let the parser handle this as a dynamic conflict, this type of conflict is so frequent that it causes significant overhead during parsing.

The rules were amended to expand all *class-name*, *enum-name*, or *typedef-name* references into *identifier* references.

[12] The *declaration* or *declaration-seq* of an *export-declaration* shall not contain an *export-declaration* or *module-import-declaration*.

declarator, *decl-specifier*

In a *simple-declaration*, an `identifier` can be part of either the *decl-specifier-seq* (as a *type-specifier*) or as part of the *declarator* (as a *qualified-id* or *unqualified-id*). There is not enough information at the moment the `identifier` is encountered to make that decision, so the parser attempts to continue both versions. We do know however that the *decl-specifier-seq* can only contain one *type-specifier*, so if a *decl-specifier-seq* already contains a *type-specifier* of some sort, then the `identifier` must be considered part of the *declarator*.

```
typedef X int;

decl-specifier-seq
decl-specifier     identifier
unsigned             X            ;
// unsigned is enough for a type, therefore the next identifier
// is treated as part of the declarator
```

During reduction, if a *decl-specifier-seq* is constructed by adding a second *type-specifier*, the reduction rule raises a `SyntaxError` to abort the parsing of that branch.

template-id, <

The conflict arises when encountering the < token. Depending on the resolution of the entity in front of the < token, it could be part of a *template-id* or a *relational-expression*.

```
unqualified-id -> literal-operator-id
template-id -> literal-operator-id  < template-argument-list? >
```

```
reduce using rule unqualified-id -> literal-operator-id

  literal-operator-id        < compare-expression
  unqualified-id─────
  id-expression──────
  primary-expression─
  postfix-expression─
  unary-expression───
  cast-expression────
  pm-expression──────
  multiplicative-expression
  additive-expression────────
  shift-expression───────────
  compare-expression─────────
  relational-expression─────
  relational-expression──────────────────────────

shift using rule template-id -> literal-operator-id  < template-argument-list >

  literal-operator-id  < template-argument-list? >
  template-id──────────────────────────────────────
  unqualified-id───────────────────────────────────
  id-expression────────────────────────────────────
  primary-expression──────────────────────────────
  postfix-expression──────────────────────────────
```

(continues on next page)

(continued from previous page)

```
unary-expression———————————————————————
cast-expression———————————————————————
pm-expression———————————————————————
multiplicative-expression———————————————
additive-expression———————————————————
shift-expression———————————————————————
compare-expression———————————————————
relational-expression———————————————————
```

The parser explores both solutions as there is no way to know which path is correct until semantic analysis. There are many solutions to this conflict; the conflict can include the `,` token as either a delimiter for the template argument, or a comma-separated *expression*, or anywhere where *type-list* are expected:

```
struct A : B< Y<1>, Z<2>
   // A has one parent class: B< Y<1>, (Z<2) >
   // or two parent classes: B< (Y<1) > and Z<2>
{
};
```

The parser creates a split (which often leads to subsequent splits) and will attempt merges in several rules:

- *expression*
- *constant-expression*
- *cast-expression*
- *postfix-expression*
- *relational-expression*
- *shift-expression*
- *fold-expression*
- *constraint-expression*
- *template-argument-list*
- *base-specifier-list*
- *type-id-list* of a *dynamic-exception-specification*

Unfortunately many of those create valid (although unlikely) possibilities that are merged in an ambiguous parse tree node in the abstract syntax tree. A semantic pass can then discard the invalid possibilities when the `identifier` can be resolved.

variadic-parameter-list, pack-declarator

```
declarator-id ->   ... id-expression
noptr-abstract-pack-declarator ->   ...
abstract-declarator?[split:declarator_end] ->
```

```
shift using rule declarator-id ->   ... id-expression
```

(continues on next page)

(continued from previous page)

```
| attribute-specifier-seq? decl-specifier-seq  ... id-expression attribute-specifier-seq?
↪ parameters-and-qualifiers trailing-return-type , variadic-parameter-list
|                                             declarator-id———
|                                             noptr-declarator————————————————
|                                            ␣
↪declarator————————————————————————————————————————————————————————————————————————
| parameter-
↪declaration————————————————————————————————————————————————————————————————————————
| parameter-declaration-
↪list———————————————————————————————————————————————————————————————————————————————
| parameter-declaration-
↪clause—————————————————————————————————————————————————————————————————————————————

shift using rule noptr-abstract-pack-declarator ->   ...

| attribute-specifier-seq? decl-specifier-seq  ...                          , variadic-
↪parameter-list
|                                             noptr-abstract-pack-declarator
|                                             abstract-pack-declarator———————
| parameter-declaration———————————————————————————————————————————————————————
| parameter-declaration-list——————————————————————————————————————————————————
| parameter-declaration-
↪clause—————————————————————————————————————————————————————————

reduce using rule abstract-declarator?[split:declarator_end] ->

| attribute-specifier-seq? decl-specifier-seq                        ...
|                                             abstract-declarator? variadic-parameter-
↪list?
| parameter-declaration———————————————————————————————————
| parameter-declaration-list?—————————————————————————————
| parameter-declaration-
↪clause—————————————————————————————————————————————
```

The C++ describes this ambiguity in section 9.3.4.6[#]. The logic cannot be implemented at the syntax analysis step so both options are accepted until the semantic analysis.

primary-expression, *pure-specifier* / bitfield declaration, *mem-initializer* / *compound-statement*, *brace-or-equal-initializer*

There are three tokens that can introduce either a function definition or an initializer:

- { can introduce a function body or a brace initializer:

```
| attribute-specifier-seq? decl-specifier-seq? declarator  { statement-seq?␣
↪}
|                                                          compound-
↪statement
|                                                          function-
↪body———————
```

(continues on next page)

(continued from previous page)

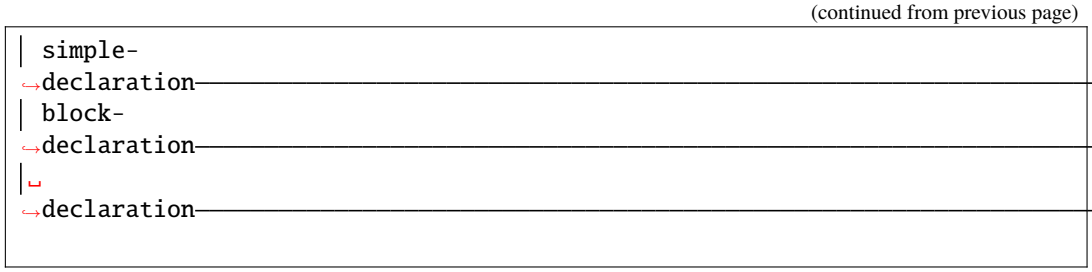
```
| function-
↪definition──────────────────────────────────────────────────────────
|␣
↪declaration──────────────────────────────────────────────────────────────



| attribute-specifier-seq? decl-specifier-seq? declarator  { }            ␣
↪            ;
|                                                          braced-init-list
|                                                          brace-or-equal-
↪initializer
|                                                         ␣
↪initializer─────────────────
|                                          init-
↪declarator──────────────────────────
|                                          init-declarator-
↪list────────────────────
| simple-
↪declaration──────────────────────────────────────────────────────────────
| block-
↪declaration──────────────────────────────────────────────────────────────
|␣
↪declaration──────────────────────────────────────────────────────────────────
```

- = can introduce an initializer or a pure specifier:

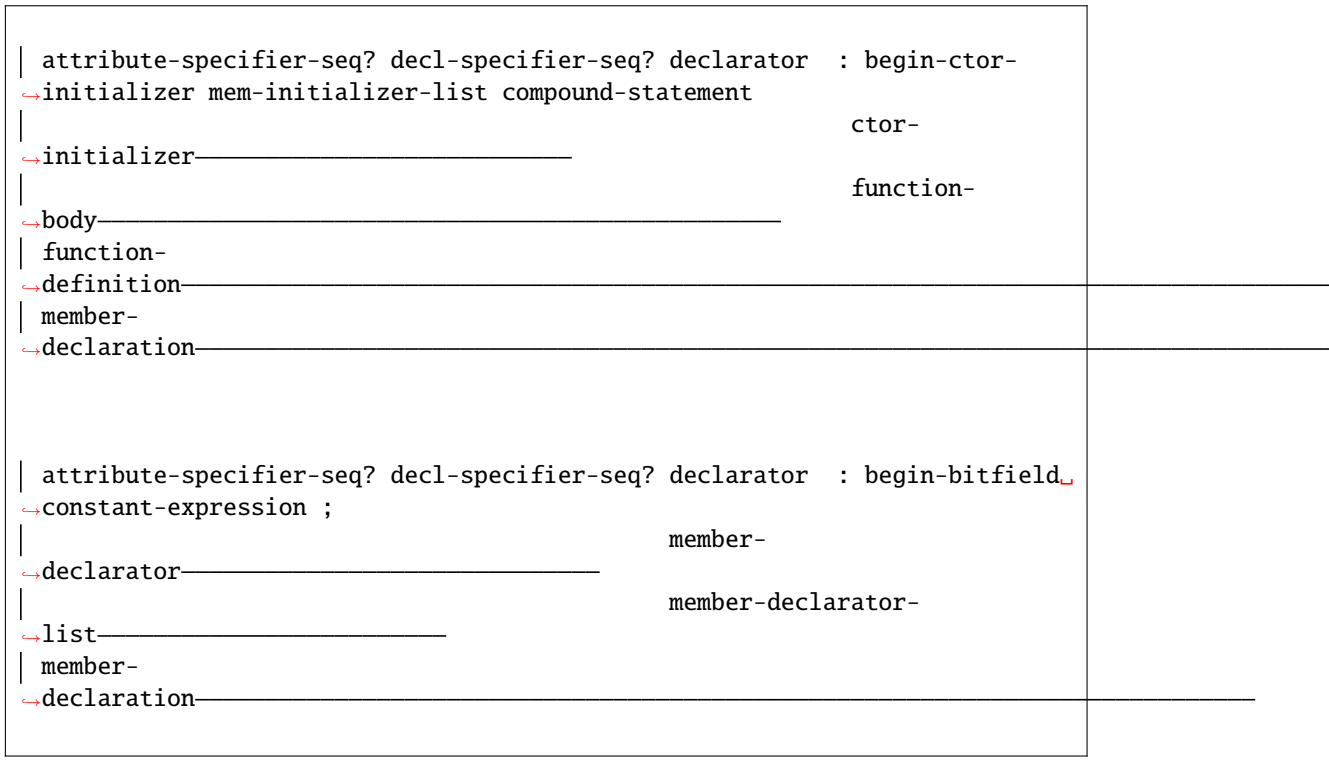
```
| attribute-specifier-seq? decl-specifier-seq? declarator               =␣
↪integer-literal ;
|                                          declarator-function-body␣
↪pure-specifier—
|                                          member-
↪declarator──────────────────────────
|                                          member-declarator-
↪list────────────────────
| member-
↪declaration──────────────────────────────────────────────────────────────



| attribute-specifier-seq? decl-specifier-seq? declarator  = initializer-
↪clause         ;
|                                                          brace-or-equal-
↪initializer
|                                                         ␣
↪initializer─────────────────
|                                          init-
↪declarator──────────────────────────
|                                          init-declarator-
↪list────────────────────
```

(continues on next page)

(continued from previous page)



- `:` can introduce a bitfield or a constructor initializer



We can however avoid parsing both branches by looking at the declarator. If the declarator is a method, we can can discard the initializer branch. If it is not a method then we can discard the function body branch.

An empty production is inserted after the declarator has been parsed; this causes a conflict, which causes a split. During the reduction of the empty production, we can raise a `SyntaxError` to discard the branch if the declarator is not a method declarator.

typename-specifier, typename-parameter

```
enum-name ->   identifier
namespace-alias ->   identifier
namespace-name ->   identifier
typedef-name ->   identifier
class-name ->   identifier
template-name ->   identifier
type-parameter-key -> typename
```

```
shift using rule enum-name ->   identifier

| template-parameter-list , attribute-specifier-seq? typename  identifier ::       ␣
↪template? simple-template-id continue-decl-specifier-seq decl-specifier-seq abstract-
↪declarator?
|                                                              enum-name—
|                                                              type-name—
|                                                              nested-name-specifier
|                                                 typename-
↪specifier———————————————————————————————————
|                                                 type-
↪specifier———————————————————————————————————
|                                                 defining-type-
↪specifier———————————————————————————————
|                                                 decl-
↪specifier———————————————————————————————————
|                                                 decl-specifier-
↪seq——————————————————————————————————————————————————————————————————————
|                           parameter-
↪declaration——————————————————————————————————————————————————————————————————
|                           template-
↪parameter————————————————————————————————————————————————————————————————————
| template-parameter-
↪list—————————————————————————————————————————————————————————————————————————

shift using rule namespace-alias ->   identifier

| template-parameter-list , attribute-specifier-seq? typename  identifier      ::   ␣
↪template? simple-template-id continue-decl-specifier-seq decl-specifier-seq abstract-
↪declarator?
|                                                              namespace-alias
|                                                              namespace-name—
|                                                              nested-name-specifier
|                                                 typename-
↪specifier———————————————————————————————————
|                                                 type-
↪specifier———————————————————————————————————
|                                                 defining-type-
↪specifier———————————————————————————————
|                                                 decl-
↪specifier———————————————————————————————————
|                                                 decl-specifier-
↪seq——————————————————————————————————————————————————————————————————————
|                           parameter-
↪declaration——————————————————————————————————————————————————————————————————
|                           template-
↪parameter————————————————————————————————————————————————————————————————————
| template-parameter-
↪list—————————————————————————————————————————————————————————————————————————

shift using rule namespace-name ->   identifier

| template-parameter-list , attribute-specifier-seq? typename  identifier     ::    ␣
```

(continues on next page)

(continued from previous page)

```
↪template? simple-template-id continue-decl-specifier-seq decl-specifier-seq abstract-
↪declarator?
│                                                          namespace-name
│                                                          nested-name-specifier
│                                                 typename-
↪specifier──────────────────────────────────────
│                                                 type-
↪specifier──────────────────────────────────────
│                                                 defining-type-
↪specifier──────────────────────────────────
│                                                 decl-
↪specifier──────────────────────────────────────
│                                                 decl-specifier-
↪seq──────────────────────────────────────────────────────────────────────────────────
│                           parameter-
↪declaration──────────────────────────────────────────────────────────────────────────
│                           template-
↪parameter────────────────────────────────────────────────────────────────────────────
│ template-parameter-
↪list─────────────────────────────────────────────────────────────────────────────────

shift using rule typedef-name ->   identifier

│ template-parameter-list , attribute-specifier-seq? typename  identifier   ::      ␣
↪template? simple-template-id continue-decl-specifier-seq decl-specifier-seq abstract-
↪declarator?
│                                                          typedef-name
│                                                          type-name──
│                                                          nested-name-specifier
│                                                 typename-
↪specifier──────────────────────────────────────
│                                                 type-
↪specifier──────────────────────────────────────
│                                                 defining-type-
↪specifier──────────────────────────────────
│                                                 decl-
↪specifier──────────────────────────────────────
│                                                 decl-specifier-
↪seq──────────────────────────────────────────────────────────────────────────────────
│                           parameter-
↪declaration──────────────────────────────────────────────────────────────────────────
│                           template-
↪parameter────────────────────────────────────────────────────────────────────────────
│ template-parameter-
↪list─────────────────────────────────────────────────────────────────────────────────

shift using rule class-name ->   identifier

│ template-parameter-list , attribute-specifier-seq? typename  identifier ::        ␣
↪template? simple-template-id continue-decl-specifier-seq decl-specifier-seq abstract-
↪declarator?
│                                                          class-name
```

(continues on next page)

(continued from previous page)

```
│                                                          type-name─
│                                                          nested-name-specifier
│                                                 typename-
↪specifier──────────────────────────────────────
│                                                 type-
↪specifier──────────────────────────────────────
│                                                 defining-type-
↪specifier──────────────────────────────────
│                                                 decl-
↪specifier──────────────────────────────────────
│                                                 decl-specifier-
↪seq─────────────────────────────────────────────────────────────────────────────
│                           parameter-
↪declaration─────────────────────────────────────────────────────────────────────
│                           template-
↪parameter───────────────────────────────────────────────────────────────────────
│ template-parameter-
↪list────────────────────────────────────────────────────────────────────────────

shift using rule template-name ->   identifier
   │ template-parameter-list , attribute-specifier-seq? typename  identifier    <␣
↪template-argument-list? > :: template? simple-template-id continue-decl-specifier-seq␣
↪decl-specifier-seq abstract-declarator?
│                                                          template-name
│                                                          simple-template-
↪id──────────────────────
│                                                          typedef-
↪name────────────────────────────
│                                                          type-
↪name──────────────────────────
│                                                          nested-name-
↪specifier──────────────────────
│                                                 typename-
↪specifier──────────────────────────────────────────────────────
│                                                 type-
↪specifier─────────────────────────────────────────────────────
│                                                 defining-type-
↪specifier──────────────────────────────────────
│                                                 decl-
↪specifier─────────────────────────────────────────────────────
│                                                 decl-specifier-
↪seq─────────────────────────────────────────────────────────────────────────────
│                           parameter-
↪declaration─────────────────────────────────────────────────────────────────────
│                           template-
↪parameter───────────────────────────────────────────────────────────────────────
│ template-parameter-
↪list────────────────────────────────────────────────────────────────────────────

reduce using rule type-parameter-key -> typename

│ template-parameter-list , attribute-specifier-seq? typename            identifier
```

(continues on next page)

(continued from previous page)

```
                                                    type-parameter-key identifier?
                          type-parameter──────────────────────────────────────────
                          template-parameter──────────────────────────────────────
 template-parameter-list──────────────────────────────────────────────────────────
```

initializer, parameters-and-qualifiers

A (token could introduce either a *parameters-and-qualifiers* clause, or an *initializer*. In some cases, the additional tokens will disambiguate the declaration. In other cases, the statement is truly ambiguous.

```
ptr-declarator -> noptr-declarator
parameters-and-qualifiers ->   ( parameter-declaration-clause ) cv-qualifier-seq? ref-
↪qualifier? exception-specification? attribute-specifier-seq?
```

```
reduce using rule ptr-declarator -> noptr-declarator

 attribute-specifier-seq decl-specifier-seq noptr-declarator  ( expression-list ) ;
                                           ptr-declarator── initializer──────
                                           declarator──────
                                           init-declarator──────────────────
                                           init-declarator-list─────────────
 simple-declaration──────────────────────────────────────────────────────────
 block-declaration───────────────────────────────────────────────────────────
 declaration─────────────────────────────────────────────────────────────────

shift using rule parameters-and-qualifiers ->   ( parameter-declaration-clause ) cv-
↪qualifier-seq ref-qualifier exception-specification attribute-specifier-seq

 attribute-specifier-seq decl-specifier-seq noptr-declarator  ( parameter-declaration-
↪clause ) cv-qualifier-seq? ref-qualifier? exception-specification attribute-specifier-
↪seq? parameters-and-qualifiers trailing-return-type function-body
                                                          parameters-and-
↪qualifiers──────────────────────────────────────────────────────────────────────────
                                           noptr-
↪declarator──────────────────────────────────────────────────────────────────────────
                                          ␣
↪declarator──────────────────────────────────────────────────────────────────────────
 function-
↪definition──────────────────────────────────────────────────────────────────────────
␣
↪declaration─────────────────────────────────────────────────────────────────────────
```

trailing-return-type, abstract-declarator / parameters-and-qualifiers / initializer

In the context of a trailing return type, the (token can mark either the beginning of a *parameter-and-qualifiers* clause, a *noptr-abstract-declarator* or an *initializer*. There is no way to know until more tokens have been parsed.

```
parameters-and-qualifiers ->   ( parameter-declaration-clause ) cv-qualifier-seq? ref-
↪qualifier? exception-specification? attribute-specifier-seq?
type-id -> type-specifier-seq
noptr-abstract-declarator ->   ( ptr-abstract-declarator )
```

```
shift using rule parameters-and-qualifiers ->   ( parameter-declaration-clause )

| decl-specifier-seq noptr-declarator parameters-and-qualifiers -> type-specifier-seq  (␣
↪parameter-declaration-clause ) cv-qualifier-seq? ref-qualifier? exception-
↪specification? attribute-specifier-seq? trailing-return-type ;
|                                                                                  ␣
↪parameters-and-
↪qualifiers──────────────────────────────────────────────────────────────────────────────
|                                                                                  ␣
↪abstract-
↪declarator──────────────────────────────────────────────────────────────────────────────
|                                                          type-
↪id──────────────────────────────────────────────────────────────────────────────
|                                                      trailing-return-
↪type──────────────────────────────────────────────────────────────────────────────
|                    ␣
↪declarator──────────────────────────────────────────────────────────────────────────────
|                    init-
↪declarator──────────────────────────────────────────────────────────────────────────────
|                    init-declarator-
↪list──────────────────────────────────────────────────────────────────────────────
| simple-
↪declaration──────────────────────────────────────────────────────────────────────────────
| block-
↪declaration──────────────────────────────────────────────────────────────────────────────
|␣
↪declaration──────────────────────────────────────────────────────────────────────────────

reduce using rule type-id -> type-specifier-seq

| decl-specifier-seq noptr-declarator parameters-and-qualifiers -> type-specifier-seq  (␣
↪expression-list ) ;
|                                                          type-id──────────␣
↪initializer──────
|                                                      trailing-return-type─
|                    declarator──────────────────────────────────────────────
|                    init-
↪declarator──────────────────────────────────────────────────────────
|                    init-declarator-
↪list──────────────────────────────────────────────
| simple-
↪declaration──────────────────────────────────────────────────────────────────────────────
| block-
```

(continues on next page)

(continued from previous page)

```
↪declaration──────────────────────────────────────────────────────────────────────────────────────────────────────────
│ ⌴
↪declaration──────────────────────────────────────────────────────────────────────────────────────────────────────────

shift using rule noptr-abstract-declarator ->   ( ptr-abstract-declarator )

│ decl-specifier-seq noptr-declarator parameters-and-qualifiers -> type-specifier-seq  (⌴
↪ptr-abstract-declarator ) parameters-and-qualifiers trailing-return-type ;
│                                                                                    ⌴
↪noptr-abstract-declarator──
│                                                                                    ⌴
↪abstract-declarator──────────────────────────────────────────────────────
│                                                                 type-
↪id──────────────────────────────────────────────────────────────────────────────────────────────────────────────────
│                                                              trailing-return-
↪type──────────────────────────────────────────────────────────────────────────────────────────────────────────────
│                    ⌴
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────
│                    init-
↪declarator──────────────────────────────────────────────────────────────────────────────────────────────────────────
│                    init-declarator-
↪list────────────────────────────────────────────────────────────────────────────────────────────────────────────────
│ simple-
↪declaration─────────────────────────────────────────────────────────────────────────────────────────────────────────
│ block-
↪declaration─────────────────────────────────────────────────────────────────────────────────────────────────────────
│ ⌴
↪declaration──────────────────────────────────────────────────────────────────────────────────────────────────────────
```

The parser maintains all possibilities in parallel and the semantic analyzer makes a pick among valid results in the next phase.

`type` template parameter, `non-type` class template parameter

When a *template-parameter* clause starts with the token `class`, it is not determined yet if the parameter will be a *type-parameter* or a *parameter-declaration* of an *elaborated-type-specifier*. Additional tokens need to be parsed to disambiguate.

```
class-key -> class
type-parameter-key -> class
```

```
reduce using rule class-key -> class

│ class                         identifier identifier
│ class-key
│ elaborated-type-specifier─────────────
│ type-specifier────────────────────────
│ defining-type-specifier───────────────
│ decl-specifier────────────────────────
│ decl-specifier-seq────────────────────
```

(continues on next page)

(continued from previous page)

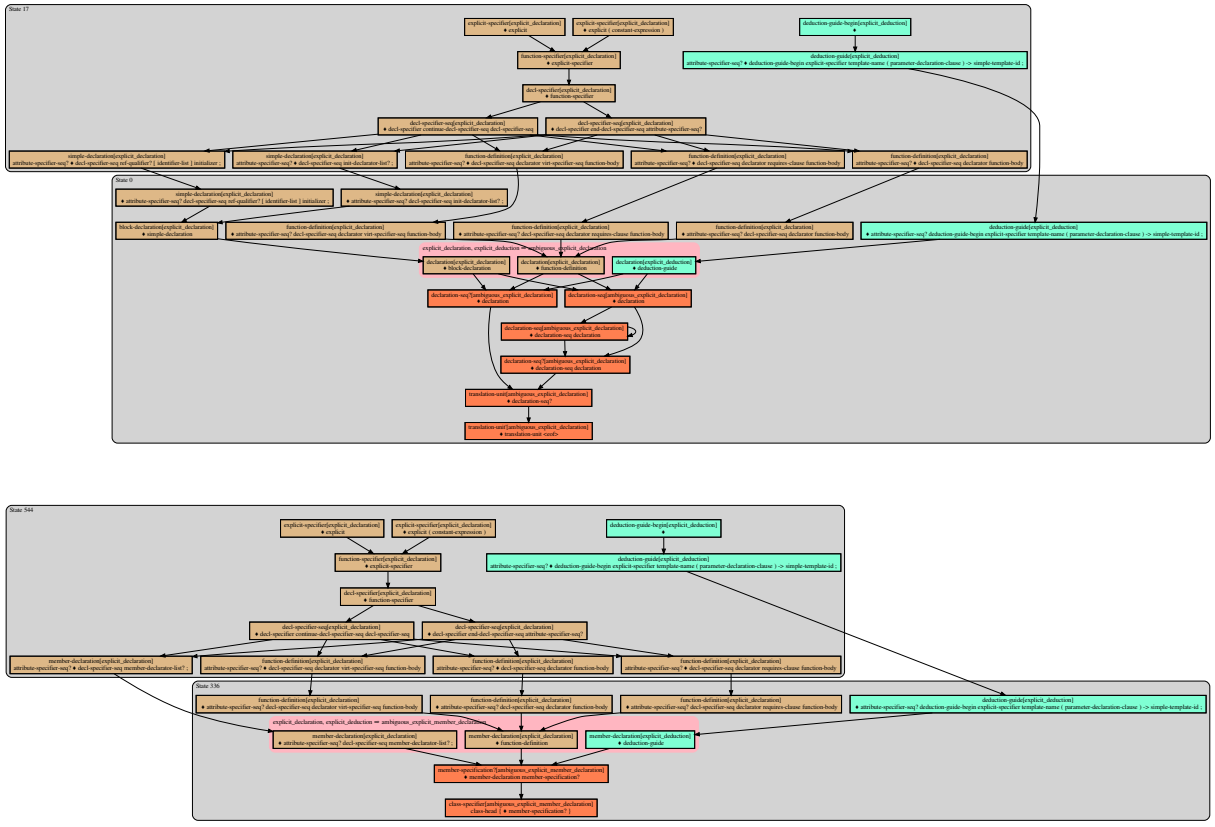
```
  parameter-declaration──────────────────────────
  template-parameter─────────────────────────────

reduce using rule type-parameter-key -> class

  class                identifier
  type-parameter-key
  type-parameter──────────────────
  template-parameter──────────────
```

deduction-guide, template-declaration

A user-defined deduction guide is defined with the same syntax as a function declaration with a trailing return type, except that it uses the name of a class template as the function name. The parser cannot disambiguate a *deduction-guide* from a *simple-declaration*, so it allows both branches to be parsed simultaneously and leaves the resolution to the semantic analyzer.

In order to aid the parser, a dummy empty reduction is added early in the rules to move the conflict early. The grammar is tagged on the dummy reduction instead.

The merge is done at the *declaration* rule or the *member-declaration* rule.



1.2.2 The OpenCL C++ toolchain

CHAPTER

TWO

API DOCUMENTATION

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search